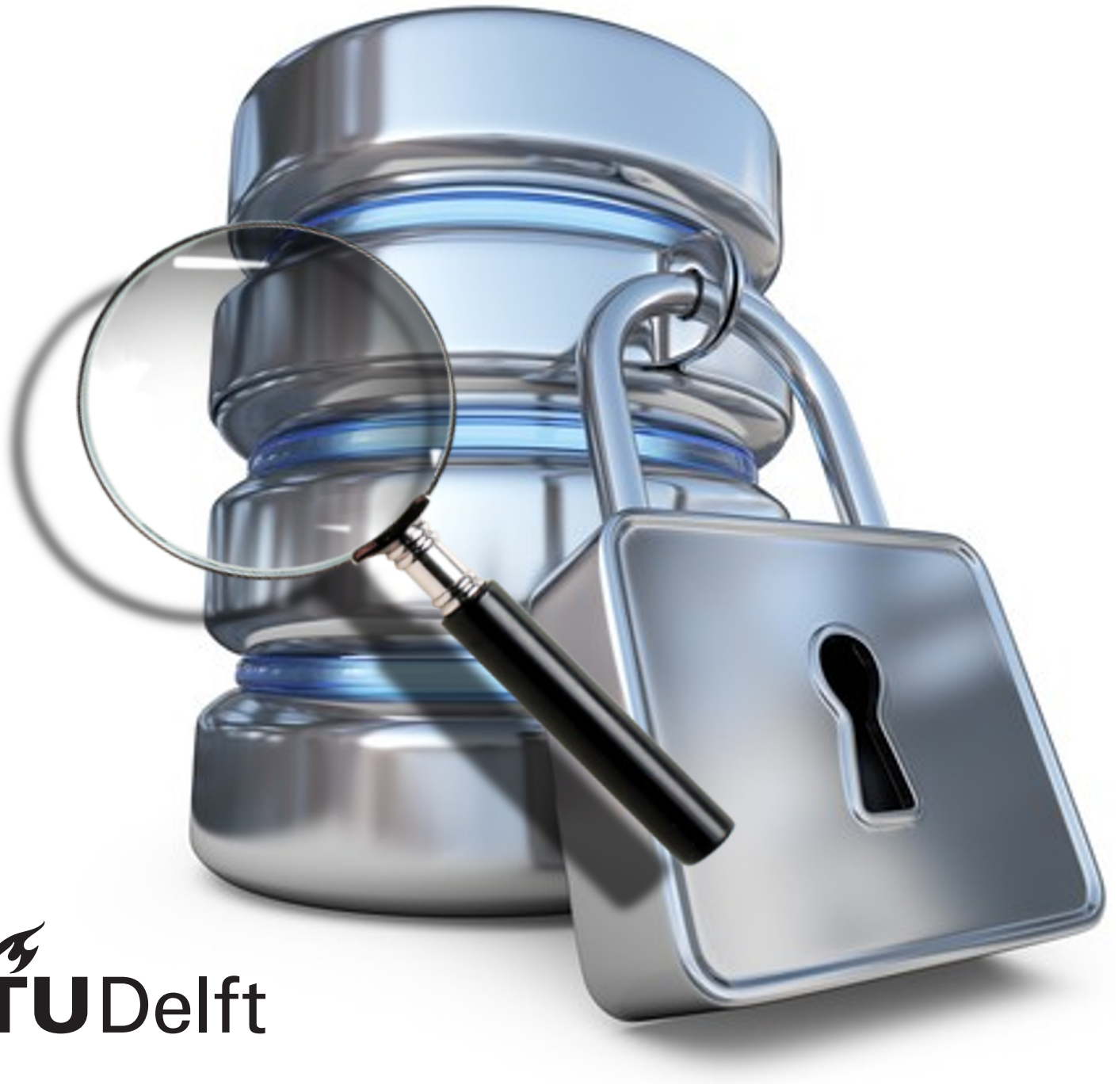
Revisit Attacks on Searchable Symmetric Encryption

Explore More, Reveal More

Steven Lambregts



Delft University of Technology

TU Delft

Revisit Attacks on Searchable Symmetric Encryption

Explore More, Reveal More

by

Steven Lambregts

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday July 4, 2022 at 02:00 PM.

Student number:	4565347	
Project duration:	November 8, 2021 – July 4, 2022	
Thesis committee:	Dr. Z. Erkin,	TU Delft
	Dr. K. Liang,	TU Delft, supervisor
	Dr. S. Roos,	TU Delft

Cover:	Encrypted database by Percona & magnifying glass (Modified)
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

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

TU Delft

Preface

This MSc thesis describes my work developing a new attack on Searchable Symmetric Encryption schemes. This thesis ends my career as a master's student at the TU Delft. Starting my student life at the TU Delft 6 years ago, I could not have thought of the person I have become now. I am very proud of what I have achieved until now and will keep growing even further.

I am pleased to present my work on 'Revisit Attacks on Searchable Symmetric Encryption'. I want to thank my supervisor Dr. Kaitai Liang and PhD. Student Huanhuan Chen for their guidance, advice and critical attitude during my master thesis. I would also like to thank Dr. Z. Erkin and Dr. S. Roos for being part of my thesis committee and taking the time to grade my work.

Steven Lambregts
Delft, June 2022

Abstract

Searchable Symmetric Encryption (SSE) schemes provide secure search over encrypted databases while allowing admitted information leakages. Generally, the leakages can be categorized into `access`, `search`, and `volume pattern`. In most existing Searchable Encryption (SE) schemes, these leakages are caused by practical designs but are considered an acceptable price to achieve high search efficiency. Many attacks on SSE schemes have shown that such leakages could be easily exploited to retrieve the underlying keywords for search queries. Each attack abuses a different leakage pattern and uses different techniques to achieve high query recovery accuracy. An attacker could be passive or active, where an active attacker can inject files in an SSE scheme, while a passive attacker only observes the queried data. Some passive attacks use the number of files returned by a query to create a match with a candidate keyword. Others use the co-occurrence of multiple keywords in the files to match a query with the same occurrence. We continue this research and design a new Volume and Access Pattern Leakage-abuse Attack (VAL-Attack) that exploits both the `access` and `volume patterns`. Our proposed attack only leverages leaked documents and the keywords present in those documents as auxiliary knowledge and can effectively retrieve document and keyword matches from leaked data. Furthermore, the recovery performs with great accuracy and without false positives. We compare VAL-Attack with two recent well-defined attacks on several real-world datasets to highlight the effectiveness of our attack and present the performance under popular countermeasures.

Contents

1 Introduction

People tend to store their personal data at a third-party server, a so-called cloud. Outsourcing the data can be done for storage or security reasons because the cloud service provider has to encrypt the stored files because of General Data Protection Regulation (GDPR). Encrypting the data enhances privacy and gives the owners the feeling that their information is stored safely. However, this encryption relatively restricts the searching ability, as the stored data does not correspond to actual search terms or vocabulary. Song et al. [50] proposed a Searchable Encryption (SE) scheme to preserve the search functionality over outsourced and encrypted data. In this scheme, the keywords of files are encrypted, and when a client wants to query a keyword, it encrypts the keyword to a search token and sends it to the server. The server then searches the data for files with the token corresponding to the query, and afterwards, it returns the matching files to the user. Since the initial SE scheme, many research works have been proposed in the literature, both with symmetrical [12–14, 16, 19, 24, 52] and asymmetrical encryption [1, 9, 10, 32, 60, 62]. Nowadays, SE schemes have been deployed in many real-world applications such as ShadowCrypt [27] and Mimesis Aegis [34].

A Searchable Symmetric Encryption (SSE) scheme is a symmetric encryption scheme, meaning that for encryption of the plaintext and decryption of the ciphertext, the same cryptographic key is used. Within this encryption scheme, a collection of documents is encrypted. Each document within this collection contains some keywords from a keyword space. The encryption key K encrypts a keyword w and generates a search query q that can be used to retrieve associated encrypted files from the server.

A server responding to a query sent by a user with the corresponding data is an example of an interaction between a user and a server. However, an adversary can intercept the query and the response in this example. This interception can happen because the messages are sent over unprotected channels or because the adversary is the cloud service provider. The cloud service provider can store all in and outgoing data, including queries and encrypted data. The adversary aims to match the observed queries to keywords such that he can understand what content is stored on the server without decrypting the data.

The query and server response are considered leakage to the SSE scheme. We consider two main types of leakage patterns: `access and search patterns`. In an SE scheme, the `access pattern` is, given a query q of keyword w, all the documents related to w. The `search pattern` is the frequency a query q is sent within a specific timeframe. Besides these two main types of leakage, we also consider the `volume pattern` as leakage. The `volume pattern` reflects the size of the stored documents on the server in bytes.

The leakage patterns can be divided into four levels, as categorized by Cash et al. [15]. For the attack we created, VAL-Attack, we consider the leakage level of the SSE scheme to be L2 in combination with the `volume pattern`. Leakage level L2 means a fully-revealed occurrence pattern. But more about these leakage levels in Chapter 2.

There are currently a lot of different attacks that work and perform differently. Most of these attacks take leaked plaintext files as auxiliary knowledge. In 2012, Islam et al. [28] presented the foundation for several attacks on SSE schemes in their IKK attack. They show that with enough auxiliary knowledge, one can create a matrix that shows the occurrence of keywords per document tuple, the co-occurrence matrix. The IKK attack uses this co-occurrence matrix for the auxiliary knowledge and the observed server files. One can map queries to the keywords based on the lowest distance with these two matrices. Cash et al. [15] presented an attack where the query can be matched to a particular keyword based on total occurrence in leaked files. These attacks with knowledge about some documents are called passive attacks with pre-knowledge.

The reason why we are so interested in creating new attacks is because of the danger it brings. We can emphasize the importance and effectiveness of countermeasures if an attack does not need many resources to be very successful. One could argue that this has been shown with only a single attack and implementing countermeasures afterwards. However, some attacks require more countermeasures to be mitigated entirely or are only reduced with other remedies. Therefore, it is essential to show different kinds of attacks on SSE schemes.

The $\text{Subgraph}_{\text{VL}}$ attack by Blackstone et al. [6] has high query recovery rates even with a small subset of leaked documents. It matches keywords based on unique document volumes as if it is the response pattern. The LEAP attack [40] combines the earlier techniques, such as co-occurrence and the unique number of occurrences, to match leaked files to server files and match known keywords to queries based on unique occurrences in the matched files. It uses the unique count from the Count attack [15], a co-occurrence matrix from the IKK attack [28] (although they inverted it to a document co-occurrence matrix) and finally, unique patterns to match keywords and files. The only thing they do not match with is the volume pattern.

1.1. Research Question

With the current situation stated above, we aim to address the issue of matching keywords by exploiting both the `access pattern` and the `volume pattern`. The following question arises naturally:

How could we match queries and documents by researching different attacks and improve the current state-of-the-art attack on SSE schemes to capture a high recovery rate against popular defences?

The research question from above can be broken down into two sub-questions. First, we investigate what kind of attacks exist and how they exploit their stated leakage pattern. Then we describe what techniques current attacks use and how they perform. Furthermore, we compare the state-of-the-art attacks with current opportunities such that we can create our attack.

The subquestions then become:

1. What is the main difference in existing attacks on SSE schemes, and what do they abuse?
2. How can we improve the state-of-the-art attacks by exploiting a new branch of leakage or matching technique?

The first sub-question is answered by conducting a literature study and reproducing the researched attack. This way, we could understand what technique they used and how we can compare attacks to each other.

For the second sub-question, we created an extensive overview of the current attacks and described what technique they use and what leakage they abuse. We found an open opportunity and created our attack based on this novelty and our knowledge.

1.2. Contributions

We performed an extensive literature study on many different attacks on SSE schemes. These attacks vary in abusing the `access pattern`, `search pattern` and `volume pattern`. These attacks all have different approaches and techniques, use different leakage information and can match either keywords,

files, or both. In our literature study, we reproduced each investigated attack to thoroughly understand what the attack requires, how it performs and what the output is.

An attack requires some input; this input can be either plaintext files that are leaked from the server, leaked underlying keywords from observed queries, or very similar files to the documents present on the server.

Besides this input, an attack has a specific matching techniques. Some attacks match only queries to keywords, while others match the leaked files to encrypted files on the server. This keyword matching can be done by taking the number of occurrences in the leaked files and matching it to a query with the same occurrence in the server files. Another technique is matching by co-occurrence, i.e. the number of files that contain two keywords. When such a co-occurrence number is unique among the leakage, the keyword can be matched to a query with similar co-occurrence.

The output of the researched attacks can be matched queries with or without matched files. Attacks can have all observed queries matched to a keyword, or only the ones they know for sure are correct.

In order to create a new attack, we created a comprehensive comparison overview where we compare each attack in exploited leakage pattern, techniques used and attack results. Based on this comparison, we developed an attack that explores a new direction and reaches higher accuracy than the existing state-of-the-art attacks.

We design an attack that shows the danger of leaking the `access and volume pattern`. The VAL-Attack follows the SE scheme and matches leaked documents and keywords to server data, and it does so by exploiting both the `pattern`. The results from the attack are all correct, i.e. no false positives. Our attack has an improved matching technique not based on exact matches, compared to the LEAP attack; moreover, we also consider the occurrence in unmatched documents. Finally, we can match more files based on the size of each document, either by direct or indirect equality.

We compare our attack to several others. Table 3.1 compares existing attacks summarizing their leakage pattern, auxiliary data, and information exploitation technique.

We answer the above research question by designing an attack that matches leaked files and keywords. Our attack expands the matching techniques from the LEAP attack [40] and exploits the `volume pattern` to match more documents. The attack improves the LEAP attack by fully exploring the leakage information and combining the uniqueness of document volume to match more files. These matches can then be used to extract keyword matches. All the matches found are correct, as we argue that false positives are not valuable in real-world attacks.

- Besides exploiting the `access pattern`, we also abuse `volume pattern` leakage. We match documents based on a unique combination of volume and number of keywords with both leakage patterns. We can match almost all leaked documents to server documents using this approach.
- We match keywords using their occurrence pattern in matched files.
- Besides matching keywords in matched files, we use all leaked documents for unique keyword occurrence, expanding the keyword matching technique from the LEAP attack. We do this to get the maximum amount of keyword matches from the unique occurrence pattern.

We run our attack against three different datasets to test the performance. The results are outstanding as we match almost all leaked documents and a considerable amount of leaked keywords. Finally, we compare our attack to the existing state-of-the-art LEAP and $\text{Subgraph}_{\text{VL}}$ attacks. Our attack performs great in revealing files and underlying keywords. In particular, it surpasses the LEAP attack, revealing significantly more leaked files and keywords. VAL-Attack recovers almost 98% of the known files and above 93% of the keyword matches available to the attacker once the leakage percentage reaches 5%. When 10% of the Enron database is leaked, which is 3,010 files with 4,962 keywords, we match 2,950 files and 4,909 queries, respectively, corresponding to 98% and 99% of server leakage. VAL-Attack can still compromise encrypted information, e.g., over 90% recovery (with 10% leakage), with volume hiding and other popular countermeasures in the Enron and Lucene datasets. However, we note that our proposed attack is vulnerable to padding and volume hiding hybrid countermeasures.

1.3. Thesis Structure

This thesis is organized in different chapters: Chapter 2 describes the background for SSE schemes, different leakage models and attack techniques. Chapter 3 describes related work to our attack, where we compare the technique, performances and required knowledge of existing attacks. We did this by re-implementing the attacks one by one, to get a good view about the attack. This literature study is necessary to create a theoretical framework and draw conclusions. Chapter 4 describes how we created our attack and what SSE scheme we use. To test the performance of our attack, we run experiments. Each experiment requires some data; we describe which datasets we used and how we prepared them for our experiments in Chapter 6. Chapter 5 describes our novel attack, the techniques and the leakage-abuse methods, with the results of our experiments shown in Chapter 7. We discuss the limitations, the conclusion and some future work in Chapter 8.

Throughout this document, we use a lot of notations. An extensive overview of all notations and their description is put in Table A.1. The attack diagram that compares all the attacks is shown in Appendix C, and the paper we created about our attack is shown in Appendix E.

2

Background

Before researching the different kinds of attacks, we must understand SSE and how an adversary can benefit from observing the query and response. We will a lso describe the techniques used by different attacks, divided into active and passive approaches. This chapter will conclude by describing when an attack is considered robust and accurate.

2.1. Searchable Symmetric Encryption Schemes

Encrypted Search Algorithms (ESA) can be designed using fully homomorphic encryption (FHE) [23], oblivious RAM (ORAM) [25, 38], functional encryption [8], property-preserving encryption (PPE) [2, 4, 7], and SE [17, 19, 50]. All these techniques have different tradeoffs in terms of leakage and efficiency.

In a general SE scheme, a user encrypts her data and uploads the encrypted data to the server, see Figure 2.1 for an example. However, before we can upload to the server, the system has to be set up using the SE scheme. Symmetric encryption is used to encrypt information; it works with a single master key K that can encrypt and decrypt personal information. Hence, the ”symmetric” in the name in SSE. The user only has this key to encrypt and decrypt the files locally. This technique differs from asymmetric encryption, where a pair of public and private keys is used to encrypt and decrypt messages. We consider two types of SSE schemes, static and dynamic.

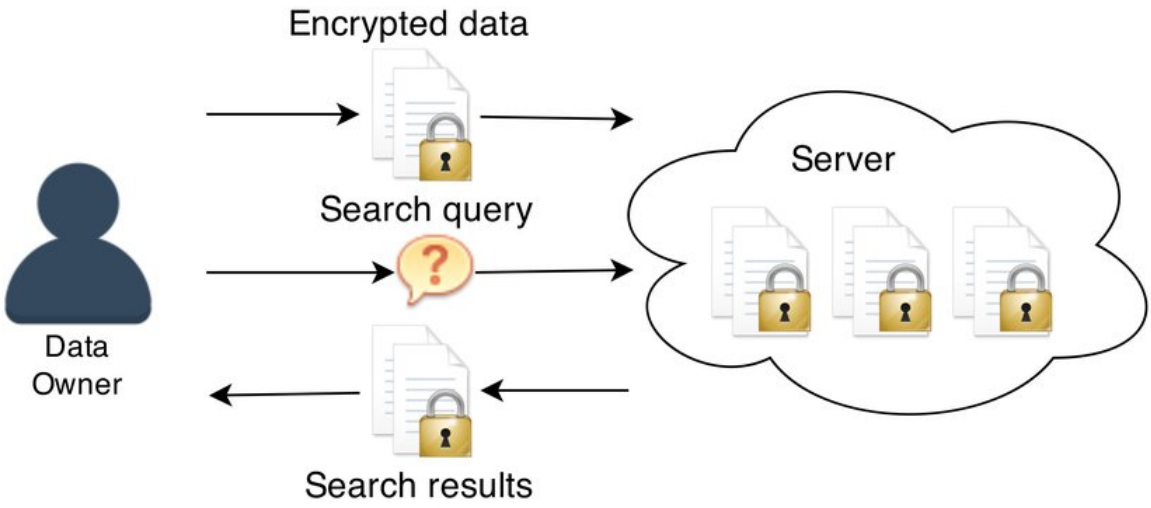


Figure 2.1: An Architecture for a Searchable Encryption Scheme [46]

2.1.1. Static SSE

A static SSE scheme consists of three polynomial-time algorithms: Enc, $\mathrm{QueryGen}$ and Search [19, 21, 35]. Definition 1 shows the algorithms in more detail.

The client runs the algorithm Enc to encrypt the set of plaintext documents (F) and the corresponding keywords with master key K before uploading it to the server. Enc outputs an encrypted database EDB, which is sent to the server. The function $\mathrm{QueryGen}$ is a deterministic algorithm run by the client to create a query q, it takes the master key K and a keyword w and outputs a query q. This query is later sent to the server to retrieve the documents matching keyword w. The last function is Search. Search is a deterministic algorithm that is executed by the server. A query q is sent to the server; the server takes the encrypted database EDB and returns the corresponding identifiers of the files $EDB(q)$. After it has retrieved the file identifiers, the user has to do another interaction with the server to retrieve the actual files. It is impossible to insert or delete documents in a static SSE scheme.

Definition 1 (Static SSE)

- $\mathrm{Enc}(K, F)$: *the encryption algorithm takes the master key K, and a document set $F = \{F_1, \ldots, F_n\}$ as input and outputs the encrypted database $EDB := \{\mathrm{Enc}_K(F_1), \ldots, \mathrm{Enc}_K(F_n)\}$;*
- $\mathrm{QueryGen}(w)$*: the query generation algorithm takes a keyword w as input and outputs a query token q;*
- $\mathrm{Search}(q, EDB)$*: the search algorithm takes a query q and the encrypted database EDB as input and outputs a subset of the encrypted database EDB, whose plaintext contains the keyword corresponding to the query q.*

A static SSE scheme is correct if for all $k \in \mathbb{N}$, for all K output by $\mathrm{KeyGen}(1^k)$*, for all $F \subseteq 2^W$, for all EDB output by $Enc_K(F)$, for all $w \in W$, where W represents a keyword dictionary,*

$$\mathrm{Search}(\mathrm{QueryGen}(w), EDB) = F(w) \wedge Dec_K(Enc_K(F_i)) = F_i, \text{for } 1 \leq i \leq n.$$

2.1.2. Dynamic SSE

A dynamic SSE scheme consists of seven polynomial-time algorithms: Enc, $\mathrm{QueryGen}$, Search, $\mathrm{AddToken}$, $\mathrm{DelToken}$, Add and Del [31, 39]. Here, Enc, $\mathrm{QueryGen}$ and Search are similar to those of static SSE, Definition 1. The detailed algorithms are described in Definition 2. $\mathrm{AddToken}$ is run by the client and requires the master key K and a new document d_{n+1}. Its output is an insertion token t_a and an encrypted document ed_{n+1}. The algorithm $\mathrm{DelToken}$ takes the master key K and a file d and returns a delete token t_d. The client runs this algorithm prior to deleting a file from the server. Add is a deterministic algorithm run by the server and requires an insertion token t_a and the new encrypted document ed_{n+1} and returns an updated encrypted database EDB'. Del is run by the server, is deterministic, requires a delete token t_d, and outputs the updated server document collection EDB'.

Definition 2 (Dynamic SSE)

- $\mathrm{AddToken}(K, d)$: *the add query token algorithm takes the master key K and a new document d as input. It outputs insertion token t_a and encrypted document ed;*
- $\mathrm{DelToken}(K, d)$*: the delete query token algorithm takes the master key K and a document d as input and outputs a delete token t_d;*
- $\mathrm{Add}(ed, t_a)$*: the add file algorithm takes an encrypted file ed and an insertion token t_a. It outputs an updated encrypted database EDB'.*
- $\mathrm{Del}(t_d)$*: the delete file algorithm takes as input a delete token t_d and outputs an updated encrypted database EDB'.*

A dynamic SSE scheme has the same correctness as a static SSE scheme, Definition 1. However, $\mathrm{Search}(q, EDB)$ *should still be correct after* $\mathrm{Add}(ed, t_a)$ *or* $\mathrm{Del}(t_d)$*.*

2.2. Leakage

Leakage is what we define as information that is (unintentionally) shared with the outer world. In our model, the attacker can intercept everything sent from and to the server. Querying documents from the server sounds secure, as all the data sent to and from the server is encrypted. However, every

message sent to the server can be intercepted and stored somewhere. In an SSE scheme, a keyword is encrypted to the same ciphertext every time it is encrypted due to deterministic encryption. So, if an adversary stores a query q and sees q being sent to the server later again, he observes a pattern. The adversary can exploit the server response to query q or the search frequency. This information is called leakage; we consider two main types of leakage, `access pattern` and `search pattern` [59]. Besides these two leakage patterns, we also consider the `volume pattern` as an exploitable leakage-abuse approach [6].

2.2.1. Access Pattern

The `access pattern` shows which files contain keyword w, i.e. the subset $F(w)$ for document set F. Furthermore, since the encrypted data is connected to encrypted keywords, the response for query token q is all the server files with the underlying keyword for q, i.e. for encrypted document set E, the subset $E(q)$.

Cash et al. [15] discuss four levels of `access pattern` leakage. Each level determines how the data is stored on the server and what the adversary can see for a queried keyword. The lower the leakage level, the lower the leakage and the more complicated it is for the adversary to discover what content is stored on the server.

We show an example of each leakage level in Figures 2.2 to 2.5. In these examples, ”dog” represents the keyword, and ”Wavtgpc” represents the query token corresponding to the keyword ”dog”. Documents 1 and 2 are the encrypted data stored in the server. In this example, the blue query tokens mark the information leakage in different leakage levels. With the description of the following paragraphs and the examples in the corresponding figures, it is clear how the leakage is determined in SSE schemes.

L4 shows in which document the query occurs, in which position, and how often it occurs. The adversary can learn the pattern of keyword locations in the text and the number of occurrences in the document set. See Figure 2.2 for an example.

Leakage level L3 shows in which document the query occurs and in which place but not how often it occurs. So the attacker can learn at which position the keyword is stored for the resulting documents but can not distinguish if the keyword occurs more than once. Figure 2.3 shows an example.

In leakage level L2, the server stores a so-called inverted index; it stores only the document identifiers corresponding to a query token. The server response only shows the document identifiers without any information about the location of the keywords. L2 is the type of leakage level implemented in basic SSE schemes. See Figure 2.4 for an example.

In L1, the server encrypts every document id. So, whenever a token is queried, it decrypts the document identifiers and sends the ones that correspond to an existing document. There is no information leakage about non-queried keywords, so the attacker can learn only something if the query is sent to the server. Figure 2.5 shows an example for leakage level L1. Dynamic SSE schemes use this leakage level as their foundation.

Document 1			
80G4gbr	Wavtgpc	TP1l2tf	optdn0n
t2EK8Sp	5LLeuwc	Wavtgpc	FzlwsWh
bZ01Hpf	hBliYbT		
Document 2			
Ba2donz	aSby7AV	Pk9MnzP	KJvrBga
ojtE0fs	Wavtgpc	isxWNus	

”dog” → ”Wavtgpc”

Document 1			
80G4gbr	**Wavtgpc**	TP1l2tf	optdn0n
t2EK8Sp	5LLeuwc	**Wavtgpc**	FzlwsWh
bZ01Hpf	hBliYbT		
Document 2			
Ba2donz	aSby7AV	Pk9MnzP	KJvrBga
ojtE0fs	**Wavtgpc**	isxWNus	

Figure 2.2: Example of leakage level L4

Document 1			
80G4gbr	Wavtgpc	TP1l2tf	optdn0n
t2EK8Sp	5LLeuwc	FzlwsWh	bZ01Hpf
hBliYbT			
Document 2			
Ba2donz	aSby7AV	Pk9MnzP	KJvrBga
ojtE0fs	Wavtgpc	isxWNus	

”dog” → ”Wavtgpc”

Document 1			
80G4gbr	**Wavtgpc**	TP1l2tf	optdn0n
t2EK8Sp	5LLeuwc	FzlwsWh	bZ01Hpf
hBliYbT			
Document 2			
Ba2donz	aSby7AV	Pk9MnzP	KJvrBga
ojtE0fs	**Wavtgpc**	isxWNus	

Figure 2.3: Example of leakage level L3

80G4gbr	D02	D08	D10	D11	D19	D77	D84		
Wavtgpc	D05	D08	D12	D35					
TP1l2tf	D11	D24	D55	D61	D63	D69	D71	D77	D91
FzlwsWh	D18	D35	D40	D59	D84	D85			

”dog” → ”Wavtgpc”

80G4gbr	D02	D08	D10	D11	D19	D77	D84		
Wavtgpc	**D05**	**D08**	**D12**	**D35**					
TP1l2tf	D11	D24	D55	D61	D63	D69	D71	D77	D91
FzlwsWh	D18	D35	D40	D59	D84	D85			

Figure 2.4: Example of leakage level L2

Czl	J57	Eyj	Fg0	SQJ	Kot	vXT	E23	U47
Pid	F17	RN7	hB0	BJI	GGI	wZV	I8H	aHc
tvo	0G0	1YC	Mlz	3dT	J07	Imb	g3L	j6n

”dog” → ”Wavtgpc”

f(”Wavtgpc”) → id

Czl	**D05**	Eyj	Fg0	SQJ	Kot	vXT	E23	U47
Pid	F17	RN7	**D08**	BJI	GGI	**D12**	I8H	aHc
tvo	0G0	1YC	Mlz	3dT	**D35**	Imb	g3L	j6n

Figure 2.5: Example of leakage level L1. The function $f()$ outputs the document identifiers to the query input.

The attacker can intercept a query that a user sends to the server and the response from the server. It then knows which document identifiers correspond to which query. This *query → document identifier* response is what we call the `access pattern`. As discussed earlier, we assume the leakage level is L2 [15] (Figure 2.4), where the attacker does not know the frequency or the position of the queried keywords in the document response. Formally, the `access pattern` leakage is defined as follows [6]:

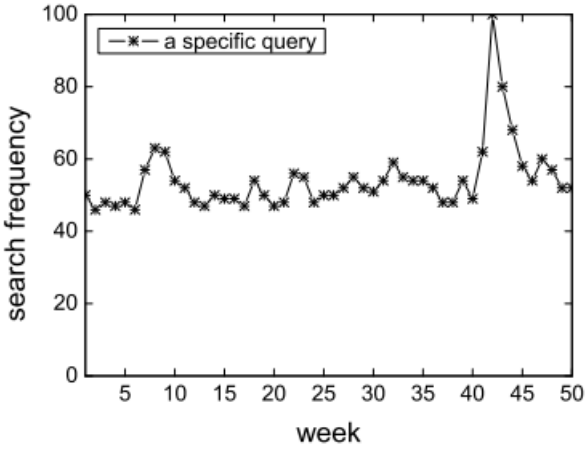


Figure 2.6: Search frequency in a specific time frame [36]

Definition 3 (`access pattern`)
The access pattern function (AP) = $(AP_{k,t})_{k,t\in\mathbb{N}} : F \times W \to [2^n]^t$ *represents all the information leakage in L2 level leakage with secret key* k*, which takes the encrypted data and any* t *queries as input, and outputs a binary column vector* $(D(w_1), ..., D(w_t))$*, where* $D(w_i)$ *is a row binary vector with j-th entry 1 if the underlying data of* D_j *contains the underlying keyword of the query* w_i*, and j-th entry 0 if not.*

2.2.2. Search Pattern

The encryption of a keyword in a specific scheme always results in the same ciphertext due to deterministic encryption and using the same master key. So if the adversary keeps track of the queries sent to the server and notices duplicates, it can determine a particular pattern, which can be abused. The `search pattern` is the information about whether any two queries are generated from the same keyword or not [36]. With this pattern, an adversary can construct a vector of searches per time, which can be used in an attack to match with auxiliary data. Figure 2.6 shows an example from a single query monitored over 50 weeks. The frequency is normalized and displayed over a scale from 0 to 100. The search frequency can also be referred to as occurrence frequency. The formal definition of the `search pattern` is [36]:

Definition 4 (`search pattern`)
The search pattern function (SP) = $(SP_{k,t})_{k,t\in\mathbb{N}} : W^t \times W^t \to \mathbb{N}^2$ *represents all the search pattern leakage with secret key* k*, which takes the encrypted data and any* t *queries as input, and outputs an* $t \times t$ *symmetric binary matrix* $\mathcal{H}$ *such that for* $1 \leq i, j, \leq n, \mathcal{H}[i][j] = 1$ *if* $w_i = w_j$ *else 0.*

2.2.3. Volume Pattern

Each document has a specific volume, i.e. the number of bytes determines how large a document is. The volume of a document stored on the server is equal to the volume of leaked documents and, therefore, can be considered leakage. This `volume pattern` can be exploited by matching unique volumes and is, therefore, considered dangerous leakage [6]. The volume leakage is formally defined as follows [6]:

Definition 5 (`volume pattern`)
The function volume pattern (Vol) = $(Vol_{k,t})_{k,t\in\mathbb{N}} : F \times W^t \to \mathbb{N}^t$ *represents the volume leakage with secret key* k*, which takes the encrypted data and any* t *queries as input, and outputs a column vector* $((|d|_w)_{d\in D(w_1)}, ..., (|d|_w)_{d\in D(w_n)})$*, where* $|d_i|_w$ *is the volume of document* d_i *in bytes.*

For ease of use, we outlined the three leakage patterns in Table 2.1. An eavesdropping attacker cannot identify which documents are returned in response to a query in ORAM schemes. Instead, the attacker can only observe the volume of the document response [56]. In SSE schemes, both the `access pattern` and the `volume pattern` are leaked [6]; therefore, we can abuse them both.

Leakage Pattern	Description
`Access Pattern`	Reveals the identifiers of the documents matching a query.
`search pattern`	Reveals the frequency a query is sent to the server.
`volume pattern`	Reveals the volume of a document matching a query.

Table 2.1: Short description of the three different leakage patterns

2.3. Attacks

Starting this research and eventually creating a new attack, we must understand the current attacks. We need to know how they work, what kind of data they require, and their results. All of these attacks have different experiments to show their accuracy and performance. Therefore, we recreated the attacks with their parameters and compared our results, and if they looked similar, we could think of an improvement or different type of attack.

We can divide attacks into passive and active attacks. Active attacks attempt to insert or delete from the system to understand the content present on the server. Because an attacker will insert files into the encrypted dataset, this can only happen in a dynamic SSE scheme, as a static scheme does not allow document insertion. Passive attacks do not alter the system; they only observe the messages sent to and from the server, trying to understand the content.

2.3.1. Passive Attacks

When the adversary has observed several queries and their responses, he can try to match the queries to keywords. It is possible to use leaked files or similar files for the matching. An adversary could have access to leaked files due to, for example, a security breach. Another possibility is to match observed encrypted files and their query tokens with similar data. An attacker can then check the frequency a specific keyword occurs to match query tokens with available keywords. Among passive attacks, we also consider `search pattern` abusing attacks. These attacks monitor the frequency of specific queries and match them with a similar search pattern.

2.3.1.1. Known Data

The attacker in our model has access to some unencrypted files stored on the server. This access is possible because of a security breach at the setup phase of the scheme, where the adversary can access the leaked files. Another scenario is if a user wants to transfer all of his e-mails from his unencrypted mail storage to an SSE storage server. The server can now access all the original mail files, but new documents will come as new e-mails arrive. Therefore, the server has partial knowledge of the encrypted data present on the server. It is also possible that the attacker has access to some underlying keywords for specific queries. They could also be used to compare frequencies with unmatched query tokens.

When the adversary has access to leaked files, he can match those files to the server files based on a unique number of keywords. He can then try to connect the query tokens to keywords [40]. Another possibility is not to match the files but to match keywords directly on occurrence in the server files and the known files. With a window on the leaked and server occurrence, the attacker can control the gap of unknown files [15].

2.3.1.2. Similar Data

Instead of having access to leaked files, an attacker could also use similar data. If it is known that the storage server contains bank records, one can try to create similar files or use some public dataset as auxiliary knowledge. Then one can try to match based on occurrence in those similar files and the server files with limited distance [20]. The keywords the adversary can match with are contained in similar files. It could happen that the available keyword set is not equal to the actual keyword set, and then the matching result will not result in 100% accuracy. Nevertheless, this attack is effective as it does not require leaked files.

2.3.2. Active Attacks

In a dynamic SSE scheme, it is possible to insert new documents. If the adversary is somehow able to do this as well, he can insert documents with specific keywords, a technique called injecting files. He

can do this by injecting files themselves or sending files to the client that the client then encrypts and stores on the server. Those documents can be used to reveal query keywords after they are injected. In theory, it is possible to retrieve all query matches by only injecting a limited number of files [53, 61].

Some servers have countermeasures to prevent injection attacks, such as limited numbers of keywords per file. However, Zhang et al. [61] showed that these countermeasures could be easily circumvented, and the number of files injected increases only a little. Other countermeasures that a server could implement are noticing if an odd number of documents are added simultaneously or if the documents added are different from the others. If strange behaviour is detected, the server could take appropriate measures against the abusing party inserting the files.

2.4. Evaluation Metrics

The performance of an attack is calculated based on the type of attack. If the attack is considered an active attack, then the attack is only successful if the attack has revealed the underlying keywords for all queries. The target of the injection attacks is to inject as few files as possible such that the server will not notice that the dataset is being altered by a third party while still retrieving all keyword matches.

The correct matches for keywords and queries determine the accuracy of a passive attack. If an attack matches all keywords to a query, then only the correct ones count for the performance, i.e. false positives are not taken into account. An attack that only matches correct queries and thus does not create false positives can be considered more potent in a real-world example. However, the accuracy is still calculated with the correct matches.

A correct match is considered the underlying keyword for a query, so in the case of Figure 2.2, ”dog” is correctly matched with ”Wavtgpc”. If ”Wavtgpc” is matched with ”telephone”, it is incorrectly matched and thus not assessed for the accuracy of the corresponding attack.

3

Related Works

There are many different attacks. For our research, we recreated 15 attacks to understand their techniques, experiments and results. Each attack requires different input; they could use leaked documents, leaked queries, or similar documents to match the queries to keywords. Therefore we created an extensive overview of these requirements, whether the attack was passive or active, either injecting files or observing server messages, what technique the attack used and a small summary. For the extensive diagram, see Appendix C.

In our research, we recreated all the studied attacks individually by starting with a small experiment with only four files such that we could understand what happened in the algorithm on a low level. After the small example, we tried to reproduce the experiments provided by the authors, such that our performance was similar, and we could think of an improvement.

3.1. Attacks

We researched 15 different attacks to eventually create our attack that uses existing leakage patterns and has improved accuracy compared to state-of-the-art attacks. We can divide these attacks into passive and active attacks. For each attack, we will discuss what novelty they brought and how our process of recreating the attack went.

3.1.1. Passive Attacks

Many attacks are proposed, starting with the fundamental attack on SSE schemes, the IKK attack. After the seminal attack, many others have been developed improving the IKK accuracy described by the authors and creating new matching techniques. All the attacks described below are passive attacks, i.e. they only passively observe the communication channel between the attacker and the server to match the queries to keywords based on their auxiliary knowledge.

3.1.1.1. IKK

The first attack we examined was the IKK attack from Islam et al. [28]. It was created in 2012, and it was the foundation for the other attacks on SSE schemes. Islam et al. created a co-occurrence matrix for leaked documents and the observed server files. Especially the co-occurrence matrices are essential techniques in SSE attacks. These matrices could store how often a query q occurs with another query q' in the server files, which could be mapped to a keyword with similar co-occurrence.

They create a co-occurrence matrix for the keywords in the leaked documents and a similar matrix for the query tokens connected to the server data. The idea of the attack is to match a query token to a keyword with equal co-occurrence. They claim that their attack can recover a lot of underlying keywords. The downside of this attack is that they require a lot of leaked documents to match keywords. Nevertheless, it was a great start as a foundation for other SSE attacks.

Since this was the first attack we researched, we had to research much extra information that we did not fully understand before. Examples are co-occurrence matrices and inverted matrices. The authors

describe these techniques but do not give an obvious explanation. Finally, after understanding the technique, they propose their attack with simulated annealing. With the new research environment and the unknown techniques, the IKK attack was challenging to recreate.

After completing the research, we noticed that every attack uses similar approaches for their experiments. The IKK attack explained which dataset was used for the experiment and how they prepared it. However, we had no previous understanding and experience with stemming algorithms and extracting keywords from an e-mail, making it a difficult but necessary and crucial first research.

As this research was the foundation, it was vital to understand what happened and what they did, but shortly after, other attacks showed improved performance over the IKK attack.

3.1.1.2. CGPR

In the Count attack from Cash et al. (CGPR) [15], the attacker has access to some leaked documents and observes some queries and results. The observed server files can assemble a query map with a unique result length, i.e., queries with a unique number of resulting files. These queries can then be mapped to keywords with an equal number of resulting files. This map is an initial result set with queries and their underlying keywords. Next, the adversary can calculate the co-occurrence matrix for the observed queries and the leaked documents. With this data, the adversary can loop over each unmatched keyword and match the correct queries based on equal co-occurrence in the leaked documents. However, this only works if all files are leaked; otherwise, there will never be an exact match. If not all the files are leaked, the adversary can use a window of co-occurrence counts rather than requiring exact equality to match the queries. With the window and the unique count match, the CGPR attack improves the aforementioned IKK attack.

In theory, this attack was obvious: matching keywords based on their unique number of corresponding files. Nevertheless, when the attacker does not access all the leaked files, there is no way to match the queries to their keywords based on the exact number of resulting files. Therefore, the window was a clever addition and did help improve the performance of the attack. However, implementing this window was a more complex part, but eventually, we got it working and recreated the attack.

This technique of matching queries on unique response length is used in future attacks and works very well with the co-occurrence matrices.

3.1.1.3. Shadow Nemesis

The Shadow Nemesis attack uses similar files to match keywords to queries based on co-occurrence frequency [45]. It shows experiments with fully leaked documents and a disjoint data set, i.e. similar files.

The attacker has some observed queries and the resulting encrypted documents. The attacker creates a co-occurrence matrix with the observed data; this matrix stores normalized values of how many files contain both query tokens. Next, the attacker generates graphs where the co-occurrence matrix acts as the adjacency matrix. Using similar data, the attacker constructs a second adjacency matrix. These graph matrices can then be permutated such that they will be most similar to each other, and with this permutation, one can calculate which tag corresponds to which keyword. These graphs are constructed as a Weighted Graph Matching problem, which can be solved by the PATH and the UMEYAMA solving algorithms. The authors created a package that executes these algorithms over the two graphs and outputs the corresponding permutation for each algorithm. The accuracy is not very high because the results for each query keyword in the list are not equal to those of the query list. Because of the incompleteness of the available keywords, this is a challenging approach to attack. However, when the number of keywords used in the attack tags are equal but only shuffled, the attacker can perform the attack successfully.

This attack is one of the first attacks performed with only similar documents, and it performs well in revealing underlying keywords when the attacker has perfect auxiliary knowledge. Nevertheless, as this attack does not use direct matching within the co-occurrence matrix, we researched a new technique for matching keywords. The accuracy is not high due to disjoint datasets, and creating similar files can be demanding in the real world.

3.1.1.4. Subgraph

Blackstone et al. present four types of attacks [6] but highlight only two of them. Namely, the volume analysis and the subgraph algorithm. The volume analysis exploits the total volume pattern; it takes some leaked documents and calculates the volume (the number of bytes of a file) in the observed files. Every observed document volume matches the leaked document's closest volume and matches keywords present to queries. This strategy can result in multiple queries being matched to the same keyword as that could be the closest one to the leaked data.

The second highlighted attack is called the Subgraph attack. This attack exploits the leakage pattern such that it reveals information on each matching document. The proposed attack is divided into the $\text{Subgraph}_{\text{VL}}$ and the $\text{Subgraph}_{\text{ID}}$ attack, where the leakage patterns are the `volume` and `response patterns`, respectively. The attacker has partial knowledge about the data used to match the queries. First, it creates a pattern for the queries, with the document identifier or the number of bytes of the server files, referred to as the leakage. Next, the algorithm creates two bipartite graphs, one for the leaked documents and one for the observed encrypted data. The first set of nodes in the graph for leaked data equals the document identifiers or the volumes (depending on the pattern used). The second set of nodes are all the keywords, and the edges are the keywords present in the auxiliary files. The observed graph is constructed with the leakage pattern from the observed files as the first nodes and the queries as the second set of nodes. The edges in the second graph are the connections between retrieving the encrypted documents.

Next, it will filter for a query the possible keywords such that there is a minimum set of possibilities. This filter first checks whether the keyword occurrence patterns are a subset of the observed query occurrence pattern, and then one knows that the keyword is a possibility for the query. From the first filter's results, one checks whether the length of patterns is a fraction (delta) away from the observed pattern length. Eventually, if there is only one possibility left after filtering, the attacker knows that this query corresponds to the resulting keyword. We wait until the other queries have their correct matching result when more than one possibility is left. The final step is iterative elimination by reducing the number of possibilities, i.e. removing them from the process when a query has only one result. This step increases the number of matches between queries and keywords.

The $\text{Subgraph}_{\text{VL}}$ attack, as described by Blackstone et al., abuses the `volume pattern` to match queries to keywords. Doing this requires partial document knowledge and matches on a unique volume of query responses.

At first, we had to analyze whether the volume is leaked from an SSE scheme, but all known ORAM-, SSE- and PPE-based Encrypted search algorithms leak information, including the volume of a document, as discussed in Section 2.1. This leakage comes in two forms: setup leakage, which is revealed at setup time by the encrypted dataset, and query leakage, which is revealed at query time by the encrypted dataset and the query operation [6].

3.1.1.5. Score

In the Score attack, Damie et al. [20] describe how an attacker reveals queries based on similar server data; in the case of the experiment, they split the original data with no overlap. Then with no known files, they make it possible to extract keywords by checking the score based on the occurrences of keywords in the files. They calculate a matching score using the vectorization of the keywords and queries. The matching score is a logarithmic distance transformation between a keyword and query vector. The attacker only has similar documents, some observed queries, and their responses, from which he can create the query-query co-occurrence matrix. Moreover, he knows a small number of underlying keywords from queries.

The difference between the Shadow Nemesis attack [45] and the Score attack is that the Score attack calculates a score with leaked revealed query tokens to match queries to keywords. In contrast, the Shadow Nemesis attack shifts the columns and rows within the co-occurrence matrix.

3.1.1.6. LEAP

In the LEAP attack from Ning et al. [40], the attacker has access to some leaked files and observes all queries and responses. With this information, he can match the encrypted files to the corresponding

known files based on techniques proposed before. He matches files with the unique number of keywords from the Count attack [15] but uses the co-occurrence matrix differently. Instead of calculating a co-occurrence matrix for queries and keywords, the LEAP attack creates a co-occurrence matrix for the leaked or server files. This co-occurrence matrix states how many keywords or queries are shared among two files. After matching as many files as possible, the attacker can find a unique occurrence of keywords and queries present in the matched files. These unique occurrences indicate a query to keyword match. So, the more files are matched, the more keywords can be matched.

The LEAP attack reworks the idea of co-occurrence matrices of keywords to files, and the idea of matching as many files as possible is very clever. Because using the files in the co-occurrence matrix, the attack requires extensive computational power and time. Nevertheless, the attack was implemented without any difficulties.

3.1.1.7. EHSA

The EHSA attack is a master thesis of a student from the TU Delft. This attack uses the matching technique from the Subgraph$_{VL}$ attack [6], where it matches keywords based on unique overlap in the volumes of leaked documents. After matching the keywords, it uses the already matched keywords as auxiliary knowledge in the matching technique from the Score attack [20]. The second matching part uses similar data to improve the number of query matches. This work is the first attack we researched that combines techniques and improves the accuracy with state-of-the-art attacks.

3.1.1.8. PowerSet

The PowerSet attack from Anzala-Yamajako et al. [3] has some known files and matches these to the encrypted files on the server. The attacker checks whether the number of queries equals the candidate's known files for each server document. By doing this, he creates a very long list of matched documents. After the file matching, he matches the keywords based on occurrence in the matched files. This technique is similar to matching unique co-occurrence in files and a unique number of queries per file. The difference is that the extensive list of correctly matched files acts as a check for each possible candidate. This check verifies if the co-occurrence for each correct file equals the verification with the candidate.

The extensive list of matches requires a lot of computing power, but the results are great because of the extensive verification approach. This tradeoff is not made in the paper, probably because they had better hardware than we did, and therefore their attack ran faster. This technique is strong in theoretical matches, as it verifies every new match with previous matches.

3.1.1.9. DIA

The Document Identification Attack algorithm (DIA) [55] uses the unique count principle from the Count attack [15]. However, it applies this to the documents to match encrypted documents with a unique number of keywords. Then it uses a co-occurrence matrix for files to increase the number of file matches and eventually maps the queries to keywords in the Query Recovery Algorithm with the matched files from the DIA attack.

3.1.1.10. DMA

Wang et al. presented a Document Mapping Algorithm (DMA) [54]. This attack matches keywords based on a unique number of occurrences in files at first. The next step is to match the queries with the intersection of resulting documents for the query and a candidate keyword, i.e. finding unique patterns over unmatched candidates.

While still abusing the access pattern and using a new approach with existing techniques to match keywords, it is still interesting to see scientists think of new ideas. The performance of this attack is only notable from 10% leakage and, therefore, not comparable with the state-of-the-art.

3.1.1.11. Fledged

Like the LEAP attack, Ning et al. [41] use the unique patterns in the occurrence matrix to match queries and files. They use the unique count from the Count attack and take candidates based on similarity. If a file or keyword has been matched, we update the corresponding row or column in the matrix and redo the loop. This way, we create more unique values which we can use to match.

When the attack described by Ning et al. exploited their columns and row matching techniques further, they would have eventually created an extensive matching technique. However, this intermediate step shows an excellent update on possibilities to match with only the access pattern.

3.1.1.12. Search Pattern Attack

The first search pattern exploiting attack that we researched was the attack from Liu et al. [36]. The attacker generates some auxiliary knowledge about the probability of certain keywords in a specific timeframe. He uses Google Trends [51] to do this and requests the normalized number of requests for each keyword in the keywords set. He receives many queries for a specific timeframe, resulting in a frequency vector. He wants to identify the underlying keywords of each of the queries. Then with the obtained results from the query, the attacker can map the auxiliary knowledge to the frequency vector, matching the exact keyword. This attack is proposed with and without the use of keyword categories. If the user is a hospital member, then an attacker will have better matching results if the queries are in the category of medicine. When using the categorical approach, they start by picking a random category and then updating the weights of each category, such that eventually, the correct category is reached, and the correct keywords for each search pattern are retrieved.

In the experiment of this attack, they used actual data from Google Trends and added some noise to make it look like authentic search queries. They do this because there is no actual data representing query search frequencies. However, this technique of adding noise does not have to work in existing applications. Nevertheless, for the idea of the experiment, we do argue that this is a valid assumption.

An assumption made in this attack is that the attacker is aware of all the keywords in the keyword set. However, an adversary would not know which keywords are used because they are already encrypted before reaching the server in a real-life situation. Nevertheless, it is promising to reveal underlying keywords from queries based on their search frequency. The adversary has to observe the channel for a long time to create similar patterns from Google trends. Also, in this attack, they assume that a query is searched somewhat as often as a search term on the Google search engine. These are debatable assumptions, in our opinion. However, the technique and idea are still promising.

While recreating this attack, we had trouble downloading the search pattern for each keyword. Google trends has a rate limit, and once someone has reached this limit, it is problematic to get out of it. In our opinion, downloading the search pattern for 3,000 keywords from several categories took a very long time.

3.1.1.13. Hiding the Access Pattern

The second attack we researched about abusing the search pattern is the attack by Oya et al. [42]. This attack also uses data from Google Trends as auxiliary data and matches the queries to keywords with a likelihood estimator. As shown in their experiment, based on the offset highness, the attacker has lower accuracy due to the data not being equal to the current searches anymore. Based on auxiliary information, i.e. its query knowledge from the google trends, it can determine the actual query using a Maximum Likelihood Estimation function.

This second search pattern abusing attack is very mathematical and therefore considered complicated. It took us a while to understand what the authors did and why they did this. The results we created were similar to the authors' results and did improve the results from the attack from Liu et al. [36]. Therefore this attack can be considered adequate, but we argue that there is still some future work on this topic of matching based on search frequency.

3.1.2. Active Attacks

In an active attack, an attacker can upload files to an SSE system, and the adversary can abuse this by uploading files with specific keywords to match the observed queries. Since the system should not detect the adversary, he aims to limit the number of files uploaded. The performance of an injection attack is therefore expressed as the lowest number of injected files.

3.1.2.1. ZKP

Zhang et al. created the ZKP attack [61], where they uploaded files with keywords divided via a binary search approach. Via this approach and overlap, the attacker can retrieve which token corresponds

to which keyword. He can compare the inserted and retrieved files, where the number of injected files equals the binary logarithm of the total number of files.

The ZKP attack divides the K keywords over $\lceil \log|K| \rceil$ files using a standard non-adaptive version of their binary search algorithm. They divide the keywords bitwise over the files that are being injected.

A countermeasure from the system could be to set a threshold T to prevent an adversary from uploading files with unlimited keywords. This threshold determines the maximum number of keywords per file present on the server. Nevertheless, Zhang et al. still manage to attack the scheme, even with the threshold. The new approach then divides the set of keywords in T sets and injects the files present in each set.

The experiment assumes that the attacker knows all keywords in the keyword set. However, in a real-life scenario, this can be very hard. Of course, they describe that an attack scheme like this one can be very time-consuming. However, in terms of the experiment, this is not very noteworthy.

Because the performance metric for active attacks is recovering all keyword matches, this attack can not easily be combined with passive attacks. The possibility is that when an attack does not create false positives, but only returns correct matches, then one can use auxiliary knowledge and decrease the set of possible keywords by running an initial attack and return 100% accuracy.

3.1.2.2. SetTheory

Injecting files using Finite Set Theory is an attack proposed by Wang et al. [53]. They describe an attack that injects files using a binomial coefficient. Using this approach, the number of injected files reduces drastically compared to the ZKP attack. Wang et al. show experiments with 20,000 keywords and only inject 200 files.

This approach is mathematical and requires some time to implement and understand. However, eventually, as we got the concept, it was implemented as the authors explained it step by step. However, creating an idea like this requires extensive mathematical knowledge and an understanding of set theory. Once the files are injected, we can match the resent queries to a keyword with the matching technique.

Here the attacker still assumes to know all the keywords of the scheme and can inject files on behalf of the user. These assumptions are debatable but are accepted in terms of the experiment.

3.2. Performance

As discussed before, the performance of the attacks differs for an active and a passive attack. The active attacks are only considered satisfactory if they recover all the underlying keywords.

The target of an injection attack is to inject a minimal amount of files. The countermeasures taken by the system to only allow a certain threshold amount of keywords per file do mess with the performance as it increases the difficulty. However, attackers manage to fulfil the requirements and still retrieve all the underlying keywords.

For a passive attack, the target is to match as many files or underlying keywords as possible. Some attacks match all queries to a keyword. Some of them are incorrectly matched and therefore considered false positives, but other attacks only math when they are confident that the match is correct. Since the accuracy is not relevant in a real-life scenario, we would argue that having false positives is ineffective in an attack.

The accuracy of an attack might be higher when all queries are matched. Some attacks make guesses and thus create false positives, and these guesses can be accidentally matched correctly, increasing the performance. For this reason, it is crucial to run the experiment multiple times and take the average of correctly matched results.

Another strategy authors apply, is running their attacks on different (sub)sets of data. Some attacks only run on a subset of a dataset, like the Subgraph attack [6], while others take only a subset of queries, like the Score attack [20], and a third group can take all the queries and files to run their experiment. To compare the attacks properly, we tried to reproduce the experiment of each attack

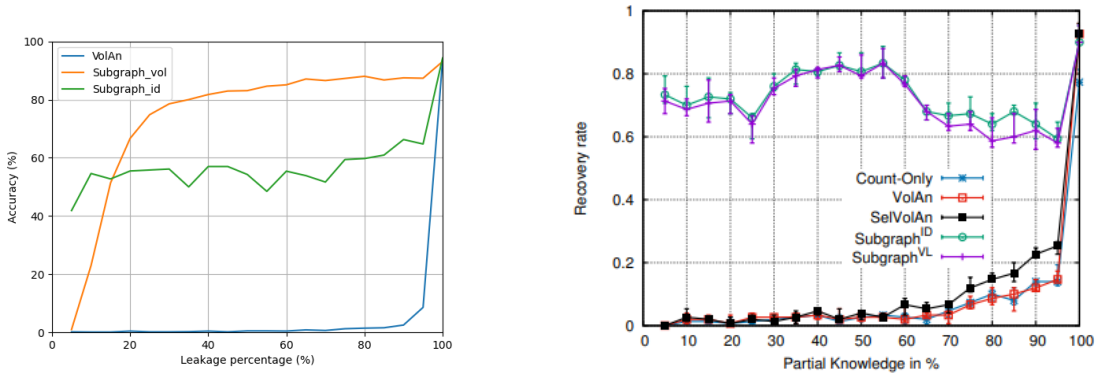
as precise as possible. Afterwards, we set up a default experiment and ran the attacks with these constraints. This way, we could compare the attacks properly. We discuss attack experiments in more detail in Chapter 7.

3.2.1. Subgraph Attack

The Subgraph attack by Blackstone et al. is only run on a subset of files in the total dataset. The dataset consists of 150 users, as shown in Chapter 6, and they only used ten users, 500 keywords, and 150 queries for a specific experiment.

With the setup of 10 users, we tried to reproduce the attack and eventually compare the attack with all users, 5,000 keywords and observing all 5,000 queries. Our comparison with the original experiment parameters can be seen in Figure 3.1. In the figure, one can see a difference in the performance of our reproduction and the claimed performance. The volume attack scores higher than the attack with the document identifiers. The difference in performance could originate from the setup of the attack, such as extracting or stemming the keywords selected.

To compare our newly created attack, we received the actual code from the authors and changed the parameters to compare it with our improved attack.



(a) Our reproduction of the $\text{Subgraph}_{\text{VL}}$ attack

(b) The results from the $\text{Subgraph}_{\text{VL}}$ attack in their work [6]

Figure 3.1: Comparison of our reproduction and the original results from the $\text{Subgraph}_{\text{VL}}$ attack for 10 users (around 19,000 files) and 500 keywords

3.2.2. Score Attack

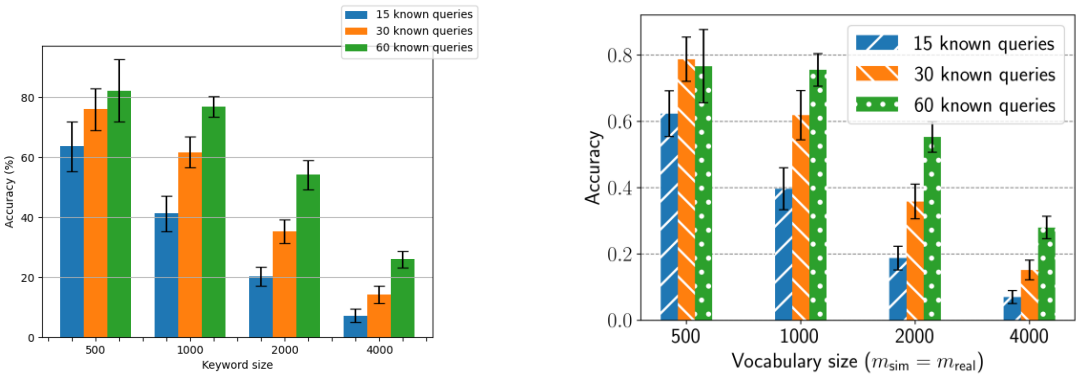
The Score attack from Damie et al. [20] runs on data from all users but only performs well with a lower number of keywords extracted. We tried to reproduce the attack and successfully compared the claimed performance with ours, concluding that there is not much difference between our results and theirs. See Figure 3.2 for the comparison and see that the attack performs less with a more extensive universal keyword set.

3.2.3. Injection Attack

For completeness, we also show our performance of an active attack, the injection attack that uses Finite Set Theory to inject files [53]. We reproduced their experiment using 20,000 keywords and a keyword threshold per file of 100, and 200, see Figure 3.3. Because they did explain almost every detail in their work, we verified that our results are close to theirs.

3.2.4. LEAP Attack

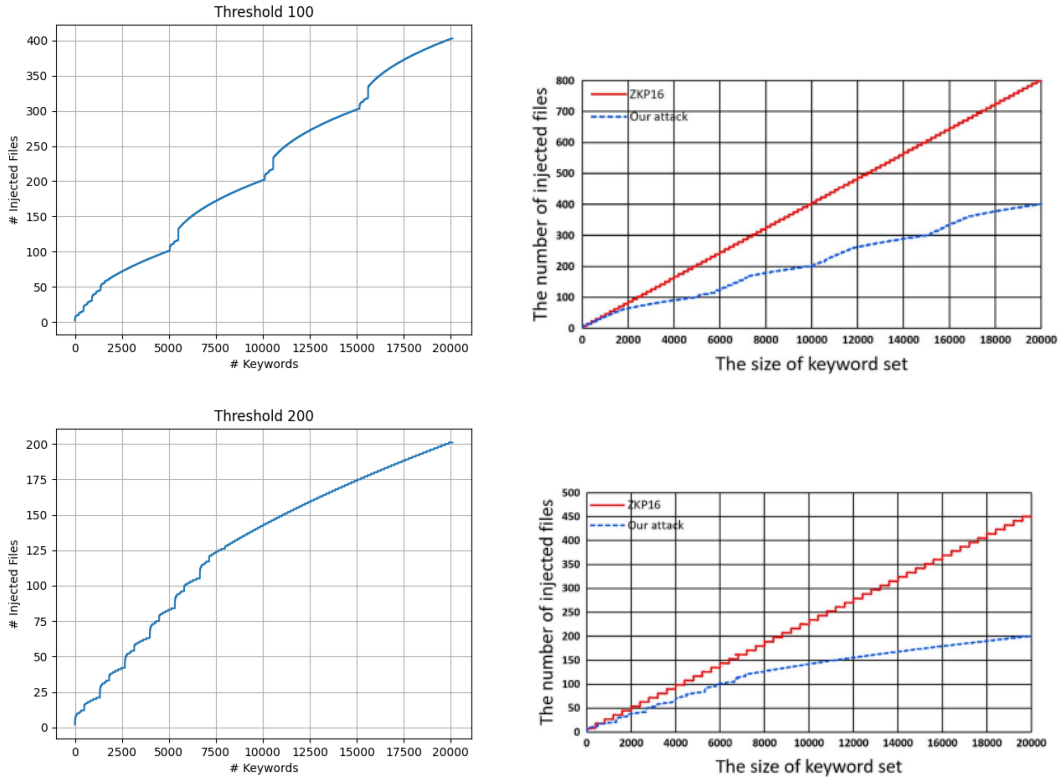
The LEAP attack from Ning et al. [40] does not present a plot with the performance of their work. They offer a table that describes how many keywords are leaked and how many are retrieved with the attack. Our implementation of their attack performs similar, and we have created a plot to see the performance of the attack more organized. They claim that for 10% leakage, they recover 98% of the queries, which can be seen and compared in Figure 3.4.



(a) Our reproduction of the Score attack

(b) The results from the Score attack in their work [20]

Figure 3.2: Comparison of our reproduction and the original results from the Score attack for all 150 users (around 30,000 files) and 500 keywords



(b) Our reproduction of the Injection attack

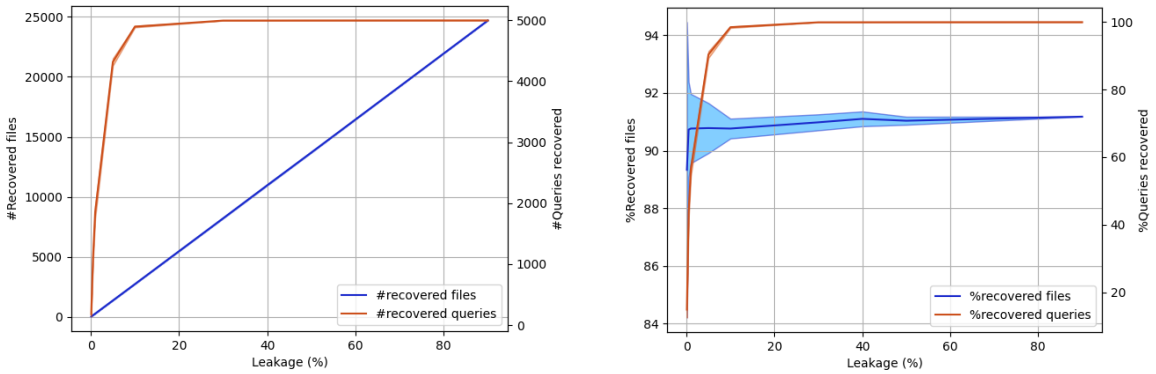
(c) The results from the Injection attack in their work [53]

Figure 3.3: Comparison of our reproduction and the original results from the Injection attack using Finite Set Theory [53]

3.3. Comparison

We compared all the attacks described above on technique, performance, how they exploited their information, what type of information they required, and the attack's style. This comparison can be extensively seen in Appendix C, but we also created a small table, see Table 3.1. This table compares attacks based on leakage patterns, false positives and what technique they use to exploit their information. We have chosen not to implement all attacks described from Section 3.1, because some attacks do not have relevant unique data to show, e.g. the Count, DMA and DIA attack would have an equal row, with only different attack title.

We concluded from the attack diagram that many attacks exploit the `access pattern` and that the



(a) The number of files and queries correctly matched

(b) The percentage of files and queries correctly matched

Figure 3.4: Our reproduction of the LEAP attack with 5,000 keywords and 30,109 files

LEAP attack performs as the state-of-the-art. We also thought of a slight improvement in the matching technique for this attack when reproducing this attack, such that it would match more queries to keywords. When doing more research on the `volume pattern`, we retained that an SSE scheme leaks both patterns.

We captured several matching techniques developed from the previous section, the attack diagram, the table, and the reproduction of the attacks. These techniques differ in varying ways; one can match by taking the number of occurrences in the leaked files and matching it to a query with the same occurrence in the server files. Another technique is matching by co-occurrence, i.e. the number of files that contain two keywords. Based on our literature review, we discovered that the IKK attack, the foundation of the attacks on SSE schemes, is now the attack that requires the most leaked files. Other attacks receive higher accuracy with less leakage; therefore, we consider the IKK as the attack that has the lowest performance among the researched attacks.

In opposition, the LEAP attack requires the least amount of leaked files while still revealing large numbers of queries with the leaked keywords. We consider the LEAP attack to be the current state-of-the-art attack. It only requires 0.1% leakage to recover 11% of the available keywords, and this recovery percentage rises relatively to the leakage percentage, up to almost 100% query recovery for 10% leakage.

We know that current attacks are missing the exploitation of multiple leakage patterns, and we know the state-of-the-art attack. We could create an attack that exploits both the `attack pattern` and the `volume pattern`, and we could exploit it further with the matching techniques of the LEAP attack. Therefore, we wanted to extend the LEAP attack to an attack designed by us with improved keyword matching and exploiting both the `access` and `volume patterns`.

3.4. Investigation Results

We reproduced 15 different attacks that all perform and score differently. When comparing all the techniques from the attacks, we concluded that the `volume pattern` had not been exploited as much as it could be. We combined the leakage of the document size with `access pattern` leakage, and we argue that false positives are not helpful in a real-life scenario and that the matching techniques from the LEAP attack are the current state-of-the-art. With this knowledge we can design a new attack that answers the research question.

3.5. Challenges

During the development of our attack, we faced several challenges. After researching the technique and processes of an attack, we were encouraged to improve the corresponding attack as is. We had to try this improvement to learn how to create an attack with higher performance results than the researched attacks. Therefore, during the whole research, we kept an extensive document describing what each attack abuses and how we could improve it.

Attack	Leakage	Auxiliary data	False positives	Exploited information
IKK [28]	Access pattern	Documents, queries	✓	Co-occurrence
Count [15]	Access pattern	Documents	✓	Co-occurrence, length
Shadow Nemesis [45]	Access pattern	Similar	✓	Co-occurrence
Score [20]	Access pattern	Similar, queries	✓	Co-occurrence
Search14 [36]	Search pattern	Search frequency	✓	Query frequency
$\text{Subgraph}_{\text{VL}}$ [6]	Volume pattern	Documents	✓	Volume, length
LEAP [40]	Access pattern	Documents	✗	Co-occurrence, length
ZKP [61] (active)	Access pattern	All keywords	✗	-

Table 3.1: Comparison on different attacks. *Documents* in the auxiliary data column refers to leaked document knowledge, *queries* refers to leaked underlying keywords for query tokens, and *similar* refers to the use of similar documents instead of leaked documents.

3.5.1. Combining Attacks

We noticed that this improvement development was quite hard at the first attacks we researched. We had no significant knowledge about the matching techniques and did not have a feeling about creating new or better attacks. Eventually, when we researched more attacks, we noticed that many attacks require input, i.e. leaked documents or leaked query matches. We also learned that each attack had an output, i.e. queries with their underlying keywords. After looking at the attacks, we noticed that this output could be used as input for other attacks. By thinking of this combination, we eventually discovered a potential research direction.

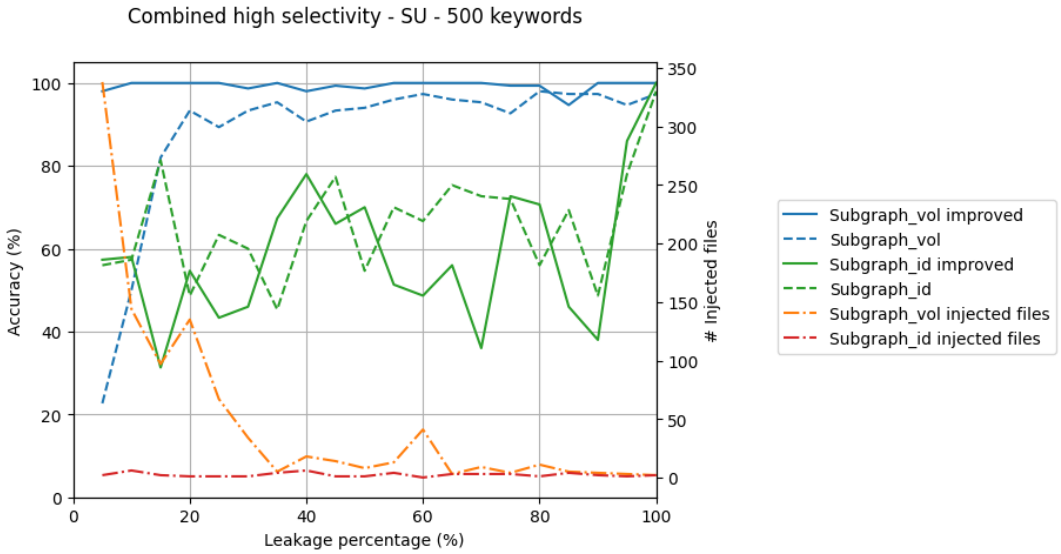


Figure 3.5: Our improvement of the Subgraph Attack, with 500 keywords over a single user. We combined the output of each of the Subgraph attacks with an injection attack. In this figure, the dotted green and blue lines are the reproduction of the original Subgraph algorithm, and the fixed lines above them are improved accuracy. The orange and red dotted-dashed lines indicate the number of injected files.

3.5.2. Injection Attack Combination

The first combination of attacks we created was the combination of the Subgraph Attack [6] with an injection attack, see Figure 3.5. Later we combined the Score attack with an injection attack, as can be seen in Figure D.1. We did this because we noticed that the attacker recovers not all query tokens, and an injection attack benefits from already matched keywords. The first combination attack was developed from this trial, and we had to do more research to improve the newly developed attack. Later, we learned that an injection attack is not always possible due to restrictions of the SSE scheme and can therefore not be combined with any attack just like that.

3.5.3. Similar Attack Combination

Eventually, when we researched more attacks that require similar knowledge, we thought of a solution where an attacker has access to leaked files but uses similar knowledge to improve their attack standpoint. Extending the auxiliary knowledge by creating similar files comparable to the leaked files gives the attacker a significant benefit in recovering queries. We designed an attack on paper and discussed it with the supervisor. He mentioned that another student had already thought of this idea and showed the paper he was developing; this paper concerns the EHSA attack, from Section 3.1.1.7. We concluded that this was a good idea, and we should research more in this direction.

3.5.4. Volume Pattern Combination

Because our previous ideas were a good start. We developed the extensive attack overview from Appendix C. This diagram showed that the `volume pattern` was not exploited to the max extent. Therefore we thought of combining an unexplored leakage pattern with the current state-of-the-art attack. With this in mind, we created a new attack that extends the LEAP attack and combines the leakage patterns of the `volume and access pattern` in VAL-Attack.

4

Methodology

There is no information leaked from the encrypted database, the queries sent, or the database setup in an ideal situation. Unfortunately, such a scheme is not practical in real life as it costs substantial performance overheads [26]. The attacker and the leakage are two different concerns in SSE schemes, and we will discuss them both in the following sections, as they can vary in different aspects.

4.1. Leakage Model

Leakage is what we define as information that is (unintentionally) shared with the outer world. In our model, the adversary can intercept everything sent from and to the server.

An adversary can intercept a query that a user sends to the server and the response from the server. It then knows which document identifiers correspond to which query. This *query* $\rightarrow$ *document identifier* response is what we call the `access pattern`. As discussed in Section 2.2, we assume the leakage level is L2 [15], where the attacker does not know the frequency or the position of the keywords queried in the document response.

The `volume pattern` is leakage that tells the size of the document. This leakage is relevant to all response leaking encryption schemes [11, 13, 17, 19, 30, 31] and ORAM-based SE schemes [38]. We defined the formal leakage definitions in Definitions 3 and 5.

4.2. Attack Model

The attacker in SSE schemes can be a malicious server that stores encrypted data. Since the server is honest-but-curious [6], it will follow the encryption protocol but wants to learn as much as possible. Therefore, the attacker is passive but still eager to learn about the content present on the server. The attacker in our model has access to some leaked plaintext documents stored on the server, keeps track of the `access and volume pattern` and tries to reveal the underlying server data. This access to leaked files is expected in case of a security breach at the setup phase of the scheme, where the adversary can access the revealed files. Another scenario about transferring files has been explained in Section 2.3.1.1. We assume that the attacker has access to all the queries and responses used in the SE scheme. This number of queries is realistic because if one waits long enough, all the queries and results will eventually be sent over the user-server channel.

The passive attacker is less potent than an active attacker, who can upload documents to the server with chosen keywords to match queries to keywords [61]. Furthermore, the attacker has no access to any existing query to keyword matches, only knows the keywords present in the leaked files, and has no access to the encryption or decryption oracle. With this information, the attacker wants to match as many encrypted document identifiers to leaked documents and queries to keywords such that he can understand what content is stored on the server.

We created a technical framework, see Figure 4.1. This framework shows two existing attacks and our new attack. It describes what auxiliary data we use, the leakage patterns, the compared attacks, and

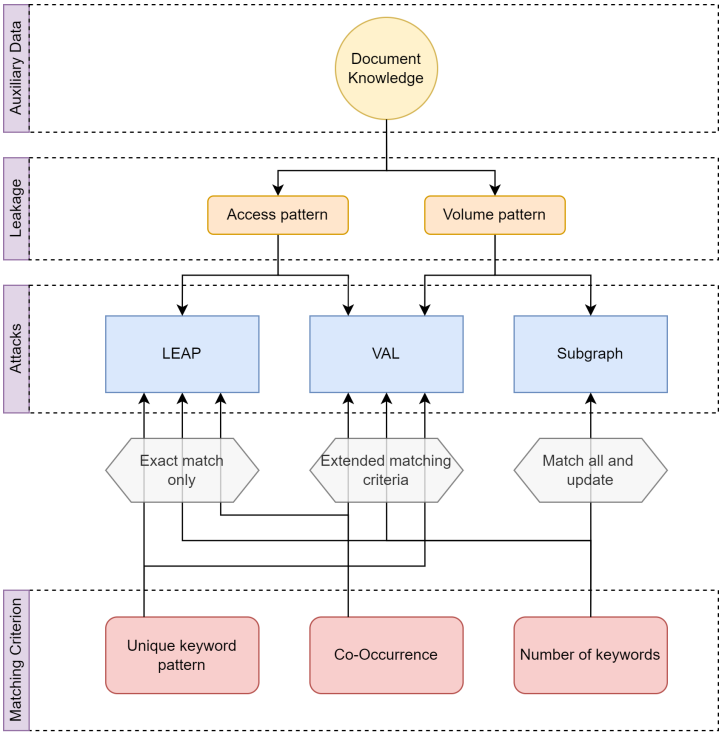


Figure 4.1: Technical Framework of Existing Attacks

the matching criteria. We can see that the LEAP attack matches with exact matches and has strict matching criteria for using a unique keyword pattern, co-occurrence and number of keywords. While our attack, the VAL attack, uses the same matching criteria while it extends the strict match criteria and abuses the volume pattern. The $\text{Subgraph}_{\text{VL}}$ abuses the volume pattern and only matches with the number of keywords. The latter also returns false positives, as it initially matches all keywords and updates them on the run.

5

VAL-Attack

Before explaining VAL-Attack, we first show the different notations used in the attack, such that it is more accessible to what part we are referring to. Next, we will describe the attack's procedure, including all the matching techniques, the leakage information, and the algorithm itself.

5.1. Notation

In the VAL-Attack, we have m' keywords (w) and m queries (q), and n' leaked-documents and n server documents, denoted as d_i and ed_i, respectively; for a single document, similarly for w_i and q_i. Note that w_i may not be the underlying keyword for query q_i, equal for d_i and ed_i. The notations are given in Table 5.1.

F	Plaintext document set, $F = \{d_1, \ldots, d_n\}$
F'	Leaked document set, $F' = \{d_1, \ldots, d_{n'}\}$
E	Server document set, $E = \{ed_1, \ldots, ed_n\}$
W	Keyword universe, $W = \{w_1, \ldots, w_m\}$
W'	Known keywords, $W' = \{w_1, \ldots, w_{m'}\}$
Q	Query set, $Q = \{q_1, \ldots, q_m\}$
A'	$m' \times n'$ matrix of leaked documents
B	$m \times n$ matrix of server documents
M'	$n' \times n'$ co-occurrence matrix of F'
M	$n \times n$ co-occurrence matrix of E
$\|d_i\|$	Number of keywords in document i
$\|d_i\|_w$	Volume (bit size) of document i
C	Set of matched documents
R	Set of matched queries

Table 5.1: Notation of parameters

5.2. The Design

We now define our attack. At a high level, our attack is built from the LEAP attack [40] by elevating the keyword matching metric to increase the number of keyword matches. The approach does not consist of only checking within the matched documents but also keeping track of the occurrence in the

unmatched files. This method results in more recovered keywords for the improvement of LEAP that provides a way to match rows that do not uniquely occur in the matched files.

Then, we expand the attack by exploiting the `volume pattern` since the document size is also leaked from response leaking encryption schemes, Section 4.1. We can expand the comprehensive attack by matching documents based on the `volume pattern`, where each document is labelled with its document volume and number of keywords. VAL-Attack matches using the uniqueness of this label, improving the recovery rate.

We expand the attack by exploiting the `volume pattern` since the document size is also leaked from response leaking encryption schemes, as described in Section 4.1. We can extend the all-out attack by matching documents based on the `volume pattern`.

Our new attack fully explores the leakage information and matches almost all leaked documents. We increase the keyword matches with the maximal file matches to provide excellent performance.

5.2.1. Leaked Knowledge

The server stores all the documents in the scheme. There are a total of n plaintext files denoted as the set $F = \{d_1, \ldots, d_n\}$, with in total m keywords, denoted as the set $W = \{w_1, \ldots, w_m\}$. We assume the attacker can access:

- The total number of leaked files (i.e. plaintext files) is n' with in total m' keywords. Suppose $F' = \{d_1, \ldots, d_{n'}\}$ is the set of documents known to the attacker and $W' = \{w_1, \ldots, w_{m'}\}$ is the corresponding set of keywords that are contained in F'. Note that $n' \leq n$ and $m' \leq m$.
- The set of encrypted files, denoted as, $E = \{ed_1, \ldots, ed_n\}$ and corresponding query tokens, $Q = \{q_1, \ldots, q_m\}$ with underlying keyword set W.
- The volume of each server observed document or leaked file is denoted as $|d_x|_w$ for document d_x or server document ed_x. The number of keywords or tokens is represented as the size of the document $|d_x|$ or $|ed_x|$ for the same documents, respectively.

The attacker can construct an $m' \times n'$ binary matrix A', representing the leaked documents and corresponding keywords. With $A'[d_x][w_y] = 1$ iff. keyword w_y occurs in document d_x. The dot product of A' is denoted as the symmetric $n' \times n'$ matrix M', whose entry is the number of keywords that are contained in both document d_x and document d_y. We give an example of the matrices with known documents in Figure 5.1.

$$
\begin{array}{c} A' \\ w_1 \\ w_2 \\ \vdots \\ w_{m'} \end{array}
\begin{array}{c} \begin{array}{cccc} d_1 & d_2 & \cdots & d_{n'} \end{array} \\ \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \cdots & 1 \end{pmatrix} \end{array}
\qquad
\begin{array}{c} M' \\ d_1 \\ d_2 \\ \vdots \\ d_{n'} \end{array}
\begin{array}{c} \begin{array}{cccc} d_1 & d_2 & \cdots & d_{n'} \end{array} \\ \begin{pmatrix} 5 & 2 & \cdots & 3 \\ 2 & 6 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 3 & 0 & \cdots & 10 \end{pmatrix} \end{array}
$$

Figure 5.1: Example of matrices A' and M'

After observing the server's files and query tokens, the attacker can construct an $m \times n$ binary matrix B, representing the encrypted files and related query tokens. $B[ed_x][q_y] = 1$ iff. query q_y retrieved document ed_x. The dot product of B is denoted as the symmetric $n \times n$ matrix M, whose entry is the number of query tokens that retrieve files ed_x and ed_y from the server. We give an example of the matrices with observed encrypted documents in Figure 5.2.

The attacker also has access to the volume of each document. This volume is observed from the server or the leaked files and is denoted as $|d_x|_w$ for document d_x or server document ed_x. The number of keywords or tokens is represented as the size of the document $|d_x|$ or $|ed_x|$ for the same documents, respectively.

$$
\begin{array}{c} B \\ q_1 \\ q_2 \\ \vdots \\ q_m \end{array}
\begin{array}{c} \begin{array}{cccc} ed_1 & ed_2 & \cdots & ed_n \end{array} \\ \begin{pmatrix} 0 & 1 & \cdots & 1 \\ 0 & 0 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 0 \end{pmatrix} \end{array}
\qquad
\begin{array}{c} M \\ ed_1 \\ ed_2 \\ \vdots \\ ed_n \end{array}
\begin{array}{c} \begin{array}{cccc} ed_1 & ed_2 & \cdots & ed_n \end{array} \\ \begin{pmatrix} 4 & 3 & \cdots & 1 \\ 3 & 9 & \cdots & 2 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 2 & \cdots & 9 \end{pmatrix} \end{array}
$$

Figure 5.2: Example of matrices B and M

5.2.2. Procedure

The basis of the attack is to recursively find row and column mappings between the two created matrices, A' and B, where a row mapping represents the underlying keyword of a query sent to the server, and a column mapping indicates the match between a server document identifier and a leaked plaintext file. Note that each leaked document is still present on the server, meaning that $n' \leq n$ and there is a matching column in B for each column in A'. Similarly to the rows, each known keyword corresponds to a query, so $m' \leq m$ as we could know all the keywords, but we do not know for sure. In theory, there is a correct row mapping for each row in A' to a row in B. The goal of VAL-Attack is to find as much as correct mappings as possible.

We divide the process of finding as many matches as possible into several steps. The first step is to prepare the matrices for the rest of the process. The algorithm then maps columns based on unique column-sum, as they used in the Count attack [15], but instead of using it on keywords, we try to match documents here. Another step is matching documents based on unique volume and the number of keywords or tokens. As this combination can be a unique pattern, we can match many documents in this step. The matrices M and M' are used to match documents based on co-occurrence. Eventually, we can pair keywords on unique occurrences in the matched documents when several documents are matched. This technique is used in the Count attack [15], but we 'simulate' our own 100% knowledge here. With the matched keywords, we can find more documents, as these will give unique rows in matrices A' and B that can be matched. We will introduce these functions in detail in the following sections.

5.2.2.1. Initialization

First, we initialize the algorithm by creating two empty dictionaries, to which we eventually add the correct matches. We create one dictionary for documents and the other for the matched keywords, C (for column) and R (for row). Next, as we want to find unique rows in the matrices A' and B, we must extend matrix A'. It could be possible that not all underlying keywords are known beforehand, in which case $n' < n$, and we have to extend matrix A' to find equal columns. Therefore we create an $m \times n'$ matrix A''_{map} that has the first m' rows equal to matrix A' and the following $m - m'$ rows of all 0s. See Figure 5.3 for an example. The set $\{w_{m'+1}, \ldots, w_m\}$ represents the keywords that do not appear in the leaked document set F'. For ease of use, we copy B to B_{map}.

$$
\begin{array}{c} A''_{map} \\ w_1 \\ w_2 \\ \vdots \\ w_{m'} \\ w_{m'+1} \\ \vdots \\ w_m \end{array}
\begin{array}{c} \begin{array}{cccc} d_1 & d_2 & \cdots & d_{n'} \end{array} \\ \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \cdots & 1 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix} \end{array}
$$

Figure 5.3: Example of matrix A''_{map}

5.2.2.2. Number of Keywords

Now that the number of rows in A''_{map} and B_{map} are equal, we can find unique column sums to match documents. This unique sum indicates that a document has a unique number of keywords and can thus be matched based on this unique factor. Similar to the technique in the Count attack [15], we sum the columns, here representing the keywords in A''_{map} and B_{map}. The unique columns in B_{map} can be matched to columns in A''_{map}, as they have to be unique in A''_{map} as well. If a column$_j$-sum of B_{map} is unique and column$_{j'}$-sum of A''_{map} exists, we can match documents ed_j and $d_{j'}$ because they have the same unique number of keywords.

5.2.2.3. Volume and Keyword Pattern

The next step is matching documents based on volume and keyword pattern. If a server document ed_j has a unique combination of volume $|d_j|_w$ and number of tokens $|ed_j|$ and a document $d_{j'}$ has the same combination, we can match document ed_j to $d_{j'}$.

However, if multiple server documents have the same pattern, we must check for unique columns with the already matched keywords between these files. Initially, we will have no matched keywords, but we will rerun this step later in the process. Figure 5.4 shows a concrete example, and Algorithm 1 describes our method.

Leaked files	···	d_4	d_6	d_8	···	$d_{n'}$
Volume	···	120	120	120	···	120
#Keywords	···	15	15	15	···	18

Server files	···	ed_6	ed_9	ed_{10}	···	ed_n
Volume	···	120	120	120	···	150
#Tokens	···	20	15	15	···	15

(a) Multiple documents with the same pattern of volume and number of keywords/tokens.

$$A''_{CR}\begin{array}{c} \\ w_2 \\ w_3 \\ w_5 \\ \vdots \\ w_t \end{array}\begin{array}{c} \begin{array}{cccc} d_4 & d_6 & d_8 & d_9 \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \vdots & \vdots & \vdots & \\ 1 & 1 & 1 & 0 \end{pmatrix} \end{array} \qquad B_{CR}\begin{array}{c} \\ q_1 \\ q_3 \\ q_{15} \\ \vdots \\ q_t \end{array}\begin{array}{c} \begin{array}{ccc} ed_8 & ed_9 & ed_{10} & ed_{15} \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \vdots & \vdots & \vdots & \\ 1 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

(b) With the already matched keywords, create unique columns to match documents. Here d_6 and ed_8 can be matched, as well as d_9 and ed_{15}.

Figure 5.4: Document matching on volume and number of keywords. Given multiple candidates, match on a unique column with the already matched keywords.

5.2.2.4. Co-occurrence

When having some matched documents, we can use the co-occurrence matrices M and M' to find other document matches. For an unmatched server document ed_x, we can try an unmatched leaked document d_y. If $M_{x,k}$ and $M'_{y,k'}$ are equal for each matched document pair $(ed_k, d_{k'})$ and no other document $d_{y'}$ has the same results, then we have a new document match between ed_x and d_y. The algorithm for this step is shown in Algorithm 2.

5.2.2.5. Keyword Matching

We match keywords using the matched documents. To this end, we create matrices B_c and A''_c by taking the columns of matched documents from matrices B and A''_{map}. Note that these columns will be rearranged to the order of the matched documents, such that column B_{c_j} is equal to column $A''_{c_{j'}}$ for document match $(ed_j, d_{j'})$. Matrices B_c and A''_c are shaped $m \times t$ and $m' \times t$, respectively, for t matched documents. We give the algorithm for this segment in Algorithm 3 and a simple example in Figure 5.5.

Algorithm 1 matchByVolume

Input: $R, A''_{map}\ (m \times n'), B_{map}\ (m \times n)$
$C' \leftarrow \{\}$
patterns $\leftarrow$ {($v_j, |ed_j|$) with volume v_j and #tokens $|ed_j|$ of document ed_j}
for $p \in$ patterns **do**
  enc_docs $\leftarrow$ [ed_j with pattern p]
  if |enc_docs| $== 1$ **then**
    $ed_j \leftarrow$ enc_docs[0]
    $C'[ed_j] \leftarrow d_{j'}$ with pattern p
  else if $|R| > 0$ **then**
    docs $\leftarrow$ [$d_{j'}$ with pattern p]
    $B_{CR} \leftarrow$ enc_docs columns and R rows of B_{map}
    $A''_{CR} \leftarrow$ docs columns and R rows of A''_{map}
    for column$_j \in B_{CR}$ that is unique **do**
      $d_{j'} \leftarrow d_{j'}$ with column$_j \in A''_{CR}$
      $C'[ed_j] \leftarrow d_{j'}$
return C'

Algorithm 2 occurrence

Input: $C, M\ (n \times n), M\ (n' \times n), A''\ (m \times n'), B\ (m \times n)$
$S \leftarrow \{1\}$
$C' \leftarrow C$
while $S \neq \emptyset$ **do**
  $S \leftarrow \emptyset$
  for $j' \in [n']$ where $d_{j'} \notin C$ **do**
    $c'_{j'} \leftarrow$ column$_{j'}$-sum of A''col
    candidates $\leftarrow ed_j$ for $j \in [n]$ where column$_j$-sum of B == $c'_{j'}$
    for $ed_j \in$ candidates **do**
      for $(ed_k, d_{k'}) \in C'$ **do**
        if $M_{j,k} \neq M'_{j',k'}$ **then**
          candidates $\leftarrow$ candidates $\setminus\ ed_j$
    if |candidates| $== 1$ **then**
      S[candidates[0]] $\leftarrow d_{j'}$
      $C' \leftarrow C' \cup S$
return C'

$$B_c = \begin{array}{c} \\ q_1 \\ q_2 \\ q_3 \\ \vdots \\ q_m \end{array} \begin{array}{c} \begin{array}{cccc} ed_3 & ed_2 & \cdots & ed_t \end{array} \\ \begin{pmatrix} 1 & 0 & \cdots & 1 \\ 1 & 1 & \cdots & 0 \\ 1 & 0 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \cdots & 0 \end{pmatrix} \end{array} \qquad A''_c = \begin{array}{c} \\ w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_m \end{array} \begin{array}{c} \begin{array}{cccc} d_1 & d_2 & \cdots & d_t \end{array} \\ \begin{pmatrix} 1 & 0 & \cdots & 1 \\ 1 & 1 & \cdots & 0 \\ 0 & 0 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 0 \end{pmatrix} \end{array}$$

Figure 5.5: Example of matrices A''_c and B_c

A row in the matrices indicates in which documents a query or keyword appears. If a row$_i$ in B_c is unique, row$_i$ is also unique in B_{map}, similar to A''_c and A''_{map}. Hence, for row$_i$ in B_c, that is unique, and if there is an equal row$_j$ in A''_c, we can conclude that the underlying keyword of q_i is w_j.

Nevertheless, if row$_i$ is not unique in B_c, we can still try to match the keyword to a query. A keyword can occur more often in the unmatched documents than their query candidates; thus, they will not be

Algorithm 3 matchKeywords

Input: $C, A''_{map}\ (m \times n'), B_{map}\ (m \times n)$

$R \leftarrow \{\}$
$B_c \leftarrow C$ columns of B_{map}
$A''_c \leftarrow C$ columns of A''_{map}

for row$_i \in B_c$ **do**
  if row$_i$ is unique in B_c **then**
    if row$_{i'} \in A''_c ==$ row$_i$ **then**
      $R[q_i] \leftarrow w_{i'}$
  else ▷ Match based on occurrence in (server) files
    docs $\leftarrow [i' \in A''_c$ where $A''_c[i'] ==$ row$_i]$
    e_docs $\leftarrow [j \in B_c$ where $B_c[j] ==$ row$_i]$

    $B_x \leftarrow$ sum of rows in B_{map}[e_docs], sort descending
    $A_x \leftarrow$ sum of rows in A''_{map}[docs], sort descending

    if $A_x[1] < A_x[0] > B_x[1]$ **then**
      $i_x \leftarrow$ index of $B_x[0] \in$ e_docs
      $j_x \leftarrow$ index of $A_x[0] \in$ docs
      $R[q_{i_x}] \leftarrow w_{j_x}$
return R

valid candidates. We create a list B_x with for each similar row$_i$ in B_c the sum of row$_i$ in B; similar for list A''_x, with row$_i$ in A''_c and the sum of row$_i$ in A''_{map}. Next, if the highest value of A''_x, which is A''_{x_j}, is higher than the second-highest value of A''_x and B_x, referred to as $A''_{x_{j'}}$ and $B_{x_{i'}}$, respectively, we can conclude that keyword w_j corresponds to the highest value of B_x, i.e. B_{x_j}, which means that w_j matches with q_j. We put an example in Figure 5.6.

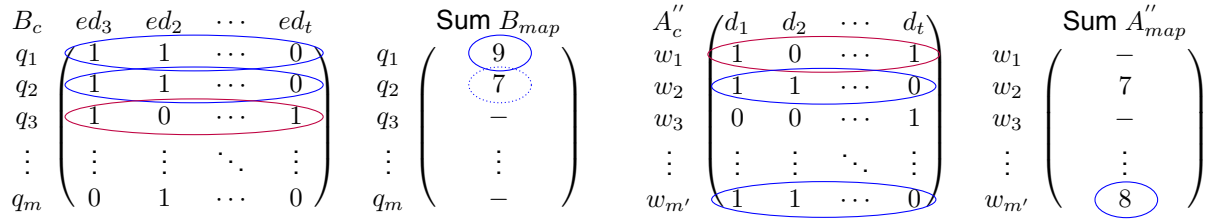


Figure 5.6: Example of matching keywords in matched documents. Query q_3 has a unique row and therefore matches with keyword w_1. Queries q_1, q_2 and keywords w_2, $w_{m'}$ have the same row. However, keyword $w_{m'}$ occurs more often in A''_{map} than w_2 and query q_2 in B_{map}. Therefore q_1 matches with $w_{m'}$.

5.2.2.6. Keyword Order in Documents

We aim to find more documents based on unique columns given the query and keyword mappings. First, we create matrices B_r and A''_r with the rows from the matched keywords in R. B_r and A''_r are submatrices of B_{map} and A''_{map}, respectively, with rearranged row order. B_r and A''_r are shaped $t \times n$ and $t \times n'$, respectively, for t matched files. Note that we show an example in Figure 5.7.

If any column$_j$ of B_r is unique and there exists an equal column$_{j'}$ in A''_r, we know that ed_j is a match with $d_{j'}$.

The next step is to set the rows of the matched keywords to 0 in B_{map} and A''_{map}. Then, similar to before, we use the technique from the Count attack [15]; we sum the updated columns in A''_{map} and

$$
\begin{array}{c} B_r \\ q_3 \\ q_5 \\ q_2 \\ \vdots \\ q_t \end{array}
\begin{array}{c} \begin{array}{cccc} ed_1 & ed_2 & \cdots & ed_n \end{array} \\ \begin{pmatrix} 0 & 0 & \cdots & 1 \\ 1 & 1 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 0 \end{pmatrix} \end{array}
\qquad
\begin{array}{c} A_r^{''} \\ w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_t \end{array}
\begin{array}{c} \begin{array}{cccc} d_1 & d_2 & \cdots & d_{n'} \end{array} \\ \begin{pmatrix} 1 & 0 & \cdots & 1 \\ 0 & 0 & \cdots & 1 \\ 1 & 0 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 0 \end{pmatrix} \end{array}
$$

Figure 5.7: Example of matrices $A_r^{''}$ and B_r

B_{map} and try to match the unique columns in B_{map} to columns in $A_{map}^{''}$. If a column$_j$-sum of B_{map} is unique and an equal column$_{j'}$-sum in $A_{map}^{''}$ exists, we can match document ed_j and $d_{j'}$.

The complete algorithm of our VAL-Attack is in Algorithm 4.

5.2.3. Countermeasure Discussion

Many countermeasures have been proposed to mitigate leakage-abuse attacks [15, 18, 28, 48, 58]. The main approaches are padding and obfuscation.

The IKK attack [28] and the Count attack [15] discussed a padding countermeasure, where they proposed a technique to add fake document identifiers to a query response. These false positives could then later be removed by the user. This technique is also called *Hiding the Access Pattern* [33]. We provided a small example in Figure 5.8. Obfuscation is a different technique, of which we provided an example in Figure 5.10. The LEAP attack [40] crucially relies on the number of keywords per document, and if the scheme adds fake query tokens to documents on the server, they will not be able to match with their known documents. However, they also proposed a technique that describes a modified attack that is better resistant to padding. This technique, also used in the Count attack [15], uses a window to match keywords. However, this will give false positives and thus reduce the performance of the attack.

Before padding

"Wavtgpc" ← | d_1 | d_2 | d_3 |
|---|---|---|
| d_4 | d_5 | d_6 |
| d_7 | d_8 | d_9 |

After padding

"Wavtgpc" ← | d_1 | d_2 | d_3 |
|---|---|---|
| d_4 | d_5 | d_6 |
| d_7 | d_8 | d_9 |

Figure 5.8: Example of padding the `access pattern`. Here we padded the query result to a multiple of 5, by adding 2 false positives.

Before volume padding

File_{08} ← This text is an example and represents a file with the purpose of describing volume padding.

After volume padding

File_{08} ← This text is an example and represents a file with the purpose of describing volume padding.

Figure 5.9: Example of padding the `volume pattern`. Here we padded the file, by adding "empty" characters, such that all files do have the same volume.

The Subgraph$_{VL}$ attack [6] depends on the volume of each document. Volume-hiding techniques from Kamara et al. [29] reduce the attack's performance, but it is unclear if they completely mitigate the attack. Figure 5.9 shows an example of volume padding.

A padding technique that will make all documents of the same size, i.e. adding padding characters, will reduce the uniqueness in matching based on the volume of a document. If the padding technique can be extended such that false positives are added to the `access pattern`, we have no unique factor in matching documents based on the number of keywords per file. Therefore, a combination of the two may decrease the performance of the VAL-Attack.

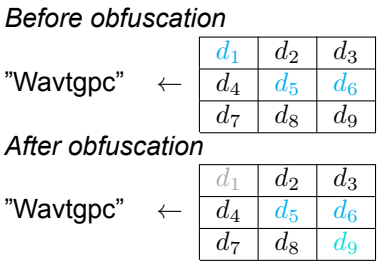


Figure 5.10: Example of obfuscating the `access pattern`. Here we 'randomly' added a false positive (d_9) and false negative (d_1).

5.3. Pseudocode of the VAL-Attack

Algorithm 4 VAL-Attack

Input: A' $(m' \times n')$, B $(m \times n)$, M' $(n' \times n')$, M $(n \times n)$

$C = R \leftarrow \{\}$ ▷ Initialization
$B_{map} \leftarrow B$
$A''_{map} \leftarrow A'$ where rows extended with 0's (m x n')

vector$_A$ = vector$_B \leftarrow [\,]$ ▷ Match documents with unique #keywords
for $j \in [n]$ **do**
    vector$_B[j] \leftarrow$ sum of column B_{map_j}
for $j' \in [n']$ **do**
    vector$_A[j'] \leftarrow$ sum of column $A''_{map_{j'}}$
for vector$_{B_j} \in$ vector$_B$ that is unique **do**
    if vector$_{A_{j'}} ==$ vector$_{B_j}$ **then**
        $C[ed_j] \leftarrow d_{j'}$

$S \leftarrow \text{matchByVolume}(R, A''_{map}, B_{map})$ ▷ Match documents with unique volume
$C \leftarrow C \cup S$

$S \leftarrow \text{occurrence}(C, M, M', A''_{map}, B_{map})$ ▷ Match docs with co-occurrence
$C \leftarrow C \cup S$

$S \leftarrow \text{matchByVolume}(R, A''_{map}, B_{map})$
$C \leftarrow C \cup S$

while R or C is increasing **do**
    $Z \leftarrow \text{matchKeywords}(C, A''_{map}, B_{map})$ ▷ Match keywords in matched docs
    $R \leftarrow R \cup Z$

    $B_r \leftarrow$ R rows of B_{map} ▷ Match documents with unique keyword order
    $A''_r \leftarrow$ R rows of A_{map}
    for column$_j \in B_r$ that is unique **do**
        if column$_{j'} \in A''_r ==$ column$_j$ **then**
            $C[ed_j] \leftarrow d_{j'}$

    $S \leftarrow \text{matchByVolume}(R, A''_{map}, B_{map})$
    $C \leftarrow C \cup S$

    row $B_{map_j} \leftarrow 0$ if $q_j \in R$ ▷ Match documents with unique #keywords
    row $A''_{map_{j'}} \leftarrow 0$ if $k_{j'} \in R$
    for $j \in [n]$ where $ed_j \notin C$ **do**
        vector$_B[j] \leftarrow$ sum of column B_{map_j}
    for $j' \in [n']$ where $d_{j'} \notin C$ **do**
        vector$_A[j'] \leftarrow$ sum of column $A''_{map_{j'}}$
    for vector$_{B_j} \in$ vector$_B$ that is unique and $ed_j \notin C$ **do**
        if vector$_{A'_j} ==$ vector$_{B_j}$ and $d_{j'} \notin C$ **then**
            $C[ed_j] \leftarrow d_{j'}$

    $S \leftarrow \text{occurrence}(C, M, M', A''_{map}, B_{map})$ ▷ Match docs with co-occurrence
    $C \leftarrow C \cup S$
return R, C

6

Datasets

Running experiments is necessary to compare attacks with each other and show how effective an attack is. Since SSE schemes store private data, such as e-mails, running experiments with potential data is essential. The IKK attack [28] used the Enron dataset [57] and argued that a vital characteristic of this dataset is that each document is an authentic e-mail sent by a person. One of the motivations for searching over encrypted data is to store e-mails on a third-party server in an encrypted format and search the e-mails from time to time. Therefore, it is appropriate to run experiments on a real e-mail dataset like that of Enron.

The IKK attack used the Enron dataset, consisting of e-mails from the Enron corporation, which were sent between 2000 and 2002. This dataset has already been extensively used in various studies, and almost all subsequent attacks have used it to show their attack performance and compare it with existing attacks. A second dataset proposed is from the Apache mailing list. As a third dataset, we used some pages from Wikipedia; we did this to verify if our attack also works on other storage data.

6.1. Enron

In the Enron dataset, there are 150 folders of all different users. These users all have folders indicating what type of e-mail a particular document is. In most experiments, they use all e-mails from the *_sent_mail* folder consisting of 30,109 e-mails. Therefore, we also used this folder in our experiment, as we argue that over 30,000 files represent an excellent purpose for an attack on an SSE scheme, Figure 6.1 provides an example of an e-mail of this dataset.

6.2. Lucene

The Count attack [15] introduced another dataset. They used a subset of the Apache mailing list archives dataset [22], besides the Enron dataset, to run their experiments. This dataset is online available and can be downloaded to a large extent. We used the ”java-user” mailing list from the Lucene project for 2001-2011, with multiple e-mails per file, resulting in around 50,000 e-mails, Figure 6.2 provides an example of an e-mail of this dataset, please not the characteristic unsubscribe text at the bottom of this e-mail and the extensive e-mail header.

6.3. Wikipedia

Wikipedia consists of a vast number of pages with all kinds of information, and they offer users free copies of all of its content. We extracted plaintext documents from Wikipedia in April 2022 using a simple wiki dump[1] and used the tool from David Shapiro [49] to extract plaintext data, resulting in 204,737 files like in Figure 6.3. The attack we created requires matrices of size $n \times n$; therefore, we limited the number of Wikipedia files to 50,000. This way, it is also more comparable to the previous two datasets.

[1] `https://dumps.wikimedia.org/simplewiki/20220401/simplewiki-20220401-pages-meta-current.xml.bz2`

```
Message-ID: <25851542.1075855686544.JavaMail.evans@thyme>
Date: Wed, 29 Nov 2000 08:22:00 -0800 (PST)
From: phillip.allen@enron.com
To: stagecoachmama@hotmail.com
Subject:
Mime-Version: 1.0
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit
X-From: Phillip K Allen
X-To: stagecoachmama@hotmail.com
X-cc:
X-bcc:
X-Folder: \Phillip_Allen_Dec2000\Notes Folders\'sent mail
X-Origin: Allen-P
X-FileName: pallen.nsf

Lucy,

Here is a rentroll for this week. The one you sent for 11/24 looked good.
It seems like most people are paying on time. Did you rent an efficiency to
the elderly woman on a fixed income? Go ahead a use your judgement on the
rent prices for the vacant units. If you need to lower the rent by $10 or
$20 to get things full, go ahead.

I will be out of the office on Thursday. I will talk to you on Friday.

Phillip
```

Figure 6.1: Example of E-mail from the Enron Dataset [57]

6.4. Preparing the Dataset

We must extract keywords from the files to use e-mail datasets as input for an SSE scheme. Additionally to using the dataset, the authors of the IKK attack also proposed a technique of using the e-mails as their corpus and how they connected keywords to the files.

For the raw e-mail files, the first few lines contain only metadata about the e-mail. Since these lines are not part of the original e-mail and will probably not be used to search, these lines are stripped from the e-mail documents. Next, the content of the e-mails is cleared from stopwords (Figure 6.4), and the rest of the words are stemmed using a stemming algorithm. The most frequent keywords are used in each experiment, and stemming is performed to find the root of each keyword and enhance the searching process. An example of a stemmed e-mail can be seen in Figure 6.5.

These stemmed keywords are then connected to the document and used as search terms. These words and the file are encrypted. A subset is leaked and copied in plaintext to the user's auxiliary data from these files. (A random subset of) these encrypted keywords and queries are used as the observed queries, with which the server response is extracted as well. Then the attacks can start matching queries to keywords and return their results.

From MAILER-DAEMON Wed Jan 9 03:27:30 2002
Return-Path:
<lucene-user-return-678-qmlist-jakarta-archive-lucene-user=jakarta.apache.org@jakarta.apache.org>
Delivered-To: apmail-jakarta-lucene-user-archive@apache.org
Received: (qmail 33513 invoked from network); 9 Jan 2002 03:27:30 -0000
Received: from unknown (HELO nagoya.betaversion.org) (192.18.49.131)
by daedalus.apache.org with SMTP; 9 Jan 2002 03:27:30 -0000
Received: (qmail 6413 invoked by uid 97); 9 Jan 2002 03:27:30 -0000
Delivered-To: qmlist-jakarta-archive-lucene-user@jakarta.apache.org
Received: (qmail 6397 invoked by uid 97); 9 Jan 2002 03:27:29 -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 6386 invoked from network); 9 Jan 2002 03:27:29 -0000
From: carlson@bookandhammer.com
Errors-To: <carlson@bookandhammer.com>
Date: Tue, 8 Jan 2002 19:27:28 -0800
Subject: Lucene FAQ down
Content-Type: text/plain; charset=US-ASCII; format=flowed
Mime-Version: 1.0 (Apple Message framework v480)
To: ”Lucene Users List” <lucene-user@jakarta.apache.org>
Content-Transfer-Encoding: 7bit
In-Reply-To: <20020109024656.50878.qmail@web12702.mail.yahoo.com>
Message-Id: <CE99F46D-04B0-11D6-BB1A-0050E4C0B243@bookandhammer.com>
X-Mailer: Apple Mail (2.480)
X-Spam-Rating: daedalus.apache.org 1.6.2 0/1000/N
X-Spam-Rating: daedalus.apache.org 1.6.2 0/1000/N

It looks like the lucene FAQ cgi is down at www.lucene.com

I clicked a blank submit and it went for a few minutes.

–Peter

–
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: Example of E-mail from the Lucene Dataset [22]

{
”id”: ”100014”,
”text”: ”Flanimals is a children’s and adults’ book written by comedian Ricky Gervais.
The book was illustrated by Rob Steen. It has 35 different characters described as species
of animal which form an imaginary world.
== Other websites ==
* Official site
* Pictures from the book in the BBC website
* Flanimals on Ricky Gervais’s site
* Flanimals on Rob Steen’s site
* Faber and Faber - UK publisher of all the ’Flanimals’ books
* Flanimals on MySpace Category:2004 books Category:Children’s books”,
”title”: ”Flanimals”
}

Figure 6.3: Example of data file from the Wikipedia Dataset [49]

i	me	my	myself	we	our	ours	ourselves	you
your	yours	yourself	yourselves	he	him	his	himself	she
her	hers	herself	it	its	itself	they	them	their
theirs	themselves	what	which	who	whom	this	that	these
those	am	is	are	was	were	be	been	being
have	has	had	having	do	does	did	doing	a
an	the	and	but	if	or	because	as	until
while	of	at	by	for	with	about	against	between
into	through	during	before	after	above	below	to	from
up	down	in	out	on	off	over	under	again
further	then	once	here	there	when	where	why	how
all	any	both	each	few	more	most	other	some
such	no	nor	not	only	own	same	so	than
too	very	s	t	can	will	just	don	should
now								

Figure 6.4: NLTK English Stopwords

The attached contract is ready for signature. Please print 2 documents and have Atmos execute both and return same to my attention. I will return an original for their records after ENA has signed. Or if you prefer, please provide me with the name / phone # / address of your customer and I will Fed X the Agreement.

attach contract signatur pleas print 2 document have execut both same will origin ena sign prefer provid name custom agreement

Figure 6.5: An example of a plaintext e-mail from the Enron dataset on top and the 20 most frequent stemmed keywords on the bottom.

7

Experiments

We set up the experiments to run the proposed attack to evaluate the performance. Furthermore, we compare the file and query recovery of the VAL-Attack with the results from the LEAP [40] and Subgraph$_{VL}$ attack [6]. We notice that the LEAP attack is not resistant to the test countermeasures. Blackstone et al. [6] argue for their Subgraph$_{VL}$ attack that it is not clear whether volume-hiding constructions may mitigate the attack altogether. From this perspective, we discuss the performance of VAL-Attack against countermeasures in Section 7.3. It would be an interesting problem to test the countermeasures on the LEAP and Subgraph$_{VL}$ attacks, but that is orthogonal to the focus of this work.

We use Python 3.9 to implement our experiments and run them on different machines with different computing power to increase running speed.

7.1. Setup

We used the Enron dataset [57] to run our comparison experiments. We leveraged the *_sent_mail* folder from each of the 150 users from this dataset, resulting in 30,109 e-mails from the Enron corporation. The second dataset we used is the Lucene mailing list [22]; we specifically chose the "java-user" mailing list from the Lucene project for 2002-2011. This dataset contains 50,667 documents. Finally, we did the tests on a collection of Wikipedia articles. We extracted plaintext documents from Wikipedia in April 2022 using a simple wiki dump[1] and used the tool from David Shapiro [49] to extract plaintext data, resulting in 204,737 files. The proposed attack requires matrices of size $n \times n$; therefore, we limited the number of Wikipedia files to 50,000. We used Python 3.9 to implement the experiments and run them on machines with different computing powers to improve running speed.

To properly leverage those data from the datasets for the experiments, we first extracted the information of the Enron and Lucene e-mail content. The title's keywords, the recipients' names, or other information in the e-mail header were not used for queries. NLTK corpus [5] in Python is used to get a list of English vocabulary and stopwords. We removed the stopwords with that tool and stemmed the remaining words using Porter Stemmer [44]. We further selected the most frequent keywords to build the keyword set for each document. For each dataset, we extracted 5,000 words as the keyword set W. Within the Lucene e-mails, we removed the unsubscribe signature because it appears in every e-mail.

The server files (n) and keywords (m) are all files from the dataset and 5,000 keywords, respectively. The leakage percentage determines the number of files (m') known to the user. The attacker only knows the keywords (n') leaked with these known documents. The server files and queries construct a matrix B of size $m \times n$; while the matrix A' of size $m' \times n'$ is constructed with the leaked files. We took the dot product for both matrices and created the matrices M and M', respectively.

[1] `https://dumps.wikimedia.org/simplewiki/20220401/simplewiki-20220401-pages-meta-current.xml.bz2`

Note that the source code to simulate the attack and obtain our results is available here: `https://github.com/StevenL98/VAL-Attack`.

Because our attack does not create false positives, the accuracy of the retrieved files and keywords is always 100%. Therefore, we calculated the percentage of files and keywords retrieved from the total leaked files and keywords. Each experiment is run 20 times to calculate an average over the simulations. We chosen 0.1%, 0.5%, 1%, 5%, 10%, 30% as leakage percentages. The lower percentages are chosen to compare with the results from the LEAP attack [40], and the maximum of 30% is chosen because of the stagnation in query recovery.

7.2. Results

The results tested with the different datasets are given in Figures 7.1a and 7.1b, which show the number and percentage of files and keywords recovered by our attack. The solid line is the average recovery in those plots, and the shades are the error rate over the 20 runs.

We can see that the VAL-Attack recovers almost 98% of the known files and above 93% of the keywords available to the attacker once the leakage percentage reaches 5%. These percentages are based on the leaked documents. When 10% of the Enron database is leaked, which is 3,010 files with 4,962 keywords, we can match 2,950 files and 4,909 queries, corresponding to 98% and 99%, respectively. The Lucene dataset is more extensive than Enron, and therefore we have more files available for each leakage percentage. One may see that we can recover around 99% of the leaked files and a rising number of queries, starting from 40% of the available keyword set. The Wikipedia dataset does not consist of e-mails but rather lengthy article texts. We reveal fewer files than the e-mail datasets, but we recover just below 90% of the leaked files, and from 1% leakage, we recover more available keywords than the other datasets. This difference is probably because of the number of keywords per file since the most frequent keywords are chosen.

With the technique we proposed, one can match leaked documents to server documents for almost all leaked documents. Next, the algorithm will compute the underlying keywords to the queries. It is up to the attacker to allow false positives and improve the number of (possible) correctly matched keywords, but we decided not to include it.

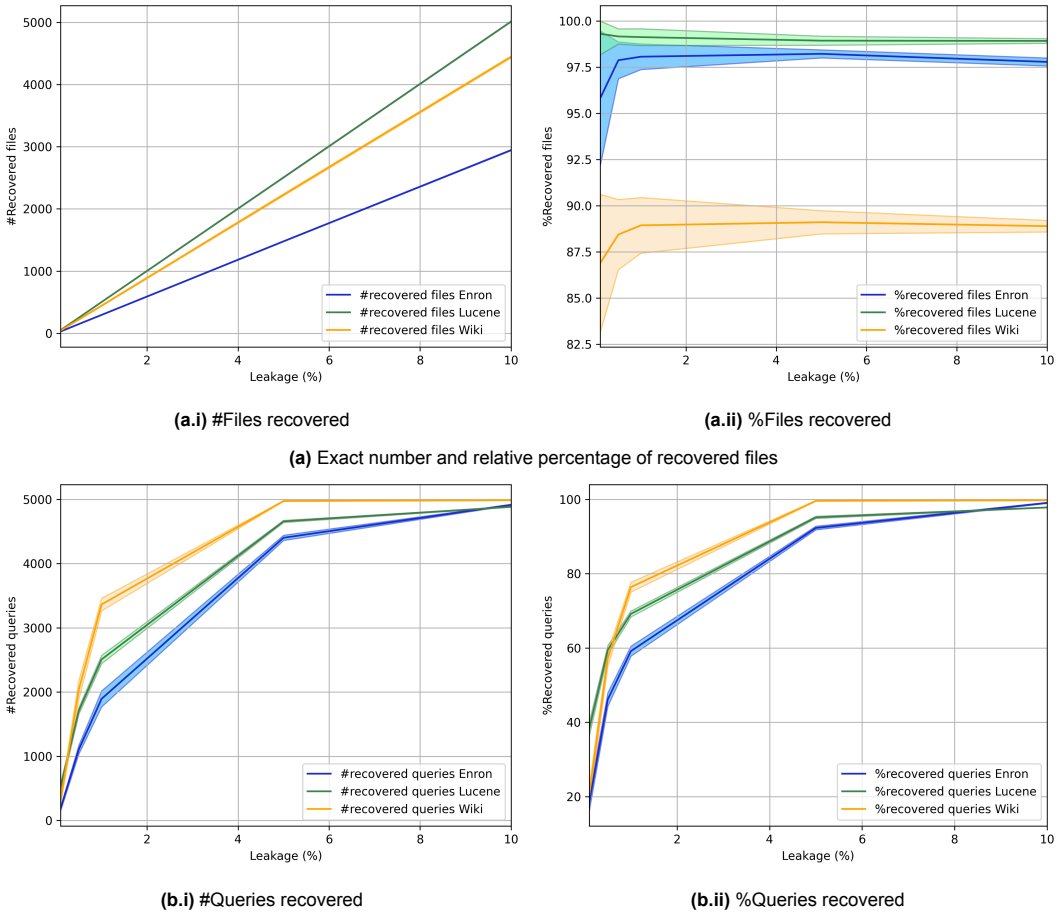
On the left side of the figure showing the results of the recovered files, i.e. Figure 7.1a.i, we see that we revealed the most files for the Lucene dataset, and the Wikipedia dataset comes second. On the right side of the figure, i.e. Figure 7.1a.ii, we see that the blue line, the result line for the Enron dataset, has moved above the Wikipedia dataset. This transition is because there are more files in the Wikipedia dataset compared to the Enron dataset. Regardless, percentage-wise, we recovered more of the available files in the Enron dataset. For example, for 5% leakage in the Enron dataset there are 1,505 plaintext files available to the attacker, of which we recovered around 1,479, resulting in 98% recovery. While with the same leakage percentage and the Wikipedia dataset, the attacker has access to 2,500 files, of which he recovered around 2,236, equal to 89% files of the available files recovered. So, while we recover more files in the Wikipedia dataset, we leave more files unmatched, resulting in a lower recovery percentage. These results also explain the increasing line in Figure 7.1a.i and the stagnated line in Figure 7.1a.ii.

7.2.1. Comparison

We compare the performance of VAL-Attack to two attacks with the Enron dataset. One is the LEAP attack [40] (which is our cornerstone), while the other is the $\text{Subgraph}_{\text{VL}}$ attack [6] (as they use the volume pattern as leakage). We divide the comparison into two parts: the first is for recovering files, and the second is for queries recovery.

As shown in Figure 7.2, we recover more files than the LEAP attack, and the gap in files recovered expands as the leakage percentage increases, see Figure 7.2a.i. The difference in the percentage of files recovered is stable, as VAL-Attack recovers about eight percentage points more files than the LEAP attack, see Figure 7.2a.ii.

The comparison outcome for recovered queries can be seen in Figure 7.2b. We can see that the recovered queries do not show a significant difference with the LEAP attack as that attack performs out-



(a.i) #Files recovered **(a.ii)** %Files recovered

(a) Exact number and relative percentage of recovered files

(b.i) #Queries recovered **(b.ii)** %Queries recovered

(b) Exact number and relative percentage of recovered queries

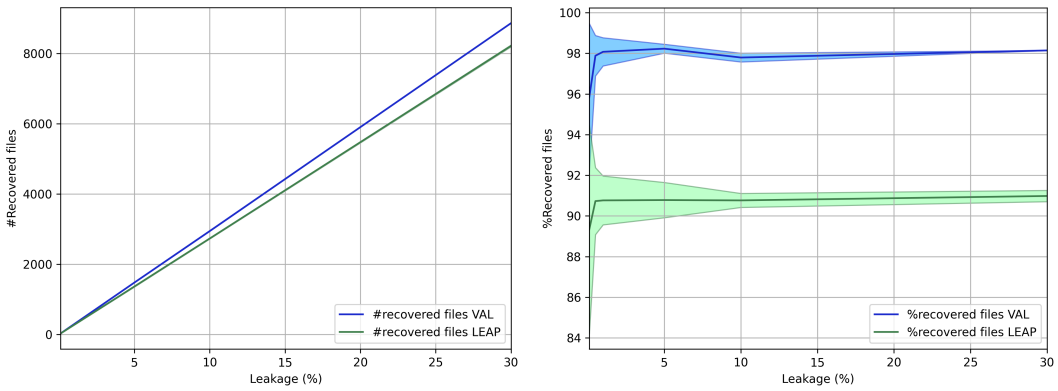
Figure 7.1: Results for VAL-Attack, with the actual number and the percentage of recovered files and queries for different leakage percentages.

standingly in query recovery. The most significant difference is around 5% leakage, where VAL-Attack retrieves around 100 queries more than the LEAP attack, which could influence a real-world application. Compared to the Subgraph$_{VL}$, we see in Figure 7.2b.ii that the combination of the `access pattern` and the `volume pattern` is a considerable improvement; we reveal about 60 percentage points more of the available queries.

With more leakage percentage, our attack and the LEAP attack recover more files, according to Figure 7.2a.i. This incremental recovery is related to the leakage percentage; the number of available files also rises if the leakage grows. On the right side, in Figure 7.2a.ii, these lines are almost straight, indicating that we recover a fixed percentage of the available files per varying leakage percentage. This stationary percentage line shows that we recover more files as the leakage increases because of the rising leakage percentage.

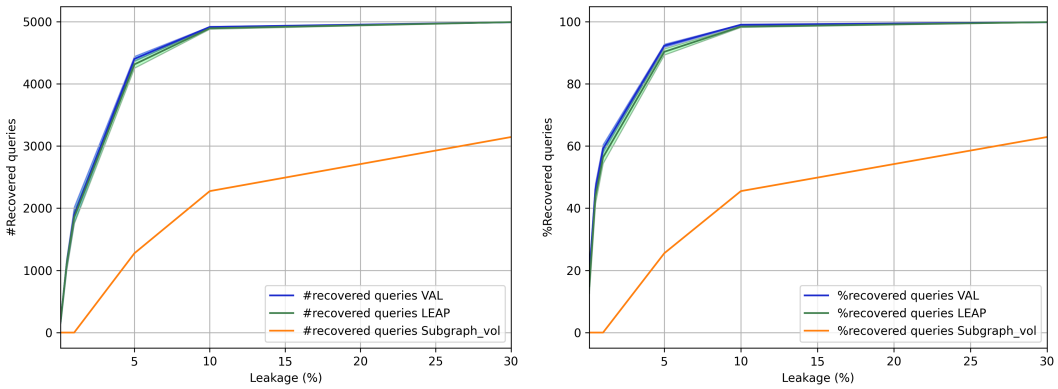
7.3. Performance under Countermeasures

As discussed in Section 5.2.3, there are several options for countermeasures against attacks on SE schemes. Moreover, since our attack exploits both the `access and volume pattern`, countermeasures must mitigate both leakage patterns. The former can be mitigated by padding the server result, while the latter may be handled using volume-hiding techniques. However, these approaches may come with impractical side effects. Padding the server response requires more work on the client-side to filter out the false positives. This padding can cause storage and reading problems because the user has to wait for the program to filter out the correct results. The volume-hiding technique [29] may easily yield significant storage overhead and could therefore not be practical in reality. Luckily, Patel



(a.i) #Files recovered **(a.ii)** %Files recovered

(a) Comparison with LEAP attack [40] based on the number and percentages of files recovered



(b.i) #Queries recovered **(b.ii)** %Queries recovered

(b) Comparison with LEAP attack [40] and $\text{Subgraph}_{\text{VL}}$ attack [6] based on the number and percentages of queries recovered

Figure 7.2: Comparison of VAL-Attack

et al. [43] illustrated how to reduce this side effect whilst mitigating the attack.

It is possible to mitigate our attack theoretically by using a combination of padding and volume hiding techniques. We tested the VAL-Attack's performance with padding, volume hiding and further a combination, but we did not examine by obfuscation due to hardware limitations.

We padded the server data using the technique described by Cash et al. [15]. Each query returned a multiplication of 500 server files, so if the original query returned 600 files, the server now returned 1,000. Padding is done by adding documents to the server response that do not contain the underlying keyword. These documents can later be filtered by the client but will obscure the client's observation; Figure 5.8 shows a small example of how padding the `access pattern` works. We took the naïve approach from Kamara et al. [29] for volume hiding, where we padded each document to the same volume, see Figure 5.9. By adding empty bytes to a document, it will grow in size, and all files will eventually have the same size that can not be distinguished from the actual size if done correctly.

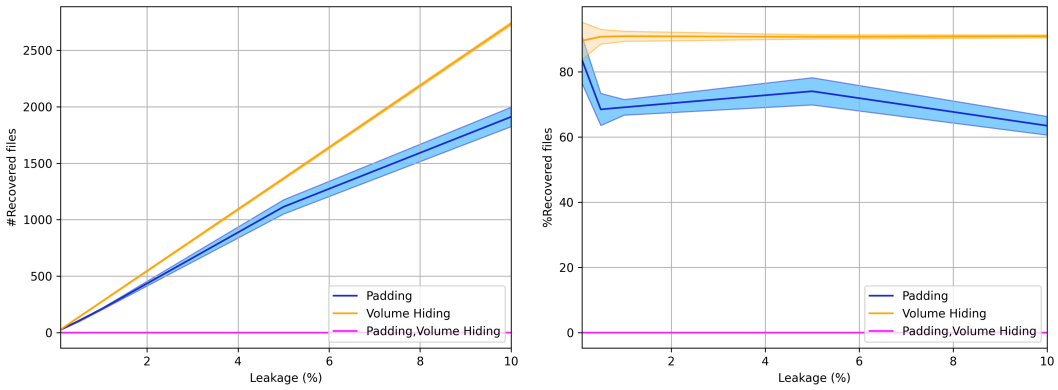
We ran the countermeasure experiments on the Enron and the Lucene dataset. We did not perform the test on the Wikipedia dataset, but we can predict that the countermeasures may affect the attack performance. We predict that a single countermeasure will not entirely reduce the attack effectiveness, but a combination may do.

Because of the exploitation of the two leakage patterns, we see in Figure 7.3 that our attack can still recover files and underlying keywords against only a single countermeasure. Under a combination of padding and volume hiding, our attack cannot reveal any leaked file or keyword.

Remarkably, the blue line, i.e. the performance of VAL-Attack under padding, rises at some point in

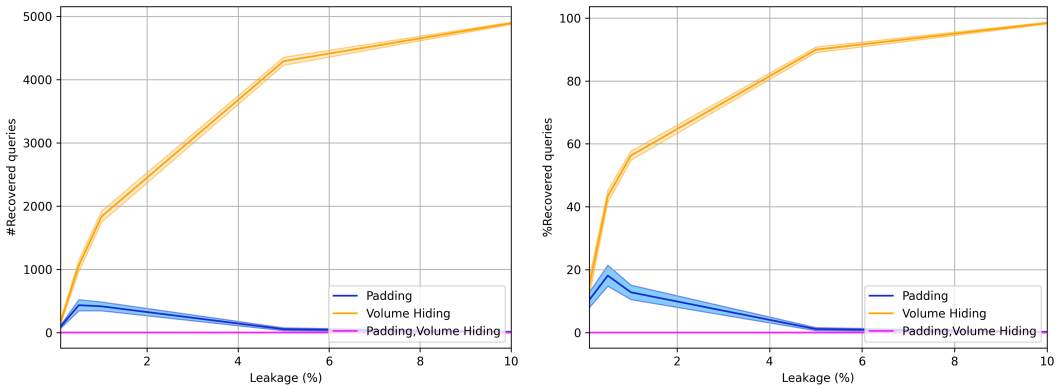
terms file recovery percentage. We see in Figure 7.3a.i, that the number of files keeps increasing as the leakage percentage rises. For 5% leakage, we recover more files in percentage terms compared to 1% leakage. For a higher leakage percentage, i.e. 10% leakage, we recover only 60% of the available files in Figure 7.3a.ii; we also see a slightly less steep incremental line on the left.

In Figure 7.3b.i, we see that the padding countermeasure is effective for recovering queries. For a low leakage percentage, we see that the attack recovers more of the available queries compared to a higher leakage percentage, and 0 keywords are matched from 5% leakage. This recovery result is possible because there are fewer queries with ”fake responses”, such that we can not match them to a leaked keyword. In Figure 7.3b.ii, we see similar growth and decrease in query recovery around 0.5% leakage, indicating that the padding countermeasure is functional. Volume hiding is not that effective, but the countermeasure effect is noticeable compared to Figure 7.1b.



(a.i) #Files recovered **(a.ii)** %Files recovered

(a) Performance of VAL-Attack with countermeasures, based on the number and percentages of files recovered



(b.i) #Queries recovered **(b.ii)** %Queries recovered

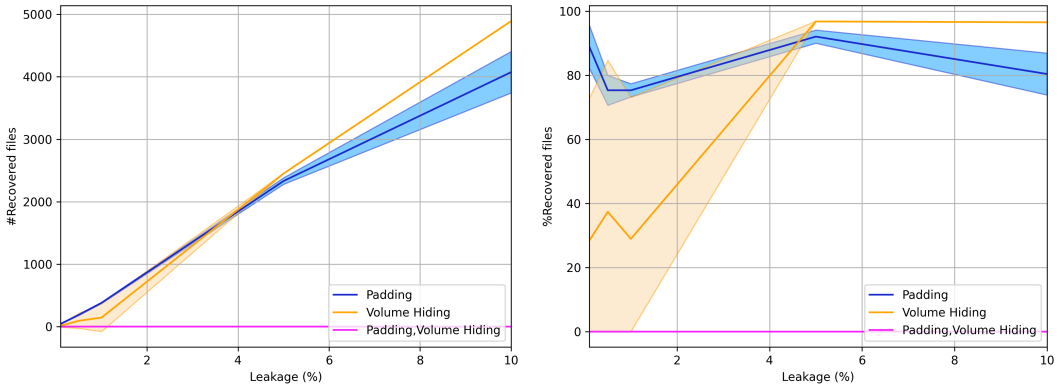
(b) Performance of VAL-Attack with countermeasures, based on the number and percentages of queries recovered

Figure 7.3: Performance of VAL-Attack with countermeasures in the Enron dataset

The plots showing the results for the VAL-Attack under countermeasures on the Lucene dataset show something remarkable: The orange line, i.e. the results with volume hiding, is very widespread, indicating that the attack did not have a fixed result but rather varies considerably. The growth and decrease in Figure 7.4a.ii around 0.5% and 1% leakage correspond to the significant error rate. The attack reveals, on average, around 30% of the leaked files but can, in specific circumstances, reveal between 80% and 0% of the available files.

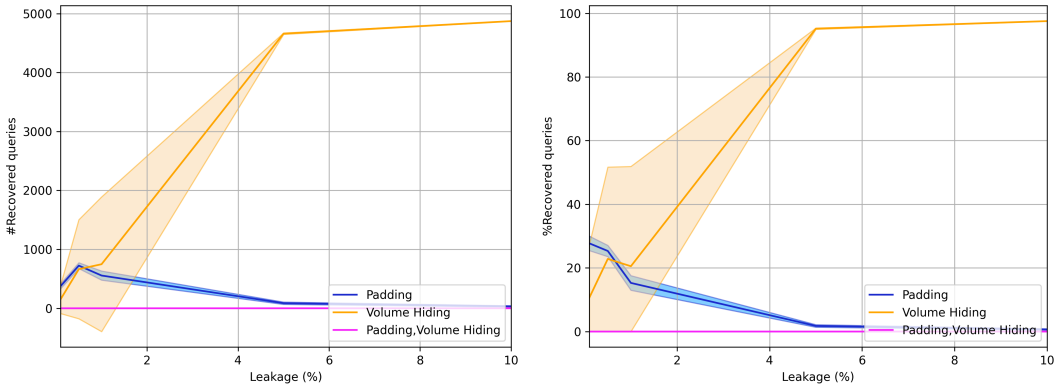
The performance under the padding countermeasure is also remarkable. It rises for 5% leakage in percentage terms of recovered files but decreases afterwards, as shown in Figure 7.4a.ii. We also see that the error rate increases when the leakage percentage reaches 10%. For the recovered queries, we see a slight increase of around 0.5% leakage in the actual number of recovered queries but a decrease in the percentage of available queries, indicating that we recover more queries compared

to the previous leakage percentage. However, fewer of the available keywords are matched, see Figure 7.4b.ii.



(a.i) #Files recovered **(a.ii)** %Files recovered

(a) Performance of VAL-Attack with countermeasures, based on the number and percentages of files recovered



(b.i) #Queries recovered **(b.ii)** %Queries recovered

(b) Performance of VAL-Attack with countermeasures, based on the number and percentages of queries recovered

Figure 7.4: Performance of VAL-Attack with countermeasures in the Lucene dataset

7.4. Discussion on Experiments

We purposely chose specific parameters in our experiments and only compared our attack with two popular attacks [6, 40]. In this section, we want to support our choices and discuss the additional options.

7.4.1. Parameters

We used 5,000 high selectivity keywords, i.e. keywords that occur the most in the dataset. This number is chosen because a practical SE application will probably not have just a few search terms in a real-world scenario. Other attacks [6, 15, 28] have experimented with only 150 query tokens and 500 keywords, and we argue that this may not be realistic. Our attack can recover almost all underlying keywords for an experiment with 500 keywords because the number of files is equal, but a slight variation in keyword occurrence[2].

We cut the number of Wikipedia files to 50,000. We did this to better present the comparison with the Enron and Lucene datasets. The attack may also take longer to run when all Wikipedia files are considered. The results will also differ as the number of files leaked increases similarly. The percentage of files recovered will probably be the same because of keyword distribution among the files.

If we ran the experiments with a higher leakage percentage, the attack would eventually recover more

[2]Tested, but results not provided

files, as more are available, but we would not recover more keywords. As with 30% leakage, we see that we have recovered all 5,000 keywords.

Our attack performs without false positives, we did so because they would not improve the performance, and an attacker cannot better understand the data if he cannot rely on it. If we allowed the attack to return false positives, we would have 5,000 matches for underlying keywords, of which not all are correct. The attack performance will not change since we will only measure the correct matches, which we already did.

7.4.2. Attack Comparison

Figure 7.2a only compared our attack with the LEAP attack rather than the $\text{Subgraph}_{\text{VL}}$ attack. We did so because the latter does not reveal encrypted files and thus cannot be compared. If we choose to compare the attack to ours, we would have to rebuild their attack using their strategy, which is out of the scope of this work.

We used the Enron dataset to compare the VAL-Attack to the LEAP and the $\text{Subgraph}_{\text{VL}}$. Their work [6, 40] used the Enron dataset to show their performance. If we used the Lucene or Wikipedia dataset instead to present the comparison, we would have no foundation in the literature to support our claim. A comparison of all the datasets would still show that our attack surpasses the attacks since, in theory, we exploit more.

We discussed other attacks, like the IKK and the Count attack, but we did not compare their performance with ours. While these attacks exploit the same leakage, we could still consider them. However, since LEAP is considered the most state-of-the-art attack and has already been compared with the other attacks in [40], we thus only have to compare the LEAP attack here. Accordingly, a comparison with all attacks would not affect the results and conclusion of this paper.

8

Discussion & Conclusion

This study has shown research in different leakage patterns and leakage pattern abusing attacks on SSE schemes. With the knowledge about previous attacks, we created an attack that extended previously proposed techniques and combined leakage patterns.

8.1. Discussion

During our research, we agreed upon several assumptions. One of them is having access to leaked documents and observing the whole keyword set before an active attack; another is the absence of countermeasures in the SSE scheme. We will discuss these assumptions in the following sections.

8.1.1. Leaked Plaintext Data

During the experiment of the research attacks, they assumed to have access to several leaked plaintext documents or underlying keywords from query tokens. We argue that this is a valid argument for the experiment and should be used to show the performance. However, an adversary that does not have access to any leaked documents will not be able to reveal the underlying keyword to a query. The only possible thing he can do is by retrieving the master key K such that he can decrypt the encrypted data. However, obtaining the master key is not within the scope of this research.

An adversary with access to the encrypted data, such as the cloud service provider, as described in Section 2.3.1.1, could have access to plaintext documents because of moving the original data to the encrypted storage server. We agree upon using the leaked plaintext data to show the attack and its performance in the experiments.

8.1.2. Countermeasures

Nowadays, many systems have countermeasures to prevent leakage or reduce the attack possibilities on SSE schemes. These countermeasures are disregarded in the proposed attack experiments. Several authors propose countermeasures against their attack, and others use previously proposed countermeasures in their experiments. However, the main vision is to run the attack without countermeasures.

We agree that this shows the effectiveness and importance of countermeasures. Therefore, implementing new data storage servers that use SSE schemes with countermeasures is essential.

An argument in favour of running a proposed attack against countermeasures is mitigation. A new attack might not be mitigated with the current countermeasures. Our attack is mitigated using multiple countermeasures, while other attacks are mitigated using only a single countermeasure. Therefore, it is still essential to test against current countermeasures.

8.2. Conclusion

This study aimed to explore the current attacks on SSE schemes and eventually create an attack that improves the current state-of-the-art. We first researched the background of SE schemes, how they work, and their purpose. Next, we explored the `access pattern`, the `search pattern` and the `volume pattern`. After this background research, we examined different existing attacks, how they work, and their target. Eventually, we created an extensive overview of these attacks and were able to create a new attack. The sub-questions as defined in Section 1.1 are answered below.

What is the main difference in existing attacks on SSE schemes, and what do they abuse?
The main difference in existing attacks is the leakage pattern they abuse, their techniques to match documents, and their exploited information. We created an extensive overview that compares all the researched attacks, and we concluded that abusing the `access pattern` is heavily exploited based on this comparison. Other differences are whether an attack outputs false positives and what kind of input data they use.

How can we improve the state-of-the-art attacks by exploiting a new branch of leakage or matching technique?
We researched different attacks that abuse different leakage patterns, have different matching techniques, exploit different information and perform differently. We created an extensive comparison and concluded that the `volume pattern` was not exploited to the full extent.

We proposed an attack that consists of a combination of prior techniques to match keywords and files. The attack improves the matching technique from the LEAP attack, and it benefits from leakage from the `access pattern` and the `volume pattern`, a combination that has not been used before.

We showed that our attack has excellent performance, and we compared it to the LEAP attack and the subgraph$_{VL}$ attack. The number of matched files is with more remarkable improvement than the number of queries recovered compared to the LEAP attack. Nevertheless, since the technique uses both the document size and the response per query, it requires more countermeasures, making it a strong attack.

The attack recovers around 98% of the leaked documents and above 90% for query recovery for very low leakage.

Therefore we can answer our main research question:

How could we match queries and documents in a passive attack by researching different attacks and improve the current state-of-the-art attack to capture a high recovery rate against popular defences?

We performed an extensive literature study on the current attacks on SSE schemes. With the knowledge from the attacks we have created a new attack, the VAL-Attack, it combines multiple leakage patterns, i.e. the `access pattern` and the `volume pattern`. Our attack also improves the current state-of-the-art LEAP attack [40] by expanding their keyword to query matching functionality in our attack.

Therefore, we can match queries and documents in a passive attack that abuses multiple leakage patterns while considering the SSE scheme regulations.

8.3. Future Work

We discovered that there are still some ideas or techniques that can be exploited further to improve attacks on SSE schemes.

8.3.1. Keyword Matches

Our attack results are promising; we match almost all leaked documents to server documents. Then we match keywords to queries based on unique occurrence patterns in our matched documents. However, the percentage of keywords matched is only 100% after 10% server leakage due to not enough unique occurrences. Possible future work is matching keywords with all leaked documents matched while the

leakage percentage is below 10%, such the results for the VAL-Attack can be improved even further. Such a technique can consist of a combination of existing techniques or any other matching procedure that we have not considered.

8.3.2. Combining Passive with Active Attacks

Our proposed attack recovers more than 90% of the available keywords with no false positives. Based on our attack, other attack combinations are still possible. If it is possible to upload files into the SSE scheme, we can recover all keywords by an injection attack. The recovered keyword matches reduce the available keyword set for a possible combination with a passive attack. This way, a combination of a leakage abusing passive attack with a file injecting active attack can exist and recover all the query matches with an enlarging reducing number of file injections.

8.3.3. Combining other Leakage Patterns

In our work, we created an attack that exploits the `access pattern` and the `volume pattern`. However, the `search pattern` is leaked simultaneously. A possible new attack could exploit all of these three patterns or any other combination than ours. The difficulty in extending an attack by exploiting the `search pattern` is the reality of the experiments. The attacks that exploit this pattern, from Liu et al. [36] and Oya et al. [42], create fake data by adding noise to actual search frequencies from Google Trends. Nevertheless, this leakage is still considered adequate and can be further exploited by combining existing attacks or creating a completely new one.

8.3.4. Range Queries

Our attack works on SSE schemes that support single query search only, and we did not do any extensive research for SSE schemes that support range queries or other search strategies. The research of the leakage patterns in range query supporting search schemes is a different topic and therefore considered for future work. Nevertheless, range query SSE schemes have leakage and can still be attacked. Therefore, it is essential to have countermeasures for these SSE schemes.

8.3.5. Conjunctive Queries

Besides single query searches, SSE schemes also exist that support conjunctive search queries [37, 47]. These schemes support combinations of queries. A user can search for documents containing multiple keywords, while in single keyword search schemes, a user can only search for one query. However, these schemes also have leakage [63] and therefore are vulnerable to attacks. We did not research attacks on these SSE schemes, but Zhang et al. [61] and Wang et al. [53] propose an injection attack on conjunctive search schemes. We leave it for future work to create a passive attack with leakage of the `access and volume pattern` on an SSE scheme that supports conjunctive search queries.

References

[1] Michel Abdalla et al. “Searchable Encryption Revisited: Consistency Properties, Relation to Anonymous IBE, and Extensions”. In: *Advances in Cryptology – CRYPTO 2005*. Ed. by Victor Shoup. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 205–222. isbn: 978-3-540-31870-5.

[2] Rakesh Agrawal et al. “Order Preserving Encryption for Numeric Data”. In: 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] Alexandre Anzala-Yamajako et al. “No Such Thing as a Small Leak: Leakage-Abuse Attacks Against Symmetric Searchable Encryption”. In: *Communications in Computer and Information Science*. E-Business and Telecommunications 14th International Joint Conference, ICETE 2017, Madrid, Spain, July 24-26, 2017, Revised Selected Paper 990 (Jan. 2019), pp. 253–277. url: https://hal.archives-ouvertes.fr/hal-01990354.

[4] Mihir Bellare, Alexandra Boldyreva, and Adam O’Neill. *Deterministic and Efficiently Searchable Encryption*. Cryptology ePrint Archive, Report 2006/186. https://ia.cr/2006/186. 2006.

[5] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. ” O’Reilly Media, Inc.”, 2009.

[6] Laura Blackstone, Seny Kamara, and Tarik Moataz. *Revisiting Leakage Abuse Attacks*. Cryptology ePrint Archive, Report 2019/1175. https://ia.cr/2019/1175. 2019.

[7] Alexandra Boldyreva et al. *Order-Preserving Symmetric Encryption*. Cryptology ePrint Archive, Report 2012/624. https://ia.cr/2012/624. 2012.

[8] Dan Boneh, Amit Sahai, and Brent Waters. *Functional Encryption: Definitions and Challenges*. Cryptology ePrint Archive, Report 2010/543. https://ia.cr/2010/543. 2010.

[9] Dan Boneh and Brent Waters. *Conjunctive, Subset, and Range Queries on Encrypted Data*. Cryptology ePrint Archive, Report 2006/287. https://ia.cr/2006/287. 2006.

[10] Dan Boneh et al. “Public Key Encryption with Keyword Search”. In: *Advances in Cryptology - EUROCRYPT 2004*. Vol. 3027. Apr. 2004, pp. 506–522. isbn: 978-3-540-21935-4. doi: 10.1007/978-3-540-24676-3_30.

[11] Raphael Bost. *Sophos - Forward Secure Searchable Encryption*. Cryptology ePrint Archive, Report 2016/728. https://ia.cr/2016/728. 2016.

[12] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. “Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1465–1482. isbn: 9781450349468. doi: 10.1145/3133956.3133980. url: https://doi.org/10.1145/3133956.3133980.

[13] David Cash et al. *Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation*. Cryptology ePrint Archive, Report 2014/853. https://ia.cr/2014/853. 2014.

[14] 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.

[15] David Cash et al. *Leakage-Abuse Attacks Against Searchable Encryption*. Cryptology ePrint Archive, Report 2016/718. https://ia.cr/2016/718. 2016.

[16] Yan-Cheng Chang and Michael Mitzenmacher. “Privacy Preserving Keyword Searches on Remote Encrypted Data”. In: *Applied Cryptography and Network Security*. Ed. by John Ioannidis, Angelos Keromytis, and Moti Yung. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 442–455. isbn: 978-3-540-31542-1.

[17] Melissa Chase and Seny Kamara. *Structured Encryption and Controlled Disclosure*. Cryptology ePrint Archive, Report 2011/010. https://ia.cr/2011/010. 2011.

[18] Guoxing Chen et al. “Differentially Private Access Patterns for Searchable Symmetric Encryption”. In: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. 2018, pp. 810–818. doi: 10.1109/INFOCOM.2018.8486381.

[19] Reza Curtmola et al. *Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions*. Cryptology ePrint Archive, Report 2006/210. https://ia.cr/2006/210. 2006.

[20] Marc Damie, Florian Hahn, and Andreas Peter. “A highly accurate query-recovery attack against searchable encryption using non-indexed documents”. English. In: *Proceedings of the 30th USENIX Security Symposium*. Publisher Copyright: © 2021 by The USENIX Association. All rights reserved.; 30th USENIX Security Symposium, USENIX Security 2021 ; Conference date: 11-08-2021 Through 13-08-2021. USENIX Association, 2021, pp. 143–160.

[21] Ioannis Demertzis and Charalampos Papamanthou. “Fast Searchable Encryption With Tunable Locality”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD ’17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 1053–1067. isbn: 9781450341974. doi: 10.1145/3035918.3064057. url: https://doi.org/10.1145/3035918.3064057.

[22] Apache Foundation. *Apache Foundation. 1999. Mail Archieves of Lucene*. 1999. url: https://mail-archives.apache.org/mod_mbox/#lucene.

[23] Craig Gentry. “Fully Homomorphic Encryption Using Ideal Lattices”. In: *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*. STOC ’09. Bethesda, MD, USA: Association for Computing Machinery, 2009, pp. 169–178. isbn: 9781605585062. doi: 10.1145/1536414.1536440. url: https://doi.org/10.1145/1536414.1536440.

[24] Eu-Jin Goh. *Secure Indexes*. Cryptology ePrint Archive, Report 2003/216. https://ia.cr/2003/216. 2003.

[25] 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.

[26] Zichen Gui, Kenneth G. Paterson, and Sikhar Patranabis. *Rethinking Searchable Symmetric Encryption*. Cryptology ePrint Archive, Report 2021/879. https://ia.cr/2021/879. 2021.

[27] Warren He et al. “ShadowCrypt: Encrypted Web Applications for Everyone”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’14. Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, pp. 1028–1039. isbn: 9781450329576. doi: 10.1145/2660267.2660326. url: https://doi.org/10.1145/2660267.2660326.

[28] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. “Access pattern disclosure on searchable encryption: Ramification, attack and mitigation”. In: *in Network and Distributed System Security Symposium (NDSS*. 2012.

[29] Seny Kamara and Tarik Moataz. “Computationally Volume-Hiding Structured Encryption”. In: *Advances in Cryptology – EUROCRYPT 2019*. Ed. by Yuval Ishai and Vincent Rijmen. Cham: Springer International Publishing, 2019, pp. 183–213. isbn: 978-3-030-17656-3.

[30] Seny Kamara and Charalampos Papamanthou. “Parallel and Dynamic Searchable Symmetric Encryption”. In: *Financial Cryptography and Data Security*. Ed. by Ahmad-Reza Sadeghi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 258–274. isbn: 978-3-642-39884-1.

[31] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. “Dynamic Searchable Symmetric Encryption”. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS ’12. Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 965–976. isbn: 9781450316514. doi: 10.1145/2382196.2382298. url: https://doi.org/10.1145/2382196.2382298.

[32] Dalia Khader. *Public Key Encryption with Keyword Search based on K-Resilient IBE*. Cryptology ePrint Archive, Report 2006/358. https://ia.cr/2006/358. 2006.

[33] Y.A.M. Kortekaas. *Access Pattern Hiding Aggregation over Encrypted Databases*. Oct. 2020. url: http://essay.utwente.nl/83874/.

[34] Billy Lau et al. “Mimesis Aegis: A Mimicry Privacy Shield–A System’s Approach to Data Privacy on Public Cloud”. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 33–48. isbn: 978-1-931971-15-7. url: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/lau.

[35] Jiangnan Li and Jinyuan Sun. “A Practical Searchable Symmetric Encryption Scheme for Smart Grid Data”. In: *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)* (2019), pp. 1–6.

[36] Chang Liu et al. *Search Pattern Leakage in Searchable Encryption: Attacks and New Construction*. Cryptology ePrint Archive, Report 2013/163. https://ia.cr/2013/163. 2013.

[37] Changshe Ma, Yiping Gu, and Hongfei Li. *Practical Searchable Symmetric Encryption Supporting Conjunctive Queries without Keyword Pair Result Pattern Leakage*. Cryptology ePrint Archive, Report 2020/048. https://ia.cr/2020/048. 2020.

[38] Qiumao Ma et al. “SE-ORAM: A Storage-Efficient Oblivious RAM for Privacy-Preserving Access to Cloud Storage”. In: *2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing (CSCloud)*. 2016, pp. 20–25. doi: 10.1109/CSCloud.2016.24.

[39] Brice Minaud and Michael Reichle1. *Dynamic Local Searchable Symmetric Encryption*. Jan. 2022. url: https://arxiv.org/pdf/2201.05006.pdf.

[40] 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.

[41] Jianting Ning et al. “Passive Attacks Against Searchable Encryption”. In: *IEEE Transactions on Information Forensics and Security* 14.3 (2019), pp. 789–802. doi: 10.1109/TIFS.2018.2866321.

[42] Simon Oya and Florian Kerschbaum. “Hiding the Access Pattern is Not Enough: Exploiting Search Pattern Leakage in Searchable Encryption”. In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 127–142. isbn: 978-1-939133-24-3. url: https://www.usenix.org/conference/usenixsecurity21/presentation/oya.

[43] Sarvar Patel et al. “Mitigating Leakage in Secure Cloud-Hosted Data Structures: Volume-Hiding for Multi-Maps via Hashing”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 79–93. isbn: 9781450367479. doi: 10.1145/3319535.3354213. url: https://doi.org/10.1145/3319535.3354213.

[44] Martin F. Porter. “An algorithm for suffix stripping”. In: *Program* 40 (1980), pp. 211–218.

[45] 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* (2016).

[46] Sarah Renwick and Keith Martin. “Practical Architectures for Deployment of Searchable Encryption in a Cloud Environment”. In: *Cryptography* 1 (Nov. 2017), p. 19. doi: 10.3390/cryptography1030019.

[47] Eun-Kyung Ryu and Tsuyoshi Takagi. “Efficient Conjunctive Keyword-Searchable Encryption”. In: *21st International Conference on Advanced Information Networking and Applications Workshops (AINAW’07)*. Vol. 1. 2007, pp. 409–414. doi: 10.1109/AINAW.2007.166.

[48] Zhiwei Shang et al. *Obfuscated Access and Search Patterns in Searchable Encryption*. 2021. doi: 10.48550/ARXIV.2102.09651. url: https://arxiv.org/abs/2102.09651.

[49] David Shapiro. *Convert Wikipedia database dumps into plaintext files. (2021)*. 2021. url: https://github.com/daveshap/PlainTextWikipedia.

[50] 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.

[51] trends.google.com. *Google Trends*. 2012. url: http://trends.google.com/trends.

[52] Peter van Liesdonk et al. “Computationally Efficient Searchable Symmetric Encryption”. Undefined. In: *Proceedings of the Seventh International Workshop on Secure Data Management, SDM 2010*. Ed. by Willem Jonker and M. Petkovic. Lecture Notes in Computer Science 6358. 10.1007/978-3-642-15546-8_7 ; null ; Conference date: 17-09-2010 Through 17-09-2010. Netherlands: Springer, Sept. 2010, pp. 87–100. isbn: 978-3-642-15545-1. doi: 10.1007/978-3-642-15546-8_7.

[53] Gaoli Wang, Zhenfu Cao, and Xiaolei Dong. “Improved File-injection Attacks on Searchable Encryption Using Finite Set Theory”. In: *The Computer Journal* 64.8 (Dec. 2020), pp. 1264–1276. issn: 0010-4620. doi: 10.1093/comjnl/bxaa161. eprint: https://academic.oup.com/comjnl/article-pdf/64/8/1264/39904343/bxaa161.pdf. url: https://doi.org/10.1093/comjnl/bxaa161.

[54] Guofeng Wang et al. “Leakage Models and Inference Attacks on Searchable Encryption for Cyber-Physical Social Systems”. In: *IEEE Access* 6 (2018), pp. 21828–21839. doi: 10.1109/ACCESS.2018.2800684.

[55] Guofeng Wang et al. “Query Recovery Attacks on Searchable Encryption Based on Partial Knowledge”. In: Jan. 2018, pp. 530–549. isbn: 978-3-319-78812-8. doi: 10.1007/978-3-319-78813-5_27.

[56] Stephanie Wang et al. “Practical Volume-Based Attacks on Encrypted Databases”. In: *2020 IEEE European Symposium on Security and Privacy (EuroS&P)* (2020), pp. 354–369.

[57] CMU William W. Cohen MLD. *Enron Email Datasets*. 2015. url: https://www.cs.cmu.edu/~enron/.

[58] Lei Xu et al. “Hardening Database Padding for Searchable Encryption”. In: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. 2019, pp. 2503–2511. doi: 10.1109/INFOCOM.2019.8737588.

[59] Lei Xu et al. *Interpreting and Mitigating Leakage-abuse Attacks in Searchable Symmetric Encryption*. Cryptology ePrint Archive, Report 2021/1593. https://ia.cr/2021/1593. 2021.

[60] Rui Zhang and Hideki Imai. “Combining Public Key Encryption with Keyword Search and Public Key Encryption”. In: *IEICE Transactions on Information and Systems* 92.5 (Jan. 2009), pp. 888–896. doi: 10.1587/transinf.E92.D.888.

[61] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. “All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption”. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 707–720. isbn: 978-1-931971-32-4. url: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/zhang.

[62] Qingji Zheng, Shouhuai Xu, and Giuseppe Ateniese. *VABKS: Verifiable Attribute-based Keyword Search over Outsourced Encrypted Data*. Cryptology ePrint Archive, Report 2013/462. https://ia.cr/2013/462. 2013.

[63] Cong Zuo et al. *Searchable Encryption for Conjunctive Queries with Extended Forward and Backward Privacy*. Cryptology ePrint Archive, Report 2021/1585. https://ia.cr/2021/1585. 2021.

A

Notations

Parameter	Description
n	$\in \mathcal{N}$, usually defines the maximum amount of files in the scheme
n'	$\in \mathcal{N}$, defines the maximum amount of leaked files in the scheme
m	$\in \mathcal{N}$, usually defines the maximum amount of keywords in the scheme
m'	$\in \mathcal{N}$, defines the maximum amount of leaked keywords in the scheme
d_i	Single plaintext document
ed_i	Single encrypted server document
K	Master (encryption) key
w	Single keyword
q	Single query token
F	Plaintext document set, $F = \{d_1, ..., d_n\}$
F'	Leaked document set, $F' = \{d_1, ..., d_{n'}\}$
E/EDB	Server document set, $E = \{ed_1, ..., ed_n\}$
W	Keyword universe, $W = \{w_1, ..., w_m\}$
W'	Known keywords, $W' = \{w_1, ..., w_{m'}\}$
Q	Query set, $Q = \{q_1, ..., q_m\}$
KeyGen	Master key K generation algorithm in SSE schemes
ENC	Encryption algorithm in SSE schemes
QueryGen	Query generation algorithm in SSE schemes
Search	Search algorithm in SSE schemes
AddToken	Algorithm in dynamic SSE schemes to create an insertion token
DelToken	Algorithm in dynamic SSE schemes to create a deletion token
Add	Add file algorithm in dynamic SSE schemes
Del	Delete file algorithm in dynamic SSE schemes
t_a	Insertion token
t_d	Deletion token
$f(q)$	Function in L1, that outputs the document identifiers to the corresponding query q
AP	Access Pattern function (Definition 3)
$D(w_i)$	Row binary vector with i-th entry 1 if the underlying data of D_j contains the underlying keyword of the query w_i, 0 otherwise
SP	Search Pattern function (Definition 4)
$\mathcal{H}$	Binary matrix storing if a query is sent multiple times
Vol	Volume Pattern function (Definition 5
$\|d_i\|$	Number of keywords in document d_i
$\|d_i\|_w$	Volume of document d_i in bytes
A'	$m' \times n'$ matrix of leaked documents
B	$m \times n$ matrix of server documents

Table A.1: All notations used in this document

Parameter	Description
A''_{map}	Extended $m \times n'$ matrix of leaked documents
B_{map}	$m \times n$ matrix of server documents (B)
M'	$n' \times n'$ co-occurrence matrix of F'
M	$n \times n$ co-occurrence matrix of E
C	Set of matched documents
R	Set of matched queries
A''_c	Matrix A''_{map} with only the columns of C
B_c	Matrix B with only the columns of C
A''_{c_j}	Column j of matrix A''_c
B_{c_j}	Column j of matrix B_c
A''_x	List of similar rows in A''_c
B_x	List of similar rows in B_c
A''_r	Matrix A''_{map} with only the rows of R
B_r	Matrix B with only the rows of R
A''_{CR}	Matrix A''_{map} with only the rows of R and columns from documents with equal volume and number of keywords
B_{CR}	Matrix B with only the rows of R and columns from documents with equal volume and number of keywords

Table A.1: All notations used in this document

B

Abbreviations

Access Pattern Reveals the identifiers of the documents matching a query. 1–3, 7–9, 15, 16, 19, 20, 22, 23, 31, 32, 40, 41, 46, 47

Cloud service provider A cloud service provider (CSP) is a third-party server that offers a cloud-based platform. Users can upload and query data on this platform. 1, 45

GDPR General Data Protection Regulation. Regulation in EU law on data protection and privacy in the European Union. 1

SE Searchable Encryption. ii, 1, 5, 23, 46

Search Pattern Reveals the frequency a query is sent to the server. 1, 2, 7, 9, 10, 16, 46, 47

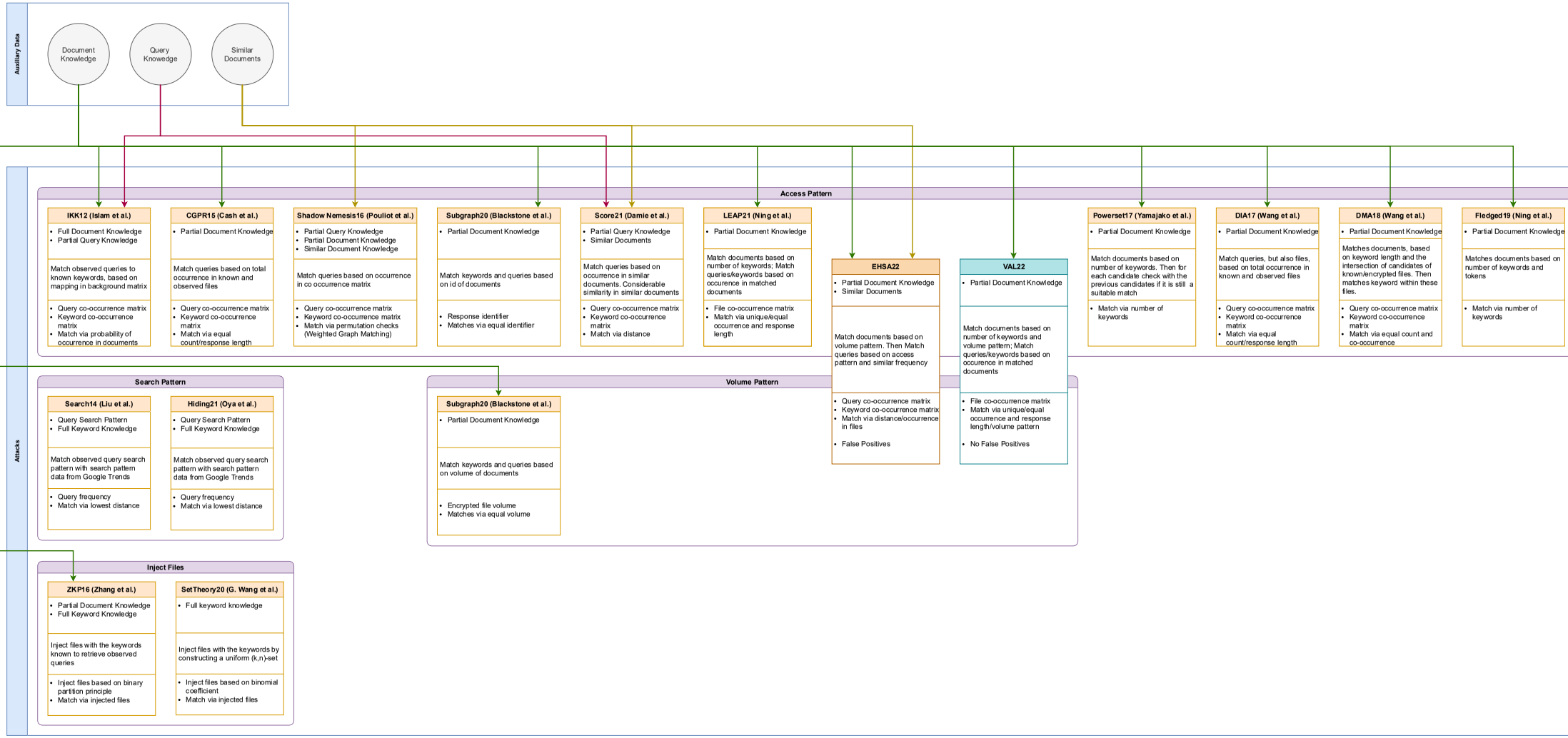
SSE Searchable Symmetric Encryption. ii, 1, 2, 4–7, 9, 10, 12, 14, 16, 20, 22, 23, 34, 35, 45–47, 52

VAL-Attack Volume and Access pattern Leakage abuse Attack. ii, 1, 3, 22, 25–27, 31, 33, 38–44, 46, 47

Volume Pattern Reveals the volume of a document matching a query. 1–3, 7, 9, 10, 14, 20, 22–24, 26, 31, 39, 40, 46, 47

C

Attack Overview Summary


Auxiliary Data
Document Knowledge
Query Knowedge
Similar Documents
Attacks
Access Pattern
IKK12 (Islam et al.)
• Full Document Knowledge
• Partial Query Knowledge
Match observed queries to known keywords, based on mapping in background matrix
• Query co-occurrence matrix
• Keyword co-occurrence matrix
• Match via probability of occurrence in documents
CGPR15 (Cash et al.)
• Partial Document Knowledge
Match queries based on total occurrence in known and observed files
• Query co-occurrence matrix
• Keyword co-occurrence matrix
• Match via equal count/response length
Shadow Nemesis16 (Pouliot et al.)
• Partial Query Knowledge
• Partial Document Knowledge
• Similar Document Knowledge
Match queries based on occurrence in co occurrence matrix
• Query co-occurrence matrix
• Keyword co-occurrence matrix
• Match via permutation checks (Weighted Graph Matching)
Subgraph20 (Blackstone et al.)
• Partial Document Knowledge
Match keywords and queries based on id of documents
• Response identifier
• Matches via equal identifier
Score21 (Damie et al.)
• Partial Query Knowledge
• Similar Documents
Match queries based on occurrence in similar documents. Considerable similarity in similar documents
• Query co-occurrence matrix
• Keyword co-occurrence matrix
• Match via distance
LEAP21 (Ning et al.)
• Partial Document Knowledge
Match documents based on number of keywords; Match queries/keywords based on occurence in matched documents
• File co-occurrence matrix
• Match via unique/equal occurrence and response length
EHSA22
• Partial Document Knowledge
• Similar Documents
Match documents based on volume pattern. Then Match queries based on access pattern and similar frequency
• Query co-occurrence matrix
• Keyword co-occurrence matrix
• Match via distance/occurrence in files
• False Positives
VAL22
• Partial Document Knowledge
Match documents based on number of keywords and volume pattern; Match queries/keywords based on occurence in matched documents
• File co-occurrence matrix
• Match via unique/equal occurrence and response length/volume pattern
• No False Positives
Powerset17 (Yamajako et al.)
• Partial Document Knowledge
Match documents based on number of keywords. Then for each candidate check with the previous candidates if it is still a suitable match
• Match via number of keywords
DIA17 (Wang et al.)
• Partial Document Knowledge
Match queries, but also files, based on total occurrence in known and observed files
• Query co-occurrence matrix
• Keyword co-occurrence matrix
• Match via equal count/response length
DMA18 (Wang et al.)
• Partial Document Knowledge
Matches documents, based on keyword length and the intersection of candidates of known/encrypted files. Then matches keyword within these files.
• Query co-occurrence matrix
• Keyword co-occurrence matrix
• Match via equal count and co-occurrence
Fledged19 (Ning et al.)
• Partial Document Knowledge
Matches documents based on number of keywords and tokens
• Match via number of keywords
Search Pattern
Search14 (Liu et al.)
• Query Search Pattern
• Full Keyword Knowledge
Match observed query search pattern with search pattern data from Google Trends
• Query frequency
• Match via lowest distance
Hiding21 (Oya et al.)
• Query Search Pattern
• Full Keyword Knowledge
Match observed query search pattern with search pattern data from Google Trends
• Query frequency
• Match via lowest distance
Volume Pattern
Subgraph20 (Blackstone et al.)
• Partial Document Knowledge
Match keywords and queries based on volume of documents
• Encrypted file volume
• Matches via equal volume
Inject Files
ZKP16 (Zhang et al.)
• Partial Document Knowledge
• Full Keyword Knowledge
Inject files with the keywords known to retrieve observed queries
• Inject files based on binary partition principle
• Match via injected files
SetTheory20 (G. Wang et al.)
• Full keyword knowledge
Inject files with the keywords by constructing a uniform (k,n)-set
• Inject files based on binomial coefficient
• Match via injected files

D

Earlier Improvement Idea Results

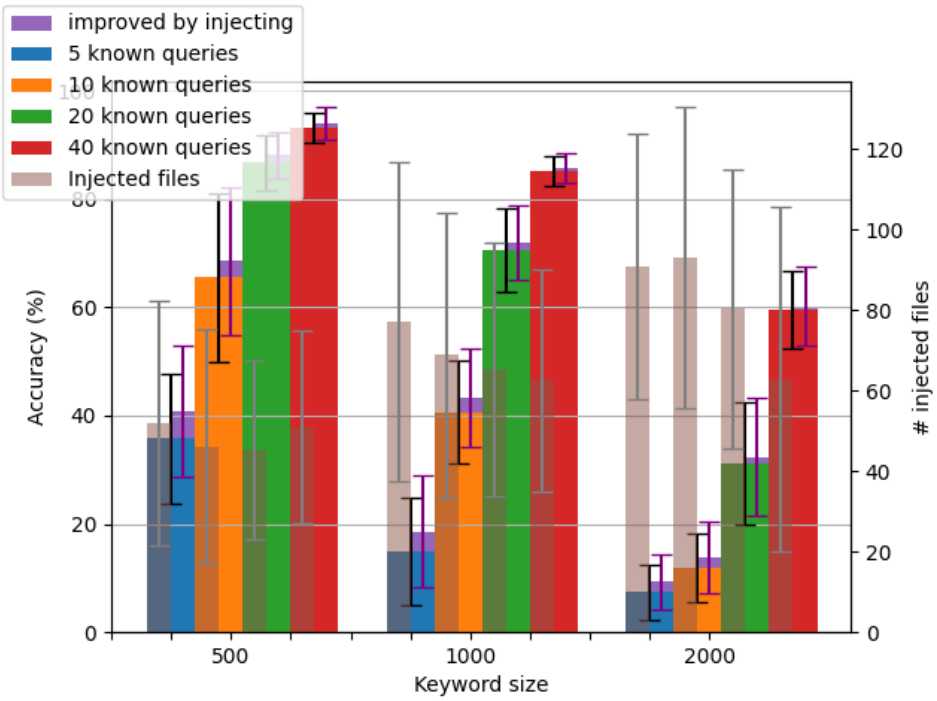


Figure D.1: Possible improvement for the Score attack, with a combination of injecting files afterwards. The results are the maximum of their Clustering strategy, described in [20, Appendix C]

E

Explore More, Reveal More - VAL: Volume and Access Pattern Leakage-abuse Attack with Leaked Documents

We created a paper describing our research, attack, and results. We uploaded this paper to the 27th European Symposium on Research in Computer Security (ESORICS) 2022 conference.

Explore More, Reveal More - VAL: Volume and Access Pattern Leakage-abuse Attack with Leaked Documents

Abstract. Searchable Encryption schemes provide secure search over encrypted databases while allowing admitted information leakages. Generally, the leakages can be categorized into `access` and `volume pattern`. In most existing SE schemes, these leakages are caused by practical designs but are considered an acceptable price to achieve high search efficiency. Recent attacks have shown that such leakages could be easily exploited to retrieve the underlying keywords for search queries. Under the umbrella of attacking SE, we design a new Volume and Access Pattern Leakage-Abuse Attack (VAL-Attack) that improves the matching technique of LEAP (CCS '21) and exploits both the `access and volume patterns`. Our proposed attack only leverages leaked documents and the keywords present in those documents as auxiliary knowledge and can effectively retrieve document and keyword matches from leaked data. Furthermore, the recovery performs without false positives. We further compare VAL-Attack with two recent well-defined attacks on several real-world datasets to highlight the effectiveness of our attack and present the performance under popular countermeasures.

Keywords: Searchable Encryption · Access pattern · Volume pattern · Leakage · Attack

1 Introduction

In practice, to protect data security and user privacy (e.g., under GDPR), data owners may choose to encrypt their data before outsourcing to a third-party cloud service provider. Encrypting the data enhances privacy and gives the owners the feeling that their data is stored safely. However, this encryption relatively restricts the searching ability. Song et al. [34] proposed a Searchable Encryption (SE) scheme to preserve the search functionality over outsourced and encrypted data. In the scheme, the keywords of files are encrypted, and when a client wants to query a keyword, it encrypts the keyword as a token and sends it to the server. The server then searches the files with the token corresponding to the query, and afterwards, it returns the matching files. Since the seminal SE scheme, many research works have been presented in the literature, with symmetrical [7, 9, 10, 13] and asymmetrical encryption [1, 5, 37, 39]. Nowadays, SE schemes have been deployed in many real-world applications such as ShadowCrypt [17] and Mimesis Aegis [23].

Leakage. In an SE scheme, an operational interaction is usually defined as a client sending a query to the server and the server responding to the query with

the matching files. Nevertheless, this interaction could be eavesdropped on by an attacker. The messages could be intercepted because they are sent over an unprotected channel, or the attacker is the cloud service provider itself, who stores and accesses all the search requests and responses. The attacker may choose to match the query with a keyword such that he can comprehend what information is present on the server. The query and response here are what we may call *leakage*. In this work, we consider two main types of *leakage patterns*: the `access pattern`, the response from the server to a query, and the `search pattern`, which is the frequency a query is sent to the server. Besides these types, we also consider the `volume pattern` as leakage. This pattern is seen as the size of the stored documents on the server. The leakage patterns can be divided into four levels, by Cash et al. [8]. In this work, we consider our leakage level to be L2, which equals the fully-revealed occurrence pattern, together with the volume pattern to create a new attack on the SE scheme. Note that a formal definition of the leakages is given in Section 3.1

Attacks on SE. There exist various attacks on SE that work and perform differently. Most of these attacks take the leaked files as auxiliary knowledge. Islam et al. [18] presented the foundation for several attacks on SE schemes. They stated that, with sufficient auxiliary knowledge, one could create a co-occurrence matrix for both the leakage and the knowledge so that it can easily map queries to the keywords based on the lowest distance. Cash et al. [8] later proposed an attack where the query can be matched to a particular keyword based on the total occurrence in the leaked files. These attacks with knowledge about some documents are known as *passive attacks with pre-knowledge*. Blackstone et al. [4] developed a $\text{Subgraph}_{\text{VL}}$ attack that provides a relatively high query recovery rate even with a small subset of the leaked documents. The attack matches keywords based on unique document volumes as if it is the response pattern. Ning et al. [28] later designed the LEAP attack. LEAP combines the existing techniques, such as co-occurrence and the unique number of occurrences, to match the leaked files to server files and the known keywords to queries based on unique occurrences in the matched files. It makes good use of the unique count from the Count attack [8], a co-occurrence matrix from the IKK attack [18] (although LEAP inverts it to a document co-occurrence matrix) and finally, unique patterns to match keywords and files. Note that we give related work and general comparison in Section 6.

Limitations. The works in [4, 8, 18, 28] explain their leakage-abusing methods, but they only abuse a single leakage pattern, while multiple are leaked in SE schemes. Besides the leakage patterns, the state-of-the-art LEAP attack abuses the `access pattern` but does not exploit its matching techniques to the full extent. In addition to extending their attack, a combination of leakage can be used to match more documents and queries.

Research question. We aim to address the issue of matching keywords by exploiting both the `access pattern` and `volume pattern`. The following question arises naturally:

Could we match queries and documents in a passive attack by exploiting the volume and access patterns to capture a high recovery rate against popular defences?

Technical Roadmap. We design an attack that shows the danger of leaking the `access and volume pattern`. The VAL-Attack follows the SE scheme and matches leaked documents and keywords to server data, and it does so by exploiting both the `pattern`. The results from the attack are all correct, i.e. no false positives. Our attack has an improved matching technique not based on exact matches, compared to the LEAP attack; moreover, we also consider the occurrence in unmatched documents. Finally, we can match more files based on the size of each document, either by direct or indirect equality.

We compare our attack to several others. The bottom part of Table 3 compares four existing attacks, including ours, showing how our attack differs from existing attacks.

Contributions. We answer the above research question by designing an attack that matches leaked files and keywords. Our attack expands the matching techniques from the LEAP attack [28] and exploits the `volume pattern` to match more documents. The attack improves the LEAP attack by fully exploring the leakage information and combining the uniqueness of document volume to match more files. These matches can then be used to extract keyword matches. All the matches found are correct, as we argue that false positives are not valuable in real-world attacks.

- Besides exploiting the `access pattern`, we also abuse `volume pattern` leakage. We match documents based on a unique combination of volume and number of keywords with both leakage patterns. We can match almost all leaked documents to server documents using this approach.

- We match keywords using their occurrence pattern in matched files.

- Besides matching keywords in matched files, we use all leaked documents for unique keyword occurrence, expanding the keyword matching technique from the LEAP attack. We do this to get the maximum amount of keyword matches from the unique occurrence pattern.

We run our attack against three different datasets to test the performance, where we see that the results are outstanding as we match almost all leaked documents and a considerable amount of leaked keywords. Finally, we compare our attack to the existing state-of-the-art LEAP and $\text{Subgraph}_{\text{VL}}$ attacks. Our attack performs great in revealing files and underlying keywords. In particular, it surpasses the LEAP attack, revealing significantly more leaked files and keywords. VAL-Attack recovers almost 98% of the known files and above 93% of the keyword matches available to the attacker once the leakage percentage reaches 5%. When 10% of the Enron database is leaked, which is 3,010 files with 4,962 keywords, we match 2,950 files and 4,909 queries, respectively, corresponding to 98% and 99%. VAL-Attack can still compromise encrypted information, e.g., over 90% recovery (with 10% leakage) under volume hiding in Enron and Lucene,

even under several popular countermeasures. We note that our proposed attack is vulnerable to a combination of padding and volume hiding.

2 Preliminaries

Before proceeding to the VAL-Attack, we first review SE and define some notations that we will use throughout the paper.

2.1 Searchable Encryption

In a general SE scheme, a user encrypts her data and uploads the encrypted data to a server. After uploading the data, the user can send a query containing an encrypted keyword to the server, and the server will then respond with the corresponding data. We assume the server is honest-but-curious, meaning that it will follow the protocol but will try to retrieve as much information as possible.
The scheme. At a high level, an SE scheme consists of three polynomial-time algorithms: Enc, QueryGen and Search [13, 15, 21, 24, 27]. Definition 1 shows the algorithms in more detail. The client runs the algorithm Enc and encrypts the plaintext documents and the corresponding keywords before uploading them to the server. Enc outputs an encrypted database EDB, which is sent to the server. QueryGen, run by the user, requires a keyword and outputs a query token that can be sent to the server. The function Search is a deterministic algorithm that is executed by the server. A query q is sent to the server; the server takes the encrypted database EDB and returns the corresponding identifiers of the files $EDB(q)$. After it has retrieved the file identifiers, the user has to do another interaction with the server to retrieve the actual files.

Definition 1 (Searchable Encryption).

- $\mathsf{Enc}(K, F)$: *the encryption algorithm takes a master key K, and a document set $F = \{F_1, \cdots, F_n\}$ as input and outputs the encrypted database $EDB := \{\mathsf{Enc}_K(F_1), \cdots, \mathsf{Enc}_K(F_n)\}$;*
- $\mathsf{QueryGen}(w)$*: the query generation algorithm takes a keyword w as input and outputs a query token q;*
- $\mathsf{Search}(q, EDB)$*: the search algorithm takes a query q and the encrypted database EDB as input and outputs a subset of the encrypted database EDB, whose plaintext contains the keyword corresponding to the query q.*

Leakage. A query and the server response are considered the `access pattern`. The documents passed over the channel have their volume; this information is considered the `volume pattern`. In Section 3.1, we will explain the leakage in more detail.

2.2 Notation

In the VAL-Attack, we have m' keywords (w) and m queries (q), and n' leaked-documents and n server documents, denoted as d_i and ed_i, respectively; for a single document, similarly for w_i and q_i. Note w_i may not be the underlying keyword for query q_i, equal for d_i and ed_i. The notations are given in Table 1.

Table 1: Notation Summary

F	Plaintext document set, $F = \{d_1, ..., d_n\}$	F'	Leaked document set, $F' = \{d_1, ..., d_{n'}\}$
E	Server document set, $E = \{ed_1, ..., ed_n\}$	W	Keyword universe, $W = \{w_1, ..., w_m\}$
W'	Leaked keyword set, $W' = \{w_1, ..., w_{m'}\}$	Q	Query set, $Q = \{q_1, ..., q_m\}$
A	$m' \times n'$ matrix of leaked documents	B	$m \times n$ matrix of server documents
M'	$n' \times n'$ co-occurrence matrix of F'	M	$n \times n$ co-occurrence matrix of E
v_i	Volume (bit size) of document i	$\|d_i\|$	Number of keywords in document i
C	Set of matched documents	R	Set of matched queries

3 Models

In an ideal situation, there is no information leaked from the encrypted database, the queries sent, or the database setup. Unfortunately, such a scheme is not practical in real life as it costs substantial performance overheads [16]. The attacker and the leakage are two concerns in SE schemes, and we will discuss them both in the following sections, as they can vary in different aspects.

3.1 Leakage Model

Leakage is what we define as information that is (unintentionally) shared with the outer world. In our model, the attacker can intercept everything sent from and to the server. The attacker can intercept a query that a user sends to the server and the response from the server. It then knows which document identifiers correspond to which query. This *query → document identifier* response is what we call the `access pattern`. As discussed earlier, we assume the leakage level is L2 [8], where the attacker does not know the frequency or the position of the queried keywords in the document response. The `volume pattern` is leakage that tells the size of the document. It is relevant to all response leaking encryption schemes [6, 9, 11, 13, 20, 21] and ORAM-based SE schemes [26]. The leakage is defined as [4]:

Definition 2 (`access pattern`). *The function access pattern (AP)* $= (AP_{k,t})_{k,t\in\mathbb{N}}$: $F(k) \times W^t(k) \to [2^{[n]}]^t$, *such that* $AP_{k,t}(D, w_1, ..., w_t) = D(w_1), ..., D(w_t)$.

Definition 3 (`volume pattern`). *The function volume pattern (Vol)* $= (Vol_{k,t})_{k,t\in\mathbb{N}}$: $F(k)\times W^t(k) \to \mathbb{N}^t$, *such that* $Vol_{k,t}(D, w_1, ..., w_n) = ((|d|_w)_{d\in D(w_1)}, ..., (|d|_w)_{d\in D(w_n)})$ *Where* $|\cdot|_w$ *represents the volume in bytes.*

3.2 Attack Model

The attacker in SE schemes can be a malicious server that stores encrypted data. Since the server is honest-but-curious [4], it will follow the encryption protocol

but wants to learn as much as possible. Therefore, the attacker is passive but still eager to learn about the content present on the server. Our attacker has access to some leaked plaintext documents, keeps track of the `access and volume pattern` and tries to reveal the underlying server data. Figure 1 shows a visualization of our attack model. We assume that the attacker has access to all the queries and responses used in the SE scheme. This number of queries is realistic because if one waits long enough, all the queries and results will eventually be sent over the user-server channel. The technical framework delineates the LEAP, $\text{Subgraph}_{\text{VL}}$ and our designed attack.

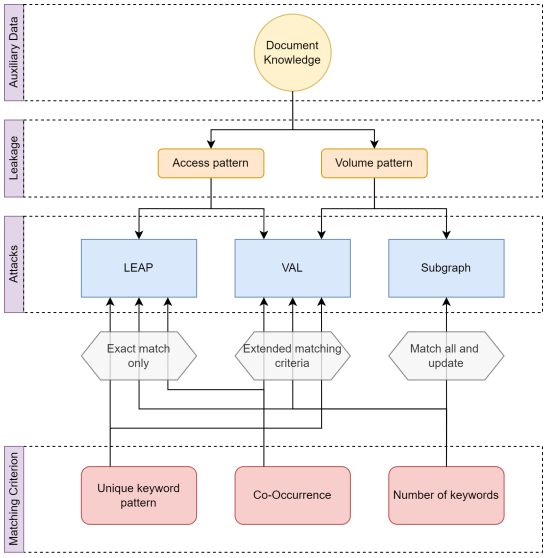


Figure 1: Technical Framework of Existing Attacks

The attacker in our model has access to some unencrypted files stored on the server. This access can be feasible because of a security breach at the setup phase of the scheme, where the adversary can access the revealed files. Another scenario is if a user wants to transfer all of his e-mails from his unencrypted mail storage to an SE storage server. The server can now access all the original mail files, but new documents will come as new e-mails arrive. Therefore, the adversary has partial knowledge about the encrypted data present on the server. The attacker has no access to any existing query to keyword matches and only knows the keywords present in the leaked files. With this information, the attacker wants to match as many encrypted document identifiers to leaked documents and queries to keywords such that he can understand what content is stored on the server.

The passive attacker is less potent than an active attacker, who can upload documents, with chosen keywords, to the server to match queries to keywords

[38]. Furthermore, the attacker has no access to the encryption or decryption oracle. Because the attacker relies on the `access and volume pattern` countermeasures that hide these patterns will reduce the attack performance.

4 The Proposed Attack

4.1 Main Idea

At a high level, our attack is built from the LEAP attack [28] by elevating the keyword matching metric to increase the number of keyword matches. Furthermore, each document is labelled with its document volume and number of keywords, and VAL-attack matches using the uniqueness of this label, improving the recovery rate. We first extend the matching technique from LEAP. The approach does not consist of only checking within the matched documents but also keeping track of the occurrence in the unmatched files. This method results in more recovered keywords for the improvement of LEAP that provides a way to match rows that do not uniquely occur in the matched files.

We expand the attack by exploiting the `volume pattern` since the document size is also leaked from response leaking encryption schemes, as described in Section 3.1. We can extend the comprehensive attack by matching documents based on the `volume pattern`.

Our new attack fully explores the leakage information and matches almost all leaked documents. We increase the keyword matches with the maximal file matches to provide excellent performance.

4.2 Leaked Knowledge

The server stores all the documents in the scheme. There are a total of n plaintext files denoted as the set $F = \{d_1, ..., d_n\}$, with in total m keywords, denoted as the set $W = \{w_1, ..., w_m\}$. We assume the attacker can access:

- The total number of leaked files (i.e. plaintext files) is n' with in total m' keywords. Suppose $F' = \{d_1, ..., d_{n'}\}$ is the set of documents known to the attacker and $W' = \{w_1, ..., w_{m'}\}$ is the corresponding set of keywords that are contained in F'. Note that $n' \leq n$ and $m' \leq m$.

- The set of encrypted files, denoted as, $E = \{ed_1, ..., ed_n\}$ and corresponding query tokens, $Q = \{q_1, ..., q_m\}$ with underlying keyword set W.

- The volume of each server observed document or leaked file is denoted as v_x for document d_x or server document ed_x. The number of keywords or tokens is represented as the size of the document $|d_x|$ or $|ed_x|$ for the same documents, respectively.

The attacker can construct an $m' \times n'$ binary matrix A, representing the leaked documents and their corresponding keywords. $A[d_x][w_y] = 1$ iff. keyword w_y occurs in document d_x. The dot product of A is denoted as the symmetric $n' \times n'$ matrix M', whose entry is the number of keywords that are contained in both document d_x and document d_y. We give an example of the matrices with known documents in Figure 6 (Appendix A).

After observing the server's files and query tokens, the attacker can construct an $m \times n$ binary matrix B, representing the encrypted files and related query tokens. $B[ed_x][q_y] = 1$ iff. query q_y retrieved document ed_x. The dot product of B is denoted as the symmetric $n \times n$ matrix M, whose entry is the number of query tokens that retrieve files ed_x and ed_y from the server. We give an example of the matrices with observed encrypted documents in Figure 7 (Appendix A).

4.3 Our Design

The basis of the attack is to recursively find row and column mappings between the two created matrices, A and B, where a row mapping represents the underlying keyword of a query sent to the server, and a column mapping indicates the match between a server document identifier and a leaked plaintext file. Note that each leaked document is still present on the server, meaning that $n' \leq n$ and there is a matching column in B for each column in A. Similarly to the rows, each known keyword corresponds to a query, so $m' \leq m$ as we could know all the keywords, but we do not know for sure. In theory, there is a correct row mapping for each row in A to a row in B. The goal of the VAL-Attack is to find as many correct mappings as possible.

We divide the process of finding as many matches as possible into several steps. The first step is to prepare the matrices for the rest of the process. The algorithm then maps columns based on unique column-sum, as they used in the Count attack [8], but instead of using it on keywords, we try to match documents here. Another step is matching documents based on unique volume and the number of keywords or tokens. As this combination can be a unique pattern, we can match many documents in this step. The matrices M and M' are used to match documents based on co-occurrence. Eventually, we can pair keywords on unique occurrences in the matched documents when several documents are matched. This technique is used in the Count attack [8], but we 'simulate' our own 100% knowledge here. With the matched keywords, we can find more documents, as these will give unique rows in matrices A and B that can be matched. We will introduce these functions in detail in the following paragraphs.

Initialization. First, we initialize the algorithm by creating two empty dictionaries, to which we eventually add the correct matches. We create one dictionary for documents and the other for the matched keywords, C (for column) and R (for row). Next, as we want to find unique rows in the matrices A and B, we must extend matrix A. It could be possible that not all underlying keywords are known beforehand, in which case $n' < n$, and we have to extend matrix A to find equal columns. Therefore we extend matrix A to an $m \times n'$ matrix that has the first m' rows equal to the original matrix A and the following $m - m'$ rows

of all 0s. See Figure 8 (Appendix A) for an example. The set $\{w_{m'+1}, ..., w_m\}$ represents the keywords that do not appear in the leaked document set F'.

Number of keywords. Now that the number of rows in A and B are equal, we can find unique column-sums to match documents. This unique sum indicates that a document has a unique number of keywords and can thus be matched based on this unique factor. Similar to the technique in the Count attack [8], we sum the columns, here representing the keywords in A and B. The unique columns in B can be matched to columns in A, as they have to be unique in A as well. If a column$_j$-sum of B is unique and column$_{j'}$-sum of A exists, we can match documents ed_j and $d_{j'}$ because they have the same unique number of keywords.

Volume and keyword pattern. The next step is matching documents based on volume and keyword pattern. If there is a server document ed_j with a unique combination of volume v_j and number of tokens $|ed_j|$ and there is a document $d_{j'}$ with the same combination, we can match document ed_j to $d_{j'}$. However, if multiple server documents have the same pattern, we need to check for unique columns with the already matched keywords between these files. Initially, we will have no matched keywords, but we will rerun this step later in the process. Figure 2 shows a concrete example, and Algorithm 1 describes our method.

Figure 2: Document matching on volume and number of keywords. Given multiple candidates, match on a unique column with the already matched keywords.

(a) Multiple documents with the same pattern of volume and number of keywords/tokens.

Leaked files	$\cdots$	d_4	d_6	d_8	$\cdots$	$d_{n'}$
Volume	$\cdots$	120	120	120	$\cdots$	120
#Keywords	$\cdots$	15	15	15	$\cdots$	18

Server files	$\cdots$	ed_6	ed_9	ed_{10}	$\cdots$	ed_n
Volume	$\cdots$	120	120	120	$\cdots$	150
#Tokens	$\cdots$	20	15	15	$\cdots$	15

(b) With the already matched keywords, create unique columns to match documents. Here d_6 and ed_8 can be matched, as well as d_9 and ed_{15}.

$$
\begin{array}{c} A_{CR} \\ w_2 \\ w_3 \\ w_5 \\ \vdots \\ w_t \end{array}
\begin{array}{c}
\begin{array}{cccc} d_4 & d_6 & d_8 & d_9 \end{array} \\
\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \vdots & \vdots & \vdots & \\ 1 & 1 & 1 & 0 \end{pmatrix}
\end{array}
\qquad
\begin{array}{c} B_{CR} \\ q_1 \\ q_3 \\ q_{15} \\ \vdots \\ q_t \end{array}
\begin{array}{c}
\begin{array}{cccc} ed_8 & ed_9 & ed_{10} & ed_{15} \end{array} \\
\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \vdots & \vdots & \vdots & \\ 1 & 1 & 1 & 0 \end{pmatrix}
\end{array}
$$

Algorithm 1 matchByVolume

Input: $R, A\ (m \times n'), B\ (m \times n)$

$C' \leftarrow \{\}$
patterns $\leftarrow \{(v_j, |ed_j|)$ with volume v_j and #tokens $|ed_j|$ of document $ed_j\}$
for $p \in$ patterns **do**
    enc_docs $\leftarrow [ed_j$ with pattern $p]$
    if |enc_docs| $= 1$ **then**
        $ed_j \leftarrow$ enc_docs[0]
        $C'[ed_j] \leftarrow d_{j'}$ with pattern p
    else if $|R| > 0$ **then**
        docs $\leftarrow [d_{j'}$ with pattern $p]$
        $B_{CR} \leftarrow$ enc_docs columns and R rows of B
        $A_{CR} \leftarrow$ docs columns and R rows of A
        for column$_j \in B_{CR}$ that is unique **do**
            $C'[ed_j] \leftarrow d_{j'}$ with column$_j \in A_{CR}$
return C'

Co-occurrence. When having some matched documents, we can use the co-occurrence matrices M and M' to find other document matches. For an unmatched server document ed_x, we can try an unmatched leaked document d_y. If $M_{x,k}$ and $M'_{y,k'}$ are equal for each matched document pair $(ed_k, d_{k'})$ and no other document $d_{y'}$ has the same results, then we have a new document match between ed_x and d_y. The algorithm for this step is shown in Algorithm 2.

Algorithm 2 coOccurrence

Input: $C, M\ (n \times n), M\ (n' \times n), A\ (m \times n'), B\ (m \times n)$

while C is increasing **do**
    for each $d_{j'} \notin C$ **do**
        sum$_{j'} \leftarrow$ column$_{j'}$-sum of A
        candidates $\leftarrow [ed_j \notin C$ where column$_j$-sum of $B =$ sum$_{j'}]$
        for $ed_j \in$ candidates **do**
            for $(ed_k, d_{k'}) \in C$ **do**
                if $M_{j,k} \neq M'_{j',k'}$ **then**
                    candidates $\leftarrow$ candidates $\setminus\ ed_j$
        if |candidates| $= 1$ **then**
            $ed_j \leftarrow$ candidates[0]
            $C[ed_j] \leftarrow d_{j'}$
return C

Keyword matching. We match keywords using the matched documents. To this end, we create matrices B_c and A_c by taking the columns of matched documents from matrices B and A. Note that these columns will be rearranged to the order of the matched documents, such that column B_{c_j} is equal to column $A_{c_{j'}}$ for document match $(ed_j, d_{j'})$. Matrices B_c and A_c are shaped $m \times t$ and

$m' \times t$, respectively, for t matched documents. We give the algorithm for this segment in Algorithm 3 and a simple example in Figure 9 (Appendix A).

Algorithm 3 matchKeywords

Input: $C, A\ (m \times n'), B\ (m \times n)$
$R \leftarrow \{\}$
$B_c \leftarrow C$ columns of B
$A_c \leftarrow C$ columns of A
for $\text{row}_i \in B_c$ **do**
  if row_i is unique in B_c **then**
    if $\text{row}_{i'} \in A_c = \text{row}_i$ **then**
      $R[q_i] \leftarrow w_{i'}$
  else ▷ Match based on occurrence in (server) files
    docs $\leftarrow [i' \in A_c$ where $A_c[i'] = \text{row}_i]$
    e_docs $\leftarrow [j \in B_c$ where $B_c[j] = \text{row}_i]$
    $B_x \leftarrow$ sum of rows in B[e_docs], sort descending
    $A_x \leftarrow$ sum of rows in A[docs], sort descending
    if $A_x[1] < A_x[0] > B_x[1]$ **then**
      $i_x \leftarrow$ index of $B_x[0] \in$ e_docs
      $j_x \leftarrow$ index of $A_x[0] \in$ docs
      $R[q_{i_x}] \leftarrow w_{j_x}$
return R

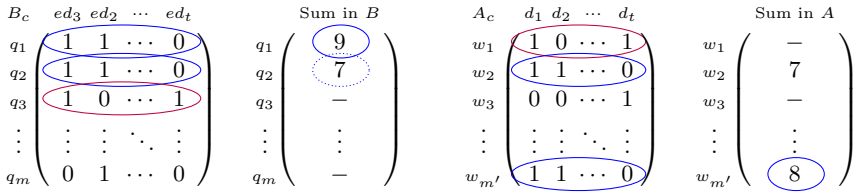
A row in the matrices indicates in which documents a query or keyword appears. If a row_i in B_c is unique, row_i is also unique in B, similar to A_c and A. Hence, for row_i in B_c, that is unique, and if there is an equal row_j in A_c, we can conclude that the underlying keyword of q_i is w_j.

Nevertheless, if row_i is not unique in B_c, we can still try to match the keyword to a query. A keyword can occur more often in the unmatched documents than their query candidates; thus, they will not be valid candidates. We create a list B_x with for each similar row_i in B_c the sum of row_i in B; similar for list A_x, with row_i in A_c and the sum of row_i in A. Next, if the highest value of A_x, which is A_{x_j}, is higher than the second-highest value of A_x and B_x, referred to as $A_{x_{j'}}$ and $B_{x_{i'}}$, respectively, we can conclude that keyword w_j corresponds to the highest value of B_x, i.e. B_{x_j}, which means that w_j matches with q_j. We put an example in Figure 3 (Appendix A).

Keyword order in documents. We aim to find more documents based on unique columns given the query and keyword mappings. First, we create matrices B_r and A_r with the rows from the matched keywords in R. B_r and A_r are submatrices of B and A, respectively, with rearranged row order. B_r and A_r are shaped $t \times n$ and $t \times n'$, respectively, for t matched files. Note that we show an example in Figure 10 (Appendix A). If any column_j of B_r is unique and there exists an equal $\text{column}_{j'}$ in A_r, we know that ed_j is a match with $d_{j'}$.

The next step is to set the rows of the matched keywords to 0 in B and A. Then, similar to before, we use the technique from the Count attack [8]; we sum

Figure 3: Example of matching keywords in matched documents. Query q_3 has a unique row and therefore matches with keyword w_1. Queries q_1, q_2 and keywords w_2, $w_{m'}$ have the same row. However, keyword $w_{m'}$ occurs more often in A than w_2 and query q_2 in B. Therefore q_1 matches with $w_{m'}$.



the updated columns in A and B and try to match the unique columns in B to columns in A. If a column$_j$-sum of B is unique and an equal column$_{j'}$-sum in A exists, we can match document ed_j and $d_{j'}$.

The complete algorithm of our VAL-attack is in Algorithm 4, Appendix B.

4.4 Countermeasure Discussions

Many countermeasures have been proposed to mitigate leakage-abuse attacks [8, 12, 18, 32, 36]. The main approaches are padding and obfuscation.

The IKK attack [18] and the Count attack [8] discussed a padding countermeasure, where they proposed a technique to add fake document identifiers to a query response. These false positives could then later be removed by the user. This technique is also called *Hiding the Access Pattern* [22]. The LEAP attack [28] crucially relies on the number of keywords per document, and if the scheme adds fake query tokens to documents on the server, they will not be able to match with their known documents. However, they also proposed a technique that describes a modified attack that is better resistant to padding. This technique, which is also used in the Count attack [8], uses a window to match keywords. However, this will give false positives and thus reduce the performance of the attack. The Subgraph$_{\text{VL}}$ attack [4] depends on the volume of each document. Volume-hiding techniques from Kamara et al. [19] reduce the attack's performance, but it is unclear if they completely mitigate the attack.

A padding technique that will make all documents of the same size, i.e. adding padding characters, will reduce the uniqueness in matching based on the volume of a document. If the padding technique can be extended such that false positives are added to the `access pattern`, we have no unique factor in matching documents based on the number of keywords per file. Therefore, a combination of the two may decrease the performance of the VAL-Attack.

5 Evaluation

We set up the experiments to run the proposed attack to evaluate the performance. Furthermore, we compare the file and query recovery of the VAL-Attack with the results from the LEAP [28] and Subgraph$_{\text{VL}}$ attack [4]. We notice that the LEAP attack is not resistant to the test countermeasures, and Blackstone et al. [4] argue for their Subgraph$_{\text{VL}}$ attack that it is not clear whether volume-hiding constructions may mitigate the attack altogether. From this perspective, we only discuss the performance of VAL-Attack against countermeasures in Section 5.3. It would be an interesting problem to test the countermeasures on the LEAP and Subgraph$_{\text{VL}}$ attacks, but that is orthogonal to the focus of this work.

5.1 Experimental Setup

We used the Enron dataset [35] to run our comparison experiments. We leveraged the *_sent_mail* folder from each of the 150 users from this dataset, resulting in 30,109 e-mails from the Enron corporation. The second dataset we used is the Lucene mailing list [2]; we specifically chose the "java-user" mailing list from the Lucene project for 2002-2011. This dataset contains 50,667 documents. Finally, we did the tests on a collection of Wikipedia articles. We extracted plaintext documents from Wikipedia in April 2022 using a simple wiki dump[1] and used the tool from David Shapiro [33] to extract plaintext data, resulting in 204,737 files. The proposed attack requires matrices of size $n \times n$; therefore, we limited the number of Wikipedia files to 50,000. We used Python 3.9 to implement the experiments and run them on machines with different computing powers to improve running speed.

To properly leverage those data from the datasets for the experiments, we first extracted the information of the Enron and Lucene e-mail content. The title's keywords, the names of the recipients or other information present in the e-mail header were not used for queries. NLTK corpus [3] in Python is used to get a list of English vocabulary and stopwords. We removed the stopwords with that tool and stemmed the remaining words using Porter Stemmer [30]. We further selected the most frequent keywords to build the keyword set for each document. For each dataset, we extracted 5,000 words as the keyword set W. Within the Lucene e-mails, we removed the unsubscribe signature because it appears in every e-mail.

The server files (n) and keywords (m) are all files from the dataset and 5,000 keywords, respectively. The leakage percentage determines the number of files (m') known to the user. The attacker only knows the keywords (n') leaked with these known documents. The server files and queries construct a matrix B of size $m \times n$; while the matrix A of size $m' \times n'$ is constructed with the leaked files. We took the dot product for both matrices and created the matrices M and M',

[1] `https://dumps.wikimedia.org/simplewiki/20220401/simplewiki-20220401-pages-meta-current.xml.bz2`

respectively. Note that the source code to simulate the attack and obtain our results is available here: `https://github.com/StevenL98/VAL-Attack`.

Because our attack does not create false positives, the accuracy of the retrieved files and keywords is always 100%. Therefore, we calculated the percentage of files and keywords retrieved from the total leaked files and keywords. Each experiment is run 20 times to calculate an average over the simulations. We chosen 0.1%, 0.5%, 1%, 5%, 10%, 30% as leakage percentages. The lower percentages are chosen to compare with the results from the LEAP attack [28], and the maximum of 30% is chosen because of the stagnation in query recovery.

5.2 Experimental Results

The results tested with the different datasets are given in Figures 4a and 4b, which show the number and percentage of files and keywords recovered by our attack. The solid line is the average recovery in those plots, and the shades are the error rate over the 20 runs.

We can see that the VAL-attack recovers almost 98% of the known files and above 93% of the keywords available to the attacker once the leakage percentage reaches 5%. These percentages are based on the leaked documents. When 10% of the Enron database is leaked, which is 3,010 files with 4,962 keywords, we can match 2,950 files and 4,909 queries, corresponding to 98% and 99%, respectively. The Lucene dataset is more extensive than Enron, and therefore we have more files available for each leakage percentage. One may see that we can recover around 99% of the leaked files and a rising number of queries, starting from 40% of the available keyword set. The Wikipedia dataset does not consist of e-mails but rather lengthy article texts. We reveal fewer files than the e-mail datasets, but we recover just below 90% of the leaked files, and from 1% leakage, we recover more available keywords than the other datasets. This difference is probably because of the number of keywords per file since the most frequent keywords are chosen.

With the technique we proposed, one can match leaked documents to server documents for almost all leaked documents. Next, the algorithm will compute the underlying keywords to the queries. It is up to the attacker to allow false positives and improve the number of (possible) correctly matched keywords, but we decided not to include it.

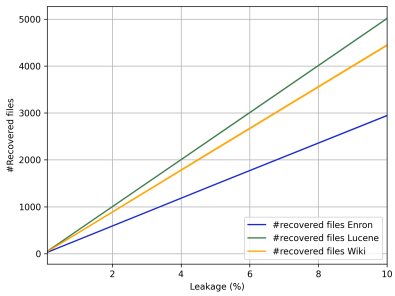
Comparison. We compare the performance of VAL-Attack to two attacks with the Enron dataset. One is the LEAP attack [28] (which is our cornerstone), while the other is the $\text{Subgraph}_{\text{VL}}$ attack [4] (as they use the volume pattern as leakage). We divide the comparison into two parts: the first is for recovering files, and the second is for queries recovery.

As shown in Figure 5, we recover more files than the LEAP attack, and the gap in files recovered expands as the leakage percentage increases, see Figure 5a.i. The difference in the percentage of files recovered is stable, as VAL-Attack recovers about eight percentage points more files than the LEAP attack, see Figure 5a.ii. The comparison outcome for recovered queries can be seen in Figure 5b.

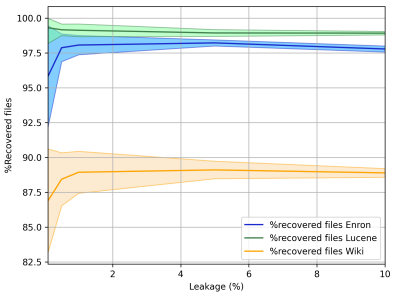
Figure 4: Results for VAL-Attack, with the actual number and the percentage of recovered files and queries for different leakage percentages.

(a) Exact number and relative percentage of recovered files

(a.i) #Files recovered

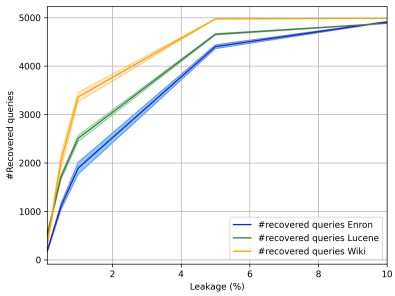


(a.ii) %Files recovered

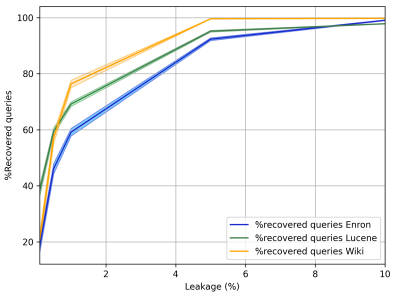


(b) Exact number and relative percentage of recovered queries

(b.i) #Queries recovered



(b.ii) %Queries recovered



We can see that the recovered queries do not show a significant difference with the LEAP attack as that attack performs outstandingly in query recovery. The most significant difference is around 5% leakage, where VAL-Attack retrieves around 100 queries more than the LEAP attack, which could influence a real-world application. Compared to the $\text{Subgraph}_{\text{VL}}$, we see in Figure 5b.ii that the combination of the `access pattern` and the `volume pattern` is a considerable improvement; we reveal about 60 percentage points more of the available queries.

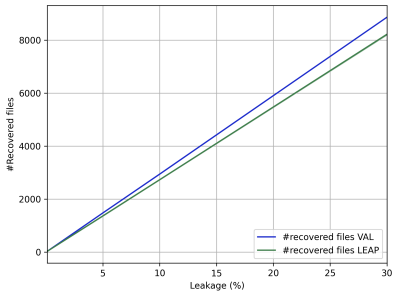
5.3 Countermeasure Performance

As discussed in Section 4.4, there are several options for countermeasures against attacks on SE schemes. Moreover, since our attack exploits both the `access and volume pattern`, countermeasures must mitigate both leakage patterns. The former can be mitigated by padding the server result, while the latter may be

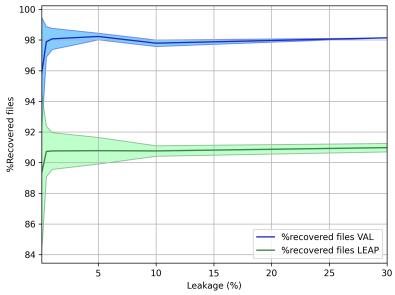
Figure 5: Comparison of VAL-Attack

(a) Comparison with LEAP attack [28] based on the number and percentages of files recovered

(a.i) #Files recovered

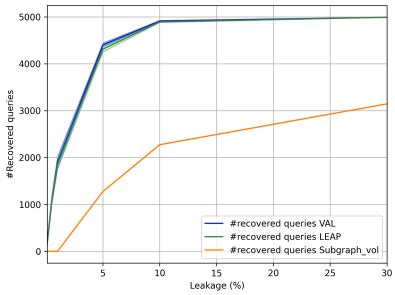


(a.ii) %Files recovered

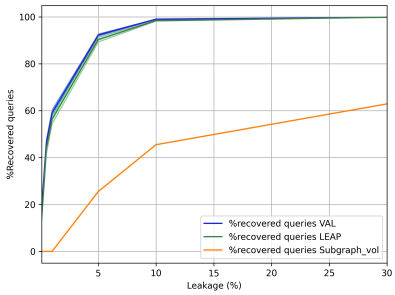


(b) Comparison with LEAP attack [28] and Subgraph$_{\text{VL}}$ attack [4] based on the number and percentages of queries recovered

(b.i) #Queries recovered



(b.ii) %Queries recovered



handled using volume-hiding techniques. However, these approaches may come with impractical side effects. Padding the server response requires more work on the client-side to filter out the false positives. This padding can cause storage and reading problems because the user has to wait for the program to filter out the correct results. The volume-hiding technique [19] may easily yield significant storage overhead and could therefore not be practical in reality. Luckily, Patel et al. [29] illustrated how to reduce this side effect whilst mitigating the attack.

It is possible to mitigate our attack theoretically by using a combination of padding and volume hiding techniques. We tested the VAL-attack's performance with padding, volume hiding and further a combination, but we did not examine by obfuscation due to hardware limitations.

We padded the server data using the technique described by Cash et al. [8]. Each query returned a multiplication of 500 server files, so if the original query returned 600 files, the server now returned 1,000. Padding is done by adding documents to the server response that do not contain the underlying keyword.

These documents can then later be filtered by the client, but will obfuscate the client's observation. We took the naïve approach from Kamara et al. [19] for volume hiding, where we padded each document to the same volume. By adding empty bytes to a document, it will grow in size. If done properly, all files will eventually have the same size that can not be distinguished from the actual size.

We ran the countermeasure experiments on the Enron and the Lucene dataset. We did not perform the test on the Wikipedia dataset, but we can predict that the countermeasures may affect the attack performance. We predict that a single countermeasure will not entirely reduce the attack effectiveness, but a combination may do.

Because of the exploitation of the two leakage patterns, we see in Table 2 that our attack can still recover files and underlying keywords against only a single countermeasure. Under a combination of padding and volume hiding, our attack cannot reveal any leaked file or keyword.

Table 2: Performance of VAL-Attack with countermeasures

Dataset		Enron						Lucene					
Counter-measure		Padding		Volume Hiding		Padding & Volume Hiding		Padding		Volume Hiding		Padding & Volume Hiding	
Files	0.1%	25	(83.7%)	27	(89.5%)	0	(0%)	45	(88.9%)	10	(28.4%)	0	(0%)
	0.5%	103	(68.4%)	137	(90.7%)	0	(0%)	191	(75.3%)	95	(37.4%)	0	(0%)
	1%	208	(69.0%)	274	(90.9%)	0	(0%)	381	(75.3%)	147	(28.9%)	0	(0%)
	5%	1,114	(74.0%)	1,365	(90.7%)	0	(0%)	2332	(92.0%)	2452	(96.8%)	0	(0%)
	10%	1,910	(63,4%)	2,736	(90.9%)	0	(0%)	4,073	(80.4%)	4,891	(96.5%)	0	(0%)
	30%	5,358	(59.0%)	8,219	(91.0%)	0	(0%)	10,343	(68.0%)	-[2]		0	(0%)
Queries	0.1%	94	(10.4%)	172	(14.8%)	0	(0%)	377	(27.7%)	153	(10.6%)	0	(0%)
	0.5%	433	(18.1%)	1,059	(43.3%)	0	(0%)	724	(25.3%)	663	(22.8%)	0	(0%)
	1%	414	(12.8%)	1,836	(56.3%)	0	(0%)	556	(15.3%)	748	(20.5%)	0	(0%)
	5%	53	(1.1%)	4,290	(89.9%)	0	(0%)	87	(1.8%)	4,659	(95.2%)	0	(0%)
	10%	11	(0.2%)	4,890	(98.4%)	0	(0%)	33	(0.7%)	4,872	(97.6%)	0	(0%)
	30%	1	(0.0%)	4,993	(99.9%)	0	(0%)	10	(0.2%)	-[2]		0	(0%)

[2] Did not run due to hardware limitations

Table 2 is read as follows: The number below the countermeasure is the exact number of retrieved files or queries, with the relative percentage between brackets. So for 0.1% leakage under the padding countermeasure, we revealed, on average, 25 files, which was 83.7% of the leaked files. Each experiment ran 20 times. Due to runtime and hardware limitations, we did not run the experiment with 30% leakage on the Lucene dataset. However, since we have the results for 10% leakage and the results for the Enron dataset, we can predict the outcome for 30%. Similar to the Enron dataset, the recovered data in Lucene increases as the leakage percentage grows. Therefore, we predict that 30% leakage results in the Lucene dataset is a bit higher than the 10% leakage.

5.4 Discussion on Experiments

Parameters. We used 5,000 high selectivity keywords, i.e. keywords that occur the most in the dataset. This number is chosen because a practical SE application will probably not have just a few search terms in a real-world scenario. Other attacks [4, 8, 18] have experimented with only 150 query tokens and 500 keywords, and we argue that this may not be realistic. Our attack can recover almost all underlying keywords for an experiment with 500 keywords because the number of files is equal, but a slight variation in keyword occurrence[3].

We cut the number of Wikipedia files to 50,000. We did this to better present the comparison with the Enron and Lucene datasets. The attack may also take longer to run when all Wikipedia files are considered. The results will also differ as the number of files leaked increases similarly. The percentage of files recovered will probably be the same because of keyword distribution among the files.

If we ran the experiments with a higher leakage percentage, the attack would eventually recover more files, as more are available, but we would not recover more keywords. As with 30% leakage, we see that we have recovered all 5,000 keywords.

Our attack performs without false positives, we did so because they would not improve the performance, and an attacker cannot better understand the data if he cannot rely on it. If we allowed the attack to return false positives, we would have 5,000 matches for underlying keywords, of which not all are correct. The attack performance will not change since we will only measure the correct matches, which we already did.

Attack comparison. In Figure 5a, we only compared our attack with the LEAP attack rather than the $\text{Subgraph}_{\text{VL}}$ attack. We did so because the latter does not reveal encrypted files and thus cannot be compared. If we choose to compare the attack to ours, we would have to rebuild their attack using their strategy, which is out of the scope of this work.

We used the Enron dataset to compare the VAL-Attack to the LEAP and the $\text{Subgraph}_{\text{VL}}$. In their work [4, 28], they used the Enron dataset to show their performance. If we used the Lucene or Wikipedia dataset instead to present the comparison, we would have no foundation in the literature to support our claim. A comparison of all the datasets would still show that our attack surpasses the attacks since, in theory, we exploit more.

We discussed other attacks, like the IKK and the Count attack, but we did not compare their performance with ours. While these attacks exploit the same leakage, we could still consider them. However, since LEAP is considered the most state-of-the-art attack and has already been compared with the other attacks in [28], we thus only have to compare the LEAP attack here. Accordingly, a comparison with all attacks would not affect the results and conclusion of this paper.

[3] Tested, but results not provided

6 Related Work

The Count attack [8] uses the number of files returned for the query as their matching technique; The $\text{Subgraph}_{\text{VL}}$ [4] matches keywords based on unique document volumes as if it is the response pattern, and the LEAP attack [28] uses techniques from previous attacks to match leaked documents and keywords with high accuracy.

Besides the attacks that exploit similar leakage to our proposed attack, we may also review those attacks that do not. An attack that leverages similar documents as auxiliary knowledge, called Shadow Nemesis, was proposed by Pouliot et al. [31]. They created a weighted graph matching problem in the attack and solved it using Path or Umeyama. Damie et al. [14] presented the Score attack, requiring similar documents, and they matched based on the frequency of keywords in the server and auxiliary documents. Both attacks use co-occurrence matrices to reveal underlying keywords. The Search attack by Liu et al. [25] matches based on the search pattern, i.e. the frequency pattern of queries sent to the server. Table 3 briefly compares the attacks based on leakage, auxiliary knowledge, false positives and exploiting techniques. The reviewed attacks described above are not mainly relevant to our proposed attack; thus, we did not put them in the comparison in Section 5.

Table 3: Comparison on Different Attacks. The lower part are those passive attacks with pre-known data compared with VAL-Attack. *Documents* in the auxiliary data column refers to leaked document knowledge, *queries* refers to leaked underlying keywords for query tokens, and *similar* refers to the use of similar documents instead of leaked documents.

Attack	Leakage	Auxiliary data	False positives	Exploited information
IKK [18]	Access pattern	Documents, queries	✓	Co-occurrence
Shadow Nemesis [31]	Access pattern	Similar	✓	Co-occurrence
Score [14]	Access pattern	Similar, queries	✓	Co-occurrence
Search14 [25]	Search pattern	Search frequency	✓	Query frequency
ZKP [38] (active)	Access pattern	All keywords	✗	-
Count [8]	Access pattern	Documents	✓	Co-occurrence, length
$\text{Subgraph}_{\text{VL}}$ [4]	Volume pattern	Documents	✓	Volume, length
LEAP [28]	Access pattern	Documents	✗	Co-occurrence, length
VAL-Attack	Access, volume pattern	Documents	✗	Volume, length, co-occurrence

7 Conclusion

We proposed the VAL-attack to improve the matching technique from the LEAP attack, leveraging the leakage from the `access pattern` and the `volume pattern` which is a combination that has not been exploited before. We showed that our attack provides excellent performance, and we compared it to the LEAP attack and the $\text{subgraph}_{\text{VL}}$ attack. The number of matched files is with more remarkable improvement than the number of queries recovered compared to the LEAP attack. The attack recovers around 98% of the leaked documents and above 90% for query recovery with very low leakage. Since the proposed attack uses both the document size and the response per query, it requires strong (and combined) countermeasures and thus, is more harmful than existing attacks.

References

[1] Abdalla, M., Bellare, M., Catalano, D., Kiltz, E., Kohno, T., Lange, T., Malone-Lee, J., Neven, G., Paillier, P., Shi, H.: Searchable encryption revisited: Consistency properties, relation to anonymous ibe, and extensions. In: CRYPTO. pp. 205–222

[2] Apache: Mail archieves of lucene (1999), https://mail-archives.apache.org/mod_mbox/#lucene
[3] Bird, S., Klein, E., Loper, E.: Natural language processing with Python: analyzing text with the natural language toolkit. " O'Reilly Media, Inc." (2009)
[4] Blackstone, L., Kamara, S., Moataz, T.: Revisiting leakage abuse attacks (01 2020)
[5] Boneh, D., Di Crescenzo, G., Ostrovsky, R., Persiano, G.: Public key encryption with keyword search. In: CRYPTO 2004. vol. 3027, pp. 506–522
[6] Bost, R.: Sophos - forward secure searchable encryption. Cryptology ePrint Archive, Report 2016/728 (2016), https://ia.cr/2016/728
[7] Bost, R., Minaud, B., Ohrimenko, O.: Forward and backward private searchable encryption from constrained cryptographic primitives. p. 1465–1482. CCS '17
[8] Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. Cryptology ePrint Archive, Report 2016/718 (2016)
[9] Cash, D., Jaeger, J., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.C., Steiner, M.: Dynamic searchable encryption in very-large databases: Data structures and implementation (2014)
[10] Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: CRYPTO '13. pp. 353–373. Springer
[11] Chase, M., Kamara, S.: Structured encryption and controlled disclosure. Cryptology ePrint Archive, Report 2011/010 (2011), https://ia.cr/2011/010
[12] Chen, G., Lai, T.H., Reiter, M.K., Zhang, Y.: Differentially private access patterns for searchable symmetric encryption. In: IEEE INFOCOM '18. pp. 810–818
[13] Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: Improved definitions and efficient constructions. In: CCS '06. pp. 79–88
[14] Damie, M., Hahn, F., Peter, A.: A highly accurate query-recovery attack against searchable encryption using non-indexed documents. In: USENIX '21. pp. 143–160
[15] Demertzis, I., Papamanthou, C.: Fast searchable encryption with tunable locality. p. 1053–1067. ACM SIGMOD '17
[16] Gui, Z., Paterson, K.G., Patranabis, S.: Rethinking searchable symmetric encryption (2021)
[17] He, W., Akhawe, D., Jain, S., Shi, E., Song, D.: Shadowcrypt: Encrypted web applications for everyone. In: ACM CCS '14. p. 1028–1039
[18] Islam, M.S., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: Ramification, attack and mitigation (2012)
[19] Kamara, S., Moataz, T.: Computationally volume-hiding structured encryption. In: CRYPTO '19. pp. 183–213

[20] Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: FC '13. pp. 258–274
[21] Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: ACM CCS '12. p. 965–976
[22] Kortekaas, Y.: Access pattern hiding aggregation over encrypted databases (2020)
[23] Lau, B., Chung, S., Song, C., Jang, Y., Lee, W., Boldyreva, A.: Mimesis aegis: A mimicry privacy shield a system's approach to data privacy on public cloud. In: USENIX Security '14. p. 33–48
[24] Li, J., Sun, J.: A practical searchable symmetric encryption scheme for smart grid data. ICC '19 pp. 1–6
[25] Liu, C., Zhu, L., Wang, M., an Tan, Y.: Search pattern leakage in searchable encryption: Attacks and new construction (2013)
[26] Ma, Q., Zhang, J., Peng, Y., Zhang, W., Qiao, D.: Se-oram: A storage-efficient oblivious ram for privacy-preserving access to cloud storage. In: CSCloud '16. pp. 20–25
[27] Minaud, B., Reichle1, M.: Dynamic local searchable symmetric encryption (1 2022)
[28] Ning, J., Huang, X., Poh, G.S., Yuan, J., Li, Y., Weng, J., Deng, R.H.: Leap: Leakage-abuse attack on efficiently deployable, efficiently searchable encryption with partially known dataset. In: ACM CCS '21. p. 2307–2320
[29] Patel, S., Persiano, G., Yeo, K., Yung, M.: Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In: ACM CCS '19. p. 79–93
[30] Porter, M.F.: An algorithm for suffix stripping. Program **40**, 211–218 (1980)
[31] Pouliot, D., Wright, C.V.: The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. ACM CCS '16
[32] Shang, Z., Oya, S., Peter, A., Kerschbaum, F.: Obfuscated access and search patterns in searchable encryption (2021)
[33] Shapiro, D.: Convert wikipedia database dumps into plaintext files (2021), `https://github.com/daveshap/PlainTextWikipedia`
[34] Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: IEEE S&P '00. pp. 44–55
[35] William W. Cohen, MLD, C.: Enron email datasets (2015), `https://www.cs.cmu.edu/~enron/`
[36] Xu, L., Yuan, X., Wang, C., Wang, Q., Xu, C.: Hardening database padding for searchable encryption. In: IEEE INFOCOM '19. pp. 2503–2511
[37] Zhang, R., Imai, H.: Combining Public Key Encryption with Keyword Search and Public Key Encryption. IEICE Trans. Inf. Syst. **92**(5), 888–896
[38] Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: The power of File-Injection attacks on searchable encryption. In: USENIX '16. pp. 707–720
[39] Zheng, Q., Xu, S., Ateniese, G.: Vabks: Verifiable attribute-based keyword search over outsourced encrypted data. In: INFOCOM '14. pp. 522–530

A Examples of Matrices

Figure 6: Matrix A and M' Example

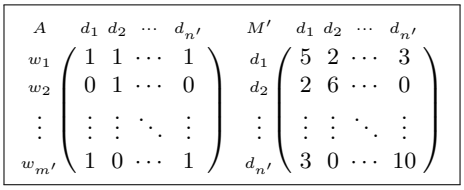


Figure 7: Matrix B and M Example

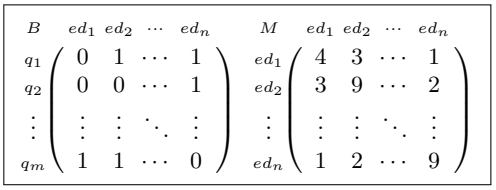


Figure 8: An Example of Extended Matrix A

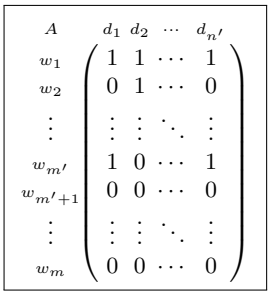


Figure 9: Matrix A_c and B_c Example

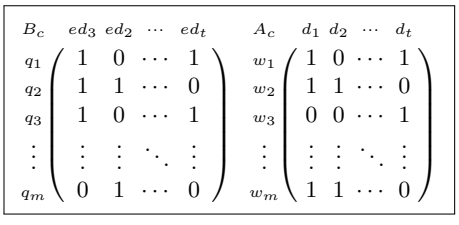
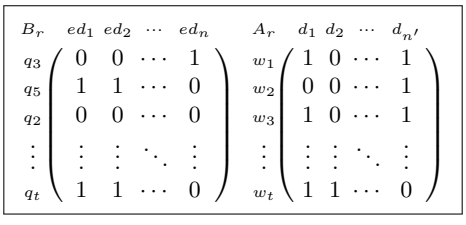


Figure 10: Matrix A_r and B_r Example



B VAL-Attack Algorithm

Algorithm 4 VAL-Attack

Input: $A\ (m' \times n'), B\ (m \times n),\ M'\ (n' \times n'), M\ (n \times n)$

$C = R \leftarrow \{\}$ ▷ Initialization
$A \leftarrow A$ where rows extended with 0's (m x n')
$\text{vector}_A = \text{vector}_B \leftarrow [\,]$ ▷ Match documents with unique #keywords
for $j \in [n]$ **do**
    $\text{vector}_B[j] \leftarrow$ sum of column B_j
for $j' \in [n']$ **do**
    $\text{vector}_A[j'] \leftarrow$ sum of column $A_{j'}$
for $\text{vector}_{B_j} \in \text{vector}_B$ that is unique **do**
    if $\text{vector}_{A_{j'}} == \text{vector}_{B_j}$ **then**
        $C[ed_j] \leftarrow d_{j'}$
$C \leftarrow C \cup \text{MATCHBYVOLUME}(R, A, B)$ ▷ Match documents with unique volume
$C \leftarrow C \cup \text{COOCCURRENCE}(C, M, M', A, B)$ ▷ Match docs with co-occurrence
$C \leftarrow C \cup \text{MATCHBYVOLUME}(R, A, B)$
while R or C is increasing **do**
    $R \leftarrow R \cup \text{MATCHKEYWORDS}(C, A, B)$ ▷ Match keywords in matched docs
    $B_r \leftarrow$ R rows of B ▷ Match documents with unique keyword order
    $A_r \leftarrow$ R rows of A
    for $\text{column}_j \in B_r$ that is unique **do**
        if $\text{column}_{j'} \in A_r == \text{column}_j$ **then**
            $C[ed_j] \leftarrow d_{j'}$
    $C \leftarrow C \cup \text{MATCHBYVOLUME}(R, A, B)$
    row $B_j \leftarrow 0$ if $q_j \in R$ ▷ Match documents with unique #keywords
    row $A_{j'} \leftarrow 0$ if $k_{j'} \in R$
    for $j \in [n]$ where $ed_j \notin C$ **do**
        $\text{vector}_B[j] \leftarrow$ sum of column B_j
    for $j' \in [n']$ where $d_{j'} \notin C$ **do**
        $\text{vector}_A[j'] \leftarrow$ sum of column $A_{j'}$
    for $\text{vector}_{B_j} \in \text{vector}_B$ that is unique and $ed_j \notin C$ **do**
        if $\text{vector}_{A'_j} == \text{vector}_{B_j}$ and $d_{j'} \notin C$ **then**
            $C[ed_j] \leftarrow d_{j'}$
    $C \leftarrow C \cup \text{COOCCURRENCE}(C, M, M', A, B)$ ▷ Match docs with co-occurrence
return R, C