A Secure Similar Document Comparison Protocol

Allan J. Fret, Samuel Li, Alexander Studer, and Maalik Winters

Abstract

A secure document comparison tool has many important applications, including comparison of medical records and plagiarism detection in academia. In this paper, we present the theoretical framework of a secure document comparison protocol we developed, followed by the implementation details. We developed the protocol by combining the document vector space model with homomorphic properties of the Paillier cryptosystem. Our application is written in Java and C, and employs the Apache Lucene and GNU GMP libraries. Finally, we discuss how the security can be improved by implementing the secure k-NN protocol created by Jiang et. al.

Keywords: Paillier cryptosystem, homomorphic, k-NN query, cloud computing, GMP, Java, JNI

1. Introduction

Text-based document comparison for similarity is common task in a variety of fields. However, when the document data is confidential, secure protocols are necessary to ensure the confidentiality of both comparison parties. The comparison context for this paper is a client-server comparison, in which the client queries a single document to be compared to a collection of documents owned by the server. This type of comparison is important in select applications. For instance, a doctor may wish to compare the symptoms of one of his patients against a database of patient records held by another doctor. However, releasing patient medical data is unlawful, so a comparison protocol would have to leak no information about patients not under a doctor's care. A second example is academic journal submissions. A journal receiving a paper submission needs to compare the paper against digital archives to detect plagiarism, without actually revealing the content of the submission.

1.1 Problem Definition

The server owns a database of n records (documents), labeled r_1, r_2, ... r_n. The client owns a query document q. The document comparison function $C(r_i, q)$ compares server document r_i with client document q, and returns a decimal number in [0,1] proportional to the document similarity, with a higher score indicating higher similarity. These scores can determine if further action is necessary in real applications. For instance, a paper with a high comparison score with a previously published paper may be investigated for plagiarism.

The client requires the top-k greatest comparison scores, with $0 < k \leq n$. However, obtaining the top-k scores must be done in a secure manner so that the security of both the client and server documents is not compromised. We define security as follows:

Definition 1: A document comparison protocol is secure if it

- does not reveal the server documents to the client
- does not reveal the client document to the server
- does not reveal any comparison scores to the server

If the protocol revealed comparison scores to the server, the server could learn which documents the client document is most similar to as well as how similar in general it is to the record collection, which is unacceptable because then the server learns information about the client document.

1.2 Our Contribution

We i a solution for the document comparison problem that satisfies Definition 1. Our contributions include:

- a secure comparison protocol
- an implementation of the protocol as a Java GUI application

The protocol fully satisfies the notion of security detailed by Definition 1.

2 Background and Related Work

To understand our protocol, it is critical to first understand the vector space model and the Paillier cryptosystem. The vector space model renders each document as a vector, and compares documents by calculating the mathematical dot product. The Paillier cryptosystem is the cryptosystem of choice in our protocol, because it supports operations on encrypted data through its homomorphic properties.

2.1 Vector Space Model of Comparing Documents

This paper employs the vector space model to compare documents. In this model, each document is represented as a vector, with all server and client vectors having an equal number of elements. We define a *term* as a word. The number of unique terms across all documents in the server collection is the dimensionality of all the vectors. Each element index in the vector space corresponds to a term, with the specific term determined by alphabetical order.

Example 1:

The server has a (simple) collection with three documents. The following list consists of the unique terms across all three documents: apple, banana, orange, pomegranate, strawberry. Since there are five unique terms, each document vector has five dimensions, in the same order.

The actual numeric content of each vector element is the product of the *term frequency* and the *inverse document frequency*. The term frequency, denoted $tf_{t,d}$, is the number of times the specific term t occurs in the specific document d. The inverse document frequency, denoted idf_t, is defined in Formula 1.

Formula 1: $$idf_t = 1 + \log(N / df_t),$$

where df_t is the document frequency of term t, or the number of documents in the collection which contain term t, and N is the number of documents in the collection.

If each vector element were simply the term frequency, the is no discernment between relatively insignificant words which are bound to occur frequently, and relatively significant words which occur less frequently. For example, many documents in a collection may contain the word “number”, while the word “cryptography” might only occur in a few documents. Including an inverse document frequency factor accounts for the flaw in pure term frequency values by making a term less significant the more collections it appears in. This is because if df_t is small, idf_t becomes larger. The complete value for each vector elements is the tf-idf score, defined in Formula 2.

Formula 2: $$\text{tf-idf}_{t,d} = tf_{t,d} * idf_t$$

The actual indexing, which is calculating the values of $tf_{t,d}$ and df_t, will be handled by a robust third-party search library. Once a document is represented as a vector of tf-idf scores, it can be used in document comparison calculations.

In order to compare two documents, we compute the dot product of two mathematically normalized server vectors. The vectors should be mathematically normalized so that the two documents being compared are given equal weight, regardless of respective document lengths and unique term counts. The dot product is an intuitive measure of similarity, because if two vectors share a common term, both will have a nonzero tf-idf value in the same index, and contribute a positive value to the dot product. Larger dot products indicate that two documents are more similar, while smaller dot products indicate they are less similar.

2.2 Paillier Cryptosystem

The Paillier cryptosystem and its properties are central to our comparison protocol. The private key generation, encryption, and decryption algorithm are as follows. Encryption can be done by both server and client, while only the client is capable of private key generation and decryption.

Private Key Generation:

1. Choose 512-bit random primes p, q
2. Compute n = pq
3. Set g = n + 1
4. Compute $\lambda = \phi(n) = (p-1)(q-1)$
5. Compute $\mu = \phi(n)^{-1} (mod\ n)$

Note that λ is equal to Euler's totient function. The public key, which is made available to both the client and server, is <g, n>. The private key, which is necessary for decryption and available to only the client, is $< \phi(n), \mu >$. After generating the public and private keys, both the client and server are capable of encryption by the following algorithm.

Encryption: For message $m \in Z_n$

1. Choose random $r \in Z_n^*$
2. Compute ciphertext $c = E(m) = g^m r^n \pmod{n^2}$

Given the corresponding private key, the client can decrypt an encrypted message.

Decryption: For ciphertext c = E(m)

Let $L(x) = \frac{x-1}{n}$

Compute plaintext $m = D(E(m)) = L(c^{\lambda} \pmod{n^2})\ \mu \pmod{n}$

2.3 Homomorphic Properties

The Paillier cryptosystem is the chosen cryptosystem for our protocol due to its homomorphic properties. These properties allow for arithmetical operations to be applied to two ciphertexts, or encrypted messages, and are vital to the operation of the application that was developed. There are two homomorphic properties that make our encrypted document comparison protocol possible: the additive homomorphic property and the multiplicative homomorphic property. The additive homomorphic property states that if you multiply two ciphertexts together, than the result of this operation is the encryption of the sum of the two plaintext messages. The multiplicative homomorphic property states that raising a ciphertext to a constant exponent is equivalent to the encryption of the origin plaintext message multiplied by the constant.

Additive Homomorphic Property: $c_1 * c_2 = E(m_1) * E(m_2) = E(m_1 + m_2)$

Multiplicative Homomorphic Property: $c^k = E(m)^k = E(m * k)$

The homomorphic properties are critical to our application because it enables dot product calculation between an encrypted and plaintext vector. They are also used in the *SMAX* protocol so that only the top-k results are returned to the client and not all of the encrypted results which further increases the overall security of the application itself as only the required amount of information is sent back to the user. Example 2 demonstrates how the dot product of an encrypted client vector and a plaintext server vector. The calculation would be handled by the server.

Example 2: Given $v_c = < E(m_{c1}), E(m_{c2}), E(m_{c3}) >$ and $v_s = < m_{s1} , m_{s2} , m_{s3} >$, we use the additive homomorphic property to obtain

$$E(m_{c1})^{ms1} = E(m_{s1} * m_{c1}), E(m_{c2})^{ms2} = E(m_{s2} * m_{c2}), \text{ and } E(m_{c3})^{ms3} = E(m_{s3} * m_{c3}).$$

Next, we apply the same property again to obtain.

$$E(m_{s1} * m_{c1}) * E(m_{s2} * m_{c2}) * E(m_{s3} * m_{c3}) = E(m_{s1} * m_{c1} + m_{s2} * m_{c2} + m_{s3} * m_{c3}).$$

This is the encryption of the dot product, and it is sent to the client to be decrypted.

3 Proposed Framework

In this section, we will discuss the proposed framework for our application. As we know, privacy is always a concern. Specially, wherever sensitive data is handled. This project addresses organizations that are in constant managing of this data. We present a client/server application to securely compute the similarities between two documents. While computing the similarities neither party will be able to access the document. The protocol that we use assures the protection of the data from unauthorized users. In our application, the user needs to select the collection where he/she wants to compare the document. After this, he/she is going to submit the document that he wants to compare to the selected collection. This process is a secure procedure because the application encrypts the document before sending it to the server. By doing this, we prevent the server of knowing the information that it contains.
For example, let say that a doctor would like to know the risk factor of heart disease in a specific patient. If the doctor request a file of a patient from a server, the privacy of that patient is being violated because unauthorized users can access that data. With our application, the doctor is going to send the document with the information history of the patient. The application will encrypt the information of the document before sending it. Then, the server receives the encrypted document, so there is no way for the server to know any information regarding the patient. Afterwards the server will calculate the similarity of the encrypted document with all the documents in the collection chose by the user and it will return a similarity score between 0 to 1 for each document.

3.1 Architecture of framework

The protocol is divided into the following components. The full protocol is shown in Algorithm 1.

3.1.1 Indexing

Prior to the client query, the server needs to create a vector of tf-idf components for each document in its collection. To do this, it needs to identify all the unique terms across all the documents in the collection. The server identifies a client which desires to query to the collection, and sends the dictionary of unique collections terms to the client. *The client creates a vector for the query document according to the server dictionary, but instead of using tf-idf values, the client vector consists only of term-frequency values*. This is because the client should not know how the dictionary terms are distributed across the server collection. Additionally, if the client document includes terms not in the server dictionary, they are not represented in the vector. Recall that the dot product value indicates similarity, so only terms occurring in both documents should contribute to the dot product.

3.1.2 Encryption

Our standard for security requires that the server and client documents are not revealed to each other. However, if the document vectors are revealed, substantial information is leaked because the vectors indicate which terms are in the document as well as the relative frequency of the terms. For secure computation, we need to encrypt the client vector before it is sent to the server. The client generates a private key as described in the previous section, and uses it to encrypt each component of the query vector. The client sends the query vector, the public key, and an integer k, to receive the top-k most similar scores.

3.1.3 Encrypted Dot Product Calculation

Once the server receives the encrypted query vector from the client, it will calculate the encrypted dot product between the query vector and every document vector in the collection, by employing the homomorphic properties of Paillier encryption. The server will send the list of encrypted products to the client, who will then decrypt them.

Algorithm 1. SecureDocumentComparison(d_c , <d_1, ...d_n>) → <$s_{1,\dots}$ s_k>

```
Require: Server has <$d_1$,...$d_n$> and client has $d_q$.
1: Server:
   (a). for i = 1 to n do
            Index($d_i$)
   (b). Create dictionary of terms
   (c). for i = 1 to n do
            create vector $v_i$
   (d). Send dictionary to client
2: Client:
   (a). Index($d_q$)
   (b). Create vector $v_q$
   (c). PaillierEncrypt($v_q$) → E($v_q$)
   (d). Send E($v_q$) to server
3: Server:
   (a). for i = 1 to n do
            homomorphic($v_q$) → E($s_i$)
   (b). Send <E($s_i$), ...E($s_n$)> to client
4: Client:
   (a). PaillierDecrypt(<E($s_i$), ...E($s_n$)>) → <$s_1$,...$s_n$>
```

4 Technical Tools

This section will elaborate on the technical tools and libraries required in the implementation of our protocol and the user interface of our application, and the next section explains how our application combines all these tools. We implemented our protocol with Java and C, and the user interface with Java GUI. This is because the necessary libraries are written in Java and C, and Java GUI applications are straightforward to build and support the client-server networking operations we require. Lucene is necessary to index the documents, and GMP is necessary to handle the cryptographic calculations.

4.1 Apache Lucene

Lucene is an information retrieval library written in Java and maintained by the Apache Software Foundation. It supports industry-standard information retrieval functions. In our application, we use Lucene to calculate the term frequency ($tf_{t,d}$) and inverse document frequency (idf_t) of a term, as described in the background section. Lucene parses the documents written in natural languages like English, and counts word frequencies to obtain $tf_{t,d}$ and idf_t. Specifically, Lucene indexes each document in the collection, and creates the comprehensive dictionary.

4.2 Java GUI

In order for users to be able to interact with our program we created a graphical user interface (GUI) in Java using the NetBeans IDE. Java GUI provides a variety of user interface tools such as text boxes, buttons, and file searchers that we need. The NetBeans IDE is useful because it allows dragging and dropping to build the GUI, instead of hard-coding the exact application details. We require two separate interfaces for the server and client. The server interface allows the user to select the directories which contain their database files, index these files, and open and close the server. On the client side, the user can connect to the server, select a file they want to compare and a database they want to compare to. Once the server is open and a client has submitted a request, the server will do the calculation and send the result to the client.

The idea behind our application is for the person behind the server to run the application, control the databases that they want to use, run the server, and then leave it on for clients to connect to. In order for clients to connect, they must be provided with the hostname of the server through some prior communication. There is the option to select a default port, or the client and server can agree on that as well. We used java's native socket library to create a client/server system. The application uses multithreading to allow multiple clients to connect at once and select databases without any interferences, and the server can be closed at any time without issue.

The server is able to set a value for the top-k similarity scores to send back to the client, the client is also able to request a top-k of its own, but the request will be overridden by the server's option.

4.3 GNU GMP Library

The GNU GMP library is a C library which is primarily used to manipulate large integers that have too many bits for a normal program to handle in an efficient manner. GMP has a multitude of built-in functions that manipulate the big integers that are created and initialized in the code. In our experiments and development, the main usage of GMP was in our Paillier Cryptosystem that we implemented to ensure data security. The encryption and decryption functions that are required for the Paillier Cryptosystem utilize 512 bit integers which are too large to process in Java, thus the GMP library is invoked in order to carry out operations on these big integers that are used to secure the data flow throughout the application and protocol itself. The GMP library also includes built-in random number generating functions and protocols that are mainly used to manipulate and generate the random elements in the encryption and decryption scheme that was implemented and utilized in our application.
The GMP library is more efficient than counterparts in other languages, namely, Java's BigInteger library. As is well know, C implementations are simpler and faster than Java implementations in general due to lack of overhead features such as garbage collection.

4.4 JNI (Java Native Interface)

The Paillier Cryptosystem is implemented with GMP which is a C library, yet our base application is coded in Java. To allow communication between Java and C code, we use the Java Native Interface, or JNI. JNI is the method that is used to communicate between our Paillier Cryptosystem coded in C, and the application functions coded in Java. JNI is a predefined library in the Java language that allows a Java developer to make function calls in Java and then generate a C Header file. Once this header file is generated, the function declarations are then copied into the C program, and developer compiles the C files into a library files using a special compiling command in the command terminal. After the library files are created, the Java code then can invoke the functions from the C code after the library is loaded into the program by the System.load() method called before the main function. In the C code, there are also some specific data types that have to be utilized so that the conversion from C to Java is complete, such as jstring, which is utilized to convert a string in C to a format that can be utilized in the Java code.

4.5 Sockets and Ports

Our method for connecting the client to the server was through sockets. A socket is a communication channel between two devices. In order to initialize a socket, the server needs to open a specific port, and then client needs to connect to to that port. Both the client and the server need to be on the same network, and the device address and port number have to be agreed upon in advance. Java supports socket communication with a Socket object. Both parties can send information through the channel through *serialization*, which is the process by which objects are converted into bitstreams.

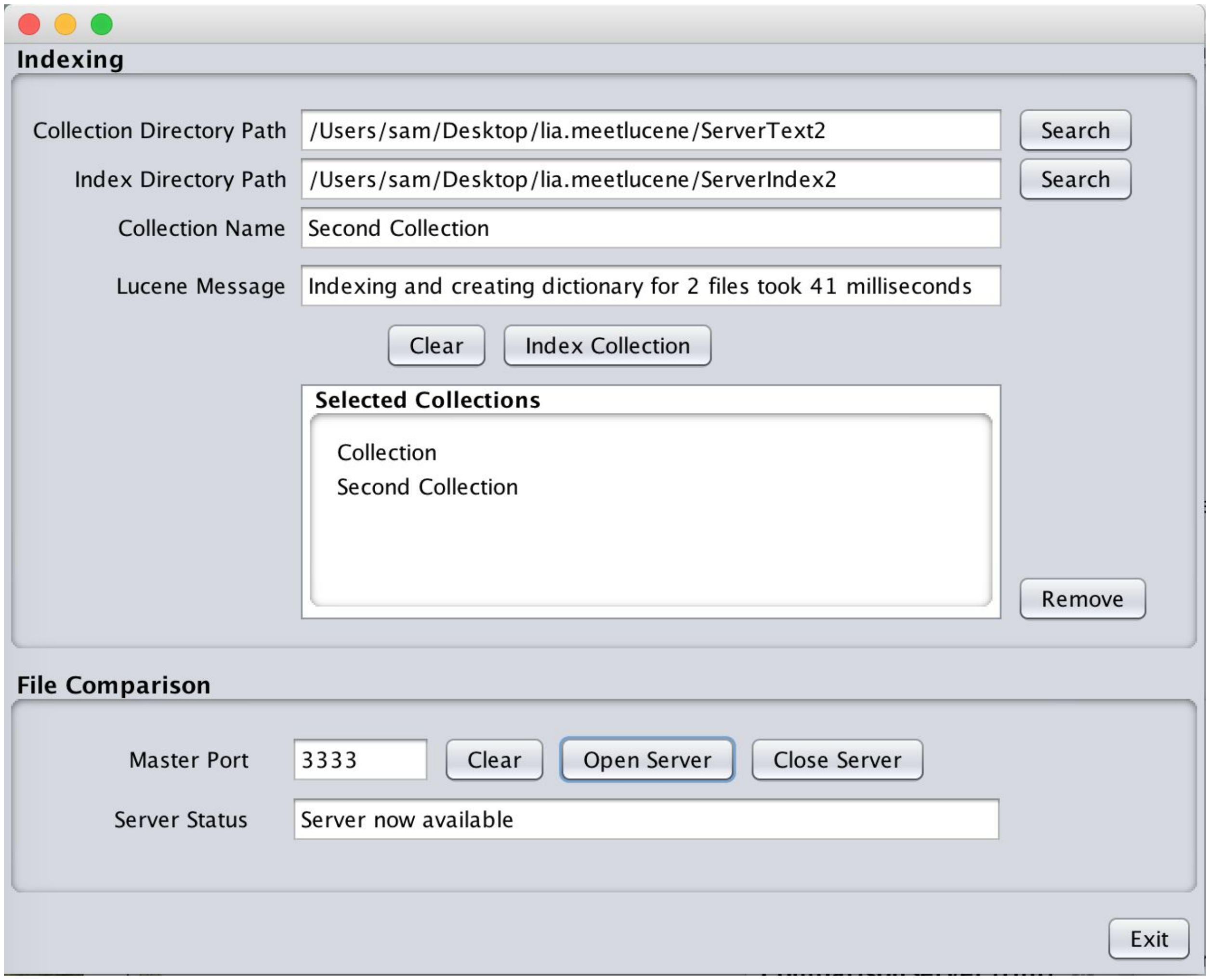
4.6 Multithreading

Ideally the comparison application should allow the server to communicate with multiple clients. More specifically, clients should not be queued, but their queries should be handled at the same time. We can implement this feature through multithreading. A thread is the smallest sub-process unit, and each process has at least one thread (the main thread). Threads can run at the same time. On the server, we create one thread within the server process for each unique client. Java supports multithreading with the Thread object and the Runnable interface.

5 Results and Analysis

This section demonstrates the user interface of our application. Additionally, it analyzes the time complexity of our protocol.

5.1 Implementation



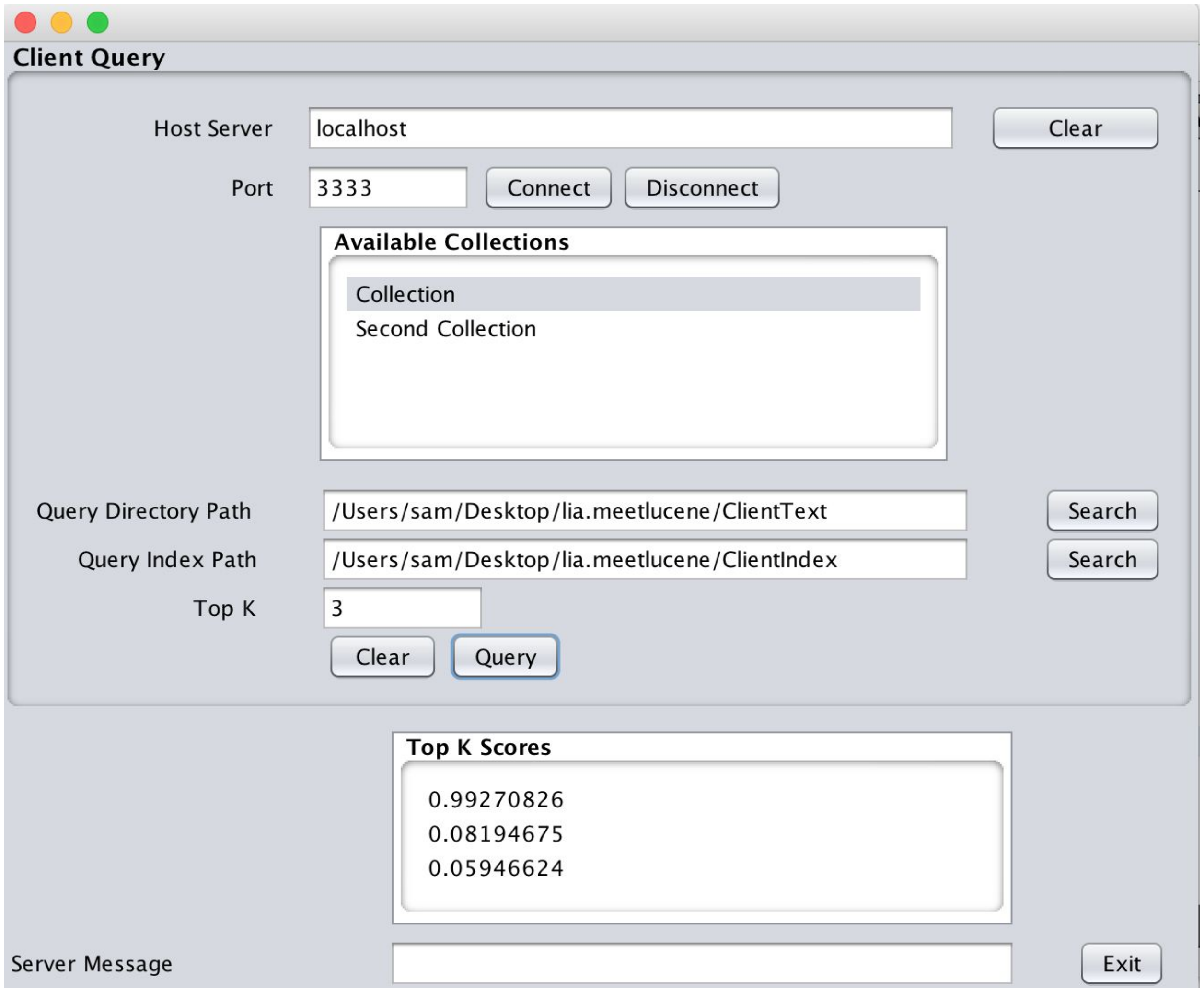
Collection Directory Path: /Users/sam/Desktop/lia.meetlucene/ServerText2
Index Directory Path: /Users/sam/Desktop/lia.meetlucene/ServerIndex2
Collection Name: Second Collection
Lucene Message: Indexing and creating dictionary for 2 files took 41 milliseconds

Selected Collections
Collection
Second Collection

Master Port: 3333
Server Status: Server now available

Figure 1: Comparison Application, Server

Figure 1 shows the server-side GUI of our application. The server selects the directory containing all the document text files. It also selects an empty directory, the index directory path, in which the Lucene-created files will be stored. The server names the collection, then clicks the “Index Collection” button to index the collections. The indexed collection will then be displayed in the “Selected Collections” list. Collections can be removed by selecting the collection name and clicking the “Remove” button. The server manager enters the master port it wishes to open, and selects the “Open Server” button. A successfully-opened port will causes a successful server status message, as displayed.



Host Server: localhost
Port: 3333

Collection
Second Collection

Query Directory Path: /Users/sam/Desktop/lia.meetlucene/ClientText
Query Index Path: /Users/sam/Desktop/lia.meetlucene/ClientIndex
Top K: 3

```
0.99270826
0.08194675
0.05946624
```

Figure 2: Comparison Application, Client

The client enters the name of the server, as well as the port number, and attempts to connect with the "Connect" button. Upon connection, the server sends the names of the available collections. The client selects the desired collection, and the server returns the dictionary for that collection. The client uses the dictionary and Lucene to index the query document and make a normalized term-frequency vector. Each component of the vector is encrypted, and the encrypted vector is sent to the server to calculate the dot product. The server returns the list of scores, which are decrypted and displayed in the GUI.

5.2 Time Complexity

We found that the client query vector encryption is the bottleneck operation. The other operations are computationally inexpensive relative to encryption. Lucene indexing is very efficient, and only needs to be done once for the entire server collection. Homomorphic calculations and decryption is only done once per server document, but every entry in the query vector requires encryption. The encryption function itself is time consuming because of the r^n factor, since r can be as great as n. While there is an exponentiated term in the homomorphic dot product calculation process, the exponent is an unencrypted server value that is significantly less than the public key value, while during the encryption process there is an r^n factor in every calculation.

The encryption of a single vector is bounded by O(t) where t is the number of unique terms in the server dictionary. This is because the number of unique terms is equal to the vector size. Additionally, encryption of a single vector is bounded by O(b^2) where $pq < 2^b$. In other words, b is the number of bits in the bitwise representation of the public variable n in the Paillier scheme.

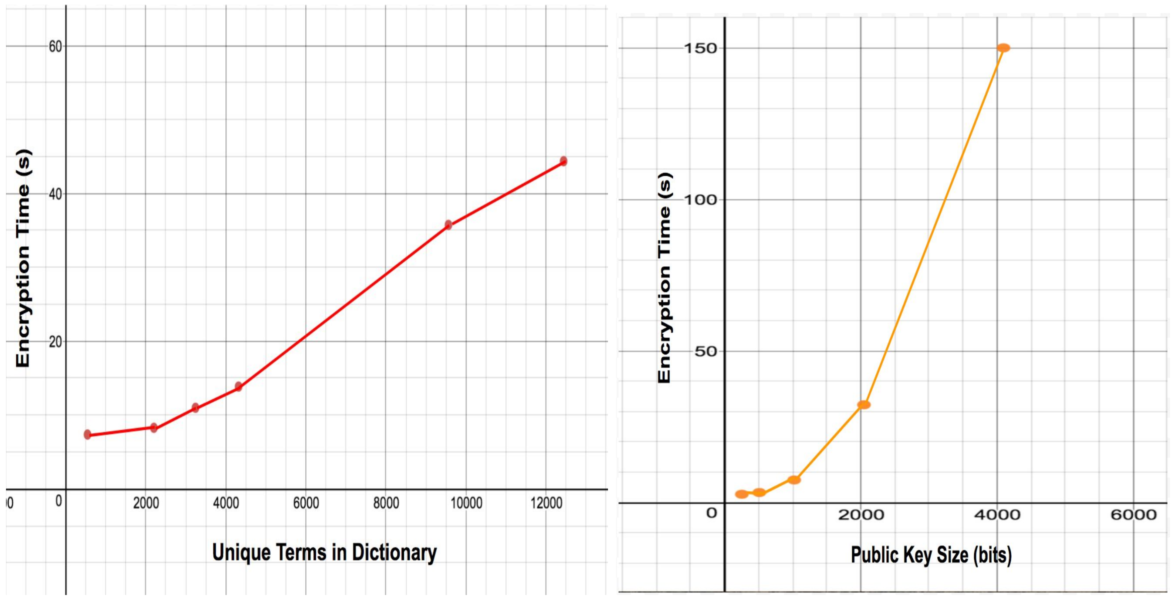


Figure 3: Query vector encryption time vs server dictionary size (left), vs Paillier key size (right)

Test cases carried out on varying parameters as shown in Figure 3 also indicate a linear and quadratic relationship between encryption time and dictionary size, and encryption time and Paillier key size respectively. Our empirical results are consistent with the Paillier cryptosystem, because when we double n, the term r^n in the encryption function may quadruple.

5.3 Further Improvements

Right now the server sends all the encrypted products to the client, who then decrypts the products and displays the top-k after sorting them. Therefore, the parameter k is not interpreted by the server in any way. A security improvement would be having the server only send the top-k scores, so that the client does not gain the scores for the entire server collection. The server can fix a ceiling for k so that the client cannot set k arbitrarily large.

In order to only send the top-k scores back to the client, the server needs to compare encrypted products. Jiang et. al. developed a protocol called SMIN achieves that. We can create a binary tree of comparisons on the encrypted scores, where the first score is compared to the second, the third to the fourth, and so on. The result of such a binary tree is the greatest encrypted score. We then remove the greatest score from the list, and repeat the binary tree comparison process on the remaining encrypted scores.

6 Conclusion

To conclude, secure document comparison in a client query and server collection context is necessary in many applications including comparison of medical data and academic papers. We have developed a protocol to compare a client document with all the document in a server collection. We employed the vector space model, which renders each document as a vector, and the Paillier cryptosystem, whose homomorphic properties permit operations on encrypted data.

We used the Apache Lucene indexing library, GNU GMP library, and the Java Native Interface to create our application. The application includes a Java GUI that communicates between server and client with sockets. Additionally, the server supports multithreading, allowing for multiple clients to communicate with the server at once. Finally, we discussed an improvement in the security of our application, adapted by the work of Jiang et. al.

7 References

C.D. Manning, P. Raghavan, and H. Schütze. *An Introduction to Information Retrieval*, chapter Scoring, Term Weighting, and Vector Space Model, pages 109-135. Cambridge University Press, Cambridge, England, 2009.

T. Granlund and the GMP development team. The GNU Multiple Precision Arithmetic Library, edition 6.1.2. Free Software Foundation Inc. 2016.

E. Hatcher, O. Gospodnetic, and M. McCandless. *Lucene in Action*, Second Edition, chapters 1-3, pages 1-103. Manning Publications, 2009.

B.K. Samanthula, H. Chung, W. Jiang. An Efficient and Probabilistic Secure Bit-Decomposition. ASIA CCS'13, May 8-10, 2013, Hangzhou, China.

B.K. Samanthula, Y. Elmehdwi, W. Jiang. k-Nearest Neighbor Classification over Semantically Secure Encrypted Relational Data. IEEE Transactions on Knowledge and Data Engineering, Vol. 27, No. 5, May 2015.

O'Keeffe, Michael. "The paillier cryptosystem." A Look Into The Cryptosystem And Its Potential Application, college of New Jersey (2008).

Y. Elmehdwi, B.K. Samanthula, W. Jiang. Secure k-Nearest Neighbor Query Over Encrypted Data in Outsourced Environments. arXiv:1307.4324v1[cs.CR].

T. Volkhausen. Paillier Cryptosystem: A Mathematical Introduction. 2006.