Implementing a Similarity Searchable Encryption Scheme for Cloud Database Usage

Christian Göge
Institut für Informatik
Goldschmidtstraße 7
37077 Göttingen, GERMANY
christian.goege@informatik.uni-goettingen.de

Tim Waage
Institut für Informatik
Goldschmidtstraße 7
37077 Göttingen, GERMANY
waage@informatik.uni-goettingen.de

Lena Wiese
Institut für Informatik
Goldschmidtstraße 7
37077 Göttingen, GERMANY
wiese@informatik.uni-goettingen.de

ABSTRACT

In this paper we address the use case of secure database-as-a-service in the cloud, for example to implement an encrypted email server. We empirically derive proper settings for a similarity searchable encryption scheme, which allows searching in encrypted emails with search terms that can contain misspells. The focus here is on describing and evaluation parameters of an existing scheme with its original settings and comparing them to an improvement to this scheme introduced by us. The scheme is embedded into an existing framework that can connect to Apache Cassandra and HBase.

Keywords

searchable encryption, similarity search, fuzzy search, secure index, encrypted database

1. INTRODUCTION

Today, more and more data is outsourced to the cloud. Especially smaller companies can outsource their data management to a remote service provider. Most business data, such as emails, are sensitive, so outsourcing them as plain text is not a good idea, since the service provider is able to read them. Even if the service includes encryption at the server side, the data needs to be sent to the service provider in plain text first. For this reason, encryption of sensitive data has to take place at the client side. However, outsourcing encrypted data prevents data users from searching in the data. To obtain search results, one would have to download everything and decrypt it first, which obviously ridicules the idea of outsourced data. One solution is searchable encryption. Searchable encryption schemes allow searching for single or multiple keywords in a collection of documents. They often achieve this by building a *secure index* [5], which is outsourced together with the data collection. A secure index can be queried using a "trapdoor", an encrypted form of a query.

29th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 30.05.2017 - 02.06.2017, Blankenburg/Harz, Germany.

This article implements and expands the searchable encryption scheme of Kuzu et. al. [6], who also achieve similarity search that allows misspelled queries. The schemes parameters are identified and evaluated with a real world email dataset.

2. RELATED WORK

The most popular work on performing queries on encrypted data is CryptDB [10]. It first introduced the onion layer model. Onion encrypted columns have several encryption layers, with the topmost being the most secure. Secondary layers leak information, but allow lookup or range queries. However, CryptDB only considers slow querying schemes because they avoid client and / or server side indexes, and it is limited to mySQL and PostgreSQL. "Monomi" [14] tries to achieve arbitrary SQL queries at the cost of higher computational requirements for the client machine. [8] introduces "BlindSeer", which addresses sub-linear searches for boolean SQL queries. A non-relational database approach is "Arx" [9], which builds on top of MongoDB. It uses two proxy servers and needs to know in advance which operations are to be executed on which fields, in order to maintain the indexes.

After having reviewed some existing database encryption solutions, we turn our attention to searchable encryption schemes. Song et. al. [13] were the first to tackle searchable encryption. Their scheme does not use an index, but an encryption construct with two layers that allows sequential search of the ciphertext. Goh [5] first described a secure index. Curtmola et. al. [2] developed a keyword-based index and defined the key security definition of *adaptive semantic security*. A similarity searchable encryption scheme that allows multi-keyword queries was introduced by Wang et. al. [16]. They use a bigram vector and LSH functions to achieve the similarity search, but pay with a large overhead in the search.

Bösch et. al. [1] provide a complete survey of provably secure searchable encryption schemes and their properties over the last decade.

3. OUR FRAMEWORK

Our group developed a framework[1] for cloud database encryption that gives full control to the database user (client). Each table column is, based on the data type, encrypted with several encryption schemes which allow different search

[1]FamilyGuard `https://github.com/dbsec/FamilyGuard`

operations: deterministic encryption (DET) for direct lookups, order-preserving encryption (OPE) for range queries and searchable encryption (SE) for text search. Besides the search over encrypted data, SE schemes provide additional functionality which is not possible in standard databases: the search for single words in text columns. Three searchable encryption schemes have been studied in this framework in [15] using the Enron email dataset [3]. We now supplement the framework with a similarity searchable encryption scheme (SSE). Similarity search is the search for words in a text corpus that are similar to a query word in terms of spelling, so queries are able to find the correct documents even if the query word was misspelled. The scheme was described by Kuzu et. al [6] and expanded by us to achieve better search results.

Before describing the search scheme, we first take a look at the existing framework, describing the main package contents. The *database* package controls all communication of the database with the cloud databases. Adding support for a new database is easy: It only requires writing a new *DBClient* class that translates the internal query language into the database's driver language and handles the connection. Currently, we use Apache Cassandra and HBase for our testing environment.

The similarity searchable encryption scheme of Kuzu et. al. [6] builds an *index* from the document collection. This index is simply stored in an additional database table at the remote side. Searching for documents containing a query word w requires two steps:

1. Querying the index. This returns an encrypted form of document identifiers.
2. Querying the dataset for the documents.

All searchable encryption schemes in our framework comply with an interface consisting of two key functions: *encrypt* and *search*. The *encrypt* function expects an input string and returns the encrypted form, such that the using class can then decide where to store the encrypted document. Thereby the scheme creates its index table. The *search* function executes the search on the index and returns a set of document identifiers for that search.

4. THE SSE INDEX

The similarity searchable encryption scheme by Kuzu et. al. [6] builds an *inverted index* on the document collection, meaning that it relates keywords to a list of documents that contain the keyword. Thereby it does not use the keywords directly. From each keyword, a fixed number of λ subfeatures are extracted using *locality sensitive hashing (LSH)* [11]. Locality sensitive hashing is a nearest-neighbor approximation algorithm for high-dimensional spaces. LSH functions have the property that they produce the same value for two inputs with a probability that is directly linked to the inputs' similarity. Used with keywords in the index, we get a measure of how likely it is to find the query word in a document: On index construction, each keyword's λ LSH features are extracted and related to the documents they contain. The query process also extracts λ LSH features from the query and returns the lists of document identifiers that are linked to each LSH feature. For each document identifier, we can now count in how many lists it is contained. All document identifiers which occur in at least one list (bucket) are candidates that are likely to contain a word that is indeed similar to the query word.

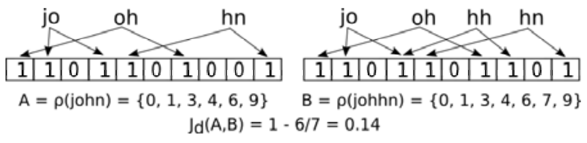

Figure 1: Kuzu et. al.: Bloom filter hashed embedding

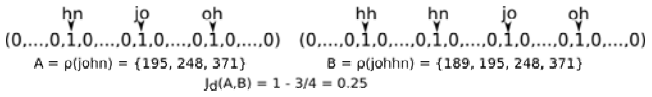

Figure 2: Our Bigram vector embedding

4.1 String distance

Using LSH requires a distance function for the keywords. The best known way of measuring the distance between two strings is the Levensthein or Edit distance [7], the number of replace, delete and insert operations to transform one string to the other. Unfortunately, no LSH functions are known for this distance [6]. Therefore, we have to embed strings in a metric space in which LSH functions are known. The embedding function is denoted as ρ.

4.1.1 Hashed embedding

Kuzu et. al. adopt their way of measuring string distance from [12]. A keyword's bigrams are hashed into a Bloom filter with a number of cryptographic hash functions. The Bloom filters are then interpreted as sets of indices, where the Bloom filter value is 1. The similarity between these sets is measured with the Jaccard index (or Jaccard similarity) $J = \frac{|A \cap B|}{|A \cup B|}$, the distance between two embedded strings is then $J_d = 1 - J$. This process is pictured in Fig. 1.

4.1.2 Direct Bigram embedding

We found we can improve the retrieval success of the scheme by using a similar, yet more precise and faster embedding, because the cryptographic properties of Kuzu et. al.'s embedding do not impact the security of the scheme. Our embedding uses the set of bigrams directly to compute distances, this way it is faster to compute and smaller in memory than the original embedding. We only consider lower case letters, then every possible bigram is numbered from "aa"= 0 to "zz"= 675. We show the embedding in Fig. 2.

4.1.3 Implementation

For index security reasons, the above mentioned lists of document ids all have to be the same size and are therefore realised as bit vectors, denoted as V_{B_k}. B_k is the bucket identifier, the LSH feature extracted from a keyword that is linked to this bit vector. The documents are simply numbered from 0 to $n-1$ and $V_{B_k}[i] = 1$ if LSH feature B_k was derived from a keyword contained in document i. The index creation process (without encryption) is shown in Fig. 3. Before outsourcing this index, B_k and V_{B_k} are encrypted separately, B_k is encrypted with a keyed hash function and the V_{B_k} are encrypted using AES with one secrect key but

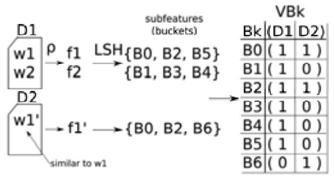

Figure 3: Index construction

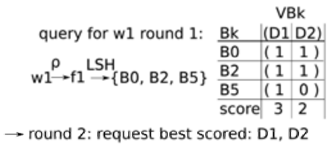

Figure 4: Querying the index

with different init vectors. The init vectors can be outsourced together with the V_{B_k}.

Querying the index for a word q runs the same steps as the index construction on the word. q is first embedded in the metric space using the embedding function ρ, then λ LSH features are extracted. The LSH features are encrypted the same way as the bucket identifiers B_k. The λ encrypted LSH features are also called a *trapdoor* for the index. After receiving the encrypted bit vectors V_{B_k}, the client can decrypt them, rank the results and request the best scored documents from the document collection. This is pictured in Fig 4.

4.2 Security

As an adversary model, we assume the cloud server to be honest-but-curious, meaning it will honestly carry out its assigned tasks while curiously trying to learn about the data it contains. The scheme suffices the security definition of *adaptive semantic security* (IND-CKA2) [2]. The search scheme is secure, if the adversary, who has access to the secure index and a query history, can not compute more information about the dataset than what was leaked from the index on purpose. The index purposely leaks

1. The number of documents and the length of each encrypted document,
2. the search pattern, e.g. which trapdoors are queried frequently (since the encrypted trapdoors are deterministic),
3. the access pattern e.g. which trapdoor corresponds to which document,
4. the similarity pattern between keywords, that is the number of shared subfeatures for two queries.

The proof uses a polynomial time simulator which constructs a fake view of this leaked information with random numbers but same properties. The adversary then has a chance of only marginally greater than 1/2 to distinguish the real and the fake view. The proof is completely carried out in [6]. Our contribution alters only the computation of the unencrypted B_k, the encryption of B_k stays the same as in [6]. Therefore the security proof given there still holds.

4.3 Restrictions

Given the index scheme, the following restrictions in usability arise.

1. The index is basically not updatable. Maintaining the security requirement of same-length bit vectors V_{B_k} makes adding a new document to the index very inefficient. The index is designed to be computed once with the complete document collection and then to be outsourced to the cloud.
2. For computing the index, we need to know the number of documents beforehand.
3. The numbering restricts us to use integers as identifiers in the data table or maintaining a second table that maps the index numbering to the actual document identifer used in the data table.
4. We need to maintain (and perhaps distribute to other data users) not only the encryption keys for the index, but also the used embedding and LSH functions.

The framework instantiates encryption scheme classes when a new table is created, and the class representing a column of a table has a reference to it. In this design, we cannot tell the encryption scheme class how many documents are to be inserted into the table at creation time, because this would make the overall table usage very inflexible. Therefore we decided to buffer all incoming documents and starting the index creation and outsourcing with another method that is called when a batch insertion is finished. Up to this point, the *encrypt* function will keep the current index state in memory. The index gets pretty large (at most $\lambda \cdot \#$distinct keywords), which makes this process rather RAM intensive.

5. EVALUATION

We evaluate our index scheme against the original from Kuzu et. al. using the Enron email dataset [3]. Business emails definitely can contain sensitive information and storing them on a foreign server should only be implemented with encryption. The email's fields (sender, subject, body, timestamp, ...) are parsed and encrypted using the framework. The searchable encryption indexes additionally achieve searching for single words in bigger text columns like email subject or body, but we need to create a searchable encryption index for each of those string columns.

5.1 Typo Generator

To evaluate the success of the search scheme in the context of error-aware keyword search, the keywords in the mails need to be misspelled. Sadly, the typo generator used by Kuzu et. al. seems to be no longer available. Also, they do not state how bad the misspellings were. For this setup, a simple typo generator was implemented, which generates one of the following spelling mistakes:

- Double Letter: The letter at a random position is inserted doubled.
- Skipped Letter: The letter at a random position is deleted.
- Switched Letters: At a random position, a letter is switched with its neighbor.

These spelling mistakes have different impacts on the Jaccard distance between a word and its misspelled version (see next section).

5.2 LSH Parameters

Crucial for the correct retrieval of documents similar to the query is the fuzzy search threshold used in LSH. Kuzu et. al. only speak about "some empirical analysis", which lead to the chosen parameters for the LSH. This shall be investigated here in detail.

The LSH algorithm can be tuned with two parameters: λ, the number of subfeatures extracted from a keyword, and k, the number of internal functions to compute one subfeature [11]. Let $g_i(\cdot), i \in [1, \lambda]$ denote the functions that compute the LSH features, then the probability that two keywords share at least one subfeature is

$$Pr[\exists i \mid g_i(x) = g_i(y)] = 1 - (1 - J(x,y)^k)^\lambda \qquad (1)$$

and $J(x,y)$ is the Jaccard similarity. This function of k and λ forms an *s-curve*, as shown in Fig. 5. We have to choose the parameters such that the LSH algorithm produces probabilities according to the dataset needs. Therefore we need to determine a *threshold* on the similarity to which keywords are hashed to the same subfeatures with high probability.

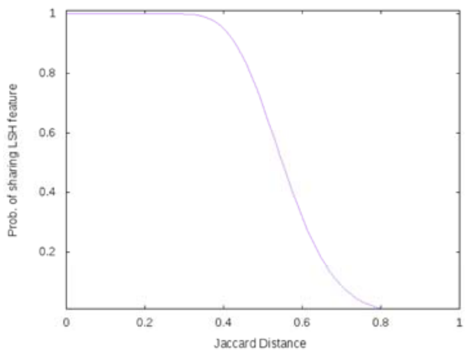

Figure 5: s-curve, $k = 5, \ \lambda = 37$

To determine the thresholds, the distance between the keywords and their misspellings was measured, this is seen in Fig. 6. More than 50 % of the words have a distance shorter than 0.5 to their generated misspells, so the threshold of retrieving these misspells with high similarity should be around this margin. Moreover, less than 10 % have a distance of 0.6 or higher. Also, as seen in Fig. 7 almost all keyword pairs existing in the documents have a distance higher than 0.8. These observations justify the chosen LSH thresholds: A mail D should be retrieved with high probability, if $\exists w \in D : J_d(w,q) \leq 0.45$, and not be retrieved if $\forall w \in D : J_d(w,q) \geq 0.8$. This leads to parameters $k = 5, \lambda = 37$.

Additionally, this investigation is also done for the bigram embedding of strings (see Fig. 8, 9). The distances appear to be distributed very similar to the hashed embedding, but it is worth noticing that distances between distinct words are in even more cases higher than 0.9. This might allow choosing the LSH parameters slightly different in order to shift the s-curve even more to the right. However, in order to compare both embeddings in the same setting, the parameters Kuzu et. al. chose for their hashed embedding were adopted.

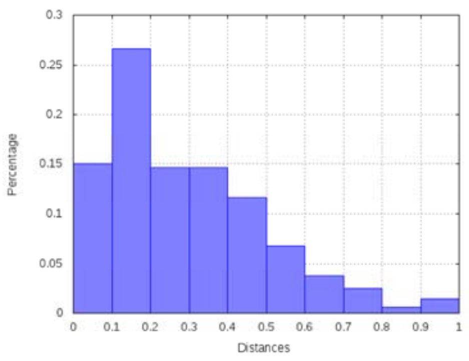

Figure 6: Distances keywords vs misspells (Bloom filter)

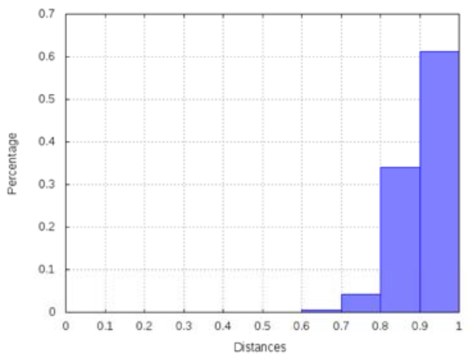

Figure 7: Distances between keyword pairs (Bloom filter)

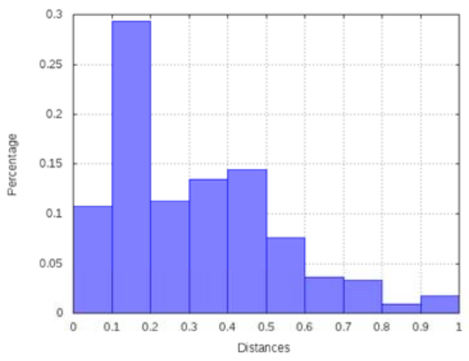

Figure 8: Distances keywords vs misspells (Bigram embedding)

Both distributions also show the effect of the three possible misspells on the Jaccard distances between the misspells and the correctly spelled query word. Theoretically, a double letter will simply add another bigram to the bigram set. The Jaccard similarity between a keyword and such a typo is then $J = n/(n+1)$, where n is the number of bigrams in the word (Ex: john vs jjohn: $J = 3/4$). If a letter was skipped, in the worst cast we delete two bigrams and create a new one. This means, the words now share two less bigrams and the total number goes up by one: $J = (n-2)/(n+1)$ (Ex: john vs jon, $J = 1/4$). Switching two letters will in the

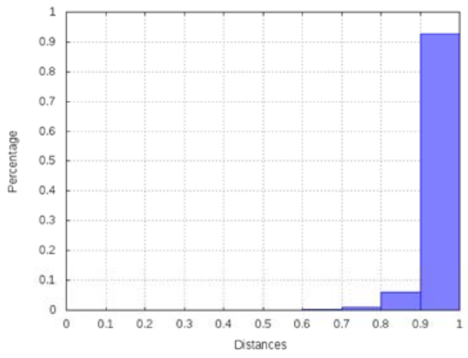

Figure 9: Distances between keyword pairs (Bigram embedding)

worst case delete three bigrams and create three new ones. (Ex: john vs jhon have no bigrams in common. $J = 0/6$). This analysis is directly applicable to the bigram embedding. For the hashed embedding this is only asymptotically correct because it can produce collisions in the Bloomfilter, which makes two strings more similar. We can see that the stability of a word against misspelling in the Jaccard metric is highly dependent on the number of bigrams it is made of. This would not be the case with the Levensthein or edit distance, but again, in our application we need a distance metric for which a LSH family is known. However, the results of our misspell analysis shows that most misspells show enough similarity to their original word.

5.3 Average Distances

The secure search scheme relies on the correlation between the distance between the documents in the database and the number of shared buckets in the LSH index. The distance between a query word q and a document D is the distance between q and the word $w \in D$, which is closest to q:

$$dist(q, D) = \min_{w \in D} dist(q, w) \tag{2}$$

If the query word q is contained in D, the distance between q and D is 0 and they share the maximum number of λ buckets. With increasing distance, the probability of sharing buckets decreases. To evaluate this property, 1000 keywords were chosen randomly from the set of all features. Their query scores (i.e. the common number of buckets with a document) and distances were evaluated against all documents in the testing sample. The average distance for a query q and the documents D^b, which share b buckets with the query is

$$adist_b(q) = \frac{\sum_i dist(q, D_i^b)}{|D_i^b|} \tag{3}$$

The average distance per bucket for the query set of 1000 keywords is shown in Fig. 10.

The average distance per bucket for the query set of 1000 keywords is shown in Fig. 10 for both string embeddings. In addition to Kuzu et. al., we also take a look at the theoretically expected value of shared buckets between a query q and documents with distance d to q. For the LSH functions g (see Sec. 5.2) holds:

$$Pr[g(q) = g(w)] = J(q, w)^k. \tag{4}$$

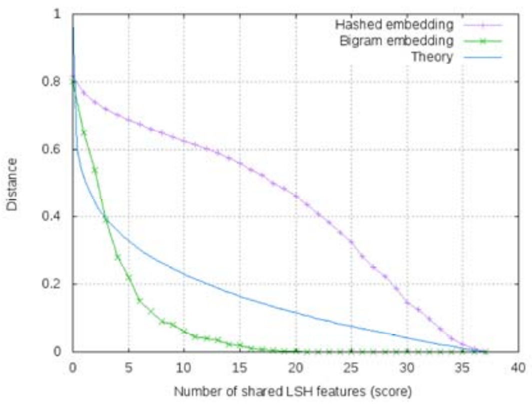

Figure 10: Average distances for number of shared buckets

If we draw λ independent LSH functions g, the number of shared buckets $B_k = g_i(q) = g_i(w)$ (the score s) is a binomially distributed random variable $s \sim \mathcal{B}(\lambda, J(q, w))$. From this follows that the expected score is $\mathbb{E}(s) = \lambda \cdot J$. Therefore the theoretical curve for the average Jaccard distance $J_d = 1 - J$ for s shared buckets is

$$J_d = 1 - \sqrt[k]{s/n} \tag{5}$$

This evaluation shows a big difference between the two embeddings. While both show a similar distance distribuiton in the previous section, the LSH step leads to very different results. With Kuzu et. al.'s hashed encoding, the distance shows an almost linear dependency on the number of common buckets while it is expected to follow the root term. The bigram vector embedding however shows distances much shorter than theoretically expected. To explain this, we will take a look at the index construct. The index was built from a total of about 48,000 keywords that were extracted from all documents. For each keyword, $\lambda = 37$ LSH features were computed, which form the buckets B_k in the index. If all words compute to λ distinct buckets, we would expect a maximum number of index entries of λ times the number of keywords. With $\lambda = 37$, this is a total of 1.7 million expected index entries. Now, due to the properties of LSH, if words are similar, they might share some of their λ buckets. For the hashed embedding, the index has a size of about 500,000 entries, which means that on average a word shares about two thirds of its bucket identifiers with other words. For the bigram vector embedding, the index size is 1.2 million, so we see words sharing a bucket in less than one third of their LSH features. This partly results from the distances to other words: for the bigram embedding, more than 90% of the words have a distance larger than 0.9 to the other words, making it less probable for them to share a bucket than the hashed embedding, where only 60% of the words have distances larger than 0.9.

Both results stem from the fact that the hashed embedding Kuzu et. al. use is computed by hashing the bigrams into a Bloom filter of length 500 with 15 different functions, which introduces the probability of collisions in the Bloom filter. That can lead to the similarity between two embedded words being larger than the similarity between their bigram vectors (see Fig. 1 and 2). Another factor is also the size of the embedding, here meaning the size of the sets that

n_d	n_f	n_w	$\lvert I_h \rvert$	$\lvert I_b \rvert$
1000	18 k	235 k	252 k	530 k
2000	30 k	532 k	340 k	809 k
3000	36 k	764 k	384 k	946 k
4000	44 k	1.08 m	440 k	1.15 m
5000	47 k	1.2 m	458 k	1.2 m

Table 1: #Index entries in a real world setting

represent the embedding. While in the bigram embedding the size is the number of distinct bigrams, in the hashed embedding this number is multiplied by 15 (minus the collisions in the Bloom filter). That also increases the chance for two embeddings to be considered similar in the hashed embedding. In conclusion, the bigram vector embedding is more suitable to building the index, because it makes the number of common buckets for a query and similar words in the documents much more meaningful, which we will also see in the retrieval evaluation in the next section.

5.4 Index Evaluation

In their work, Kuzu et. al. [6] evaluate the performance of their scheme in an artifical setting: From the default values of $n_d = 3000$ documents indexed with $n_f = 3000$ preselected distinct keywords and $k = 5$, $\lambda = 37$ as LSH parameters, they proceed to alter one of the parameters while leaving the others at these default values and showing the impact on the query performance. Increasing the number of documents n_d linearly increases the search time because the ranking step involves addition of bit vectors of length n_d to obtain the scores. Increasing n_f has very little impact on the search performance, because a query always has the constant size of λ encrypted LSH features. Altering λ linearly increases the search time for this exact reason. Increasing k however decreases the search time. This is because with increasing k, the probability that query shares a LSH feature decreases, and such a query has an increasing probability of not hitting a LSH bucket in the index. This results in less than λ bit vectors having to be added to achieve the scores.

We are now more interested in the performance of the scheme in a real world scenario. Therefore we build indexes from $n_d = 1000$ to 5000 randomly chosen mails from the Enron email set. Because the LSH parameters are crucial for retrieval performance and would be chosen to fit the retrieval needs, we fix them at $k = 5$, $\lambda = 37$, as altering them would greatly change the retrieval probabilities. Table 1 shows the number of distinct keywords n_f in the mails, the total number of words n_w across all mails and the sizes of the indexes (that is, the number of index entries) for both string encodings, I_h for the hashed encoding and I_b for the bigram embedding. We see that the number of distinct features is much larger than chosen in Kuzu et. al.'s artificial testing set. Also, the index sizes are greatly different for the two encodings, the reason for this was discussed earlier.

6. CONCLUSIONS

In this article we provided a practical evaluation of choosing appropriate parameters for a similarity searchable encryption scheme. We compared the scheme in [6] based on hashed Bloom filter embedding to our proposed direct bigram embedding. Future work will include benchmarking in combination with other encryption schemes in our framework (DET, OPE, SE) as well as development of improved fuzzy searchable encryption schemes.

7. REFERENCES

[1] C. Bösch, P. Hartel, W. Jonker, and A. Peter. A survey of provably secure searchable encryption. *ACM Computing Surveys (CSUR)*, 47(2):18, 2015.

[2] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. *IACR Cryptology ePrint Archive*, 2006(Rep. 210), 2006.

[3] Enron email dataset. `https://www.cs.cmu.edu/\textasciitilde./enron/`, 2015.

[4] C. Göge, T. Waage, D. Homann, and L. Wiese. Improving fuzzy searchable encryption using bigram embedding. under review.

[5] E.-J. Goh. Secure indexes. *IACR Cryptology ePrint Archive*, 2003, 2004. Rep. 216.

[6] M. Kuzu, M. S. Islam, and M. Kantarcioglu. Efficient similarity search over encrypted data. In *Data Engineering (ICDE) 2012*, pages 1156–1167. IEEE, 2012.

[7] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[8] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 359–374. IEEE, 2014.

[9] R. Poddar, T. Boelter, and R. A. Popa. Arx: A strongly encrypted database system. *IACR Cryptology ePrint Archive*, 2016:591, 2016.

[10] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: processing queries on an encrypted database. *Communications of the ACM*, 55(9):103–111, 2012.

[11] A. Rajaraman, J. D. Ullman, and J. Lescovec. *Mining of massive datasets*, volume 1. Cambridge University Press Cambridge, 2010.

[12] R. Schnell, T. Bachteler, and J. Reiher. Privacy-preserving record linkage using Bloom filters. *BMC Medical Informatics and Decision Making*, 9(1):41, 2009.

[13] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, pages 44–55. IEEE, 2000.

[14] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *Proceedings of the VLDB Endowment*, volume 6, pages 289–300. VLDB Endowment, 2013.

[15] T. Waage, R. S. Jhajj, and L. Wiese. Searchable encryption in Apache Cassandra. In *FPS*, pages 286–293. Springer, 2015.

[16] B. Wang, S. Yu, W. Lou, and Y. T. Hou. Privacy-preserving multi-keyword fuzzy search over encrypted data in the cloud. In *INFOCOM*, pages 2112–2120. IEEE, 2014.