Improving Fuzzy Searchable Encryption with Direct Bigram Embedding

Christian Göge, Tim Waage, Daniel Homann, and Lena Wiese

Georg-August-Universität Göttingen
Institut für Informatik
Goldschmidtstraße 7, 37077 Göttingen, Germany
{christian.goege, waage, homann, wiese}@informatik.uni-goettingen.de

Abstract. In this paper we address the problem of fuzzy search over encrypted data that supports misspelled search terms. We advance prior work by using a bit vector for bigrams directly instead of hashing bigrams into a Bloom filter. We show that we improve both index building performance as well as retrieval ratio of matching documents while providing the same security guarantees. We also compare fuzzy searchable encryption with exact searchable encryption both in terms of security and performance.

Keywords: searchable encryption, similarity search, fuzzy search, semantic security, locality sensitive hashing

1 Introduction

For several applications, users prefer virtual machines on cloud platforms instead of maintaining expensive hardware at their premises. Furthermore, many applications – such as databases – are available *as a service*, where cloud providers facilitate and maintain also the software at their sites. Prominent providers for database-as-a-service products are Amazon Web Services and Microsoft Azure.

In conjunction with outsourcing of private or sensitive data to remote servers comes the need of protecting the data, not only from outside attackers, but also from the service provider, because it might try to learn information from its customers' data and data flow. The simplest solution is to encrypt all data before outsourcing it to a remote location. However, this prevents even simple processing in the cloud. In order to search through the encrypted data, one would have to download the whole database, decrypt it and then run the search on the decrypted data, which obviously ridicules the idea of outsourcing the database in the first place. The solution here is searchable encryption (SE). An SE scheme enables a server to search in encrypted data while preventing it from gaining information about the plaintext data.

Most SE schemes involve a preprocessing step on the data to build an *index*. The index can be built in two shapes: A *document-based*, also called *forward* index [3], relates documents or their unique identifiers to the keywords they contain. This allows a search time of $\mathcal{O}(n)$, where n is the number of documents,

because each document's index is processed during a query. Sublinear query time can be achieved with a *keyword-based* index, also called *inverse* index [3] or *collection-based* index [5]. It relates a keyword to the documents it is contained in. Then, the optimal query time to be achieved is $\mathcal{O}(D(w))$, where $D(w)$ is the number of documents which contain the query term [3]. Updates in a document-based index are easier because the new index entry can simply be added, while in the keyword-based index, updates will require more effort. The choice of the index type depends on the use case: In a write-heavy scenario the former performs better, while in a read-heavy scenario the latter should be chosen [20].

A client can query an encrypted index using a *trapdoor* (an encrypted form of a query). The server can run a *search* algorithm given a trapdoor, and decide which documents contain the query term belonging to the trapdoor. It can then return the encrypted documents without having to see any plaintext. A searchable encryption scheme will however always *leak* some information from the index and the queries. [3] divide the leaked information in three groups:

- *Index information* is the information leaked directly from the stored ciphertexts of the index, as well as the documents. It may include the number of keywords in a document or the database, the total number of documents, their length, identifiers and possibly the similarity between them.
- The *search pattern* captures the information held by two queries returning the same result. With a deterministic trapdoor-generating function, this information is directly leaked, because a query will always compute to the same trapdoor. With non-deterministic trapdoors, the search pattern will at least give the possibility to determine whether two trapdoors were generated from the same query. The search pattern allows the server to possibly gain information about the keywords through statistical analysis. It might also leak similarity between two queries [12].
- The query results yield the *access pattern*, showing which query returned which documents. If a query q returns document x and a second query w returns x and several other documents, the server can learn that q is more restrictive than w.

Most SE schemes choose to leak the search and access pattern on purpose in order to be computationally efficient. A scheme not leaking either of them can be built using *oblivious RAM* [10]; but it requires $\log n$ rounds of communication for one query, where n is the number of documents stored, which is not scalable for large databases [5]. With some inverted indexes like [12], a client can hide the access pattern by outsourcing the index and the encrypted data to two different servers. A query to the index server will return the encrypted set of document identifiers which can then be decrypted and queried from the second server. However, this will always require two rounds of server communication and superior security is only provided if the two servers do not collaborate.

Depending on the use case, SE schemes can be built using symmetric or public key encryption. Symmetric searchable encryption (SSE) [17] is used when one data owner wants to outsource his data collection. Access to the data can be granted to other trusted parties by sharing the secret key among all data users. In

a public key encryption scheme (PEKS) [2], users can add encrypted keywords to an index with a public key and only the private key holder can search on the encrypted data. This is useful to make public-key encrypted documents like emails searchable for the recipient. This paper focuses on a symmetric searchable encryption scheme that additionally allows *fuzzy search*. Fuzzy search is the ability to find documents containing terms the spelling of which is similar to a query term – a key feature known from web search engines like Google. Taking the scheme in [12] as our baseline we make the following contributions:

- When embedding bigrams into a metric space, we replace the hash-based approach of [12] by a direct bigram vector.
- We show that our approach achieves better query performance and retrieval success for misspelled queries.
- In addition, we make a comparison between our scheme, the original fuzzy searchable encryption scheme in [12] and the exact searchable encryption scheme in [18], and hence analyze the impact of the fuzzy search functionality in a unified framework.

The paper is outlined as follows: Section 2 presents related work. Our approach is based on the work of Kuzu et al. [12]; this scheme is described in detail in Section 3. Our own contribution is described in Section 4. The security of our approach is analyzed in Section 5. Section 6 provides details of the implementation and presents the comparative results based on our implementations of the schemes in a common framework. Section 7 concludes the paper.

2 Related Work

A recent survey of the field of searchable encryption can be found in [3]. Here, we will concentrate on important approaches for exact symmetric searchable encryption as well as encrypted fuzzy search.

The first approach addressing searchable encryption was proposed by Song et al. [17]. They use a two-layered encryption construct that makes a sequential search of the ciphertext possible and hence does not require an index. Because both, encryption and search, need to iterate over the whole collection, this scheme does not scale for large databases. It also relies on fixed-size words.

Goh [9] was the first to introduce a *secure index* built on individual documents. For each document, it uses a Bloom filter that holds all keywords extracted from the document. A query has to check set membership of a string in each index Bloom filter, which leads to a constant lookup time for one document and results in a linear query time on the document collection.

Curtmola et al. [5] developed a keyword-based index consisting of an array of linked list nodes. It allows for optimal query time but the encryption is costly, because it has to encrypt all nodes separately.

Stefanov et al. [18] use a keyword-based index stored in a set of exponentially growing hashmaps. The main advantage of the scheme is that its leakage is considerably lower than the leakage of other schemes – the authors call this

"forward security". The disadvantages of the scheme are the complexity of the data structure as well as the search time which is, although sublinear, quite high.

The above approaches only find exact matches. We now move on to the topic of *fuzzy* encrypted search. The term fuzzy or similarity search is used in the literature with two different meanings. The first meaning covers search schemes which return documents which contain a subset of a given keyword set. Schemes for this direction of research are given by [19, 23, 8]. In contrast, we use the term similarity search as the search for a single keyword, which may be misspelled, but nevertheless can be resolved to the correct documents.

Several fuzzy search schemes are based on the idea of considering all keywords in a certain edit distance of a keyword [13, 14, 22, 11]. In order to decrease the required storage space, they do not generate all possible keywords but introduce a special wildcard character which stands for any single letter of the alphabet.

Wang et al. [21] achieve fuzzy queries for multiple keywords at once. Their index is a Bloom filter for each document. Keywords are inserted into the index with locality-sensitive hashing (LSH) functions, which makes similar words likely to hit the same indices in the Bloom filter. Since the Bloom filters are large, encrypting them and computing their score without decrypting them involves a large matrix and matrix-vector multiplication. In order to allow multiple fuzzy keyword search, this scheme pays with a large asymptotic constant to the $\mathcal{O}(n)$ query time.

Boldyreva and Chenette [1] provide two formal definitions of fuzzy search which are very strict with regard to result quality and leakage. They show that there can not exist a space-efficient scheme fulfilling the first definition and give a fuzzy scheme for fingerprint data satisfying the second definition.

Chua et al. [4] also use Bloom filters for the encoding of the keywords and store them in a special tree structure for efficient search.

3 Background

Formally, a searchable encryption scheme (SE) provides the key functions

- *Keygen(s):* Computes the master key K_{priv} given the security parameter s.
- *Trapdoor(K_{priv}, q):* Computes the trapdoor T_q for query q with the key K_{priv}.
- *BuildIndex(D, K_{priv}):* Computes the index I_D for the document collection D.
- *Search(I_D, T_q):* Outputs a set of document identifiers, for which the documents contain the query term q.

Our approach is based on prior work by Kuzu et al. [12]. The *search* is relaxed such that it allows retrieval of documents within a certain distance to the query.

- *FuzzySearch:* Let D be a collection of documents consisting of features f. A query for a feature f_q returns a document $D_i \in D$ with high probability if $\exists f \in D_i : dist(f, f_q) < \alpha$. Furthermore, the query will not return documents if $\forall f \in D_i : dist(f, f_q) > \beta$.

We now formally describe the fuzzy search approach in [12].

Locality Sensitive Hashing. Locality sensitive hashing (LSH) is used to efficiently approximate distances between two terms. Let M be a metric space with distance function $d : M \rightarrow \mathbb{R}$. A family $\mathcal{H}$ of hash functions $h : M \rightarrow S$ is called (r_1, r_2, p_1, p_2)-sensitive, if for all $h \in \mathcal{H}$ and arbitrary $x, y \in M$:

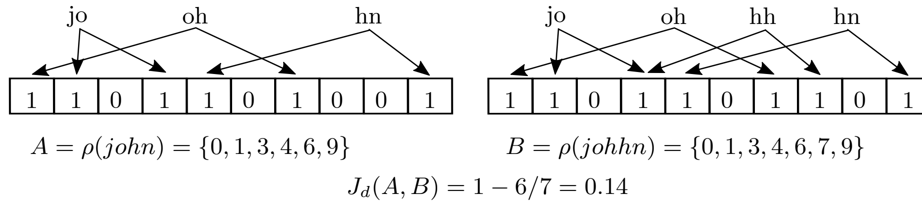
$$d(x, y) \leq r_1 \Rightarrow Pr[h(x) = h(y)] \geq p_1$$
$$d(x, y) \geq r_2 \Rightarrow Pr[h(x) = h(y)] \leq p_2$$

with $r_1 < r_2$, $p_1 > p_2$. Given a similarity function $\phi : U \times U \rightarrow [0, 1]$ between elements x, y of a universe U, a LSH family has the property $Pr[h(x) = h(y)] = \phi(x, y)$. The probabilities can be amplified to match the *FuzzySearch* thresholds α, β by combining several uniformly chosen functions $h \in \mathcal{H}$. Let $Pr[h(x) = h(y)] = p$. Kuzu et al. form a $(r_1, r_2, (1 - p_1^k)^\lambda, (1 - p_2^k)^\lambda)$-sensitive family $\mathcal{F}$ (as in [15]) by combining the basic hash functions with logical AND and OR:

$$g_i(x) = h_{i_1}(x) \wedge \cdots \wedge h_{i_k}(x) \tag{1}$$
$$f(x) = g_1(x) \vee \cdots \vee g_\lambda(x), \tag{2}$$

in the sense that $g_i(x) = g_i(y)$ iff $\forall j : h_{i_j}(x) = h_{i_j}(y)$ and $f(x) = f(y)$ iff $\exists i : g_i(x) = g_i(y)$.

Metric space embedding ρ. The set F of features of a document consists of all the words it contains. The use of LSH to efficiently approximate the distance between two words requires a distance function for which such approximation functions are known. This is not the case for the well known Levensthein or Edit distance [12]. Therefore the documents' features have to be embedded in a metric space first. Kuzu et al. [12] embed them in the metric space of sets with an embedding function ρ, that is presented in [16]: The strings are identified by their bigrams, that are the continuous substrings of length two extracted from the string. For the word "john", the bigram set is {jo, oh, hn}. The bigrams are then inserted into a Bloom filter of length $m = 500$ with $k = 15$ cryptographic hash functions. The Bloom filters are interpreted as sets and the distance is measured using Jaccard distance $J_d(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$ (see Fig. 1).



$A = \rho(john) = \{0, 1, 3, 4, 6, 9\}$ $B = \rho(johhn) = \{0, 1, 3, 4, 6, 7, 9\}$

$J_d(A, B) = 1 - 6/7 = 0.14$

Fig. 1. Hashed bigram embedding of length 10 with 2 hash functions

Minhash. The minhash functions form a $(r_1, r_2, 1-r_1, 1-r_2)$-sensitive LSH family for the Jaccard distance. Let Δ be the ordered domain of the set, i.e. $\Delta = \{1, \ldots, m\}$. A minhash function is defined with a random permutation $P(\Delta)$ on the domain Δ. Let $X \subset \Delta$. The minhash of X is then

$$h_P(X) = \min_{i=0,\ldots,m} \{i \mid P[i] \in X\}. \tag{3}$$

For example, let $\Delta = [0, \ldots, 9]$, $P(\Delta) = \{3, 4, 1, 5, 2, 6, 7, 9, 8, 0\}$ and $X = \rho(\text{john}) = \{0, 1, 2, 4, 5, 8, 9\}$. Then, the first index i in P that is a member of X is $P[1] = 4$, hence $h_P(X) = 1$. For two sets X, Y, the probability that $h_P(X) = h_P(Y)$ equals the Jaccard similarity $J = \frac{|A \cap B|}{|A \cup B|}$. In practice, a permutation on Δ can be defined with a simple hash function $h_\Delta : [1, m] \rightarrow [1, m]$ that pseudo-permutes the domain. The minhash can then be efficiently computed as

$$h(x) = \min_{x \in X} h_\Delta(x). \tag{4}$$

Index. The index is built with keyword subfeatures extracted with the LSH functions g as described by Eq. (1) and (2). The LSH parameters k, λ are chosen beforehand and the λ subfeatures sort the documents into buckets. If two documents appear in the same bucket (i.e. have a keyword subfeature in common), they are likely to both contain the keyword. All documents D_i to be inserted into the index are labelled from 1 to n. For each document, the features are the words contained in it. The features are embedded into the metric space of sets with the embedding ρ. Its λ subfeatures are extracted using the LSH functions $(g_1(\rho(f)), \ldots, g_\lambda(\rho(f))$; each subfeature is a bucket identifier in the index: $B_k = g_i(\rho(f))$. For each bucket, a bit vector V_{B_k} of length n is stored. If the feature f that yielded the bucket B_k was extracted from document D_i, then $V_{B_k}[i] = 1$. Both the bucket identifiers and the bit vectors are encrypted with two different secret keys to form the encrypted index, where $\pi_{B_k} = Enc_{K_{id}}(B_k)$ is a pseudorandom permutation and $\sigma_{V_{B_k}} = Enc_{K_{pay}}(V_{B_k})$ is PCPA-secure [1]. In the end, a number of random fake records (R_1, R_2) with $|R_1| = |\pi_{B_k}|$ and $|R_2| = |\sigma_{V_{B_k}}|$ are inserted to keep the index at the constant size $MAX \cdot \lambda$, where MAX is the maximum number of features.

Query. A query requires two rounds of server communication. The query feature f is embedded in the metric space with the same embedding function ρ and then subfeatures are extracted and encrypted with the same functions as in the index construction. The query trapdoor is then $T_q = (Enc(g_1(\rho(f))), \ldots, Enc(g_\lambda(\rho(f))))$. The trapdoors are queried from the server who returns the encrypted bit vectors $Enc(V_{B_k})$ for the issued trapdoor. The bitvectors are added up to obtain a score between 0 and λ for each document. In a second round, we can request the top t scored documents from the server via their ids.

[1] PCPA-security: An encryption scheme is PCPA secure (pseudo–randomness against chosen plaintext attacks) if the ciphertexts are indistinguishable from a random. [5]

4 Direct bigram embedding

Using the original embedding of [12], there is always the possibility of collisions in the Bloom filter, meaning that because of the involved hashing two different bigrams can yield the same position to be 1 in the representing set and hence making strings similar even if they are not. The security of this scheme does not rely on the cryptographic properties of the embedding, since the output is encrypted again, so we can replace this embedding with a new one.

We found that we can improve the retrieval of misspelled documents significantly by using another metric space embedding. We adopt the approach of Wang et al. [21] who also use LSH for string distance but use a bigram vector instead of a Bloom filter. The bigram vector has the maximum size 26^2; every position in the vector accounting for one bigram. If a bigram is present in the string, the vector is set to one at the according position.

hn jo oh — hh hn jo oh

$(0, ..., 0, 1, 0, ..., 0, 1, 0, ..., 0, 1, 0, ..., 0)$ $(0, ..., 0, 1, 0, ..., 0, 1, 0, ..., 0, 1, 0, ..., 0, 1, 0, ..., 0)$

$A = \rho(john) = \{195, 248, 371\}$ $B = \rho(johhn) = \{189, 195, 248, 371\}$

$J_d(A, B) = 1 - 3/4 = 0.25$

Fig. 2. Hashed bigram embedding of length 10 with 2 hash functions

More formally we proceed as follows: The keywords are converted to lower case. Letters are identified by their alphabetical order with $ord(\text{'a'}) = 0$ to $ord(\text{'z'}) = 25$. The position of a bigram $\alpha_1\alpha_0$ in the vector is then $pos(\alpha_1\alpha_0) = ord(\alpha_1) \cdot 26 + ord(\alpha_0)$. Finally, we set the bigram vector at the according position to one: $v[pos] = 1$. Most importantly, we represent the vector as the set of indices set to 1, making the representation as compact as possible. That makes our embedding also smaller in memory than Kuzu's, who has to store (at most) $k \cdot n$ positions while we store n positions. (k is the number of hash functions Kuzu uses, n the number of bigrams of the word).

While Wang et al. [21] use euclidean distance between the bigram vectors, we can clearly interpret the bigram vector as the set of indices which are set to one. This allows us to keep using the Jaccard distance. Fig. 2 pictures the bigram vector. Note that in comparison to the Bloom filter embedding (see Fig. 1), the distances between the words become larger. This allows us to choose smaller LSH parameters k and λ (see Eq. (1) and (2)), resulting in fewer LSH functions to be computed per keyword.

As mentioned before, the original embedding can produce collisions in the Bloom filter and thus can make non-similar strings look similar. This has indeed a great impact on the minhash LSH subfeatures that form the buckets in the index construction. We found that with our bigram embedding without such collisions, we produce twice as many distinct subfeatures from the keywords as with the original embedding. With this we achieve an increase in the search

success, because in our larger index the subfeatures of different terms have less collisions. More details are provided in Section 6.

5 Security analysis

The server is assumed to be *honest-but-curious*, meaning that it will carry out its tasks as it is expected, but tries to learn about the data it hosts. Note that when assuming more malicious attackers, the security of our or any other searchable encryption scheme might not be maintainable. For example, Zhang et al. [24] consider a server that sometimes injects files in the index. This leads to attacks in which the server can quite easily figure out the content of the user's files.

Our scheme has still the same security properties as the original scheme: it has the adaptive semantic security property (see Def. 7). Informally this definition means, that exactly the following information is leaked to an adversary:

- Search pattern: Hashes of searched keywords
- Access pattern: Document identifiers matching queries and document identifiers of added or deleted documents
- Similarity pattern: Similarity between the encrypted queries

5.1 Leaked Information

In an optimal world, the query process will not leak any information, not even which item corresponds to which query. While this can be achieved by *Oblivious RAM* [10], it is computationally expensive and not suitable for large databases. Secure indexes leak some information on purpose, in order to achieve linear or constant query times [5] in the number of documents contained in the database.

The trapdoor generating function is a deterministic function, that means a query q will always compute to the same trapdoor T_q. That allows the adversary to see the search pattern, e.g. which trapdoors are requested how often.

Definition 1. ***Search Pattern*** π *([12], III-B.1): Let* $\{f_1, \ldots, f_n\}$ *be the feature set for* n *consecutive queries. The search pattern* π *is a binary symmetric matrix with* $\pi[i, j] = 1$ *if* $f_i = f_j$ *and* 0 *otherwise.*

The deterministic trapdoors also yield the connection between a query trapdoor T and the returned documents. This is captured by the access pattern.

Definition 2. ***Access Pattern*** A_P *([12], III-B.2): Let* $D(f_i)$ *be a collection that contains the identifiers of data items with feature* f_i *and* $\{T_1, \ldots, T_n\}$ *be the trapdoors for the query set* $\{f_1, \ldots, f_n\}$*. Then, the Access Pattern is defined as the matrix* $A_p(T_i) = D(f_i)$*.*

Kuzu's [12] FuzzySearch algorithm extracts subfeatures from a query feature via LSH, making a query for feature f consist of λ subfeatures. The number of shared subfeatures for two queries is an indicator for the similarity between them. This information is also leaked and captured in the similarity pattern. Note that this definition naturally includes the search pattern π.

Definition 3. ***Similarity Pattern*** S_P *([12], III-B.3): Let* $\{f_i^1, \ldots, f_i^\lambda\}$ *be the subfeatures of feature* f_i*. For* n *queries,* $\{(f_1^1, \ldots, f_1^\lambda), \ldots, (f_n^1, \ldots, f_n^\lambda)\}$ *is the feature set. Let* $i[j]$ *define the* j^{th} *subfeature of feature* f_i*. Then, the similarity pattern is* $S_P[i[j], p[r]] = 1$ *if* $f_i^j = f_p^r$ *and* 0 *otherwise for* $1 \leq i, p \leq n$ *and* $1 \leq j, r \leq n$*. In other words, it contains all matrices* $S_{i,p}$ *capturing the similarity between features* f_i *and* f_p *by setting* $S_{i,p}[j, r] = 1$ *if* f_i *and* f_p *share subfeatures* f_i^j *and* f_p^r*.*

The similarity pattern is derived from the queries posed to the encrypted index. The server is not able to deduce similarity between documents from the index alone, nor can he see if the returned documents were a fuzzy or exact hit for a query. Therefore, this fuzzy scheme makes it harder to deduce document similarity from queries than an exact scheme with comparable index structure does, because in an exact scenario, the link between (encrypted) query and returned document (the *access pattern*) is present, while in our fuzzy scheme it is covered with the uncertainty whether the returned document *really* includes the query. The access pattern can be hidden completely by using two different (non-collaborating) servers to store the index and the document collection.

In order to capture the information an adversary has, we first define a sequence of n consecutive queries as an n*-query history*. The *trace* $\gamma(H_n)$ captures the maximum amount of data that is purposely allowed to leak from the secure index scheme, meaning the maximum amount of information that should be computable from what an adversary sees from an n-query history. The data visible to the attacker is called the *view*.

Definition 4. ***History*** H_n *([12], III-B.4): Let* D *be the data collection and* $Q = \{f_1, \ldots, f_n\}$ *the features for* n *consecutive queries. Then* $H_n = (D, Q)$ *is an* n*-query history.*

Definition 5. ***Trace*** $\gamma(H_n)$ *([12], III-B.5): Let* C *be the collection of encrypted documents,* $id(C_i)$ *the identifiers and* $|C_i|$ *the size of the encrypted documents. The trace* $\gamma(H_n) = \{(id(C_1)...id(C_l)), (|C_1|, \ldots, |C_l|), S_p(H_n), A_p(H_n)\}$ *is the maximum amount of information that is allowed to leak.*

Definition 6. ***View*** $v(H_n)$ *([12], III-B.6): Let* C *be the collection of encrypted documents,* $id(C_i)$ *the identifiers,* I *the secure index and* T *the trapdoors of the history* H_n*. All the information seen by an adversary is captured by the view* $v(H_n) = \{(id(C_1), \ldots, id(C_l)), C, I, T\}$*.*

In contrast to the scheme of Kuzu, the scheme of Stefanov et al. [18] does not allow for similarity search. As the authors show, only the search pattern and the access pattern as defined above are leaked. The fact that their scheme allows dynamic updates does not affect the leakage; they call this forward security.

5.2 Semantic Security

The general idea of the security definition is to play a game against the attacker, who is modelled as a probabilistic polynomial time (p.p.t.) algorithm. If the

attacker has a probability of winning not better than a coin toss win ($\frac{1}{2}$), then the scheme is secure. This is formulated as a simulator based definition, where a p.p.t. simulator can generate a random *view* $v_S(H_n)$ to the real query view $v_R(H_n)$ only using information available in the *trace* $\gamma(H_n)$ and the adversary again has a probability not greater than $\frac{1}{2}$ of deciding which view is the real one.

The definition was given by Curtmola et al. [5] and widely adopted afterwards (e.g., [12, 21]). They distinguish between two types of adversaries: non-adaptive (IND-CKA1) and adaptive (IND-CKA2). The non-adaptive adversary generates queries (the *history*) without taking into account information he might have learned from previous queries. This is of course rarely the case in a real world scenario where the secure index scheme is running on a server, and the server is the adversary. The adaptive adversary can generate his queries adaptively during his examination.

Definition 7. ***Adaptive Semantic Security*** *([5], 4.1) An SSE scheme is secure if for all p.p.t. adversaries A there is a p.p.t. simulator S which can adaptively construct a view $v_S(H_n)$ from the trace $\gamma(H_n)$ such that the adversary cannot distinguish between the simulated view $v_S(H_n)$ and the real view $v_R(H_n)$.*

More formally, the scheme is secure according to the security definition if one can define a simulator S such that for all polynomial time attackers A holds $\Pr(A(v(H_n)) = 1) - \Pr(A(S(\gamma(H_n))) = 1) < p(s)$, where s is a security parameter and $p(s)$ a negligible function.[2] The probability is taken over all possible H_n and all possible encryption keys.

5.3 Security Proof

The proof that our adaptation of Kuzu et al.'s algorithm is still secure according to the given definition is analogous to the original paper. Let $v_R(H_n)$ and $\gamma(H_n)$ be the real view and the trace. Then a p.p.t. simulator S can adaptively generate the simulated view $v_S(H_n) = \{(id(C_1)^*, \ldots, id(C_l)^*), (C_1^*, \ldots, C_l^*), I^*, (T_{f_1}^*, \ldots, T_{f_n}^*)\}$ as follows:

- Identifiers of documents can simply be copied since they are available in the trace. Hence, both identifier lists in v_S and v_R are identical.
- S can choose n random values $\{C_1^*, \ldots, C_l^*\}$ with $|C_i^*| = |C_i|$. Since the C_i result from an encryption scheme which is pseudo-random against chosen-plaintext attacks (PCPA) [5], they are computationally indistinguishable from random values.
- Let π_{B_k} and $\sigma_{V_{B_k}}$ be the encrypted bucket id and encrypted bucket content of the index. S chooses $MAX \cdot \lambda$ random pairs (R_{i_1}, R_{i_2}) with $|R_{i_1}| = |\pi_{B_k}|$ and $|R_{i_2}| = |\sigma_{V_{B_k}}|$ and inserts them into I^*, where MAX is the maximum number of features and λ is the number of components of a trapdoor. Since π_{B_k} is a pseudorandom permutation and $\sigma_{V_{B_k}}$ is PCPA-secure, pairs in I^* are computationally indistinguishable from pairs in I. Since both contain

[2] p is negligible if for all positive polynomials f holds: $p(x) < 1/f(x)$ [5].

$MAX \cdot \lambda$ records by construction, I^* is computationally indistinguishable from I.

– The trapdoors $(T^*_{f_1}, \ldots, T^*_{f_n})$ can be constructed from the similarity pattern S_p. They are filled such that $T_i[j]^* = T_p[r]^*$ if $S[i[j], p[r]] = 1$ and otherwise $T_i[j]^* = R_{i_j}$, where R_{i_j} is a random value with $|R_{i_j}| = |\pi_{B_k}|$ and $R_{i_j} \neq R_{p_r} \forall 1 \leq p < i$ and $1 \leq r < \lambda$. Again, the simulated trapdoors are indistinguishable from the real encrypted trapdoors because they are computed by pseudorandom permutation. Also they show the same similarity pattern as the real trapdoors by construction.

Since the components of v_S and v_R are computationally indistinguishable, the scheme satisfies the security definition.

6 Implementation and Results

Even if the cloud databases use encryption at the server side, the plaintexts would be sent to the server, which undermines our goal of security. Therefore, the index generation and encryption of documents has to take place at the client side. While this burdens all computation to the client, it has the advantage that we can easily connect to all existing databases in the cloud without the need of altering one of the database's implementation. We can also seperate index and the data storage to two different servers. Provided that the servers do not collaborate, this also hides the access pattern.

We implemented the scheme of Kuzu et al. [12] and our improvements in Java 1.8[3]. For keyword extraction, we used the Apache Lucene (6.2.1) classes *StandardTokenizer* together with *Standard-, Stop-* and *EnglishPossessiveFilter*. We continued filtering all remaining words containing numbers. The minhashes were computed using the implementation available at [6]. In the implementation of the LSH algorithm, we define g_i by $g_i(x) = \sum_{j=1}^{k} h_{i,j}(x) \cdot m^{j-1} \mod 2^{64}$, with $h_{i,j}$ being the jth minhash function of g_i and m the length of the metric space embedding. For encryption we used the AES-128 implementation available in *javax.crypto*.

To achieve similar results as in the original paper [12], the same testing setup was applied. 5000 mails were randomly selected from the publicly available Enron email dataset [7]. The features describing a mail are chosen as the words in the mail's body. In Kuzu et al.'s paper, they are embedded into Bloom filters of length 500 using 15 hash functions by hashing the bigrams of a feature (the authors adopt these settings from [16]). To determine the fuzzy search thresholds for LSH, Kuzu et al. measured the distances between keywords and possible misspellings. Their resulting minhash LSH algorithm uses $\lambda = 37$ stages of length $k = 5$, building a $(0.45, 0.8, 0.85, 0.01)$-sensitive LSH family.

For keyword misspells, we randomly introduced one of the following errors with equal probability: (1) deleting a random letter, (2) doubling a random letter and (3) switching two adjacent letters. Analyzing the keyword–misspell

[3] The implementation can be found at https://github.com/dbsec/FamilyGuard

distances with the new bigram embedding shows that dissimilar words are further apart than in Kuzu et al.'s original embedding. To account for that, we choose $k = 4$, $\lambda = 25$ to build a $(0.5, 0.85, 0.85, 0.01)$-sensitive family. This also has the advantage of a smaller number of minhashes $(k \cdot \lambda)$ to be computed.

The performance results were obtained by running the schemes on a PC with an Intel i5-6500 CPU with 16 GB RAM and a Samsung EVO 840 SSD.

6.1 Retrieval

For retrieval evaluation we choose *precision* and *recall* as a metric, which is often used in information retrieval. Let w denote a keyword. A query for w is denoted with w'. If the query introduces a spelling error, then $w' \neq w$. Let $D(w)$ be the number of documents containing word w. A document D_i is correctly retrieved for a query w' if $w \in D_i$. Let $R^{D(w')}$ be the retrieved set of documents for query w' and $R^{D(w)}$ the subset that was correctly retrieved. Then precision and recall are defined as

$$prec(w') = \frac{|R^{D(w)}|}{|R^{D(w')}|}, \quad rec(w') = \frac{|R^{D(w)}|}{|D(w)|}. \tag{5}$$

Kuzu et al. [12] misspell 25% of 1000 randomly chosen queries and return the t top-scored documents, while changing t. We believe that this approach has two problems: First, since not misspelled queries will always compute to a maximum score of λ, it is easy to return all exact hits with a precision of 0.99. (There might be two different words having the same bigrams, therefore it is not exactly 1). Second, the recall will depend on the choice of t. If we fix t too low, there might be much more relevant documents than we requested.

Because of these reasons, we choose in our evaluation to misspell all queries. We uniformly selected a total of 10,000 queries from the set of all extracted keywords and introduced one random spelling mistake mentioned above. We then measured precision and recall not depending on a fixed t, but by returning all documents with a score greater than x for $x \in \{\lambda, \dots, 1\}$. The result in Fig 3 shows the results. On average, a query had 13 relevant documents to it. With our Bigram embedding, we achieve at best a precision of 0.36 with a recall of about 0.5, which is achieved returning all documents with a score greater than 6. Compared to Kuzu et al.'s embedding this is a significant improvement.

6.2 Performance and Comparison

First, we wanted to compare the time needed to compute the query trapdoors for our bigram embedding compared to Kuzu's. Therefore, for words with n bigrams for $n = 1$ to 20, we computed 100,000 trapdoors each. Fig. 4 shows the averaged time needed to compute one trapdoor. As said before, our embedding is smaller in size, because we only compute one position for a bigram instead of k hashes. Also, our embedding lets us choose a LSH family with fewer functions. The Kuzu measurement shows a slight curve, this can be explained by collisions

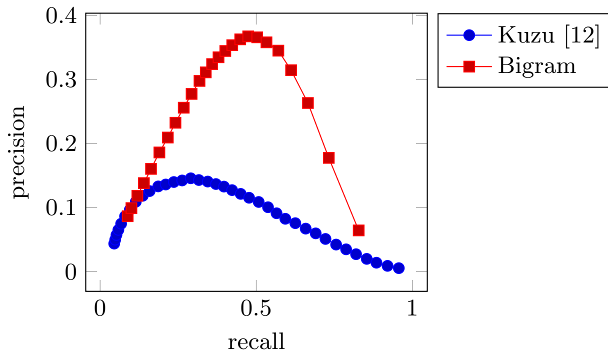


Fig. 3. Retrieval success

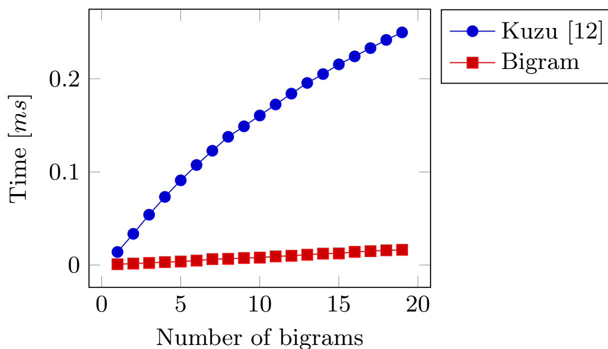


Fig. 4. Performance: Trapdoor computation

in the bloom filter: The more bigrams we hash, the more collisions happen and therefore the number of ones in the filter does not grow linearly.

We then measured the time needed to construct the index for 1,000 to 10,000 mails taken randomly from the Enron dataset [7]. The times are averaged over three index constructions each. We then query the indexes with 1000 keywords randomly selected from the documents. We compare the performance of our contribution to Kuzu et al.'s original scheme. Additionally, we compare the performance to an exact-keyword matching scheme of Stefanov et al. [18].

The results are shown in Fig. 5. All schemes build an inverted index. Comparing our scheme to Kuzu et al.'s original, we find that the index construction performs slightly worse. Because of the properties of the embedding (see Sec. 4), we find that our index is larger than Kuzu's. Therefore we have to encrypt about twice as many index entries. Since this index construct is built to be computed only once and performs badly with updates anyway, this increase is insignificant. The scheme of Stefanov et al. [18] builds the index such that online updates are still possible, but follows the same assumption that updates are infrequent. This flexibility leads to larger index creation times. The scheme partially rebuilds

the index on updates (with an amortized cost of $\mathcal{O}(\log^2 N)$); the decryption of the entries, oblivious sorting and reencryption introduce large constants in this asymptotic runtime. The scheme hence needs more time than [12].

For query performance, the exact scheme only has to do a lookup operation followed by collecting all entries and therefore is the fastest. The fuzzy search algorithms pay for the similarity search with 2–3 times slower queries because the query process also involves addition of at most λ bit vectors of length equal to the size of the document collection. Our approach outperforms Kuzu et al.'s scheme by about 20%. This is due to the fact that the bigram embedding is computed faster and we can choose an LSH family with fewer minhash functions.

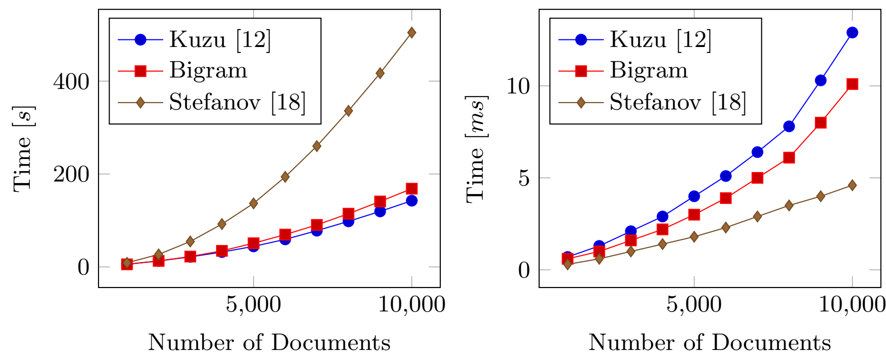


Fig. 5. Performance: Index construction (left) and queries (right)

7 Conclusion

We provided an improvement to the encrypted fuzzy search scheme of Kuzu et al. [12] by replacing the metric space embedding for strings based on a Bloom filter with a direct bigram embedding used in [21]. We showed that this improves the retrieval success of misspelled queries significantly, and makes the queries 20% faster. The performance of both the original and the modified fuzzy search scheme was also compared to the performance of an exact searchable encryption scheme [18]; the results show that there is only a modest overhead for the additional feature of fuzzy search.

Acknowledgement. This work was partially funded by the DFG under grant number Wi 4086/2-2.

References

1. Boldyreva, A., Chenette, N.: Efficient fuzzy search on encrypted data. In: FSE. vol. 8540, pp. 613–633. Springer (2014)

2. Boneh, D., Di Crescenzo, G., Ostrovsky, R., Persiano, G.: Public key encryption with keyword search. In: Eurocrypt. pp. 506–522. Springer (2004)
3. Bösch, C., Hartel, P., Jonker, W., Peter, A.: A survey of provably secure searchable encryption. ACM Computing Surveys (CSUR) 47(2), 18 (2015)
4. Chuah, M., Hu, W.: Privacy-aware bedtree based solution for fuzzy multi-keyword search over encrypted data. In: ICDCS. pp. 273–281. IEEE (2011)
5. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. IACR Cryptology ePrint Archive 2006(Rep. 210) (2006)
6. Debatty, T.: java-lsh. https://github.com/tdebatty/java-LSH
7. Enron email dataset. https://www.cs.cmu.edu/~./enron/ (2015)
8. Fu, Z., Shu, J., Wang, J., Liu, Y., Lee, S.: Privacy-preserving smart similarity search based on simhash over encrypted data in cloud computing. Journal of Internet Technology 16(3), 453–460 (1971)
9. Goh, E.J.: Secure indexes. IACR Cryptology ePrint Archive 2003 (2004), rep. 216
10. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. Journal of the ACM (JACM) 43(3), 431–473 (1996)
11. Hu, C., Han, L.: Efficient wildcard search over encrypted data. Int. J. Inf. Sec. 15(5), 539–547 (2016)
12. Kuzu, M., Islam, M.S., Kantarcioglu, M.: Efficient similarity search over encrypted data. In: Data Engineering (ICDE) 2012. pp. 1156–1167. IEEE (2012)
13. Li, J., Wang, Q., Wang, C., Cao, N., Ren, K., Lou, W.: Fuzzy keyword search over encrypted data in cloud computing. In: INFOCOM. pp. 441–445. IEEE (2010)
14. Liu, C., Zhu, L., Li, L., Tan, Y.: Fuzzy keyword search on encrypted cloud storage data with small index. In: IEEE CCIS. pp. 269–273 (2011)
15. Rajaraman, A., Ullman, J.D., Lescovec, J.: Mining of massive datasets, vol. 1. Cambridge University Press Cambridge (2010)
16. Schnell, R., Bachteler, T., Reiher, J.: Privacy-preserving record linkage using Bloom filters. BMC Medical Informatics and Decision Making 9(1), 41 (2009)
17. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: IEEE Symposium on Security and Privacy. pp. 44–55. IEEE (2000)
18. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: NDSS. vol. 14, pp. 23–26 (2014)
19. Sun, W., Wang, B., Cao, N., Li, M., Lou, W., Hou, Y.T., Li, H.: Privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking. In: ACM SIGSAC. pp. 71–82. ACM (2013)
20. Waage, T., Jhajj, R.S., Wiese, L.: Searchable encryption in Apache Cassandra. In: FPS. pp. 286–293. Springer (2015)
21. Wang, B., Yu, S., Lou, W., Hou, Y.T.: Privacy-preserving multi-keyword fuzzy search over encrypted data in the cloud. In: INFOCOM. pp. 2112–2120. IEEE (2014)
22. Wang, C., Ren, K., Yu, S., Urs, K.M.R.: Achieving usable and privacy-assured similarity search over outsourced cloud data. In: INFOCOM. pp. 451–459. IEEE (2012)
23. Yuan, X., Cui, H., Wang, X., Wang, C.: Enabling privacy-assured similarity retrieval over millions of encrypted records. In: ESORICS. pp. 40–60. Springer (2015)
24. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: The power of file-injection attacks on searchable encryption. In: USENIX. pp. 707–720 (2016)