Searchable Symmetric Encryption Attacks

More power with more knowledge

B.I.Y.L. Ho

Delft University of Technology

TUDelft

Searchable Symmetric Encryption Attacks

More power with more knowledge

by

B.I.Y.L. Ho

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday December 7, 2023 at 10:00 AM.

Student number:	4320867	
Project duration:	January 3, 2023 – December 7, 2023	
Thesis committee:	Prof. G. Smaragdakis,	TU Delft, thesis advisor
	Dr. K. Liang,	TU Delft, daily supervisor
	Dr. J. Decouchant,	TU Delft
Daily co-supervisor:	H. Chen,	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Preface

The preface is the beginning of my thesis, and also near the end of my journey as a master's student at TU Delft. As a student, I have experienced many challenges and met many wonderful people who have contributed to my growth as a person.

I would like to present my thesis project on *Searchable Symmetric Encryption Attacks* which I started back in January. First, I would like to thank Dr. Kaitai Liang for supervising me, and guiding me through this project, without his assistance, this work would not have been possible. Furthermore, many thanks to my daily co-supervisor Huanhuan Chen who dedicated some of his time discussing topics with me during the meetings. In addition, I would like to thank Prof. G. Smaragdakis for his excellent feedback during the first stage and greenlight review, and for his participation in my thesis committee. Particularly, his patience and support in helping me arrange the defense date with all committee members and the Board of Examiners was crucial in making this possible. Moreover, I would like to thank Dr. J. Decouchant for his time and flexibility to participate in my thesis committee. Finally, I would like to thank my family and friends, that have supported me since the beginning of my studies at TU Delft.

B.I.Y.L. Ho
Delft, November 2023

Abstract

A searchable symmetric encryption (SSE) scheme allows a user to securely perform a keyword search on an encrypted database. This search capability is useful but comes with the price of unintentional information leakage. An attacker abuses leakage to steal confidential information by launching SSE attacks. In this work, our goal is to design a new inference attack that improves the query recovery accuracy of an existing attack. We combine an additional volume leakage pattern and investigate the effectiveness of existing countermeasures against it. Our attack utilizes similar data knowledge and known queries to perform the attack. The results show that usage of an additional volume leakage pattern results in an improved query recovery accuracy, and a more stabilized spread in the results. When an attacker knows up to 4 known queries, we observe an improved query recovery accuracy between 5 and 19.5%. Furthermore, we investigate if the attack can be improved even further by utilizing clustering. However, the results are too close with a high trade-off in performance. From our findings, we can generalize that additional knowledge available to the attacker improves query recovery accuracy. More leakage combinations and their impact are open to future research.

Contents

Preface i

Abstract ii

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Research Question 2
- 1.3 Contribution 2
- 1.4 Thesis structure 3

2 Background 4
- 2.1 Encrypted Search Algorithms 4
- 2.2 Searchable Symmetric Encryption 4
  - 2.2.1 Static SSE 5
  - 2.2.2 Dynamic SSE 5
- 2.3 SSE Attack Model 5
- 2.4 Adversary's capabilities 7
- 2.5 Adversary's knowledge 8
  - 2.5.1 Document knowledge 8
  - 2.5.2 Query knowledge 8
- 2.6 Adversary's target and accuracy 9
- 2.7 Leakage patterns 9
  - 2.7.1 Access pattern 9
  - 2.7.2 Co-occurrence pattern 9
  - 2.7.3 Result length pattern 10
  - 2.7.4 Search pattern 10
  - 2.7.5 Volume pattern 11
  - 2.7.6 Total Volume pattern 12
- 2.8 Leakage Profiles 12

3 Related work 14
- 3.1 IKK attack 14
- 3.2 Count attack 15
- 3.3 Search 15
- 3.4 Graph Matching 16
- 3.5 Volan 16
- 3.6 SelVolan 17
- 3.7 Subgraph 17
- 3.8 LEAP 17
- 3.9 Score 18
- 3.10 RefScore 18
- 3.11 Overview of the attacks 19

4 Revisiting Score and RefScore 20
- 4.1 The Score Attack 20
- 4.2 The RefScore Attack 21
- 4.3 Improvement intuition and discussion 22
  - 4.3.1 Frequency and result length 22
  - 4.3.2 Refspeed parameter 22
  - 4.3.3 Clustering 23
  - 4.3.4 Volume pattern 23

5 VolScore Attacks **24**
5.1 Notation . . . 24
5.2 Assumptions . . . 25
5.3 The design idea . . . 25
5.4 VolScore . . . 26
5.4.1 Preparing covolume matrices . . . 27
5.4.2 Scoring mechanism . . . 29
5.4.3 Decision making . . . 29
5.5 RefVolScore . . . 30
5.6 ClusterVolScore . . . 31
5.7 Countermeasures . . . 34
5.7.1 Padding . . . 34
5.7.2 Volume hiding . . . 35

6 Experiments **36**
6.1 Datasets . . . 36
6.1.1 Enron dataset . . . 36
6.1.2 Apache Lucene . . . 36
6.1.3 Wikipedia . . . 36
6.1.4 Dataset comparison . . . 39
6.2 Methodology . . . 39
6.2.1 Keyword extraction . . . 39
6.2.2 Query selection . . . 40
6.2.3 Adversary's knowledge . . . 40
6.2.4 Evaluation . . . 40
6.2.5 Hardware and implementation . . . 40
6.3 Results . . . 40
6.3.1 Score and RefScore . . . 40
6.3.2 RefScore and VolScore attacks . . . 41
6.3.3 Low known queries comparison . . . 42
6.3.4 RefSpeed comparison . . . 44
6.3.5 Countermeasures . . . 46

7 Discussion **49**
7.1 Key findings . . . 49
7.2 Interpretation and implications . . . 49
7.3 Limitations . . . 50
7.3.1 Dataset in-depth analysis . . . 50
7.3.2 Known queries . . . 50
7.4 Future work . . . 51

8 Conclusion **52**

Reference **54**

List of Figures **57**

A Apache Lucene dataset **59**

1 Introduction

In this chapter, we describe the motivation behind this work. The problems that concern storing sensitive data on the cloud, and how searchable symmetric encryption can provide confidentiality and the ability to access selective data. Furthermore, we formulate the main research question and sub-questions as the focus of this paper. We end this chapter with the contribution of our work and outlining the thesis structure.

1.1. Motivation

It is common nowadays that the majority of organizations store sensitive data on the cloud [24]. Storing data on the cloud provides the availability of the data, which makes the data easily accessible from outside and guarantees data uptime. Redundancy and backup [31] is also common in which data can be easily recovered when a backup is stored offsite.

One of the downsides of storing data on the cloud is that a third party is involved, and we have less control over who has access to the data. This becomes even more troublesome when sensitive data is concerned. This provides an incentive for cybercriminals to steal sensitive data on the cloud by launching cyber attacks. The stolen confidential data is then sold on the dark web. Therefore, confidentiality becomes critical to keep our data secure from unauthorized access. Instead of storing the data in plain text, encryption is used on the data before it is uploaded to the cloud.

By encrypting all our data, our data remains secure on the cloud when encryption is applied correctly. However, there are always trade-offs. When the data is fully encrypted, it can not be read and accessed as easily and becomes hard to keep track of. After a client has stored the data securely, eventually, the client would like to retrieve a part of the data. One solution would be to download all the encrypted data stored on the cloud server, and then decrypt it locally to access the original data. However, this is not a feasible approach when the stored data size could be very large when the client is only interested in a small set of documents.

One solution is to use a searchable encryption scheme to obtain the ability to search on encrypted data. Searchable Symmetric Encryption (SSE) is a pioneering work from Song et al. [37] which is a single writer and single reader searchable encryption scheme. This work motivated other researchers to expand into this subject, such as to a multiple writer and single reader scenario using public key encryption (PKE) as done by Boneh et al. [4].

In our previous problem, the client had no search capability and therefore had to download all the data from the cloud. By utilizing SSE, the client is able to perform a single keyword search on encrypted data stored on the cloud. After a search, only the encrypted files that contain the keyword are returned back to the client. SSE has the term symmetric because symmetric keys are used for encryption and decryption. The advantage is in speed and efficiency, compared to using asymmetric keys as used in a PKE scheme. The client now retains the confidentiality aspect, while also gaining the ability to selectively search for a subset of documents.

Unfortunately, there is no free lunch for the client. The ability to perform a single keyword search on encrypted data by construction hides the keyword itself, but sending the query and retrieving the data also leaks some information unintentionally. For example, an attacker can observe which encrypted files were returned when a search was performed. This leakage can be used by an attacker to perform an attack on SSE systems. This motivates us to perform more research on this topic to understand the potential of such attacks, to design an improved attack, and to research how to guard against such an attack.

1.2. Research Question

To improve our understanding of attacks on SSE systems and to design an improved attack, we first perform a literature review. Once we have a better understanding of past SSE attacks, only then can we attempt to design an improved attack. This brings us to the following main research question for this thesis:

RQ: *How can we design an improved inference attack with a higher query recovery rate, and how can we guard against it?*

To answer the main research question, we have decomposed this into sub-questions. And finding an answer to these sub-questions will answer the main research question. The sub-questions are as follows:

SQ1: *What can we learn from past attacks on SSE systems, and how do they differ from each other?*

SQ2: *How can we combine techniques from past SSE attacks to improve the recovery rate of an inference attack?*

SQ3: *How effective are existing countermeasures against the improved attack?*

Sub-question 1 is answered by performing research on related work on past SSE attacks, and by comparing them with each other.

For sub-question 2 we choose an existing inference attack as a baseline and investigate its algorithm in detail to determine possibilities to improve this attack. Afterward, we attempt to design an improved attack.

We answer sub-question 3 by reviewing existing countermeasures from the literature on SSE attacks and determining which countermeasures to apply in the experiments for the new attack. The results from the experiments will show which countermeasures are effective against the improved attack.

1.3. Contribution

In this work, we have the following contributions.

Firstly, the main intuition behind this work is that when the attacker has more knowledge, an existing attack can be improved further. We implemented this concept by combining the volume leakage pattern with the co-occurrence pattern and designed an improved attack on the Refined Score attack [12]. By doing so, we managed to improve the attack on two aspects. We improved the query recovery rate in the setting where the attacker has a low amount of known queries available. The second improvement is a more stabilized spread of results when using the additional volume leakage pattern.

Secondly, we explored the idea of utilizing clustering to select a nonstatic amount of predictions by using a dynamic refinement speed to improve the query recovery rate even further. This is a different approach than the clustering method the original author explored. We return a single candidate for each prediction instead of a cluster of candidates.

This idea did not manage to meet our expectations, because on average the chosen refinement speeds were far lower than we expected which resulted in minimal gains in query recovery accuracy with high runtime. This creates a false sense of improvement when the other attacks are running with less precision on a higher refinement speed.

Finally, we evaluated the performance of our new attack against the original attack using different datasets and investigated the effectiveness of countermeasures against our new attack.

1.4. Thesis structure

This thesis is structured as follows.

In **chapter 2** we provide in-depth background to SSE. We describe SSE itself, the different types of SSE attacks, the simplified attack model, the capabilities and knowledge of an adversary, and the leakage patterns that are utilized for the attacks.

In **chapter 3** we provide an overview of the related work that we reviewed, their main differences in type, knowledge, and exploited leakage patterns.

In **chapter 4** we revisit the original Score and Refined Score attack algorithms and provide our initial intuitions and ideas to improve this attack further.

In **chapter 5** we describe our new attack design and procedure and briefly discuss potential countermeasures.

In **chapter 6** we show the results of the experiments that we conducted on the original and new attacks.

In **chapter 7** we discuss the results we obtained from our experiments by listing our key findings, interpretation and implications. Furthermore, we evaluate the limitations of our analysis and provide future work possibilities.

In **chapter 8** we conclude this work by answering the research question and sub-questions.

2
Background

In this chapter, we describe the background of SSE. Furthermore, we illustrate the capabilities of an adversary in the types of attack that can be performed, and what kind of knowledge may be required to initiate a SSE attack. In addition, the leakage patterns are explored, which is an important aspect that is exploited in SSE attacks.

2.1. Encrypted Search Algorithms

Previously, we have briefly described searching for encrypted data in the introduction by using SSE to perform a single keyword search. This is only one way of using an encrypted search algorithm (ESA). The topic of ESA is very broad, and there are also other ways to search on encrypted data. Depending on the use case, fully homomorphic encryption (FHE) [16, 28] is also an option in which it is possible to do mathematical computations like addition and multiplication on encrypted data, without revealing the data itself and preserves the privacy [40]. However, the speed and efficiency is a drawback.

Another option would be to use Oblivious RAM (ORAM) [17, 15] that makes it possible to search on encrypted data while hiding the access pattern leakage, but again we trade in performance. Furthermore, there is also a property preserving encryption (PPE) scheme [2, 29, 33] in which certain properties of the plaintext is preserved in the ciphertext. The property can be tested and computed on the plaintext, as well as on ciphertext, where both give the same result. This can used to search and decide based on the results of the tests. But, the drawback is that some information is leaked that can be abused to perform an attack.

2.2. Searchable Symmetric Encryption

Structured Encryption (STE) [8, 21] is another ESA that makes it possible to encrypt structured data in such a way, that data can be retrieved by using a query token. SSE is a special case of STE, in which we encrypt a document collection, and then perform searches on the encrypted documents. In SSE it uses a symmetric key for encryption and decryption, and there are many different types of query support. The simplest type is the single keyword search [37]. But there are also other variants in which other operations are allowed, such as conjunctive boolean search queries [6, 27], and ranged search queries [34, 41].

There is also ESA that uses asymmetric encryption [1, 4] instead of symmetric encryption. In asymmetric encryption, the sender encrypts a message with the public key of the receiver, and the private key of the receiver is used for decryption. In this case, it is easier to share files privately and securely among multiple users, but the performance and efficiency are lower compared to SSE.

We have seen that there are many different types of ESA. Each scheme has a different trade-off, between query expressiveness, efficiency, and security [5]. In this work, the focus is on attacks on SSE systems. More specifically, attacks on single keyword search SSE. This means that query expressiveness only supports searching for one single keyword. In SSE we distinguish between two types: static

SSE and dynamic SSE.

2.2.1. Static SSE

A static SSE scheme [11, 37] contains a collection of five polynomial time algorithms [11]:
Static SSE = (Gen, Enc, Trpdr, Search, Dec).

1. Gen(1^k): An algorithm with security parameter k as input, and outputs a symmetric key K.
2. Enc(K, D): An encryption algorithm with symmetric key K and document collection $D = (d_1, d_2 ..., d_n)$ as input, and outputs an encrypted index I and encrypted documents $ED = (ed_1, ed_2, ...ed_n)$.
3. Trpdr(K, w): An algorithm with symmetric key K and keyword w as input, and outputs a trapdoor t.
4. Search(I, ED, t): A deterministic algorithm with encrypted index I, encrypted documents ED, and trapdoor t as input, and outputs a sequence of document identifiers.
5. Dec(K, ed_i): A deterministic algorithm with symmetric key K and encrypted document ed_i as input, and outputs a document d_i.

The SSE scheme allows a user to upload encrypted documents with an encrypted index to a cloud server. Furthermore, it allows a client to securely search and retrieve specific documents from the server while keeping the search query and documents private. The client creates a trapdoor (also known as search token) using the Trpdr function and sends this to the server. The server utilizes the Search function to search within the encrypted index and returns the document identifiers back to the client. The client can now retrieve the encrypted documents and use the decryption function Dec locally to successfully recover the requested files.

2.2.2. Dynamic SSE

The problem with static SSE is that once the encrypted index is uploaded, it cannot be changed anymore. The client has to generate a new encrypted index and encrypt all files, before uploading again to the server. This is not efficient, and therefore a new solution called dynamic SSE (DSSE) is more suited. A DSSE scheme [23, 9, 41, 22] allows adding and removing files from the server. In addition to the previous five polynomial time algorithms, there are four more algorithms [23] as specified below.

6. AddToken(K, d_i): An algorithm with symmetric key K and document d_i as input, and outputs an add token τ_a and an encrypted document ed_i.
7. DelToken(K, d_i): An algorithm with symmetric key K and document d_i as input, and outputs a delete token τ_d.
8. Add(I, ED, τ_a, ed_i): An algorithm with encrypted index I, encrypted documents ED, add token τ_a, encrypted document ed_i and outputs a new encrypted index I', and new encrypted documents ED'.
9. Del(I, ED, τ_d): An algorithm with encrypted index I, encrypted documents ED, delete token τ_d as input, and outputs a new encrypted index I', and new encrypted documents ED'.

The functions AddToken and DelToken are run by the client since only the client has access to the secret key. The algorithms Add, Del algorithms are run by the server upon receiving the request and the tokens from the client.

2.3. SSE Attack Model

For the model, it is assumed that the adversary is honest-but-curious. This means that the attacker will follow the protocols correctly, but has a hidden motive and tries to learn as much information as possible. Information such as documents, keywords, and trapdoors are possible learning goals. The adversary can be seen as the untrusted cloud server, where the encrypted data is stored, or as a third party that has access to the communication line between client and server.

We follow a simplified model [19] in order to setup and use an SSE scheme. This model is important for the attacks that we will discuss later in this paper. The client has a document set that will be stored

on the cloud server. First, the client has to extract keywords from each document from the document set. These keywords become the vocabulary that will be used to perform searches.

The keywords and documents are stored together in an index. This index is a table where for each document, there is a list of keywords that were extracted from the document. Instead of storing it this way, the usage of an inverted index is more common to achieve sublinear search time [11]. In an inverted index, there is a binary matrix with *i* rows as keywords, and *j* columns as documents. Each cell *(i,j)* contains a 0 or a 1 value. The value of 1 indicates that the keyword is found inside the document, and otherwise, 0 is stored.

Each row of this inverted index is encrypted independently with a symmetric key, and stored together with a trapdoor value that will be used for pattern matching during the search. Each document is also encrypted with a symmetric key, and then both encrypted documents and the encrypted index are uploaded to the cloud server. See Figure 2.1 for the SSE setup.

Figure 2.1: SSE Setup

After the data is uploaded to the cloud server, the client can perform a keyword search, also known as a search query. The client has access to the symmetric key, and uses this key together with a keyword as input for a trapdoor function. The trapdoor function outputs a trapdoor, which is an encrypted keyword and is sent to the server.

The server performs a search on the encrypted index by pattern matching based on the trapdoor value. Once a match is found, the encrypted row from the encrypted index is returned to the client. The client can now decrypt locally with the symmetric key, and obtain the document identifiers that contain the keyword from the search. So in the last step, the client will request these encrypted documents from the server, and decrypt the documents locally. This process is slightly simplified in Figure 2.2.

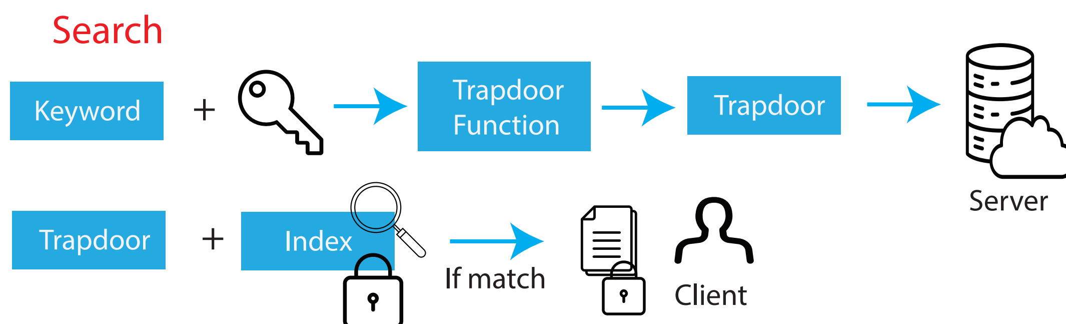


Figure 2.2: SSE Search

As we can see there are many actions from the client itself for the setup, and also interactions between the client and the server during the search. For the attacks that we discuss in this paper, the main focus is on the search phase. As described in [19], we can simplify the attack model as a two-step process, see Figure 2.3.

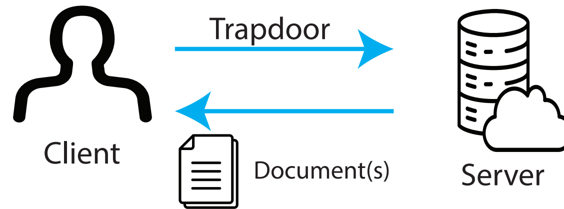


Figure 2.3: Simplified Attack Model

In the first step, the client sends a trapdoor, which can be seen as an encrypted keyword to the server. And in the second step, the server delivers the documents back that contain this keyword. During this process, the server has not learned the actual keyword behind the trapdoor, or the contents of the documents. This model simplification is essentially the view of an attacker that has access to the communication line between the client and the server, therefore the additional communication steps between client and server can be omitted.

Implicitly, it is assumed here that both parties follow the protocols correctly and that some information is leaked during the search to enable efficient keyword search. A very common leakage pattern is the access pattern where document identifiers are leaked. In such a case, the attacker learns which trapdoor and encrypted documents belong to each other. As discussed before in section 2.1, it is possible to hide this access pattern, but this might not be feasible for practical use.

This simplified model is the basis for how many SSE attacks will start, based on some knowledge that is leaked here during this two-step interaction. An attacker records this interaction for many {trapdoor, documents} pairs, and with this knowledge, and possibly other types of knowledge, an attack could be launched to derive even more information than is intended originally by the client. We discuss the adversary's capabilities and knowledge in more depth in the following sections.

2.4. Adversary's capabilities

We have discussed the simplified attack model and in this section, we describe the different types of attacks that an adversary can execute. In the SSE literature there are 3 common types of attacks: Leakage Abuse Attacks (LAA), Inference Attacks, and Injection attacks, see Figure 2.4.

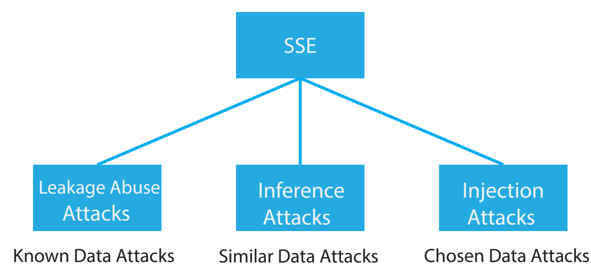


Figure 2.4: SSE Attack Types

LAA is also referred to as known data attacks, in which the adversary commonly has a high amount or even full knowledge about the stored data. Many attacks are of this type, and we will discuss these attacks in chapter 3 Related work. Although these attacks might not be fully practical due to strong assumptions, they are still important to investigate to what extent an adversary can exploit such SSE systems.

The second type is inference attacks, also known as similar data attacks. In this type, the attacker has access to a similar dataset that is similar to the stored data on the server. It does not have to be a subset, instead, the distribution of keywords and documents from a similar dataset is close to the real dataset. This is a weaker assumption compared to the previous attack type, and our new attack is this

attack type. We will revisit a similar data attack called the Score Attack in chapter 4 Revisiting Score and RefScore.

The final attack type is injection attacks, also known as chosen data attacks. In this type of attack, the assumption is that the adversary has the capability to inject some files into the server to perform the injection attack. This is beyond the scope of this paper, and we have not investigated these attacks.

2.5. Adversary's knowledge

Before an attacker performs an SSE attack, the adversary commonly has some prior knowledge. The assumption that the adversary has zero knowledge is too strict, and often they do know some information. In SSE attacks, the most common types of knowledge an adversary may have are document knowledge and query knowledge.

2.5.1. Document knowledge

The document knowledge is common among SSE attacks. This means that the adversary has some information about the stored documents on the cloud server. Depending on the attack and the assumptions, the adversary may require full knowledge, partial knowledge, similar knowledge, or keyword knowledge.

In the full knowledge assumption, the attacker has a complete copy of the dataset that is stored on the server. It is mostly for theoretical purposes because in practice it is unlikely that an attacker has the full document knowledge. Furthermore, when an attacker already knows all the information about the stored documents, there is not much incentive for an attacker.

For partial knowledge, this means that the attacker has a subset of the real dataset. This could have been obtained in a variety of ways, for example, partial data that was obtained by the attacker during a data breach.

In the similar document knowledge setting, the adversary has a dataset that is similar to the real dataset in terms of keyword and document distribution, but the documents themselves do not have to be a subset of the real dataset. An example would be outdated documents that have been phased out and removed from the server, however, the new documents are still very similar to the old ones that are obtained by the adversary. This is a weaker assumption compared to the previous ones.

In the keyword knowledge setting, the adversary knows something about the keywords that are stored inside the documents. This means the attacker might know from which field or profession these keywords are, which will limit the possible keyword space to a specific category.

2.5.2. Query knowledge

Apart from document knowledge, the other auxiliary information is query knowledge. Queries are trapdoors that a client sends to the cloud server to retrieve documents, as seen previously in Figure 2.3. The attacks that we have investigated require either no query knowledge, partial knowledge, or query search pattern knowledge.

In the first case of no query knowledge, the attacker does not know which keyword corresponds to any of the sent trapdoors. This could be the case when an attack does not require this query knowledge to launch the attack.

In the second case for partial knowledge, the attacker knows a part of all the trapdoors. This knowledge can then be used to derive even more information during the attack. None of the attacks that we have investigated require full knowledge of the queries.

In the final case, we have query search pattern knowledge. In this case, the adversary has access to the search pattern of the queries. This means that over a period of time, the attacker knows how often certain keywords are searched for.

2.6. Adversary's target and accuracy

The adversary performs an SSE attack with a specific target in mind. In the literature [3] two types are described: data recovery attack and query recovery attack. The former is focused on obtaining the document information that is stored on the server, and the latter is focused on solving the matching problem between trapdoor and keyword. They are closely related because by solving this matching problem, the attacker also learns information about the document itself since the keywords are originally extracted from the stored documents themselves.

In the attacks that we investigated, the emphasis is on solving this matching problem. The adversary observes a set of trapdoors and has access to some auxiliary knowledge, and then the attacker constructs an answer that assigns a guess to each trapdoor. The more keywords that are correctly assigned to a trapdoor, the higher the accuracy of an attack is. A high accuracy is not preferable from the client's perspective, since the adversary has learned confidential information.

2.7. Leakage patterns

Previously we have discussed the simplified attack model as seen in Figure 2.3. And that for each trapdoor request by a client, a response is sent back by the server. During this interaction, some information is unintentionally leaked. This information is called leakage and can be abused by an adversary because the trapdoor function that outputs the trapdoor is deterministic. This means that when a client searches for the same keyword, the same trapdoor is sent to the server.

Therefore, the attacker can record the request and response pairs over a period of time and has learned that there exists a pattern between trapdoor and document pairs. This leakage pattern is one of the ingredients to perform the SSE attack, together with the previously discussed adversary's knowledge.

We consider three main types of leakage patterns: access pattern, search pattern, and volumetric pattern. Moreover, there are other leakage patterns that can be considered as part of a main type. These are all patterns that we have observed in the literature and will be discussed for the attacks in chapter 3 Related work.

2.7.1. Access pattern

The access pattern is also known as the response identity [3] or identifier pattern and is a very common pattern in SSE attacks in the literature. In this pattern, the document identifiers in the response are leaked to the adversary for each trapdoor that is sent to the server. This means that for each trapdoor, the adversary knows which encrypted documents contain the encrypted keyword.

The formal definition is given as follows in the literature [3]:

Definition 1. *The identifier pattern is the function family rid={rid$_{k,t}$}$_{k,t\in\mathbb{N}}$ with rid$_{k,t}$* $: \mathbb{D}_k \times \mathbb{W}_k^t \rightarrow [2^{[n]}]^t$ *such that rid$_{k,t}$*$(\mathbf{D}, w_1, ..., w_t) = ($*ids*$(w_1), ...,$*ids*$(w_t))$.

The rid function family maps each keyword to a set of document identifiers. In the formal definition, $2^{[n]}$ is the power set of n, and n are all integers that are used as document identifiers for n documents. More specifically, this set contains document identifiers, where the keyword is found inside the document.

2.7.2. Co-occurrence pattern

This pattern was first described by Islam et. al [19]. The co-occurrence pattern is part of the access pattern because when the access pattern is leaked, the attacker also obtains the co-occurrence pattern.

Assume we have two keywords w_i and w_j, then the number of documents that contain both keywords is the co-occurrence. It can also be seen as a probability, which is the amount of documents that contain both keywords, divided by the total amount of documents.

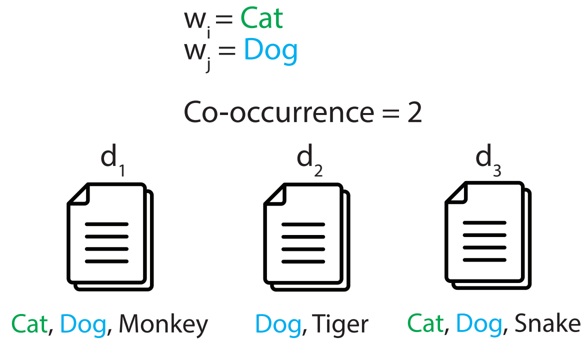


Figure 2.5: Co-occurrence example

In Figure 2.5 an example is shown where the co-occurrence is equal to two. When this number is unique, the attacker compares the co-occurrence of a pair of observed queries, calculates the co-occurrence of queries, and compares this with the co-occurrence of keywords for a match. When many queries are observed, a query to keyword mapping can be deduced from this leakage. As a result, the search query is revealed to the attacker.

2.7.3. Result length pattern

The result length pattern is also known as the response length [3]. This pattern is described in the count attack by Cash et. al [7]. In this leakage pattern, the count or amount of documents is leaked to the adversary. When the Access pattern is leaked, the identifiers of documents automatically leak the result length since the attacker can simply count identifiers.

When certain keywords have unique result lengths, the unique result lengths from keywords can be compared with the results lengths from the observed queries. When a match is found, the adversary has found the correct keyword that corresponds to the trapdoor that was sent by the client.

The formal definition for this pattern is described as follows in the literature [3]:

Definition 2. *The response length pattern is the function family rlen = { rlen$_{k,t}\}_{k,t\in\mathbb{N}}$ with rlen$_{k,t}$: $\mathbb{D}_k \times \mathbb{W}_k^t \rightarrow \mathbb{N}$, such that rlen$_{k,t}(\mathbf{D}, w_1, ..., w_t) = (\#\,\mathbf{D}(w_1), ..., \#\,\mathbf{D}(w_t))$.*

The function family is similar in structure compared to the identifier pattern definition. The difference is that instead of mapping keywords to document identifiers, we now map to an integer in $\mathbb{N}$.

2.7.4. Search pattern

For the search pattern, the adversary is able to distinguish whether two observed queries are the same query or not. Therefore, the pattern is also known as the query equality pattern [3]. By having this distinction and knowing when the same search query is used, the search frequencies of each keyword are leaked to the adversary. The deterministic trapdoor function in SSE schemes reveals when the same search query is used. In addition, Liu et al. [26] mentioned that using a probabilistic algorithm that still reveals the search pattern due to the same documents being accessed.

The formal definition of this pattern is described as follows in the literature [3]:

Definition 3. *The query equality pattern is the function family qeq = {qeq$_{k,t}\}_{k,t\in\mathbb{N}}$ with qeq$_{k,t}$: $\mathbb{D}_k \times \mathbb{W}_k^t \rightarrow \{0,1\}^{t\times t}$ such that qeq$_{k,t}(\mathbf{D}, w_1, ..., w_t) = M$, where M is a binary $t \times t$ matrix such that $M[i,j] = 1$ if $w_i = w_j$ and $M[i,j] = 0$ if $w_i \neq w_j$.*

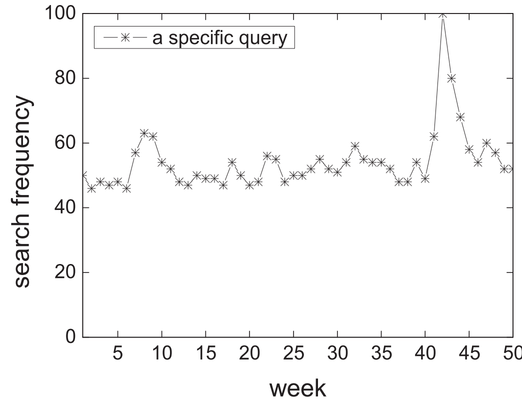


Figure 2.6: Search pattern example [26]

An example is shown in Figure 2.6 in which the search frequency of a specific query is measured over a period of a year. Once the search pattern of a query is known to the adversary, the adversary can compare the search frequency to some auxiliary information to determine which trapdoors correspond to which keyword. As noted by Liu et al. [26], the most vulnerable keywords are the keywords that have high frequencies.

2.7.5. Volume pattern

The volume pattern is considered to be a part of the volumetric pattern [3]. In this pattern, the volume is leaked from each document that is returned to the response of a query. The volume is the size of each document, and the size is the word length of each document.

An example is given in Figure 2.7 where keyword `Cat` is issued, and volumes 14 and 13 are leaked to the attacker. For keyword `Dog`, volumes (14,9,13) are leaked since each document contains the word "Dog".

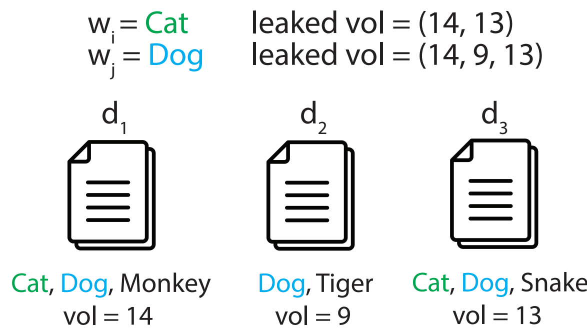


Figure 2.7: Volume pattern example

The adversary can build a known keyword to document mapping from auxiliary information. Moreover, the attacker counts the word length (volume) from the known dataset. With this in mind, the volume pattern is used to compare known volumes to observed volume numbers. In essence, the adversary knows which keyword(s) are possibly used for the query and has obtained a query to keyword mapping.

The formal definition of this pattern is described as follows in the literature [3]:

Definition 4. *The volume pattern is the function family vol = {vol*$_{k,t}\}_{k,t\in\mathbb{N}}$ *with vol*$_{k,t} : \mathbb{D}_k \times \mathbb{W}_k^t \to \mathbb{N}^t$ *such that vol*$_{k,t}(\mathbf{D}, w_1, ..., w_t) = ((|D|_w)_{D\in\mathbf{D}(w_1)}, ..., (|D|_w)_{D\in\mathbf{D}(w_t)})$.

2.7.6. Total Volume pattern

The total volume pattern is part of the volumetric pattern, and as mentioned by Blackstone et. al, it reveals for each query the sum of the volume of documents [3]. This is different from the volume pattern, which reveals the volume of each individual document during the response. Similarly, total volume can be precomputed with auxiliary information and compared with observed total volumes to make a query to keyword mapping.

The formal definition of this pattern is described as follows in the literature [3]:

Definition 5. *The total volume pattern is the function family tvol =* $\{tvol_{k,t}\}_{k,t\in\mathbb{N}}$ *with* $tvol_{k,t} : \mathbb{D}_k \times \mathbb{W}_k^t \to \mathbb{N}^t$ *such that* $tvol_{k,t}(\mathbf{D}, w_1, ..., w_t) = \left(\sum_{D\in\mathbf{D}(w_1)} |D|_w, ..., \sum_{D\in\mathbf{D}(w_t)} |D|_w \right)$.

This means in this case there is a small change in the example given in Figure 2.7. When the keyword `Cat` is issued, instead of learning the leaked volumes (14,13), the attacker only learns the sum of volumes that contain the keyword, which is 27.

2.8. Leakage Profiles

The leakage profile is a collection of leakage patterns [3]. As described by Cash et al. [7], the leakage profile is the view of the adversary, where each scheme can have a different leakage profile. We simulate the view of an attacker based on the leakage profile, which consists of one or more leakage patterns.

In the paper, four different leakage profiles are described from L4 to L1. Leakage profile L4 has the highest amount of leakage, and L1 has the least. We will discuss these 4 leakage profiles and create some examples for each leakage profile, based on figures from their paper [7].

In leakage profile L4, substitution takes place for every keyword in the document. Each keyword is substituted by the output of a deterministic encryption algorithm. In this case, the adversary learns the keyword existence, the keyword order, and the keyword count which is the total amount of keyword occurrences in a document.

In Figure 2.8 an example is shown of two documents, where each document consists of some keywords. The top frame indicates the view of the adversary before a query is sent, and the bottom frame highlights the view of the attacker after a query is sent. In this example, the client searches for the keyword Kitty and sends the query Gds82m to the server. The keyword order, the keyword count, and the access pattern are revealed to the attacker.

The next leakage profile is L3 which is a slightly stronger profile compared to L4. In this leakage profile, the adversary still learns the keyword order inside each document. However, the attacker does not learn the total amount of occurrences for each keyword. Instead, only the first appearance or location of a keyword is learned. This means that the adversary knows the occurrence pattern for a keyword, but not the occurrence count which will only be one or zero. In Figure 2.9, an example is shown for Leakage profile L3.

In leakage profile L2, the occurrence pattern is still known before a query is issued. The change from L2 is that the order of the first appearance of keywords is hidden since the keywords are unordered. A scheme that uses the inverted index that maps keywords to documents is considered to be from leakage profile L2, where each keyword inside the index is stored in an unordered way. In Figure 2.10 an example is shown, where the rows are shuffled in comparison to the previous leakage profile L3.

The final leakage profile that Cash et al. [7] described is leakage profile L1. In Figure 2.11, a difference can be observed in comparison to the previous leakage profile L2. The big change from L2 is that the occurrence pattern is only revealed after the query and response interaction starts between the client and server. Therefore, the document identifiers D7 and D9 in Figure 2.11 are only revealed in the bottom frame of the figure and not the top which is the view at the setup stage. Even in leakage profile L1, the access pattern is revealed. This is revealed for each query and response that is captured by the adversary.

Document 1			
7Hayaizm	Gds82m	Kio1Gzfg	Gds82m
9YUzmQW	Gds82m	0ly5Bnzi	B4NuYTL
YU6bGzm	YA232Aw	P0YBnZe	889AmzT
Document 2			
KKL71RtZ	S7dsmYu	Gds82m	L6TnzHG
99HABzE	1RjaMXu	01aTRk	GGzYtoA

Keyword: Kitty-> Gds82m

Document 1			
7Hayaizm	Gds82m	Kio1Gzfg	Gds82m
9YUzmQW	Gds82m	0ly5Bnzi	B4NuYTL
YU6bGzm	YA232Aw	P0YBnZe	889AmzT
Document 2			
KKL71RtZ	S7dsmYu	Gds82m	L6TnzHG
99HABzE	1RjaMXu	01aTRk	GGzYtoA

Figure 2.8: Leakage profile L4

Document 1			
7Hayaizm	BNO11zV	Kio1Gzfg	Gds82m
9YUzmQW	5YaTEzw	0ly5Bnzi	B4NuYTL
YU6bGzm	YA232Aw	P0YBnZe	889AmzT
Document 2			
KKL71RtZ	S7dsmYu	Gds82m	L6TnzHG
99HABzE	1RjaMXu	01aTRk	GGzYtoA

Keyword: Kitty-> Gds82m

Document 1			
7Hayaizm	BNO11zV	Kio1Gzfg	Gds82m
9YUzmQW	5YaTEzw	0ly5Bnzi	B4NuYTL
YU6bGzm	YA232Aw	P0YBnZe	889AmzT
Document 2			
KKL71RtZ	S7dsmYu	Gds82m	L6TnzHG
99HABzE	1RjaMXu	01aTRk	GGzYtoA

Figure 2.9: Leakage profile L3

7Hayaizm	D4, D8, D9
Gds82m	D7, D9
BNO11zV	D2, D4, D6
KKL71RtZ	D1, D3

Keyword: Kitty -> Gds82m

7Hayaizm	D4, D8, D9
Gds82m	D7, D9
BNO11zV	D2, D4, D6
KKL71RtZ	D1, D3

Figure 2.10: Leakage profile L2

8Yz	7BY	X3T	L8R	D4Q	3TY
BNO	N6C	8T2	N8X	P0Y	Q2Z
9XP	BBC	C4T	LT9	AWQ	0G3

Keyword: Kitty -> Gds82m

8Yz	D7	X3T	L8R	D4Q	3TY
BNO	N6C	8T2	D9	P0Y	Q2Z
9XP	BBC	C4T	LT9	AWQ	0G3

Figure 2.11: Leakage profile L1

3 Related work

The previous chapter discussed the background knowledge of SSE that is applied in this chapter. In this chapter, we describe the related work regarding attacks on SSE. Furthermore, we explain each attack that we investigated. Each attack has certain characteristics regarding the adversary's knowledge, leakage patterns, and trade-offs. Learning about related work is necessary to answer the main research question to design an improved attack. Finally, we provide an overview of the attacks and illustrate how our new attacks differ in knowledge and exploited patterns from other attacks.

3.1. IKK attack

The IKK attack is the first attack that we investigated by Islam et al. [19]. This is the first work that demonstrated an SSE attack using access pattern leakage and from this work, many others were inspired to research other possibilities and improvements.

It was first described as an inference attack (similar data attack), however due to the amount of background knowledge that is required to be an effective attack, this attack was later categorized as a known data attack with a minimum leakage profile of L1 [7]. The target of the attack is query recovery, which means that the attack tries to assign the correct keyword for each query that is sent.

The IKK attack uses full document knowledge as auxiliary information, and it exploits the co-occurrence pattern. The attack first computes the co-occurrence of keywords using either the full document knowledge or an approximation matrix that is very close to the real dataset, as a background matrix.

Next, the co-occurrence of queries is calculated based on the observed queries that are intercepted by the adversary. This abuses the access pattern leakage, where each search result reveals the document identifiers.

Finally, an optimization problem is solved that finds the best match by creating a mapping between the co-occurrence of queries, and the background matrix, which is the co-occurrence of keywords. The smallest sum of the distance from the co-occurrence of queries to the background matrix is the most optimal solution.

The advantage of this attack is that it is mostly independent of the known query set size, based on their results [19]. However, their results also show the disadvantage in which the keyword set size and the query set size have a large influence on the accuracy of the attack. When the keyword set size is increased, the accuracy becomes lower. But, the query set size has the opposite effect. The larger the query set size, the higher the accuracy. The reason is that a larger proportion of the background matrix is used for the prediction [19].

However, in practice solving the optimization problem is more complicated since it is an NP-complete problem. So, the authors use an approximation heuristic algorithm called simulated annealing. Essentially a random query to keyword assignment configuration is set, and a cost is calculated. And then a

slight change is made to the configuration to hopefully gain an improvement in cost. This is repeated many times until certain conditions are met, and the final assignment is returned.

Cash et al. [7] observed that the success rate is low if not all documents are known beforehand. We implemented this attack and we also noticed that this attack requires full document knowledge to have an effective query recovery. Overall this attack is not ideal and not realistic but it is a good start to investigate, and was quickly improved by the next attack, the Count attack.

3.2. Count attack

The Count attack [7] is an improvement over the IKK attack. It uses the same assumptions as the IKK with the same goal of search query recovery, and also the same leakage profile of L1.

Similarly, it utilizes the access pattern and requires full document knowledge. The difference from IKK is that in addition to the co-occurrence pattern, the count attack also utilizes the result length pattern.

The algorithm first tries to match keywords and queries based on the unique result length. If there is a unique count, then we have found the keyword that matches the unknown query. But, if there are multiple keywords that share the same count, the algorithm falls back to the usage of co-occurrence to filter out candidate keywords in the candidate keyword set.

When only one keyword remains in the candidate keyword set, then the query to keyword match is found. Otherwise, only the possible set of candidate keywords is found to match a query. This number of candidate keywords can still be reduced to one when new information is found during the algorithm, which is another query to keyword assignment. This is because new information leads to a smaller subset of possible candidate keywords. Nevertheless, a set of possible candidate keywords that match a query still gives more information compared to IKK, even when reduction to 1 candidate is not possible.

We have implemented the Count attack that uses the full document knowledge with zero known queries, and we observed that the count attack runs faster than IKK. The accuracies when using different numbers of keywords were also the same as shown in their paper [7].

Overall this attack is an improvement over IKK, it delivers a higher performance in numbers of accuracy and is faster than IKK which requires simulated annealing. It also does not require known query knowledge. And if the exact keyword is not found, the adversary still learns a possible set of candidate keywords.

The drawback is that this attack requires a high known data rate, which is almost full document knowledge to be effective for query recovery. The authors Cash et al. [7] also experimented with partial document knowledge, in which the algorithm was modified to use a window instead of exact equality. However, false positives are now also possible, and it still requires a high percentage of known documents.

3.3. Search

This is a completely different type of attack compared to the previous attacks. This attack by Liu et al. [26] utilizes the search pattern and requires knowledge about keywords that are issued by a client. From those keywords, the search pattern of queries needs to be known.

Firstly, the adversary records all queries that are issued by a client over a period of time, measuring the frequency. Secondly, the frequencies are compared to known auxiliary search patterns that are known beforehand for keywords. Thirdly, a match between query and keyword is assigned based on minimizing the distance of the frequencies between observed and known search patterns.

This attack when compared to the previous leakage abuse attacks does not require full document knowledge. However, it also has some drawbacks. The first drawback is that the attacker needs to obtain the search pattern auxiliary information. This might be a difficult task. In the paper [26], the authors utilized Google Trends as auxiliary information. When the actual search pattern is in line with

the known pattern, it does provide accurate results. However, it is also possible that there could be a large deviation between the observed and known search patterns.

Liu et al. [26] applied Gaussian noise on the auxiliary information to simulate search patterns issued by a client, since the Enron data set does not provide search pattern statistics. This still provides accurate results and looks promising, but it is unclear how close the auxiliary information has to be to closely match the real observations in a real scenario.

The second drawback is that this attack requires the client to issue a large number of queries to the server to obtain the search pattern of the client's keywords. If the client only issues queries very rarely or a small amount of queries, the search pattern of keywords might not be distinguishable. In this case, the matching with auxiliary information would provide incorrect results, resulting in poor accuracy.

3.4. Graph Matching

The graph matching attack [35] is a similar data attack and utilizes the co-occurrence leakage pattern. This attack is also different than the others since it aims to attack systems that utilize the ”efficiently deployable efficiently searchable encryption” (EDESE) schemes with high leakage profiles, which are used to support legacy applications. EDESE schemes have a higher attack surface than SSE schemes since they leak more information for performance.

The attack reduces the problem to a weighted graph problem. The adversary has access to some similar data as auxiliary data. Nodes for the first graph are created based on the keywords from similar data, and the edges contain a weight which is the co-occurrence probability that both keywords occur in a document. In the same way, a second graph is created for the target data, based on the observations of queries or so-called tags.

The problem is now reduced to the weighted graph matching problem in which it tries to find a solution to permute the target graph to make it as close as possible to the graph created from the auxiliary information. And then an appropriate solver can be used to solve it like PATH or Umeyama's algorithm to reveal the keyword and tag pairs.

Pouliot and Wright [35] mention that the advantage is that this attack does not require any query auxiliary knowledge, no known keywords, or knowledge about almost all the documents like the IKK attack. The drawback is that this attack does not perform too well when not having complete knowledge. Although there is a high recovery rate seen from their results for a limited group of users. Also, this attack is specifically designed with the EDESE scheme in mind, while other attacks that we have investigated so far are for the stronger SSE scheme. This inference attack can be seen as an improved IKK attack, but with the strong assumption that the target data and auxiliary data are strongly correlated [18].

3.5. Volan

The volume Analysis (Volan) attack [3] is a known data attack that utilizes the volumetric leakage pattern. More specifically, it exploits the total volume pattern (tvol) which is the sum of volumes. In this pattern, the total volume of documents that contain the issued query is leaked, and volume is the word length of a document. As auxiliary data, this attack only requires partial document knowledge, instead of full document knowledge like the IKK attack.

The technique is to match the closest known volume from the auxiliary data. This is done by observing the leaked total volumes of each query sent by a client, followed by a comparison with the calculated volumes from the auxiliary data. In other words, for each keyword in the keyword space, compare and choose the keyword where the total volume of the known keyword is smaller or equal to the observed total volume from a query, and choose the keyword that is the closest.

The advantage of this attack is that it does not rely on the access pattern and only on the volumetric pattern. According to Blackstone et al. [3], an attack that exploits the document size works on SSE-based schemes and even ORAM-based schemes. This attack also does not require any known queries.

The drawback of this attack is that the assignment is done with a single keyword instead of a candidate set. The reassignment of the same keyword among different queries is also possible in the algorithm.

Furthermore, to achieve a decent query recovery accuracy, almost full document knowledge is required which can be highly impractical.

3.6. SelVolan

The Selective Volume Analysis (SelVolan) attack [3] is an improvement from Volan. It still uses partial document knowledge as auxiliary data. The leakage patterns consist of the same total volume pattern as the previous attack, with an additional new pattern, the result length pattern.

There are two steps to this attack. The first step is window matching in which a query is matched to a set of keywords that are within a window. Keywords with a known volume that are within the window of the observed volume are added to this candidate set. The second step is selectivity filtering. It utilizes the response length to filter out keywords from the candidate set. The first keyword of the candidate set is chosen and assigned to a query.

The advantage of this attack is that similar to Volan, it utilizes the volumetric pattern, which does not rely on co-occurrence. It also does not require known queries as auxiliary knowledge. The accuracy is higher than Volan, based on the results from their paper [3]. And it utilizes a candidate set, which Volan did not use. This means that a candidate set provides more useful information than a completely wrong or uncertain assignment.

The drawback is that although the auxiliary data is called partial knowledge, their results [3] indicate that over 90% of data knowledge is required to obtain high query recovery rates. Both attacks work well in a high selectivity keyword setting. A high selectivity indicates that keywords are chosen that appear the most frequently in documents. When both attacks are done in a low selectivity setting, the attacks only recover less than 20% in the paper, even when 100% of the documents are known. Depending on the goal of the attacker, this might be an acceptable recovery rate.

3.7. Subgraph

The Subgraph attack [3] is an attack that exploits an atomic pattern. Blackstone et al. [3] describe this as any pattern that reveals a function of each matching document. In their paper two types of Subgraph attacks are shown, one that utilizes the volume pattern, and the other utilizes the identifier pattern, both are atomic patterns. In contrast to the total volume pattern, the volume pattern reveals the volume of each individual document that is queried, and not only the sum of volumes.

This attack maps the adversary's knowledge into a graph representation. Matching is based on partial data property observations and applied to the graph. In addition, an iterative elimination technique is applied so that keywords that are chosen for a query cannot be chosen again as a candidate in another candidate set.

The advantage of this attack is that no known queries are required as auxiliary data. Based on their results [3], a low known data rate also performs well in accuracy and performs better than Volan and SelVolan. However, it also shares the same drawbacks. The attack performs much worse for queries chosen in a low selectivity setting compared to a high selectivity setting, even with full document knowledge. The other disadvantage in our view is that it is more difficult to understand and improve in a graph setting.

3.8. LEAP

The LEAP [30] attack utilizes the co-occurrence pattern and introduces a new way of utilizing partial document knowledge. Unlike other attacks that have false positives due to the lack of information in the partial document setting like the Count attack, the LEAP attack can recover keywords accurately without false positives.

In the LEAP attack, there are two important matrices that are used to derive new information, the B and A" matrices. The B matrix is a matrix that consists of queries as rows, and encrypted documents as columns. Each cell is either a 1 or a 0, which depends on whether the query is in the encrypted document or not. This can be derived without sending queries by the attacker since the LEAP attack is based on the leakage profile of L2, and the attacker has access to the encrypted server files.

The other matrix is the A” matrix, which is an extension of the A’ matrix. The A’ matrix, is a matrix that consists of keywords as rows, and documents as columns. This can be derived from the partial document set that the adversary has as auxiliary information. Each cell is either a 1 or a 0, depending on whether the document contains the keyword specific in the row or not.

The problem is that matrix B and matrix A’ cannot be directly compared. The row sum is not applicable because matrix A’ has fewer documents, or partial documents compared to matrix B. However, column sum is applicable if the rows of matrix A’ are extended to the same amount of rows as matrix B. This new matrix is called matrix A”. From the query tokens that are stored on the server, it is possible to derive unique tokens that are not inside the partial document set from the attacker, and therefore extend the keyword rows in matrix A’ to A”, which means the rows of matrix B and A” are now equal in size. The newly added rows contain zeroes in the cells since these are new keywords that are clearly not inside the partial document set by the attacker.

LEAP then uses a variety of methods to derive row and column mappings between matrix B and A”. The authors described 5 methods that in the end will derive encrypted document-to-document mappings and query-to-keyword mappings. The details can be found in their paper [30].

The advantage of this attack is how accurate the results are without false positives and the low amount of partial knowledge that is required. In their results [30], LEAP performs much better compared to the Count attack that we described earlier. When given 1% partial document knowledge as auxiliary information, the count attack has a correct keyword recovery rate of 0.06%, whereas LEAP has a 52.86% recovery rate. When the partial knowledge is slightly increased to 5%, the Count attack still performs poorly with a recovery rate of 0.1%, whereas LEAP achieves 92.86%.

This attack is based on the leakage profile of L2, whereas the count attack is on L1. The disadvantage of this attack is the same as the Count attack, namely that the attacker requires a subset of documents that are stored on the server as auxiliary information.

3.9. Score

The Score attack [12] is a similar data attack that exploits the co-occurrence pattern. This inference attack requires known queries as auxiliary information and documents that are distribution-wise similar to the documents that are stored on the server, unlike partial document knowledge in which the attacker actually has a subset of the documents.

In the Score attack, the known queries are used to create two co-occurrence submatrices, one for the keyword-to-keyword submatrix and the other as trapdoor to trapdoor sub matrix. This is extracted from the original matrices for keyword co-occurrence and trapdoor co-occurrence. It is a submatrix because it is based on column selection from known queries, and the columns between the two submatrices share the same order in order to compare them directly.

The technique is to assign a score for each keyword, based on the distance between the two submatrices and append the result to a list. The candidates are then sorted in descending order, and the candidate keyword with the highest score is chosen as the prediction for a trapdoor. This is repeated for every trapdoor so that each trapdoor obtains a keyword prediction.

The advantage of this attack is the usage of similar documents and candidates set for keywords. The disadvantage is that this attack requires known queries for it to work, the more the better. It also has some problems handling keyword candidates that share close scores, in this case, the wrong keyword could be chosen.

3.10. RefScore

The Refined Score attack (RefScore) [12] is an improved version of the Score attack. In addition to the scoring mechanism, certainty is utilized to improve the accuracy of the attack. The certainty is the difference between the candidate keyword with the highest score and the candidate keyword with the second highest score. In the case when these two scores are close to each other, the certainty will be low.

The list of keyword and certainties are sorted in descending order, and the most certain predictions are

chosen. A difference with the Score attack is that in this attack, newly added (query, keyword) pairs are utilized to improve the amount of known queries. In other words, once more knowledge is acquired, it is used to create new submatrices with an increased amount of columns to improve the predictions for the next iteration.

It is clear that this is an improvement over the basic Score attack, and also improves the handling of close score candidate keywords. This attack also extends the knowledge during execution to improve prediction accuracy. The attack has better query recovery accuracy than the Score attack and, therefore, requires less known queries compared to the Score attack.

One major drawback of this attack is how sensitive it is to the amount of known queries and also the full reliance on known queries. This attack cannot be run without known queries.

3.11. Overview of the attacks

From the attacks that we have discussed, we have created an overview to compare them with each other in Table 3.1. The access pattern is not listed, but the co-occurrence pattern is part of the access pattern. The purple rows indicate the attacks which are closely related to our new attacks which are highlighted in green. We have designed three new attacks that are based on the Score and RefScore attack.

Table 3.1: Comparison of SSE attacks

Attack name	From	Type	Document knowledge	Query knowledge	Exploited patterns
IKK	IKK12	Known data	Full	None	Co-occ
Count	CGPR15	Known data	Partial	None	Co-occ, rlen
Search	LZWT14	Inference	Keywords	Query search patterns	Search
Graph matching	PW16	Inference	Similar	None	Co-occ
Volan	BKM20	Known data	Partial	None	Tvol
SelVolan	BKM20	Known data	Partial	None	Tvol, rlen
Subgraph	BKM20	Known data	Partial	None	Rid or Vol
LEAP	NHPY21	Known data	Partial	None	Co-occ
Score	DHP21	Inference	Similar	Partial	Co-occ
RefScore	DHP21	Inference	Similar	Partial	Co-occ
VolScore	This paper	Inference	Similar	Partial	Co-occ, vol
RefVolScore	This paper	Inference	Similar	Partial	Co-occ, vol
ClusterVolScore	This paper	Inference	Similar	Partial	Co-occ, vol

To design an improvement is a challenge. The attacks do share some similarities, but also many differences. Many attacks that we have investigated are known data attacks, and each may have different auxiliary knowledge requirements and exploited patterns. Full or partial document knowledge is a common requirement for known data attacks, and many attacks utilize the co-occurrence pattern.

The idea we had is to utilize a technique from an existing attack and to explore combining an additional exploited pattern to improve the accuracy of an attack. Our intuition was that an additional exploited pattern would increase the query recovery accuracy since the attacker has more information. We thought about combining the volume pattern with the co-occurrence pattern. Lambregts et al. [25] combined the volume pattern into the LEAP attack successfully to improve the accuracy, and this gave us the confidence to also explore this combination in a different setting.

This overview consists of only attacks that we investigated in the limited time we had, and is by no means a complete list. There is an SSE attack that combines search pattern, volume pattern, and co-occurrence pattern in the inference type setting [32]. There are even more inference attacks, and Gui et al. [18] has provided an overview of more inference attacks and their differences.

We had to set a scope with the constraints in time we had, and to improve an attack from one of the attacks that we have investigated so far. So we challenged ourselves to apply the volume pattern in the Refined Score attack, which to our knowledge has not been done before.

4 Revisiting Score and RefScore

We briefly discussed both Score and Refined Score (RefScore) attacks before in the related work. In this chapter we revisit both attacks [12], and analyze the algorithms in more depth in order to discuss possible ways of improvement.

4.1. The Score Attack

Algorithm 1 Score Attack [12]

Require: $K_{sim}, C^s_{kw}, Q, C^s_{td}$
1: $pred \leftarrow []$
2: **for all** $td \in Q$ **do**
3: candidates=[]
4: **for all** $kw \in K_{sim}$ **do**
5: $s = -\ln(||C^s_{kw}[kw] - C^s_{td}[td]||)$
6: append (kw, s) to candidates
7: **end for**
8: candidates =$sort$(candidates, $desc$)
9: append $(td, candidates[0])$ to $pred$
10: **end for**
11: **return** $pred$

There are three important parts to this algorithm. Firstly to notice is the knowledge requirements. This algorithm requires known queries and similar documents as auxiliary information in order to obtain K_{sim}, C^s_{kw}, to run the algorithm, and Q contains observed queries. C^s_{kw}, C^s_{td} are co-occurrence submatrices for keyword and trapdoor respectively that are created using known queries and by observing trapdoors.

Secondly, is the scoring mechanism that is key to deciding which keyword is the best candidate. The two co-occurrence vectors $C^s_{kw}[kw], C^s_{td}[td]$ are for keyword and trapdoor respectively, where each keyword is a keyword extracted from K_{sim}, and each trapdoor is from the observed queries. These co-occurrence vectors are special because it is essentially a mapping to the space of known queries since the submatrices only consist of columns from known queries. This means that a keyword co-occurrence vector can directly be compared to a trapdoor co-occurrence vector. The closer they are, the more similar they are, which means that a possible match between the keyword and trapdoor is found.

Damie et al. [12] defined a scoring that calculates the distance between the keyword and trapdoor co-occurrence vector, and then the Euclidean norm is applied to obtain the distance as a single number. The authors also applied a negative logarithm on this number to obtain a score number that focuses on the order of magnitude. This makes it much easier to interpret distances that are very small numbers

close to zero. For example, a small distance of e^{-13} results in a score of 13, where the keyword and trapdoor are very similar. But, another small distance of e^{-14} results in a score of 14, where the order of magnitude is larger, and the preferred solution.

Finally, the list of candidate keywords is sorted in descending order on the score for the current trapdoor. The candidate keyword with the highest score is chosen for the current trapdoor, and then the algorithm is repeated for every observed trapdoor.

In this algorithm, a couple of problems can be observed. Close scores are a problem. For example, if one candidate has a score of 30, and another candidate has a score of 29, then the algorithm chooses the candidate with the highest score. But, the second highest candidate could actually be the correct answer.

Another problem is the utilization of knowledge. At the very beginning, the co-occurrence submatrices are created based on the known queries. However, when the algorithm assigns keywords to trapdoors, some new known queries are acquired. However, this knowledge is not utilized at all, and new iterations of the algorithm are still using the old knowledge.

Both problems are solved in the new improved version, the RefScore Attack.

4.2. The RefScore Attack

Algorithm 2 Refined Score Attack [12]

Require: $K_{sim}, C^s_{kw}, Q, C^s_{td}$, KnownQ, RefSpeed
1: $final_pred \leftarrow []$
2: unknownQ $\leftarrow Q$
3: **while** unknown $Q \neq \emptyset$ **do**
4: % 1. Extract the remaining unknown queries
5: unknownQ $\leftarrow \{td : (td \in Q) \wedge (\nexists kw \in K_{sim} : (td, kw) \in KnownQ)\}$
6: $temp_pred \leftarrow []$
7:
8: % 2. Propose a prediction for each unknown query
9: **for all** $td \in$ unknownQ **do**
10: $cand \leftarrow []$ { The candidates for trapdoor td}
11: **for all** $kw \in K_{sim}$ **do**
12: $s = -\ln(||C^s_{kw}[kw] - C^s_{td}[td]||)$
13: append {"kw": kw, "score":s} to $cand$
14: **end for**
15: Sort $cand$ in descending order according to the score.
16: certainty $\leftarrow$ score($cand$[0] - score($cand$[1])
17: append (td, kw($cand$[0]), certainty) to $temp_pred$
18: **end for**
19:
20: % 3. Either stop the algorithm or keep refining.
21: **if** $|$ unknownQ $|$ < RefSpeed **then**
22: $final_pred \leftarrow$ KnownQ $\cup$ $temp_pred$
23: unknownQ $\leftarrow \emptyset$
24: **else**
25: Append the RefSpeed most certain predictions from $temp_pred$ to KnownQ
26: Add the columns corresponding to the new known queries to C^s_{kw} and C^s_{td}
27: **end if**
28: **end while**
29: **return** $final_pred$

In the RefScore attack, the requirements are the same, but there is now an additional RefSpeed input parameter. This parameter is the refinement speed and has an influence on the accuracy of the results. Damie et al. [12] added this additional parameter to speed up the runtime of the algorithm. The higher

this value is, the faster the algorithm refines, therefore predictions are assigned faster. The lower the value, the slower the algorithm runs. This value cannot be set too high to avoid too many wrong keyword assignments to trapdoors. The authors presented their results with a RefSpeed of 10, which seems to be a good middle ground between performance and accuracy.

In the first part of the algorithm, unknown queries are determined for the current refinement iteration. In the second part, there are some changes observed compared to the Score attack. The scoring mechanism remains the same using co-occurrence vectors. However, certainty is now used as a decider for predictions. The certainty of a candidate keyword is defined as the score difference between the candidate with the highest score, and the second highest score. This means that when the top two candidates have very close scores, the certainty will be low. Each trapdoor with a candidate keyword will have a certainty assigned to it temporarily.

In the final part of the algorithm, the decisions are made for the assignments of candidate keywords to trapdoors. The number of unknown queries together with the RefSpeed parameter determines if the algorithm should stop, or continue with the refinement. If the algorithm continues, only the most certain predictions are added as known queries, with a number equal to the RefSpeed. Afterward, the co-occurrence submatrices are recreated by using the new known queries as a base, before the next iteration starts again.

A major advantage in the RefScore attack is that newly acquired knowledge is utilized to improve the accuracy. And the usage of certainty also improves the accuracy of this attack. In their results [12], the RefScore attack performs much better than the base Score attack with the same amount of known queries.

4.3. Improvement intuition and discussion

Since the RefScore Attack is the best version of the Score attack, it is the best attack to improve. It was also a big challenge because the RefScore attack was already an attack that performed well, and any adjustment to this algorithm could result in worsening the performance. We have considered many options to improve this attack. Many ideas did not work out as initially expected. In the end, we have chosen to explore clustering and utilizing the volume pattern for the new attack.

4.3.1. Frequency and result length

We believe that the scoring mechanism already worked well, so instead of changing how scoring and certainty are computed, the frequency of a keyword or trapdoor that occurs over all documents might improve the accuracy. We also studied the source code [13] and observed that word occurrences and frequencies were already calculated. The idea was to utilize frequency as a filter to filter out candidate keywords. Similarly, result length could also be used as a filter since this is also known to the attacker.

However, by creating some very small experiments with these filters, it quickly became apparent that this approach is too naive and we could not improve the accuracy this way. The problem is that the filters are not good enough to reduce the amount of candidates. The candidate keywords that need to be filtered out could have large deviations of more than 50% from the auxiliary information for a correct keyword. This makes it impossible to distinguish the correct from the incorrect candidate since a keyword with a closer distance could be wrong. In other words, the addition of these filters had no positive effect on the accuracy of the RefScore attack.

4.3.2. Refspeed parameter

The next idea that we had was to adjust this new RefSpeed parameter during the early, and final stages of the algorithm iterations. The main intuition was to make less drastic decisions during the start and the end of the algorithm, in other words, to slow down the refinement speed, when there is a high uncertainty, and to increase the RefSpeed when more known queries become available except the end. The reason is that in the end, only low certainty candidate keywords remain.

This idea was also not an improvement. The main issue is that at the start of the algorithm, when we are too careful it accidentally chose only the wrong predictions, whereas if the algorithm was more greedy and had a higher Refspeed, it could also have chosen the correct ones that in the end resulted

in much higher accuracy. When the amount of predictions is low at the start and wrong, it leads to very bad performance.

When we applied the RefSpeed slowdown at the final part of the algorithm, it also did not increase the accuracy. We overlooked that at the final iterations of the algorithm, only low certainty keywords remain. So even if we are more careful and perform a slowdown, this will not make a difference compared to simply adding the normal RefSpeed amount of predictions.

4.3.3. Clustering

Clustering is an interesting idea that Damie et al. [12] also mentioned in their paper. Instead of assigning one keyword to one trapdoor as a prediction, the authors return a cluster of keywords as predictions for each trapdoor. In their paper, this made an improvement of about 5% for RefScore with clustering. However, this comparison does not feel right when the definition of accuracy is different among the two as highlighted by the authors. The original defines accuracy when a single keyword is correctly assigned to the corresponding trapdoor, whilst for the clustering the keyword only needs to be within the returned cluster.

Our intuition is to also utilize clustering but in a different way. Instead of having a static RefSpeed parameter, the RefSpeed can dynamically increase or decrease depending on the certainty. A large cluster with high certainty should be accompanied by a higher RefSpeed, whilst reducing the RefSpeed when the certainty is low, therefore limiting the amount of of wrong predictions. We decided to explore this more, and use this idea when designing the new attack to improve RefScore.

4.3.4. Volume pattern

As we saw before in Table 3.1, the co-occurrence leakage pattern is often exploited by many attacks. Some attacks utilize the volume pattern like Volan, SelVolan, and Subgraph. Our intuition is that combining the volume pattern with the co-occurrence pattern in RefScore results in more knowledge for the attacker. And by having more knowledge, an existing attack's accuracy can be improved.

We have tried a naive approach from Volan to combine into RefScore as a filter, where we utilize volume to decrease the number of candidate keywords. We were unsuccessful and encountered the same problems as described before when using frequency as a filter. Nevertheless, our initial intuition still remains that we should be able to increase the attack accuracy with this additional leakage pattern, but we have to combine the new pattern in a different way. So, we decided to explore this pattern more and utilize it in the new attack.

5 VolScore Attacks

In this chapter, we describe the new attacks that we designed to improve the RefScore attack. First, we describe the notations that are used in the algorithms and experiments. Next, the assumptions are described and are prerequisites for the attack. Furthermore, in the design idea section, we illustrate the intuition behind the VolScore attacks with a visualization of the concept. In addition, each VolScore attack is explained in detail together with the algorithms. Finally, we end this chapter by discussing the countermeasures that we considered to guard against the attacks.

5.1. Notation

Notation	Definition
D	Document collection D = $(d_1, d_2, ..., d_n)$
ED	Encrypted documents ED = $(ed_1, ed_2, ..., ed_n)$
D_{sim}	Similar document set, available to attacker
$\|D_{sim}\|$	Amount of similar documents
K_{sim}	Keyword vocabulary, extracted from D_{sim}
m_{sim}	Amount of keywords in similar vocabulary
D_{real}	Real document set, stored on server
$\|D_{real}\|$	Amount of real documents
K_{real}	Real keyword set which are usable by the client
m_{real}	Amount of keywords in real vocabulary
Q	Queries that are sent by the client and observed by attacker
$\|Q\|$	Amount of queries
KnownQ	Known queries (td, kw) pairs
|KnownQ|	Amount of known queries
UnknownQ	The unknown queries for the attacker, i.e. Q - KnownQ
C^s_{kw}	Co-occurrence submatrix for keywords
C^s_{td}	Co-occurrence submatrix for trapdoors
V^s_{kw}	Covolume submatrix for keywords
V^s_{td}	Covolume submatrix for trapdoors
K	Abbreviation for 1000, for example 1.2K = 1200
td	Abbreviation for trapdoor
kw	Abbreviation for keyword
kq_set	Set of keywords or queries
inv_index	Mapping from keywords to documents, or query to document identifiers
vol_array	An array that consists of the volume of each document
nr_docs	Amount of documents stored locally, or estimated amount of documents stored on the server
RefSpeed	The refinement speed
MaxRefSpeed	The maximum refinement speed

5.2. Assumptions

In this attack, we have made some assumptions that are common among SSE attacks. We consider the leakage profile to be L1 in which the co-occurrence pattern as well as the volume are leaked after the queries are sent.

Firstly, we assume the adversary is honest but curious and could be the untrusted server itself. The attacker follows the protocol but also tries to learn as much information as possible. The attacker has the capability to observe the traffic between the client and the server.

Secondly, we assume that two leakage patterns are applicable and that both co-occurrence (access pattern) and volume pattern are leaked to the adversary for each query that is issued to the server. This means that each issued trapdoor leaks the individual document identifiers that are returned, as well as the individual volume for each document in the response.

Thirdly, we assume that the attacker has access to a similar dataset that shares a similar keyword distribution to the real dataset that is stored on the server. In addition, we assume the attacker is using the same keyword extraction algorithm as the client, as well as that the most frequent keywords are chosen for the vocabulary.

Finally, the attacker has acquired a set of known queries to perform the attack. These consist of keywords that are within the vocabulary of both the attacker and the client.

5.3. The design idea

The Refined Score Attack (RefScore) [12] is already a powerful inference attack that obtains an accuracy that is viable for an attacker with as low as five known queries. The main intuition behind our design is that an attacker has more attack power with more knowledge, therefore it improves the accuracy of the original attack.

We increase the knowledge of the adversary by utilizing the `volume pattern` and designed the VolScore attack while keeping many elements and the core matching technique the same as RefScore from Damie et al. [12]. The goal of this initial attack is not to achieve better results than RefScore, instead, it is a way to obtain additional knowledge and initial query to keyword predictions. We combine the results obtained via the `volume pattern` and `co-occurrence pattern` by utilizing VolScore, with results via the `co-occurrence pattern` that are obtained from RefScore.

RefScore itself creates query-to-keyword predictions by using a number of known queries, but the adversary does not know which predictions are assigned correctly, and which are assigned incorrectly. By using the additional predictions that are obtained from VolScore, and intersecting with the predictions from RefScore, the attacker becomes more certain about some of the predictions, or in other words the attacker has acquired more knowledge. The attacker can now start a fresh new attack but with an increased amount of known queries, and with more known queries, the accuracy of the attack is improved.

In Figure 5.1 we visualized this concept. The chain of attacks that consists of running VolScore, RefScore and another RefScore with increased knowledge is called RefVolScore. We can also replace the last RefScore attack with a modified version that utilizes clustering, and that chain is called ClusterVolScore. We explain each of these attacks in more detail in the following sections.

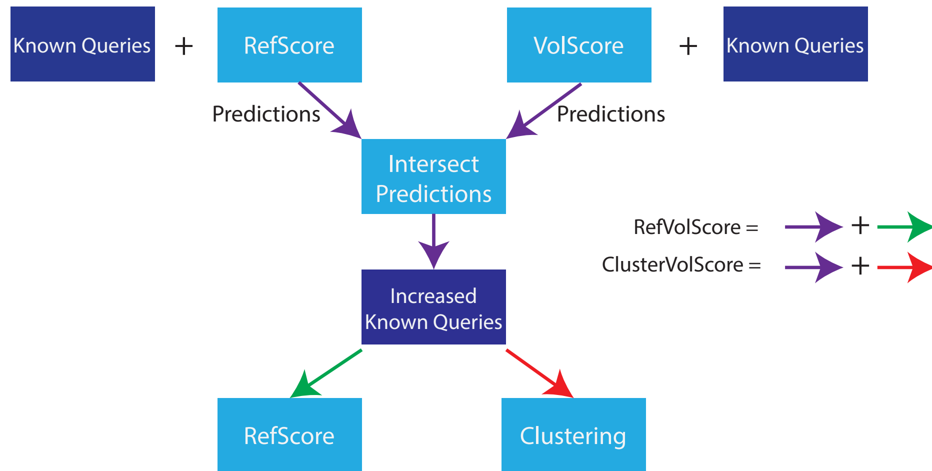


Figure 5.1: Volscore Attacks Overview

5.4. VolScore

The VolScore is the first attack that we designed by utilizing the volume pattern in the RefScore algorithm. The VolScore attack is shown below as Algorithm 3. We have highlighted the changes in blue compared to RefScore as shown in Algorithm 2. Instead of using the co-occurrence submatrices for keyword and trapdoor as input, we now utilize the volume pattern to create covolume submatrices as input. The covolume submatrices replace each line of code where the original co-occurrence submatrices are used, such as for the scoring on line 12, and also for adding columns using new known queries on line 26 of the algorithm. The core elements and the matching technique remain the same as originally shown by Damie et al. [12] in Algorithm 2. The only difference is that we apply this technique to a covolume matrix instead of a co-occurrence matrix.

This change has consequences for the accuracy. The prediction accuracy is much lower compared to RefScore, but it does contain a small portion of useful predictions. As mentioned before, we do not aim here to have a higher accuracy here than RefScore, instead, we aim to acquire additional knowledge with this small part of useful predictions.

Algorithm 3 VolScore

Require: $K_{sim}, V^s_{kw}, Q, V^s_{td}$, KnownQ, RefSpeed
1: $final_pred \leftarrow []$
2: unknownQ $\leftarrow Q$
3: **while** unknown $Q \neq \emptyset$ **do**
4: % 1. Extract the remaining unknown queries
5: unknownQ $\leftarrow \{td : (td \in Q) \wedge (\nexists kw \in K_{sim} : (td, kw) \in KnownQ)\}$
6: $temp_pred \leftarrow []$
7:
8: % 2. Propose a prediction for each unknown query
9: **for all** $td \in$ unknownQ **do**
10: $cand \leftarrow []$ { The candidates for trapdoor td}
11: **for all** $kw \in K_{sim}$ **do**
12: $s = -\ln(||V^s_{kw}[kw] - V^s_{td}[td]||)$
13: append {"kw": kw, "score":s} to $cand$
14: **end for**
15: Sort $cand$ in descending order according to the score.
16: certainty $\leftarrow$ score($cand$[0] - score($cand$[1])
17: append (td, kw($cand$[0]), certainty) to $temp_pred$
18: **end for**
19:
20: % 3. Either stop the algorithm or keep refining.
21: **if** | unknownQ | < RefSpeed **then**
22: $final_pred \leftarrow$ KnownQ $\cup$ $temp_pred$
23: unknownQ $\leftarrow \emptyset$
24: **else**
25: Append the RefSpeed most certain predictions from $temp_pred$ to KnownQ
26: Add the columns corresponding to the new known queries to V^s_{kw} and V^s_{td}
27: **end if**
28: **end while**
29: **return** $final_pred$

The algorithm has three important phases. In the first phase, we need to have covolume submatrices prepared and extract the remaining unknown queries for which a prediction needs to be made. In the second phase, the scoring mechanism is applied and we make a prediction for each unknown query. And in the last phase, the algorithm either stops and returns its predictions, or expands the new knowledge into the covolume submatrices for the next iteration. We will discuss each phase in the following sections.

5.4.1. Preparing covolume matrices

As we discussed before in 4.3.4, a simple approach where we utilize the volume in a way similar to VolAn [3] as a filter inside RefScore is not effective. We believe it is not effective because in the case of RefScore, the document knowledge is only similar documents, instead of partial document knowledge. When using volume numbers that are returned by issuing single keywords, the numbers can have large deviations between the auxiliary volume numbers and the real dataset.

We found a better way to utilize the volume pattern more effectively, by using covolume. In the RefScore attack, the access pattern is leaked. The co-occurrence pattern is the backbone of this attack. The attacker is able to learn which document identifiers are accessed on single keywords. And when more queries/trapdoors are issued, the attacker is able to learn information about documents that contain both keywords. The co-occurrence is calculated and utilized for the attack.

In the VolScore attack, we use a similar approach but with an additional volume pattern. From each query that is issued, we learn the document identifiers (access pattern/co-occurrence pattern) as well as the volume of each returned document (volume pattern). From each pair of queries, the attacker learns about document identifiers that contain both keywords, as well as the volume of those documents. The

sum of the returned document volumes that contain both keywords (or queries) divided by the number of documents is the covolume. The details to compute the covolume are shown below in Algorithm 4.

Algorithm 4 Compute covolume

Require: kq_set, inv_index, vol_array, nr_docs
1: results ← []
2: **for all** $kw_i \in$ kq_set **do**
3: kw_i_docs ← inv_index[kw_i]
4: **for all** $kw_j \in$ kq_set **do**
5: kw_j_docs ← inv_index[kw_j]
6: co_docs ← intersect(kw_i_docs, kw_j_docs)
7: **end for**
8: **if** length(co_docs > 0) **then**
9: tvol ← 0
10: **for all** doc ∈ co_docs **do**
11: tvol ← tvol + vol_array[doc]
12: **end for**
13: co_vol_result ← tvol / nr_docs
14: **else**
15: co_vol_result ← 0
16: **end if**
17: append(co_vol_result) to results
18: **end for**
19: **return** results

The parameter `kq_set` is a set of keywords or queries. This is because the covolume needs to be calculated for each keyword pair, as well as each trapdoor pair. The `inv_index` is the inverted index that contains a mapping from keyword to documents, or from query to document identifiers. The volume of each document is stored in `vol_array`. The parameter `nr_docs` is the amount of documents from the similar auxiliary document knowledge when computing for keywords, or the predicted amount of documents stored on the server when computing for queries.

We predict this number by using known queries that are known as input for RefScore. A known query is a pair (td_i, kw_i) with a known trapdoor and keyword by the attacker. Due to the access pattern leakage, the amount of documents that contain trapdoor td_i is known and can be divided by the amount of documents that contain kw_i from the auxiliary dataset. If we repeat this calculation for every known query and compute the mean, we obtain an estimated ratio of real documents stored on the server.

In the algorithm, we compute co_docs that are documents that contain both kw_i and kw_j in line 6. And then, the covolume is computed by dividing the total volume of documents that contains both keywords by nr_docs in line 13.

In Figure 5.2 we show an example of two covolume matrices V_{kw} and V_{td} that are covolume matrices for keyword and trapdoors respectively. The diagonal consists of zeroes since we are not interested in covolumes that consist of the same keywords.

$$\text{Covol matrix } V_{kw} = \begin{array}{c} \\ kw_1 \\ kw_2 \\ \vdots \\ kw_n \end{array} \begin{array}{c} \begin{array}{cccc} kw_1 & kw_2 & \cdots & kw_n \end{array} \\ \begin{pmatrix} 0 & 20 & \cdots & 80 \\ 20 & 0 & \cdots & 40 \\ \vdots & \vdots & \ddots & \vdots \\ 80 & 40 & \cdots & 0 \end{pmatrix} \end{array} \qquad \text{Covol matrix } V_{td} = \begin{array}{c} \\ td_1 \\ td_2 \\ \vdots \\ td_l \end{array} \begin{array}{c} \begin{array}{cccc} td_1 & td_2 & \cdots & td_l \end{array} \\ \begin{pmatrix} 0 & 40 & \cdots & 80 \\ 40 & 0 & \cdots & 20 \\ \vdots & \vdots & \ddots & \vdots \\ 80 & 20 & \cdots & 0 \end{pmatrix} \end{array}$$

Figure 5.2: Covolume matrix example

From the covol matrices that we created, we need to use known queries to create two submatrices. Let's assume that the attacker has two known queries (td_l, kw_1) and (td_2, kw_2). To create the keyword

covol submatrix, we extract columns that consist of keywords from our known queries from the original covol submatrix V_{kw}, in this case, it is kw_1 and kw_2. In Figure 5.3 the column extraction is shown, and the new covol submatrix V^s_{kw} is created.

$$\text{Covol matrix } V_{kw} = \begin{array}{c} \\ kw_1 \\ kw_2 \\ \vdots \\ kw_n \end{array} \begin{pmatrix} kw_1 & kw_2 & \cdots & kw_n \\ 0 & 20 & \cdots & 80 \\ 20 & 0 & \cdots & 40 \\ \vdots & \vdots & \ddots & \vdots \\ 80 & 40 & \cdots & 0 \end{pmatrix} \xrightarrow{\text{Extract columns}} \text{Covol submatrix } V^s_{kw} = \begin{array}{c} \\ kw_1 \\ kw_2 \\ \vdots \\ kw_n \end{array} \begin{pmatrix} kw_1 & kw_2 \\ 0 & 20 \\ 20 & 0 \\ \vdots & \vdots \\ 80 & 40 \end{pmatrix}$$

Figure 5.3: Covolume submatrix for keywords

We create another covol submatrix V^s_{td} for the trapdoors as shown in Figure 5.4 but now we extract columns based on the known trapdoors instead of keywords. We also have to reorder the columns in the same order as V^s_{kw} such that the submatrices can be compared.

$$\text{Covol matrix } V_{td} = \begin{array}{c} \\ td_1 \\ td_2 \\ \vdots \\ td_l \end{array} \begin{pmatrix} td_1 & td_2 & \cdots & td_l \\ 0 & 40 & \cdots & 80 \\ 40 & 0 & \cdots & 20 \\ \vdots & \vdots & \ddots & \vdots \\ 80 & 20 & \cdots & 0 \end{pmatrix} \xrightarrow{\text{Extract columns + reorder}} \text{Covol submatrix } V^s_{td} = \begin{array}{c} \\ td_1 \\ td_2 \\ \vdots \\ td_l \end{array} \begin{pmatrix} td_l & td_2 \\ 80 & 40 \\ 20 & 0 \\ \vdots & \vdots \\ 0 & 20 \end{pmatrix}$$

Figure 5.4: Covolume submatrix for trapdoors

5.4.2. Scoring mechanism

In the second phase of the algorithm, we need to make a prediction for each unknown query and apply a scoring mechanism. For all keywords that are in the similar dataset a score number is assigned for the current unknown trapdoor. From the previous example td_1 is an unknown trapdoor, and the trapdoor vector from covol submatrix V^s_{td} is row td_1. This trapdoor vector is compared with each possible keyword vector that comes from each row from covol submatrix V^s_{kw} by calculating the Euclidean distance between the two vectors, followed by applying the negative natural logarithm on the result. This transforms the score numbers so that the focus is on the order of magnitude and readability, especially when score numbers could be very small and close to zero.

$$\text{Covol submatrix } V^s_{td} = \begin{array}{c} \\ td_1 \\ td_2 \\ \vdots \\ td_l \end{array} \begin{pmatrix} td_l & td_2 \\ 80 & 40 \\ 20 & 0 \\ \vdots & \vdots \\ 0 & 20 \end{pmatrix} \xleftrightarrow{\text{Compare row vectors}} \text{Covol submatrix } V^s_{kw} = \begin{array}{c} \\ kw_1 \\ kw_2 \\ \vdots \\ kw_n \end{array} \begin{pmatrix} kw_1 & kw_2 \\ 0 & 20 \\ 20 & 0 \\ \vdots & \vdots \\ 80 & 40 \end{pmatrix}$$

Figure 5.5: Computing the score

In Figure 5.5 trapdoor vector [80, 40] from V^s_{td} is compared with each row vector: [0, 20], [20, 0], ..., [80, 40] from V^s_{kw}. The last row for keyword kw_n has an Euclidean distance of exactly zero and is the closest to td_1. Usually, this does not occur and we omitted this check in Algorithm 3, but in the implementation the score is set to infinite. In the end, all scores from each keyword are calculated and sorted in descending order. The certainty is the score between the highest and the second highest candidate keyword. In this example, kw_n has the highest score of infinite, so the certainty will also be very high. So this candidate keyword, together with the current trapdoor td_1 and its certainty will be added to a list of temporary predictions `temp_pred`. The algorithm proceeds to do the same process but now with the remaining unknown trapdoors and fills the temporary predictions list, where each entry contains a trapdoor, a candidate keyword, and a certainty. The actual decision is made in the last phase of the algorithm which we discuss next.

5.4.3. Decision making

In the last phase of the algorithm, the algorithm either stops or keeps refining. If the algorithm keeps refining, a RefSpeed amount of predictions from `temp_pred` is added to the list of known queries. These

are the predictions with the highest certainties. The algorithm has acquired more known queries now, so the covolume submatrices V_{kw}^s and V_{td}^s can expand its columns with new known queries. So for the next refinement, a larger portion of the original covolume matrices will be used as submatrices.

In the previous examples, we originally had two known queries (td_l, kw_1) and (td_2, kw_2). For simplicity, let's assume that we have only one new known query: (td_1, kw_n). In Figure 5.6 the old covol submatrices are expanded by a green column, which reflects the newly added known query. After expansion, the algorithm repeats itself and uses the new covol submatrices until the stop condition is met. The algorithm stops if the number of unknown queries is less than the refinement speed. If so, the stopping criteria are set and the known queries with the current `temp_pred` are returned as the final prediction.

$$\text{New covol submatrix } V_{td}^s = \begin{array}{c} \\ td_1 \\ td_2 \\ \vdots \\ td_l \end{array} \begin{array}{c} \begin{array}{ccc} td_l & td_2 & td_1 \end{array} \\ \begin{pmatrix} 80 & 40 & 0 \\ 20 & 0 & 40 \\ \vdots & \vdots & \vdots \\ 0 & 20 & 80 \end{pmatrix} \end{array} \qquad \text{New covol submatrix } V_{kw}^s = \begin{array}{c} \\ kw_1 \\ kw_2 \\ \vdots \\ kw_n \end{array} \begin{array}{c} \begin{array}{ccc} kw_1 & kw_2 & kw_n \end{array} \\ \begin{pmatrix} 0 & 20 & 80 \\ 20 & 0 & 40 \\ \vdots & \vdots & \vdots \\ 80 & 40 & 0 \end{pmatrix} \end{array}$$

Figure 5.6: Covol submatrices expansion

5.5. RefVolScore

Previously, we discussed the RefScore and VolScore attacks. We combined both attacks and call this the RefVolScore attack as shown in Algorithm 5.

The idea behind this attack is that the algorithm first runs the VolScore attack which uses the `volume pattern` with `co-occurrence pattern` to obtain results, as well as running the RefScore attack to obtain another result. From both results, we find predictions that are found in both results and are not already known. This means by using two different methods we have found the same new trapdoor to keyword assignments. Whereas if we only run VolScore or RefScore, the attacker does not know which prediction is correct. Since the same predictions are made with two different methods, it is highly likely that this is a correct prediction. These new known queries are then appended to the original known queries list. Then a fresh run of RefScore is run using these updated known queries to obtain a higher accuracy than was previously possible with RefScore by itself.

Algorithm 5 RefVolScore

Require: requirements $VolScore$, requirements $RefScore$

```
1:  % Runs VolScore attack
2:  results_vol_score ← VolScore
3:
4:  % Runs Refined Score attack
5:  results_ref_score ← RefScore
6:
7:  % Retrieves the list of trapdoors from the {td : kw} predictions
8:  tds1 ← get keys from results_vol_score
9:  tds2 ← get keys from results_ref_score
10:
11: % Intersecting trapdoors between the two results
12: intersect_tds ← intersect(tds1, tds2)
13:
14: % Initialize new known queries
15: new_known_queries ← { }
16:
17: % Find new known queries
18: for all td ∈ intersect_tds do
19:     % Check if same keyword assigned and not used before
20:     if results_ref_score[td] == results_vol_score[td] and
21:             results_ref_score[td] not in new_known_queries and
22:             not in KnownQ then
23:         append (td : results_ref_score[td]) to new_known_queries
24:     end if
25: end for
26:
27: % Add the new known queries to the original known queries list
28: update(KnownQ, new_known_queries)
29:
30: % Run fresh run of RefScore, but with new known queries
31: results_ref_vol_score ← RefScore
32:
33: return results_ref_vol_score
```

5.6. ClusterVolScore

The original authors Damie et al. [12] use clustering in a different way where a cluster of keywords is assigned to each trapdoor and returned, instead of a single trapdoor to keyword assignment which is what we aim to do. We utilize clustering as a new attack in ClusterVolScore to improve the last part of the attack chain after running RefScore and VolScore. In subsection 4.3.3 we discussed that the clustering idea we had, is to dynamically adjust the refinement speed based on the certainty instead of using a fixed refinement speed during the refinement process. When the refinement speed is set too high, the predictions become less accurate since many predictions are added at the same time even with low certainty. And when the refinement speed is too low, the algorithm performs slower and it could also impact the accuracy if only a small amount of predictions are added which by coincidence could be incorrect.

We illustrate the idea behind ClusterVolscore with an example that shows how we apply clustering in ClusterVolScore in Figure 5.7. We have a list of temporary predictions with certainties, those are numbers that indicate how certain trapdoor to keyword assignments are. The max refinement speed is in this example shown as 10, this is a maximum and not the same as a fixed refinement speed as in RefScore.

The original RefScore attack as shown in Algorithm 2 would choose to add all 10 elements that are

highlighted in bold to the new known queries if the RefSpeed is 10. However, we also accidentally add predictions that have low certainties compared to the rest. Instead, we utilize clustering to compute candidate clusters from size 1 up to `Max_Ref_Speed` which is 10 in this example. For each cluster size, the difference is computed between the last element in the cluster and the next element outside of the cluster. The cluster size that has the largest difference value is chosen, therefore all elements within that cluster are added as known queries which are highlighted in green in Figure 5.7.

Temp_pred with certainties = [**8.2** **8** **7.9** **3** **2.5** **2.4** **2.3** **2** **1.9** **1.5** 1.4 ...]
Max_Ref_Speed = **10**

Best candidate clustering:
Cluster_size_1 = [8.2], diff = 8.2 - 8 = 0.2
Cluster_size_2 = [8.2, 8], diff = 8 - 7.9 = 0.1
Cluster_size_3 = [8.2, 8, 7.9], diff = 7.9 - 3 = 4.9
...
Cluster_size_Max_RefSpeed = [8.2, 8, ..., 1.5], diff = 0.1

Figure 5.7: Clustering example

In algorithm 6 the computation of differences is shown, in which the index that has the largest difference of certainty is computed. Considering the previous example, the index with the largest difference of certainty is 2 if the index starts as zero. So we can then proceed to add all elements from zero up to and including index 2 as known queries.

This algorithm shares some similarities to the best candidate clustering algorithm from Damie et al. [12] since it is based on their idea. The main difference is that we determine the best candidate index instead of the best candidate cluster. In addition we use a list of certainties as input instead of a score set. Their algorithm finds the best candidate cluster from a score set that is assigned to a trapdoor, or in other words it assigns a cluster of candidate keywords to a single trapdoor. Instead, we have a list of certainties that consists of all trapdoor to keyword assignments. From this list, we aim to find the index that is the cluster with the largest difference in certainty. By finding this index, we change the refinement speed so that we only add those predictions and those predictions consist of a single keyword to trapdoor assignment.

Algorithm 6 index_max_diff

```
1:  % sorted_tuples contain tuples of the form (td, kw, certainty) sorted on certainty
Require: sorted_tuples, max_ref_speed
2:
3:  % Take a subset of all sorted tuples.
4:  % We add 1 element so that we can make a comparison for the last element.
5:  sub_tuples ← sorted_tuples[:(max_ref_speed + 1)]
6:
7:  diff_list ← []
8:  current_index ← 0
9:
10: % Loop through tuples but without the additional element we added before.
11: for all tuple ∈ sub_tuples[:-1] do
12:     % Calculates difference of certainties for current_index
13:     append (current_index, tuple[2] - sub_tuples[current_index + 1][2]) to diff_list
14:     current_index ← current_index + 1
15: end for
16:
17: % Get max diff and its index
18: ind_max_diff ← 0
19: if len(diff_list) > 0 then
20:     % Max based on diff value and retrieve index
21:     ind_max_diff ← get_index(max(diff_list))
22: else
23:     % Only 1 element, so index is zero.
24:     ind_max_diff ← 0
25: end if
26:
27: % Returns the index that has the largest difference of certainties
28: return ind_max_diff
```

In algorithm 7 the clustering algorithm is shown. Highlighted in blue are the differences between this algorithm and RefScore from algorithm 2. We use a maximum refinement speed instead of a static refinement speed. The current new refinement speed is set as the index with the largest difference of certainty that is retrieved from calling algorithm 6.

We have not shown the ClusterVolScore algorithm, because it is only a small modification from RefVolScore as shown in algorithm 5. Instead of running a fresh run of RefScore on line 31, it should be replaced by calling the clustering algorithm with a maximum refinement speed.

Algorithm 7 Clustering

```
Require: requirements RefScore, max_ref_speed
 1: final_pred ← []
 2: unknownQ ← Q
 3: while unknown Q ≠ ∅ do
 4:     % 1. Extract the remaining unknown queries
 5:     unknownQ ← {td : (td ∈ Q) ∧ (∄kw ∈ K_sim : (td, kw) ∈ KnownQ)}
 6:     temp_pred ← []
 7:
 8:     % 2. Propose a prediction for each unknown query
 9:     for all td ∈ unknownQ do
10:         cand ← [] { The candidates for trapdoor td}
11:         for all kw ∈ K_sim do
12:             s = − ln(||C^s_kw[kw] − C^s_td[td]||)
13:             append {"kw": kw, "score":s} to cand
14:         end for
15:         Sort cand in descending order according to the score.
16:         certainty ← score(cand[0] - score(cand[1])
17:         append (td, kw(cand[0]), certainty) to temp_pred
18:     end for
19:
20:     Sort temp_pred on certainties in descending order
21:
22:     % Call index_max_diff algorithm.
23:     % We add 1 for correct array splicing, and if index is 0, it means new_ref_speed becomes 1.
24:     new_ref_speed ← index_max_diff(temp_pred, max_ref_speed) + 1
25:
26:     % 3. Either stop the algorithm or keep refining.
27:     if | unknownQ | < max_ref_speed then
28:         final_pred ← KnownQ ∪ temp_pred
29:         unknownQ ← ∅
30:     else
31:         Append the new_ref_speed most certain predictions from temp_pred to KnownQ
32:         Add the columns corresponding to the new known queries to C^s_kw and C^s_td
33:     end if
34: end while
35: return final_pred
```

5.7. Countermeasures

Now we discuss countermeasures that are relevant against these new attacks. The VolScore utilizes the co-occurrence leakage pattern (access pattern), together with the volume pattern. So in order to guard against the new attacks, these leakage patterns need to be suppressed to reduce the effectiveness of the attacks.

5.7.1. Padding

The first countermeasure that is relevant is to use padding as a countermeasure to the access pattern leakage from Cash et al. [7]. The idea behind this countermeasure is to disguise the exact amount of documents that are returned when issuing a query. This is done by adding fake documents in the response to a client.

These fake entries can be created in two ways. Let's assume we have an occurrence matrix with documents as rows and keywords as columns, and each cell with a value of 1 or 0 indicates that a keyword is inside a document or not, respectively. And let integer n be a predetermined padding number, where we pad each response to a multiple of n.

In the first way, we perform the following. For each keyword which is the column, we can count how many documents contain this keyword, where the cell value is equal to 1. Each cell that has a value of zero can be used as a possible fake entry for padding so that we have a multiple of n documents that are returned. We save all possible row indices into a list of possible fake entries. From this list, we choose a random sample of indices and change the cell values to 1 so that it matches the required amount of fake entries that we need for padding. This means that documents that originally did not contain the keyword are now also returned by having their value changed to a 1.

The second way is when we do not have enough documents to pad to a multiple of n. In this case, we extend the original matrix with fake documents by adding rows that contain all zeroes and adding these indices to the list of possible fake entries.

In the end, the fake entries are selected randomly from the list of possible fake entries, and the value of the cells is changed to a 1. This means that each response will return a multiple of n documents for each query. The drawback of this countermeasure is that the receiver has now additional overhead, and needs to filter out the fake documents.

5.7.2. Volume hiding

The second countermeasure is to hide the volume pattern using a naive approach from Kamara and Moataz [20]. It is naive because it adds additional storage overhead, but for this study, it is sufficient to observe the effectiveness of a simplistic volume hiding approach.

For each document, we compute the volume and determine the maximum volume among all documents. For each document, we increase the volume by adding spaces inside a document to match the maximum volume. This means that all documents will return the same volume during a response to a query.

6 Experiments

In this chapter, we describe the experiments to evaluate the performance of the new attacks. First, we provide some background information about the specific data we used as our datasets, as well as a comparison between them. Next, we describe the methodology of how the experiments are setup and evaluated. Finally, we analyze and discuss the results that we obtained from running the attacks on the datasets.

6.1. Datasets

6.1.1. Enron dataset

The first dataset that we used for our experiments is the Enron dataset [10] from May 7, 2015. This dataset is very popular and commonly used to show the performance of SSE attacks such as for IKK, Count, and Score attacks [19, 7, 12]. This dataset consists of a collection of approximately 500.000 real emails from within a company called Enron. This a representative dataset to test the effectiveness of our attacks, since this is a dataset that could be used to store and upload encrypted emails to a cloud server, and later selectively search and retrieve it. Therefore, this fits the background of SSE. Besides, there is a lack of other realistic alternatives as mentioned by Gui et al. [18]. Another reason is that since the Score and RefScore attacks utilized this dataset, we should be able to reproduce their results and compare them with our new attacks.

The Enron dataset contains a `maildir` directory, and within there are exactly 150 folders which are the users. For each user, there are more folders such as `_sent_mail`, `inbox`, and more. Inside the `_sent_mail` folder there are many numbered files, which are the sent emails for this user. We used all the emails inside the `_sent_mail` folder for each user as a dataset for our experiments which is 30.109 emails. Below in Figure 6.1 is an example email from file 3 inside folder `king-j/_sent_mail`.

6.1.2. Apache Lucene

The second dataset we used for the experiments is from the Apache Lucene mailing list [14]. This dataset is also used in other related works such as the Count Attack [7] and Score attack [12]. We follow the setup from the Score attack to use the "java-user" mailing list between the years 2002 and 2011. We utilized a small script [13] that downloads and extracts each email automatically for that time period. We also provided this in the appendix Appendix A. The dataset consists of <year><month>.mbox files, where each .mbox file contains all the emails from that date. Below in Figure 6.2 we show an example email from `200203.mbox`.

6.1.3. Wikipedia

The third dataset we used is the simplified Wikipedia dataset from October 1st, 2023 [39] which is a smaller dataset than a full Wikipedia dump. We utilized a tool called PlainTextWikipedia [36] to extract and parse the dataset to plaintext files. In total, there are 236.679 documents after extraction, which is much larger than the previous two datasets. Therefore, we decided to take a subset to make it more

comparable to the Enron dataset. For this purpose, we utilized the first 30.000 documents as a dataset for the experiments. In Figure 6.3 we show an example file, where the content inside the ”text” field is used for the experiments.

Message-ID: <25368250.1075855501164.JavaMail.evans@thyme>
Date: Tue, 15 May 2001 02:09:00 -0700 (PDT)
From: jeff.king@enron.com
To: dbartley@o2wireless.com
Subject: Re: Weather
Mime-Version: 1.0
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit
X-From: Jeff King
X-To: ”Dan Bartley” <dbartley@o2wireless.com>
X-cc:
X-bcc:
X-Folder: \Jeff_King_Jun2001\Notes Folders\’sent mail
X-Origin: King-J
X-FileName: jking.nsf

remember how i whipped that sharks ass with the billy club? That’s how my day has started. me being the shark.

Figure 6.1: Enron example email

```
From MAILER-DAEMON Fri Mar 1 06:21:18 2002
Return-Path:
<lucene-user-return-1039-qmlist-jakarta-archive-luceneuser=
jakarta.apache.org@jakarta.apache.org>
Delivered-To: apmail-jakarta-lucene-user-archive@apache.org
Received: (qmail 17978 invoked from network); 1 Mar 2002 06:21:18 -0000
Received: from unknown (HELO nagoya.betaversion.org) (192.18.49.131)
by daedalus.apache.org with SMTP; 1 Mar 2002 06:21:18 -0000
Received: (qmail 18104 invoked by uid 97); 1 Mar 2002 06:21:27 -0000
Delivered-To: qmlist-jakarta-archive-lucene-user@jakarta.apache.org
Received: (qmail 18043 invoked by uid 97); 1 Mar 2002 06:21:26 -0000
Mailing-List: contact lucene-user-help@jakarta.apache.org; run by ezmlm
Precedence: bulk
List-Unsubscribe: <mailto:lucene-user-unsubscribe@jakarta.apache.org>
List-Subscribe: <mailto:lucene-user-subscribe@jakarta.apache.org>
List-Help: <mailto:lucene-user-help@jakarta.apache.org>
List-Post: <mailto:lucene-user@jakarta.apache.org>
List-Id: ”Lucene Users List” <lucene-user.jakarta.apache.org>
Reply-To: ”Lucene Users List” <lucene-user@jakarta.apache.org>
Delivered-To: mailing list lucene-user@jakarta.apache.org
Received: (qmail 18032 invoked from network); 1 Mar 2002 06:21:25 -0000
Message-ID: <002501c1c0e8$e9c60920$0100a8c0@medusa>
From: ”Philipp Chudinov” <morpheus@basko.ru>
To: ”Lucene Users List” <lucene-user@jakarta.apache.org>
Subject: lucene web-app & russian language
Date: Fri, 1 Mar 2002 11:18:30 +0500
MIME-Version: 1.0
Content-Type: text/plain;
charset=”koi8-r”
Content-Transfer-Encoding: 7bit
X-Priority: 3
X-MSMail-Priority: Normal
X-Mailer: Microsoft Outlook Express 6.00.2600.0000
X-MimeOLE: Produced By Microsoft MimeOLE V6.00.2600.0000
X-Spam-Rating: daedalus.apache.org 1.6.2 0/1000/N
X-Spam-Rating: daedalus.apache.org 1.6.2 0/1000/N

Hi! I was trying the lucene web-app (lucene-1.2-rc5-dev.jar). I’ve created
and indexed a simple html document with both english and russian words. it
was ANSI encoded, if I check _3.fdt from created index, I can see my
document indexed and both russian and english terms indexed (it opens in utf
encoding, i suppose). but the problem starts when searching. If i search
with russian word, it returns nothing, if I search with engglish, it returns
a result, but all russian words are returned as ? signs. I’ve changed .jsp
contenttypes to return in UTF-8 encoding, but the resukt is still the same.

So, finally, does Lucene those multilingual search or not? What am I do-
ing
wrong? I am trying to make it work since version 1.0 with russian docs, but
still no idea and no resutls :((((((


_
To unsubscribe, e-mail: <mailto:lucene-user-unsubscribe@jakarta.apache.org>
For additional commands, e-mail: <mailto:lucene-user-help@jakarta.apache.org>
```

Figure 6.2: Apache example email

{ ”id”: ”58”,
”text”: ”The word application has several uses. *In medicine, ’application’ means putting some drug or ointment usually on the skin where it is absorbed into the human body. *In computer software, an application is a type of program which is designed for a particular function. Example: word processing. It is most used to mean a Mobile app. *In business or government, an application is a (usually paper) form filled out and handed in by a person seeking a privilege from a state or company, such as work, credit, some type of license or permit, or a place to live. *At work, generally engineering, when dealing with certain materials or objects, an äpplicationïs a purpose that material or object can be used for. Wood and steel have many applications.”,
”title”: ”Application” }

Figure 6.3: Wikipedia example

6.1.4. Dataset comparison

From the previous example figures we can see that an email from Apache is longer than from Enron, and Wikipedia is in between the two. We show more details below in Table 6.1 in which we compare Enron with the Apache and Wikipedia datasets.

Table 6.1: Dataset comparison after pre-processing

	Enron	Apache	Wikipedia
Total amount of documents	30.109	50.564	30.000
Total number of unique keywords	63.029	92.402	162.074
Number of unique volumes	4940	7094	4811
Avg. amount of keywords/document	57,37	77,99	65,0

The Apache dataset contains much more documents than Enron with a larger variety of keywords. There is also a more unique number of volumes. One email from Apache on average contains 77,99 keywords, whilst on Enron it is 57,37 keywords per document. Wikipedia contains many more keywords compared to Enron and Apache, which is expected since the number of subjects is very broad in an encyclopedia. By limiting the amount of documents to 30.000, it becomes more comparable to Enron. The average amount of keywords per document is 65, which is more than Enron. The values from the table are obtained after pre-processing. We describe the methodology in the next section.

6.2. Methodology

We run the experiments among the previously discussed datasets. Each experiment is repeated 20 times to gain meaningful results for discussion. This means that for each parameter change, 20 trials are conducted within the same experiment. In each experiment, some parameters are fixed to make it close and comparable to the original results from RefScore.

At the beginning of each trial, we randomly split the dataset into a 40/60 ratio between a similar dataset and a real dataset. The similar dataset is the data that the attacker has available as auxiliary knowledge and the real dataset is the data that is stored on the server. The keywords are extracted from each dataset and described in the next section.

6.2.1. Keyword extraction

The keyword extractor parses the documents and extracts keywords from them. We used the Natural Language Toolkit (NLTK) [38] for Python to extract sentences and keywords from the documents. Each keyword is then stemmed using the PorterStemmer from the NLTK library. The keywords are also matched with stopwords to exclude, such as in, a, an, and, which, etc. In addition to these words, we also exclude common keywords in email: from, to, subject, cc, forward. The keywords are sorted based on the occurrences. From this list, the most frequent keywords are chosen to be in the vocabulary for the similar dataset and the real dataset. The amount of keywords chosen depends on the chosen

vocabulary size. We assume that the attacker is using the same keyword extraction algorithm as the client that stores documents on the server.

6.2.2. Query selection

There are many ways to select queries such as choosing queries using uniform, Zipfian, or inverse Zipfian distribution. In the experiments, we assume that each query is uniformly distributed. Therefore, within the keyword space that the server has, a random sample is chosen as queries where each keyword has the same probability of being chosen as a query. The size depends on the amount of queries which is a fixed parameter for the experiment. We do not choose duplicate queries, because the attacker is able to observe duplicate queries based on the access pattern, as well as observing the same deterministic trapdoor.

6.2.3. Adversary's knowledge

In the attacks, the adversary has acquired knowledge to perform an attack. The attacker has access to a similar dataset and is able to observe the communication traffic between the client and server, or the attacker is the server itself. We assume that both the access pattern and volume pattern are leaked to the attacker. In addition, we assume the attacker has access to some known queries. These are selected randomly for each trial from keywords that are in both similar and real keyword space.

6.2.4. Evaluation

We measure the performance of each attack by using query recovery accuracy. We follow the same definition as defined by Damie et al. [12] that excludes known queries in the calculation. The query recovery accuracy is the number of unknown queries (trapdoors) that are correctly assigned to a keyword by the attacker divided by the number of unknown queries.

$$QR_{acc} = \frac{|correctPred(UnknownQ)|}{|Q| - |KnownQ|}$$

We compare the mean attack accuracy between RefScore and all VolScore attacks in 20 trials. In the experiments, we analyze the results in different areas with specific parameter changes that include: medium to high amount of known queries, low amount of known queries, a variety of refinement speeds, vocabulary size adjustments, and countermeasures.

6.2.5. Hardware and implementation

We ran all our experiments on a laptop with the following specifications. Operating System: Arch Linux 64 bits, CPU: AMD Ryzen 7 5800H, RAM: 16GB. The code is built upon the original Score/RefScore attack [13]. We received permission from the original author to do so. We modified this code to suit our new attacks. The algorithms are implemented and run in Python 3.11.

6.3. Results

6.3.1. Score and RefScore

Our attack is based on the RefScore attack, and before we make any comparisons, we first need to set a baseline which is the RefScore attack. In order to do so, we reproduced both Score and RefScore attacks. In the following figures, we present bar plots where the bars are the mean, and the error bars indicate the standard deviation.

In Figure 6.4, the original results are shown from Damie et al. [12]. And in Figure 6.5 are the results that we obtained after running each attack for 20 runs for the different amount of known queries, with the same parameters as the original on the Enron dataset.

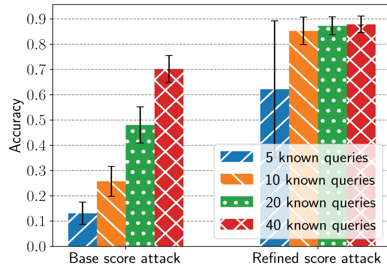


Figure 6.4: Results from Damie et al. [12]
$|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = 1.2K$
$m_{real} = 1K, |Q| = 150$, RefSpeed = 10

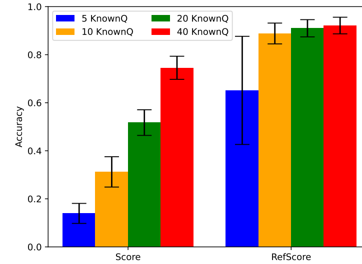


Figure 6.5: Score comparison reproduction with same parameters.

In the RefScore attack in Figure 6.5, we obtain better results for the same amount of known queries compared to the Score attack, which is the same behavior as in the original results. We can also observe that in both figures, the difference in accuracy is minimal when the attacker has 10 or more known queries for the RefScore attack. Since the attacker already has many known queries at the start, many correct predictions can be made. But, when the attacker has a minimal amount of knowledge (e.g. 5 known queries), it is much more difficult to make correct predictions. In other words, the amount of known queries plays an important role in achieving high accuracy.

On a low amount of known queries, the spread of accuracy becomes larger, and on a higher amount of known queries, it is smaller and more stable. We can observe this for 10 or more known queries in RefScore compared to 5 known queries. The reason is that on a low amount of known queries, there is more uncertainty and more room for error for assigning trapdoor and keyword predictions.

Our reproduction looks identical to the original results, therefore we can use RefScore as a baseline to compare the performance with our new attacks. Since the RefScore is the improved attack from the base score attack, we omit the base score from the next experiments and only directly compare it to RefScore.

6.3.2. RefScore and VolScore attacks

For the next experiments, We omit the 40 known queries variable to save computation time, since it does not provide much difference in accuracy compared to 20 known queries as seen from the previous results. In figures 6.6, 6.7, 6.8 we run RefScore and VolScore attacks with 5, 10 and 20 known queries on all three datasets.

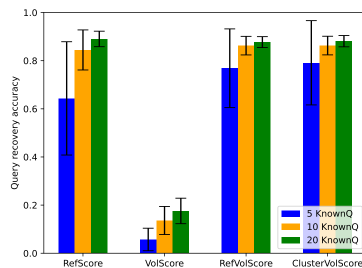


Figure 6.6: (Enron) Accuracy comparison between RefScore and VolScore attacks. Fixed parameters are:
$|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = 1.2K$
$m_{real} = 1K, |Q| = 150$, RefSpeed = 10 = MaxRefSpeed

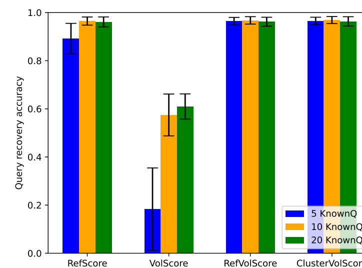


Figure 6.7: (Apache) Accuracy comparison between RefScore and VolScore attacks. Fixed parameters are:
$|D_{sim}| = 20K, |D_{real}| = 30K, m_{sim} = 1.2K$
$m_{real} = 1K, |Q| = 150$, RefSpeed = 10 = MaxRefSpeed

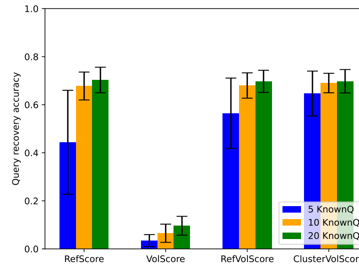


Figure 6.8: (Wiki) Accuracy comparison between RefScore and VolScore attacks. Fixed parameters are: $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = 1.2K$
$m_{real} = 1K, |Q| = 150,$ RefSpeed = 10 = MaxRefSpeed

In Figure 6.6, the VolScore attack has the lowest accuracy on all three amount of known queries. We believe that covolume has more deviation and fewer unique values compared to co-occurrence which is closer when comparing similar documents to server documents. Therefore, even when a computed distance between query and keyword is close, it is more likely that RefScore assigns a correct prediction than VolScore.

For 20 known queries the attacks RefScore, RefVolScore and ClusterVolScore score similar to each other. But, for 10 known queries, RefVolScore and ClusterVolScore have a slightly higher accuracy than RefScore. The most interesting highlight is the result on 5 known queries, where the RefVolScore and ClusterVolScore outperform RefScore with a mean of 0.76 and 0.79, respectively, compared to the mean of 0.64 from RefScore.

This trend seems to suggest that on a high amount of known queries, the attack performance is similar among all of the attacks except VolScore, and is also observed from the Apache dataset in Figure 6.7, as well as the Wikipedia dataset in Figure 6.8. The attacks already have a high amount of known queries to make accurate predictions, and therefore the effect of utilizing VolScore is limited. However, when on a low amount of known queries such as 5, the knowledge obtained from VolScore can be effectively used to outperform RefScore with RefVolScore and ClusterVolScore. In addition, the spread of accuracy is smaller when utilizing the volume pattern if we compare RefScore with RefVolScore and ClusterVolScore on all three datasets.

When we compare all three results with each other, we can observe that Apache has the highest query recovery accuracy, and Wikipedia has the lowest. We believe this is related to the properties of each dataset as we showed previously in Table 6.1. Wikipedia has the most amount of keywords with a similar amount of documents as Enron. This makes it much harder to create accurate predictions compared to Enron. The Apache dataset also has more keywords than Enron but has the best results. This is because Apache has the most amount of average keywords per document, the most documents available for the attacker, and also the most number of unique volumes.

6.3.3. Low known queries comparison

On a high amount of known queries, there is not much difference in accuracy between the attacks. Therefore, in the next experiment, we aim to investigate with a lower amount of known queries to see if we still outperform RefScore. In Figure 6.9 the results are shown for 2, 3, and 4 known queries for the Enron dataset.

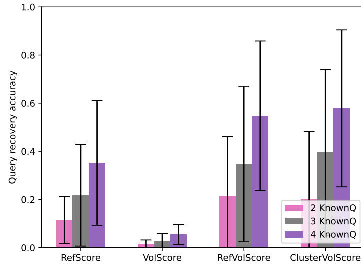


Figure 6.9: (Enron) Low known queries comparison.
Fixed parameters are:
$|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = 1.2K$
$m_{real} = 1K, |Q| = 150$, RefSpeed = 10 = MaxRefSpeed

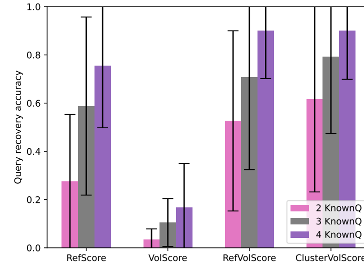


Figure 6.10: (Apache) Low known queries comparison.
Fixed parameters are:
$|D_{sim}| = 20K, |D_{real}| = 30K, m_{sim} = 1.2K$
$m_{real} = 1K, |Q| = 150$, RefSpeed = 10 = MaxRefSpeed

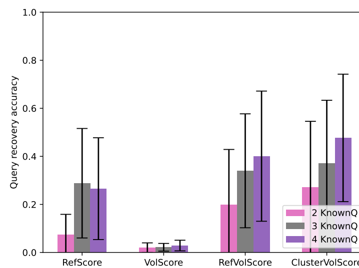


Figure 6.11: (Wiki) Low known queries comparison.
Fixed parameters are:
$|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = 1.2K$
$m_{real} = 1K, |Q| = 150$, RefSpeed = 10 = MaxRefSpeed

We observe that RefVolScore and ClusterVolScore outperform RefScore on a low amount of known queries. The figure suggests that the amount of known queries heavily impacts the performance of each attack, and the more known queries an attack has, the better it performs to a certain extent. On a low amount of known queries, RefVolScore and ClusterVolScore are able to utilize the additional known queries obtained via VolScore, and therefore obtain a higher accuracy than RefScore. Even if the accuracy of VolScore itself is low compared to RefScore, the slight edge of acquiring at least 1 more known query using VolScore has an impact on the query recovery accuracy. We can observe this from the mean of RefScore for 3 known queries is roughly the same as the mean of RefVolScore on 2 known queries. The results from Apache as shown in Figure 6.10, look similar to the Enron dataset but with overall higher accuracy. The Wikipedia results as shown in Figure 6.11 have the lowest accuracy, but RefVolScore and ClusterVolScore still perform better than RefScore. On all three datasets, the spread of accuracy is very large on a low amount of known queries. On a low amount of known queries, there is much more room for error and it becomes harder to stabilize the spread. There are two situations that can occur on a low amount of known queries. We find more correct new known queries, which results in a high accuracy. Or we find incorrect new known queries, which result in low accuracy.

We listed the details of the results from all datasets in Table 6.2. From the table, we can observe that the VolScore accuracy is correlated with the amount of newly found known queries. The higher the accuracy of VolScore is, the more likely that it finds more known queries. The 'Total KnownQ Accuracy' shows the percentage of total known queries that are correct. This consists of the original known queries with the additional new ones by using VolScore.

Table 6.2: Detailed comparison on low known queries

Dataset	Nr KnownQ	KnownQ Accuracy (%)	Newly found KnownQ	Total KnownQ Accuracy (%)	RefScore Accuracy (%)	VolScore Accuracy (%)	RefVolScore Accuracy (%)	ClusterVolScore Accuracy (%)
Enron	2	100	2.15	73.85	11.39	1.55	21.39	20.07
Enron	3	100	4.4	74.31	21.77	2.55	34.73	39.52
Enron	4	100	5.1	87.77	35.21	5.45	54.76	57.84
Apache	2	100	2.75	81.44	27.47	3.34	52.64	61.59
Apache	3	100	13.1	86.4	58.78	10.48	70.75	79.29
Apache	4	100	23.2	92.95	75.55	16.82	89.97	90.07
Wikipedia	2	100	2.15	74.04	7.47	1.99	19.83	27.06
Wikipedia	3	100	2.75	80.51	28.81	2.14	33.98	37.11
Wikipedia	4	100	3.4	85.36	26.54	2.88	40.07	47.67

In this table, we want to highlight the most important variables that influence the accuracy and how to stabilize the spread. The most important variables are the number of known queries the attacker has available, the amount of newly found known queries by using VolScore, the correctness of the total known queries, and the VolScore accuracy.

The Apache dataset with 4 known queries is the most stable in spread for multiple reasons. The VolScore accuracy is the highest because of the positive properties of the Apache dataset. Therefore, it found the most new known queries with a high total known query accuracy. If the VolScore accuracy can be improved further, the accuracy of RefVolScore and ClusterVolScore will be higher with a more stabilized spread.

6.3.4. RefSpeed comparison

In this experiment we aim to investigate the impact of the RefSpeed parameter on the accuracy of the attacks, as well as taking a deeper look at the ClusterVolScore attack. The main intuition with refinement speed is that on a lower RefSpeed, the attack is more precise but slower. On a higher RefSpeed, the attack has lower accuracy but runs faster. We fixed the number of known queries to four for this experiment.

In Figure 6.12 and Figure 6.13 we observe that RefScore has a higher query recovery accuracy when the RefSpeed is lower. This is due to being more careful with adding additional knowledge to the co-occurrence submatrices, but the trade-off is the performance. Wikipedia has the lowest accuracy for RefScore as shown in Figure 6.14. The mean of the bars are closer to each other and the refinement speed does not have as much impact compared to the other datasets.

Next, we compare RefVolScore and ClusterVolScore. In general, we can see that both RefVolScore and ClusterVolScore have very similar trends and accuracy when compared within the same dataset. RefSpeed 2 performs better than RefSpeed 8 on the Enron dataset, but performs similarly in the Apache dataset due to the high accuracy of VolScore. In the Wikipedia dataset in Figure 6.14, the difference is minimal between RefSpeed 2 and 8.

The impact of the refinement speed depends on the attack and the dataset properties. For the original RefScore, a lower refinement speed generally means more precise results. For RefVolScore and ClusterVolScore, the refinement speed impact depends on the dataset properties. For the Apache case where a high amount of documents are available to the attacker, and a high average amount of keywords per document, the refinement speed has minimal impact. For the Wikipedia case, fewer documents are available to the attacker with a very high total amount of unique keywords, in this case, the refinement speed also has a minimal impact. The only case where the refinement speed has an impact is the Enron case, which has the lowest total amount of unique keywords from all datasets.

The spread is more controlled on all RefSpeeds for RefVolScore compared to RefScore. The best result is to use RefVolScore with a low RefSpeed which results in a smaller spread and higher accuracy compared to RefScore. A higher RefSpeed is preferred when preferring a faster runtime but with less accuracy and higher spread.

When comparing RefVolScore and ClusterVolScore it might not be immediately apparent that the latter performs worse than RefVolScore, the results are too close to each other. In Table 6.3 we show additional details to support our analysis.

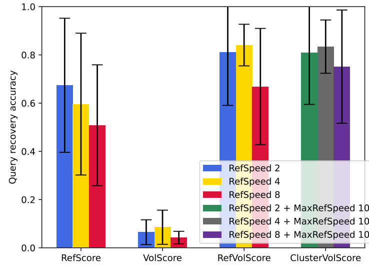


Figure 6.12: (Enron) RefSpeed comparison.
Fixed parameters are:
$|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = 1.2K$
$m_{real} = 1K, |Q| = 150$, KnownQ = 4, MaxRefSpeed = 10

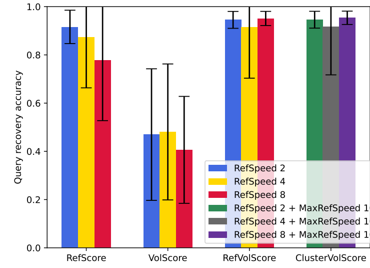


Figure 6.13: (Apache) RefSpeed comparison.
Fixed parameters are:
$|D_{sim}| = 20K, |D_{real}| = 30K, m_{sim} = 1.2K$
$m_{real} = 1K, |Q| = 150$, KnownQ = 4, MaxRefSpeed = 10

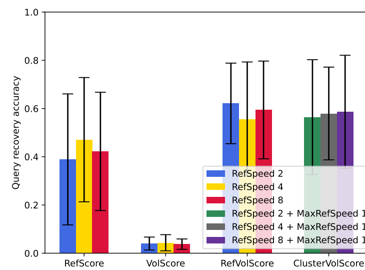


Figure 6.14: (Wiki) RefSpeed comparison.
Fixed parameters are:
$|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = 1.2K$
$m_{real} = 1K, |Q| = 150$, KnownQ = 4, MaxRefSpeed = 10

Table 6.3: Detailed comparison on RefSpeed

Dataset	Attacks	RefSpeed	Mean Dynamic RefSpeed	Runtime (sec)	Accuracy (%)
Enron	RefVolScore	2	N/A	40.93	81.03
Enron	ClusterVolScore	2	3.18	25.35	80.89
Enron	RefVolScore	4	N/A	20.97	84.08
Enron	ClusterVolScore	4	3.07	25.62	83.42
Enron	RefVolScore	8	N/A	11.39	66.88
Enron	ClusterVolScore	8	3.18	25.93	75.14
Apache	RefVolScore	2	N/A	22.82	94.52
Apache	ClusterVolScore	2	3.94	11.39	94.59
Apache	RefVolScore	4	N/A	11.61	91.51
Apache	ClusterVolScore	4	3.9	11.06	91.82
Apache	RefVolScore	8	N/A	7.35	95.1
Apache	ClusterVolScore	8	3.77	13.74	95.38
Wikipedia	RefVolScore	2	N/A	37.98	62.12
Wikipedia	ClusterVolScore	2	2.76	26.68	56.44
Wikipedia	RefVolScore	4	N/A	19.19	55.65
Wikipedia	ClusterVolScore	4	2.85	25.59	57.95
Wikipedia	RefVolScore	8	N/A	10.28	59.42
Wikipedia	ClusterVolScore	8	2.86	26.37	58.63

The runtime indicates the time to execute the third part of the algorithm, which is either one fresh run of RefScore or the clustering algorithm. On RefSpeed 8 for the Enron dataset, ClusterVolScore outperforms RefVolScore. However, it is not fair because the mean dynamic RefSpeed is 3.18 which is much lower than 8, and the runtime is also slower than RefVolScore.

If we compare RefSpeed 4 on the Enron dataset which is closer to the mean dynamic RefSpeed, RefVolScore is slightly better than ClusterVolScore. This is not great, because ClusterVolScore on average runs at a slightly lower speed of 3.07 with higher runtime.

On the Apache dataset, we can see the same problem. For RefSpeed 8, ClusterVolScore performs very similarly to RefVolScore, but runs much slower with a lower mean dynamic RefSpeed of 3.77, and a much longer runtime.

The same is observed when comparing on the Wikipedia dataset for Refspeed 4 and 8. Similar results but ClusterVolScore has much lower dynamic RefSpeed compared to RefSpeed, and longer runtime.

Overall, from these results, we can conclude that ClusterVolScore unfortunately does not perform as well as RefVolScore. We did not expect that the mean dynamic RefSpeed would be so low, which is comparable to running RefVolScore on a low RefSpeed between 3 and 4. Using clustering and using a dynamic RefSpeed based on the difference of certainties is an interesting concept, but it seems most of the final decisions were adding a very small amount of predictions instead of a larger amount of predictions. Therefore, this method is not as good as we initially expected. The results in accuracy are too similar, and the longer runtime makes RefVolScore a more effective attack.

6.3.5. Countermeasures

In this experiment, we explore the effectiveness of countermeasures on RefScore and RefVolScore. We omitted the other VolScore attacks and chose the best attack instead. We do this experiment in a worse case scenario, in which the attacker has more known queries available in comparison to our previous experiments. Also, we now vary the vocabulary size to observe the impact it has on the accuracy, whereas in the previous experiments, we fixed this vocabulary size. We have also set some fixed parameters in a way to match the countermeasure setup from the original authors.

For the Enron dataset results as seen in Figure 6.15, we observe that by increasing the vocabulary size, it becomes harder for the attacker to recover queries. A vocabulary size of 2000 with no countermeasures in place is still insufficient to guard against RefScore. The volume hiding countermeasure has no impact on the query recovery accuracy. This is correct behavior because the RefScore does not rely on the volume pattern. Padding, however, is very effective against RefScore, especially on a large vocabulary size.

For the RefVolScore in Figure 6.16, the volume hiding again has minimal impact. Volume hiding can be effective against the volume pattern leakage, however, our new attack does not solely rely on volume pattern. And with a sufficient amount of known queries, the additional volume leakage is not necessary to achieve high accuracy. In this experiment, the attacker has access to 15 known queries. From the previous results as seen in Figure 6.4 and Figure 6.5, increasing the number of known queries above 10 had minimal impact on the accuracy.

Padding is very effective on RefVolScore, this is because RefVolScore by design relies on RefScore. And when the accuracy of RefScore is largely affected, RefVolScore is also affected. On a vocabulary size of 2000 with padding, the attack is successfully countered.

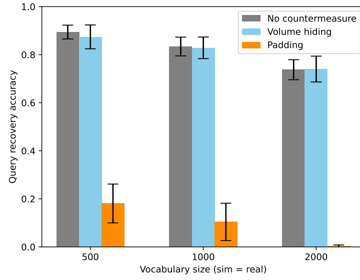


Figure 6.15: (Enron) RefScore countermeasure comparison. $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = m_{real}$, $|Q| = 0.15 * m_{real}$, KnownQ = 15, RefSpeed = $0.05 * |Q|$, $n_{pad} = 500$

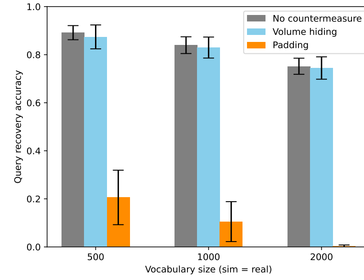


Figure 6.16: (Enron) RefVolScore countermeasure comparison. $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = m_{real}$, $|Q| = 0.15 * m_{real}$, KnownQ = 15, RefSpeed = $0.05 * |Q|$, $n_{pad} = 500$

The results for the Apache dataset are shown in Figure 6.17 and Figure 6.18. On both no countermeasures and volume hiding, the attacker is able to recover almost all queries on all three vocabulary sizes. This suggests that only increasing the vocabulary size without countermeasures is insufficient.

The effectiveness of padding is observed again in this dataset. A vocabulary size of 1000 with padding is not sufficient to guard against both attacks. Whereas, a vocabulary size of 2000 with padding is sufficient for both attacks.

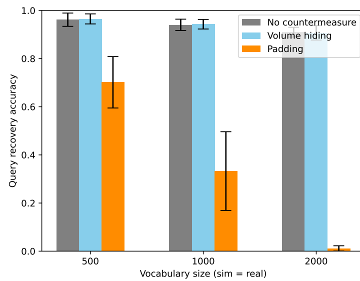


Figure 6.17: (Apache) RefScore countermeasure comparison. $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = m_{real}$, $|Q| = 0.15 * m_{real}$, KnownQ = 15, RefSpeed = $0.05 * |Q|$, $n_{pad} = 500$

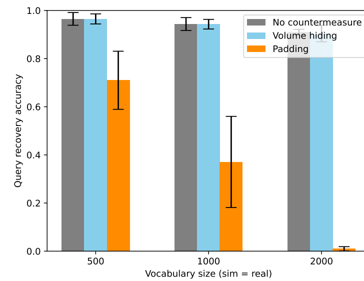


Figure 6.18: (Apache) RefVolScore countermeasure comparison. $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = m_{real}$, $|Q| = 0.15 * m_{real}$, KnownQ = 15, RefSpeed = $0.05 * |Q|$, $n_{pad} = 500$

In figures 6.19 and 6.20 the results are shown for the Wikipedia dataset. The padding countermeasure has similar impact compared to the Enron dataset, because the amount of documents in the dataset is very close to each other and is much lower than Apache.

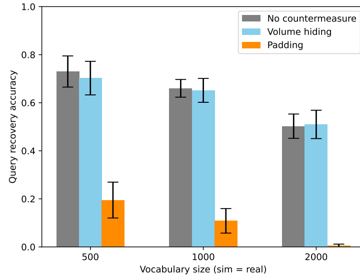


Figure 6.19: (Wiki) RefScore countermeasure comparison. $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = m_{real}$, $|Q| = 0.15 * m_{real}$, KnownQ = 15, RefSpeed = $0.05 * |Q|$, $n_{pad} = 500$

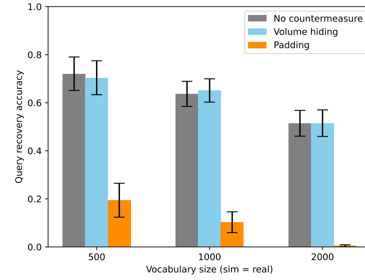


Figure 6.20: (Wiki) RefVolScore countermeasure comparison. $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = m_{real}$, $|Q| = 0.15 * m_{real}$, KnownQ = 15, RefSpeed = $0.05 * |Q|$, $n_{pad} = 500$

From the results, we can conclude that from the defender's perspective, the effectiveness of an attack depends on the dataset properties. When an attacker has potential access to a similar dataset with many documents, and where each document stored on the server has a high average amount of keywords per document, it is much more susceptible to an attack as we observed from the Apache dataset. The other two datasets contain fewer documents and have a smaller average amount of keywords per document, but they still have a non-negligible query recovery rate on low vocabulary sizes. An effective way to counter this attack is to apply padding with an increased vocabulary size.

7 Discussion

In this chapter, we discuss the results that we obtained from our experiments. First, we discuss the key findings that we obtained from our experiments. Secondly, we discuss in more detail our interpretation and implications where we identify the patterns and insight that we obtained from the results. In addition, we evaluate the limitations of our analysis and the relevance of our current analysis. Lastly, we provide future work possibilities that were not possible with our current time constraints.

7.1. Key findings

Our research problem was to design an improved inference attack with a higher query recovery rate, and how to guard against it. In the experiments that we conducted, we found that VolScore on its own is not sufficient enough to improve on RefScore. When we combine VolScore and RefScore together into RefVolScore and ClusterVolScore, we obtain an improved query recovery rate.

A key finding of utilizing the additional volume pattern is a higher query recovery accuracy and a more stabilized spread in the results. Furthermore, RefScore and RefVolScore outperform RefScore in a low known query scenario. However, it becomes much harder to stabilize the spread in this scenario. A solution would be to improve VolScore to control the results and stabilize the spread.

In addition, we compared RefVolScore with ClusterVolScore and discovered that the query recovery rate performance of ClusterVolScore is too close to RefVolScore. Considering the closeness of the performance, the worse runtime, and the low mean dynamic refspeed, we concluded that our application of clustering is not as effective as we originally expected. Therefore, RefVolScore is the preferred attack.

Finally, we applied two existing countermeasures to our attack, volume hiding and padding. Under no countermeasures with an increased vocabulary size, the query recovery rate remains high and is not recommended. We discovered that across three datasets we were able to significantly reduce the query recovery rate by utilizing padding with a large vocabulary size.

7.2. Interpretation and implications

Firstly, we compared the Score attack with RefScore and observed that RefScore is an improvement, and our reproduction of both attacks is in line with the original results. We identified a pattern in the RefScore attack. When given a high amount of known queries, the accuracy becomes higher, with a smaller spread in values, and more stable. And, when given a low amount of known queries, the accuracy becomes lower with a bigger spread. It becomes less stable with more uncertainty and more room for error. This implies that the attack is very dependent on the amount of known queries, which is one of the pieces of information an attacker has. This is in line with the analysis from the original author that mentioned documents, queries, and known queries as possible sources of information [12] that influence attack performance. A new insight is that not only the attack performance but also the spread of results or stability is influenced by the amount of information known by the attacker.

Secondly, we compared RefScore with the VolScore attacks. Here we also identified a pattern. When using the additional volume pattern in the attacks RefVolScore and ClusterVolScore, the accuracy becomes higher with a more stabilized spread, compared to the original RefScore in all 3 datasets. In addition, the dataset properties also influence the results, where Apache scores much higher than Enron and Wikipedia. Both interpretations match our previous insight. An additional leakage pattern is another piece of useful information, and some datasets might provide more information to an attacker than others. We can generalize this insight to the more an attacker knows, the higher the query recovery accuracy becomes.

Pursuing this further, we compared the attacks in a low known query scenario. Our new attacks outperform RefScore. We also noticed a pattern that there is a large spread on all results because of the low amount of known queries. It becomes much harder to stabilize the spread. When we find incorrect new known queries it can result in low accuracy, and when we find correct new known queries, it can result in high accuracy. This implies that we can make the results more stable and control the spread if VolScore can be improved further.

Additionally, we compared using different refspeeds. The main intuition is that the lower the refinement speed, the higher the accuracy becomes. From the results, we noticed that the impact depends on the dataset as well as the attack. We noticed a pattern that RefVolScore is very close to ClusterVolScore in all 3 datasets. We looked into more details and observed that ClusterVolScore has a worse runtime and low mean dynamic refspeed compared to RefVolScore. This implies that the clustering is not as effective as we originally expected, therefore we consider RefVolScore as the better attack.

Finally, we compared RefScore with RefVolScore with and without countermeasures in place. We noticed several patterns. Under no countermeasures, a higher vocabulary size is slightly more difficult for an attacker, and results in lower accuracy. However, the query recovery accuracy still remains very high in all three datasets. This implies that no countermeasures are insufficient to defend against the attacks. Volume hiding had minimal impact on all attacks and datasets, because the attacker has a high amount of known queries, and also relies heavily on co-occurrence. The Apache dataset is the easiest for the attacker, in this case, the attacker has access to many similar documents, and a high average amount of keywords per document makes it easier to recover queries. Padding is very effective for the attacks but requires a large vocabulary size for mitigation. Overall we obtained the insight that existing countermeasures are still effective to mitigate this new type of attack.

7.3. Limitations

7.3.1. Dataset in-depth analysis

From the results, we observed that depending on the dataset, the query recovery performance can have differences. We only compared the datasets on a high level in Table 6.1 which is a limitation in our analysis. Based on the results we have an intuition about which properties may have an influence on the query recovery rate, but we lack some details such as how keyword distribution differences among the datasets could play a role. Nevertheless, our current analysis is still relevant to answer the main research question since the new attack is an improvement in all three datasets that we used.

7.3.2. Known queries

There are two limitations regarding known queries. The first limitation is the known query requirement for the old and new attacks. The attacks fully rely on having some known queries available, and the attacker is not able to run the attack without known queries.

The second limitation is regarding the analysis. We chose the known queries randomly for the experiments as long as these known queries were available in both similar and real datasets. We did not take into account the quality of known queries, where each known query can have a different occurrence rate across documents, which results in a wider spread. Although we do not take the quality of known queries into account, the current approach is more practical in that the quality of known queries is uncontrolled, and therefore still relevant.

7.4. Future work

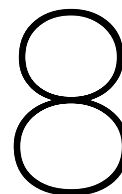
We had to limit our scope to meet the time constraints. We recognize there are possible ways to improve upon the current research we have done in this work, and we list them below.

Obfuscation countermeasure: We have explored padding and volume hiding, where volume hiding is also a form of padding. We recognize that aside from padding, the other countermeasure type is obfuscation. A future work would also investigate the performance under obfuscation, and how this compares to padding in different vocabulary sizes. And also, which countermeasure provides the least additional overhead.

Combining with file injection : In this work, we did not investigate file injection techniques. A major weakness in RefScore and our new attack is the known queries requirement, and file injection can help with obtaining known queries by injecting files into the target system.

Extending to Dynamic SSE: In this work, the scenario is modeled as static SSE, however extending this into a Dynamic SSE scenario is more relevant in a real-world scenario where the client is able to perform updates on the stored data.

Combining with other leakage patterns: In this work, we explored combining the co-occurrence pattern with the volume pattern. We recognize there are other possible ways to improve our attack. An alternative approach is to combine co-occurrence, volume, and search patterns to make this attack more powerful. We think the search pattern can be used either as an additional filter into VolScore to improve VolScore itself or as a completely separate algorithm in which we intersect predictions between the original RefScore, VolScore, and the new algorithm to improve the accuracy of new known queries.



Conclusion

In this research, we explored the impact additional leakage knowledge can have on an existing SSE attack. Specifically, we combined the `co-occurrence pattern` with the `volume pattern` into the Refined Score Attack to create a new attack with improved query recovery accuracy. We have listed the sub-questions from section 1.2 below to answer the main research question.

SQ1: *What can we learn from past attacks on SSE systems, and how do they differ from each other?*

We learned that each attack can differ in terms of attack type, document knowledge, query knowledge, and exploited leakage patterns. The matching technique between keywords and trapdoors depends on the available auxiliary information and the abused leakage pattern. Many attacks that we investigated utilize the co-occurrence leakage pattern.

SQ2: *How can we combine techniques from past SSE attacks to improve the recovery rate of an inference attack?*

We analyzed the Score and Refined Score attack algorithms [12] to determine possible ways of improvement. The main intuition we had is that an attack can be improved when the attacker has more knowledge. Therefore, we combined the `volume pattern` with the existing `co-occurrence pattern` in the Refined Score attack to create three new attacks: VolScore, RefVolScore and ClusterVolScore. These attacks utilize the matching and refinement technique that was shown in the original attack that slowly utilizes more knowledge when new known queries are acquired.

By utilizing the additional `volume pattern`, we managed to improve the query recovery rate and gain a more stabilized spread in the results. We also experimented on the setting of a low amount of known queries. The new attack has improved accuracy compared to the original attack in this setting. One problem that comes up in the low known query setting is that it becomes much more difficult to stabilize the spread. We think the spread can be more controlled if VolScore achieves higher accuracy, and one way is to combine it with other leakage patterns.

We also explored utilizing clustering to achieve a dynamic refinement speed based on the difference of certainties in the ClusterVolScore attack. This idea did not work as intended, because we did not expect that the mean dynamic refinement speed would be so low for minimal gains compared to higher refinement speeds. Therefore, we consider the RefVolScore as the better attack.

SQ3: *How effective are existing countermeasures against the improved attack?*

We explore padding and volume hiding as countermeasures. Volume hiding on its own is not sufficient to counter the RefVolScore attack on a high amount of known queries. Padding is able to decrease the query recovery accuracy on the new attack.

From these sub-questions, we can answer the main research question: *How can we design an improved inference attack with a higher query recovery rate, and how can we guard against it?*

We designed an improved inference attack by performing a literature study on related work, and revisiting the Score and Refined Score algorithm [12]. Then we improved the query recovery rate by acquiring more known queries by utilizing the `volume pattern`. The RefVolScore attack can successfully be mitigated by utilizing padding together with a large vocabulary size of keywords since an attacker can still recover sufficient queries on a small vocabulary size.

In addition, we observed during the experiments that the dataset itself can influence the query recovery accuracy. From the results, there is a correlation that having a dataset with more documents, more unique volumes, and a higher average amount of keywords per document results in high query recovery accuracy of the attack. When the attacker has more knowledge due to the properties of a dataset, all attacks achieve higher accuracy. The properties of the dataset should be taken into account when the performance of an attack is evaluated.

Reference

[1] Michel Abdalla et al. *Searchable Encryption Revisited: Consistency Properties, Relation to Anonymous IBE, and Extensions*. Cryptology ePrint Archive, Paper 2005/254. https://eprint.iacr.org/2005/254. 2005. URL: https://eprint.iacr.org/2005/254.

[2] Rakesh Agrawal et al. "Order Preserving Encryption for Numeric Data". In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD '04. Paris, France: Association for Computing Machinery, 2004, pp. 563–574. ISBN: 1581138598. DOI: 10.1145/1007568.1007632. URL: https://doi.org/10.1145/1007568.1007632.

[3] Laura Blackstone, Seny Kamara, and Tarik Moataz. "Revisiting Leakage Abuse Attacks". In: Jan. 2020. DOI: 10.14722/ndss.2020.23103.

[4] Dan Boneh et al. "Public Key Encryption with Keyword Search". In: *Advances in Cryptology - EUROCRYPT 2004*. Ed. by Christian Cachin and Jan L. Camenisch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 506–522. ISBN: 978-3-540-24676-3.

[5] Christoph Bösch et al. "A Survey of Provably Secure Searchable Encryption". In: *ACM Comput. Surv.* 47.2 (Aug. 2014). ISSN: 0360-0300. DOI: 10.1145/2636328. URL: https://doi.org/10.1145/2636328.

[6] David Cash et al. "Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries". In: *Advances in Cryptology – CRYPTO 2013*. Ed. by Ran Canetti and Juan A. Garay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 353–373. ISBN: 978-3-642-40041-4.

[7] David Cash et al. "Leakage-Abuse Attacks Against Searchable Encryption". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 668–679. ISBN: 9781450338325. DOI: 10.1145/2810103.2813700. URL: https://doi.org/10.1145/2810103.2813700.

[8] Melissa Chase and Seny Kamara. "Structured Encryption and Controlled Disclosure." In: *IACR Cryptology ePrint Archive* 2011 (Jan. 2011), p. 10.

[9] Tianyang Chen et al. "The Power of Bamboo: On the Post-Compromise Security for Searchable Symmetric Encryption". In: *Proceedings 2023 Network and Distributed System Security Symposium* (2023). URL: https://api.semanticscholar.org/CorpusID:257503147.

[10] William W. Cohen. *Enron Email Dataset*. 2015. URL: https://www.cs.cmu.edu/~enron/ (visited on Sept. 16, 2023).

[11] Reza Curtmola et al. "Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions". In: *Proceedings of the 13th ACM Conference on Computer and Communications Security*. CCS '06. Alexandria, Virginia, USA: Association for Computing Machinery, 2006, pp. 79–88. ISBN: 1595935185. DOI: 10.1145/1180405.1180417. URL: https://doi.org/10.1145/1180405.1180417.

[12] Marc Damie, Florian Hahn, and Andreas Peter. "A Highly Accurate Query-Recovery Attack against Searchable Encryption using Non-Indexed Documents". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 143–160. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/damie.

[13] Marc. Damie. *Github*. 2023. URL: https://github.com/MarcTOK/Refined-score-atk-SSE (visited on Aug. 26, 2020).

[14] Apache Software Foundation. *Apache Lucene emails*. 2002. URL: https://lists.apache.org/list?java-user@lucene.apache.org (visited on Sept. 19, 2023).

[15] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. "TWORAM: round-optimal oblivious RAM with applications to searchable encryption". In: *Cryptology ePrint Archive* (2015).

[16] Craig Gentry. “Fully Homomorphic Encryption Using Ideal Lattices”. In: vol. 9. May 2009, pp. 169–178. DOI: 10.1145/1536414.1536440.

[17] Oded Goldreich and Rafail Ostrovsky. “Software Protection and Simulation on Oblivious RAMs”. In: *J. ACM* 43.3 (May 1996), pp. 431–473. ISSN: 0004-5411. DOI: 10.1145/233551.233553. URL: https://doi.org/10.1145/233551.233553.

[18] Zichen Gui, Kenneth G. Paterson, and Sikhar Patranabis. *Rethinking Searchable Symmetric Encryption*. Cryptology ePrint Archive, Paper 2021/879. https://eprint.iacr.org/2021/879. 2021. URL: https://eprint.iacr.org/2021/879.

[19] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. “Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation.” In: *NDSS*. The Internet Society, 2012. URL: http://dblp.uni-trier.de/db/conf/ndss/ndss2012.html#IslamKK12.

[20] Seny Kamara and Tarik Moataz. “Computationally Volume-Hiding Structured Encryption”. In: Apr. 2019, pp. 183–213. ISBN: 978-3-030-17652-5. DOI: 10.1007/978-3-030-17656-3_7.

[21] Seny Kamara, Tarik Moataz, and Olya Ohrimenko. “Structured encryption and leakage suppression”. In: *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I 38*. Springer. 2018, pp. 339–370.

[22] Seny Kamara and Charalampos Papamanthou. “Parallel and dynamic searchable symmetric encryption”. In: *Financial Cryptography and Data Security: 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers 17*. Springer. 2013, pp. 258–274.

[23] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. “Dynamic Searchable Symmetric Encryption”. In: (Oct. 2012). DOI: 10.1145/2382196.2382298.

[24] Tilly Kenyon. *How secure is sensitive data stored in the cloud?* 2022. URL: https://cybermagazine.com/cloud-security/how-secure-is-sensitive-data-stored-in-the-cloud (visited on July 12, 2023).

[25] Steven Lambregts et al. “VAL: Volume and Access Pattern Leakage-Abuse Attack with Leaked Documents”. In: *Computer Security – ESORICS 2022*. Ed. by Vijayalakshmi Atluri et al. Cham: Springer International Publishing, 2022, pp. 653–676. ISBN: 978-3-031-17140-6.

[26] Chang Liu et al. “Search pattern leakage in searchable encryption: Attacks and new construction”. In: *Information Sciences* 265 (2014), pp. 176–188. ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2013.11.021. URL: https://www.sciencedirect.com/science/article/pii/S0020025513008293.

[27] Changshe Ma, Yiping Gu, and Hongfei Li. “Practical Searchable Symmetric Encryption Supporting Conjunctive Queries Without Keyword Pair Result Pattern Leakage”. In: *IEEE Access* 8 (2020), pp. 107510–107526. DOI: 10.1109/ACCESS.2020.3001014.

[28] Chiara Marcolla et al. “Survey on fully homomorphic encryption, theory, and applications”. In: *Proceedings of the IEEE* 110.10 (2022), pp. 1572–1609.

[29] Muhammad Naveed, Seny Kamara, and Charles V. Wright. “Inference Attacks on Property-Preserving Encrypted Databases”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS ’15. Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 644–655. ISBN: 9781450338325. DOI: 10.1145/2810103.2813651. URL: https://doi.org/10.1145/2810103.2813651.

[30] Jianting Ning et al. “LEAP: Leakage-Abuse Attack on Efficiently Deployable, Efficiently Searchable Encryption with Partially Known Dataset”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 2307–2320. ISBN: 9781450384544. DOI: 10.1145/3460120.3484540. URL: https://doi.org/10.1145/3460120.3484540.

[31] Oracle. *The benefits of cloud storage*. 2023. URL: https://www.oracle.com/cloud/storage/what-is-cloud-storage/benefits/ (visited on Oct. 27, 2023).

[32] Simon Oya and Florian Kerschbaum. "IHOP: Improved Statistical Query Recovery against Searchable Symmetric Encryption through Quadratic Optimization". In: *CoRR* abs/2110.04180 (2021). arXiv: 2110.04180. URL: https://arxiv.org/abs/2110.04180.

[33] Omkant Pandey and Yannis Rouselakis. "Property preserving symmetric encryption". In: *Advances in Cryptology–EUROCRYPT 2012: 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings 31*. Springer. 2012, pp. 375–391.

[34] Vasilis Pappas et al. "Blind Seer: A Scalable Private DBMS". In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 359–374. DOI: 10.1109/SP.2014.30.

[35] David Pouliot and Charles V. Wright. "The Shadow Nemesis: Inference Attacks on Efficiently Deployable, Efficiently Searchable Encryption". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 1341–1352. ISBN: 9781450341394. DOI: 10.1145/2976749.2978401. URL: https://doi.org/10.1145/2976749.2978401.

[36] David Shapiro. *Github PlainTextWikipedia*. 2020. URL: https://github.com/daveshap/PlainTextWikipedia (visited on Oct. 19, 2023).

[37] Dawn Xiaoding 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*. 2000, pp. 44–55. DOI: 10.1109/SECPRI.2000.848445.

[38] Ewan Klein Steven Bird Edward Loper. *Natural Language Toolkit*. 2001. URL: https://www.nltk.org/ (visited on Oct. 1, 2023).

[39] *Wikipedia dumps*. 2023. URL: https://dumps.wikimedia.org/simplewiki/ (visited on Oct. 19, 2023).

[40] Mark A. Will and Ryan K.L. Ko. "Chapter 5 - A guide to homomorphic encryption". In: *The Cloud Security Ecosystem*. Ed. by Ryan Ko and Kim-Kwang Raymond Choo. Boston: Syngress, 2015, pp. 101–127. ISBN: 978-0-12-801595-7. DOI: https://doi.org/10.1016/B978-0-12-801595-7.00005-7. URL: https://www.sciencedirect.com/science/article/pii/B9780128015957000057.

[41] Cong Zuo et al. "Dynamic searchable symmetric encryption schemes supporting range queries with forward (and backward) security". In: *Computer Security: 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II 23*. Springer. 2018, pp. 228–246.

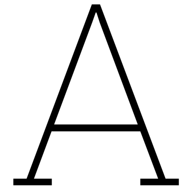
List of Figures

2.1 SSE Setup. Icon credits: client by The Icon Z, Document by Win Ningsih, Lock by Mol Media, cloud server by Kholifah from thenounproject.com 6
2.2 SSE Search. Icon credits: client by The Icon Z, Document by Win Ningsih, Lock by Mol Media, cloud server by Kholifah, Key by Darrin Loeliger, magnifier by Alone forever from thenounproject.com 6
2.3 Simplified Attack Model. Icon credits: client by The Icon Z, Document by Win Ningsih, cloud server by Kholifah from thenounproject.com 7
2.4 SSE Attack Types 7
2.5 Co-occurrence example. "Document" icon by Win Ningsih, from thenounproject.com CC BY 3.0 10
2.6 Search pattern example 11
2.7 Volume pattern example. "Document" icon by Win Ningsih, from thenounproject.com CC BY 3.0 11
2.8 Leakage profile L4 13
2.9 Leakage profile L3 13
2.10 Leakage profile L2 13
2.11 Leakage profile L1 13

5.1 Volscore Attacks Overview 26
5.2 Covolume matrix example 28
5.3 Covolume submatrix for keywords 29
5.4 Covolume submatrix for trapdoors 29
5.5 Computing the score 29
5.6 Covol submatrices expansion 30
5.7 Clustering example 32

6.1 Enron example email 37
6.2 Apache example email 38
6.3 Wikipedia example 39
6.4 Results from Damie et al. [12] $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = 1.2K\ m_{real} = 1K, |Q| = 150,$ RefSpeed = 10 41
6.5 Score comparison reproduction with same parameters. 41
6.6 (Enron) Accuracy comparison between RefScore and VolScore attacks. Fixed parameters are: $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = 1.2K\ m_{real} = 1K, |Q| = 150,$ RefSpeed = 10 = MaxRefSpeed 41
6.7 (Apache) Accuracy comparison between RefScore and VolScore attacks. Fixed parameters are: $|D_{sim}| = 20K, |D_{real}| = 30K, m_{sim} = 1.2K\ m_{real} = 1K, |Q| = 150,$ RefSpeed = 10 = MaxRefSpeed 41
6.8 (Wiki) Accuracy comparison between RefScore and VolScore attacks. Fixed parameters are: $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = 1.2K\ m_{real} = 1K, |Q| = 150,$ RefSpeed = 10 = MaxRefSpeed 42
6.9 (Enron) Low known queries comparison. Fixed parameters are: $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = 1.2K\ m_{real} = 1K, |Q| = 150,$ RefSpeed = 10 = MaxRefSpeed 43
6.10 (Apache) Low known queries comparison. Fixed parameters are: $|D_{sim}| = 20K, |D_{real}| = 30K, m_{sim} = 1.2K\ m_{real} = 1K, |Q| = 150,$ RefSpeed = 10 = MaxRefSpeed 43
6.11 (Wiki) Low known queries comparison. Fixed parameters are: $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = 1.2K\ m_{real} = 1K, |Q| = 150,$ RefSpeed = 10 = MaxRefSpeed 43
6.12 (Enron) RefSpeed comparison. Fixed parameters are: $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = 1.2K\ m_{real} = 1K, |Q| = 150,$ KnownQ = 4, MaxRefSpeed = 10 45

6.13 (Apache) RefSpeed comparison. Fixed parameters are: $|D_{sim}| = 20K, |D_{real}| = 30K, m_{sim} = 1.2K$ $m_{real} = 1K, |Q| = 150,$ KnownQ = 4, MaxRefSpeed = 10 45
6.14 (Wiki) RefSpeed comparison. Fixed parameters are: $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = 1.2K$ $m_{real} = 1K, |Q| = 150,$ KnownQ = 4, MaxRefSpeed = 10 45
6.15 (Enron) RefScore countermeasure comparison. $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = m_{real}, |Q| = 0.15 * m_{real}$, KnownQ = 15, RefSpeed = $0.05 * |Q|$, $n_{pad} = 500$ 47
6.16 (Enron) RefVolScore countermeasure comparison. $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = m_{real}, |Q| = 0.15 * m_{real}$, KnownQ = 15, RefSpeed = $0.05 * |Q|$, $n_{pad} = 500$ 47
6.17 (Apache) RefScore countermeasure comparison. $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = m_{real}, |Q| = 0.15 * m_{real}$, KnownQ = 15, RefSpeed = $0.05 * |Q|$, $n_{pad} = 500$ 47
6.18 (Apache) RefVolScore countermeasure comparison. $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = m_{real}, |Q| = 0.15 * m_{real}$, KnownQ = 15, RefSpeed = $0.05 * |Q|$, $n_{pad} = 500$ 47
6.19 (Wiki) RefScore countermeasure comparison. $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = m_{real}, |Q| = 0.15 * m_{real}$, KnownQ = 15, RefSpeed = $0.05 * |Q|$, $n_{pad} = 500$ 48
6.20 (Wiki) RefVolScore countermeasure comparison. $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = m_{real}, |Q| = 0.15 * m_{real}$, KnownQ = 15, RefSpeed = $0.05 * |Q|$, $n_{pad} = 500$ 48



Apache Lucene dataset

Script to download Apache dataset between 2002 and 2011.

```
#!/bin/bash

# Apache Dataset
mkdir apache_ml
cd apache_ml
for y in {2002..2011}; do
    for m in {01..12}; do
        wget "http://mail-archives.apache.org/mod_mbox/lucene-java-user/${y}${m}.mbox"
    done
done
```