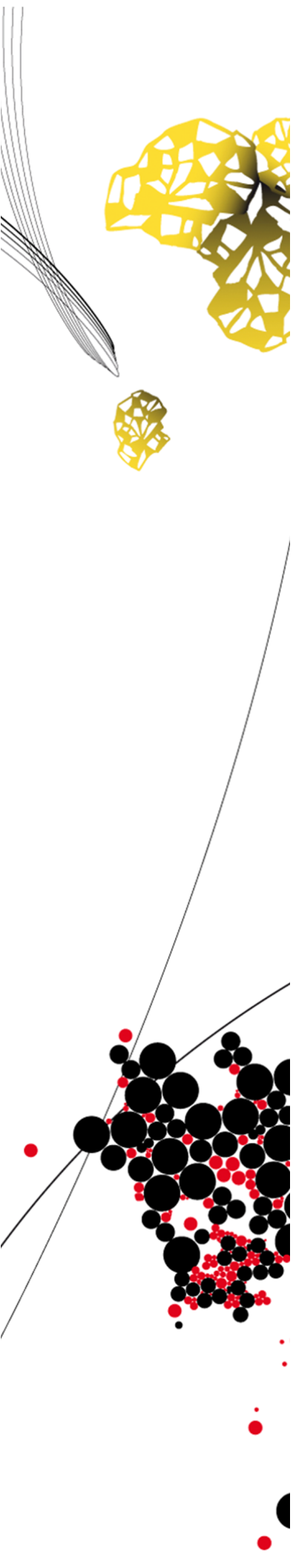

UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering,
Mathematics & Computer Science**

Experimental Review of the IKK Query Recovery Attack: Assumptions, Recovery Rate and Improvements

Ruben Groot Roessink

M.Sc. Thesis
July 2020

Supervisors:
Dr. A. Peter
Dr. Ing. F.W. Hahn
Dr. Ir. M. Jonker

Services and Cyber-Security
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

***Abstract*—In light of more data than ever being stored using cloud services and the request by the public for secure, privacy-enhanced, and easy-to-use systems, Searchable Encryption schemes were introduced. These schemes enable privacy-enhanced search among encrypted documents yet disclose (encrypted) queries and responses. The first query recovery attack, the IKK attack, uses the disclosed information to (partly) recover what plaintext words the client searched for. This can also leak information on the plaintext contents of the encrypted documents. Under specific assumptions, the IKK attack has been shown to potentially cause serious harm to the security of Searchable Encryption schemes.**

We empirically review the IKK query recovery attack to improve the understanding of its feasibility and potential security damage. In order to do so, we vary the assumed query distribution, which is shown to have a severe (negative) impact on the accuracy of the attack, and the input parameters of the IKK attack to find a correlation between these parameters and the accuracy of the IKK attack.

Furthermore, we show that the recovery rate of the attack can be increased up to 10 percentage points, while decreasing the variance of the recovery rate up to 78 percentage points by combining the results of multiple attack runs. We also show that the including deterministic components in the probabilistic IKK attack can increase the recovery rate up to 21 percentage points and decrease its variance up to 57 percentage points.

***Index Terms*—Searchable Encryption, IKK, query recovery**

I. Introduction

The use of currently available encryption schemes allows users to securely upload and retrieve documents anywhere in the world using cloud services. A user encrypts a set of documents and sends these encrypted documents to a server for storage. The server can return documents upon request by the user, which the user can decrypt to obtain the original documents, while the server is not capable of reading the contents of the documents.

A downside of using encryption schemes is that they, in general, limit the functionalities of the cloud service. One such functionality is the possibility to search for word occurrences among documents. To overcome this loss in functionality, while also taking into regard data confidentiality, Song et al. [1] introduced the notion of Searchable Encryption (SE).

In general, SE schemes provide a client with a way to search for the occurrence of a certain (plaintext) word, a *keyword*, among a set of encrypted documents, while neither the client nor the server has to decrypt all documents which the client wants to search among. A client generates a search token, a *query*, which it sends to a server hosting a set of (encrypted) documents. The server uses the query to find a subset of the encrypted documents corresponding with the search token and returns this subset to the client.

Nearly all of these SE schemes leak at least some information, usually in the form of *data access patterns* [1]–[3], meaning that an adversary can observe the issued queries from the client and the document identifiers of documents corresponding to said queries in the response by the server and is thus capable of making a connection between the queries and the corresponding documents.

Some schemes were proposed [3], [4] that hide these access patterns. However, these schemes are quite inefficient as they require an extensive number of computations after each query. Other schemes propose to obfuscate the access patterns which can both lead to inconsistencies in the search results (false positives or false negatives) and an increase in communication and storage overhead [5].

Islam et al. [6] elaborate on the implications of the leakage of access patterns by proposing the first *query recovery attack*, dubbed IKK attack in subsequent research after the first initials of the authors of the paper. Their attack is a statistical attack which tries to map queries to their corresponding real-world keywords. This mapping process is dubbed *query recovery*. A correctly 'recovered' query tells an adversary what the client searched for and possibly even tells something about the contents of (encrypted) documents stored on the server.

In their attack Islam et al. use the relative co-occurrence counts of queries, which denotes the number of documents a certain number of queries occur in together, relative to the total number of documents. These counts can be calculated from leaked access patterns. The IKK attack also assumes the adversary has access to (a close approximation of) the co-occurrence counts of the plaintext (key)words in these documents, dubbed *background knowledge*.

Islam et al. show that a large percentage of queries is recoverable, expressed as the *(query) recovery rate*, if the adversary has perfect background knowledge, meaning that the co-occurrence counts of keywords exactly match the co-occurrence counts of their corresponding queries. They also briefly show that the recovery rate drops significantly in simulations with non-perfect background knowledge.

We revisit the IKK attack and empirically evaluate assumptions Islam et al. make in their proposal of the IKK attack. Additionally, we research correlations between certain parameters and the accuracy/recovery rate of the IKK attack and propose improvements to the attack that increase the recovery rate of the attack.

Our contributions

- We show that assumptions on the (Zipfian) distribution of queries/natural search behavior Islam et al. made positively influences the accuracy of the IKK attack.
- We show that there is a correlation between input parameters of the IKK attack and the accuracy of attack runs, independent of the (email) dataset used in the targeted Searchable Encryption scheme, potentially allowing an adversary to reuse the same values of parameters across different datasets.
- We show the correlations between different methods to simulate background knowledge and the accuracy of the IKK attack, potentially allowing an adversary to quantify the accuracy of the attack run.
- We show that the accuracy of the IKK attack can be increased significantly when combining multiple runs using a majority voting scheme as the median recovery rate is increased up to 10 percentage points, whereas

the variance of the recovery rate is decreased up to 78 percentage points.
- We show that a more deterministic approach to select new states in the IKK attack, inspired by the Count attack [7], increases the accuracy of the attack, while decreasing the average number of states visited. The median recovery rate is increased up to 21 percentage points and the variance of recovery rates is decreased up to 57 percentage points, while the average visited number of states is decreased by 28 percent.

II. The IKK Query Recovery Attack

A. Searchable Encryption (SE)

The first SE scheme was proposed by Song et al. [1] to provide a client with a way to search for the occurrence of a plaintext word among a set of encrypted documents stored on a server without an adversary being able to learn the (plaintext) contents of these documents.

The server stores a set of encrypted documents, for example email files. The client wants to retrieve emails that contain information on an upcoming merger and thus requests all emails that contain the (plaintext) word *merger*. It does so by generating a so-called query using a keyed trapdoor function, $Trapdoor_{k_t}(merger)$, for example, a keyed hash function. Only users with key k_t can generate valid queries. We assume, just like Islam et al. [6], that queries are deterministic.

In order to retrieve the corresponding documents, the client sends the query to the server, which on its turn, performs a matching algorithm. Most proposed SE schemes either encrypt every single word in a document (*In-place SE* [7]) to encrypt a document or encrypt every document using a traditional encryption scheme, such as AES, while also generating an encrypted inverted index of documents and trapdoors (*Encrypted-index SE* [7]) to allow the server to perform the search query. No matter the SE scheme, the server returns all the documents that match the search query, which can be decrypted by the client.

B. Access pattern disclosure

Just like Cash et al. [7], we deem the server the most likely adversary as it has access to the most information. Nonetheless, any adversary with access to the communication channels is able to connect a query to the identifiers of the documents that were returned and thus is able to see which documents were *accessed* upon the query of the client. This has been dubbed *(data) access pattern disclosure* in the literature [6]. Almost all SE schemes, except schemes that re-encrypt the documents or the encrypted index stored on the server after each query [3], [4], disclose access patterns of particular queries, and thus each query gives an adversary more information on which queries are connected to which documents.

Some schemes propose to obfuscate access patterns which can lead to inconsistencies in the search results (false positives or false negatives) or an increase in communication and storage overhead [5]. In our research, we assume no such inconsistencies were added to the search results.

Although we note that different SE schemes leak different levels of information, we only research the disclosure of access patterns and assume that the adversary is able to get $\langle query, response \rangle$ pairs, where $response$ is a list of documents that matched the issued query.

The leaked $\langle query, response \rangle$ pairs allow the adversary to construct an inverted index from queries/trapdoors and documents. Each cell in the matrix contains a 1 if the document matched the query, i.e. the plaintext keyword occurs at least once in said document, or a 0 if not. An example of an (observed) query inverted index is shown in Table I.

TABLE I
Example of a query inverted index

Query	Documents		
	Doc_1	Doc_2	Doc_3
$q_1 = Trapdoor_{k_t}(merger)$	1	1	0
$q_2 = Trapdoor_{k_t}(corporate)$	1	1	0
$q_3 = Trapdoor_{k_t}(report)$	0	1	1

C. Statistical processing

The IKK attack is grounded on the assumption that some words are more likely to occur together in any piece of natural language than others. Islam et al. [6] give the example of the words *New*, *York* and *Yankees*, where the words *New* and *York* are more likely to occur together than *New* and *Yankees* or *York* and *Yankees* because they are also used to refer to the city and the state and not only the baseball team.

Islam et al. propose a model where the co-occurrence counts of 2 queries are used to recover which plaintext words correspond to which queries. The authors use a so-called co-occurrence matrix to express all co-occurrence counts of the queries in an attack run, as an co-occurrence matrix lists the (relative) co-occurrence count for each of the queries with all other queries and with itself. The probability that two words appear together in a given document is expressed using the following formula by Islam et al.:

$$\beta = \frac{R_{Q_s} \cdot R_{Q_t}}{n} \tag{1}$$

In this formula, Q_t and Q_s are two queries, and R_{Q_x} denotes a vector with ones and zeros indicating whether the word corresponding to the query occurs at least once in the document corresponding with the place in the vector (1) or not (0). The co-occurrence count is calculated by taking the *dot* product of R_{Q_s} and R_{Q_t}. To get the relative co-occurrence count this value is simply divided by n, which denotes the total number of documents in the dataset.

A query co-occurrence matrix simply lists all the co-occurrences of queries, is symmetric by nature and is easily generated using an inverted index as every row in the index, corresponding to query Q_x, is already represented as vector R_{Q_x}.

The relative co-occurrence count of a query with itself is the total number of documents said query occurs in divided by n. The example inverted index in Table I gives us the co-occurrence matrix in Table II.

TABLE II
EXAMPLE OF A QUERY CO-OCCURRENCE MATRIX

Queries	q_1	q_2	q_3
q_1	0.67	0.67	0.33
q_2	0.67	0.67	0.33
q_3	0.33	0.33	0.67

D. Background knowledge assumptions

In most of their simulations Islam et al. [6] assume the adversary has perfect background knowledge of the co-occurrence counts of plaintext words in the documents stored encrypted on the server. They mention that it is difficult, if not impossible, to obtain perfect background knowledge and briefly experiment on the accuracy of their attack in simulations with non-perfect background knowledge by adding various degrees of Gaussian noise to a co-occurrence matrix corresponding to perfect background knowledge.

Cash et al. [7] further research the effect of non-perfect background knowledge, but instead of adding various degrees of Gaussian noise to a perfect representation of the background knowledge (of the adversary) the authors assume the adversary (server) has access to a *fraction* of the plaintext documents and thus the adversary is capable of calculating both inverted indices and co-occurrence matrices from the documents it knows. The authors report that the IKK attack performs quite poorly if the background knowledge is made up of less than 99% of the documents.

An example of a background knowledge co-occurrence matrix is shown in Table III.

TABLE III
EXAMPLE OF A (PERFECT) BACKGROUND KNOWLEDGE CO-OCCURRENCE MATRIX

Keywords	*corporate*	*merger*	*report*
corporate	0.67	0.67	0.33
merger	0.67	0.67	0.33
report	0.33	0.33	0.67

E. Simulated Annealing

Islam et al. [6] use two algorithms to recover queries from a query co-occurrence matrix and a background knowledge co-occurrence matrix:

Their *Optimizer* algorithm (Algorithm 1) assigns a random 1-to-1 mapping for each query to a random keyword in the background co-occurrence matrix as the *initial state* variable. The mapping corresponds to a mapping between the query co-occurrence matrix and a subset of the background knowledge co-occurrence matrix which is equal in its dimensions to the query co-occurrence matrix. Each cell in the query co-occurrence matrix is therefore mapped to a single cell in the background knowledge co-occurrence matrix. The *initial state* is given as input to the ANNEAL algorithm.

Their *ANNEAL* (Algorithm 2) is a *Simulated Annealing* algorithm [8], which first copies the *initial state* to a *current state* variable and enters a while loop. Each iteration the algorithm randomly selects both a mapping (of a single query to a single keyword, i.e. $q_1 \mapsto k_1$) and a keyword (k_2) from the list of potential keywords. If the selected keyword is already mapped to another query, i.e. $q_2 \mapsto k_2$ is in *current state*, the mappings are simply interchanged, i.e. $q_1 \mapsto k_2$ and $q_2 \mapsto k_1$, otherwise the selected mapping is changed to $q_1 \mapsto k_2$ to obtain the *next state*.

The algorithm determines whether it should accept or reject *next state* in favor of or against *current state* respectively. The algorithm calculates the sum of the squared Euclidean distance of the co-occurrence counts of each of the mappings with other mappings for both *current state* and *next state*. Depending on the calculated squared Euclidean distance either *next state* is accepted (and becomes *current state* in the iteration of the while loop) or is rejected. If the Euclidean distance of *next state* is lower than that of *current state next state* is accepted. A *next state* with a higher Euclidean distance is not necessarily rejected, but might be accepted with a small probability, depending how close the Euclidean distance is to 0. This is included to decrease the possibility of the algorithm finishing its run in a local optimum state as opposed to finding the global optimum state.

The *ANNEAL* algorithm takes three input parameters, next to the co-occurrence matrices. These input parameters *initial temperature*, *cool down rate* and *rejection rate* are used to ensure the algorithm has a finite run time.

The *initial temperature* initializes the internal *current temperature* variable of the algorithm. Each iteration in the while loop *current temperature* is decreased by multiplying it with *cool down rate* (a value between 0 and 1, close to 1). The algorithm returns *current state* as the final mapping if the system *freezes*, i.e. *current temperature becomes 0*. *initial temperature* and *current temperature* therefore together determine the maximum number of loops the algorithm goes through. The algorithm can also finish before it freezes if no *next state* has been accepted for a certain (consecutive) number of iterations, which is determined by the value of *rejection rate*. The *current temperature* variable is also used also in deciding whether to accept a worse *next state* with a small probability.

F. Simulations

1) Datasets Used: Both Islam et al. [6] and Cash et al. [7] use the *_sent_mail* data folder of the ENRON dataset [9] (containing 30109 emails) as the dataset to run simulations on. Additionally, Cash et al. use the Apache Lucene project's *java-user* mailing list [10] (containing 50116 emails) in their simulations.

2) Tokenization/Stemming algorithm: Both papers tokenize all emails in the dataset to specific words before they are able to stem individual words, but neither elaborate on the

tokenization algorithm used in their simulations. Stemming is done using Porter's stemming algorithm [11] to get the stem of each word, meaning that words like 'has' and 'have', which in principle have the same meaning, are stemmed to the same word.

3) Keyword Generation: The stemmed keywords are sorted in decreasing order of overall occurrence. The 200 most occurring (stemmed) words in a dataset, that are likely to occur in every file (for example 'a' and 'the'), are removed as they are not deemed useful in a Searchable Encryption scheme. The next x words are regarded as the keyword set.

4) Query Generation: Both Islam et al. and Cash et al. simulate a certain number of queries by using the Zipfian distribution on the keyword set. Due to the nature of the Zipfian distribution words with a higher occurrence count are more likely to be simulated as a query.

5) Reported results: Islam et al. report recovery rates ranging from 60%-100% depending on the number of keywords, number of queries and the % of queries 'known' before the attack run. With different levels of Gaussian noise added to the background knowledge, the accuracy of the attack ranges between 40% and 85%. Cash et al. report recovery rates of the IKK attack ranging between 0% and 100% and show an exponentially decreasing correlation between the size of the input matrices (query and background knowledge co-occurrence matrices) and the recovery rate. Cash et al. also report recovery rates ranging between 0% and 60% for different percentages of documents 'known' to the adversary.

III. Revisiting the IKK attack

Islam et al. [6] introduced the study on query recovery attacks by proposing the IKK query recovery attack. The authors report high query recovery rates that would allow an adversary to determine what a user searched for. More importantly, as Cash et al. [7] note in their paper, correctly recovered queries are inherently a part of the plaintext of encrypted documents and thus disclose part of the plaintext of the document stored on the server. We therefore stress that it is important to get a more broad understanding of query recovery attacks. In this research we revisit the following facets of the IKK attack:

- We evaluate the assumption on query distribution following the Zipfian distribution made by Islam et al. while simulating runs of the IKK attack (Section IV).
- We look at the correlation between the *initial temperature*, *cool down rate* and the *rejection rate* input parameters and the accuracy of the IKK attack (Section V).
- We look at the correlation between the similarity between input (co-occurrence) matrices and the accuracy of the IKK attack (Section V).

Furthermore, we propose and research the following improvements to the IKK attack:

- We propose to use a majority voting scheme to increase the accuracy of the IKK attack by combining the results of multiple runs (Section VI).
- We propose to (more) deterministically choose the next state of the ANNEAL algorithm to increase the accuracy of and decrease the number of visited states by the IKK attack. This method incorporates the (relative) word occurrence method, as proposed by Cash et al. in their Count attack (Section VI).

In order to run simulations of the IKK attack to address the points above we implemented the IKK attack as proposed by Islam et al. in Python3 and published it on Github [12]. The implementation allows the user to select:

- the distribution used to simulate queries (*Zipfian*, *reverse Zipfian*, *Uniform*)
- values for the parameters of the ANNEAL algorithm (*initial temperature*, *cool down rate*, *rejection rate*)
- sizes of the query and background knowledge co-occurrence matrices (resp. *number of queries*, *number of keywords*)
- datasets/email folders to use in the simulation (*ENRON/_sent_mail*, *ENRON/inbox*, *ApacheLucene-java-user* (*Apache*))
- different methods to simulate non-perfect background knowledge (*Gaussian noise addition*, *using a fraction of the keywords*, *using a fraction of the documents*)
- the number of consecutive runs with exactly the same input parameters
- whether to more deterministically select new states using word occurrences as also proposed in the Count attack by Cash et al.

To give the reader an idea of the input parameters used in our simulations we mention the standard values for the different parameters of the IKK attack in Table IV.

TABLE IV
Standard parameter values in the IKK attack simulations

Variable	*Value*
initial temperature	1.0
cool down rate	0.999
rejection rate	50000

keyword percentage	1.0
document percentage	1.0

Gaussian noise scaling factor	0.0

Variable	*Value*
nr of keywords	1500
nr of queries	150

dataset / email folder	*ENRON / _sent_mail*

distribution	*Zipfian*

nr of runs	1

We briefly capture our generalized method below. Simulation specific methodologies are elaborated upon in their correlated sections (Sections IV, V and VI).

1) Tokenize and stem the words in all documents in a specific dataset. Tokenization is done by splitting the document on whitespaces. Stemming is done using Porter's stemming algorithm [11].
2) Sort all unique (stemmed) words in decreasing order of occurrence (count) (the total number of times a word occurs in the dataset, not the number of matching documents).

3) Disregard the first 200 most occurring words, just like Islam et al., and take the subsequent x words as *keyword set*. x is equal to the *number of keywords* input variable in our simulations.
4) Simulate y queries from the x selected keywords using a specified query distribution as the *query set*. y is equal to the *number of queries* input variable in our simulations.
5) Generate the query and background knowledge inverted indices from the selected queries and keywords, and the list of documents.
6) Generate the query and background knowledge co-occurrence matrices from the inverted indices.
7) Input the co-occurrences matrices and the input parameters *initial temperature*, *cool down rate* and *rejection rate* in the ANNEAL algorithm (Algorithm 2).
8) Calculate the (query) recovery rate by dividing the number of correctly mapped queries, where $query = keyword$, by the total number of queries.

Islam et al. also use a *known queries* variable in their experiments, a method also adopted by Cash et al. This variable denotes $\langle query, keyword \rangle$ pairs that the adversary knows to be mapped correctly before the attack run. We argue that the actual value of this variable is likely to be (close to) 0 and we therefore excluded this variable from our experiments.

IV. Assumptions Evaluation

In their simulations, Islam et al. [6] make an assumption on distribution of queries in a real-world SE scheme in order to estimate real-world search behavior of users. They assume natural search behavior can be estimated by simulating queries using the Zipfian distribution as they argue that search behavior might follow a Zipfian distribution as the simulations are run on a natural language corpus. In their paper, the authors state that 'according to Zipf's law, in a corpus of natural language utterances, the frequency of appearance of an individual word is inversely proportional to its rank' [13]. The Zipfian distribution is also used by Cash et al. [7] to simulate queries for their simulations.

In order to simulate queries, from the simulated keyword set (of size x), Islam et al. first sort the words in the keyword set in decreasing order of overall occurrence. For the word in the j^{th} position in this list (rank j) the following formulas are used to determine the probability the word is selected as a query:

$$Pr_j = \frac{\frac{1}{j}}{N_x} = \frac{1}{j \times N_x} \tag{2}$$

$$N_x = \sum_{i=1}^{x} \frac{1}{i} \tag{3}$$

A word with a higher total occurrence count is therefore more likely to be simulated as a query. Islam et al. also note that duplicate queries are removed.

We argue that the assumption that search behavior follows a Zipfian distribution is counter-intuitive in the sense that users are more likely to search for a specific email in their mail archive and thus issue a (single word) query that is likely to return the sought after document while also not returning too much other emails (false positives). We therefore argue that search behavior might instead follow a reverse Zipfian distribution and thus a word that has a lower occurrence count has a higher chance of being selected as a query. The reverse Zipfian distribution can be calculated using the same formulas as the Zipfian distribution, but the list of word occurrences is sorted in ascending order of occurrence as opposed to descending order.

To compare the effect of the distribution used to simulate the queries we conducted three different simulations for the Zipfian distribution, reverse Zipfian Distribution and Uniform distribution respectively. The Uniform distribution denotes the setting where queries are simulated from the keyword set uniformly at random. The results of our simulations are shown in Figure 1, where each box plot is the aggregation of 20 simulations.

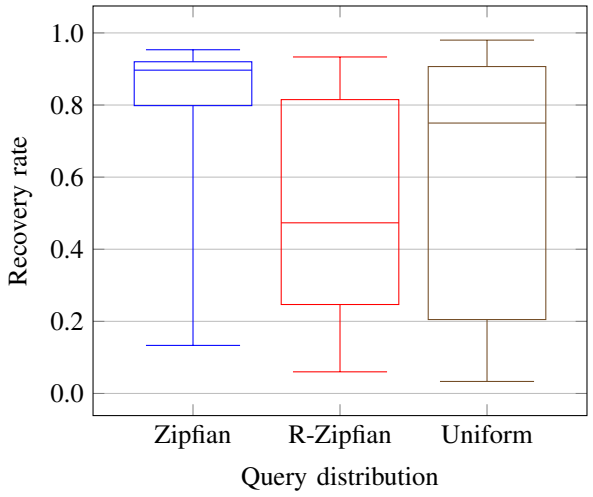

Fig. 1. Correlation of different Query distribution and Recovery rate

It can be seen that the distribution chosen to simulate queries influences the results of the IKK attack quite a lot and that simulations where the queries were simulated using the Zipfian distribution in general have a much higher recovery rate than simulations where a different distribution was used. Unfortunately, we simply do not know what distribution real-world search behavior follows in a Searchable Encryption scheme as, to the best of our knowledge, there exists no dataset which contains query search behavior of real-world users in an SE setting. We can only conclude that the actual distribution determines the accuracy of the IKK attack and therefore whether using a Searchable Encryption scheme poses a risk for search privacy and potentially data confidentially.

V. Recovery Rate Quantification

A. Input Parameter Correlation

Islam et al. [6] show that their IKK attack allows an adversary to recover (most of the) queries in a simulated

setting. Their ANNEAL algorithm (Algorithm 2), which is part of their attack algorithm, takes three input parameters *initial temperature*, *cool down rate* and *rejection rate* to ensure the algorithm has a finite run time.

The *initial temperature* initializes the internal *current temperature* variable of the algorithm. Each iteration in the while loop *current temperature* is decreased by multiplying it with *cool down rate* (a value between 0 and 1, close to 1). The algorithm returns *current state* as the final mapping if the system *freezes*, i.e. *current temperature becomes 0*. *initial temperature* and *current temperature* therefore together determine the maximum number of loops the algorithm goes through. The algorithm can also finish before it freezes if no *next state* has been accepted for a certain (consecutive) number of iterations, which is determined by the value of *rejection rate*.

The values of these parameters have a significant influence on the number of visited states of the IKK attack as the *initial temperature* and *cool down rate* together determine the maximum number of states the algorithms visits, whereas the value of *rejection rate* determines whether the algorithm finishes before the *current temperature* reaches 0 or not. We argue that the accuracy of the IKK attack is therefore dependent on the values of these input parameters. This means that a proven correlation between these three input parameters, independent of the underlying dataset, and the recovery rate might allow an adversary to use simulations on another dataset to find the optimal input parameters for the IKK attack.

To answer the question whether there is a correlation between the three input parameters and the recovery rate, independent of the dataset, we used the same datasets as used by Islam et al. and Cash et al. The first dataset is the ENRON dataset [9], specifically its *_sent_mail* data folder which contains 30109 emails. Cash et al. also experiment on the *java-user* mailing list of the Apache Lucene project (henceforth Apache) [10] (reportedly containing about 38.000 emails). However, the exact dataset they used was unavailable and thus we crawled the archive site of the java-user mailing list and retrieved 50116 emails. The crawled Apache dataset is included in our Python3 implementation of the IKK attack on Github [12].

In order to test our hypothesis we conducted three different experiments. In all of the experiments we kept one of the input parameters (*initial temperature*, *cool down rate*, *rejection rate*) constant while varying the other two. The experiments were repeated for both the Apache and ENRON dataset, with both the query and background knowledge co-occurrence matrix from the same dataset and with perfect background knowledge. Each point in Figures 2, 3, 4 is the average of 5 simulations of the IKK attack.

Figure 2 shows the aggregation of simulation results with a constant *initial temperature*. The results of simulations on the ENRON dataset and the Apache dataset are roughly the same. The only exceptions are the simulations with a *rejection rate* of 50000 and a *cool down rate* of 0.9999 respectively 0.99999, which we attribute to the relatively low number of simulations

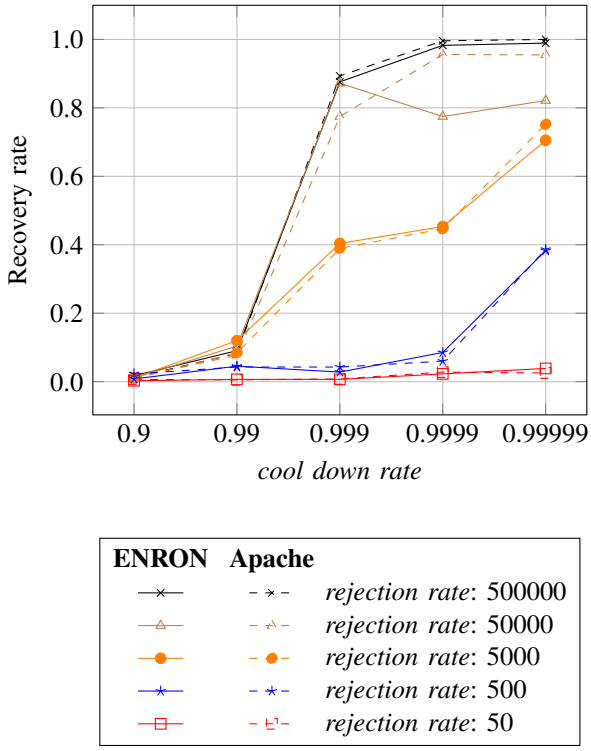

Fig. 2. Correlation between *cool down rate* and Recovery rate, with different values of *rejection rate*

(5) aggregated in data point. With more simulations these results might become more similar. Furthermore, the recovery rate increases with both the *cool down rate* and the *rejection rate*. This makes sense as the maximum number of loops is increased with a *cool down rate* closer to 1 and a higher *rejection rate* increases the likelihood of finding the best mapping as the algorithm does not halt prematurely.

Figure 3 shows the aggregation of simulations with a constant *rejection rate*. We see that the value of recovery rate is only dependent on the value of *cool down rate* as the correlation between *initial temperature* and recovery rate is relatively constant. The recovery rate is also not dependent on the underlying dataset used as the results for both the ENRON and Apache dataset are roughly the same.

Figure 4 shows the aggregation of simulations with a constant *cool down rate*. We see that the value of recovery rate is dependent on the value of *rejection rate* and not on the value of *initial temperature* as we again see a constant correlation between *initial temperature* and recovery rate. We can also see that the value of recovery rate is not dependent on the underlying dataset used as the results are quite similar for both the ENRON and Apache datasets.

We conclude that the values of *rejection rate* and *cool down rate* significantly influence the recovery rate of the IKK attack. Furthermore, we conclude that it is possible for an adversary to find the optimal values for *cool down rate* and *recovery rate* using simulations on a different dataset as the recovery rate is independent of the underlying dataset used. This means that it

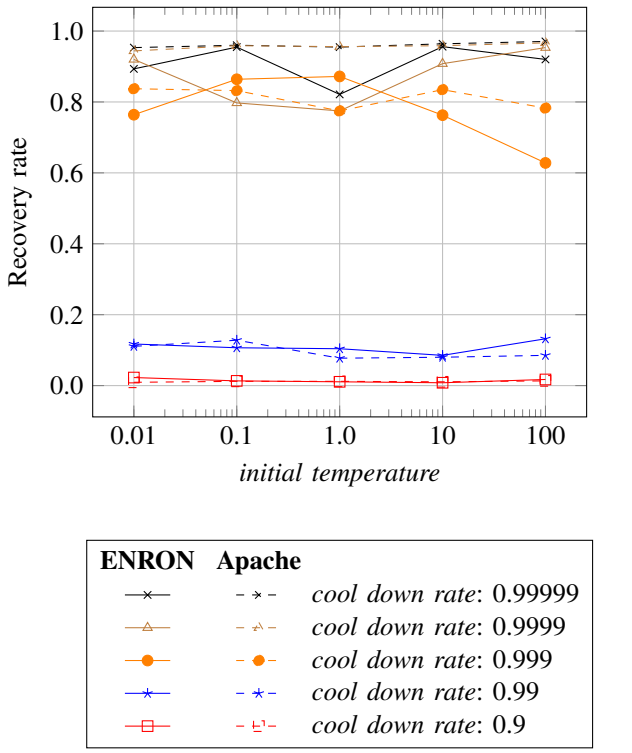

Fig. 3. Correlation between *initial temperature* and Recovery rate, with different values of *cool down rate*

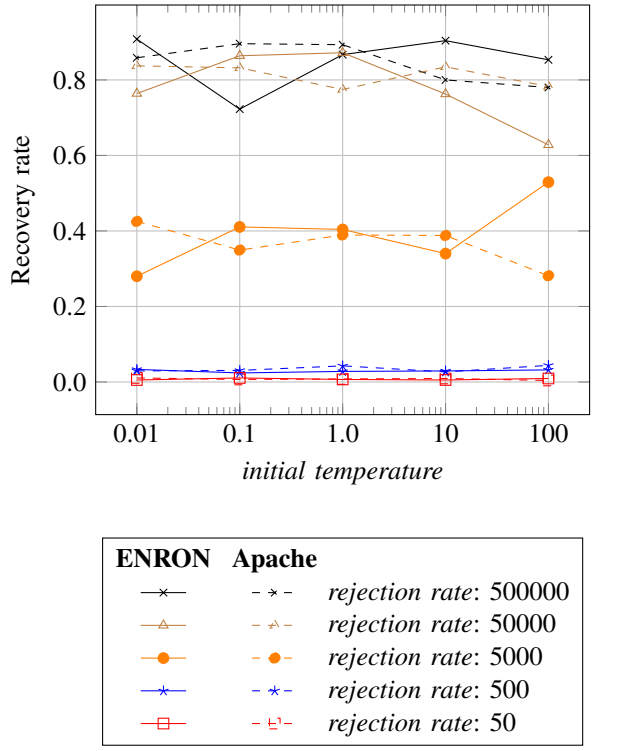

Fig. 4. Correlation between *initial temperature* and Recovery rate, with different values of *rejection rate*

is possible to use simulations on the ENRON dataset to select the optimal input parameter values for runs on the Apache dataset and vice versa. We argue that email datasets are quite similar due to the nature of the files they contain as emails are structured in a certain way, are limited in length and are used for specific purposes and thus might contain similar data. More research should be conducted to find out whether our findings hold true for completely different datasets as well.

B. Co-occurrence Matrix Correlation (Partial background knowledge)

Both Islam et al. [6] and Cash et al. [7] both briefly elaborate on the recovery rate of the IKK attack in the case where the adversary only has partial background knowledge.

Islam et al. add various degrees of Gaussian noise to individual cells in the co-occurrence matrix representing perfect background knowledge to simulate this setting, whereas Cash et al. simulate non-perfect background knowledge co-occurrence matrix by taking a fraction of all documents in the dataset. Both papers show that the accuracy of the attack is greatly dependent on the level of background knowledge the adversary has. We therefore argue that it is important to get a better understanding of the correlation between the level of background knowledge the adversary has and the recovery rate of the IKK attack. We also argue that the level of background knowledge can be expressed as a similarity between the query and background knowledge co-occurrence matrices, i.e. the *co-occurrence matrix similarity*.

In order to express co-occurrence matrix similarity we propose a metric that returns a similarity score between 0 (no similarity) and 1 (equivalent matrices) between two co-occurrence matrices of the same dimensions. For two matrices M_1 and M_2 and arbitrary words a, b (corresponding to a row and column) the following formulas are used:

$$\Delta^2_{a,b} = \begin{cases} (M_1[a,b] - M_2[a,b])^2, & \text{if } a,b \in M_1 \text{ and } a,b \in M_2 \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

$$\Delta^2_{total} = \sum_{\forall a,b \in M_1} \Delta^2_{a,b} \tag{5}$$

$$\epsilon_{a,b} = \begin{cases} 1, & \text{if } a,b \in M_1 \text{ and } a,b \in M_2 \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

$$\epsilon_{total} = \sum_{a,b \in M_1} \epsilon_{a,b} \tag{7}$$

$$Co-ocsim. = \left(1 - \frac{\Delta^2_{\text{total}}}{\epsilon_{\text{total}}}\right) * \left(\frac{K_{\text{overlap}}}{K_{\text{total}}}\right) \tag{8}$$

Equations 4 and 5 are used to calculate the total squared Euclidean distance of cells that occur both in M_1 and M_2.

Equations 6 and 7 are used to calculate the number of cells that occur in both M_1 and M_2.

In Equation 8 we calculate the average squared Euclidean distance of cells that occur in both matrices and multiply this

TABLE V
EXAMPLE CO-OCCURRENCE MATRICES

M_1	a	b	**c**
a	1	1	1
b	1	1	1
c	1	1	1

M_2	a	b	c
a	1	1	1
b	1	1	1
c	1	1	1

M_3	a	b	c
a	0	0	0
b	0	0	0
c	0	0	0

M_4	a	b	**d**
a	1	1	1
b	1	1	1
d	1	1	1

by the ratio of *row identifiers that occur in both matrices ($K_{overlap}$)* to the *total number of rows in both matrices (K_{total})*.

In Table V matrices M_1 and M_2 are exactly the same. Δ^2_{total} is 0 as the squared Euclidean distance between each of the cells is $(1-1)^2 = 0$. The average is therefore also 0. As all keywords in both matrices also occur in the other matrix, $\frac{K_{\text{overlap}}}{K_{\text{total}}} = 3/3 = 1$. The similarity between the matrices is calculated as $co-ocsim. = (1-0) * 1 = 1$ meaning that the matrices are exactly the same. The calculation for the similarities between matrices M_1 and M_3, and M_1 and M_4 gives the values 0 and $2/3$ respectively.

In our simulations we calculate the co-occurrence matrix similarity using the perfect background knowledge co-occurrence matrix M_F (which corresponds with an unquerified query co-occurrence matrix) and a non-perfect background knowledge co-occurrence matrix M_P. Both matrices have the same dimensions. In order to simulate partial background co-occurrence matrix M_P we use the following methods:

Gaussian noise addition - We use the method by Islam et al. to add Gaussian noise in various degrees to the cells in M_F to obtain M_P.

Document percentage - In this setting we use 10% to 100% of the user folders in a dataset to generate M_P. This method differs a bit from the method by Cash et al. as we argue that the adversary is more likely to obtain a percentage of the mail boxes of users (and all documents that are in these folders) than a percentage of all documents, selected uniformly at random, in a dataset. We believe that this choice might influence the results as different users are likely to use specific language in (all of) their emails.

Keyword percentage - In this setting we, uniformly at random, select 10% to 100% of the keywords in M_F to obtain M_P. To keep the dimensions of M_P consistent throughout all our simulations we supplement the selected keywords with words with a lower occurrence count in the dataset used, i.e. that were not in the keyword set.

Different input folder - In this setting we use a different, but similar dataset to generate M_P. In our simulations we use the *inbox* folder (containing 44859 emails) of the ENRON dataset.

The results of the different methods are shown in Figures 5, 6 and 7. In these figures we group the values into certain buckets to group similarity scores. If a co-occurrence similarity score is between 0 and 0.1 it is put in the 10% similarity bucket, a value between 0.1 and 0.2 is put in the 20% bucket and so on.

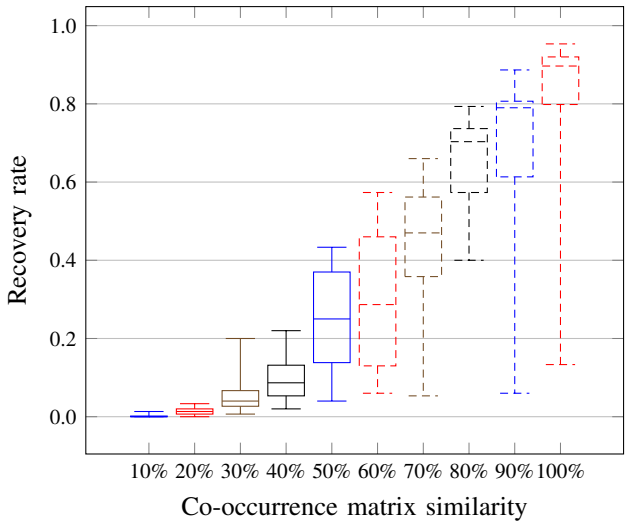

Fig. 5. Correlation between co-occurrence matrix similarity and recovery rate, simulating non-perfect background knowledge by regarding a *percentage* of keywords

Figure 5 shows the correlation between the co-occurrence similarity and the recovery rate in the setting where we use a certain percentage of the keywords in the keyword set to simulate partial background knowledge. Each bucket represents 20 simulations. Due to the nature of our similarity score using 90% of keywords from the keyword set will result in a similarity score of exactly 90%. The figure shows a clear linear correlation between the co-occurrence similarity score and the recovery rate. This makes sense as in this setting entire rows (and thus also columns) that are present in M_F are changed while simulating M_P. The highest possible percentage of recoverable queries therefore is linearly dependent on the percentage of keywords regarded. As the individual cell values are not changed while simulating M_P (as opposed to the other methods) the algorithm is likely to recover (most of the) queries of which the corresponding keywords were selected in M_P as these have the optimal Euclidean distance of 0.

Figure 6 shows the correlation between the co-occurrence matrix similarity and the recovery rate in simulations where non-perfect background knowledge is simulated by taking a percentage of user folders in a dataset to generate M_P. We ran 20 simulations for each percentage ranging from 10%, 20% to 100% each. The first thing that we notice is that the buckets do not contain the same number of simulations per bucket, which is shown in the figure as the number between brackets. This is due to the fact that we uniformly at random select a percentage of all user folders in a dataset and these user folders do not contain the same number of documents. Different writing styles can also be of influence to the overall co-occurrence similarity.

The results in Figure 6 show a different correlation than the results in Figure 5. This makes sense as in these simulations

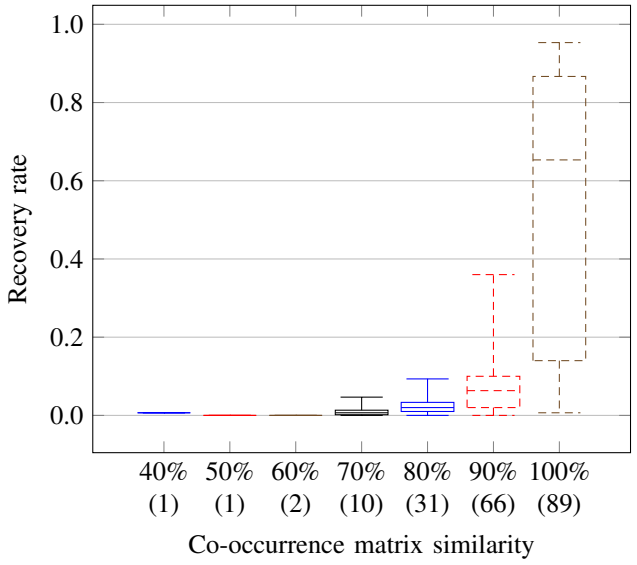

Fig. 6. Correlation between co-occurrence matrix similarity and recovery rate, simulating non-perfect background knowledge by regarding a *percentage* of user folders in the dataset

both row/column identifiers as well as individual cell values are changed. The algorithm is less likely to correctly map queries to keywords that do occur in M_P as the changed individual cell values, in Figure 6, make it less likely to find the optimal mapping. We note that the results in the 40%-60% bucket do not give much information, as each bucket consists of a single simulation. We conclude from the rest of the results that the co-occurrence matrix similarity and the recovery rate show an exponential correlation in Figure 6.

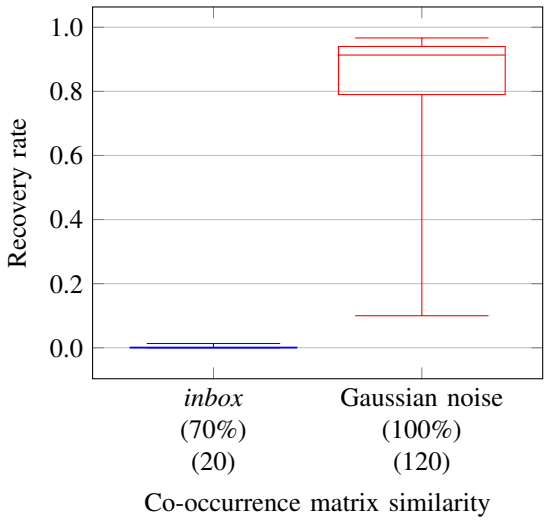

Fig. 7. Correlation between co-occurrence matrix similarity and recovery rate, simulating non-perfect background knowledge by using the *ENRON/inbox* data or by adding Gaussian noise

The results in Figure 7 show the correlation between the co-occurrence matrix similarity and the recovery rate of simulations where M_P was generated using a similar, but different dataset (*ENRON/inbox*) and when we add Gaussian noise to various degrees.

First of all, if we generate M_P using the *ENRON/inbox* data folder we obtain a similarity score of approximately 70% to the *ENRON/_sent_mail* dataset. The recovery rate of almost 0 is consistent with our results in Figure 6.

With the addition of various degrees of Gaussian noise (with C values 0.0, 0.2, ..., 1.0) the similarity of the co-occurrences matrices is always between 0.999 and 1.0. This can be explained as this method does not change the row/-column identifiers, but only changes the individual cell values (co-occurrence counts). As only a little noise is added most of these cell values stay relatively the same. The recovery rate distribution among 120 simulations is relatively high as opposed to other methods to generate non-perfect background knowledge.

We conclude that it is not possible to use our matrix similarity metric to find a single correlation between the similarity of co-occurrence matrices and the recovery rate. The different methods change non-perfect background knowledge M_P in different manners and this influences the results of the IKK attack a lot. The IKK attack correctly recovers queries if the co-occurrence counts in M_P exactly match (or are close to) those in M_F, which is shown in Figures 5 and 7 (*Gaussian noise addition*). If the co-occurrence counts in M_P are further away from those in M_F, which is the case in Figures 6 and 7 (*Inbox folder*) the accuracy of the IKK attack decreases drastically.

We argue that the scenario where the background knowledge, represented as M_P, is generated by taking a percentage of the user folders in a dataset is the most realistic one in a real-world scenario. It is not unlikely that an adversary, somehow, gets access to a certain set of the plaintext contents of the email boxes of specific users. The IKK attack proves to be a powerful attack which can break the privacy of queries as well as data confidentially of documents stored encrypted of the server, yet it is only exploitable by a powerful adversary, which has access to a dataset which results in background knowledge that is at least 90% similar the actual dataset encrypted on the server, as can be seen in Figure 6.

The interested reader can find correlations between the values of the different methods (document percentage, keyword percentage, added Gaussian noise) and the co-occurrence similarity/recovery rate in Appendices B and C.

VI. Improvements

A. Combining multiple runs

The IKK attack returns a 1-to-1 mapping between queries and keywords. An adversary cannot, from the mapping alone, determine which queries were recovered correctly and which were not, as even with perfect background knowledge the IKK attack shows a lot of variance. For example, Figure 8 shows that the recovery rates of 20 simulations of the IKK attack with equal input parameters and perfect background knowledge return recovery rates ranging between 0.1 and 0.98, which almost spans the entire range of possible recovery rates.

The IKK attack is a probabilistic algorithm in the sense that the algorithm uniformly at random selects a new mapping to change in the current state to determine the next state to explore. We argue that the IKK attack shows a large variance in the recovery rate as the algorithm merely approximates the optimal state yet does not necessarily always return it. A deterministic algorithm that simply visits every possible mapping is less likely to show a large variance, but the single attack runs will have to evaluate far more different states in order to be successful, making such an algorithm quite inefficient as the total number of potential states is given using the following equation:

$$\text{nr. of states} = \frac{(\text{nr. of keywords})!}{(\text{nr. of keywords} - \text{nr. of queries})!}. \quad (9)$$

For 50 observed queries and 500 keywords in the background knowledge this would mean that there are already $7.039 * 10^{133}$ potential states to explore.

As every single run of the IKK attack still approximates the optimal mapping, we argue that it is possible to combine the results of different attack runs using a simple majority voting scheme to better approximate the optimal solution. We conducted 20 simulations each consisting of 20 attack runs on the same query co-occurrence matrix and background knowledge co-occurrence matrix (representing perfect knowledge) per simulation. In each of the simulations, we combined a certain number of runs by selecting the most prevalent keyword mapped to each of the queries. If no prevalent keyword could be found (two or more keywords are most prevalent) the majority voting scheme did not assign a most prevalent keyword to a query and instead assigned the *None* value.

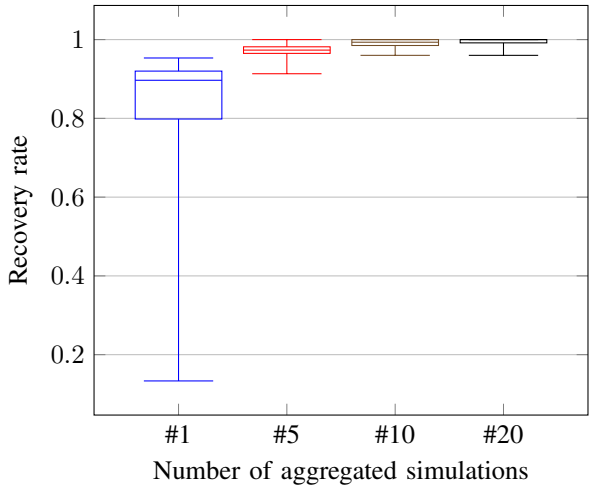

Fig. 8. Aggregation of a number of different runs of the same simulation using a majority voting scheme

Figure 8 shows the results of combining multiple runs on the same query and background co-occurrence matrices. It can be seen that the accuracy of the attack significantly decreases the variance that is observed with single runs of the IKK attack. When combining 5 runs per simulation (#5) the results are already very promising, which is even more the case when the aggregation contains either 10 or 20 runs per simulation. Our proposed aggregation method also has the advantage that the single attack runs can executed in parallel and then aggregated, ensuring the execution time overhead is limited. The median recovery rate between 1 run per simulation (#1) and 20 runs per simulation (#20) is increased with more than 10 percentage points, whereas the variance is decreased with 78 percentage points.

B. Deterministic IKK attack

As the IKK attack is a probabilistic algorithm, it does not necessarily return the optimal query-to-keyword mapping. We argue a more deterministic approach to finding the right mapping might increase the recovery rates of the IKK attack.

The Count attack, as proposed by Cash et al. [7], takes a more deterministic approach to map queries to keywords, by eliminating candidate mapping keywords using the relative document occurrence count of keywords. Cash et al. assume the adversary has access to not only the co-occurrence counts of queries and keywords, but is also in possession of the (relative) document occurrence counts from queries and keywords, i.e. the number of documents a query or keyword occurs in (relative to the total number of documents in the dataset). The theory behind this is that, while assigning a keyword to a query, a lot of potential keywords can already be eliminated as their relative document occurrence count is not within a certain range of the relative document occurrence count of the query. These keywords therefore are not likely to be the right keyword corresponding to the query and thus can be disregarded.

The Count attack incorporates eliminating candidate keywords using their document occurrence count 'and brute-forces all possible mappings for a small number of queries and returns the mapping which maximizes the number of disambiguated queries'. Cash et al. report much higher recovery rates from their deterministic Count attack as opposed to the probabilistic IKK attack.

We propose to incorporate the candidate keyword elimination method of the Count attack while selecting new mappings in the IKK attack to both decrease the number of potential states to visit as well as increase its accuracy. We also argue that the accuracy of the attack will increase as the algorithm is likely to visit better states on average as the worst potential states are eliminated.

In order to eliminate candidate keywords Cash et al. construct a confidence interval for the document occurrence count of each of the keywords using Hoeffding's inequality [14]. The lowerbound (LB) and upperbound (UB) of the confidence interval per keyword k are calculated using the following formula(s):

$$LB_k,\ UK_k = \frac{c_k^s}{p_{pk}} \mp \sqrt{0.5\ n\ log\ 40} \quad (10)$$

In this formula c_k computes the document occurrence count of keyword k in the background knowledge dataset and p_{pk} denotes the size relativity between the query and background knowledge dataset. $\frac{c_k}{p_{pk}}$ therefore denotes the expected document occurrence count of k in the query dataset. $\epsilon = \sqrt{0.5\ n\ log\ 40}$ is used by Cash et al. to ensure the confidence interval has a confidence level of 95%. n denotes the number of documents in the query dataset.

After calculating a confidence interval for each of the keywords the candidate keywords for a query can be calculated as $S_q = \{k' \in K | LB_{k'} \leq c_q \leq UB_{k'}\}$. S_q denotes the candidate keyword set, K the keyword set and c_q denotes the document occurrence count of query q.

The Original IKK attack maps queries to keywords in two places in the algorithm, namely when selecting the *initial state* (Algorithm 1) and while selecting a *next state* (Algorithms 2 and 3). We therefore incorporated the method of Cash et al. in two places in our Deterministic IKK attack (Algorithms 2 and 4):

While selecting the initial state we first assign a *None* value to queries of which S_q is an empty set, meaning that no keywords are in range. These queries are left unchanged throughout the entire algorithm run and thus are assigned *None* in the final mapping as well. This also allows an adversary to determine which queries were not mapped to a keyword.

Then all queries with a non-empty candidate set S_q are ordered in ascending order of the size of S_q and each of the queries, starting at the query with the lowest size of S_q, is assigned a random keyword in S_q that was not yet assigned to another query. As we enforce the 1-to-1 mapping property of the IKK attack this potentially creates the edge case where all keywords in S_q of a query are already assigned to other queries. The algorithm tries, with a depth of one, whether it is possible to re-assign one of the other queries to 'free up' a keyword in S_q. If it succeeds the 'freed' keyword is assigned to the query, otherwise the query is assigned *None* and is thus disregarded during the rest of the algorithm run.

While selecting a new state we choose a random query, keyword mapping, e.g. $q_1 \mapsto k_1$, from the queries in the *current state* that were not assigned *None* and we select a random keyword k_2 from S_{q_1} as opposed to the full keyword set, while ensuring $k_1 \neq k_2$. Then, just like in the Original IKK attack there are two possibilities:

If k_2 was mapped to a query q_2 we try whether keyword k_1 is in range of query q_2 and interchange the mapping if so. If not, we keep (uniformly at random) selecting a new keyword k_2 and checking whether the new k_2 adheres to the right properties. If we cannot find a satisfactory candidate k_2 for a certain number of loops (2 times the size of the keyword set in our simulations) the algorithm returns the current state as the next state, which is rejected as the Euclidean distance is not better than the old current state (as they are the same).

If k_2 was not mapped to any query in the current state we change the next state so that $q_1 \mapsto k_2$.

In order to compare our deterministic version of the IKK attack to the Original IKK attack we ran the same simulations as we did in Section V-B, where we researched the correlation between the similarity between the co-occurrence matrices and the recovery rate as we argue that it is important to research the effect of our improvements on simulations with different levels of background knowledge to get a broad understanding of the effects of our improvements.

TABLE VI
NR. OF STATES VISITED BY THE IKK AND DETERMINISTIC IKK ATTACK

Parameters	IKK version	Min.	Max.	Avg.
# Total loops	Original	531733	737741	733213
	Deterministic	196790	737741	526038
# Accepted loops	Original	7783	9535	8533
	Deterministic	7287	101145	14252
$\frac{\text{\# Accepted loops}}{\text{\# Total loops}}$	Original	0.0105	0.0159	0.0117
	Deterministic	0.0099	0.1371	0.0263

Table VI, the aggregation of 500 simulations of both algorithms, shows that the Deterministic IKK algorithm visits much less total states on average than the Original IKK attack. Additionally, the average number of iterations where the next state is accepted is much higher and the ratio between the number of *accepted* loops and total loops is more than twice as high for the Deterministic IKK attack. It is useful to note that both attacks at most visit 737.741 different states and then return their current state as the final mapping. This is due to the chosen values of the input parameters of the ANNEAL algorithm and explains the values in the **Max.** column of the *# Total loops* row.

In Figure 9 we compare recovery rate of the Original and Deterministic IKK attack when only a fraction of the actual keywords simulates background knowledge. The recovery rates of the Original IKK attack and methods to generate non-perfect background knowledge are the same as expressed in Figure 5 and each bucket in the figure is the aggregation of 20 simulations. We see the same linear correlation for the Deterministic IKK attack as we saw before for the Original IKK attack, however, recovery rates of the Deterministic IKK attack show much less variance as well as a higher median value.

Figure 10 shows the comparison of the Original and Deterministic IKK attack when non-perfect background knowledge is simulated using other methods than using a percentage of all keywords. Figure 10 therefore also contains the same information as Figures 6 and 7. The (non-percentage) numbers between brackets denote the number of simulations aggregated in that box plot.

In simulations where we took a different, but similar dataset as background knowledge (*ENRON/inbox*) we see that both attacks have recovery rates close to 0.

In simulations where we added Gaussian noise to the background knowledge we see that the Deterministic IKK attack again shows less variance and higher recovery rates.

In simulations where a percentage of user folders in a dataset was used to simulate background knowledge we see the same exponential correlation between co-occurrence similarity

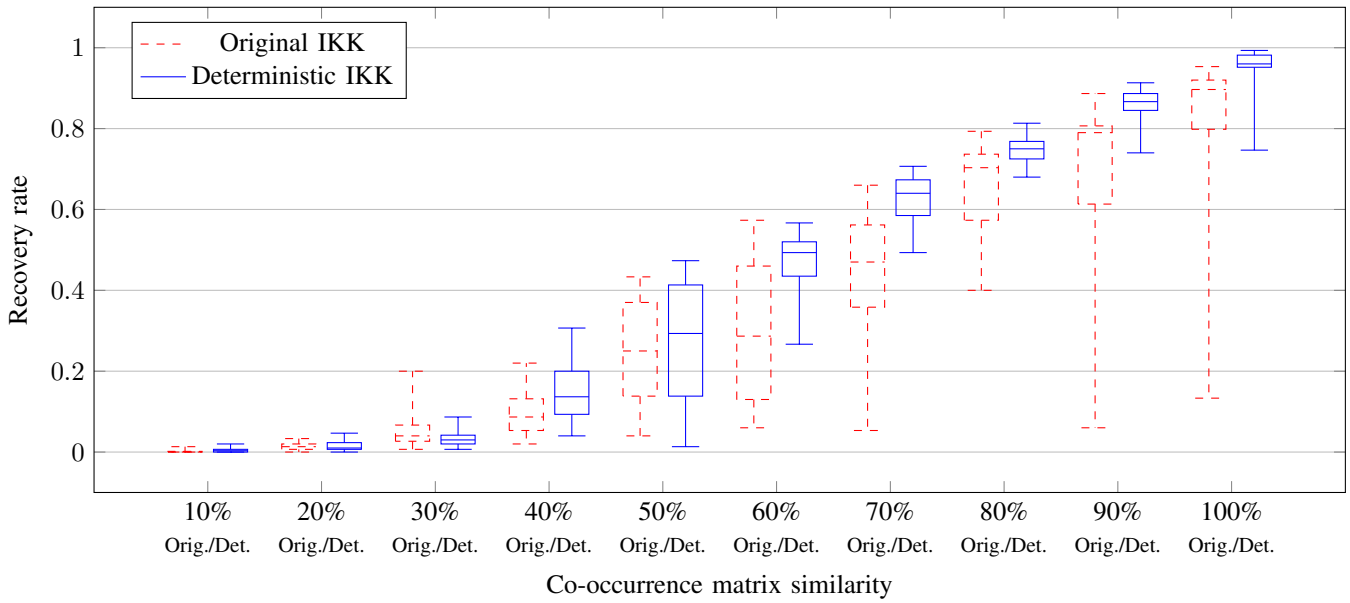

Fig. 9. Original/Deterministic IKK recovery rates and recovery rate, simulating non-perfect background knowledge by regarding a *percentage* of keywords

and the recovery rate as we see for simulations using the Original IKK attack. Additionally, we see that the Deterministic IKK attack achieves higher recovery rates on average, but we do not see the drop in variance that we saw in simulations using the other methods to simulate non-perfect background knowledge.

All in all, we conclude that using components of the Count attack by Cash et al. [7], that make the IKK attack more deterministic, is a promising method to both decrease the number of states visited in a single attack run (28% decrease) and increase the recovery rate, as the median recovery rate is increased up to 21 percentage points (Figure 10, 100% box plot) and the variance is decreased up to 57 percentage points (Figure 9, 100% box plot).

VII. Related work

The first Searchable Encryption scheme was introduced by Song et al. [1] to allow for (plaintext) search among a set of encrypted documents. Their paper introduces the first In-place SE scheme which uses a stream cipher to scan for the occurrence of a plaintext word as well as introduces the notion of the potentially more efficient Encrypted-Index SE schemes. Song et al. already note that these schemes leak access patterns and that statistical attacks might disclose information of encrypted documents, but do not research this further.

The notion of Oblivious RAM (ORAM) [4], introduced before the first SE scheme, is frequently mentioned as a method to not disclose access patterns. Oblivous RAM, however, in a Searchable Encryption scheme is computationally quite expensive. A less expensive version specifically targeted for encrypted search, proposed by Curtmola et al. [3], still is computationally inefficient.

Other papers propose to obfuscate access patterns by introducing inconsistencies in the search results by modifying the internal encrypted index of the SE scheme [5] or by using Bloom filters [2], [15]. These schemes are reportedly computationally expensive as well.

The first statistical attack, the IKK attack, on Searchable Encryption schemes that leak access patterns was proposed by Islam et al. [6]. This attack uses co-occurrence counts of observed queries to determine what plaintext word(s) the client searched for.

Cash et al. [7] recognize that a recovered query inherently discloses part of the plaintext of encrypted documents and propose their Count attack as a response to the IKK attack. The Count attack uses the (relative) document occurrence counts next to the co-occurrence counts of queries to deliver better results faster as opposed to the IKK attack. Cash et al. also define different levels of leakage of SE schemes and coin the term *leakage-abuse attacks* to more broadly describe attacks that are intended to disclose information on the contents of encrypted documents in SE schemes as opposed to attacks that only disclose what the client searched for. Leakage-abuse attacks were further researched by Blackstone et al. [16].

Both the IKK attack and the Count attack are passive attacks, meaning that the adversary acts according to the protocol of the SE scheme, but tries to additionally obtain as much information and potentially runs calculations in parallel. Zhang et al. [17] show that an adversary capable of injecting files into a Searchable Encryption scheme that leaks access patterns 'is devastating for query privacy'.

VIII. Conclusion

In this paper, we revisited the IKK query recovery attack on Searchable Encryption schemes as proposed by Islam et al. [6].

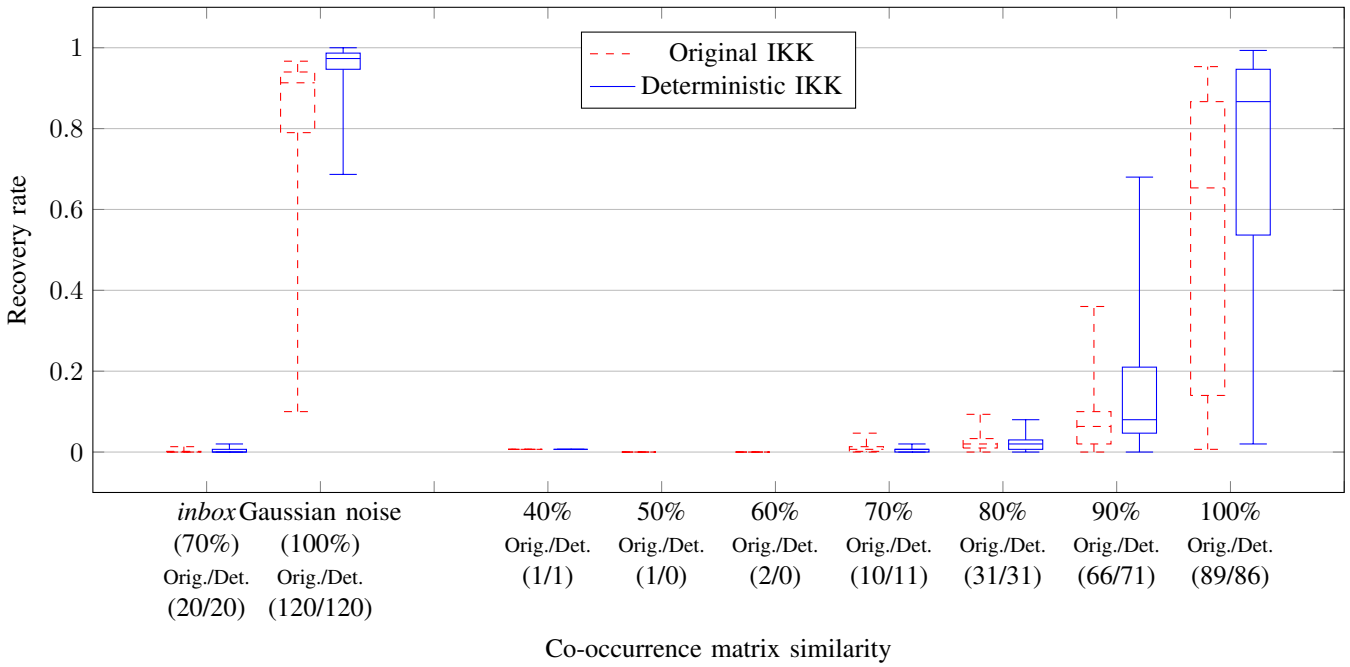

Fig. 10. Original/Deterministic IKK recovery rates and recovery rate, simulating non-perfect background knowledge using other methods

We show that the assumption that queries in a Searchable Encryption scheme follow a Zipfian (query) distribution, as Islam et al. made while simulating queries, positively influences the recovery rate of the IKK attack.

Furthermore, we show a correlation between input parameters of the IKK attack, of which the values were left unexplained by Islam et al., and the recovery rate of the IKK attack, independent of the underlying dataset used in the SE scheme. This potentially allows the adversary to optimize the parameter values using a different dataset before executing the actual attack.

We also propose improvements to the IKK attack by showing that the accuracy of the attack can be improved significantly by combining multiple attack runs, as we show that median recovery rates can be increased up to 10 percentage points, whereas the variance of recovery rates of simulation can be decreased up to 78 percentage points.

In addition to that, we show that the accuracy of the IKK attack can be increased, while the number of states visited can be decreased by incorporating deterministic components, based on notions made by Cash et al. [7] in their Count attack, to the IKK attack. The average number of states visited is decreased by 28%, the median recovery rate is shown to be increased up to 21 percentage points in different simulations, whereas its variance is decreased up to 57 percentage points.

References

[1] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*. IEEE, 2000, pp. 44–55.

[2] E.-J. Goh *et al.*, "Secure indexes." *IACR Cryptology ePrint Archive*, vol. 2003, p. 216, 2003.

[3] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," *Journal of Computer Security*, vol. 19, no. 5, pp. 895–934, 2011.

[4] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM (JACM)*, vol. 43, no. 3, pp. 431–473, 1996.

[5] G. Chen, T.-H. Lai, M. K. Reiter, and Y. Zhang, "Differentially private access patterns for searchable symmetric encryption," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 810–818.

[6] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: ramification, attack and mitigation." in *NDSS*, vol. 20. Citeseer, 2012, p. 12.

[7] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. ACM, 2015, pp. 668–679.

[8] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[9] "ENRON email dataset, version May 7th, 2015," https://www.cs.cmu.edu/ ./enron/, [Online; accessed 19-May-2020].

[10] "Apache Lucene java-user email dataset, September 2001-July 2011," http://mail-archives.apache.org/mod_mbox/lucene-java-user/, [Online; accessed 19-May-2020].

[11] M. F. Porter, "Snowball: A language for stemming algorithms," 2001.

[12] "IKK query recovery attack implementation (Python)," https://github.com/rubengrootroessink/IKK-query-recovery-attack, [Online; accessed 27-June-2020].

[13] G. K. Zipf, "Selected studies of the principle of relative frequency in language," 1932.

[14] W. Hoeffding, "Probability inequalities for sums of bounded random variables," in *The Collected Works of Wassily Hoeffding*. Springer, 1994, pp. 409–426.

[15] D. Boneh, E. Kushilevitz, R. Ostrovsky, and W. E. Skeith, "Public key encryption that allows pir queries," in *Annual International Cryptology Conference*. Springer, 2007, pp. 50–67.

[16] L. Blackstone, S. Kamara, and T. Moataz, "Revisiting leakage abuse attacks." *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 1175, 2019.

[17] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 707–720.

APPENDIX

A. IKK algorithms

In this section, we cite the Simulated Annealing (SA) algorithms as proposed by Islam et al. [6] as well as formalize our proposed algorithms for the Deterministic version of the IKK attacks. In short:

- Algorithm 1: Optimizer is used to select the initial state of the ANNEAL algorithm.
- Algorithm 2: ANNEAL is the heart of the IKK attack and is the actual algorithm that maps queries to keywords (apart from setting the initial state).
- Algorithm 3: findNextState is a sub algorithm of the ANNEAL algorithm which is included to easier show the differences to our findNextStateDet algorithm. This algorithm selects a new state for the ANNEAL algorithm to visit.
- Algorithm 4: findNextStateDet is our proposed sub algorithm of the ANNEAL algorithm which selects a new state of the algorithm more deterministically.
- Algorithm 5: reduced ANNEAL is an improved version of the ANNEAL algorithm in terms of computational complexity.

In order to easily annotate differences between the Original IKK attack (as proposed by Islam et al.) and our proposed Deterministic IKK attack we use the following colors:

- **Black** annotates lines that are present in both the Original and Deterministic IKK attack.
- **Red** annotates lines that are present in the Original IKK attack, but not in the Deterministic IKK attack. Red lines are replaced by blue lines.
- **Blue** annotates lines that are not present in the Original IKK attack, but are in the Deterministic IKK attack. Blue lines replace red lines.

The Original IKK attack and the Deterministic IKK attack are elaborated upon in Sections II-F and VI respectively.

We note that the pseudo code in the algorithms as shown below does not fully match with our implementation of the Original/Deterministic IKK attack [12]. First of all, we used Python specific methods to easily implement both attacks and, in order to reduce the number of lines we re-used as much of the code of the Original IKK attack as possible to implement our Deterministic IKK attack. We also note that the Original IKK attack can be improved by reducing the computational complexity of the while loop (Algorithm 2, lines 4-24) from $O(q^2)$ to $O(q)$, where q denotes the number of queries, which we implemented in our implementation.

In order to calculate the total squared Euclidean distance of a mapping, Islam et al., retrieve three co-occurrence counts from their corresponding co-occurrence matrices (Algorithm 2, line 14-15) and perform subtraction and squaring operations. This is done for every single cell in the query co-occurrence matrix M_c, meaning that the cost calculation has a complexity of $O(q^2)$ per loop. As matrices M_c and M_p are symmetric we can improve this by re-using squared Euclidean distances of counts that we calculated before, meaning that our complexity would be reduced to $O(n^2/2)$.

We note that the IKK algorithm changes only 1 or 2 mappings from a query to a keyword in the current state to get the next state per iteration in the while loop (Algorithm 2 lines 4-24 and Algorithm 5 lines 7-40). As a mapping between a query and keyword corresponds to a single row (and column due to the similarity of the matrices) in the query and the background knowledge co-occurrence matrices we can reduce the computational complexity by keeping the *current cost* variable outside of the while loop and only looping over the changed rows once as opposed to looping over all cells, instead of all cells. Instead of calculating the total squared Euclidean distance of both the current state and the next state we only calculate the total squared Euclidean distance of the *changes*, i.e. the distance from the current state to the total squared Euclidean distance of current state without the changed mappings and the distance from current state without the changed mappings to the next state. We can then easily calculate both *current cost* and *next cost* and E and continue the IKK algorithm as proposed by Islam et al. Our calculation reduction method reduces the complexity of the cost calculation algorithm from $O(q^2)$ to $O(q)$ and is shown in Algorithm 5.

We can loop over all queries once and calculate the cost changes (of the changed methods) as both co-occurrence matrices are symmetrical. Looping over a row is therefore the same as looping over a column. We consider the case where only one mapping was changed between current state and next state (i.e. $q_1 \mapsto k_1$ was changed to $q_1 \mapsto k_2$). To calculate the squared Euclidean distance of the mapping $q_1 \mapsto k_1$ in the current state we simply take the co-occurrence counts of q_1 with all queries in the query co-occurrence matrix and calculate the squared Euclidean distance to the corresponding co-occurrence counts of k_1 with the corresponding keywords in the background knowledge co-occurrence matrix and add these to get the total squared Euclidean distance of the row. As a query corresponds to both a row and a column in the query co-occurrence matrix we have to multiply our total squared Euclidean distance of the row by two and subtract the squared Euclidean distance of the query (and its corresponding keyword) with itself (as this value is added twice when we multiply with two). We can do the same calculation to obtain the squared Euclidean distance of the mapping $(q_1 \mapsto k_2)$ in next state. A similar calculation is needed to calculate the cost change of two changed mappings, by adding the cost change of the individual mappings together. However, we also need to twice remove the squared Euclidean distance of the changed mappings with each other (i.e. $(M_c[q_1, q_2] - M_p[k_1, k_2])^2$) as this value was added to the cost change 4 times due to our *2 operation.

Algorithm 1: Optimizer

```
input
    V : variable list                          // List of all (non-mapped) queries
    D : domain list                            // List of all (non-mapped) keywords
    K : known assignments                      // Known query-keyword mappings
    Mc                                         // Query co-occurrence matrix
    Mp                                         // Background knowledge co-oc matrix
    Q = {q: S_q}                               // Queries and their candidate keywords

initState ← {}                                 // Initial state
valList ← copy D                               // Copies values in D to variable valList
foreach var ∈ V do
    val ← random.choice(valList)               // Randomly selects a keyword from valList
    add [var ↦ val] to initState               // Adds mapping to initState
    remove val from valList                    // As query to keyword mappings are 1-to-1
end

nonAssignableQueries ← {}                      // Var. containing None-assigned queries
sortedQ ← sort(Q, key=len(S_q), ascending=True)// Sorts Q on number of candidate keywords
foreach var, S_var ∈ sortedQ do
    candKeywords ← S_var                       // Gets candidate keywords for query v
    if len(candKeywords) == 0 then
        add v to nonAssignableQueries          // Query v added to nonAssignable Queries
    else
        assignedKWs = []                       // Every candidate keyword per query is
        nonAssignedKWs = []                    // already assigned to another query or not
        foreach cand ∈ candKeywords do
            if cand ∈ valList then
                add cand to nonAssignedKWs     // cand not assigned to another query
            else
                add cand to assignedKWs        // cand assigned to another query
            end
        end
        if len(nonAssignedKWs) ≠ 0 then
            val ← random.choice(nonAssignedKWs)// Selects keyword from nonAssignedKWs
            add {var ↦ val} to initState       // Adds mapping to initState
            remove val from valList            // Query/keyword mappings are 1-to-1
        else
            foreach k ∈ assignedKWs do
                q ← initState.getByValue(k)    // Get query q, mapped to keyword k
                if k ∈ S_q then
                    remove { q ↦ k } from initState // Removes old mapping from initState
                    add { q ↦ val } to initState    // Adds new mapping to initState
                    add { var ↦ k } to initState    // Adds new mapping to initState
                    break
                end
            end
            if initState.get(var) == None then
                add var to nonAssignableQueries// If no suitable mapping could be found
            end
        end
    end
end

add K to initState                             // Adds known mappings to initState
return ANNEAL(initState, D, Mp, Mc)            // Returns result of function ANNEAL()
return ANNEAL(initState, D, Mp, Mc, nonAssignableQueries, Q)
```

Algorithm 2: ANNEAL

```
   input                                          // Simulated Annealing parameters
       initState
       D
       Mc
       Mp
       initTemperature                            // initial temperature variable
       coolingRate                                // cool down rate variable
       rejectThreshold                            // rejection rate variable
       nonAssignableQueries                       // List of None assigned queries
       Q = {q: S_q}                               // Queries and their candidate keywords

 1 currentState ← initState                       // Search continues until temp. reaches 0
 2 succReject ← 0                                 // or the system is frozen (no new state
 3 currT ← initTemperature                        // is accepted for large number of times)
 4 while (currT ≠ 0 and succReject < rejectThreshold) do
 5     currentCost ← 0
 6     nextCost ← 0
 7     nextState ← findNextState(currentState, D)  // Selects a nextState
 8     nextState ← findNextStateDet(currentState, D, Q)  // Selects a nextState deterministically
 9     foreach cells i,j in Mc do
10         (i,k) ← currentState.get(i)            // Queries i,j correspond to cells in query
11         (i,k') ← nextState.get(i)              // co-occurrence matrix. Keywords k & l
12         (j,l) ← currenState.get(j)             // and k' & l' are mapped to i & j in
13         (j,l') ← nextState.get(j)              // currentState and nextState resp.
14         currentCost += (Mc[i,j]-Mp[k,l])^2     // Diff. in squared Euclidean distance
15         nextCost += (Mc[i,j]-Mp[k',l'])^2      // between co-occurrence counts calculated
       end
16     E = nextCost - currentCost                 // If nextState is better than currentState,
17     if E < 0 then
18         accept new state                       // nextState is accepted, if not
       else
19         accept new state with prob. exp(-E/currT)  // nextState accepted with low probability
       end
20     if new state is accepted then
21         succReject ← 0
22         currentState ← nextState
       else
23         succReject++
       end
24     currT = coolingRate*currT                  // temperature is decremented each loop
   end
25 foreach query ∈ nonAssignableQueries do
26     add {query ↦ None} to currentState         // Maps non-assignable queries to None
   end
27 return currentState
```

Algorithm 3: findNextState

```
input
    currentState                          // 1-to-1 mapping of all queries to keywords
    D                                     // A list of all possible keywords

nextState ← copy currentState
{x ↦ y} ← random.choice(nextState)
y' ← random.choice(D), y ≠ y'
remove {x ↦ y} from nextState
add {x ↦ y'} to nextState
if {z ↦ y'} ∈ currentState then
    remove {z ↦ y'} from nextState        // If y' is already mapped to query z
    add {z ↦ y} to nextState              // Map query z to y instead of y'
end
return nextState
```

Algorithm 4: findNextStateDet

```
input
    currentState                          // 1-to-1 mapping of all queries to keywords
    Q = {q: S_q}                          // Queries and their candidate keywords
    D

nextState ← copy currentState
{x ↦ y} ← random.choice(nextState)

S_x ← Q.get(x)                            // Gets candidate keywords for query x
cands ← S_x.remove(y)                     // Keyword y cannot be selected again
y' ← None                                 // Initializes y'
if len(cands) ≠ 0 then
    y' = random.choice(cands)             // Selects random keyword from candidates
else
    return currentState                   // No new mapping could be found
end
count ← 0                                 // Initializes count variable
while {z ↦ y'} ∈ currentState and y ∉ S_z do
    y' = random.choice(cands)             // Selects a new candidate keyword y'
    if len(count) ≤ 2 * len(D) then
        return currentState               // No new mapping could be found
    end
    count += 1
end
remove {x ↦ y} from nextState
add {x ↦ y'} to nextState
if {z ↦ y'} ∈ currentState then
    remove {z ↦ y'} from nextState        // If y' is already mapped to query z
    add {z ↦ y} to nextState              // Map query z to y instead of y'
end
return nextState
```

Algorithm 5: Cost calculation reduced ANNEAL

```
input
    initState
    Mc
    Mp

currState ← initState                          // Initializes currState
currCost ← 0                                   // 'cost' variable of currState
foreach cells i,j in Mc do
    (i,k) ← currState.get(i)                   // We calculate the total squared Euclidean
    (j,l) ← currState.get(j)                   // distance of the currState/initState
    currCost += (Mc[i,j]-Mp[k,l])^2            // mapping once
end
while do
    nextState ← findNextState(X)               // Finds the nextState of the algorithm
    q1, q2 ← findDiff(currState, nextState)    // Remapped queries, q2 can be None
    nextCost ← 0                               // 'cost' of nextState is initialized at 0
    currCostChange ← 0                         // Variables containing the cost of q1 and
    nextCostChange ← 0                         // q2 (if not None) in current/next states
    queries ← currState.getKeys()              // Returns all the queries in the system
    foreach j ∈ queries do
        (q1, k1) ← currState.get(q1)           // Gets mapping of q1 in currState
        (q1, k1') ← nextState.get(q1)          // Gets mapping of q1 in nextState
        if q2 ≠ None then
            (q2, k2) ← currState.get(q1)       // Gets mapping of q2 in currState
            (q2, k2') ← nextState.get(q1)      // Gets mapping of q2 in nextState
        end
        (j,l) ← currState.get(j)               // Gets mapping of j in currentState
        (j,l') ← nextState.get(j)              // Gets mapping of j in nextState
        currCostChange += (Mc[q1,j]-Mp[k1,l])^2   // Diff. in squared Euclidean distance q1
        nextCostChange += (Mc[q1,j]-Mp[k1',l])^2  // between co-occurrence counts calculated
        if q2 ≠ None then
            currCostChange += (Mc[q2,j]-Mp[k2,l])^2   // Diff. in squared Euclidean distance q1
            nextCostChange += (Mc[q2,j]-Mp[k2',l])^2  // between co-occurrence counts calculated
        end
    end
    currCostChange ← currCostChange * 2        // (*2) due to matrix symmetry and therefore
    nextCostChange ← nextCostChange * 2        // rows/columns being the same
    currCostChange -= (Mc[q1,q1]-Mp[k1,k1])^2  // Removes co-oc count of q1 with itself
    nextCostChange -= (Mc[q1,q1]-Mp[k1',k1'])^2  // added twice due to (*2)
    if q2 ≠ None then
        currCostChange -= (Mc[q2,q2]-Mp[k2,k2])^2   // Removes co-oc count of q2 with itself
        nextCostChange -= (Mc[q2,q2]-Mp[k2',k2'])^2   // added twice due to (*2)

        currCostChange -= 2*(Mc[q1,q2]-Mp[k1,k2])^2   // Removes co-oc count of q1 with q2
        nextCostChange -= 2*(Mc[q1,q2]-Mp[k1',k2'])^2  // added 4 times instead of 2 (due to *2)
    end
    nextCost = currCost - currCostChange + nextCostChange
    E = nextCost - currCost
    if E < 0 then
        accept new state                       // nextState is accepted, if not
    else
        accept new state with prob. exp(-E/currT)   // nextState accepted with low probability
    end
    if new state is accepted then
        currState ← nextState
        currCost ← nextCost
    end
end
```

B. Co-occurrence similarity

In this section we briefly show the correlation between the different methods to generate non-perfect background knowledge for our simulations and our proposed co-occurrence matrix similarity metric for the interested reader. We recap these methods as follows:

- Keyword percentage
- Document percentage
- Different, but similar dataset
- Adding Gaussian noise

The first method, keyword percentage, uses only a certain percentage of the keywords from the keyword set while generating inverted indices and co-occurrence matrices. The disregarded keywords are replaced by keywords that have a lower occurrence count. The keyword percentage method to generate non-perfect background knowledge changes entire rows (and columns) corresponding to a keyword and thus also the corresponding co-occurrence counts in the cells of these rows (and columns). Our co-occurrence matrix similarity metric does not look at the co-occurrence counts of words that are not represented in the (non-querified) query and the background knowledge co-occurrence matrices and simply multiplies the similarity of co-occurrence counts by the relative number of words that are represented in both matrices. The correlation between the keyword percentage method and our proposed co-occurrence matrix similarity metric is therefore $y = x$, meaning that if we take 90% as the keyword percentage the co-occurrence similarity will be 90% as well.

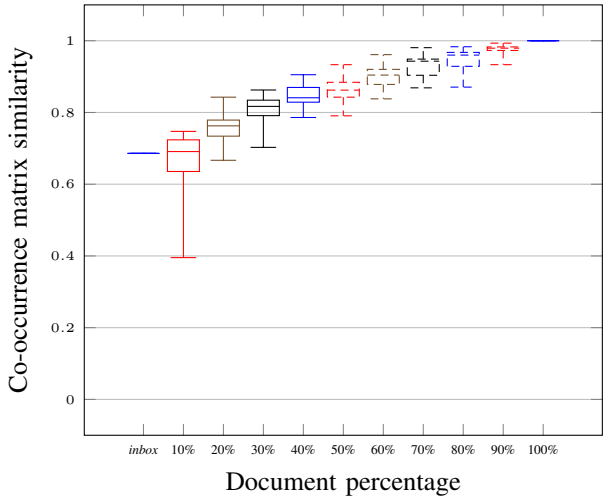

Fig. 11. Correlation between percentage of documents (user folders) and co-occurrence matrix similarity

The second and third methods, document percentage and using a different, but similar dataset to simulate non-perfect background knowledge are expressed in Figure 11. The document percentage method takes only a certain percentage of the user folders in a dataset, as the ENRON dataset [9] is split up into the email folders of different users, while generating the (non-perfect) background knowledge co-occurrence matrix instead of all the user folders. The 'different, but similar dataset' method generates the background knowledge co-occurrence matrix in the same way, but instead of taking a percentage of the user folders in the original dataset (*ENRON/_sent_mail*) it takes 100% of the user folders in a different, but similar dataset (*ENRON/inbox*). Figure 11 shows the aggregation of 20 different simulations per box plot. First of all the *inbox* column shows the 'different, but similar dataset' method at a consistent co-occurrence matrix similarity of 70%. The actual similarity between two dataset is, of course, dependent on the actual datasets used. The document percentage method shows a somewhat (sub-)linear correlation with higher variances in the similarity score when a lower percentage of user folders is considered. This makes sense as the number of files in the user folders differs per user and different users use different writing styles.

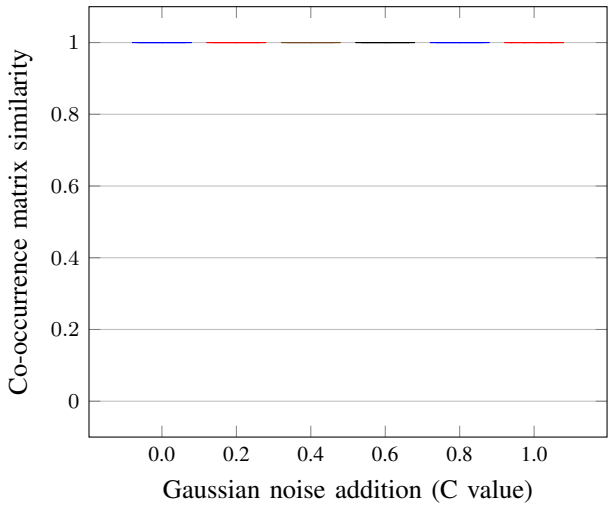

Fig. 12. Correlation between Gaussian noise added and co-occurrence matrix similarity

The last method, adding Gaussian noise to the background knowledge co-occurrence matrix, was first used by Islam et al. [6] to simulate non-perfect background knowledge. We see that the value of C (the noise scaling factor) does not influence the co-occurrence matrix similarity at all. All similarity scores (are close to) 1 as the keywords on the axis of the background knowledge co-occurrence matrix are not changed and the co-occurrence counts in the cells of the matrix are only changed slightly. The different box plots in Figures 11 and 12 are the aggregation of 20 different simulations.

C. Recovery rate

In this Appendix we briefly elaborate on the correlation between the different methods to generate non-perfect background knowledge (as explained in Appendix B) and the recovery rate.

First of all we see the correlation between the keyword percentage and the recovery rate in Figure 13. This graph shows a somewhat linear correlation between the different box plots (each aggregated of 20 simulations), although we still see

a lot of variance. It makes sense that the correlation is quite linear as the IKK attack is still quite capable of mapping the right queries and keywords as the co-occurrence counts of different queries/keywords are not changed, only a number of keywords is replaced by another set of keywords. The keywords that were not replaced are therefore quite likely to be mapped correctly to their corresponding keyword.

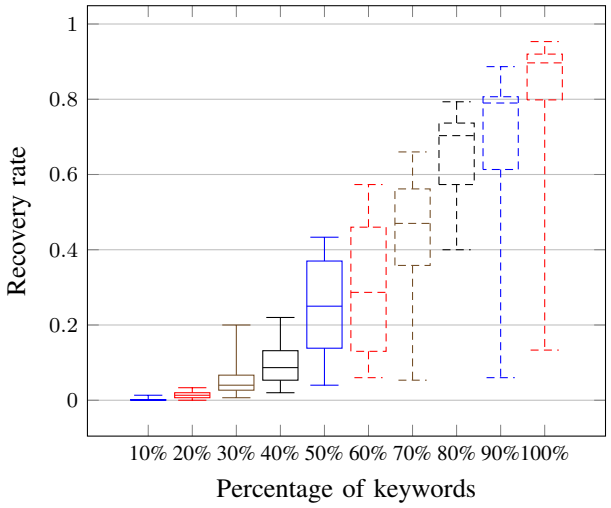

Fig. 13. Correlation between percentage of keywords and recovery rate

Figure 14 shows the correlation between the document percentage (percentage of user folders) method and the recovery rate, as well as the correlation between the 'different, but similar dataset' method and the recovery rate (*inbox*). We see that using the different dataset, or using a low percentage of user folders returns a low recovery rate, whereas only 100% of the documents returns a recovery rate close to 1. All box plots in between show a high variance. This makes sense as the document percentage method ensures that the keywords in the matrices do not overlap as well as influence the co-occurrence counts in the individual cells of the background knowledge co-occurrence matrix. As the user folders each contain different numbers of documents and different users have different writing styles we argue that which user folders are regarded in the background knowledge influences the recovery rate of the IKK attack a lot, even with the same percentage of user folders regarded. The box plots in Figure 14 all denote the aggregation of 20 simulations.

Figure 15 shows the correlation between the last method to generate non-perfect background knowledge, adding Gaussian noise to the co-occurrence counts in the background knowledge co-occurrence matrix, and the recovery rate. Gaussian noise is added symmetrically, i.e. the same Gaussian noise is added to the co-occurrence counts in the row as is added to the counts in the column corresponding to the same query. The Gaussian noise is generated using the same method that Islam et al. describe in [6], where a normal distribution is used together with a C parameter. This C parameter, the 'noise scaling factor', determines the maximum level of Gaussian

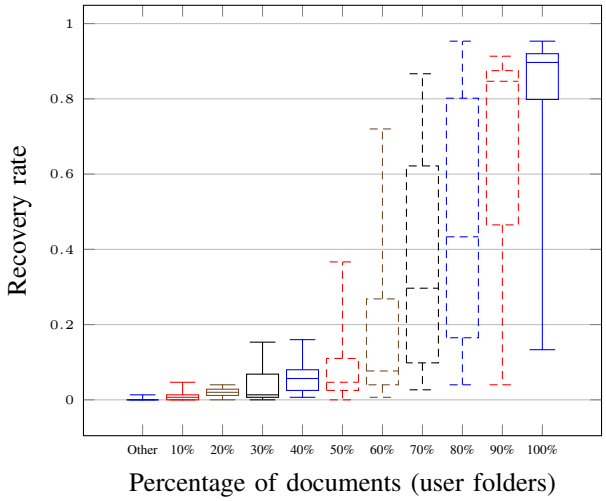

Fig. 14. Correlation between percentage of documents (user folders) and recovery rate

noise added to a single co-occurrence count. A higher C value allows for the addition of more Gaussian noise. We do not see a clear correlation between the value of C and the recovery rate. Each box plot in Figures 13, 14 and 15 is the aggregation of 20 simulations.

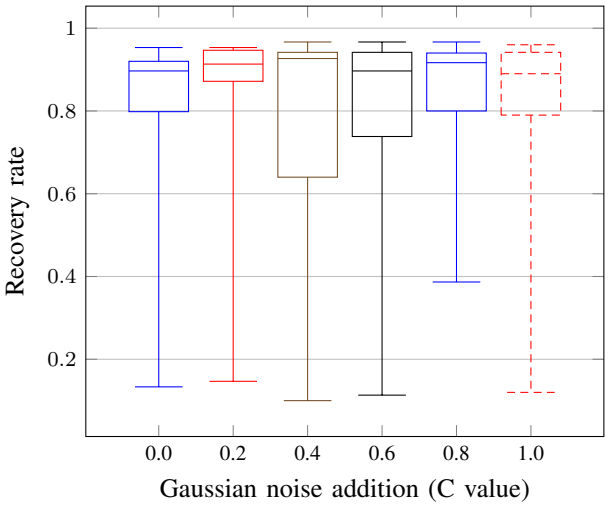

Fig. 15. Correlation between Gaussian noise added and recovery rate