S3C: An Architecture for Space-Efficient Semantic Search over Encrypted Data in the Cloud

Jason Woodworth*†, Mohsen Amini Salehi†, Vijay Raghavan*
*The Center for Advanced Computer Studies
†High Performance Cloud Computing (HPCC) Laboratory
School of Computing and Informatics
University of Louisiana at Lafayette, Louisiana, USA
Email: {jww7675, amini, raghavan}@louisiana.edu

***Abstract*—The recent rapid growth in Internet speeds and file storage requirements has made cloud storage an appealing option on both a personal and enterprise level. Despite the many benefits offered by cloud storage, many potential users with sensitive data refrain from fully utilizing this service due to valid concerns about information privacy. An established solution to this concern is to perform encryption on the user side with the key stored on a local machine, meaning the cloud will never see the user's plaintext data. However, by encrypting data on the user side data processing capabilities (*e.g.,* searching) are lost. In particular, the ability to semantically search is of the user's interest in large datasets. In this paper, we present S3C, a system that provides a semantic search functionality over encrypted data in the cloud. S3C combines approaches from traditional keyword-based searchable encryption and semantic web searching. It offers a user transparent experience that accepts a simple multi-phrase query and returns a list of documents ranked by semantic relevance to the query. Our proposed approach is space-efficient, which makes it suitable for large scale datasets. Our minimal processing also allows the system to be run on thin clients such as smart-phones or tablets. We evaluate the performance of our system against various real-world datasets, and our results show that it produces accurate search results while maintaining minimal storage overhead (∼0.3% of the dataset size).**

***Index Terms*—Cloud services, Searchable Encryption, Semantic Search.**

I. Introduction

Cloud storage is an efficient and scalable solution for companies and individuals who want to store large to huge numbers of files without the burden of maintaining their own data center. Despite the advantages offered by these solutions, many potential clients abstain from using them due to valid concerns over the security of the files once they are on remote servers, and thus desire stronger cloud security [9]. Traditionally, cloud providers provide security by encrypting user documents on their own servers and storing the encryption key remotely, allowing internal attackers to access unauthorized data. One proven solution that addresses this concern is to perform the encryption locally on the user's machine before it is transferred to the cloud [18]. Unfortunately, this limits the user's ability to interact with the data, most importantly limiting the ability to search over it. Although solutions for searchable encryption exist, they often do not consider the *semantic* meaning of the user's query, impose a large storage *overhead*, or do not *rank* documents based on their relevance to the query. This research is an attempt to solve these issues.

Our motivation in this research is an organization with increasingly large amounts of data with sensitive information, with 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. One example of such organization is a hospital with encrypted patient records on the cloud and staff who would like to be able to find patients with similar diagnoses using a tablet. Another example is a police organization with officers who would like to search over encrypted police records on government clouds while on the move with their PDAs. An organization with this desire would need a system that provides security for their documents and a searching mechanism in which the user only has to enter a plaintext search query, and receive search results ranked based on the document's relevance to the entered query. Therefore they can see the most relevant documents first and will not have to comb through a list of all relevant documents.

Other solutions to the problem of searchable encryption often fall short in three ways that make them unsuitable for the examples described above. *First*, they do not offer semantic searching, meaning the user would need to remember exact keywords in the documents they are searching for. This is inappropriate for a big data environment as it is unlikely that the users would be able to remember exact keywords very well [2]. *Second*, solutions that do offer semantic searching either only make the system more forgiving of typos and words with similar spellings, or utilize a large semantic networks that need to be stored locally making them inappropriate for thin-clients (*e.g.,* [19]). *Third*, solutions that offer semantic searching often do not rank the related files by their relevance to the query (*e.g.,* [12]), instead offering only a boolean search which returns a potentially huge pool of all related files.

Therefore, we can define our problem as needing to answer these four questions:

- How to semantically search a multi-phrase query over encrypted files stored in the cloud?
- How to rank results of a search based on semantic relevance to the user's query?
- How to do the search processing on the cloud without revealing data to the cloud?
- How to provide an approach that imposes the minimum storage and processing overhead?

In this paper, we introduce **S**ecure **S**emantic **S**earch over encrypted data in the **C**loud (S3C), a scalable system that

978-1-4673-9005-7/16/$31.00 ©2016 IEEE

performs a semantic search on locally encrypted data that lives on the cloud. Our approach only parses and encrypts data on the client side so the user machine is the only part of the architecture that sees plaintext data. Documents are parsed and indexed in a manner that takes constant storage space per document. The search system resides on the cloud server, relieving the client machine of the search processing. Users are able to upload documents to a remote storage location, perform a semantic search over their encrypted data, and receive a list of documents ranked by their relevance to the query. Experiments that we have performed on real-world datasets demonstrate the accuracy, performance, and scalability of S3C.

In summary, the contributions of this paper are as follows:

- Proposing a secure method for searching a multi-phrase query over encrypted data in the cloud.
- Providing a method for ranking search results based on their semantic relevance to the user's query.
- Presenting a working prototype of our system.
- Analyzing the performance of this system, and discuss tradeoffs between performance and security.
- Analyzing the relevance of the documents retrieved by the system.
- Analyzing the impact of query length on the performance of the system.

The rest of the paper is organized as follows. Section II reviews related works in the literature, establishing the need for our solution. Section III provides a concise formal formulation of the problem and our system model. Section IV gives an overview of our proposed system architecture. Section V gives an overview of our general method, while section VI goes into scheme-specific details. Section VII reviews the threat model we are working with and provides a security analysis of our solution. Section VIII presents the results of our evaluations using real-world datasets. And finally, section IX concludes the paper and shows a plan for future works and extensions.

II. Related Work

We provide a review on other research works undertaken in the three fields most related to this work and position the contribution of our works against them.

A. 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 cyryptographic 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 [18], especially when operating on large storage blocks with large files. This approach was pioneered by Song *et al.* [18], 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 [3] 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 [6] 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.* [4] 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 [21] 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.

B. Semantic Search

Much of the work into searching semantically has been done in the context of searching the web [1], [13], [20]. Some of these works, such as RQL by Karvounarakis [10], 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 [5], [11] aim for a completely user-transparent solution where the user needs only to write a simple query with possible tags, while others still [7], [8] 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.

C. 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 [12] a system which could handle minor user typos through a fuzzy keyword search. Wang *et al.* [22] 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 [17] a system for searching for regular expressions, though this still neglects true semantics for another form of similarity.

Moataz *et al.* [15] used various stemming methods on terms in the index and query to provide more general matching. Sun *et al.* [19] 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.

III. System Model

The problem of multi-phrase searching can be formally represented using the following elements:

- 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 encryption 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.

IV. S3C Architecture

S3C has three main components, namely the *client application*, *cloud processing server*, and *cloud storage*. 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 1 presents an overview of the three components and processes associated with them in the system.

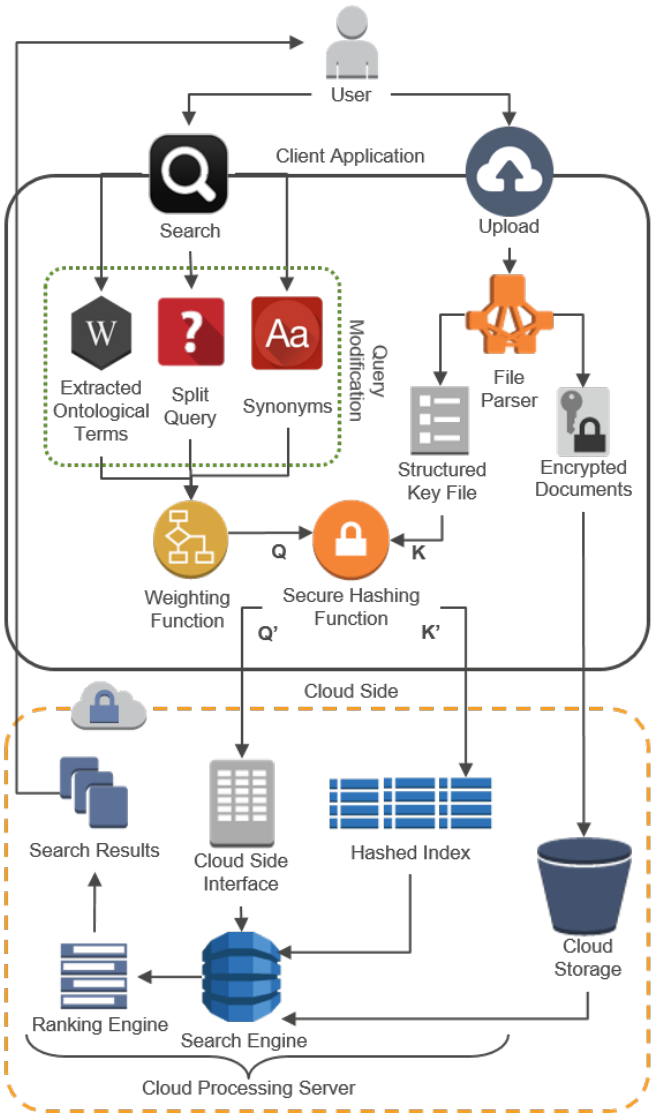

Fig. (1) Overview of S3C architecture and processes. Parts within the solid-line group indicate items or processes on the client side which are considered trusted. Parts in the dashed-line group indicate those in the cloud processing server. All components in the cloud are considered untrusted.

A. 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 keywords from plaintext documents and encrypting them before they are uploaded to the cloud.

When the user requests to search, S3C expands the query based on the system's semantic 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.

B. Cloud Processing Server

The cloud server is responsible for constructing and updating the inverted 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 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 its index (see section V) 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.

C. 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, so long as the computing server knows where each document is stored and the index is updated accordingly.

V. Overview of Upload and Search Processes

A. Overview

The main idea behind our approach is that if we can use a searching method that is agnostic towards the meaning of the terms in the documents or the query and only considers their occurrence and frequency, then we can perform the search over encrypted data. For that purpose, we should assure that we transform each occurrence of a distinct word in every document into the same token, and consequently apply the same transformation when that word appears in the search query. Doing this ensures that a match is still produced during the search process. Hashing is a good way to achieve this.

The Okapi BM25 algorithm [16], frequently used for standard text retrieval, is a term-frequency, inverse-document-frequency model that works using an inverted index. The algorithm does not need to consider actual meaning of the terms in the document, and only needs to know in which documents they exist. This feature makes it very applicable to our use case.

S3C has two main functionalities: *uploading* documents from the client, and *searching* over documents in the cloud. We explain these functionalities in the rest of this section.

B. 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 file. 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 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 within the index.

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

C. Search Process

The search process consists of two main phases: *query modification* and *index searching and ranking*. The query modification 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 in the original query q and expands into the modified query set Q. It involves three phases: query splitting, semantic expansion, and weighting.

The goal of splitting the query is to break q into smaller components. This is done because a multi-phrase string hashes to a different value than the sum or concatenation of the hash values 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 naïve approach to this is to perform a synonym lookup for each member of Q (termed Q_i) 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, our system pulls from more advanced ontological networks. For example, in this research we use the contents of Q to pull entries from Wikipedia and perform keyphrase extraction 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 store semantic networks locally.

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. 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 splitting are weighted as $1/n$ where n is the number of terms derived from splitting.
- 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 our modification of the BM25 equation described in the following equations:

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

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

$$IDF(Q'_i) = \log \frac{N - n(Q'_i) + 0.5}{n(Q'_i) + 0.5} \quad (2)$$

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 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 and returns the list to the client, sorted by score in descending order.

VI. Search Schemes

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

A. 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 splitting 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 splitting the query into larger groups of words would add no value.

Ranking for FKSS is performed with no modification from Equation (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.

B. 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 (in our implementation, we considered collecting 10 keyphrases). 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.

In an effort to further increase security, term frequency is eliminated on 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 split the query, SKSS splits not only into individual words, but into all possible adjacent subsets. An example of this can be seen in Figure 2. 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 splitting is complete, synonyms and related terms are looked up for all of the resulting phrases in the query set.

"`Failure in Wireless Sensor Networks`", "`Failure Wireless Sensor`", "`Wireless Sensor Networks`", "`Failure Wireless`", "`Wireless Sensor`", "`Sensor Networks`", "`Failure`", "`Wireless`", "`Sensor`", "`Networks`"

Fig. (2) A sample of the query splitting done by SKSS.

When performing ranking, SKSS modifies Equation (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 $f(q_i, d_i)$.

C. 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 split 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 split 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.

VII. 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 some definitions.

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.

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.

Trace: The trace is the precise information leaked about H_q. For S3C, this includes file identifiers 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 the 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 process 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.

VIII. Performance Evaluation

A. Overview

To evaluate the performance of S3C and provide proof of concept, we tested it 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 our system under Big data scale datasets, we utilized a second dataset, the Common Crawl Corpus from AWS, a web crawl composed of over five billion web pages We evaluated our system against the RFC using three types of metrics: *Performance*, *Overhead*, and *Relevance*.

B. Metrics for Evaluation

1) 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 [14]. This method is a modification of the precision-recall method commonly used for judging text retrieval systems. It is defined as follows:

$$Score = \frac{\sum_{i=0}^{N} r_i}{N} \tag{3}$$

Where i is the rank of the document determined by the system and N is the cutoff number (10 in our 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.

2) Performance: We define performance as the time it takes to perform the search operation. The aspects of performance we measure 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.

3) 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.

C. 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 the nurse attempting to find all patients with a similar disease or diagnosis 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.

- 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)`

Fig. (3) 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.

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 our 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

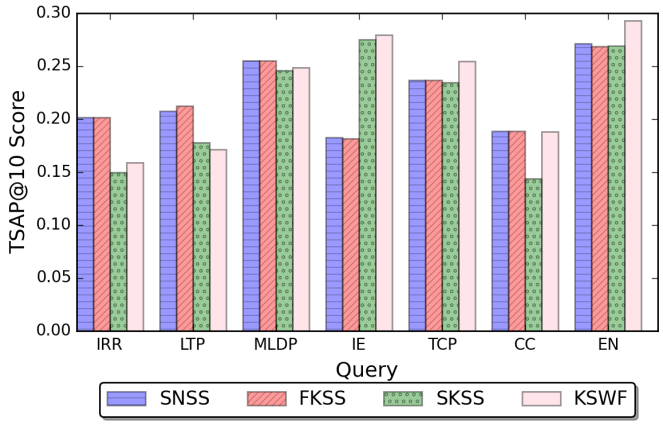

Fig. (4) TSAP@10 score for the specified query for each system. Once the system has returned a ranked list of results, a score is computed based off of a predetermined relevance each file has to the given query.

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.

D. Evaluating Relevance

Figure 4 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. Our 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.

E. Evaluating Performance

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.

Results can be seen in Figures 5, 6, and 7. 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

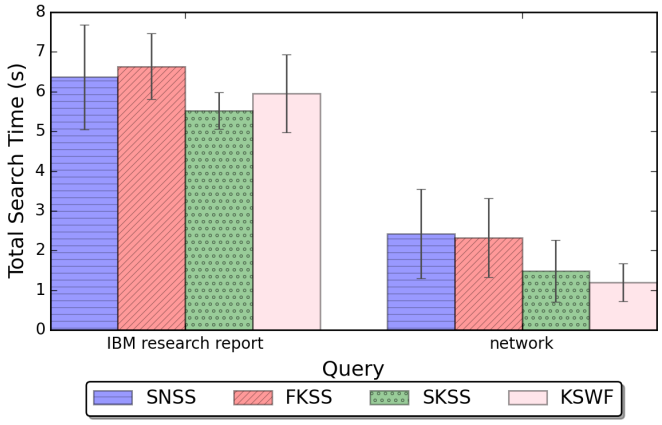

Fig. (5) Total search time in each scheme. This includes the time taken to process the query, communicate between client and server, and perform searching over the index. The results are averaged over 50 runs.

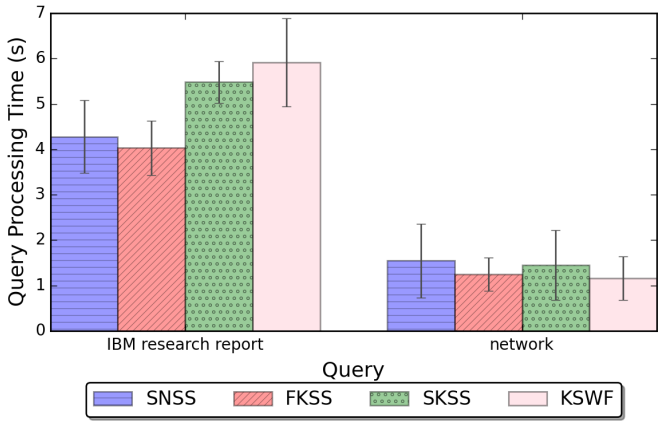

Fig. (6) Time to process the query. This includes query modification and hashing into the trapdoor. The results are averaged over 50 runs.

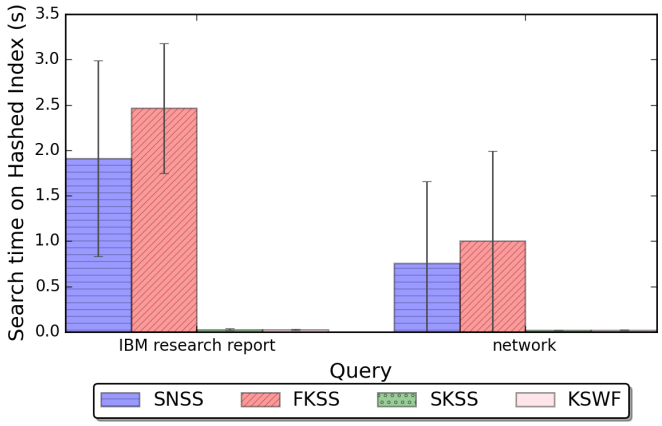

Fig. (7) Time it takes to perform the search on the hashed index on the cloud. This includes the time taken to find all files in the hashed index that contain any hashed terms in the query trapdoor and rank them with the scheme's respective functions. The results are averaged over 50 runs.

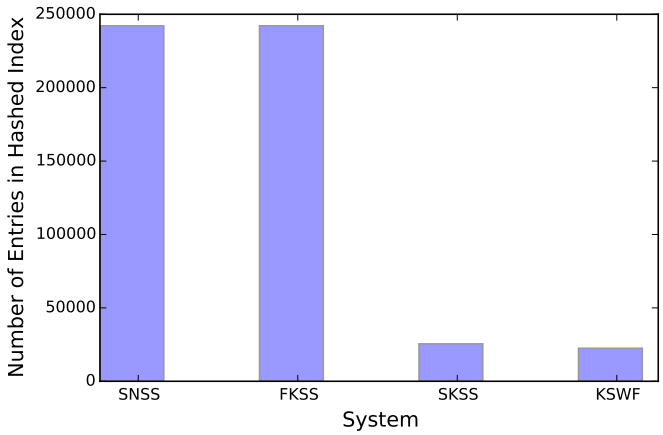

Fig. (8) Size of the inverted index for each system. An entry denotes a hashed keyword mapped to a set of file identifiers.

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.

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.

F. Evaluating Overhead

To demonstrate space-efficiency in our 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 8 and 9. 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).

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

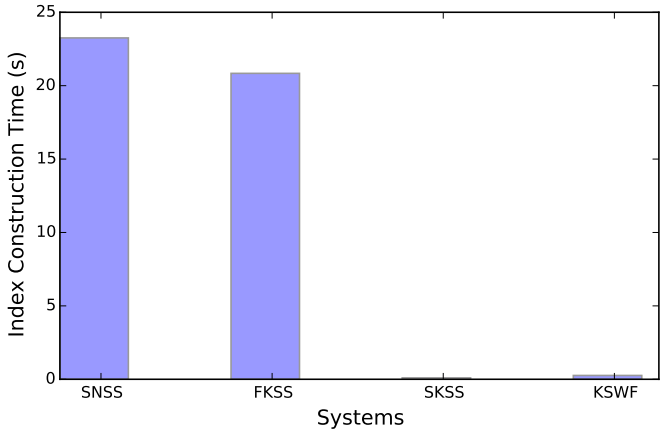

Fig. (9) Time it takes to construct the hashed index upon server startup. This operation includes sequentially reading an index file hosted on the cloud server which contains all data for the inverted index and document sizes table and storing it in hash tables.

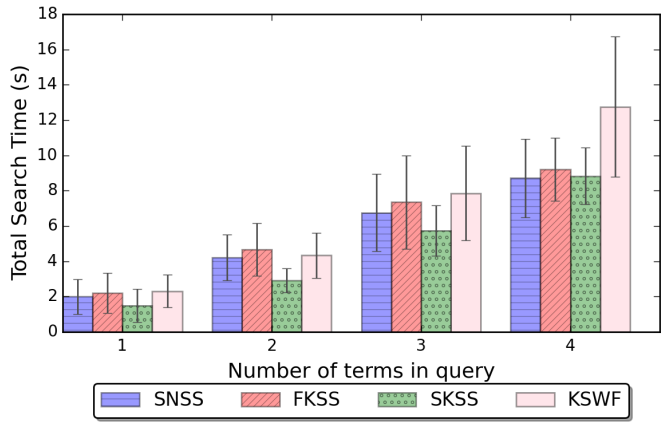

Fig. (10) Total search time for an expanding query. This includes the time to process the search query, communicate between client and server, and rank in the cloud. The horizontal axis shows the number of words (minus stopwords) in the query. The results were averaged over 50 runs.

time regardless of the size of the dataset due to the hash table structure of the index.

G. Evaluating the Impact of Query Length

In addition to measuring search times for individual queries, we are interested in measuring the effect of expanding the size of a single query from one term to four. For example, 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 10 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

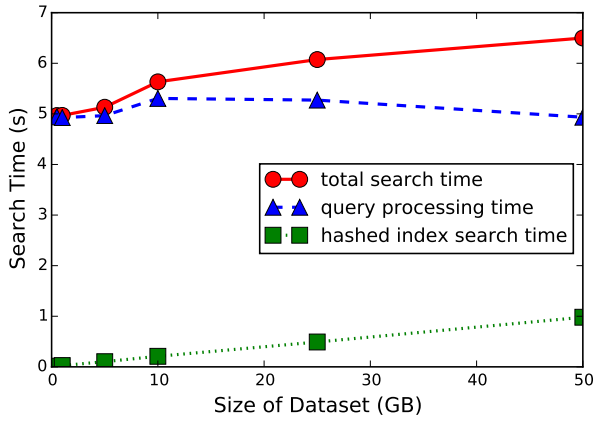

Fig. (11) Time taken to search for different dataset sizes. Resulting times are the mean of 50 runs performed with multiple three word queries. The dotted line shows the time taken to search on the hashed index in the cloud, the dashed line shows the time taken for query modification, and the solid line shows total time taken for the search (including query modification and index searching).

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.

H. 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. The resulting search times are an average of mid-sized (three word) queries. Figure 11 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 our system, we measured the size of the index at each size increase during our test. The results are shown in figure 12. 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.

IX. Discussion and Future Works

In this paper, we presented S3C, a system for securely and semantically searching data in the cloud, with three different schemes fit for different use cases. Our system improves upon existing encrypted data search techniques by providing

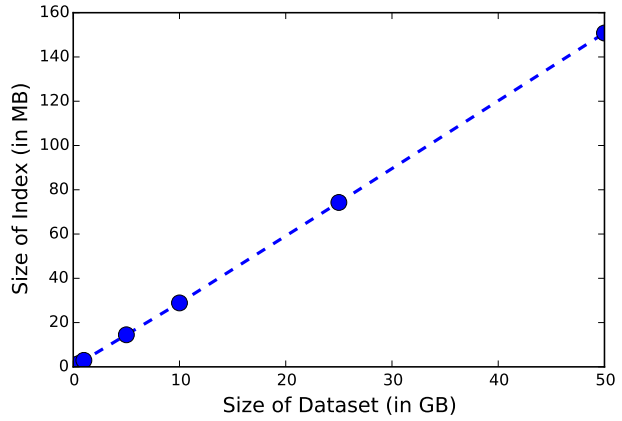

Fig. (12) Size of the index file for different dataset sizes. The horizontal axis plots the size of the dataset used in gigabytes, while the vertical axis plots the associated index size in megabytes.

a solution that is space-efficient (*i.e.,* SKSS) 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. The system requires only a single plaintext query to be entered and is easily portable to thin-clients, making it simple and quick for users to use. The system is also shown to be secure and resistant to attacks.

Our experiments with a working prototype of each of our 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, hospital records with encrypted diagnosis 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 big data scale datasets.

Because of the nature of our semantic query expansion, we discuss how this work can be expanded. As new online semantic networks are added to the internet and made available for applications, they too could be extracted from and added to the query as part of the semantic processing step. This could lead to more accurate domain-specific searching. Additionally, we are working towards pruning the search processing through topic-based clustering on the hashed index and only searching over related clusters.

REFERENCES

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

[2] Christian Bizer, Peter Boncz, Michael L. Brodie, and Orri Erling. The meaningful use of big data: Four perspectives – four challenges. *ACM SIGMOD Record*, 40(4):56–60, Jan. 2012.

[3] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. *Public Key Encryption with Keyword Search*, pages 506–522. Springer, 2004.

[4] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934, Nov. 2011.

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

[6] Eu-Jin Goh et al. Secure indexes. *Cryptology ePrint Archive*, page 216, 2003.

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

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

[9] M. Javanmard, M. A. Salehi, and S. Zonouz. Tsc: Trustworthy and scalable cytometry. In *Proceedings of the 7th IEEE International Symposium on Cyberspace Safety and Security*, pages 1356–1360, Aug. 2015.

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

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

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

[13] Christoph Mangold. A survey and classification of semantic search approaches. *International Journal of Metadata, Semantics and Ontologies*, 2(1):23–34, 2007.

[14] 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*, pages 4–6, Dec. 2012.

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

[16] Stephen E Robertson, Steve Walker, Susan Jones, Micheline M Hancock-Beaulieu, and Mike Gatford. Okapi at TREC-3. *Overview of the Third Text Retrieval Conference (TREC-3)*, 3:109–126, 1995.

[17] 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. In *Proceedings of the 7th IEEE International Conference on Cloud Computing*, pages 673–680, June 2014.

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

[19] 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(3), May 2014.

[20] 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*, 3738:170 – 183, Mar. 2016.

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

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