# PRIVACY-PRESERVING QUERY PROCESSING ON TEXT DOCUMENTS

by

SAHIN BUYRUKBILEN

A dissertation submitted to the Graduate Faculty in Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy, The City University of New York

2013

This manuscript has been read and accepted by the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

| | |
|---|---|
| | (Spiridon Bakiras) |
| Date | Chair of Examining Committee |
| | (Theodore Brown) |
| Date | Executive Officer |
| | Bilal Khan |
| | Abdullah Uz Tansel |
| | William E. Skeith III. |
| | Gabriel Ghinita |
| | Supervisory Committee |

THE CITY UNIVERSITY OF NEW YORK

Abstract

**PRIVACY-PRESERVING QUERY PROCESSING ON TEXT DOCUMENTS**

by

SAHIN BUYRUKBILEN

Advisor: Spiridon Bakiras

Privacy-preserving query processing is an essential component for data processing, especially in outsourced databases, or in data operations which have special security and privacy requirements such as sharing of sensitive data. While cloud computing and data outsourcing attract an increasing number of customers, the security and privacy of sensitive data still remains an open problem. Encryption secures the data against unauthorized access, but it does not provide the ability to query the data unless the encryption scheme is searchable. Searchable encryption can be either private or public key depending on the needs of the user. In general, private-key solutions are faster but suffer from a key management problem. On the other hand, public-key solutions provide more flexibility but their running times are much higher than private-key protocols. Furthermore, parties may sometimes be forced to share data in order to comply with regulations or agreements. For example, different health care companies or intelligence agencies may need to find whether they have similar records in their databases without compromising privacy. Consequently, privacy-preserving similarity search between text documents is an emerging field as sensitive data sharing becomes inevitable. In this dissertation we present two privacy-preserving text processing protocols: (i) a ranked keyword search mechanism over outsourced public-key

encrypted data and (ii) a similar document detection system. We introduce efficient algorithms for answering these query types and illustrate their feasibility in real-life applications.

# ACKNOWLEDGMENTS

Many people helped me to accomplish this work. They deserve my appreciation and deep thanks. First of all, I would like to express my gratitude to my supervisor, Prof. Spiridon Bakiras who was abundantly helpful and offered invaluable assistance, support and guidance. Deepest gratitude are also due to the members of the supervisory committee, Prof. Bilal Khan, Prof. Abdullah Uz Tansel, Prof. William E. Skeith III., and Prof. Gabriel Ghinita without whose knowledge and assistance this study would not have been successful. I would also like to thank the Executive Director Prof. Ted Brown for his support and assistance.

Special thanks also to all my graduate friends, especially members of Turkish National Police (TNP); Dr. Omer Demir, Dr. Zeki Bilgin for their company and invaluable assistance.

I would like to convey thanks to Turkish National Police for providing the financial support and opportunity of having a graduate education abroad. I would also like to thank The Graduate Center, and John Jay College of Criminal Justice for providing facilities during my education.

Last but not the least, I would like to express my love and gratitude to my beloved family; for their understanding and endless love, through the duration of my studies.

***To my beloved wife and sons, and in memory of my dear mother.***

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

The rapid evolution of network and data technologies changed the way we store, process or share our data. We spend more time interacting with the online world. In 2013 the volume of e-Commerce transactions is expected to be about $963 Billion [44], and the anticipated revenue for cloud computing in 2013 is $16.7 Billion [7]. Obviously, online privacy is a high priority area that cannot be overlooked. Sensitive data has to be encrypted and privacy-preserving protocols have to be deployed in order to keep data secure and private while utilizing open resources of the Internet.

In this dissertation, we address privacy-preserving query processing on text documents. The objective is to run certain query types on text documents while keeping their contents secret. In particular, we consider two application scenarios where data privacy is essential. In the first case, the documents are outsourced to a cloud provider and must, thus, be encrypted. Furthermore, the client is interested in performing keyword searches on these documents, which necessitates that the cloud provider is able to process such queries directly on the encrypted data. In the second case, there are two parties that maintain their own private datasets that they do not want share with each other. Nevertheless, these parties are interested in discovering the existence of similar documents within their respective datasets. In this scenario, the documents are stored in plaintext format, but privacy-preserving protocols are essential in order

to identify the similar documents without sharing the actual data.

## 1.1 Searchable Encryption

Cloud computing is the new trend in IT that offers users great flexibility in purchasing off-site, third-party resources (ranging from software to infrastructure) at competitive prices. Popular cloud computing services, such as Amazon's Elastic Compute Cloud (EC2)[1], provide on-demand computing, network, and storage resources in an attractive "pay-as-you-go" pricing model [3]. The flexibility of provisioning services on-demand and the virtually endless amount of resources, enable entrepreneurs to deploy their business instantly, without purchasing expensive hardware/software or hiring technically skilled system administrators [39]. Similarly, cloud storage has become ubiquitous and numerous providers, such as Google Drive, SkyDrive, and iCloud, offer multi-GB storage space at no cost. Consequently, users are motivated to move their personal data to the cloud, which gives them the ability to access them anytime from anywhere.

Despite the aforementioned advantages, most cloud computing platforms do not provide adequate security and privacy for the outsourced data. As a result, owners of sensitive information such as emails, personal health records, financial transactions, etc., may be skeptical in purchasing such services, given the risks associated with the unauthorized access to their data. To mitigate these security risks, sensitive information should be encrypted prior to being transferred to the cloud provider. Nevertheless, outsourcing encrypted data would not be practical if the service provider is

1. `http://aws.amazon.com/ec2/`

unable to process queries, as this would necessitate the transfer and decryption of all records at the client site. Therefore, designing privacy-preserving query mechanisms for encrypted data is of paramount importance.

To this end, *searchable encryption* is a family of cryptographic protocols that facilitate private keyword searches directly on encrypted data. These protocols allow users to upload encrypted versions of their documents to the cloud, while retaining the ability to query the database with traditional plaintext keyword queries. Fully functional searchable encryption can be achieved with oblivious RAMs [37], since they can simulate any data structure in a private manner. Even though oblivious RAMs can hide everything from the server (including the access pattern), they incur a very high computational cost. Therefore, the majority of the research work on searchable encryption has focused on efficiency improvements, by weakening the underlying security definitions. In particular, most searchable encryption schemes aim at hiding all but the *access* and *search* patterns. Access pattern is defined as the documents retrieved during a search, and search pattern refers to the possibility of inferring whether two queries were performed for the same keyword.

In the symmetric key setting, Searchable Symmetric Encryption (SSE) [21, 64] is a well-studied problem and there exist numerous techniques that support different types of keyword searches. The work by Cao et al. [15] offers the most practical SSE scheme to date, as it implements *ranked* keyword searches. Specifically, the client can issue an arbitrary multi-keyword query (similar to web search engines) and the server will return the top-$k$ most relevant documents in the database. The main advantage of symmetric encryption is its computational efficiency that allows the server to perform linear searches on the encrypted documents at low cost. Nevertheless, symmetric

encryption has an inherent key management problem, so all SSE methods assume that the data owner is the only entity that may upload encrypted data/indexes to the server.

On the other hand, public-key cryptography allows any user to update an encrypted database (i.e., add a new item) without knowledge of the private key. A real-world scenario that leverages this functionality is given by Boneh et al. in [11]. Alice uses an email gateway for her communications and, as she considers her emails sensitive, she asks her friends to send their emails encrypted with her public key. A searchable encryption scheme would then enable Alice to retrieve all emails containing a specific keyword (e.g., "from:Bob"). Public-key Encryption with Keyword Search (PEKS) [11] is the most representative solution in this field that works by scanning all keywords from every document in the database. However, PEKS and all of its variants [4, 38] are very restrictive in the types of queries that they allow and, most importantly, none of them implements ranked keyword search.

## Our contribution

In this dissertation, we introduce the *first* method that provides ranked results from multi-keyword searches on public-key encrypted data. Since public-key cryptography is computationally expensive, we incorporate the following two design principles in our algorithms: (i) avoid a linear scan of the documents and (ii) parallelize the computations as much as possible. The first principle clearly necessitates the use of an indexing structure. To this end, we encrypt the keyword information for each document in a Bloom filter [8], and hierarchically aggregate (using homomorphic encryption) the individual indexes into a tree structure. Query processing is performed

at the client side, and entails the traversing of the tree in a *best-first* manner. To hide the content of the query from the server, the client utilizes an efficient private information retrieval (PIR) protocol [32] to extract the necessary Bloom filter entries from the tree nodes.

To speed-up the search process, we leverage the abundance of computational resources that are available in most cloud computing platforms. In particular, we split the indexing structure into multiple chunks, and utilize several CPUs in parallel in order to execute the PIR queries efficiently. Using measurements from Amazon's Elastic Compute Cloud, we show that our method provides reasonable response times with low communication cost.

## 1.2 Similar Document Detection

Similar document detection is an important problem in computing, and has attracted a lot of research interest since its introduction by Manber [51]. Specifically, with digital data production growing exponentially, efficient file system management has become crucial. Detecting similar files facilitates better indexing, and provides efficient access to the file system. Furthermore, it protects against security breaches by identifying file versions that are changed by a virus or a hacker. Similarly, web search engines periodically crawl the entire web to collect individual pages for indexing [52]. When a web page is already present in the index, its newer version may differ only in terms of a dynamic advertisement or a visitor counter and may, thus, be ignored. Therefore, detecting similar pages is of paramount importance for designing efficient web crawlers. Finally, plagiarism detection and copyright protection are two other

major applications that are built upon similar document detection.

While plaintext similar document detection is extremely important, it is not sufficient for secure and private operations over sensitive data. In many cases, owners of sensitive data may be forced to share their datasets with the government or other entities, in order to comply with existing regulations. For example, health care companies may be asked to provide data to monitor certain diseases reported in their databases. This may be accomplished by identifying similar attribute patterns in patient diagnosis information from different entities. Obviously, such pattern searches can not be performed without secure protocols, since they may lead to severe privacy violations for the individuals included in the various databases.

Data sharing for intelligence operations also involves risks when disclosing classified information to other parties. A person of interest may have records at several intelligence agencies under different names with similar attributes. To identify similar records, the participating agencies may only wish to disclose the existence of records akin to the query. Detecting violations of the academic double submission policy is another problem with similar restrictions. For example, a conference's organization committee may want to know whether the articles submitted to their conference are concurrently submitted to other publication venues. Since research articles are considered confidential until published, their contents cannot be revealed unless a similar article is found in another venue.

Secure similar document detection (SSDD) leverages secure two-party computation protocols, in order to solve the above problems that arise due to the distributed ownership of the data. In particular, SSDD involves two parties, each holding their own private dataset. Neither party wants to share their data in plaintext format, but

they both agree to identify any similar documents within their respective databases. The objective is to compute the similarity scores between every pair of documents without revealing any additional information about their contents. In existing work, document similarity is computed with either the inner product of public key encrypted vectors [45, 54, 46] or with secure set intersection cardinality methods based on $N$-grams [9]. However, the computational cost of inner product based similarity is very high, due to numerous public key operations. On the other hand, $N$-gram based methods are more computationally efficient, but they incur a high communication cost as the number of documents increases.

## Our contribution

In this dissertation, we present a novel method based on *simhash* document fingerprints[2]. Simhash is essentially a dimensionality reduction technique that encodes all the document terms and their frequencies into a fixed-size bit vector (typically 64 bits). Unlike classical hashing algorithms that produce uniformly random digests, the simhash digests of two similar documents will only differ in a few bit positions [43]. This enables us to (i) evaluate the similarity over a fairly small data structure rather than large vectors, and (ii) reduce the similarity calculation to a secure XOR computation between two bit vectors. To further improve the privacy preserving properties of our approach, we modify the basic method to hide the similarity scores of the compared documents. In particular, the enhanced version of our protocol returns all the document pairs whose similarity is above a user-defined threshold, while maintaining the exact scores secret. This is the *first* protocol in the literature that

2. We follow the simhash definition of Charikar [17].

provides this functionality. Our experimental results demonstrate that the proposed methods improve the computational and communication costs by at least one order of magnitude compared to the current state-of-the-art protocol. Moreover, they achieve a high level of precision and recall.

## 1.3 Organization

The remainder of this dissertation is organized as follows. Chapter 2 describes the tools we use in our research, Chapter 3 presents our work on privacy-preserving ranked keyword search, Chapter 4 introduces privacy-preserving similar document detection in detail, and finally Chapter 5 provides conclusions and our plans for future work.

# CHAPTER 2

# PRELIMINARIES

In this chapter we give a brief description of the primitives incorporated in our methods. Section 2.1 discusses homomorphic encryption, Section 2.2 describes secure two-party computation, Section 2.3 provides information about document simhashes and Section 2.4 introduces private information retrieval.

## 2.1 Homomorphic Encryption

Homomorphic encryption allows certain algebraic operations on two plaintexts to be carried out on their corresponding ciphertexts, without any intermediate decryptions. In particular, *fully* homomorphic encryption [31] enables both addition and multiplication operations on the ciphertext space and, therefore, such cryptosystems could be used to build searchable encryption schemes with perfect privacy. However, research on fully homomorphic encryption is still in its infancy and all existing methods are extremely expensive. Because of this high computational cost, current proposed methods mostly provide partially homomorphic encryption, i.e. *additive* or *multiplicative*, which incur more reasonable computational costs.

### 2.1.1 Paillier

In our first proposed work, we utilize Paillier's cryptosystem [56], which is an efficient *additively* homomorphic encryption scheme. Specifically, given the Paillier encryptions $E(m_1)$ and $E(m_2)$ of two plaintext messages $m_1$ and $m_2$, we can compute the encryption of $E(m_1 + m_2)$ by multiplying the two ciphertexts:

$$E(m_1 + m_2) = E(m_1)E(m_2)$$

Furthermore, any message $m$ can be multiplied with a plaintext constant $c$ as follows:

$$E(cm) = E(m)^c$$

The Paillier cryptosystem is semantically secure, i.e., it is infeasible to derive any information about a plaintext, given its ciphertext and the public key that was used to encrypt it. Its security is based on the decisional composite residuosity assumption. The cryptosystem works as follows.

**Key generation.** Choose two large primes $p$ and $q$ of equal length, and compute the RSA modulus $n = pq$. For security, each prime should be at least 512 bits in length. The public key is $n$ and the private key is $\varphi(n) = (p-1)(q-1)$.

**Encryption.** To encrypt a message $m \in \mathbb{Z}_n$, choose a uniformly random integer $r \in \mathbb{Z}_n^*$, and compute the ciphertext $c \in \mathbb{Z}_{n^2}^*$ as $c = (n+1)^m r^n \bmod n^2 = (mn + 1)r^n \bmod n^2$.

**Decryption.** Given a ciphertext $c$, compute the plaintext

$$m = \frac{(c^{\varphi(n)} \bmod n^2) - 1}{n} \cdot \varphi(n)^{-1} \bmod n$$

where $\varphi(n)^{-1}$ is the multiplicative inverse of $\varphi(n) \bmod n$.

### 2.1.2 ElGamal

In our second work, we utilize ElGamal's *additively* homomorphic encryption scheme [23, 19]. The scheme incorporates key generation, encryption, and decryption algorithms as given below.

**Key generation.** Instantiate a cyclic group $G$ of prime order $p$, with generator $g$ ($G$, $g$, and $p$ are public knowledge). Then choose a *private* key $x$, uniformly at random from $\mathbb{Z}_p^*$. Publish the *public* key $h = g^x$.

**Encryption.** Let $m$ be the private message. Choose $r$, uniformly at random from $\mathbb{Z}_p^*$, and compute ciphertext $(c_1, c_2) = (g^r, h^{r+m})$.

**Decryption.** Compute $h^m = c_2 \cdot (c_1^x)^{-1}$ and solve the discrete logarithm to retrieve $m$.

ElGamal's scheme is also *semantically* secure and it's security is based on the decisional Diffie-Hellman assumption. Note that the decryption process involves a discrete logarithm computation. If the encrypted values are not too large (which is the case in our protocol) it is possible to precompute all possible results and use them as a

lookup table to speed up the decryption process.

## 2.2 Secure two-party computation

A secure two-party computation protocol [49] allows two parties, Alice and Bob, to jointly compute a function based on their inputs, while maintaining their inputs secret (i.e., they only learn the function output). Yao's *garbled circuit* technique [69] is a generic two-party computation protocol that can evaluate securely any function, given its Boolean circuit representation. Nevertheless, Yao's technique is efficient only for relatively simple functions, i.e., when the number of input wires and logic gates is small. In particular, every input wire (for one of the parties) necessitates the execution of an Oblivious Transfer (OT) [55] protocol that is computationally expensive, while the total number of gates affects the overall communication and circuit construction/evaluation costs.

Besides Yao's generic protocol, researchers have also devised application dependent protocols that typically leverage the properties of additively homomorphic encryption. As an example, consider the *secure inner product* computation that is used extensively in previous work [54, 46]. For simplicity, assume that Alice holds vector $\langle a_1, a_2 \rangle$ and Bob holds vector $\langle b_1, b_2 \rangle$. The objective is for Alice to securely compute $S = a_1 b_1 + a_2 b_2$. Initially, Alice encrypts her input with her public key and sends $E(a_1), E(a_2)$ to Bob. Next, Bob utilizes the properties of homomorphic encryption to produce $E(S) = E(a_1)^{b_1} E(a_2)^{b_2}$. Finally, Alice decrypts the result and learns the value of $S$.

## 2.3 Simhash

Simhash maps a high dimensional feature vector into a fixed-size bit string [17]. However, unlike classical hashing algorithms, simhash produces fingerprints that have a large number of matching bits when the underlying documents are similar. Computing the simhash fingerprint from a text document is a fairly simple process. First, one has to extract all the document terms along with their weights (e.g., how many times they appear in the document). Then, a vector of $l$ counters $\langle c_0, c_1, \ldots, c_{l-1} \rangle$ is initialized, where $l$ is the size of the simhash fingerprint (e.g., 64 bits). Each of the document's terms is then hashed with a standard hashing algorithm, such as SHA1. If the bit at position $i$ ($i \in \{0, 1, \ldots, l-1\}$) in the resulting SHA1 digest is 0, $c_i$ is decremented by the weight of that term; otherwise, $c_i$ is incremented by the same weight. When all document terms are processed, the simhash fingerprint is constructed as follows: for all $i \in \{0, 1, \ldots, l-1\}$, if $c_i > 0$, set the corresponding bit to 1; otherwise, set the bit to 0.

## 2.4 Private Information Retrieval

Private information retrieval (PIR) was first introduced by Chor et al. [18], and is formally defined as follows: The server holds a database with $N$ records and the client wants to retrieve the $i$-th record, without the server knowing the value of index $i$. *Information theoretic* PIR [18, 35, 68] is secure against computationally unbounded adversaries, but it is not practical as it requires that the database be replicated into multiple *non-colluding* servers. On the other hand, *computational* PIR protocols [32, 48, 50] work with a single server, and employ well known cryptographic primitives

that are secure against computationally bounded adversaries.

In our methods, we leverage the computational PIR protocol of Gentry and Ramzan [32], because (i) it has very low communication cost, and (ii) it allows the retrieval of multiple records with a single query. The security of the protocol is based on the $\varphi$-hiding assumption, and its operation can be summarized as follows.

**Setup.** During a setup phase, the server associates each record $j$ with a prime power $\pi_j = p_j^{c_j}$, where $p_j$ is a small prime. Assuming that each record is $\ell$ bits in size, $c_j$ is the smallest integer such that $\log \pi_j > \ell$. All the above values are public knowledge. Before participating in query processing, the server computes a value $\beta$, which is the unique solution to the congruences $\beta \equiv D_j \pmod{\pi_j}$, for all $j \in \{1, 2, \ldots, N\}$, where $D_j$ is the binary representation of record $j$. This is a straightforward application of the Chinese Remainder Theorem (CRT). Note that all client queries are processed on the transformed database $\beta$.

**Query generation.** As noted by Groth et al. [40], Gentry and Ramzan's scheme can be used to retrieve multiple records with a single query. Let $i_1, i_2, \ldots, i_k$ be the indexes of the records that the client wants to retrieve. Initially, the client computes $\pi = \prod_{j=1}^{k} \pi_{i_j}$. He then chooses two large primes $p$ and $q$, such that $p = 2\pi r + 1$ and $q = 2st + 1$, where $r$, $s$, and $t$ are large random integers. After setting $m = pq$, the client selects a random element $g \in \mathbb{Z}_m^*$ with order $\pi v$, where $\gcd(\pi, v) = 1$. Finally, he sends $(g, m)$ to the server. For security, we want $m$ to be at least 1024 bits in size, and $\log m > 4 \log \pi$.

**Query processing.** The server simply computes $c = g^{\beta} \bmod m$ and returns the

result to the client. Note that $\beta$ is at least equal to the size of the original database, so the computational complexity at the server is linear in $N$.

**Result extraction.** To reconstruct the $k$ records, the client computes, for each $i_j$, $c_{i_j} = c^{\pi v/\pi_{i_j}} \bmod m$. This value should be equal to $g_{i_j} = (g^{\pi v/\pi_{i_j}})^{D_{i_j}} \bmod m$, and thus, the client can retrieve record $D_{i_j}$ using the Pohlig-Hellman algorithm for discrete logarithms [58].

# CHAPTER 3

# PRIVACY-PRESERVING RANKED KEYWORD SEARCH OVER PUBLIC-KEY ENCRYPTED DATA

## 3.1 Background

The work of Song et al. [64] is the first Searchable Symmetric Encryption (SSE) scheme proposed in the literature. Their method does not employ indexes, but instead encrypts each document in a way that allows keyword search. Searching takes linear time in the combined size of all documents, as the server has to test the trapdoor against every encrypted block in a document. Although this model is proven to be a secure encryption scheme, it is not a secure searchable scheme, because of a vulnerability against statistical attacks. In particular, every keyword search reveals the exact position(s) in the document where there is a positive match.

Goh [33] introduces secure indexes that are based on Bloom filters and pseudo-random functions as hash functions. Specifically, each document has its own index that is constructed as follows. Every word in the document is first hashed with the master key and the output is hashed again with the document *id*, in order to differentiate a word's hash values in different files. The produced output is called a *codeword* and is inserted into the Bloom filter. Finally, the Bloom filter is blinded by inserting a random uniform distribution of 1's. To determine whether a certain keyword exists

in a file, the querier computes and sends the corresponding codeword to the server which, in turn, checks whether the bits associated with the codeword are all set (with a probability of false positives that is inherent in Bloom filters).

Chang and Mitzenmacher [16], on the other hand, use a predefined dictionary of all possible words in the database and, for each document, they build a binary array (the index) that identifies the keywords appearing in the document. Next, the user masks the bits in the index with the output of a pseudo-random function. To search for a keyword, the user sends short seeds for the pseudo-random functions to the server, so as to help recover the necessary parts of the index.

Curtmola et al. [21] maintain, for each keyword, an inverted index (stored as a linked list) comprising of document identifiers. Every node in the list stores information about the position and the decryption key of the next node. Then, the nodes from all inverted indexes are encrypted with random keys and are randomly inserted into an array. With this construction, given the position and decryption key of the first node of an inverted index, it is possible to find all documents which include the corresponding keyword. The recent study of Kamara et al. [47], uses inverted indexes to construct a dynamic encryption scheme, which allows document insertions and deletions.

To support ranked keyword searches, Wang et al. [66] build an inverted index for every keyword in the dataset. For security, the actual scores are encrypted with a modified Order-Preserving Symmetric Encryption (OPSE) scheme [10], where the numeric ordering of the plaintexts is preserved in the ciphertexts. During query processing, the user sends a trapdoor that allows the server to decrypt the corresponding entry in the inverted index. Then, the server compares the encrypted scores and returns

to the client the top-$k$ results. Cao et al. [15] address multi-keyword ranked queries by leveraging inner product similarity as the scoring function. Similar to the work of Chang and Mitzenmacher [16], they use a predefined dictionary of all possible words in the database, and construct an encrypted binary array for each document. To compute the similarity scores from the encrypted indexes, they employ the secure $k$NN computation method of [67].

In the public-key setting, Boneh et al. [11] first introduced a solution called Public Key Encryption with Keyword Search (PEKS). In their approach, the sender selects a set of keywords related to the message and, for each keyword, he produces its corresponding PEKS encryption. The encrypted message and keywords are subsequently sent to the server for storage. When the owner wants to search for messages containing a specific keyword, he creates a trapdoor for that keyword and sends it to the server. The server tests the trapdoor with each PEKS encrypted keyword and, if the test returns true, the message is returned to the owner.

Baek et al. [4] address several limitations of the PEKS framework by (i) preventing the server from reusing the trapdoors, (ii) eliminating the need for a secure authenticated channel between the owner and the server, and (iii) adding multi-keyword search capabilities. Although this work allows queries with multiple keywords, it does not provide ranked results. Similarly, further studies to improve PEKS [20, 30, 41] do not include multi-keyword search or ranked result features. Golle et al. [38] address conjunctive keyword searches on public-key encrypted data. However, as noted by the authors, their solution reveals the keyword fields that are searched by the client. Moreover, it does not support ranked results.

Finally, Boneh et al. [13] propose a searchable encryption scheme that provides perfect

privacy. They use encrypted Bloom filters to store keyword membership information, and leverage the homomorphic encryption scheme of Boneh, Goh, and Nissim [12] to allow the senders to modify the index in a secure manner. Nevertheless, their solution is computationally expensive, as it hides everything from the server (including the access pattern).

## 3.2 Threat Model and Security

We assume that the adversary is the cloud provider (i.e., the server[1]) and its goal is to derive any non-trivial information regarding (i) the plaintext keywords included in an encrypted document and (ii) the plaintext keywords from a client query. We also assume that the adversary runs in polynomial time and is "curious but not malicious," i.e., it will follow the protocol correctly, but will try to gain any advantage by analyzing the information exchanged during the protocol execution.

Similar to most searchable encryption schemes in the literature, we want to hide everything but the access and search pattern. In other words, the server will see the documents that are retrieved during a search, and also the index nodes that were accessed as part of that search. However, the following information will not be leaked:

- The keywords associated with an encrypted document (including their number).
- The keywords contained in a query (including their number).
- The ranking scores of the result set (including their actual order).

1. Henceforth, we use terms *server* and *cloud provider* interchangeably.

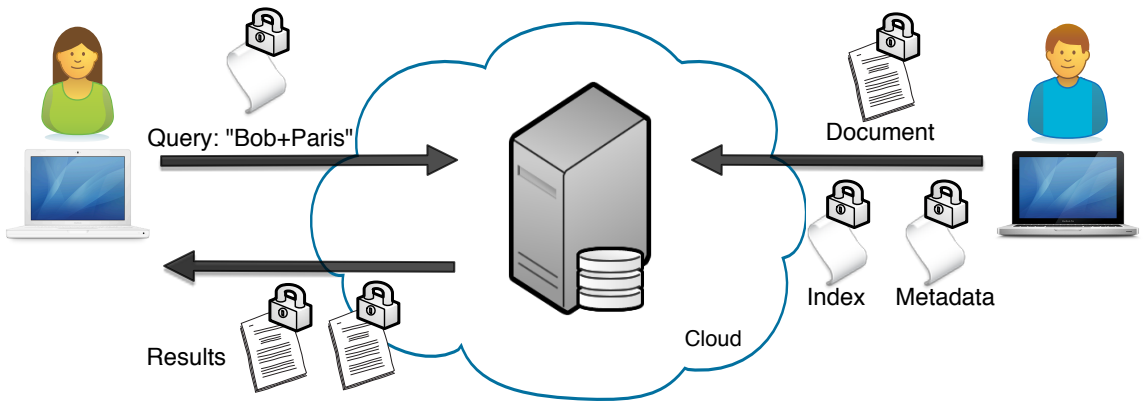



Figure 3.1: System architecture

## 3.3 Ranked Keyword Search

In this section we present in detail our privacy-preserving search mechanism. Section 3.4 describes the system architecture, while Section 3.5 presents our indexing scheme and discusses the handling of data updates. Section 3.6 introduces our top-$k$ query processing algorithm.

## 3.4 System Architecture

Alice is subscribed to a cloud computing service that allows her friends to send her documents, such as emails, in encrypted form, in order to enforce data confidentiality. When Bob wants to send Alice a new document (Figure 3.1), he first creates a set of keywords that are stored in a metadata file. For example, if the document is an email, the keywords could be the sender's name, the keywords appearing in the subject line, the most frequent keywords from the email's body, etc. Every keyword is associated with an integer value (score) that indicates the importance of that keyword in the

document. The metadata file may also include some additional information about the document, such as a file name, format, size, etc. Bob also creates an appropriate index structure that encodes membership and score information for all the keywords. Finally, Bob uses Alice's public key to encrypt each file separately (document, index, metadata) and transmits all the encrypted files to the cloud provider[2]. If Alice wants to view the emails from Bob's trip to Paris, she can create a query such as "Bob Paris" in order to retrieve the top-$k$ most relevant documents from the database. In particular, the query will trigger a two-party protocol between Alice and the cloud provider, which will allow Alice to download the corresponding encrypted documents.

Table 3.1: Summary of symbols

| **Symbol** | **Description** |
|---|---|
| $N$ | Database size (number of documents) |
| $M$ | Size of Bloom filter vector |
| $b$ | Block size of Bloom filter vector |
| $d$ | Bit size of Bloom filter counter |
| $f$ | Node fan-out |
| $e_i$ | Node $i$ |
| $D_i$ | Document $i$ |
| $W_i$ | Keyword set for document $i$ |
| $\|W\|$ | Max number of keywords per document |
| $\|K\|$ | Number of keywords in database |
| $n$ | RSA modulus for Paillier encryption |
| $m$ | RSA modulus for PIR scheme |

The document index consists of a *counting* Bloom filter [29] with a single hash function $H$ (which is public knowledge). The counting Bloom filter is essentially a vector of counters (where each counter is initialized to zero) that is used to probabilistically encode set membership information. As shown in Figure 3.2, for every keyword $w \in W_i$ the sender adds $w$'s score to the counter at position $H(w)$. (Table 3.1

2. For better performance, the document and metadata files can be encrypted with a hybrid cipher, as in the PGP protocol.

summarizes the most frequently used symbols in the scheme.) Note that, in this work, we assume that every document $D_i$ is allowed to define at most $|W|$ number of keywords. To reduce the number of ciphertexts required to store the Bloom filter, the sender creates groups of $b$ counters that are encrypted together as a single binary value (using the additively homomorphic Paillier cryptosystem). In the example of Figure 3.2, the number of counters in the index (Bloom filter) is $M = 9$ and the number of counters in a group to be encrypted together is $b = 3$, so the index is stored in $M/b = 3$ Paillier ciphertexts.

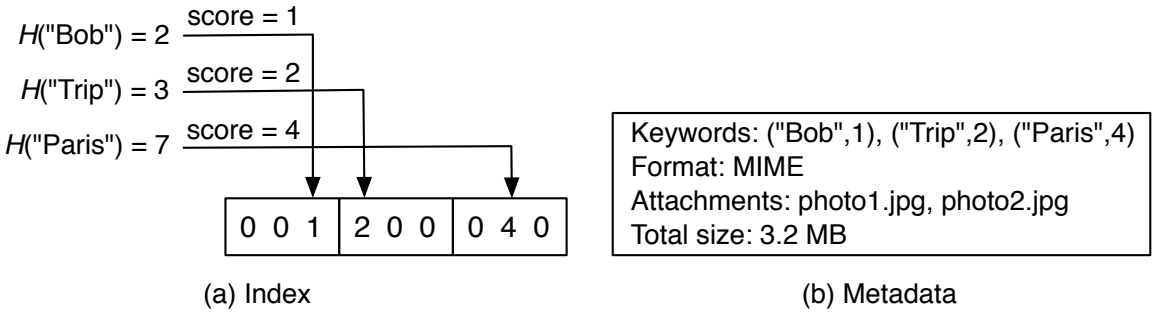



Figure 3.2: Document index and metadata

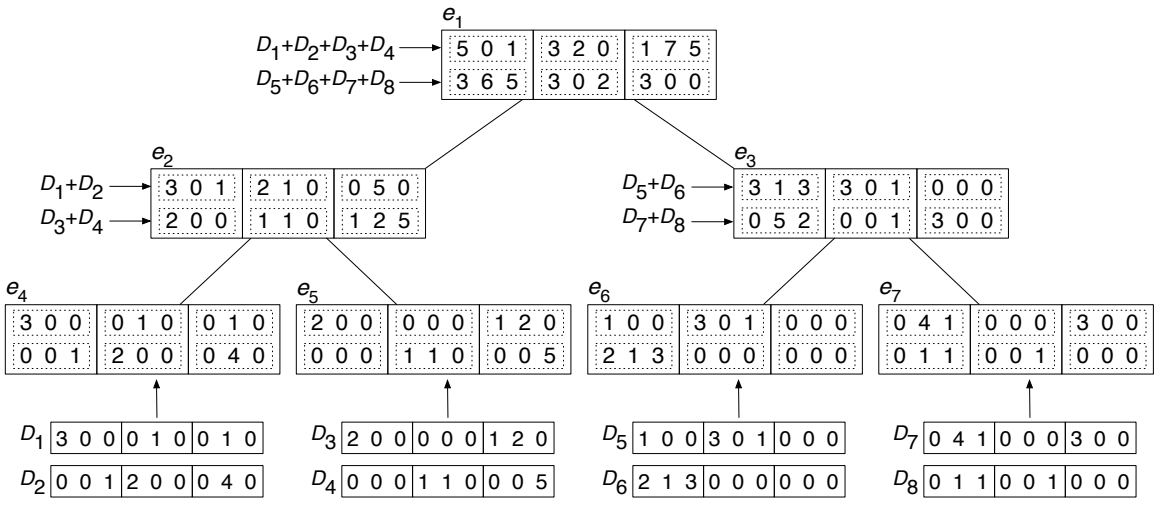



Figure 3.3: Database index

The reason for using both an index and a metadata file is twofold. First, documents can be quite large (e.g., emails with multimedia attachments) and the client may want to see a brief description of the document before downloading it locally. Second, Bloom filters are probabilistic structures and, thus, introduce several false positives in the result set. For instance, any document with a keyword that hashes to position 7 (Figure 3.2) is a potential match for query "Paris." On the other hand, if we have the client download the (compact) metadata files first, we can guarantee the correctness of the top-$k$ result set.

## 3.5 Index Construction and Update

### 3.5.1 Index structure

The cloud provider maintains a single index for all the client's documents. It is constructed by hierarchically aggregating the encrypted Bloom filters into a tree structure, as illustrated in Figure 3.3. Specifically, every *internal* node in the tree stores $f$ Bloom filters (where $f$ is the node fan-out), each being the aggregation of all document indexes in the subtree of the corresponding child. On the other hand, *leaf* nodes store the individual indexes from $f$ distinct documents.

To understand why each node can store multiple Bloom filters, observe that the nodes in Figure 3.3 are identical to document indexes, i.e., they consist of exactly $M/b$ Paillier encryptions. However, the Paillier cryptosystem allows for the encryption of very large integers that are similar in size to the RSA modulus $n$ (typically a 1024-bit number). Consequently, unlike individual document indexes, each Paillier

encryption in the tree structure will utilize most of the available storage space, in order to store $f$ groups of counters. To compute an appropriate value for $f$, given the group size $b$, we need to derive a lower bound for the counter size (in bits), such that the overflow probability is negligible. Assuming a database with $|K|$ distinct keywords, the probability that at least $x$ different keywords hash in the same counter is [29]:

$$\Pr(count \geq x) \leq M \left( \frac{e|K|}{xM} \right)^x$$

As an example, if $|K| = 20000$ and $M = 5000$, the probability that more than 30 keywords hash in the same location is less than $3 \times 10^{-10}$. If we can also estimate the distribution of the keywords in the document collection, we can determine a suitable bit size $d$ for the individual counters.

Nevertheless, even if we end up underestimating the value of $d$, there are several ways for the client to detect the resulting overflows. For instance, since the counters are aggregated in a bottom-up fashion, the client can recognize such overflows during query processing (Section 3.6). Alternatively, the client can periodically download the root node locally, and identify counters that experience large drops in their values. If such overflows are detected, the client can reconstruct the index tree from scratch, using the existing document metadata. Note that, in our experiments, we used $d = 15$ bits per counter and never experienced any overflows at the root node. Based on this value, and for a 1024-bit modulus $n$ and groups of size $b = 4$, a single Paillier encryption can store $f = 17$ groups of counters.

### 3.5.2 Index construction

Prior to indexing any documents, the cloud provider creates an empty version of the tree structure, i.e., every node on the tree will comprise of Paillier encryptions of 0. Specifically, we assume there is a limit (set by the cloud provider) on the number of encrypted documents $N$ that the client can store, which allows the server to initialize all nodes according to the predetermined fan-out $f$. However, since all nodes are initially identical, the server can simply construct one version and then create multiple copies, as needed. This is not a security concern, as the encryptions are re-randomized during every aggregation operation.

### 3.5.3 Index update

When the cloud provider receives a new document index from the sender (i.e., Bob), it selects an empty location at the leaf level of the database index where it can be stored (this is trivially done with a bitmap of size $N$). Next, it determines the nodes at all levels of the tree that need to incorporate the new document information. In the example of Figure 3.3, the insertion of document $D_6$ will affect nodes $e_6$, $e_3$, and $e_1$. Since each Paillier ciphertext in a node consists of $f$ groups of counters, the document index counters must be shifted accordingly so that they are added to the correct position at the different nodes. Specifically, each of the $M/b$ plaintext values of the document index must be shifted (to the left) $j \times d \times b$ times, where $0 \leq j \leq f-1$. Assuming counters of size $d = 4$ bits (Figure 3.3), the index values of $D_6$ must be shifted 12 times for nodes $e_6$ and $e_1$, and 0 times for node $e_3$. In the ciphertext space, the shifting operation of the underlying plaintext values is performed with a single

modular exponentiation, where the exponent is the value $2^{j \times d \times b}$.

To summarize, given a ciphertext $c_1$ from the document index and the corresponding ciphertext $c_2$ from an arbitrary tree node, the updated ciphertext $c_2$, following the insertion of the new document, is computed as:

$$c_2 = c_1^{2^{j \times d \times b}} c_2$$

where $j$ is calculated by the cloud provider, based on the underlying tree structure. In addition, in order to save storage space at the server site, document indexes can be destroyed after the update operation.

Finally, note that document deletions also necessitate index updates, as the individual keyword scores have to be subtracted from the corresponding tree nodes. In particular, when deleting a stored document, the client first re-constructs the encrypted document index from the metadata file and sends it to the cloud provider. The cloud provider then updates the index in a manner similar to the one explained above. The only difference is that the document index values have to be negated, an operation that is trivially performed (in the ciphertext space) by computing the multiplicative inverse modulo $n^2$ of the Paillier ciphertext.

## 3.6 Top-$k$ Query Processing

Query processing is performed at the client side, by traversing the database index with a *best-first* search algorithm, as illustrated in Figure 3.4. Specifically, the client initializes and maintains two max-heaps: *nodeHeap* and *resultHeap* (line 1). The

**Top**_$k(Q, k)$

1: $nodeHeap \leftarrow \emptyset$; $resultHeap \leftarrow \emptyset$;
2: $\theta \leftarrow 0$;
3: Compute the Bloom filter indexes $\{h_i\}$ associated with all keywords in $Q$;
4: Using PIR, retrieve from the root node the Paillier ciphertexts corresponding to all $h_i$'s;
5: Compute the aggregate score for every child of the root node, and insert that node into $nodeHeap$;
6: **while** ($nodeHeap$ **not** empty) **do**
7: Remove the top entry from $nodeHeap$ and store it into $e$;
8: **if** ($e.score \leq \theta$) **then**
9: **break**;
10: **end if**
11: Using PIR, retrieve from node $e$ the Paillier ciphertexts corresponding to all $h_i$'s;
12: Compute the aggregate score for every child of $e$;
13: **if** ($e$ is a leaf node) **then**
14: Retrieve the metadata files for all documents with score $> \theta$;
15: Compute the actual scores of these documents and insert them into $resultHeap$;
16: $\theta \leftarrow$ score of $k$-th document in $resultHeap$;
17: **else** /* $e$ is an internal node */
18: Insert every child of $e$ into $nodeHeap$;
19: **end if**
20: **end while**
21: **return** $resultHeap$;

Figure 3.4: Top-$k$ query processing algorithm

first one is used to visit nodes in decreasing score value, while the later one (which can be of size $k$) stores the result set that is eventually returned to the client. In addition, the client maintains a threshold value $\theta$ (line 2) that keeps track of the score value of the $k$-th document in $resultHeap$. This threshold is used to terminate the algorithm early, i.e., when there exists no other document that can alter the current top-$k$ result set.

Initially, the client computes the Bloom filter indexes associated with all keywords comprising query $Q$ (line 3). As an example, consider a top-2 query with two keywords, mapping into positions $h_1 = 0$ and $h_2 = 7$ (from left to right) in the Bloom filter vectors shown in Figure 3.3. For simplicity, assume that there are no false positives, i.e., every keyword hashes into a unique location. The algorithm starts from the root node ($e_1$), and utilizes the PIR protocol of Section 2.4 to retrieve the corresponding Bloom filter entries. In our example, the client will retrieve the first and third ciphertexts that contain the required values.

Next, the client computes the aggregate scores of $e_1$'s children ($e_2$ and $e_3$) by adding the corresponding values at positions $h_1$ and $h_2$. Since $e_1$ is an internal node, the client simply inserts $\langle e_2, 12\rangle$ and $\langle e_3, 3\rangle$ into $nodeHeap$ (lines 17-18). Node $e_2$ is visited next, as it has the highest score in the heap (line 7). The same process is repeated, i.e., the client retrieves privately the first and third ciphertexts of $e_2$ and computes the aggregate scores of $e_4$ and $e_5$ (lines 11-12). Subsequently, the client inserts $\langle e_4, 8\rangle$ and $\langle e_5, 4\rangle$ into $nodeHeap$.

The next node removed from $nodeHeap$ is $e_4$, and the client computes (privately) the *estimated* scores of $D_1$ and $D_2$. Node $e_4$ is now a leaf node, so the client retrieves the encrypted metadata files of *both* $D_1$ and $D_2$ (line 14), since both documents can potentially be part of the result set (i.e., their actual scores could be larger than $\theta$). From the metadata files, the client calculates the actual scores (line 15) and, as in our example we assume the absence of false positives, $resultHeap$ is updated to $\{\langle D_1, 4\rangle, \langle D_2, 4\rangle\}$. Additionally, since there are exactly $k = 2$ documents in $resultHeap$, the threshold value $\theta$ is updated to 4 (line 16).

Finally, node $e_5$ is visited next, which has an aggregate score of 4. Given that docu-

ment indexes are aggregated in a bottom-up fashion, none of the other documents in the database can have a score larger than 4. Consequently, the two existing documents in *resultHeap* are at least as "good" as any of the remaining ones, so the algorithm can terminate safely (lines 8-9). At this point, the result set stored in *resultHeap* is returned to the client, along with the corresponding document metadata. The client may then look into the metadata files and decide whether she wants to retrieve the encrypted full documents from the database server.

## 3.7 Optimizations

Public key operations are computationally expensive and, thus, the query processing algorithm described above would be inapplicable to large document collections. Therefore, in this section, we present a number of optimizations, targeting the cryptographic operations of our protocol, which may lead to reasonable query response times, even for large datasets. Section 3.7.1 presents several optimizations at the database server, while Section 3.7.2 introduces a few optimizations at the client side.

### 3.7.1 Server optimizations

**Multiple CPUs.** The PIR protocol of Section 2.4 is the major performance bottleneck in our top-$k$ retrieval algorithm. As shown in a recent study [57], Gentry and Ramzan's scheme (which is arguably one of the more efficient computational PIR protocols) requires many seconds of computing time, even for databases of size less than 100 KB. In our system, a typical index node consists of several thousands of Paillier ciphertexts, each of size 256 bytes. Consequently, executing the PIR protocol

in a single CPU is not a practical implementation.

In our work, we adopt the striping technique of [57] that sacrifices some communication cost in order to achieve significantly faster PIR query response times. The idea is to partition each node into $t$ blocks (Figure 3.5), so that the required Paillier ciphertexts are retrieved by querying each of the $t$ blocks in parallel. This is an ideal scenario for cloud computing platforms (such as Amazon's EC2) that offer large CPU clusters at low cost. Recall that the PIR protocol of Gentry and Ramzan associates a prime power $\pi_j$ with each database record (in our case, Paillier ciphertext). Therefore, the client can simply construct a single PIR query that is applied to all $t$ blocks comprising the node.

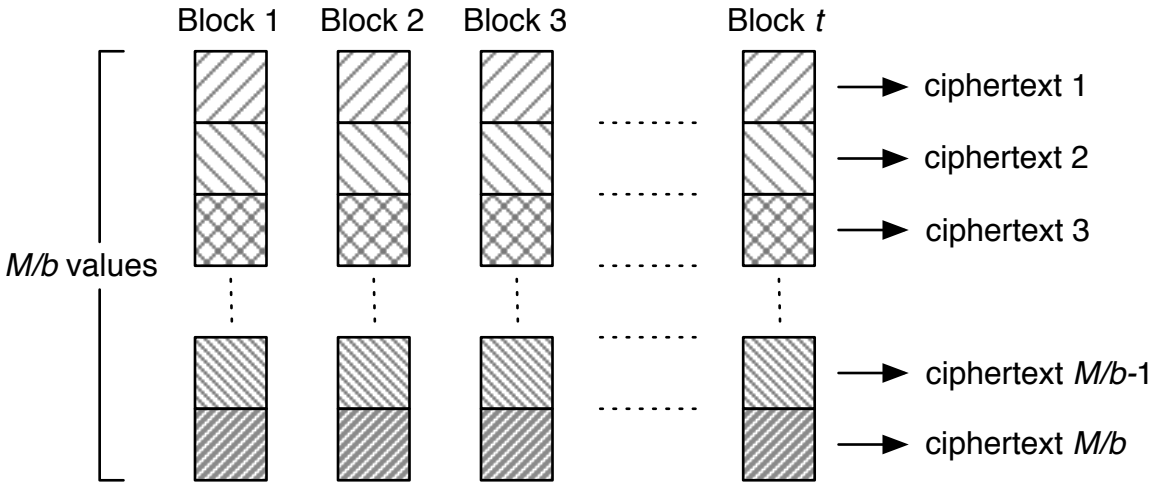



Figure 3.5: Node partitioning

In our implementation, nodes are partitioned into $t = 128$ blocks, i.e., every block holds 2 bytes from each of the $M/b$ ciphertexts. (Using the notation of Section 2.4, $\ell = 16$.) The reason behind this choice is the security constraint of the query generation algorithm that sets a limit on the bit size of the product of all the prime powers $\pi_j$ corresponding to the requested records. Setting $\ell = 16$ allows us to securely retrieve

up to 10 different ciphertexts with a single PIR query, i.e., our system supports keyword search queries with up to $|Q| = 10$ terms.

When the client wants to retrieve a Paillier encryption from a node, she sends a PIR request to the server. The server then utilizes multiple (128) CPUs to return each 2-byte partition of the encryption in a parallel manner. Retrieving small chunks of a Paillier encryption in parallel speeds up the PIR process considerably. Note that, if the cloud infrastructure can provide more than 128 CPUs, we can easily reduce the block size $\ell$ (e.g., to 1 byte or less) in order to utilize the additional CPUs and further reduce the PIR cost.

Finally, note that the multiple CPUs can be leveraged in the index update process as well. Since every Paillier ciphertext is updated independently, we can always process multiple ciphertexts in parallel.

**Chinese Remainder Theorem.** During the setup phase of the PIR protocol, the server computes the transformed database $\beta$ by applying the Chinese Remainder Theorem (CRT) on the original database records. In our setting, document insertions and deletions alter the entire node content, so the CRT has to be computed from scratch after each update operation. Consequently, an efficient implementation of the CRT is of paramount importance. In our work, we chose Garner's algorithm [53], which includes an expensive preprocessing step, but is very efficient during data updates. Specifically, as long as the number of records remains constant (which is true for the index nodes) the preprocessing step has to be executed only once and is independent of the database records (it only depends on the prime powers $\pi_j$ of the PIR protocol). As shown in our experimental results, calculating the CRT using the

pre-computed values is very inexpensive.

### 3.7.2 Client optimizations

**Discrete logarithm.** The most expensive operation at the client side is solving the discrete logarithm problem when extracting the result from the server's reply. This cost is amplified in our implementation, due to the aforementioned node partitioning method. In particular, our approach requires $t|Q|$ discrete logarithm computations per visited node, in order for the client to retrieve $|Q|$ Paillier ciphertexts. Therefore, instead of relying on the Pohlig-Hellman algorithm, we chose to pre-compute all possible results during the query generation algorithm. As it is evident in the protocol description (Section 2.4), when the client generates the PIR query she can compute the $g_{i_j}$ values corresponding to all possible (in our case, $2^{16}$) database records $D_{i_j}$. Furthermore, these pre-computations are very efficient if we apply successive modular multiplications.

**Multiple CPUs.** Similar to the server case, clients can also benefit from the availability of multi-core CPUs and/or high-performance GPU chips. Most operations, including discrete logarithm computations, are fully parallelizable, since they can be performed independently. Nevertheless, we did not explore this possibility in our experiments, i.e., we assumed that the client utilizes a single CPU.

## 3.8 Experimental Results

In this section, we evaluate experimentally the performance of our methods, in terms of storage, communication, and CPU cost at the various entities of our architecture. Section 3.8.1 describes the setup of the experiments, while Section 3.8.2 presents the results.

### 3.8.1 Setup

We developed our programs in C++, utilizing the GMP[3] arithmetic library, and the LiDIA[4] library for computational number theory. We run the client and sender (Alice and Bob) programs on an Intel Core i7, 2.8 GHz CPU, and the server program on an Amazon EC2 instance with 1 Compute Unit. In other words, we did not utilize multiple CPUs at the cloud provider, but rather measured the CPU times required to execute the cryptographic protocols of our methods on Amazon's infrastructure. Table 3.2 summarizes the costs of the basic cryptographic primitives at the three parties involved in our system architecture.

For the document collection, we downloaded the Enron email dataset[5], which is an excellent real life example of the scenario described in Section 3.4. We created several versions of the document collection, corresponding to different values of $N$ and $|W|$. When selecting the keywords for a specific document, we used the terms from the "From:" and "Subject:" fields (excluding certain stop words) and, if there was more room, we chose the most frequent terms from the email's body. For each experiment,

3. `http://gmplib.org/`
4. `http://www.cdc.informatik.tu-darmstadt.de/TI/LiDIA`
5. `http://www.cs.cmu.edu/~enron/`

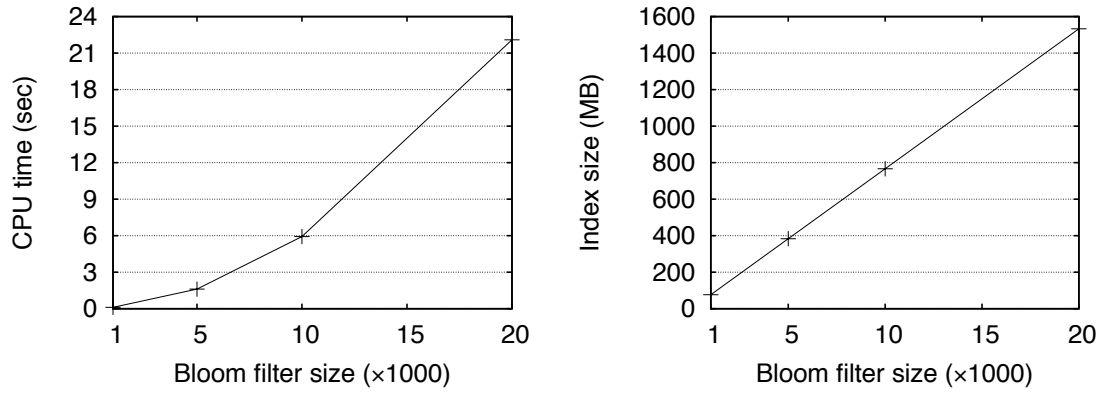
we generated 1000 random queries and run the top-$k$ query processing algorithm of Figure 3.4. We measured the average number of index nodes that were accessed per query, and then used Table 3.2 to derive the corresponding costs. When creating the queries, we tried to mimic the typical user behavior by searching based on the sender and subject fields. If there were not enough keywords in these fields to fill the query, we used random keywords from the email's body.

Table 3.3 summarizes our system parameters, with their default values appearing in bold face. When testing the effect of a single parameter, we fixed the remaining ones to their default values. In the following plots we illustrate the effectiveness of our approach, based on these performance metrics:

- The offline processing and storage cost at the server to store the database index.
- The CPU time at the server to update the database index.
- The CPU time and communication cost at the sender to send a new document to the server.
- The query response time at the client, i.e., the time that elapses from the instance the query is posed, until the actual answer in obtained.
- The communication cost between the client and the server during query processing. This cost includes the transferred metadata files, whose size is fixed to 512 bytes.

### 3.8.2 Results

Figure 3.6(a) shows the CPU time at the server for constructing the complete index tree (for $N = 10000$ documents), as a function of the Bloom filter size $M$. As explained in Section 3.5, the index is initially empty, i.e., every node consists of Paillier encryptions of zero. The cost is dominated by the preprocessing step of Garner's CRT algorithm, because the time to construct the Paillier ciphertexts is negligible (the server simply creates a single Paillier ciphertext and populates all index nodes with the same value). This figure also illustrates the computational complexity of the CRT problem that justifies the expensive preprocessing step at the cloud provider. Figure 3.6(b) depicts the storage cost at the server to hold the database index. As expected, it grows linearly in $M$, and reaches approximately 1.5 GB for $M = 20000$. Note that, the actual storage cost of the index tree is only half of what is shown in this figure. The reason is that, due to the underlying PIR protocol, the server needs to maintain two versions of the index: the "plaintext" version consisting of the Paillier ciphertexts, and the transformed version $\beta$ required by Gentry and Ramzan's scheme.




(a) Index construction time (b) Index size

Figure 3.6: Index construction cost

Figure 3.7 plots the processing time at the cloud provider for updating the index, as a function of $M$. Recall that, for a single node, the update process requires (i) one modular exponentiation and one modular multiplication per Paillier ciphertext, and (ii) one CRT computation per block. Using multiple CPUs in parallel (Section 3.7.1) reduces considerably the processing time at the server, which remains below 350 ms in all cases.

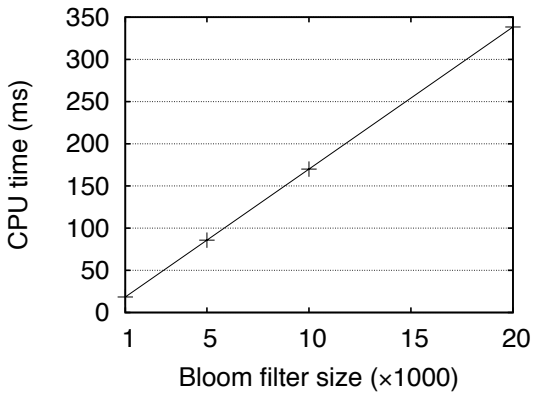
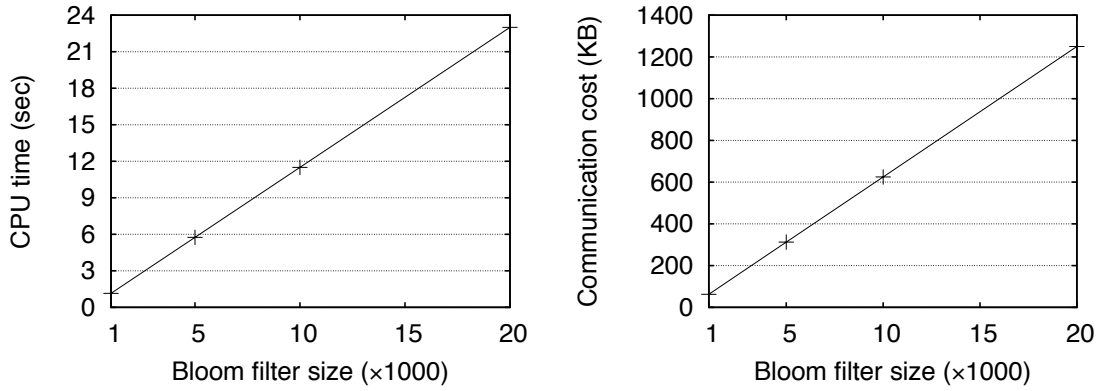



Figure 3.7: Index update cost

Figure 3.8(a) illustrates the CPU time at the sender, as a function of $M$. When sending a new document to the server, the sender has to encode the keyword information on the Bloom filter, and then perform numerous Paillier encryptions to construct the document index. Clearly, the cost of these encryptions can be significant, requiring over 20 sec of processing time for $M = 20000$. One optimization that can be applied to reduce this cost, is for the sender to pre-compute (offline) Paillier encryptions of zero. This may alleviate significantly the online cost, because the vast majority of the Bloom filter counters will always remain zero. Figure 3.8(b) shows the communication cost for the same experiment. It grows linearly in $M$, and ranges from 62 KB to 1.2 MB.

Figure 3.9(a) shows the response time of our top-$k$ query processing algorithm, with

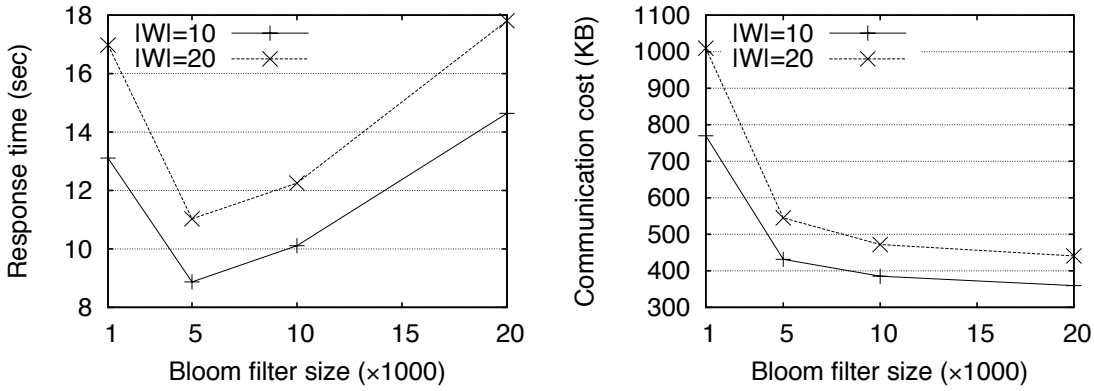



(a) CPU time (b) Communication cost

Figure 3.8: Cost at the sender

respect to the Bloom filter size $M$. When $M$ is small, the number of false positives at the individual Bloom filters is high, thus affecting the accuracy of the computed aggregate scores. As a result, the best-first search algorithm has to visit a lot more index nodes than necessary, a fact that increases the overall query response times. On the other hand, larger Bloom filter vectors are more accurate, and the top-$k$ algorithm visits significantly less number of nodes that result in better performance. Nevertheless, as $M$ increases even further, the cost of the PIR retrievals becomes a dominant factor that eliminates the advantage of the reduced false positive rates. Similarly, Figure 3.9(b) depicts the communication cost at the client for the same experiment. As $M$ increases, the Bloom filters become more accurate, leading to fewer node accesses and, thus, lower communication cost. As evident in Figure 3.9, the value $M = 5000$ achieves a good trade-off between query response time and communication cost.

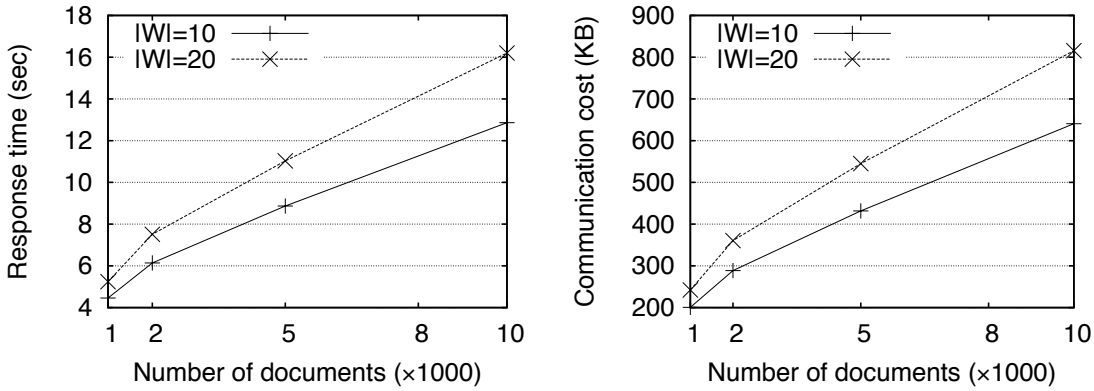
Figure 3.10 illustrates the response time and communication cost of the query processing algorithm, as a function of the number of documents $N$. Recall that, in this experiment, $M$ is fixed to 5000, so increasing the number of documents produces




(a) Response time (b) Communication cost

Figure 3.9: Query processing cost vs. $M$

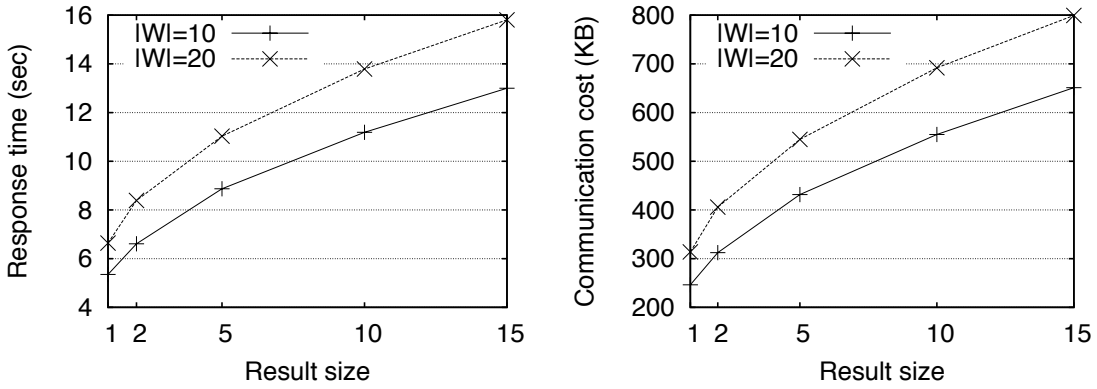
more false positives and, therefore, the performance of our algorithm deteriorates. However, even for $N = 10000$ documents, the algorithm terminates in less than 17 sec and incurs less than 900 KB of communication cost.




(a) Response time (b) Communication cost

Figure 3.10: Query processing cost vs. $N$

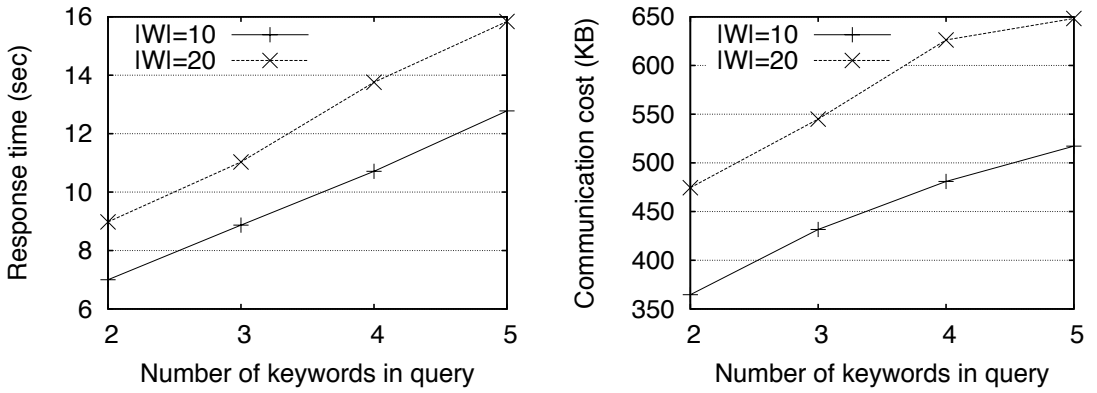
Figure 3.11 depicts the cost of the query processing algorithm, with respect to the number of requested documents $k$. When the client wants to retrieve more relevant documents, the threshold value $\theta$ that terminates the algorithm (Figure 3.4) becomes smaller, thus allowing the algorithm to run further and access more nodes. As a result, both the query response time and the communication cost increase with $k$.




(a) Response time (b) Communication cost

Figure 3.11: Query processing cost vs. $k$

Figure 3.12 shows the response time and communication cost of the query processing algorithm, as a function of the number of keywords $|Q|$ in the client's query. As $|Q|$ increases, the effect of the false positives is amplified, since more Bloom filter indexes are involved in the score computations. Consequently, the number of visited nodes increases slightly with $|Q|$. Note that, the query response time increases faster than the communication cost, because the PIR-related computations at the client side are more expensive for larger values of $|Q|$ (Table 3.2).

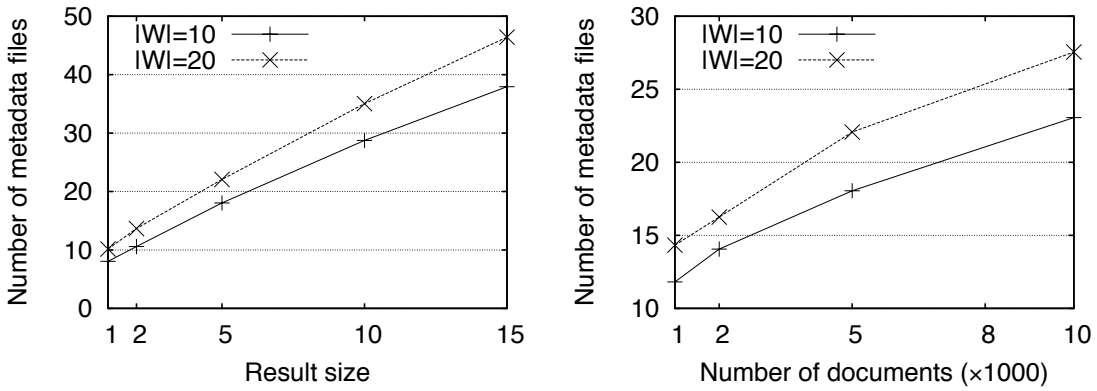



(a) Response time (b) Communication cost

Figure 3.12: Query processing cost vs. $|Q|$

Finally, Figure 3.13 illustrates the number of metadata files that are transferred to

the client during query processing, with respect to $k$ and $N$. As explained previously, increasing either $k$ or $N$ leads to more node accesses and, therefore, more metadata files are sent to the client. Nevertheless, under all settings, less than 1% of the total number of document metadata are transferred to the client. Consequently, the resulting overhead is negligible, thus justifying their use to offset the effect of false positives in the Bloom filters.




(a) Number of metadata files vs. $k$ (b) Number of metadata files vs. $N$

Figure 3.13: Number of metadata files accessed

Table 3.2: Cost of cryptographic primitives at different entities

| **Server (1 Amazon EC2 Compute Unit)** | |
|---|---|
| *Paillier Cryptosystem* | |
| Modular exponentiation | 2.1 ms |
| Modular multiplication | 0.005 ms |
| Encryption | 6.5 ms |
| *PIR (250 elements, 2 bytes each)* | |
| Preprocessing (for CRT) | 103 ms |
| CRT (per block) | 0.4 ms |
| Query (per block) | 7.8 ms |
| *PIR (1250 elements, 2 bytes each)* | |
| Preprocessing (for CRT) | 1.6 sec |
| CRT (per block) | 5 ms |
| Query (per block) | 53 ms |
| *PIR (2500 elements, 2 bytes each)* | |
| Preprocessing (for CRT) | 5.9 sec |
| CRT (per block) | 19 ms |
| Query (per block) | 143 ms |
| *PIR (5000 elements, 2 bytes each)* | |
| Preprocessing (for CRT) | 22 sec |
| CRT (per block) | 81 ms |
| Query (per block) | 377 ms |
| **Sender (Intel Core i7, 2.8 GHz)** | |
| *Paillier Cryptosystem* | |
| Encryption | 4.6 ms |
| **Client (Intel Core i7, 2.8 GHz)** | |
| *Paillier Cryptosystem* | |
| Decryption | 4.5 ms |
| *PIR (2 keywords)* | |
| Query generation | 441 ms |
| Result retrieval | 192 ms |
| *PIR (3 keywords)* | |
| Query generation | 661 ms |
| Result retrieval | 204 ms |
| *PIR (4 keywords)* | |
| Query generation | 667 ms |
| Result retrieval | 230 ms |
| *PIR (5 keywords)* | |
| Query generation | 822 ms |
| Result retrieval | 262 ms |

Table 3.3: System Parameters

| **Parameter** | **Range** |
|---|---|
| $N$ | 1000, 2000, **5000**, 10000 |
| $M$ | 1000, **5000**, 10000, 20000 |
| $k$ | 1, 2, **5**, 10, 15 |
| $\|Q\|$ | 2, **3**, 4, 5 |
| $f$ | **17** |
| $b$ | **4** |
| $\|W\|$ | **10**, **20** |
| $\log n, \log m$ | **1024** |

# CHAPTER 4

# PRIVACY-PRESERVING SIMILAR DOCUMENT DETECTION

## 4.1 Background

The problem of secure similar document detection was first introduced by Jiang et al. [45]. In their approach, Alice and Bob first run a secure protocol to identify the common terms that appear in both datasets (dictionary). Then, similarity is computed with the *cosine* of the angle between two document term vectors. The cosine computation requires a secure inner product protocol, identical to the one described in Section 2.1. Specifically, for Alice to compare a single document against Bob's database, she first uses her public key to encrypt the weights of every term in the dictionary (if the term does not exist in Alice's document, its weight is 0). After Bob receives the encrypted vector, he uses his plaintext term vectors to blindly compute the encryptions of the inner products for all documents. Finally, Alice decrypts the results and computes the similarity between her document and each document in Bob's database. This protocol is computationally expensive, because of numerous public key operations at both parties. Furthermore, its performance degrades as the size of the dictionary space increases. For example, the similarity search between two document sets, each containing 500 documents, takes about a week to complete [45].

The authors of [45] extend their work in [54] with two optimizations. First, to reduce the number of modular multiplications at Bob, they ignore every ciphertext in Alice's vector where the corresponding plaintext value at Bob is zero. Second, to reduce the number of document comparisons, each party applies (in a pre-processing step) a $k$-means clustering algorithm on their documents. The idea is to initially compare only the cluster representatives and measure their similarity. If that similarity value is above a certain threshold, then the documents in both clusters are compared in a pairwise manner. Nevertheless, the drawback of clustering is that it is sensitive to the value of $k$. If $k$ does not accurately reflect the underlying document similarities, it may result in a significant loss in query precision and recall.

Jiang and Samanthula [46] propose the use of $N$-grams in their SSDD protocol. An $N$-gram representation of a document consists of all the document's substrings of size $N$ (after removing all punctuation marks and whitespaces). In general, $N$-grams are considered a better document representation method than term vectors, because they are language independent, more sensitive to local similarity, simple, and less sensitive to document modifications [46]. Specifically, Jiang and Samanthula utilize 3-gram sets and define the similarity between two documents as the Jaccard index of their 3-gram sets. Prior to protocol execution, both parties create the 3-gram sets of their documents and Bob discloses his *global* 3-gram set to Alice. To compare a pair of documents, Alice and Bob create the binary vectors of the corresponding 3-gram sets with respect to Bob's global 3-gram set (let $A$ be Alice's vector and $B$ be Bob's vector). Next, the two parties invoke a secure two-party computation protocol to compute $|A \cap B|$ in an additively split form. Finally, they run a secure division protocol to compute the Jaccard index $J = \frac{|A \cap B|}{|A \cup B|}$. Unfortunately, the above protocol is not secure [9], because Bob has to reveal his global 3-gram set to Alice.

By utilizing this information, Alice can easily check whether a word appears in Bob's global collection, which is an obvious security breach.

Blundo et al. [9] introduce EsPRESSo, a protocol for privacy-preserving evaluation of sample set similarity. It is based on the private set intersection cardinality (PSI-CA) protocol of De Cristafaro et al. [22]. The authors show that one possible application of EsPRESSo is similar document detection and propose a solution based on 3-grams. To compare two documents, Alice and Bob first create the 3-gram sets of their respective documents. Next, Alice hashes her 3-grams and raises the resulting digests to a random number $R_a$ (let's call this set $A$). She then sends $A$ to Bob who, in turn, raises these values to his random number $R_b$ and randomly permutes the set. He also hashes his 3-gram set members and raises the hash values to $R_b$ (let's call this set $B$). Bob then sends both sets back to Alice. Alice removes $R_a$ from $A$ and computes the cardinality of the intersection between $A$ and $B$ ($|A \cap B|$). From this value, she computes the Jaccard index as $J = \frac{|A \cap B|}{|A|+|B|-|A \cap B|}$.

The limitation of the basic EsPRESSo protocol is that its performance depends on the total number of 3-grams that appear in the compared documents. To this end, the authors of [9] introduce an optimization based on the MinHash technique. In particular, instead of incorporating every available 3-gram in the corresponding 3-gram sets ($A$ and $B$), Alice and Bob agree on $k$ distinct hash functions ($H_1, H_2, \ldots, H_k$) to produce sets of size $k$, independent of the total number of 3-grams. Specifically, for $i \in \{1, 2, \ldots, k\}$, each party hashes all their 3-grams with the $H_i$ hash function and select the digest with the minimum value as a representative in their respective set. Once sets $A$ and $B$ are constructed, the EsPRESSo protocol is invoked to compute the Jaccard index between the two documents. The MinHash approximation

reduces considerably the computational and communication costs and is currently the state-of-the-art protocol in secure similar document detection.

## 4.2 Problem Definition

Bob (the server) holds a collection of $N$ documents $\mathbb{D} = \{D_1, D_2, \ldots, D_N\}$. Each document $D_i \in \mathbb{D}$ is represented as a set of pairs $\langle w_i, f_i \rangle$, where $w_i$ is a term appearing in the document and $f_i$ is its frequency (i.e., the number of times it appears in the document). Alice (the client) holds a single document $q$ that is represented in a similar fashion. Alice wants to know which documents in Bob's collection $\mathbb{D}$ are similar to $q$. Note that, if Alice herself holds a collection of $M$ documents, the query is simply evaluated $M$ distinct times.

In this work we propose two protocols with different privacy guarantees. The security of the basic protocol (Simhash, Section 4.3) is identical to the security provided by all existing SSDD protocols:

- For all $i \in \{1, 2, \ldots, N\}$, Alice learns the similarity score between $q$ and $D_i$.
- Bob learns nothing.

On the other hand, the enhanced version of our protocol (Simhash*, Section 4.4) provides some additional security to the server (Bob):

- For all $i \in \{1, 2, \ldots, N\}$, Alice learns whether $D_i$'s similarity score is above a user-defined threshold $t$ (boolean value). The exact score remains secret.

- Bob learns nothing.

We assume that both parties could behave in an adversarial manner. Their goal is to derive any additional information other than the existence of similar documents and their similarity scores. For example, they could be interested in the contents of the other party's documents, statistical information about the terms in the other party's document collection, etc. Finally, we assume that both parties run in polynomial time and are "semi-honest," i.e., they will follow the protocol correctly, but will try to gain any advantage by analyzing the information exchanged during the protocol execution.

## 4.3 Basic protocol

In this section we introduce our basic protocol that reveals the exact similarity score for each one of Bob's documents to Alice. Section 4.3.1 presents the protocol and Section 4.3.2 outlines its security proof.

### 4.3.1 The Simhash protocol

Prior to protocol execution, each party runs a preprocessing step to generate the simhash fingerprints of their documents. The preprocessing includes lower case conversion, stop word removal, and stemming. In the end, each document is reduced to a set of terms and their corresponding frequencies. The simhash fingerprints are then created according to the algorithm described in Section 2.3. In what follows, we use $a$ to denote Alice's simhash (from document $q$) and $b_i$ ($i \in \{1, 2, \ldots, N\}$) to denote the

simhash of document $D_i$ in Bob's database. Recall that all fingerprints are binary vectors of size $l = 64$ bits.

Similarity based on simhash fingerprints is defined as the number of *non-matching* bits between the two bit vectors. In other words, a similarity score of 0 indicates two possibly identical documents, while larger scores characterize less similar documents. Consequently, it suffices to securely compute (i) the bitwise XOR of the two vectors and (ii) the summation of all bits in the resulting XOR vector. Figure 4.1 shows the detailed protocol, where $E(\cdot)$ denotes encryption with Alice's ElGamal public key (which is known to Bob).

**Simhash**

**Input:** Alice has a simhash fingerprint $a$
Bob has $N$ simhash fingerprints $\{b_1, b_2, \ldots, b_N\}$
**Output:** Alice gets $N$ similarity scores $\{\sigma_1, \sigma_2, \ldots, \sigma_N\}$

**Alice**
1: Alice sends to Bob $E(a[0]), E(a[1]), \ldots, E(a[l-1])$;

**Bob**
2: **for** $(i = 1; i \leq N; i++)$ **do**
3: Set $\sigma_i \leftarrow 0$ and compute $E(\sigma_i)$;
4: **for** $(j = 0; j < l; j++)$ **do**
5: **if** $(b_i[j] == 0)$ **then**
6: $E(\sigma_i) \leftarrow E(\sigma_i)E(a[j])$;
7: **else**
8: $E(\sigma_i) \leftarrow E(\sigma_i)E(1)E(a[j])^{-1}$;
9: **end if**
10: **end for**
11: **end for**
12: Bob sends to Alice $E(\sigma_1), E(\sigma_2), \ldots, E(\sigma_N)$;

**Alice**
13: Alice decrypts all ciphertexts and retrieves $\sigma_1, \sigma_2, \ldots, \sigma_N$;

Figure 4.1: The Simhash protocol

First (line 1), Alice encrypts every bit of her fingerprint $a$ and sends $l$ ciphertexts to

Bob. Bob cannot decrypt these ciphertexts but is still able to blindly perform the required XOR and addition operations. In particular, for every document $D_i$ in his database, Bob initializes the encryption of the similarity score to $E(\sigma_i) = E(0)$ (line 3). Next, he iterates over the $l$ bits of the corresponding fingerprint $b_i$. If the bit at a certain position $j$ is 0, then the result of the XOR operation is equal to $a[j]$ and Bob simply adds the value to the encrypted score (line 6). Otherwise, the result of the XOR operation is $(1 - a[j])$ which is also added to $E(\sigma_i)$ in a similar fashion (line 8). After all documents are processed, Bob sends the encrypted results to Alice (line 12). Finally, Alice uses her private key to decrypt the scores and identify the most similar documents to $q$ (line 13).

### 4.3.2 Security

In this section we prove the security of the Simhash protocol for semi-honest adversaries, following the simulation paradigm [49]. In particular, we will show that, for each party, we can simulate the distribution of the messages that the party receives, given only the party's input and output in this protocol. This is a sufficient requirement for security because, if we can simulate each party's view from only their respective input and output, then the messages themselves cannot reveal any additional information.

Alice's input consists of a bit vector $a$ and her output is $\{\sigma_1, \sigma_2, \ldots, \sigma_N\}$. The only messages that Alice receives from Bob are the encryptions of the $N$ similarity scores. The simulator knows Alice's public key and it also knows her output. Therefore, it can simply generate the encryptions of the corresponding scores.

In Bob's case, the input is $N$ bit vectors and there is no output. In the beginning of the protocol, Bob receives $l$ encryptions from Alice. Here, the simulator can simply generate $l$ encryptions of zero. Given the assumption that the underlying encryption scheme is semantically secure, Bob cannot distinguish these ciphertexts from the ones that are produced by Alice's real input.

## 4.4 Enhanced protocol

The basic Simhash protocol has the same security definition as all existing SSDD protocols in the literature. That is, Alice learns the similarity score for every document $D_i$ in Bob's database. Nevertheless, making all this information available to Alice may allow her to construct some "malicious" queries that reveal whether a certain term (or 3-gram) exists in Bob's database. Consider the EsPRESSo protocol as an example. Alice's query may consist of a number of fake 3-grams (i.e., 3-grams that could not appear in Bob's documents) plus a real one that Alice wants to test against Bob's database. After completing the protocol execution, Alice can infer that the 3-gram is present in Bob's database if at least one of the similarity scores is non-zero. This attack is not as trivial to perform with the simhash or MinHash techniques, but it is still possible for sophisticated adversaries to devise similar attacks.

To this end, in this section, we introduce Simhash$^*$, an enhanced version of the basic Simhash protocol that maintains the similarity scores secret. This is the first SSDD protocol in the literature that provides this functionality. In particular, Alice and Bob agree on a similarity threshold $t$ and the protocol returns, for each document $D_i$, a boolean value $\theta_i$ that indicates whether $\sigma_i \leq t$. The detailed protocol is shown in

Figure 4.2.

The first steps of the protocol (lines 1–10) are identical to Simhash, i.e., Bob blindly computes the encryptions of all $N$ similarity scores. However, instead of sending these ciphertexts to Alice, Bob computes, for each $D_i \in \mathbb{D}$, the encryptions of $r_j(\sigma_i - j)$ where $j \in \{0, 1, \ldots, t\}$ (lines 12–13). Specifically, $r_j$ is a uniformly random value that masks the actual similarity score ($\sigma_i$) when it is not equal to $j$. On the other hand, if $\sigma_i$ is equal to $j$, then the computed value is an encryption of 0. Next, Bob uses a random permutation $\pi_i$ for each set of $(t+1)$ ciphertexts corresponding to document $D_i$, and eventually sends a total of $(t+1) \cdot N$ ciphertexts back to Alice (line 16). The different permutations are required in order to prevent Alice from inferring the value of $j$ (i.e., similarity score) that produces the encryption of 0. Finally, Alice concludes that document $D_i$'s similarity score is within the predetermined threshold $t$, if and only if one of the $(t+1)$ ciphertexts corresponding to $D_i$ decrypts to 0 (lines 19–28).

The security proof of the Simhash* protocol is trivial and follows the proof outlined in Section 4.3.2. In particular, only Alice's case is different, since (i) her output is $N$ boolean values $\{\theta_1, \theta_2, \ldots, \theta_N\}$ and (ii) she receives $(t+1) \cdot N$ ciphertexts from Bob. Nevertheless, the simulator knows Alice's output and also knows how the protocol operates. Therefore, for all documents $D_i$ where $\theta_i$ is *true*, the simulator generates $t$ random encryptions plus one encryption of 0. On the other hand, for documents where $\theta_i$ is *false*, the simulator generates $(t+1)$ random encryptions.

**Simhash**$^*$

**Input:** Alice has a simhash fingerprint $a$
Bob has $N$ simhash fingerprints $\{b_1, b_2, \ldots, b_N\}$
**Output:** Alice gets $N$ binary values $\{\theta_1, \theta_2, \ldots, \theta_N\}$

**Alice**
Alice sends to Bob $E(a[0]), E(a[1]), \ldots, E(a[l-1])$;

**Bob**
**for** $(i = 1; i \leq N; i++)$ **do**
  Set $\sigma_i \leftarrow 0$ and compute $E(\sigma_i)$;
  **for** $(j = 0; j < l; j++)$ **do**
    **if** $(b_i[j] == 0)$ **then**
      $E(\sigma_i) \leftarrow E(\sigma_i)E(a[j])$;
    **else**
      $E(\sigma_i) \leftarrow E(\sigma_i)E(1)E(a[j])^{-1}$;
    **end if**
  **end for**
  **for** $(j = 0; j \leq t; j++)$ **do**
    Choose $r_j$, uniformly at random from $\mathbb{Z}_p^*$;
    $E(x_{ij}) \leftarrow [E(\sigma_i)E(j)^{-1}]^{r_j}$;
  **end for**
**end for**
Bob sends to Alice $\{E(x_{ij})\}, \forall i \in \{1, 2, \ldots, N\}, j \in \pi_i(\{0, 1, \ldots, t\})$;

**Alice**
Alice decrypts all ciphertexts and retrieves $\{x_{ij}\}$;
**for** $(i = 1; i \leq N; i++)$ **do**
  **for** $(j = 0; j \leq t; j++)$ **do**
    **if** $(x_{ij} == 0)$ **then**
      **break**;
    **end if**
  **end for**
  **if** $(j > t)$ **then**
    $\theta_i \leftarrow$ **false**;
  **else**
    $\theta_i \leftarrow$ **true**;
  **end if**
**end for**

Figure 4.2: The Simhash$^*$ protocol

## 4.5 Experimental evaluation

In this section we experimentally compare the performance of our methods against existing SSDD protocols. Section 4.5.1 describes the experimental setup and Section 4.5.2 illustrates the results of our experiments.

### 4.5.1 Setup

We compare our protocols against the work of Murugesan et al. [54] that utilizes cosine similarity (labeled as "Cosine" in our results), and EsPRESSo (both the basic protocol and the MinHash optimization) [9] that is based on 3-grams. We implemented all protocols in C++ and leveraged the GMP[1] library for handling large numbers. We ran both the client and the server applications on a 2.4 GHz Intel Core i5 CPU. The performance metrics that we tested include the CPU time, the communication cost, and the precision/recall of the document retrieval process.

The document corpus is a collection of Wikipedia[2] articles. In particular, we selected 103 main articles from diverse topics and, for each article, we also selected a number (around 10) of its previous versions from the history pages of this topic. The total number of documents in the corpus is 1152. For Simhash and Cosine, we applied lower case conversion, stop word removal, and stemming, in order to derive the document terms along with their frequencies. For the EsPRESSo protocols, we extracted the 3-grams as explained in [9]. The total number of terms in the documents is 152,571 and the total number of 3-grams is 10,392.

1. `http://gmplib.org`
2. `http://en.wikipedia.org`

### 4.5.2 Results

In the first set of experiments we investigate the document retrieval performance of the various protocols. We did not test the actual cryptographic protocols, but instead compared the three document representation methods (term vectors, simhash, and 3-grams). The experiments were performed as follows. We run 103 queries, where the query documents were selected to be the most recent versions of the 103 unique articles. Using different threshold values, we observed the precision and soundness of the retrieved documents (recall that we know in advance the "correct" results, since the different versions of each article are very similar to each other). For our methods we used the threshold values $\{2, 3, 4, 5, 6\}$, while for the rest of the protocols we used the values $\{0.6, 0.7, 0.8, 0.9, 0.99\}$. Observe that, for Simhash, larger threshold values imply less similar documents, whereas for the other methods the opposite is true.

We used the *precision* and *recall* as the performance metrics for the document retrieval process. Precision is defined as:

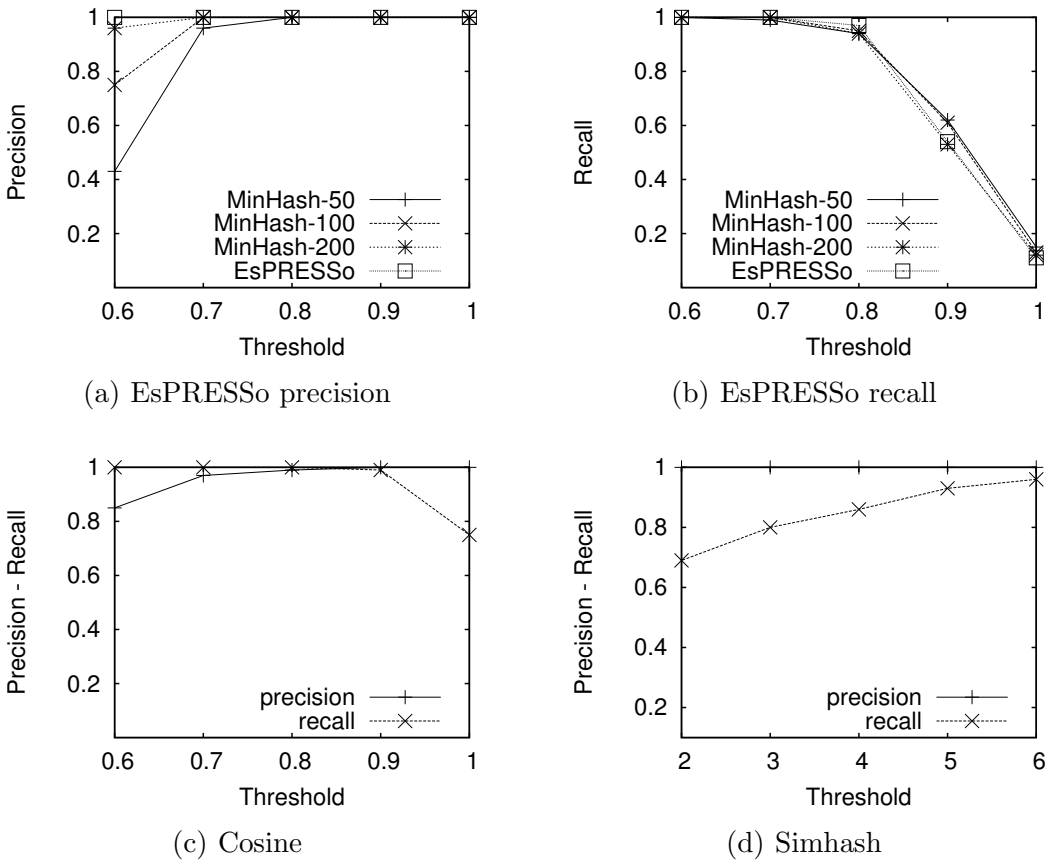
$$precision = \frac{|R \cap V|}{|R|}$$

where $R$ is the set of retrieved documents and $V$ is the total number of documents that satisfy the query. In other words, precision is equal to the fraction of retrieved documents that belong to the result set. Recall, on the other hand, indicates the fraction of the result set that is retrieved by the query and is computed as:

$$recall = \frac{|R \cap V|}{|V|}$$

Figures 4.3(a) and 4.3(b) show the precision and recall curves for the various EsPRESSo protocols. As expected, the basic protocol has the best overall performance and maintains a precision of 1.0 for all threshold values. The MinHash approximations sacrifice some precision for better running times, but they all perform very well for threshold values larger than 0.7. In terms of recall, all EsPRESSo variants are very sensitive to the underlying threshold value, experiencing a large drop when the threshold is larger than 0.8. The Cosine method has a very stable performance, as shown in Figure 4.3(c). In particular, both the precision and recall values remain over 0.75 under all settings. Finally, Simhash exhibits excellent query precision for all threshold values (Figure 4.3(d)). Furthermore, the query recall raises steadily with increasing threshold values and, when the threshold is 6, Simhash retrieves over 96% of the relevant documents.

In the next experiment we measure the computational cost of the various methods. We select MinHash-50 (i.e., MinHash with $k = 50$ hash functions) to represent the EsPRESSo family of protocols, since it has the best performance in terms of CPU time. The experiments were performed as follows. We run the cryptographic protocols for the 103 unique queries and measured the total CPU time, excluding the initial query encryption time (which is performed only once, independent of the database size $N$). From this value we determined the average time needed to compare a pair of documents. Using this measurement, Figure 4.4 depicts the CPU time required to compare one document against a database of size $N$, where $N \in \{100, 500, 1000, 3000, 5000\}$ (the curves also include the query encryption time). Simhash is by far the best protocol among all competitors and it is one order of magnitude faster than MinHash-50. Cosine incurs a very high computational cost, mainly due to the query encryption step that involves tens of thousands of public key encryptions. MinHash-50 is sig-




(a) EsPRESSo precision

(b) EsPRESSo recall

(c) Cosine

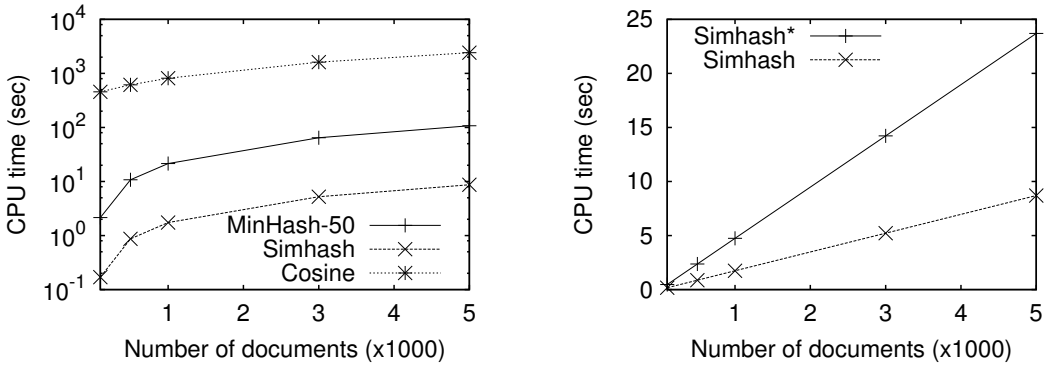
(d) Simhash

Figure 4.3: Precision and recall

nificantly slower than Simhash, because it involves numerous (expensive) modular exponentiations for every document in the server's database.

Figure 4.4(b) shows the CPU overhead of the Simhash* protocol, where the similarity threshold is set to $t = 6$. The additional cost is due to the $(t + 1)$ modular exponentiations that are required to hide a document's similarity score. However, Simhash* is considerably faster than MinHash-50, incurring 23.7 sec of CPU time to compare 5000 documents, as opposed to 107.5 sec for MinHash-50.

Figure 4.5(a) illustrates the communication cost for Simhash, MinHash-50, and Cosine. Clearly, Simhash outperforms significantly both competitor methods, incurring

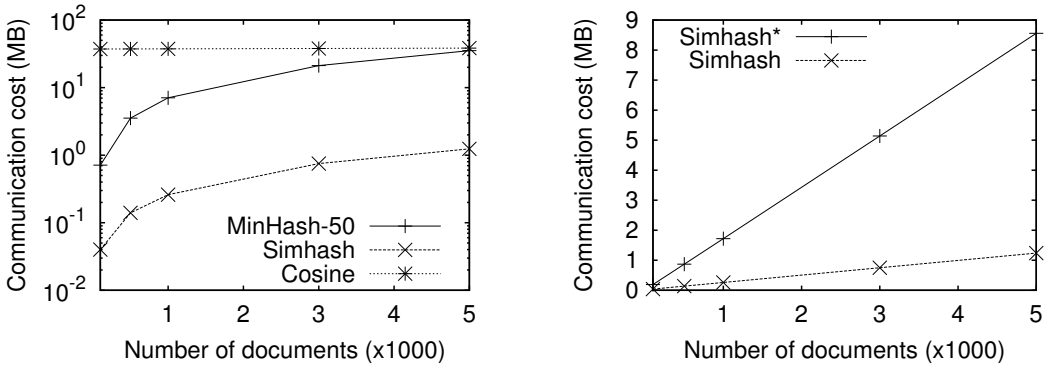



(a) Simhash, MinHash, Cosine (b) Simhash, Simhash*

Figure 4.4: CPU time

a communication cost that is at least 18 times smaller under all settings. For example, to compare one document against a database of size $N = 5000$, requires 1.24 MB of data communication for Simhash, 35.29 MB for MinHash-50, and 38.47 MB for Cosine. The drawback of MinHash-50 is that it has to send 50 ciphertexts plus 50 SHA1 hashes for every document in the database. On the other hand, the overhead for Cosine lies exclusively on the transmission of the encrypted term vector, which is why it seems to remain unaffected by the database size $N$.

Finally, Figure 4.5(b) shows the communication overhead for the Simhash* protocol. In this experiment, the threshold $t$ is set to 6, which necessitates the transmission of 7 ciphertexts for every document in the database. As a result, the communication cost of Simhash* is around 7 times larger than the cost of the basic Simhash protocol. Nevertheless, it is still significantly lower than the competitor methods, requiring just 8.56 MB of data for $N = 5000$ documents.




(a) Simhash, MinHash, Cosine (b) Simhash, Simhash*

Figure 4.5: Communication cost

# CHAPTER 5

# CONCLUSIONS AND FUTURE WORK

## 5.1 Conclusions

Searchable encryption is an important cryptographic primitive that facilitates private keyword searches directly on encrypted data. While this problem is studied extensively in the symmetric key setting, existing public-key algorithms are very restrictive in the types of keyword queries that they allow. To this end, our work introduces the first method for privacy-preserving ranked keyword search on public-key encrypted data. Our solution employs a simple indexing structure, and leverages homomorphic encryption and private information retrieval protocols to process queries in a privacy-preserving manner. Furthermore, we introduce several optimizations for the cryptographic primitives of our approach that reduce the query response times by several orders of magnitude. Using measurements from Amazon's EC2 infrastructure, we show that our method can process ranked keyword searches in less than 17 sec, while incurring less than 900 KB of communication cost.

Secure similar document detection (SSDD) is a new and important research area with numerous application domains, such as patent protection, intelligence collaboration, etc. In these scenarios, two parties want to identify similar documents within their databases, while maintaining their contents secret. Nevertheless, existing SSDD pro-

tocols are very expensive in terms of both computational and communication cost, which limits their scalability with respect to the number of documents. To this end, we introduce a novel solution based on simhash document fingerprints that is both simple and robust. In addition, we propose an enhanced version of our protocol that, unlike existing work, hides the similarity scores of the compared documents from the client. Through rigorous experimentation, we show that our methods improve the computational and communication costs by at least one order of magnitude compared to the current state-of-the-art protocol. Furthermore, they perform very well in terms of query precision and recall.

## 5.2 Future Work

e-Commerce is another area that gains popularity with the advent of easing online technologies. A standard user, without any advanced computer skills, has now the ability to do online shopping. To offer the user different alternatives, e-Commerce providers like Amazon[1] and Ebay[2] use recommender systems [63]. Recommender systems present suggestions to the user by analyzing the users who have similar interests or the items rated by the user in the past.

The most common definition of the concept is the problem of estimating ratings for the items that have not been seen by a user. Once we can estimate ratings for the yet unrated items, we can recommend to the user the item(s) with the highest estimated ratings [5].

1. `http://www.amazon.com`
2. `http://www.ebay.com/`

Recommender systems are usually classified into the following categories, based on how recommendations are made [1] :

- Content-based: Recommendations are based on the items which are rated by the user in the past.
- Collaborative: Recommendations are calculated from the items which are rated by similar users to the user who needs the recommendation.
- Hybrid: Recommendations are based on the combined results of collaborative and content based methods.

In the early 90s, the rapid growth of the Internet started recommender systems based on collaborative filtering. Tapestry [34] is a manual collaborative filtering system which allows a user to get information build upon other users actions or choices. Grouplens [61] is an automated version of Tapestry which was deployed to recommend Usenet articles to their users. Research on recommender systems accelerated while the Internet kept evolving [62, 42, 36].

Despite its popularity, many recommender systems do not take the privacy of the data collected about users into account. The accumulated information about users is a valuable source and is vulnerable to sharing, selling or unauthorized access by other interested parties (e.g. competitors).

One of the pioneering work in privacy-preserving recommender systems is Canny's [14]. He proposed a scheme built upon singular value decomposition (SVD) and homomorphic encryption. This method suffers from high computational cost caused by vector additions over homomorphic encrypted values. Moreover, it assumes to

have many active users online to contribute the calculation of the recommendation securely. Ahmad and Khokhar modified Canny's protocol with El-Gamal encryption [2]. In [59] a method which uses randomized perturbation techniques is proposed, and the same authors built a scheme for vertically partitioned data in [60], however, providing recommendations by using perturbation is proven to be insecure [70]. Tada et.al. developed a method which focuses on the similarity between items [65]. Very recently, Basu et.al. suggested a scheme designed specifically to work in the cloud [6]. Erkin et.al. proposed cryptographic protocols in [24, 26, 27], however, active user contribution is needed, which makes the system more prone to latency. In [28] Erkin et.al. proposed a semi-trusted third party based approach in order to avoid user participation in the calculations. Finally, Erkin et.al. introduced a cryptographic content-based solution in [25].

Current privacy-preserving recommender systems proposed in the literature struggle with, i) The need for active user participation ii) Sacrificed privacy or insecure solutions iii) Third party involvement iv) High communication and computational costs. The practicality of the state-of-the-art solutions are questionable because of the mentioned problems.

As future work, we plan to investigate a better approach to introduce a more applicable scheme by means of security, computational and communicational costs.

# REFERENCES


[1] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *Knowledge and Data Engineering, IEEE Transactions on*, 17(6):734–749, 2005.

[2] W. Ahmad and A. Khokhar. An architecture for privacy preserving collaborative filtering on web portals. In *Information Assurance and Security, 2007. IAS 2007. Third International Symposium on*, pages 273–278. IEEE, 2007.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[4] J. Baek, R. Safavi-Naini, and W. Susilo. Public key encryption with keyword search revisited. In *ICCSA*, pages 1249–1259, 2008.

[5] M. Balabanović and Y. Shoham. Fab: content-based, collaborative recommendation. *Communications of the ACM*, 40(3):66–72, 1997.

[6] A. Basu, J. Vaidya, H. Kikuchi, T. Dimitrakos, and S. Nair. Privacy preserving collaborative filtering for saas enabling paas clouds. *Journal of Cloud Computing: Advances, Systems and Applications*, 1(1):8, 2012.

[7] R. Blaisdell. The cloud commanputing market in 2013, http://www.enterprisecioforum.com/en/blogs/rickblaisdell/cloud-computing-market-2013.

[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[9] C. Blundo, E. D. Cristofaro, and P. Gasti. Espresso: Efficient privacy-preserving evaluation of sample set similarity. In *DPM/SETOP*, pages 89–103, 2012.

[10] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *EUROCRYPT*, pages 224–241, 2009.

[11] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *EUROCRYPT*, pages 506–522, 2004.

[12] D. Boneh, E.-J. Goh, and K. Nissim. Evaluating 2-DNF formulas on ciphertexts. In *TCC*, pages 325–341, 2005.

[13] D. Boneh, E. Kushilevitz, R. Ostrovsky, and W. E. Skeith. Public key encryption that allows PIR queries. In *CRYPTO*, pages 50–67, 2007.

[14] J. Canny. Collaborative filtering with privacy. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 45–57. IEEE, 2002.

[15] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. In *IEEE INFOCOM*, pages 829–837, 2011.

[16] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *ACNS*, pages 442–455, 2005.

[17] M. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM, 2002.

[18] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *IEEE FOCS*, pages 41–50, 1995.

[19] R. Cramer, R. Gennaro, and B. Schoenmakers. A secure and optimally efficient multi-authority election scheme. *European transactions on Telecommunications*, 8(5):481–490, 1997.

[20] G. D. Crescenzo and V. Saraswat. Public key encryption with searchable keywords based on jacobi symbols. In *INDOCRYPT*, pages 282–296, 2007.

[21] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM CCS*, pages 79–88, 2006.

[22] G. P. De Cristofaro, E. and G. Tsudik. Fast and private computation of set intersection cardinality. Technical report, Cryptology ePrint Archive, 2011.

[23] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology*, pages 10–18. Springer, 1985.

[24] Z. Erkin, M. Beye, T. Veugen, and R. L. Lagendijk. Privacy enhanced recommender system. In *Thirty-first Symposium on Information Theory in the Benelux, Rotterdam*, pages 35–42, 2010.

[25] Z. Erkin, M. Beye, T. Veugen, and R. L. Lagendijk. Privacy-preserving content-based recommender system. In *Proceedings of the on Multimedia and security*, pages 77–84. ACM, 2012.

[26] Z. Erkin, M. Beye, T. Veugeri, and R. L. Lagendijk. Efficiently computing private recommendations. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 5864–5867. IEEE, 2011.

[27] Z. Erkin, T. Veugen, and R. Lagendijk. Generating private recommendations in a social trust network. In *Computational Aspects of Social Networks (CASoN), 2011 International Conference on*, pages 82–87. IEEE, 2011.

[28] Z. Erkin, T. Veugen, T. Toft, and R. L. Lagendijk. Generating private recommendations efficiently using homomorphic encryption and data packing. *Information Forensics and Security, IEEE Transactions on*, 7(3):1053–1066, 2012.

[29] L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[30] L. Fang, W. Susilo, C. Ge, and J. Wang. A secure channel free public key encryption with keyword search scheme without random oracle. In *CANS*, pages 248–258, 2009.

[31] C. Gentry. Fully homomorphic encryption using ideal lattices. In *ACM STOC*, pages 169–178, 2009.

[32] C. Gentry and Z. Ramzan. Single-database private information retrieval with constant communication rate. In *ICALP*, pages 803–815, 2005.

[33] E.-J. Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003.

[34] D. Goldberg, D. Nichols, B. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, 1992.

[35] I. Goldberg. Improving the robustness of private information retrieval. In *IEEE S&P*, 2007.

[36] K. Goldberg, T. Roeder, D. Gupta, and C. Perkins. Eigentaste: A constant time collaborative filtering algorithm. *Information Retrieval*, 4(2):133–151, 2001.

[37] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[38] P. Golle, J. Staddon, and B. R. Waters. Secure conjunctive keyword search over encrypted data. In *ACNS*, pages 31–45, 2004.

[39] L. M. V. Gonzalez, L. Rodero-Merino, J. Caceres, and M. A. Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2009.

[40] J. Groth, A. Kiayias, and H. Lipmaa. Multi-query computationally-private information retrieval with constant communication rate. In *PKC*, pages 107–123, 2010.

[41] C. Gu, Y. Zhu, and H. Pan. Efficient public key encryption with keyword search schemes from pairings. In *Inscrypt*, pages 372–383, 2007.

[42] W. Hill, L. Stead, M. Rosenstein, and G. Furnas. Recommending and evaluating choices in a virtual community of use. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 194–201. ACM Press/Addison-Wesley Publishing Co., 1995.

[43] L. Huang, L. Wang, and X. Li. Achieving both high precision and high recall in near-duplicate detection. In *CIKM*, pages 63–72, 2008.

[44] A. S. J. de Lange, A. Longoni. Report ecommerce europe online payments 2012, http://www.ecommerce-europe.eu/stream/report-online-payments-2012.

[45] W. Jiang, M. Murugesan, C. Clifton, and L. Si. Similar document detection with limited information disclosure. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 735–743. IEEE, 2008.

[46] W. Jiang and B. Samanthula. N-gram based secure similar document detection. *Data and Applications Security and Privacy XXV*, pages 239–246, 2011.

[47] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 965–976. ACM, 2012.

[48] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *IEEE FOCS*, pages 364–373, 1997.

[49] Y. Lindell and B. Pinkas. Secure multiparty computation for privacy-preserving data mining. *Journal of Privacy and Confidentiality*, 1(1):59–98, 2009.

[50] H. Lipmaa. An oblivious transfer protocol with log-squared communication. In *ISC*, pages 314–328, 2005.

[51] U. Manber et al. Finding similar files in a large file system. In *Proceedings of the USENIX winter 1994 technical conference*, pages 1–10. San Fransisco, CA, USA, 1994.

[52] G. S. Manku, A. Jain, and A. D. Sarma. Detecting near-duplicates for web crawling. In *WWW*, pages 141–150, 2007.

[53] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC Press, 1997.

[54] M. Murugesan, W. Jiang, C. Clifton, L. Si, and J. Vaidya. Efficient privacy-preserving similar document detection. *The VLDB Journal*, 19(4):457–475, 2010.

[55] M. Naor and B. Pinkas. Computationally secure oblivious transfer. *Journal of Cryptology*, 18(1):1–35, 2005.

[56] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238, 1999.

[57] S. Papadopoulos, S. Bakiras, and D. Papadias. pCloud: A distributed system for practical PIR. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 9(1):115–127, 2012.

[58] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Transactions on Information Theory*, 24(1):106–110, 1978.

[59] H. Polat and W. Du. Privacy-preserving collaborative filtering using randomized perturbation techniques. In *Proceedings of the Third IEEE International Conference on Data Mining*, ICDM '03, pages 625–, Washington, DC, USA, 2003. IEEE Computer Society.

[60] H. Polat and W. Du. Privacy-preserving collaborative filtering on vertically partitioned data. *Knowledge Discovery in Databases: PKDD 2005*, pages 651–658, 2005.

[61] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 175–186. ACM, 1994.

[62] U. Shardanand. *Social information filtering for music recommendation*. PhD thesis, Massachusetts Institute of Technology, 1994.

[63] S. Sohail, J. Siddiqui, and R. Ali. Product recommendation techniques for ecommerce-past, present and future. *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, 1(9):pp–219, 2012.

[64] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE S&P*, pages 44–55, 2000.

[65] M. Tada, H. Kikuchi, and S. Puntheeranurak. Privacy-preserving collaborative filtering protocol based on similarity between items. In *Advanced Information*

*Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 573–578. IEEE, 2010.

[66] C. Wang, N. Cao, J. Li, K. Ren, and W. Lou. Secure ranked keyword search over encrypted cloud data. In *IEEE ICDCS*, pages 253–262, 2010.

[67] W. K. Wong, D. W.-L. Cheung, B. Kao, and N. Mamoulis. Secure kNN computation on encrypted databases. In *ACM SIGMOD*, pages 139–152, 2009.

[68] D. P. Woodruff and S. Yekhanin. A geometric approach to information-theoretic private information retrieval. In *IEEE CCC*, 2005.

[69] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, pages 162–167, 1986.

[70] S. Zhang, J. Ford, and F. Makedon. Deriving private information from randomly perturbed ratings. In *Proceedings of the Sixth SIAM International Conference on Data Mining*, pages 59–69, 2006.