Secure Semantic Search Over Encrypted Big Data in the Cloud

A Thesis

Presented to the

Graduate Faculty of the

University of Louisiana at Lafayette

In Partial Fulfillment of the

Requirements for the Degree

Master of Science

Jason W. Woodworth

Summer 2017

Secure Semantic Search Over Encrypted Big Data in the Cloud

Jason W. Woodworth

APPROVED:

Mohsen Amini Salehi, Chair Assistant Professor of Computer Science The Center for Advanced Computer Studies	Vijay Raghavan Professor of Computer Science The Center for Advanced Computer Studies
Anthony Maida Associate Professor of Computer Science The Center for Advanced Computer Studies	Xindong Wu Associate Professor of Computer Science The Center for Advanced Computer Studies
Mary Farmer-Kaiser Dean of the Graduate School	

DEDICATION

To my parents, Roswitha Dauenhauer, Kevin Woodworth, and Rebecca Woodworth, and to all my friends and loved ones.

ACKNOWLEDGMENTS

I sincerely thank my supervisor, Professor Mohsen Amini Salehi, for his constant encouragement and passion for computer science and, especially, for his guidance and support. Thanks to my thesis committee, Xindong Wu, Anthony Maida, and Vijay Raghavan. Thanks to Hoang Pham for his assistance in the work of this thesis. Finally, thanks goes to the Center for Advanced Computer Studies and the Graduate School at the University of Louisiana at Lafayette for their support and guidance.

TABLE OF CONTENTS

DEDICATION . . . iv

ACKNOWLEDGMENTS . . . v

LIST OF FIGURES . . . viii

CHAPTER 1: Introduction . . . 1
- 1.1. Motivations for Semantic Searchable Encryption . . . 2
- 1.2. Research Problem and Objectives . . . 3
- 1.3. Methodology Overview . . . 3
- 1.4. Thesis Organization . . . 4

CHAPTER 2: Survey of Related Literature . . . 6
- 2.1. Overview . . . 6
- 2.2. Searchable Encryption . . . 6
- 2.3. Semantic Search . . . 8
- 2.4. Semantic Search over Encrypted Data . . . 8
- 2.5. Clustering Methods for Searching . . . 9
- 2.6. Summary . . . 10

CHAPTER 3: Problem Formulation and Architectural Overview . . . 12
- 3.1. Problem Formulation . . . 12
- 3.2. Architecture . . . 13
- 3.3. Client Application . . . 14
- 3.4. Cloud Processing Server . . . 14
- 3.5. Cloud Storage . . . 15
- 3.6. Summary . . . 15

CHAPTER 4: Secure Semantic Search Over Encrypted Data in the Cloud . . . 17
- 4.1. Overview of Upload and Search Processes . . . 17
  - 4.1.1. Upload Process . . . 17
  - 4.1.2. Search Process . . . 18
- 4.2. Search Schemes . . . 21
  - 4.2.1. Naïve Scheme: Full Keyword Semantic Search (FKSS) . . . 21
  - 4.2.2. Space-Efficient, Fully Secure Scheme: Selected Keyphrase Semantic Search (SKSS) . . . 22
  - 4.2.3. Space-Efficient, Accuracy Driven Scheme: Keyphrase Search With Frequency (KSWF) . . . 23
- 4.3. Security Analysis . . . 23
- 4.4. S3C Performance Evaluation . . . 26

4.4.1. Metrics for Evaluation . . . 26
4.4.2. Benchmarks . . . 28
4.4.3. Evaluating Relevance . . . 29
4.4.4. Evaluating Performance . . . 30
4.4.5. Evaluating Overhead . . . 32
4.4.6. Evaluating the Impact of Query Length . . . 33
4.4.7. Evaluating Scalability . . . 35
4.5. Discussion . . . 36

CHAPTER 5: Secure Semantic Search Over Encrypted Big Data in the Cloud . . . 38
5.1. Overview . . . 38
5.2. Partitioning Process . . . 40
5.2.1. Initializing Centroids . . . 40
5.2.2. Computing Distance . . . 41
5.2.3. Shard Distribution . . . 42
5.2.4. Shard Abstraction . . . 42
5.3. Search Process . . . 43
5.3.1. Abstract Comparison . . . 44
5.4. Evaluation . . . 44
5.4.1. Evaluating Performance . . . 44
5.5. Discussion . . . 46

CHAPTER 6: Implementation . . . 48
6.1. Command Line Implementation . . . 48
6.2. Web Interface . . . 49

CHAPTER 7: Conclusion and Future Works . . . 51
7.1. Discussion . . . 51
7.2. Future Works . . . 52
7.2.1. Search-Level Access . . . 52
7.2.2. Text Summarization for Search-Level Access . . . 53
7.2.3. Dynamic Shard Creation . . . 53
7.2.4. Dynamic Pruning Aggression . . . 53
7.2.5. Dynamic Keyphrase Extraction . . . 54
7.2.6. Caching Abstract-Query Similarities . . . 54

BIBLIOGRAPHY . . . 55

ABSTRACT . . . 61

BIOGRAPHICAL SKETCH . . . 62

LIST OF FIGURES

Figure 2.1. A taxonomy of the major topics reviewed. Here is some extra text to just fill space and junk. 6

Figure 3.1. Overview of the system architecture and processes. Parts within the solid-line group indicate items or processes on the client side which are considered trusted. Parts in the dashed-line group indicate those in the cloud processing server. All components in the cloud are considered untrusted. 16

Figure 4.1. A sample of the query parsing done by SKSS. 22

Figure 4.2. Queries used for testing relevance. Queries in category 1 target a small set of specific, known documents within the collection, while queries in category 2 target a broad set of documents not necessarily known to the user. 28

Figure 4.3. TSAP@10 score for the specified query for each system. Once the system has returned a ranked list of results, a score is computed based off of a predetermined relevance each file has to the given query. 29

Figure 4.4. Total search time in each scheme. This includes the time taken to process the query, communicate between client and server, and perform searching over the index. The results are averaged over 50 runs. 30

Figure 4.5. Time to process the query. This includes query modification and hashing into the trapdoor. The results are averaged over 50 runs. 31

Figure 4.6. Time it takes to perform the search on the hashed index on the cloud. This includes the time taken to find all files in the hashed index that contain any hashed terms in the query trapdoor and rank them with the scheme's respective functions. The results are averaged over 50 runs. 32

Figure 4.7. Size of the inverted index for each system. An entry denotes a hashed keyword mapped to a set of file identifiers. 33

Figure 4.8. Time it takes to construct the hashed index upon server startup. This operation includes sequentially reading an index file hosted on the cloud server which contains all data for the inverted index and document sizes table and storing it in hash tables. 34

Figure 4.9. The total search time to for an expanding query. This includes the time to process and modify the search query, communicate between client and server, and rank in the cloud. The horizontal axis shows the number of words (minus stopwords) in the query. The results were averaged over 50 runs. 35

Figure 4.10. Time taken to search for different dataset sizes. Resulting times are the mean of 50 runs performed with multiple three word queries. The dotted line shows the time taken to search on the hashed index in the cloud, the dashed line shows the time taken for query modification, and the solid line shows total time taken for the search (including query modification and index searching). 36

Figure 4.11. Size of the index file for different dataset sizes. The horizontal axis plots the size of the dataset used in gigabytes, while the vertical axis plots the associated index size in megabytes. 37

Figure 5.1. Time taken to search over the chosen shards on the cloud. This includes time taken to find files in the shard that match the query trapdoor and rank them. Results are the average of 200 searches spanning 10 different queries consisting of 3 words. 45

Figure 5.2. The average number of terms in a single shard being searched over (i.e. the size of the shard). Each result is the average of the size of 3 chosen shards from 10 benchmark queries. 46

Figure 6.1. A screenshot of the web interface for an S3C client. For each file returned in a search, the client is able to edit, remove, and download the file. 49

Figure 6.2. A screenshot of the interface used to edit a document. 50

CHAPTER 1: INTRODUCTION

The rapid growth of networking speeds, file size requirements, and the need to access data remotely has made cloud storage a necessary and widely utilized solution for companies and individuals who want to store many files without the burden of maintaining a personal data center. However, despite the advantages cloud storage offers, many potential clients abstain from using it due to valid concerns over file security on public remote servers, and thus desire stronger cloud security [FSL15, SCF$^+$14]. Most notably, many businesses note security as their top concern for utilizing cloud services [Bro15].

Cloud storage providers traditionally offer security by encrypting user documents on their remote servers and storing the encryption key remotely. However, this allows internal and external attackers to access unauthorized data if they can locate the key. In particular, internal attacks (e.g. an employee stealing data) are difficult to protect against, as they cannot be stopped by stronger network protections [Sen13].

One proven solution that addresses this concern is to perform the encryption on the user's local machine before it is transferred to the cloud [SWP00]. Unfortunately, this has two major negative effects. First, it introduces a higher space and time overhead than standard cloud storage solutions. Second, it limits the user's ability to interact with the data, as the cloud server's ability to read, manipulate, and parse encrypted data will be limited. Most notably, the ability to search over the data in the cloud is removed.

Although solutions for searchable encryption exist, they often do not consider the *semantic* meaning of the user's query, do not *rank* documents based on their relevance to the query, or do not support multi-phrase search queries.

Additionally, these solutions impose a large storage, memory, and/or processing *overhead*. This often makes them ill-suited for big-data-scale datasets. This thesis offers a solution to these problems.

1.1. Motivations for Semantic Searchable Encryption

The motivation for this research comes from problems typically faced by an organization with increasingly large amounts of data with sensitive information. This organization would have system users who may not remember exact keywords in the documents they are looking for, or may want to retrieve documents similar to what they are asking for. Users can potentially also require the ability to perform the search on their thin client devices (e.g. tablets or smart phones).

One example of such an organization is a law enforcement agency with encrypted police records, with officers who would like to search over records with their PDAs while on the move. This organization would require a system that provides security for their documents, as well as a search mechanism which only required the user to enter a plaintext search query in order to receive search results based on the documents' relevance to the query. They would also need a semantic system that would search for documents under related terms, thus cutting down on repetitive searching. For example, searches for "robbery" should also bring up results for "burglary", "break in" should bring up results for "forced entry", and so on. Due to the potentially huge number of files the organization would need to store, the solution should incur a limited storage overhead to minimize costs and scale well to big data.

1.2. Research Problem and Objectives

The overall research objective of this thesis is to establish a system that can perform a semantic search over big-data-scale encrypted data living on the cloud, which has been encrypted at the user end. The system should be able to perform a semantic search while maintaining minimum processing and storage overhead. For that purpose, in this thesis we answer the following research problems:

- How to semantically search a multi-phrase query over encrypted data stored in the cloud? Additionally, how to do this search processing on the cloud without revealing sensitive data to the cloud?
- How to rank results of a search based on semantic relevance to a user's query?
- How to provide an approach that incurs minimum storage and processing overhead so that it is scalable for big data?

1.3. Methodology Overview

This thesis presents two systems which attempt to solve the research problems described in the previous section. The first, entitled **S**ecure **S**emantic **S**earch over encrypted data in the **C**loud (S3C), performs a semantic search on locally encrypted data that lives in the cloud. Its approach parses and encrypts data on the client side, which is then used to build a centralized hashed index on the cloud's processing server. When performing a search, the system uses online resources to inject semantic information into the query, before hashing it, to match similar tokens in the hashed index.

The second system, named **S**ecure **S**emantic **S**earch over encrypted **B**ig **D**ata on the **C**loud (S3BD), expands upon S3C by breaking down the centralized hashed index into smaller partitions and clusters them based on topic. These partitions can then be selected based on the user's search query and searched in parallel. The result is a much more scalable and memory efficient system.

The rest of this thesis is dedicated to explaining how these two systems work and how they meet the research goals.

1.4. Thesis Organization

This thesis is organized into chapters as follows:

- Chapter 2 presents background information and a survey of research works in this area and positions our work against them. Four major areas of related works are reviewed: searchable encryption, semantic searching, semantic searchable encryption, and clustering methods as they apply to searching. Works will be discussed in terms of what they contributed and how they relate to this thesis.

- Chapter 3 presents the core of the architecture used by both S3C and S3BD. It gives an overview of the components that exist in the architecture and their responsibilities and position on the user premises and on the cloud. It also presents a problem formulation that mathematically defines the problem of semantically searching encrypted data.

- Chapter 4 explains schemes for semantic searching over encrypted data. It investigates the general approach for parsing data to allow it to be searched over when encrypted, and how the system extracts semantic data from the query to search semantically. Several specific schemes for implementing the general approach are proposed, each being tuned to a different type of dataset to achieve a different result. Finally, it presents an experiment carried out in a realistic environment to show the effectiveness of the system. Materials in this chapter are derived from [WSR16]:

  - **Jason Woodworth**, Mohsen Amini Salehi, Vijay Raghavan, *S3C: An Architecture for Space-Efficient Semantic Search over Encrypted Data in the Cloud*, in Workshop on Privacy and Security of Big Data (PSBD) as part of the 2016 IEEE Big Data Conference, USA, 2016.

- Although the methods developed in chapter 4 enable semantically searching over encrypted data, they do not scale to big data scale datasets. In Chapter 5, we develop methods that are customized to scale for big data. It details how methods from S3C are promoted for a big data scenario by partitioning the data set into topic-based shards and pruning irrelevant shards. It then presents an experiment showing the effectiveness of this method.

- Chapter 6 offers details on the implementation of these systems and notes practical problems that arose with them. Both a command line interface and web interface were developed as a prototype and proof of concept for the ideas presented.

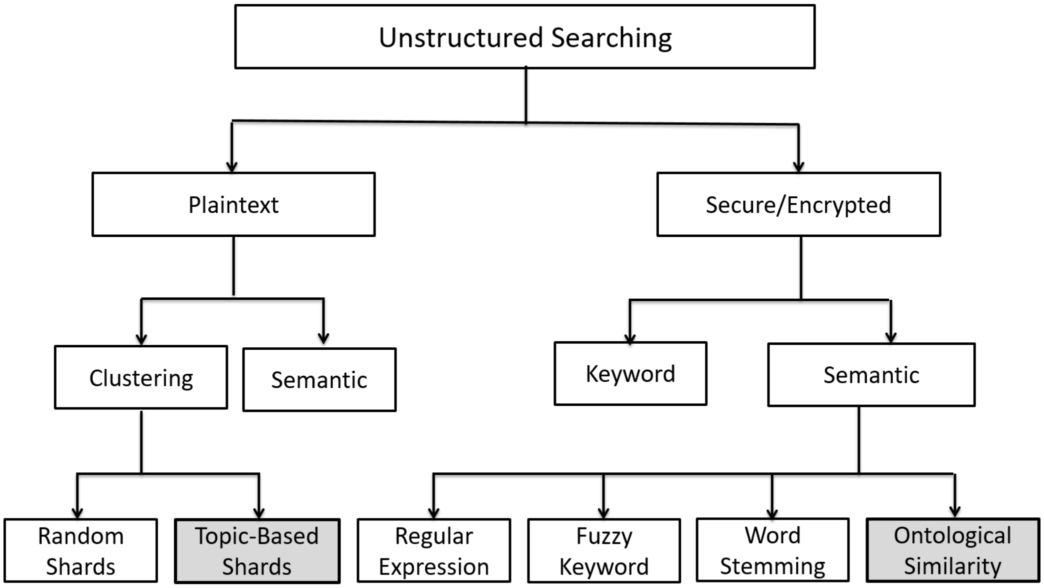
- Chapter 7 concludes the work, discusses results, and notes directions of further research.

CHAPTER 2: SURVEY OF RELATED LITERATURE

2.1. Overview

This chapter provides a survey of other research works undertaken in the fields most related to this work and position the contribution of our works against them. This work primarily builds upon four fields of research: searchable encryption, semantic searching, semantically searchable encryption, and clustering used for searching. Figure 2.1 shows a taxonomy of topics reviewed in this chapter.

Figure 2.1. A taxonomy of the major topics reviewed. Here is some extra text to just fill space and junk.



2.2. Searchable Encryption

Solutions for searchable encryption (SE) are imperative for privacy preservation on the cloud. The majority of SE solutions follow one of two main approaches, the first of which being to use cryptographic algorithms to search the encrypted text directly. This approach is generally chosen because it is provably secure and requires no storage overhead on the server, but solutions utilizing this method are generally slower [SWP00], especially when operating on large storage blocks with large files. This approach was pioneered by Song *et al.* [SWP00], in which each word in the document is encrypted independently and the documents are sequentially scanned while searching for tokens that match the similarly encrypted query. Boneh *et al.* produced a similar system in [BDCOP04] which utilized public key encryption to write searchable encrypted text to a server from any outside source, but could only be searched over by using a private key. While methods following this approach are secure, they often only support equality comparison to the queries, meaning they simply return a list of files containing the query terms without ranking.

The second major approach is to utilize database and text retrieval techniques such as indexing to store selected data per document in a separate data structure from the files, making the search operation generally quicker and well adapted to big data scenarios. Goh [G$^+$03] proposed an approach using bloom filters which created a searchable index for each file containing trapdoors of all unique terms, but had the side effect of returning false positives due to the choice of data structure. Curtmola *et al.* [CGKO11] worked off of this approach, keeping a single hash table index for all documents, getting rid of false positives introduced by bloom filters. The hash table index for all documents contained entries where a trapdoor of a word which appeared in the document collection is mapped to a set of file identifiers for the documents in which it appeared. Van Liesdonk *et al.* further expanded on this in [vLSD$^+$10] with a more efficient search by using an array of bits where each bit is either 0 or its position represents one of the document identifiers. These methods are generally faster, taking constant time to access related files, but are less provably secure, opening up new amounts of data to potential threat. All of the mentioned methods only offer an exact-keyword search, leaving no room for user error through typos and cannot retrieve works related to terms in the query.

2.3. Semantic Search

Much of the work into searching semantically has been done in the context of searching the web [Man07, AMBR15, TCP$^+$16]. Some of these works, such as RQL by Karvounarakis [KAC$^+$02], require users to formulate queries using some formal language or form, which leads to very precise searching that is inappropriate for naïve or everyday users. Others [GLBG99, LUM06] aim for a completely user-transparent solution where the user needs only to write a simple query with possible tags, while others still [GMM03, HH00] aim for a hybrid approach in which the system may ask a user for clarification on the meaning of their query. All of these methods use some form of query modification coupled with an ontology structure for defining related terms to achieve their semantic nature. In addition, these ontology structures often need to be large and custom-tailored to their specific use cases or domain, making them very domain-dependent and unadaptable to different areas. Surprisingly, few of the works in this field offer a ranking of results, instead having the user choose from a potentially large pool of related documents.

2.4. Semantic Search over Encrypted Data

Few works at the time of writing have combined the ideas of semantic searching and searchable encryption. Works that attempt to provide a semantic search often only consider word similarity instead of true semantics.

Li *et al.* proposed in [LWW$^+$10] a system which could handle minor user typos through a fuzzy keyword search. Wang *et al.* [WRYU12] used a similar approach to find matches for similar keywords to the user's query by using edit distance as a similarity metric, allowing for words with similar structures and minor spelling differences to be matched. Amini *et al.* presented in [SCTB16] a system for searching for regular expressions, though this still neglects true semantics for another form of similarity. Moataz *et al.* [MSCBC13] used various stemming methods on terms in the index and query to provide more general matching. Sun *et al.* [SZXC14] presented a system which used an indexing method over encrypted file metadata and data mining techniques to capture semantics of queries. This approach, however, builds a semantic network only using the documents that are given to the set and only considers words that are likely to co-occur as semantically related, leaving out many possible synonyms or categorically related terms.

2.5. Clustering Methods for Searching

The clustering hypothesis states that "Closely associated documents tend to be relevant to the same requests" [Rij79]. This idea has been expanded upon in many ways to form the body of research that investigates document clustering and its effects in information retrieval and searching. Clustering has largely been used in two main ways: partitioning the central index into static shards, independent of user search queries, and clustering in a query specific manner based on the results of searches with the query [LC04]. Solutions following the latter approach have the potential to outperform the static clustering approach [TVR02], they are largely impractical for large data sets.

The former approach has been studied extensively, especially in the domain of web searching [BYMH09, BDH03], but these systems still demand a high computational cost to search over big data. Relatively few works have specifically focused on the idea of clustering the central index into shards based on topics. This idea was pioneered by Xu and Croft [XC99], who showed that making shards of a dataset's index more homogeneous (i.e. the contents of the shards are based around the same topic) improved the effectiveness of a system over standard distributed information retrieval. They used the k-means clustering algorithm with a KL-divergence distance metric to create the shards, then determine which shard should be searched by a query by estimating the likelihood that the query would come from the shard's language model. Liu and Croft [LC04] expanded upon this by using more updated language modeling techniques to better smoothen their estimations. However, neither of these works were appropriate for large scale data.

Kulkarni *et al.* [KC10] adapted these methods to larger scale datasets by performing the k-means clustering on a smaller sample of the dataset, then inferring from the documents' language models which shard those not included in the sample would belong to. These works differ from ours in that they are only designed to operate on plaintext datasets. Before this work, there was no attempt to create a topic-based clustering system that would operate on secured encrypted datasets. Additionally, these models perform clustering on documents, whereas our work is designed to cluster terms from the documents, which was more effective given our encrypted approach.

2.6. Summary

There has been much work done in the fields of searchable encryption and semantic searching. However, there has been little done in the intersection of these fields, where our work lies, specifically in the domain of ontological similarity. Additionally, while there has been work on clustering data into topic-based shards to enhance search performance on plaintext data, there has been no work to apply this approach to encrypted data. In the next chapters, we describe our approach to combining ideas from these fields to create a system that can semantically search over encrypted big data.

CHAPTER 3: PROBLEM FORMULATION AND ARCHITECTURAL OVERVIEW

Both S3C and S3BD share several elements in their architectural setup and system model, as they are both designed to solve the same research problems. This chapter presents a formal definition of our research problems and a description of the architecture as a workflow of actions spanning through local and cloud machines [PSRB16].

3.1. Problem Formulation

The research problems can be formally defined in the following formulation:

- A Vocabulary of plaintext words $V = \{v_1, v_2, v_3, \ldots, v_n\}$ which constitutes a language (e.g. English)
- A Document (represented as a set of words) $d_i = \{d_{i1}, d_{i2}, d_{i3}, \ldots, d_{in}\}$ where $d_{ij} \in V$
- A multi-phrase Query $q = \{q_1, q_2, q_3, \ldots, q_n\}$ where $q_i \ \in \ V$
- A Collection of documents $C = \{d_1, d_2, d_3, \ldots, d_N\}$
- A list of Relevant Documents $R(q) \subseteq C$ where R is a function for determining relevance based on a query

The aim of the search system is to find $R(q)$ using q as a guide for what elements of C it should contain. To ensure the results of $R(q)$ are as relevant as possible, we consider adding semantics to the searching process. This adds the following elements:

- A modification process $M(q)$ which enriches q with semantic data.
- A modified query set $Q = M(q)$ which contains additional related terms and ideas related to q.

- A weighting system $W(Q)$ to weight the terms in Q based on their closeness to the original query.

Introducing semantic data to the search process allows the system to return results that are more meaningfully related to the original query. Weighting is utilized to ensure that the original terms in a document contribute more to that document's ranking than a related term. The introduction of encrypting data to the problem adds the following elements:

- A ciphertext version of the original Vocabulary $V' = \{H(v_1), H(v_2), H(v_3), \ldots, H(v_n)\}$ where H is a hash function
- A Collection of encrypted documents $C' = \{E(d_1), E(d_2), E(d_3), \ldots, E(d_N)\}$ where E is an encryption method
- A list of relevant documents $R'(q) \subseteq C'$

Formally, the challenge is to find the relevant list of elements in C' while still using a plaintext multi-phrase query, and to have $R'(q)$ be as similar to $R(q)$ as possible

3.2. Architecture

The systems have three main components, namely the *client application*, the *cloud processing server*, and the *cloud storage block.* The lightweight client application is hosted on the user's machine, and is the only system in the architecture that is assumed to be trusted. Both cloud units are expected to be maintained by a third party cloud provider and are thus considered "honest but curious". In our threat model, both cloud systems and the network channels between all machines should be considered open to both external and internal attacks. Figure 3.1 presents an overview of the three components and processes associated with them in the system. The partitioner and shards components, however, are exclusive to S3BD.

3.3. Client Application

The client application provides an interface for the user to perform a document upload or search over the data in the cloud. It is responsible for parsing and extracting information from plaintext documents and encrypting them before they are uploaded to the cloud.

When the user requests to search, the system expands the query based on the semantic search scheme and transforms the query into the secure query set (i.e. trapdoor) to be sent to the cloud. The user will then receive a ranked list of documents and can select a file for the system to download and decrypt.

In the S3BD system, the client application is also responsible for determining which partition of the central index to search over at search time. In order to determine this, the client houses information about what is the cloud's partitions.

3.4. Cloud Processing Server

The cloud server is responsible for constructing, updating, and housing the central index and other related data structures based on the parsed and processed data sent from the client. The structures are created entirely out of hashed or encrypted tokens to keep the server oblivious to the actual file content.

When the server detects that the client has requested to search, it will receive the trapdoor and perform the search over either the index or the relevant partitions and gives each related document a score. Once the highest ranking documents are determined, the server can request to retrieve them from the cloud storage and send them back to the client.

In the S3BD system, the server also has the partitioner which fractures the central index into smaller shards. Once the client requests to search, it will load the client-picked shards into memory and perform the search over them instead of the central index.

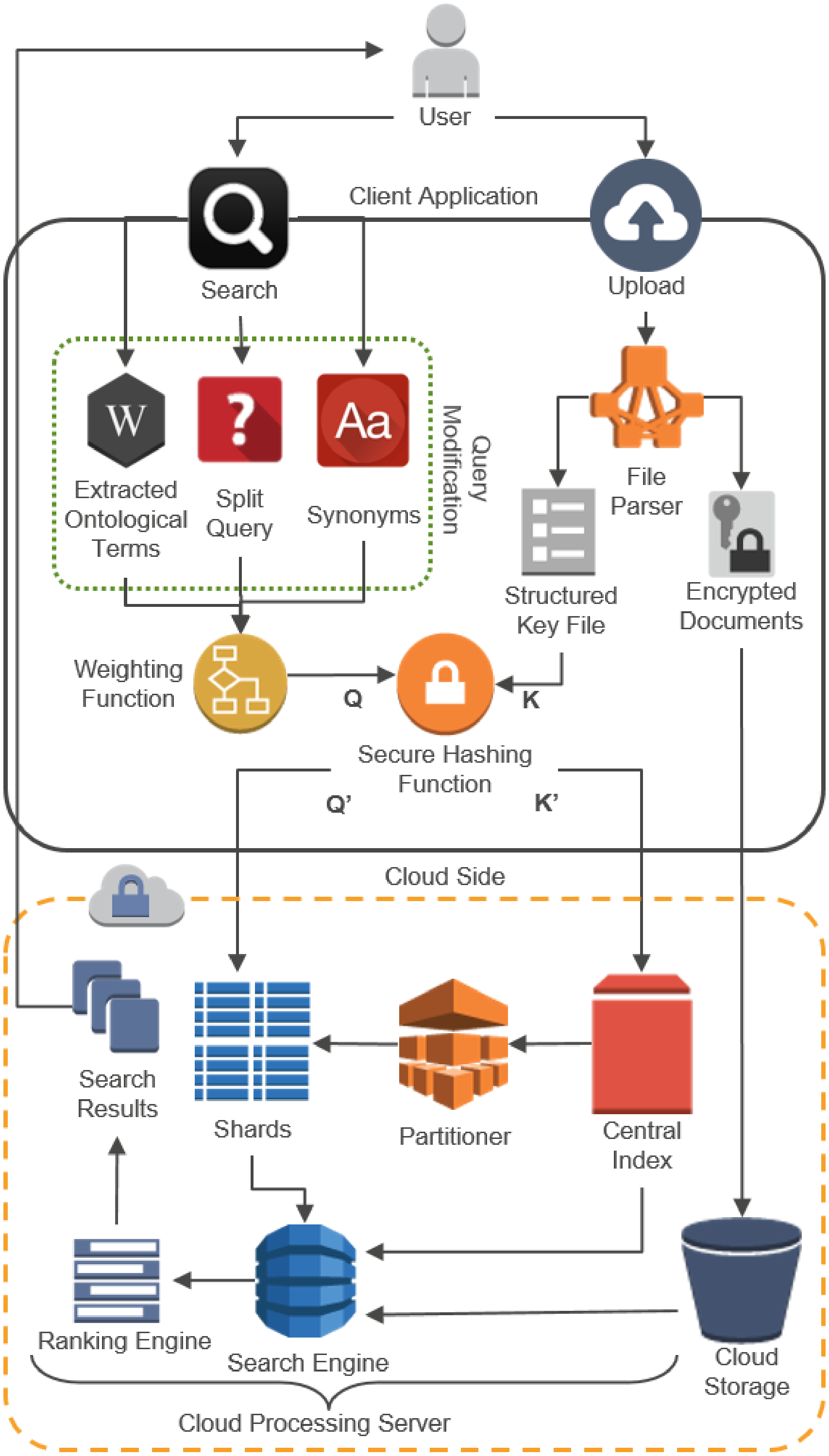
3.5. Cloud Storage

The cloud storage block is used to store the encrypted files that the user uploads. It will not see any representation of the user's query. The storage can potentially span multiple clouds, as long as the computing server knows where each document is stored and the index is updated accordingly.

3.6. Summary

The architecture of the systems is divided primarily into a cloud side and a client side. The client side is the only part of the architecture that is considered to be trusted with sensitive data. The following chapters explain in detail how these components come together to create a system that can semantically search over encrypted data in the cloud.

Figure 3.1. Overview of the system architecture and processes. Parts within the solid-line group indicate items or processes on the client side which are considered trusted. Parts in the dashed-line group indicate those in the cloud processing server. All components in the cloud are considered untrusted.



CHAPTER 4: SECURE SEMANTIC SEARCH OVER ENCRYPTED DATA IN THE CLOUD

The main idea behind the approach of S3C is to use a search method that is agnostic to the meaning of the terms in the documents or the query and only considers their occurrence and frequency, so that a search can be performed over the encrypted data which has lost all semantic meaning. For that purpose, we assure that each occurrence of a distinct word in every document is transformed into the same token, and consequently the same transformation is applied when the word appears in a search query. Doing this ensures that a match is still produced during the search process. Hashing is a good way to achieve this.

S3C has to main functionalities: *uploading* documents from the client, and *searching* over documents in the cloud. Uploading documents constructs a centralized index out of the transformed tokens, and the search function is able to perform a lookup on documents using similar tokens.

S3C considers three schemes of how documents are parsed and searches are performed. An overview of major functionalities and details of each scheme are explained in the following sections.

4.1. Overview of Upload and Search Processes

4.1.1. Upload Process

The goal of the upload process is to parse the desired document into indexable information and encrypt it before being sent to the cloud. In general, a subset of terms from the document (termed *keywords*) is selected to represent the semantics of that document. In addition, term frequency of the keywords within that document is gathered, then the terms are transformed individually into their hashed form and written to a temporary key file to be sent to the cloud along with the full encrypted text file.

Once the cloud processing server receives the encrypted document file and associated key file, it moves the encrypted document into cloud storage. Then the terms and frequencies in the key file will be added to the hashed index, which associates a hashed term with a list of documents it appeared in. The size of the uploaded document is also recorded with the index.

The system also supports batch uploading of many documents at once and processes them as a series of individual files with linear complexity.

4.1.2. Search Process

The search process consists of two main phases: *query modification* and *index searching and ranking*. Query modification is meant to inject semantic information into it and to match its terms to those in the cloud's central hashed index. This phase starts with the user entering a plaintext query into the client application. The query is then modified on the client side and then sent to the cloud processing server, where index searching and ranking is performed. The process of query modification takes the original query q and expands it into the modified query set Q. It involves three phases: query parsing, semantic expansion, and weighting.

The goal of parsing the query is to break q into smaller components. This is done because a mutli-phrase string hashes to a different value than the sum or concatenation of its parts, and once on the cloud, the terms must match the entries in the hashed index exactly. Once this phase is complete, Q will consist of q and its split parts.

In order to achieve semantic expansion, the system injects semantic data through the use of online ontological networks. The most naï approach to this is to perform a synonym lookup for each member of Q (termed Qi) through an online thesaurus and add the results to Q. This assures that the search results will include documents containing terms synonymous with, but not exactly matching, the user's query.

However, this approach alone does not cover ideas that are semantically related to the user's query, but are not synonymous. To achieve this, S3C pulls from moure advanced ontological networks. For example, in this research, the contents of Q are used to pull entries from Wikipedia, and keyphrase extraction is performed on them to get related terms and phrases (hereafter referred to as related terms). These related terms are then added to Q. The result of this is that the search can retrieve documents that contain concepts more abstractly related to the user's query (e.g. related diseases). In addition, the use of online resources relieves the client of the need to locally store semantic networks.

The goal of weighting is to ensure that the search results are more relevant to the user's original query than the synonyms and related terms. For example, a document that matches the entire original query should be weighted higher and considered more relevant than a document that only matches synonyms or individual parts of the query. To achieve this, we introduce the following weighting scheme with weights ranging from 0 to 1:

- The original query q is weighted as 1.
- Results of query parsing are weighted as $1/n$ where n is the number of terms derived from parsing.
- Synonyms or related terms of a term Q_i are weighted as $W(Q_i)/m$, where $W(Q_i)$ is the weight of Q_i and m is the number of synonyms or related terms derived from Q_i.

These weights are added to all members of Q to complete the modified query set.

Once the entirety of Q is built, its members are hashed to create the trapdoor Q' which is sent to the cloud to perform the index search and ranking. On the cloud processing server, the system goes through each member of Q' and checks them against the hashed index to compile a list of files that could be considered related to the query. These related files are further ranked using a modification of the Okapi BM25 (cite OKAPI) equation described in the following equations:

$$r(d_i, Q') = \sum_{i=1}^{n} IDF(Q'_i) \cdot \frac{f(Q'_i, d_i) \cdot (\alpha + 1)}{f(Q'_i, d_i) + \alpha \cdot (1 - \beta + \beta \cdot \frac{|d_i|}{\delta}} \cdot W(Q'_i) \tag{4.1}$$

IDF in this equation refers to the inverse document frequency for the term, which can be defined as:

$$IDF(Q'_i) = \log \frac{N - n(Q'_i) + 0.5}{n(Q'_i) + 0.5} \tag{4.2}$$

We define the terms in these equations as follows:

- Q_i - an individual term in the original plaintext query
- Q'_i - the hashed version of Q_i in the hashed query set.
- $r(d_i, Q')$ - the ranking score attributed to document d_i for hashed query set Q'
- $f(Q'_i, d_i)$ - the frequency of term Q_i in document d_i
- N - the total number of documents in the document collection C
- $n(Q'_i)$ - the total number of documents containing the query term Q_i
- $|d_i|$ - the length of document d_i in words
- δ - the average length of all documents in C

- $W(Q_i')$ - the weight associated with term Q_i
- α and β - constants (in this work we considered the values 1.2 and 0.75, respectively)

The cloud processing server computes this equation for all documents in the collection C against the query Q' and returns the list to the client, sorted by score in descending order.

4.2. Search Schemes

S3C considers three main schemes for how to implement the general approach. The primary differences among the proposed schemes are: how to select the subset of terms to represent the document, parse the user search query, and perform ranking.

4.2.1. Naïve Scheme: Full Keyword Semantic Search (FKSS)

FKSS follows the naïve method of selecting terms as keywords. It simply goes through the document and collects and counts the frequency of each individual word that is not considered a stopword. This gives the hashed index the full scope of the document, as no meaningful text is left out, but bloats it with possibly unneeded terms.

FKSS also follows a naïve method of parsing the query, as it just divides it into singular words. This is all that is necessary, as the keyword selection for the hashed index only considers single words. Thus parsing the query into larger groups of words would add no value.

Ranking for FKSS is performed with no modification to Equation (4.1).

Though FKSS follows a naïve approach, it can be useful for scenarios in which small documents are used or it is integral for the full text to be considered. For example, searching over encrypted media tags or social media updates. It is the least secure scheme, however, as it leaves the full scope of each document in the hashed index.

4.2.2. Space-Efficient, Fully Secure Scheme: Selected Keyphrase Semantic Search (SKSS)

SKSS creates a space-efficient index by running the document through a keyphrase extractor to obtain a constant number of the most important keywords and phrases within the document. These phrases can be considered to convey general information on what the document is about. Because they can contain more than one word, they are broken down into their individual distinct words, so that the key file sent to the server contains both hashed representations of the full phrase and each word within it. The use of a constant number of terms per document keeps storage overhead small and increases security, as much of the document is not put in the hashed index on the cloud.

In an effort to further increase security, term frequency is eliminated with the justification that each term is considered to be equally important to the meaning of the document, and thus can be considered equally frequent within the document.

To parse the query, SKSS divides not only into individual words, but into all possible adjacent subsets. An example of this can be seen in Figure 4.1. While some of the phrases added to the set might be meaningless (”`Failure Wireless Sensor`”, for example), others will carry meaning that will be important during the semantic lookup (”`Sensor Networks`”, for example). Once the parsing is complete, synonyms and related terms are looked up for all of the resulting phrases in the query set.

Figure 4.1. A sample of the query parsing done by SKSS.

”`Failure in Wireless Sensor Networks`”, ”`Failure Wireless Sensor`”, ”`Wireless Sensor Networks`”, ”`Failure Wireless`”, ”`Wireless Sensor`”, ”`Sensor Networks`”, ”`Failure`”, ”`Wireless`”, ”`Sensor`”, ”`Networks`”

When performing ranking, SKSS modifies Equation (4.1) to compensate for the lack of frequency data. Because the keyphrae extractor pulls a limited number of terms from the document, all extracted phrases are considered equally frequent. Thus, a 1 is put in place of $f(q_i, d_i)$.

4.2.3. Space-Efficient, Accuracy Driven Scheme: Keyphrase Search With Frequency (KSWF)

KSWF is a combination of the two previous schemes. The keyphrase extractor is still used to obtain keywords for the index, similar to SKSS, and the phrases are subsequently divided into their individual words. After this, it makes a second pass through the document to collect the frequency information for each word and phrase, similar to FKSS, which is stored alongside the terms in the index.

The user query is parsed in the same manner as SKSS, with each adjacent subset added to the overall query set. Because the frequency data is now present for all of the terms and phrases, it uses the same ranking method as FKSS. This scheme was developed primarily to analyze the impact of utilizing term frequency with a method like SKSS. Intuitively, adding term frequency should bring up more relevant search results, as there is more accurate data for the ranking.

The addition of frequency data to KSWF adds greater accuracy to the ranking function. For this reason, it is useful in scenarios in which the highest accuracy possible is desired while maintaining minimal storage overhead.

4.3. Security Analysis

S3C provides a trustworthy architecture for storing confidential information securely in clouds while maintaining the ability to search over them. The only trusted component of the architecture is the user machine, which has access to all sensitive information, such as the full plaintext documents and the document key files. Keeping the client machine trusted is a reasonable assumption in the real world, as it can be kept with minimal exposure to outside attackers.

Our threat model assumes that adversaries may intend to attack the communication streams between client and cloud processing server and between cloud processing server and cloud storage, as well as the cloud processing server and storage machines themselves. To explain what exactly the attacker could see or do, we will first introduce several definitions.

Definition 4.1. *History*: For a multi-phrase query q on a collection of documents C, a history H_q is defined as the tuple (C, q). In other words, this is a history of searches and interactions between client and cloud server.

Definition 4.2. *View*: The view is whatever the cloud can actually see during any given interaction between client and server. For our system, this includes the hashed index I over the collection C, the trapdoor of the search query terms (including its semantic expansion) Q', the number and length of the files, and the collection of encrypted documents C'. Let $V(H_q)$ be this view.

Definition 4.3. *Trace*: The trace is the precise information leaked about the history H_q. For S3C, this includes file identifiers of associated with the search results of the trapdoor Q'. It is our goal to allow the attacker to infer as little information about H_q as possible.

The view and trace encompass all that a potential attacker would be able to see. For the sake of this analysis, we will assume that the chosen encryption and hashing methods are secure, and so C' itself will not leak any information. I only shows a mapping of a single hashed term or phrase to a set of file identifiers with frequencies, meaning a distribution of hashes to files could be compiled, but minimal data could be gained from the construction. Similarly, Q' only shows a listing of hashed search terms with weights. The addition of the weights could potentially enable the attacker to infer which terms in the trapdoor were part of the original query, but they would still only have a smaller set of hashed terms.

However, we must consider the small possibility that, if the attacker was able to know the hash function used on the client side, they could in theory build a dictionary of all words in the vocabulary V that the documents are comprised of mapped to their hashed counterparts, and reconstruct I in plaintext. In this scenario, the attacker could put together the terms that the documents are comprised of, but since I carries no sense of term order, they could not reconstruct the entire file. The KSWF scheme adds additional security by only showing a small portion of the important terms and phrases from the document, meaning the attacker would only be able to ascertain how many times those specific terms and phrases were in the document. The SKSS scheme adds more security by removing those term frequencies.

An attacker monitoring the network channels during a search could see the resultant file identifiers that are associated with the given Q'. This would show an encrypted history as (C', Q'). However, since the attacker would not be able to discern the query (without the use of the above dictionary), this data would be of little use.

Attackers could also potentially attempt to alter data in C'. These attacks, however, could be recognized, as the client would not be able to decrypt them.

4.4. S3C Performance Evaluation

To evaluate the performance of S3C and provide proof of concept, it was tested with the Request For Comments (RFC) dataset, a set of documents containing technical notes about the Internet from various engineering groups. The dataset has a total size of 357 MB and is made up of 6,942 text files. To evaluate the system under Big data scale datasets, performance tests were run on a second dataset, the Common Crawl Corpus from AWS, a web crawl composed of over five billion web pages The system was evaluated against the RFC using three types of metrics: *Performance*, *Overhead*, and *Relevance.*

4.4.1. Metrics for Evaluation

Relevance

We define relevance as how closely the returned results meet user expectations. To evaluate the relevance of our schemes, we used the TREC-Style Average Precision (TSAP) method described by Mariappan et. al. in [MSB12]. This method is a modification of the precision-recall method commonly used for judging text retrieval systems. It is defined as follows:

$$Score = \frac{\sum_{i=0}^{N} r_i}{N} \tag{4.3}$$

Where i is the rank of the document determined by the system and N is the cutoff number (10 in this case, hence the term TSAP@10). r_i takes three different values:

- $r_i = 1/i$ if the document is highly relevant
- $r_i = 1/2i$ if the document is somewhat relevant
- $r_i = 0$ if the document is irrelevant

This allows for systems to be given a comparative score against other schemes in a relatively fast manner.

Performance

We define performance as the time it takes to perform the search operation. The aspects of performance measured are as follows:

- Time it takes to process the user query in seconds. This includes semantic query modification and hashing into the trapdoor.
- Time it takes to search over the index in the cloud in seconds. This includes retrieving the related files from the index and ranking them based on the query.
- Total time to perform the search in seconds. This encapsulates both of the steps above, plus any additional time taken with communication over the network.

Overhead

We define overhead to be the imposed cloud server storage space taken by the hashed index and the computing involved with it. The aspects of overhead we measure are as follows:

- Size of the inverted index, measured in the form of number of entries.
- Time it takes to construct the index in seconds. This operation reads the data files for the index and compiles them into a hash table. It is only performed on the cloud server startup.

4.4.2. Benchmarks

We derived a set of benchmark queries based on the information presented in the dataset. For testing relevance, we looked at two categories of queries which a user may desire to search. In the first category we consider a user who already knows which document they are looking for, but may not remember where the document is located in their cloud or may not want to look through a large number of files to find it. Such queries are typically specific and only a small number of documents should directly pertain to them. The search system is expected to bring up these most desired documents first.

In the second category we consider a user who wants to find all of the documents related to an idea, such as an officer trying to find all reports with similar crimes in our motivation. Such queries would be broad with many possible related documents, and the search system should bring up the most relevant ones first.

Figure 4.2. Queries used for testing relevance. Queries in category 1 target a small set of specific, known documents within the collection, while queries in category 2 target a broad set of documents not necessarily known to the user.

Category 1 - Specific:

`IBM Research Report (IRR)`, `Licklider Transmission Protocol (LTP)`, `Multicast Listener Discovery Protocol (MLDP)`

Category 2 - Broad:

`Internet Engineering (IE)`, `Transmission Control Protocol (TCP)`, `Cloud Computing (CC)`, `Encryption (EN)`

To measure performance, we measured time for a small (single word) query and a mid-size (three word) query. In addition, to measure the effects of expanding the size of the search query, we measured times for queries that expanded from one word to four words, taking measurements at each single word increment. Due to the inherent variety in the performance results, we report the mean and 95% confidence interval of 50 rounds of running each experiment.

For scalability tests, we measured search times and storage overhead for several three word queries against increasingly large portions of the dataset. Specifically, we tested against datasets of sizes: 500 MB, 1 GB, 5 GB, 10 GB, 25 GB, and 50 GB.

As a baseline for performance testing, we implemented a standard non-secure (SNSS) version of the system, utilizing the same semantic processing but with no encryption or hashing. Due to their similarities in indexing, the SNSS and FKSS schemes can be seen as being grouped together, as they both consider the entirety of the document text. Similarly, the SKSS and KSWF schemes can be grouped together since they both consider a small subset of the document text.

4.4.3. Evaluating Relevance

Figure 4.3. TSAP@10 score for the specified query for each system. Once the system has returned a ranked list of results, a score is computed based off of a predetermined relevance each file has to the given query.

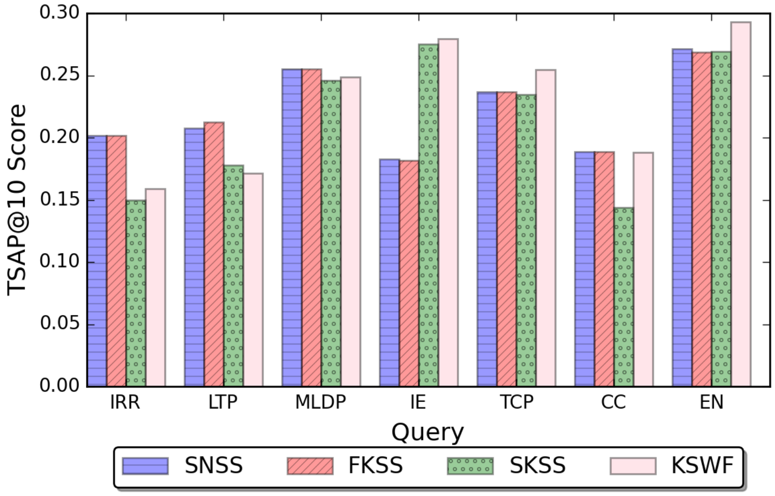


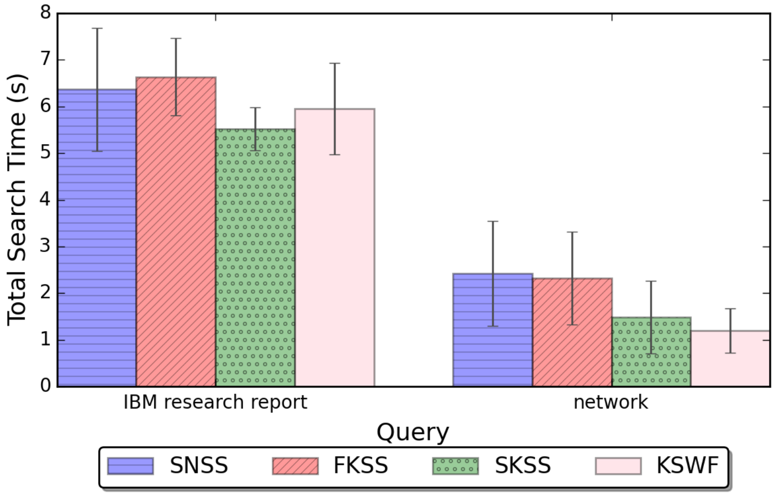
Figure 4.3 shows the TSAP scores of each of the four schemes searching with each

of the benchmark queries. For queries in category 1, the main desired results were ranked the highest for all schemes. The space-efficient schemes (the SKSS and KSWF), which might intuitively seem to suffer greatly in accuracy, only show to suffer a small amount when compared to the schemes that utilize the documents full text.

For queries in category 2, the SKSS and KSWF schemes showed to return just as relevant results, and in some cases were more relevant. Most interestingly, the KSWF scheme does not actually show much benefit from the addition of term frequency, meaning that when working with a small subset of the document's text, finding the frequency of those key phrases may be unnecessary due to causing a longer indexing time, unless the highest possible accuracy is desired.

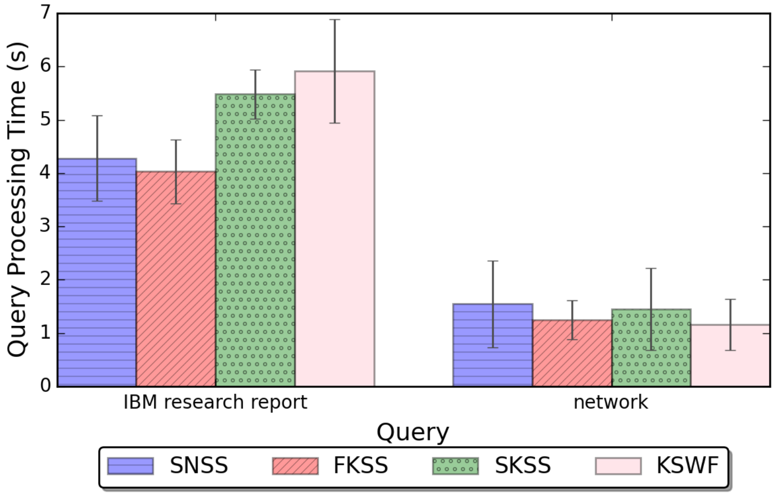
4.4.4. Evaluating Performance

Figure 4.4. Total search time in each scheme. This includes the time taken to process the query, communicate between client and server, and perform searching over the index. The results are averaged over 50 runs.



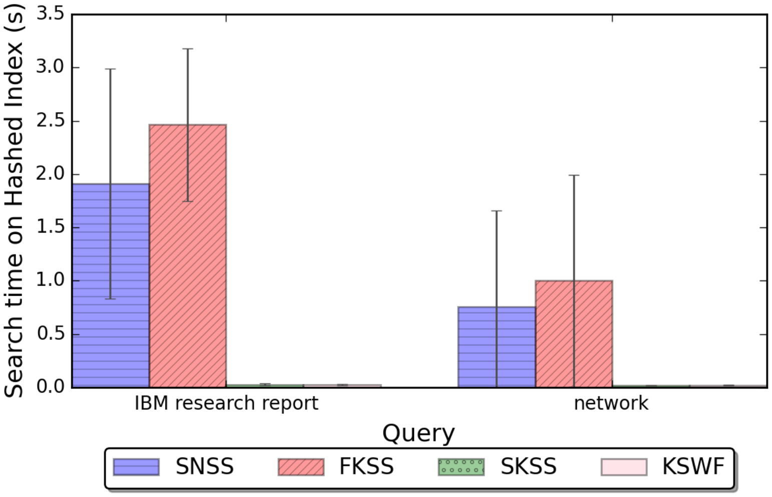
In the experiments, we measured the performance of each scheme with a small (one-word) and mid-sized (three-word) query, gathering the total time it takes to perform the search. In addition, we measured the two main components of the total search time: the time taken for query modification and the time taken to perform the index search and ranking on the cloud.

Figure 4.5. Time to process the query. This includes query modification and hashing into the trapdoor. The results are averaged over 50 runs.



Results can be seen in Figures 4.4, 4.5, and 4.6. All schemes can be seen to be reasonably similar in terms of total search time. The majority of search time across all models is taken up by the query processing phase, as our system needs to pull information from across the Internet in the form of synonyms and Wikipedia entry downloads. SKSS and KSWF both take slightly longer to process longer queries due to the addition of the adjacent query subsets which need to be looked up as well. Query processing time is thus linked to Internet speeds and the size of the Wikipedia entry for each of the query terms. The results indicate that under fast Internet speeds, the performance time of this system will naturally improve. While pulling information from the Internet does naturally increase search times, it was included intentionally to reduce storage size needed for the local client which would otherwise need to house an onboard ontological network.

Figure 4.6. Time it takes to perform the search on the hashed index on the cloud. This includes the time taken to find all files in the hashed index that contain any hashed terms in the query trapdoor and rank them with the scheme's respective functions. The results are averaged over 50 runs.



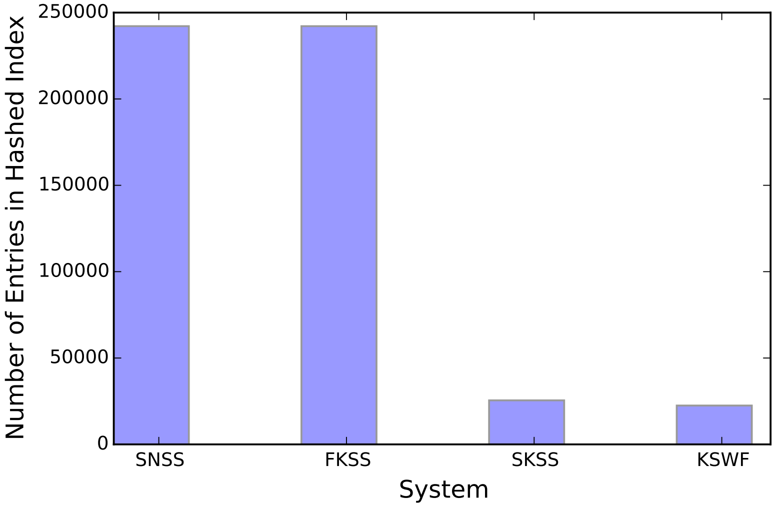
Most important to note is the difference in index searching times. The space-efficient SKSS and KSWF schemes take a near-negligible amount of time to search over the index. This can be explained by the vastly decreased index size (as shown in the next subsection) as only key phrases are stored, meaning that the initial set of potentially relevant documents is significantly smaller and the ranking equation must be run a lower number of times.

Because the greatest amount of time is taken during query processing and index search time is very small for the space-efficient schemes, these two schemes can be scaled up to work on larger datasets without facing a huge growth in search time.

4.4.5. Evaluating Overhead

To demonstrate space-efficiency in this evaluation, we measured the overhead for each scheme in terms of how many entries were stored in the hashed index. These results can be seen in figures 4.7 and 4.8. The two groups of schemes show a vast difference in this regard, due to the number of terms selected from each document. The linear growth per document of the index guaranteed by the constant number of key phrases extracted keeps the index small while maintaining the relevance of search results (as shown previously).

Figure 4.7. Size of the inverted index for each system. An entry denotes a hashed keyword mapped to a set of file identifiers.

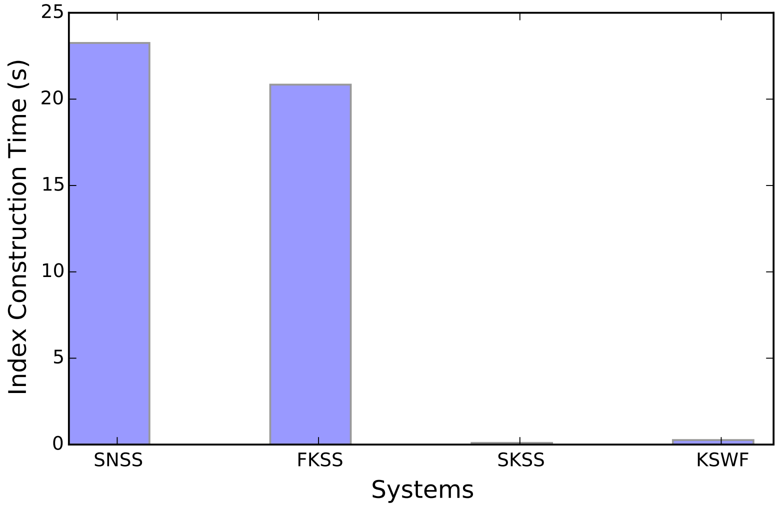


In addition, we measured the effect that the size of the inverted index had on the time it takes to construct the index from the utility files on the server. The differences are again vast, with construction times being almost negligible for SKSS and KSWF. It is worth noting that this operation needs only to be performed at startup of the cloud server, and that additions to the index at runtime operate at near constant time regardless of the size of the dataset due to the hash table structure of the index.

4.4.6. Evaluating the Impact of Query Length

In addition to measuring search times for individual queries, we are interested in measuring the effect on performance of expanding the size of a single query. For example,

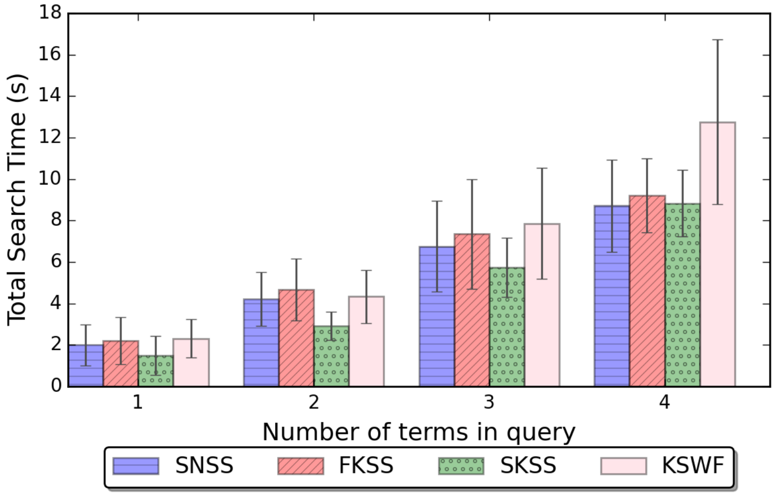
Figure 4.8. Time it takes to construct the hashed index upon server startup. This operation includes sequentially reading an index file hosted on the cloud server which contains all data for the inverted index and document sizes table and storing it in hash tables.



one query used in this experiment started as `protocol` which expanded to `transmission protocol` which further expanded to `transmission control protocol` which finally expanded to `network transmission control protocol`. Figure 4.9 shows the results of this experiment, with queries grouped by the number of meaningful terms in them (query length minus stopwords) in the horizontal axis.

In these results, the time it takes to search (vertical axis) can be seen to be linearly related to the number of meaningful terms in the query. This is because the majority of search time is taken up by the query processing phase, which grows with the number of terms in the query there are to be processed. The SKSS and KSWF schemes can be shown to have a faster growth due to the greater amount of query processing necessary as the query expands. Interestingly, SKSS consistently performs as well or better than the others despite the additional query processing. This is due to its small index size and lack of frequency data collection.

Figure 4.9. The total search time to for an expanding query. This includes the time to process and modify the search query, communicate between client and server, and rank in the cloud. The horizontal axis shows the number of words (minus stopwords) in the query. The results were averaged over 50 runs.



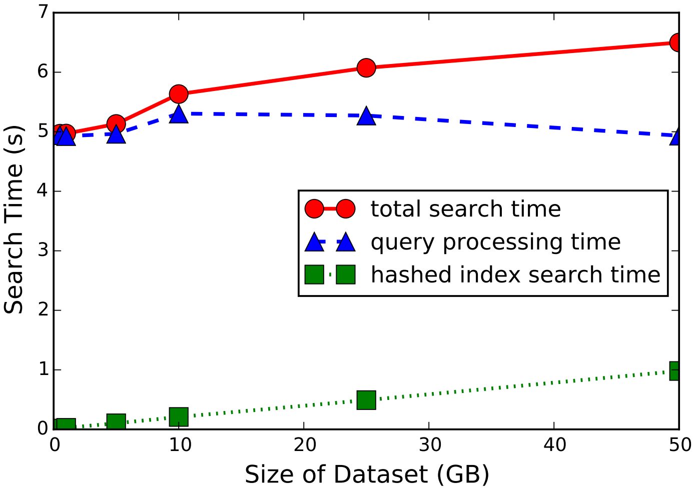
4.4.7. Evaluating Scalability

To test the scalability of our system, we ran searches against an increasingly large set of data. For simplicity, evaluations of this were only performed using our most space-efficient scheme, SKSS. Figure 4.10 shows the results of this evaluation.

These results show that as the size of the dataset increases, the time taken for query modification remains relatively constant, while the time spent searching the hashed index on the cloud increases linearly. As a result, the total search time increases by only 30.8% as the dataset increases from 500 MB to 50 GB.

In addition, to show the low overhead provided by the system, we measured the size of the index at each size increase during our test. The results are shown in figure 4.11. We conclude that though the relation between dataset size and index size is linear, the slope is as low as 0.003. The index size always remains at ∼0.3% of the size of the dataset.

Figure 4.10. Time taken to search for different dataset sizes. Resulting times are the mean of 50 runs performed with multiple three word queries. The dotted line shows the time taken to search on the hashed index in the cloud, the dashed line shows the time taken for query modification, and the solid line shows total time taken for the search (including query modification and index searching).

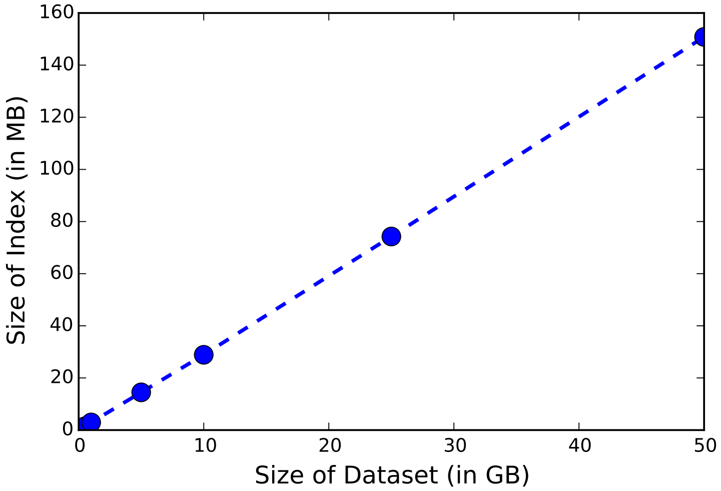


4.5. Discussion

The techniques used in S3C improve upon existing encrypted data search techniques by providing a solution that is space-efficient on both the cloud and client sides, considers the semantic meaning of the user's query, and returns a list of documents accurately ranked by their similarity to the query. Further, the semantics are achieved without the need for a highly specific semantic network to be built and maintained by the client.

Experiments with a working prototype of each of the presented schemes show that S3C is accurate and gives reasonable performance with low overhead. We argue that each of our schemes could be tuned to certain use cases. SKSS could be used for documents with a mid-sized amount of encrypted text where the key phrase extraction can capture the meaning of the document well, providing a very low overhead solution; for example,

Figure 4.11. Size of the index file for different dataset sizes. The horizontal axis plots the size of the dataset used in gigabytes, while the vertical axis plots the associated index size in megabytes.



police records with encrypted officer notes. KSWF could be used in similar cases in which the slight raise in accuracy is considered worth the slight decrease in performance and security. FKSS could be used for small documents where the whole of the text is considered important; examples including twitter updates or media tags. In addition, experiments showed that, due to low overhead, SKSS and KSWF schemes can be utilized for searching larger scale datasets.

However, the work system on the building of a single central hashed index to be searched. Despite the small size of the index with the given datasets, using larger big data scale datasets will push search times past what would be considered acceptable. While it is possible to break the index into even partitions and parallelize the search process, much of the large index will be irrelevant to the search, leading to much wasted computational time. The following chapter expands upon S3C and provides a solution for this problem.

CHAPTER 5: SECURE SEMANTIC SEARCH OVER ENCRYPTED BIG DATA IN THE CLOUD

A considerable portion of computational resources and time is wasted when searching through the entirety of a large central index. Common practice is to reduce the impact of this large index by partitioning it into equal disjoint shards, which facilitates parallelization through multiple threads or computers. While this reduces the search time, it can require a large computing infrastructure while still wasting computational resources, as much of the index will be irrelevant for any given search.

Another technique for reducing the impact of a large index is to partition it into topic-based shards, then determine at search time which shards are relevant to the query and only search over those. This substantially reduces the resources used per searching, but the topic-based partitioning is difficult to perform on encrypted data due to a lack of semantic information. S3BD achieves this using the clustering hypothesis, stating that pieces of data that appear together frequently are likely related. The result is ultimately that the search is *pruned*.

S3BD follows much of the same approach as S3C, but with the added process of partitioning the index after substantial data has been added to the system. Thus, the main procedures of the system can be considered to be the upload, partitioning, and search processes. This chapter describes these procedures as they relate to S3BD.

5.1. Overview

The system starts with the uploading of documents. The upload process remains much the same as that of KSWF in S3C. Documents are parsed with a key phrase extractor to select 10 phrases to represent the document. They are split into their individual words and their frequency data is collected. The document is then probabilistically encrypted while the key phrases are secured using deterministic encryption and compiled into the central index. This encryption method is needed to allow the key phrases to be matched in the index during the search process while still being able to be decrypted on the client's machine. Ability to perform decryption will be necessary for the search process to determine which clusters to search over.

Once the central index is compiled it can be partitioned into topic-based shards, which is accomplished using the K-Means clustering algorithm. K-Means allows data to be clustered as long as a distance can be defined between two data points. To that end, we consider that two main operations need to be defined to perform K-Means: picking starting data points for clustering (creating initial centroids) and computing distance between two data points.

In order to search over the most relevant shards, we consider that the system should be able to compare the query to the shards semantically. Because this would be impossible on the cloud processing server due to all data having lost semantic meaning, the shards must be picked on the client's machine. Thus, the client must have some information about the shards. To that end, S3BD abstracts the data from the shards into smaller samples to send to the client, where it can be decrypted back into plaintext and compared to the query to determine which shards to search.

The client notifies the cloud processing server of the shards to be searched over so that it can load those into memory. After this, the search is carried out much in the same way as KSWF. The query is expanded with semantic information and is encrypted with the same key as the key phrases so it can be matched to the data in the index shards.

5.2. Partitioning Process

5.2.1. Initializing Centroids

The first step to partitioning the central index into shards with the K-Means clustering algorithm is to pick initial "centers" (i.e. centroids) of each shard to form shards around. These centroids are terms and their associated lists of files and frequencies picked from the central index, and should ideally represent diverse sections of the data set. The initialization of centroids has a large impact on the resulting shards. For effective search pruning, the shards should be distributed as evenly as possible while maintaining semantic relationship, so that the amount of the index that is pruned is not heavily varied depending on the clusters chosen at search time.

A naïve method for initializing centroids is to simply pick a number of centroids equal to the number of desired shards (k) randomly from the central index. This method can potentially result in terms with very few associated files being chosen as centroids. Because the distance from any term to a centroid is a factor of their co-occurrence in files, this would lead to very small shards being formed. To that end, ensuring that centroids have enough associated files to "attract" other data points is prioritized.

Another method is to sort the terms in the central index by the number of associated files, and choose the top k terms as centroids. This ensures that no small terms are chosen, but the centroids can potentially have a high overlap of associated files, leading again to uneven distribution in the shards. To that end, ensuring diversity among centroids is prioritized.

A final method is to sort the terms and pick terms only if they are not associated with many terms associated with previously chosen centroids. Specifically, a set of all files associated with all centroids will be maintained and added to whenever a centroid is chosen. In order to be picked as a centroid, a term must be able to contribute more unique files to this set than repeated ones. The result is that centroids are guaranteed to be associated with enough files to attract other terms, and will be diverse enough to attract different groups of terms.

5.2.2. Computing Distance

Once k centroids have been chosen, the system calculates the distance between the centroids and all terms in the index. With plaintext data this would be possible using a semantic graph to calculate the relatedness of two terms, but the use of encrypted data makes this impossible. The clustering hypothesis states that data that appears together can be considered related; to that end we consider the co-occurrence of terms in a file is a reasonable metric for their similarity. In other words, if two terms or phrases appear in the same file, they could be considered related.

To compute the distance between a term and centroid, the system counts through each file associated with the term. For each file, it considers how often that term appears in that file compared to how frequently the term and centroid appear in that file (i.e. a co-occurrence factor). The formal equation is defined below:

$$dist(C_i, T) = \sum_{f \in I[T]} \frac{c(f, I[T])}{|I[T]|} \cdot \log \frac{\frac{c(f,I[T]}{|I[T]|}}{\frac{(c(f,I[T])+c(f,I[C_i])}{(|I[T]|+|I[C_i]|)}} \tag{5.1}$$

We define the terms in this equation as follows:

- $dist(C_i, T)$ - The distance between the centroid C_i and index term T. This is a metric for similarity.

- $I[T]$ The list of files associated with term T in the central index.
- $|I[T]|$ - The total frequency count of term T; the total number of times it appears in the dataset.
- $c(f, I[T])$ - The count of how many times term T appears in the file f.

5.2.3. Shard Distribution

Once the similarity between each centroid and term has been computed, terms can be distributed to their proper shards. An initial method for distribution was to assign each term to the shard with the centroid it was most similar to. This method, however, lead to uneven shard distribution, which does not make for effective search pruning.

A more effective method involves limiting the growth of each shard so that it can only hold up to a certain amount of its closest terms. Each shard's growth is limited to $\frac{2k}{|I|}$ of its closest terms, in which $|I|$ is the total number of terms in the central index. If a shard reaches this threshold, it disassociates the term furthest from its centroid, and the system calculates the next closest shard that has not yet reached this threshold.

Once the terms have been distributed amongst the shards, the system will calculate an average of the terms in each shard to create a new centroid, and the process of shard distribution will repeat, which aids in evening out the shard sizes. Ideally the process would repeat until the makeup of the shards no longer change after a repetition, but practically the shards only change minimally after the fifth repetition.

5.2.4. Shard Abstraction

Because the shards on the cloud processing server will consist only of encrypted data, it is impractical to identify on the cloud which shards to search over for each query. To that end, the we consider the idea of abstracting the shards into a much smaller amount of data to be sent back to the client and decrypted. This data serves as a sample which the search query can be queried against to determine which shards are most likely to contain relevant terms and files. These samples (termed abstracts) are made up of terms that can be compared to the query on the client's machine.

A naïve approach to forming the abstracts is to pick a number of terms randomly from the shard. However, this does not guarantee that these terms are indicative of the general topic of the shard. The list may contain too many terms that have a high distance from the shard's centroid.

We consider that the centroids for each shard are at the center of each shard, and thus most likely to relate to the general topic of the shard. Because each centroid contains a list of files, and each file is associated with a small list of terms and key phrases, the system chooses terms from those files to make up that shard's abstract. Specifically, it chooses the most frequent term from that file.

The abstract, now a list of terms related to the topics in the shard, is sent to the client to be decrypted and used at search time. The following section provides details how the search is performed.

5.3. Search Process

The search process of S3BD follows a similar pattern to that of S3C. The system first compares the query to the abstracts to determine which shards to search in the cloud. The client application will then semantically expand and encrypt the search query and send it to the cloud. Because the shards are formatted in the same manner as the central index, the cloud can perform the search in the same way as S3C.

5.3.1. Abstract Comparison

We consider that the terms in the abstracts are semantically linked to the topics in shards. In turn, we consider that comparing the query to the abstracts will let the system detect most accurately which shards are appropriate to search. This comparison is done by comparing the terms in the query to the terms in the abstracts using the WuPalmer word similarity metric. WuPalmer computes the semantic similarity between two words by evaluating the distance from one word to the other in a large semantic graph.

Using WuPalmer and normalizing results for the query against all abstracts allows the system to rank the abstracts by relevance to the query, and by extension, the shards. Once a sufficient number of shards have been chosen through this method, the cloud processing server is notified and it begins loading those shards into memory. Because the central index is not needed at search time, the cloud processing server can use a low amount of memory by only keeping relevant shards in memory at a time.

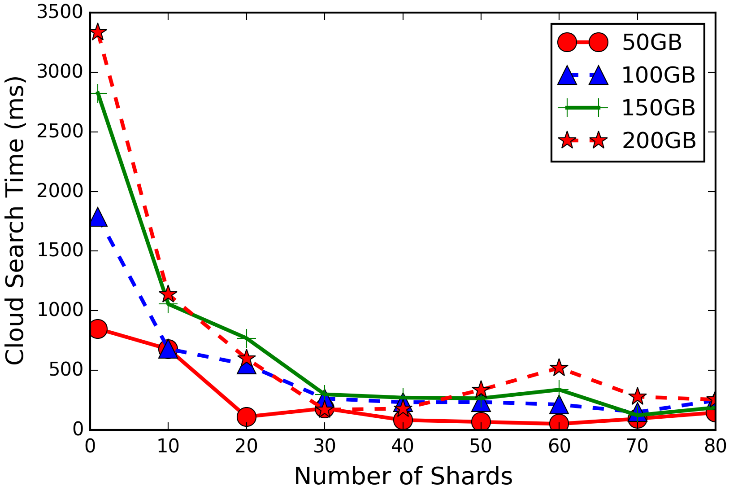
5.4. Evaluation

To evaluate the performance of S3BD, we tested it utilizing portions of the Common Crawl Corpus dataset from AWS. We evaluated performance by analyzing the time to search over shards in the cloud for various dataset sizes using various numbers of shards formed.

5.4.1. Evaluating Performance

Experiments were performed using 10 benchmark search queries, each query being three words long. We measured the time taken to search on the cloud with various numbers of shards created to analyze the direct effect of partitioning the index. Additionally, we analyzed the size of the shards chosen to be searched over. Time taken to expand the query was not analyzed, as it is assumed to be the same that of S3C.

Figure 5.1. Time taken to search over the chosen shards on the cloud. This includes time taken to find files in the shard that match the query trapdoor and rank them. Results are the average of 200 searches spanning 10 different queries consisting of 3 words.

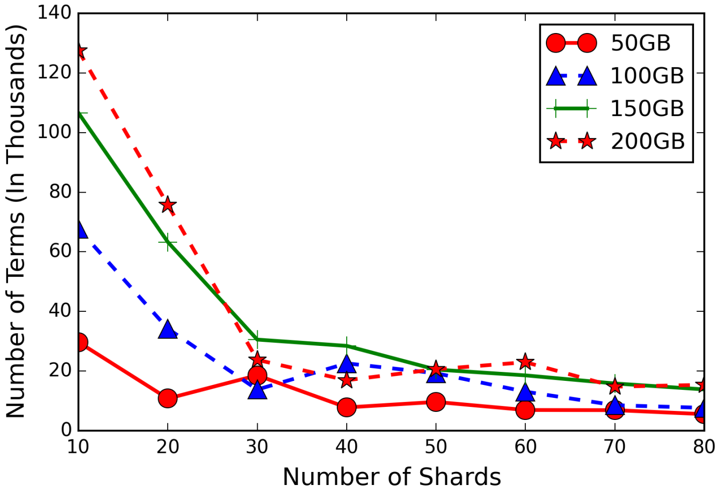


Results can be seen in figure 5.1. Time to search is seen to substantially decrease as more shards are formed. Important to note is that negative changes in search time cease being substantial for 100, 150, and 200 GB past 30 shards. This implies that partitioning into only 30 shards is enough to drastically reduce search times.

Interestingly, the search time does not strictly decrease as more shards are formed. This can be seen primarily in the spike in time for the 150 and 200 GB datasets at 60 shards. This is due to uneven shard distribution; the size of a shard plays a large role in determining how much time it contributes to search time. The system may determine that it needs to search a query among the smallest shards when there are fewer shards created, and search among the largest shards when there are more, despite the fact that the average shard size is becoming smaller as more shards are created.

Figure 5.2 shows the average size of a shard that is search for its respective dataset size and number of shards created. It shows that when there is a sharp turn in number of terms in the searched shards, it is roughly reflected in the time spent searching over them. That this is not always the case, however, suggests that there may be other factors affecting the search time over a shard as well.

Figure 5.2. The average number of terms in a single shard being searched over (i.e. the size of the shard). Each result is the average of the size of 3 chosen shards from 10 benchmark queries.



5.5. Discussion

The techniques used in S3BD adapt clustering methods used in others to enable topic-based partitioning of a central index on encrypted data. Using this approach in combination with those presented with S3C makes the system significantly more scalable, enabling it to be used on big data. A working prototype shows that the system can be utilized in real time to search over many encrypted documents that remain secure and untouched in cloud storage during the search process. Determining which shards must be searched over at search time allows the system to load into memory only those parts of the index which are necessary, saving on memory use.

While the addition of topic-based partitioning enhances the scalability and capabilities of the system, there are ways in which it can improve. Chapter 7 concludes the thesis and gives ideas for how a system for semantically searching over encrypted big data could be further expanded.

CHAPTER 6: IMPLEMENTATION

All aspects of both S3C and S3BD were implemented for purposes of testing and proof of concept. A simple command Line interface was created for both, while a full web interface was created for S3C. This chapter gives details on how the systems were implemented.

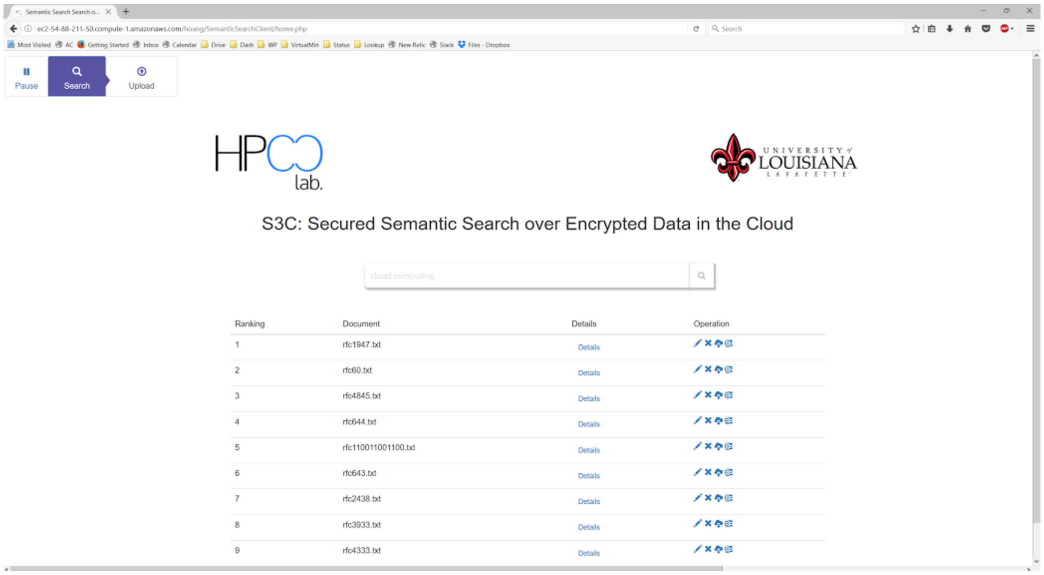
6.1. Command Line Implementation

A simple command line implementation was created using Java 1.8. A separate process was created for both the cloud processing server and the client application. Both were run on AWS EC2 instances to demonstrate cloud feasibility. Once the server application is started, a user only needs to interface with the client application. A user can choose to upload a batch of files, search over the files, or, if using S3BD, start to partition the central index.

The central index is implemented as a hashmap, mapping a string (the encrypted term) to another hashmap. The second hashmap maps a string (the file identifier) to a number (the frequency of the term in the file). In this way, access to the information for any term in the index is always constant. Shards each contain a portion of the index that is implemented in the same manner. When a search is performed, the server receives the encrypted query and polls against the index or shards to retrieve the required information.

Practically, it is not possible to partition the index every time a single file is uploaded, as the process can take hours for larger datasets. As a result, the shards can potentially become “out of date”. To ensure that a search will not exclude newly added files, a special shard is created for these files. This shard is included in the search every time until the partitioning process is rerun.

Figure 6.1. A screenshot of the web interface for an S3C client. For each file returned in a search, the client is able to edit, remove, and download the file.



S3C: Secured Semantic Search over Encrypted Data in the Cloud

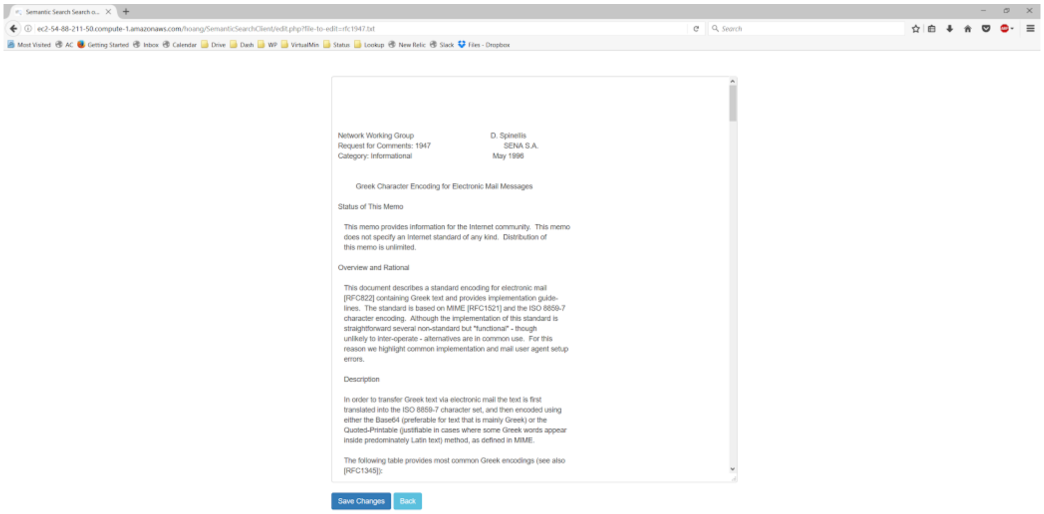
Ranking	Document	Details	Operation
1	rfc1947.txt	Details	✎ ✖ ☁ ▤
2	rfc60.txt	Details	✎ ✖ ☁ ▤
3	rfc4845.txt	Details	✎ ✖ ☁ ▤
4	rfc644.txt	Details	✎ ✖ ☁ ▤
5	rfc110011001100.txt	Details	✎ ✖ ☁ ▤
6	rfc643.txt	Details	✎ ✖ ☁ ▤
7	rfc2438.txt	Details	✎ ✖ ☁ ▤
8	rfc3933.txt	Details	✎ ✖ ☁ ▤
9	rfc4333.txt	Details	✎ ✖ ☁ ▤

6.2. Web Interface

The web interface is hosted in an AWS EC2 instance using PHP/HTML/CSS/Javascript and Bootstrap for the front- and back-end; Java 1.8 is used as the running environment. The web interface has two parts: server side and client side. On the server-side, a jar executable is constantly run. The server-side will listen to different requests on different ports, with each listener being processed by different threads. Three ports listen for three main operations to be sent from the user: upload, search, and remove.

The upload listener receives a path name of the folder to upload as parameter. After that, it will run to upload all the file in the folder of the given path and clean up that folder when completes uploading. If any error happens, it would notify. The search listener receives the query as a list of strings, and returns the result as a list of strings, which is parsed into a cleanly formatted table. The remove listener receives the name of the file needed to remove and removes the file from the index and storage.

Figure 6.2. A screenshot of the interface used to edit a document.


Semantic Search Search o...
ec2-54-88-211-50.compute-1.amazonaws.com/hoang/SemanticSearchClient/edit.php?file-to-edit=rfc1947.txt
Search
Most Visited
AC
Getting Started
Inbox
Calendar
Drive
Dash
WP
VirtualMin
Status
Lookup
New Relic
Slack
Files - Dropbox
Save Changes
Back

```
Network Working Group                    D. Spinellis
Request for Comments: 1947                  SENA S.A.
Category: Informational                      May 1996


     Greek Character Encoding for Electronic Mail Messages

Status of This Memo

  This memo provides information for the Internet community.  This memo
  does not specify an Internet standard of any kind.  Distribution of
  this memo is unlimited.

Overview and Rational

  This document describes a standard encoding for electronic mail
  [RFC822] containing Greek text and provides implementation guide-
  lines.  The standard is based on MIME [RFC1521] and the ISO 8859-7
  character encoding.  Although the implementation of this standard is
  straightforward several non-standard but "functional" - though
  unlikely to inter-operate - alternatives are in common use.  For this
  reason we highlight common implementation and mail user agent setup
  errors.

  Description

  In order to transfer Greek text via electronic mail the text is first
  translated into the ISO 8859-7 character set, and then encoded using
  either the Base64 (preferable for text that is mainly Greek) or the
  Quoted-Printable (justifiable in cases where some Greek words appear
  inside predominately Latin text) method, as defined in MIME.

  The following table provides most common Greek encodings (see also
  [RFC1345]):
```

The user's interface for performing a search can be seen in figure 6.1. The client is able to perform basic interactions with the search results: remove, download, and edit. When downloading a file, the cloud system sends the encrypted file through the network, and the client machine uses a local encryption key to decrypt it. Due to the complications with editing encrypted files directly, to edit a file the system downloads and decrypts it and displays it in an editable text window in the browser (seen in figure 6.2). Once the user has finished making edits, the system performs a standard upload on the new file.

CHAPTER 7: CONCLUSION AND FUTURE WORKS

This chapter summarizes the research and findings of this thesis. Additionally, we discuss further research topics that emerged during the course of this research but were not discussed in this thesis.

7.1. Discussion

This thesis began focusing on finding a way to perform a semantic search over encrypted data hosted in the cloud.

We presented the feasibility of a system for achieving the research goals through two main techniques. First, the system parsed uploaded documents, extracting key phrases and terms and hashing them to build a centralized index that is kept separate from the encrypted documents themselves. Second, the system injected a user's query with semantic information at search time using online resources before hashing them to be sent to the server.

Evaluation of this system showed that search results remained accurate while diminishing spatial and performance overhead. However, exploration of this system showed that it was not fully scalable for big data scale datasets. Based on that, we recognized the potential benefits of fracturing the central index to prune the work done by a single search.

Further exploration found that the index could be fractured into topic-based shards, which could be examined at search time to determine which shards would be necessary for the search. This was done through adapting a clustering algorithm, enabling it to partition the index into statistically separate shards. These shards are abstracted and sent to the client's machine, and the abstracts are checked at search time to determine which of them need to be searched by comparing terms in the query and abstracts with a semantic similarity metric.

By only loading into memory the shards that are required for search, the system lowers requirements for memory and cuts down on search time by eliminating irrelevant parts of the index. Our experimental evaluation shows that the this partitioning reduces search time significantly. The practicality of this system was shown by developing a working prototype, which was used for the evaluations.

7.2. Future Works

The focus of this thesis was on developing a fast method for semantic search over encrypted data. There are several points where the work could be expanded upon that were not covered in this thesis.

7.2.1. Search-Level Access

The index and shards are kept separate, and the encrypted uploaded documents are not directly used in the search process. As a result, it is possible to restrict access to the documents for certain users, providing what can be called “search-level” access: the ability to search over the dataset, but not read the results. This would be valuable to a data owner who has many potential users that they wish to be able to see only if data exists, but not necessarily view the data. For example, the law enforcement agency could wish to obscure the full data of their police records to certain officers, while still allowing them to see that they exist on certain individuals.

7.2.2. Text Summarization for Search-Level Access

In addition to providing search-level access, text summarization tools could be run on documents when they are uploaded, and processed using a different encryption key to provide access to some information for those restricted users. This would be valuable for datasets in which the titles of the documents do not provide enough information to be usable, and the dataset is too large to manually provide summary text.

7.2.3. Dynamic Shard Creation

Currently, the system requires a cloud server operator to specify a number of shards to generate based on previous knowledge of the behavior of the system and what would be best for a user's current dataset. This is standard for a system using the k-means clustering algorithm. However, the system can potentially suggest its own number and adjust this dynamically during the clustering process.

By knowing a size limit for each shard, based on the index size, the system could split shards that are growing too large and merge part of them with smaller shards or start to grow new a new shard. This could aid in evening the shard distribution, which would ultimately provide more stable searching times.

7.2.4. Dynamic Pruning Aggression

Currently, the system requires an operator to manually specify how many shards should be searched over. This number determines how "aggressively" the system prunes the search, or how much of the dataset is included in the search. However, not every search query has the same number of relevant files in the dataset; some may require more or less of it to be searched. By attributing a relevance score to each abstract on a normalized scale from 0 to 1, we can potentially automatically determine how many shards should be searched by examining the relation among these scores. This would allow for more data to be searched over if there exists more data relevant to the user's query, thus raising accuracy.

7.2.5. Dynamic Keyphrase Extraction

Currently, the system extracts 10 key phrases from each uploaded document. This method was designed for a somewhat homogeneous dataset, in which documents were of similar lengths. A more big data-tolerant approach would be to dynamically choose the number of key phrases extracted based on the size of the document. A decision on how to do this would have to be done through various rounds of testing, as it would heavily affect the accuracy and spatial overhead of the system.

7.2.6. Caching Abstract-Query Similarities

Currently, when the client machine chooses which shards should be searched, it compares the terms in the query to the terms in the abstract using a semantic similarity metric. This takes a lot of processing time, as the algorithm needs to traverse a large graph for each comparison. Considering that certain queries may be searched frequently, the client could maintain a cache of the results from these comparisons for frequently searched terms. Doing so would improve client-side performance and allow the server more time to load desired shards.

BIBLIOGRAPHY

[AMBR15] A. Andrejev, D. Misev, P. Baumann, and T. Risch, *Spatio-temporal gridded data processing on the semantic web*, Proceedings of the 2015 IEEE International Conference on Data Science and Data Intensive Systems, Dec. 2015, pp. 38–45.

[BDCOP04] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano, *Public key encryption with keyword search*, pp. 506–522, Springer, 2004.

[BDH03] L. A. Barroso, J. Dean, and U. Holzle, *Web search for a planet: The google cluster architecture*, IEEE Micro **23** (2003), no. 2, 22–28.

[Bro15] Chad Brooks, *Businesses still wary of cloud technologies*, 2015.

[BYMH09] Ricardo Baeza-Yates, Vanessa Murdock, and Claudia Hauff, *Efficiency trade-offs in two-tier web search systems*, Proceedings of the 32Nd International ACM SIGIR Conference on Research and Development in Information Retrieval (New York, NY, USA), SIGIR '09, ACM, 2009, pp. 163–170.

[CGKO11] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky, *Searchable symmetric encryption: improved definitions and efficient constructions*, Journal of Computer Security **19** (2011), no. 5, 895–934.

[FSL15] Reza Fathi, Mohsen Amini Salehi, and Ernst L. Leiss, *User-friendly and secure architecture for authentication of cloud services*, Proceedings of the 8th International Conference on Cloud Computing (IEEE CLoud '15), 2015.

[G^{+}03] Eu-Jin Goh et al., *Secure indexes.*, Cryptology ePrint Archive (2003), 216.

[GLBG99] Eric J. Glover, Steve Lawrence, William P. Birmingham, and C. Lee Giles, *Architecture of a metasearch engine that supports user information needs*, Proceedings of the 8th International Conference on Information and Knowledge Management, Nov. 1999, pp. 210–216.

[GMM03] R. Guha, Rob McCool, and Eric Miller, *Semantic search*, Proceedings of the 12th International Conference on World Wide Web, WWW '03, May 2003, pp. 700–709.

[HH00] Jeff Heflin and James Hendler, *Searching the web with SHOE*, Proceedings of the 17th Association for the Advancement of Artificial Intelligence Workshop on AI for Web Search, AAAI '00, July 2000, pp. 35–40.

[KAC^{+}02] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl, *RQL: A declarative query language for RDF*, Proceedings of the 11th International Conference on World Wide Web, May 2002, pp. 592–603.

[KC10] Anagha Kulkarni and Jamie Callan, *Topic-based index partitions for efficient and effective selective search*, Proceedings of the 8th Workshop on Large-Scale Distributed Systems for Information Retrieval, ACM, 2010, pp. 19–24.

[LC04] Xiaoyong Liu and W. Bruce Croft, *Cluster-based retrieval using language models*, Proceedings of the 27th International ACM SIGIR Conference on Research and Development in Information Retrieval (New York, NY, USA), SIGIR '04, ACM, 2004, pp. 186–193.

[LUM06] Yuangui Lei, Victoria Uren, and Enrico Motta, *Semsearch: A search engine for the semantic web*, Proceedings of the 15th international conference on Managing Knowledge in a World of Networks, Springer, Oct. 2006, pp. 238–245.

[LWW$^+$10] J. Li, Q. Wang, C. Wang, N. Cao, K. Ren, and W. Lou, *Fuzzy keyword search over encrypted data in cloud computing*, Proceedings of the 29th IEEE International Conference on Computer Communications, INFOCOM '10, Mar. 2010, pp. 1–5.

[Man07] Christoph Mangold, *A survey and classification of semantic search approaches*, International Journal of Metadata, Semantics and Ontologies **2** (2007), no. 1, 23–34.

[MSB12] A. K. Mariappan, R. M. Suresh, and V. Subbiah Bharathi, *A comparative study on the effectiveness of semantic search engine over keyword search engine using tsap measure*, International Journal of Computer Applications EGovernance and Cloud Computing Services (2012), 4–6.

[MSCBC13] T. Moataz, A. Shikfa, N. Cuppens-Boulahia, and F. Cuppens, *Semantic search over encrypted data*, Proceedings of the 20th International Conference on Telecommunications (ICT), May 2013, pp. 1–5.

[PSRB16] Deepak Poola, Mohsen Amini Salehi, Kotagiri Ramamohanarao, and Rajkumar Buyya, *A taxonomy and survey of fault-tolerant workflow management systems in cloud and distributed computing environments*, Software Architectures for Cloud and Big Data Book (2016).

[Rij79] C. J. Van Rijsbergen, *Information retrieval*, 2nd ed., Butterworth-Heinemann, Newton, MA, USA, 1979.

[SCF$^+$14] M. A. Salehi, T. Caldwell, A. Fernandez, E. Mickiewicz, E. W. D. Rozier, S. Zonouz, and D. Redberg, *Reseed: Regular expression search over encrypted data in the cloud*, Proceedings of the 7th IEEE International Conference on Cloud Computing, June 2014, pp. 673–680.

[SCTB16] Mohsen Amini Salehi, Thomas Caldwell, A. Nadjaran Toosi, and R. Buyya, *Reseed: Regular expression search over encrypted data in the cloud*, Software-Practice and Experience (2016).

[Sen13] Jaydip Sen, *Security and privacy issues in cloud computing*, Architectures and Protocols for Secure Information Technology Infrastructures (2013), 1–45.

[SWP00] Dawn Xiaodong Song, D. Wagner, and A. Perrig, *Practical techniques for searches on encrypted data*, Proceedings of the 17th IEEE symposium on Security and Privacy, May 2000, pp. 44–55.

[SZXC14] Xingming Sun, Yanling Zhu, Zhihua Xia, and Lihong Chen, *Privacy preserving keyword based semantic search over encrypted cloud data*, International Journal of Security and Its Applications **8** (2014), no. 3.

[TCP+16] Alberto Tonon, Michele Catasta, Roman Prokofyev, Gianluca Demartini, Karl Aberer, and Philippe Cudr-Mauroux, *Contextualized ranking of entity types based on knowledge graphs*, Web Semantics: Science, Services and Agents on the World Wide Web **3738** (2016), 170 – 183.

[TVR02] Anastasios Tombros, Robert Villa, and C.J Van Rijsbergen, *The effectiveness of query-specific hierarchic clustering in information retrieval*, Information Processing & Management **38** (2002), no. 4, 559 – 582.

[vLSD+10] Peter van Liesdonk, Saeed Sedghi, Jeroen Doumen, Pieter Hartel, and Willem Jonker, *Computationally efficient searchable symmetric encryption*, Proceedings of the 7th VLDB Workshop on Secure Data Management, Springer, Sep. 2010, pp. 87–100.

[WRYU12] C. Wang, K. Ren, Shucheng Yu, and K. M. R. Urs, *Achieving usable and privacy-assured similarity search over outsourced cloud data*, Proceedings of the 31st IEEE International Conference on Computer Communications, INFOCOM '12, Mar. 2012, pp. 451–459.

[WSR16] Jason Woodworth, Mohsen Amini Salehi, and Vijay Raghavan, *S3c: An architecture for space-efficient semantic search over encrypted data in the cloud*, Proceedings of the 3rd International Workshop on Privacy and Security of Big Data (PSBD), 2016.

[XC99] Jinxi Xu and W. Bruce Croft, *Cluster-based language models for distributed retrieval*, Proceedings of the 22nd International ACM SIGIR Conference on Research and Development in Information Retrieval (New York, NY, USA), SIGIR '99, ACM, 1999, pp. 254–261.

Woodworth, J.W. Bachelor of Science, University of Louisiana at Lafayette, Spring, 2015;
Master of Science, University of Louisiana at Lafayette, Summer, 2017;
Major: Computer Science
Title of Thesis: Secure Semantic Search Over Encrypted Big Data in the Cloud
Thesis Director: Mohsen Amini Salehi
Pages in Thesis: 71; Words in Abstract: 173

ABSTRACT

Cloud storage is a widely used service for both a personal and enterprise demands. However, despite its advantages, many potential users with sensitive data refrain from fully using the service due to valid concerns about data privacy. An established solution to this problem is to perform encryption on the client's end. This approach, however, restricts data processing capabilities (e.g. searching over the data). In particular, searching semantically with real-time response is of interest to users with big data. To address this, this thesis introduces an architecture for semantically searching encrypted data using cloud services. It presents a method that accomplishes this by extracting and encrypting key phrases from uploaded documents and comparing them to queries that have been expanded with semantic information and then encrypted. It presents an additional method which builds off of this and uses topic-based clustering to prune the amount of searched data and improve performance times for big-data-scale. Results of experiments carried out on real datasets with fully implemented prototypes show that results are accurate and searching is efficient.

BIOGRAPHICAL SKETCH

Jason W. Woodworth received his Bachelor of Science in the spring of 2015 in computer science from the University of Louisiana at Lafayette in Louisiana. He immediately began his pursuit of a master's degree in the fall of 2015 at the same university. He completed the requirements for it in summer of 2017, and intends to further pursue a doctorate.