CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Towards Higher-Dimensional Attacks on Searchable Symmetric Encryption

Security Analysis of TWo-IN-one-SSE via Active and Passive Attacks on Boolean Queries

Master's thesis in Computer science and engineering

ZINAN MA

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Towards Higher-Dimensional Attacks on Searchable Symmetric Encryption

Security Analysis of TWo-IN-one-SSE via Active and Passive Attacks on Boolean Queries

ZINAN MA

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Towards Higher-Dimensional Attacks on Searchable Symmetric Encryption
Security Analysis of TWo-IN-one-SSE via Active and Passive Attacks on Boolean Queries
ZINAN MA

Supervisor: Christoph Egger, Computer Science and Engineering
Examiner: Rhouma Rhouma, Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Towards Higher-Dimensional Attacks on Searchable Symmetric Encryption
Security Analysis of TWo-IN-one-SSE via Active and Passive Attacks on Boolean Queries
ZINAN MA
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In recent years, Searchable Symmetric Encryption (SSE) has become a crucial tool for securely storing and querying encrypted documents in the cloud. However, third-party cloud servers may act as honest-but-curious adversaries, attempting to infer sensitive information from encrypted queries or documents. Notably, even SSE schemes that are provably secure can remain vulnerable to practical attacks.

Such attacks typically exploit the schemes leakage profile in combination with auxiliary information about the encrypted dataset. The effectiveness of an attack depends heavily on both the extent of leakage and the quality of auxiliary knowledge available to the adversary.

TWINSSE is a recently proposed SSE scheme that supports conjunctive and disjunctive Boolean queries, but its security guarantees are not fully understood. In this work, we conduct a comprehensive security analysis of TWINSSE under both passive and active adversarial models. We first examine **passive attack paths** under various search pattern leakage assumptions. We then propose two **active attack strategies** adapted to different levels of adversarial knowledge. Finally, we present a **statistical attack** targeting conjunctive queries and demonstrate its practical feasibility via experiments.

keywords: Structured Symmetric Encryption; Leakage; Boolean Query.

Acknowledgements

I would like to express my deepest gratitude to Christoph Egger for his exceptionally patient supervision. His guidance has been invaluable in shaping my thinking and supporting my work.

My sincere thanks also go to Rhouma Rhouma, who served as my examiner and actively engaged with my project. His insightful suggestions and surprising perspectives greatly enriched my research. Beyond his role as an examiner, his genuine care and encouragement have been a tremendous motivation for me.

Lastly, I am forever grateful to my family: Xueye Ma, Minyi Ma, and Miaohua Zheng, for their unconditional support, both financially and in daily life, throughout my masters studies. Without them, this journey would not have been possible.

May curiosity and the pursuit of knowledge continue to guide me in the years to come, may the peace, joy and health be with us.

Zinan Ma, Gothenburg, 2025-06-25

Contents

List of Figures

List of Tables

1

Introduction

Cloud storage has become a vital solution for individuals and enterprises to store their data. With the rapid growth of data volumes, users require an efficient way to store and query data on the cloud. However, relying on such a data outsourcing method raises concerns about data security, making it necessary for users to encrypt their data before uploading.

We consider the following scenario: a client intends to outsource a collection of documents to an untrusted third-party server while retaining the ability to perform keyword searchesthat is, the client wishes to retrieve documents that contain a specified keyword. At the same time, the client wishes to keep the content of the query (i.e., the search keyword) and plain-text document private from the server.

The technique that enables this functionality is known as *Searchable Symmetric Encryption* (SSE). Different SSE schemes support various functionalities such as Boolean queries, range queries, multi-user access control, and dynamic updates. Designing SSE schemes often involves balancing trade-offs among functionality, security, and performance.

Early works such as [1] proposed basic SSE schemes and security definitions. For instance, the concept of *indistinguishability under chosen-keyword attacks* (IND2-CKA) was introduced, where the server should not be able to distinguish between search queries corresponding to two chosen keywords.

However, recent studies have demonstrated that even provably secure SSE schemes can be vulnerable in practice. By leveraging leakage together with auxiliary information, an untrusted server can recover sensitive information such as the plaintext search keywords that SSE schemes aim to protect.

While most prior attacks [2]–[6] have focused on single keyword queries, this thesis investigates the security of an SSE scheme named TWINSSE, which supports both *conjunctive* and *disjunctive* queries. We explore various attack paths under different scenarios and propose two feasible attacks targeting TWINSSE in this thesis.

1.1 Searchable Symmetric Encryption

In typical SSE constructions, documents are encrypted and stored on the server. An encrypted index, constructed using *Structured Encryption* (STE) techniques, is built and uploaded along with the encrypted documents. This secure index aims to allow the server to retrieve the identifiers of documents containing a queried keyword, without learning the actual keyword itself. After receiving the document identifiers, the client can request and decrypt the corresponding encrypted documents. Most SSE schemes consist of two stages: **Setup** and **Query**, which are illustrated in Figure 1.1.

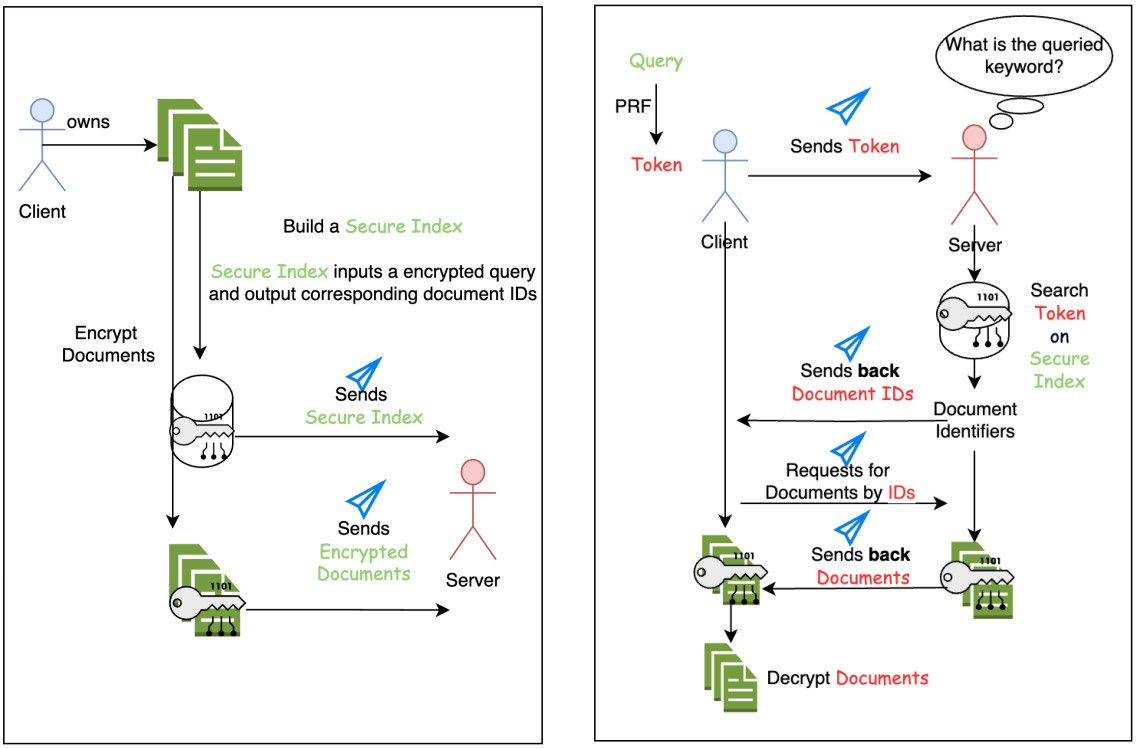
In the **Setup** stage, the client processes the target document set by *constructing a secure index* and *encrypting the documents*, then uploads both the secure index and the encrypted documents to the server. The secure index enables mapping from encoded keywords to document identifiers. Note that the index does not store plaintext keywords directly; instead, keywords are typically encoded into tokens using a pseudorandom function (PRF) or a hash function, which the server cannot interpret. The document identifiers in the index uniquely identify the corresponding encrypted documents.

In the **Query** stage, when the client wants to query which documents contain a certain keyword, it first encodes the keyword into a *token* and sends this token to the server. The server searches the secure index for document identifiers corresponding to the token and returns these identifiers to the client. The client then requests the encrypted documents by these identifiers from the server and decrypts them locally. Since the server is not always trustworthy, *it may be curious about the hidden queried keywords.*

Several SSE schemes have been proposed to support different functionalities. For example, Bost [7] introduced a dynamic SSE construction that supports document insertion, deletion, and modification. The paper also formalized the notions of forward and backward security. In addition, other dynamic constructions have been proposed based on various cryptographic designs, such as those by Miers et al.[8] and Sun et al.[9].

To support multi-user environments, Wang et al. [10] proposed a concrete multi-user SSE scheme that allows multiple users to store encrypted documents on the same database server.

Regarding query expressiveness, several schemes have been developed to support conjunctive keyword queries, such as OXT [11] and HXT [12]. Building on this line of work, TWINSSE [13] further extended the capability by supporting both conjunctive and disjunctive queries within a unified framework.



(a) Setup interaction of SSE

(b) (Query interaction of SSE

Figure 1.1: The interactions of setup and query stage between client and server.

1.2 Leakage Profile

In Searchable Symmetric Encryption (SSE) schemes, interactions between the client and the server inevitably produce leakage. A malicious server may exploit this leakage to infer sensitive information and recover parts of the client's private data. Even provably secure SSE schemes can remain vulnerable due to their inherent leakage. Since most known attacks rely on exploiting leakage, analyzing the leakage profile is essential for evaluating the security of SSE systems.

While some advanced SSE schemes that support specific functionalitiessuch as the conjunctive SSE scheme OXT [11]have relatively unique leakage profiles, the leakage profiles of most SSE schemes follow common patterns. Consequently, many attack algorithms are designed to exploit these common leakage patterns.

1.2.1 Leakage Classification

From a system-wide perspective, leakage can be categorized into *setup leakage* and *query leakage*, corresponding to information revealed during the initialization and querying phases, respectively. Among these, query leakage is the primary target of most attacks and can be further divided into **result leakage** and **query leakage**.

Result leakage refers to information about the documents returned by a query. This thesis focuses on two primary aspects: the **Result Pattern (RP)**, which

reveals the identities of the returned documents, and the **Volume Pattern (VP)**, which discloses the number of matching documents.

Query leakage refers to structural or relational information about the queries themselves. Even without knowing the exact query terms, the server may detect recurring patterns among queries. A common form of query leakage is the **Equality Pattern (EP)** (also known as the *search pattern*), which allows the server to determine whether two queries are identical. This thesis particularly focuses on query leakage in *conjunctive* and *disjunctive* query scenarios, introducing the notions of **Query-level Equality Pattern (QEP)** and **Keyword-level Equality Pattern (KEP)**. While not all Boolean SSE schemes adhere to these specific leakage patterns, they are commonly observed in practice. Formal definitions of the leakage patterns studied in this work will be presented in Chapter 2.

It is important to note that the information leaked to the server is not necessarily equivalent to the actual query result over the plaintext database. In this thesis, *leakage* refers to any information that becomes directly observable by the server during the execution of the SSE protocol, under the assumption that no *leakage suppression* techniques are employed.

1.2.2 Leakage Suppression

To limit the information that the server can observe, a variety of *leakage suppression techniques* have been proposed, including Oblivious RAM (ORAM)-based methods such as SEAL [14], and volume-hiding multi-map constructions [15], [16]. More recently, TWINSSE [13] was introduced as a Boolean SSE scheme that, to some extent, suppresses the keyword-level equality pattern (KEP) by issuing *meta-queries* and returning *spurious results* (i.e., noise) to obscure the actual query intent.

These methods typically aim to obfuscate access and volume patterns by introducing noisy or fake results to mask the expected outputs. As a result, they prevent the server from learning which documents are actually returned and how many matches the query yields.

Different leakage suppression techniques are applied to different components of an SSE system. Specifically, SEAL serves as a general-purpose mechanism that can be integrated into various SSE schemes, allowing users to adjust parameters to control the level of leakage. TWINSSE, on the other hand, supports Boolean queries and has a built-in query mechanism that inherently provides leakage suppression, especially for disjunctive queries. Recent volume-hiding multi-map approaches [15], [16] apply suppression techniques directly to the secure index by truncating or padding entries, thereby obfuscating the result pattern from the server.

However, leakage suppression might introduce overhead. Leakage suppression techniques often introduce substantial computation and storage costs, which reduces the practicality of SSE schemes. For example, TWINSSE incurs significant overhead due to the need to store approximately $|\Delta|^2$ meta-keywords, where $|\Delta|$ is the keyword space size. Gui et al. [17] empirically evaluated the runtime impact of several leak-

age suppression techniques, and concluded that some of the techniques introduce significant cost. As a result, recent research focuses on finding an effective trade-off between leakage reduction and system efficiency, as well as investigating whether specific leakage profiles remain secure under realistic threat models.

Additionally, spurious results increase the client's burden, a common example is when a user, in the presence of noise results, needs to perform additional computation to identify and filter out the noise in order to retrieve the originally expected results.

1.2.3 Leakage Analysis

Several frameworks for *quantitative leakage analysis* have been proposed in recent years. Kamara et al. [18] introduced a set of Bayesian tools for mounting leakage-abuse attacks and evaluating security under a given leakage profile. Their framework models query and auxiliary data distributions but is mainly applicable to low-dimensional or simple query scenarios, and does not scale well to high-dimensional settings [17].

Boldyreva et al. [19] proposed a more fine-grained approach using *gain functions* to quantify leakage. This method offers improved flexibility and scalability, making it suitable for a wider range of threat models and use cases.

1.3 Auxiliary Information

Auxiliary information plays a crucial role in attacks towards SSE schemes. In this thesis, we distinguish concepts between **auxiliary data** and **auxiliary information**. *Auxiliary data* refers to the original dataset known to the adversary, while *auxiliary information* is the processed form of that data, which can be directly used in concrete attack algorithms.

Unlike the **target data** (or target dataset), which represents the set of documents involved in *actual* query scenarios, the auxiliary data denotes additional information that the server is assumed to possess for the purpose of mounting an attack. We argue that a persuasive approach to assume auxiliary information is to start from a plausible source of auxiliary data and clearly define the process by which it is transformed into auxiliary information. In contrast, directly assuming access to such information without specifying its origin is a considerably stronger and less realistic assumption.

To better understand auxiliary information, we categorize it from **content-based** and **source-based**.

To categorize auxiliary information by content, we distinguish between **query information** and **document information**. *Document information* is the auxiliary data that directly extracted from known plaintext documents, and further processed into instances of specific data structure for attacks. *Query information*, on the other

hand, is typically derived from a known set of query keywords, sometimes annotated with time-related metadata such as timestamp. This type of information may come from real query histories of the same resource of dataset or be synthesized from publicly available datasets such as Wikipedia or Google Trends. The actual availability of real query histories remains uncertain. Notably, previous works such as [2], [20] made use of query-related auxiliary information but did not explicitly mention the query data resource.

To classify auxiliary information by its source, we divide it into *internal* and *external* auxiliary information, and this classification method is more commonly applied in document information. Internal auxiliary information is derived from a portion of the plaintext documents in the target dataset. For example, Blackstone et al. [3] utilized internal information to perform known-data attacks. In contrast, external auxiliary information is extracted from a separate dataset that is typically assumed to share similar keyword distribution characteristics with the target dataset.

1.4 Attacks

Since encrypted data is outsourced to an untrusted third-party server, the server may act maliciously by attempting to recover unknown queries or data. In this thesis, we focus on security aspects and use the terms *server*, *malicious server*, *attacker*, or *adversary* interchangeably to refer to entities that aim to compromise confidentiality.

The primary goal of the adversary is to recover queries or underlying data. As leakage information mainly produced in query interaction, and query keywords can reveal sensitive user intent or even identity, most research efforts including this thesis focus on **query-recovery attacks**.

Query-recovery attacks are generally categorized into **passive** and **active attacks** based on the server's capabilities. A *passive attack* assumes a semi-honest server that only observes leakage and attempts to infer query terms. In contrast, an *active attack* involves a malicious server that deviates from the protocol, for instance, by create and inject a document with selected keywords.

1.4.1 Passive Attack

A common **passive attack** scenario involves the client querying certain keywords. These keywords are first processed into tokens by the client and then sent to the server. The server assigns a unique tag to each distinct query token it receives, as it can identify repeated queries based on the search equality pattern. It also observes the associated query leakage for each tag. The core objective of a passive attack is to determine an optimal mapping from tags to the known original query keywords.

Following the classification proposed by Oya et al. [21], passive attacks can be further divided into *ground-truth attacks* and *statistics-based attacks*. In the following

sections, we adopt this categorization to provide a detailed discussion of passive attacks.

1.4.1.1 Ground-truth Attack

Ground-truth attack use internal documents in target document set as auxiliary data.

The first such attack was proposed by Islam et al. [6], known as the *IKK attack*, which demonstrated that query recovery is feasible under a strong assumption, that the adversary knows the entire plaintext document set. The server builds a *co-occurrence volume matrix* for known auxiliary data and query leakage information, and uses the **simulated annealing (SA)** algorithm to seek an optimal mapping that aligns two matrices.

Blackstone et al. [3] proposed *Subgraph attack*, which leverages a fraction of original dataset. This attack constructs two bipartite graphs representing keywords-searching results as auxiliary and leakage information, and iteratively recovers the mapping from tags to keywords, by search result or volume leakage.

1.4.1.2 Statistics-based Attack

Figure 1.2 illustrates a general workflow of a statistics-based attack on SSE schemes. In such attacks, the adversary is assumed to have access to an auxiliary dataset, which shares a similar structural and statistical properties with the target document collection. The attacker extracts statistical features from the auxiliary data and constructs auxiliary information. During the query process, the adversary observes leakage during the query interaction to construct an instance of same data structure as auxiliary information. In the matching stage, an optimization algorithm is applied to find the best mapping from encoded tokens (tags) to plain-text keywords, by maximizing similarity between the data structure instances constructed from observed leakage and auxiliary information.

Pouliot et al. [4] proposed a *graph matching attack* that constructs co-occurrence matrices from both auxiliary data and observed leakage, and organizes them into weighted graphs. In these graphs, nodes represent known keywords or query tags, respectively. Edges are weighted by the normalized volume of intersection between node pairs. The adversary aims to solve a graph matching problem by finding an optimal map that maximizes structural similarity between the graphs. Similarly, Gui et al. [17] propose an attack by building co-occurrence matrices, which leverages the scalability of SA algorithm, by adapting adapt initialization and scoring subroutines to exploit existing leakage suppression techniques, including volume-hiding multi-maps and some ORAM based techniques[14]–[16].

The attack proposed by Liu et al. [22] relies solely on the search equality pattern. In this attack, the adversary constructs a set of vectors from auxiliary data, where each vector represents the query frequency of keywords within a specific time interval (e.g., an hour or a day), and builds another vector set from the observed frequencies of tags.

The best permutation between keywords and tags is then obtained by measuring the similarity between the two vector sets. Likewise, the *QCCP* attack [20] constructs query co-occurrence matrices from conjunctive queries, assuming KEP leakage to determine the likelihood that two keywords appear together in a query.

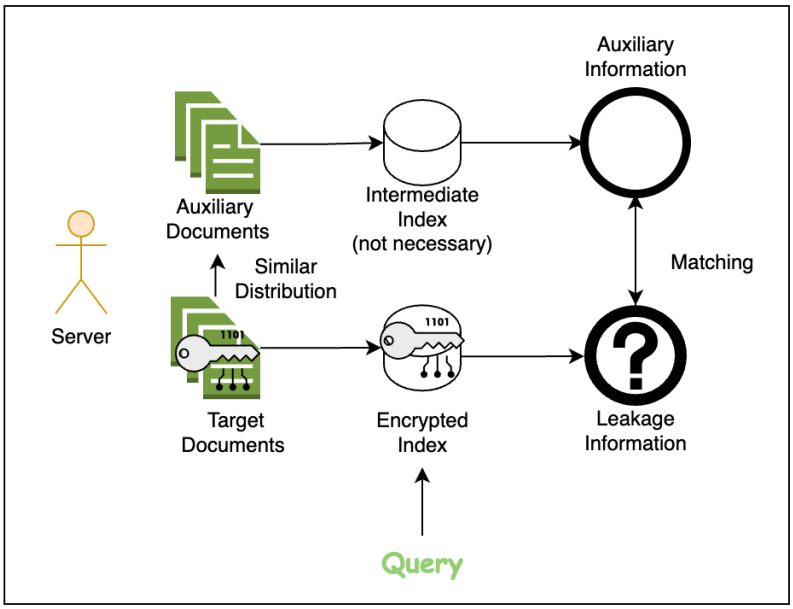


Figure 1.2: General work flow of a Statistics-based Attack

In this thesis, we propose a **Naïve Hypergraph Matching Attack** applied as a intermediate attack algorithm to recover keywords in conjunctive query under the assumption of KEP leakage.

1.4.2 Active Attack

The choice of specific attack strategies depends on the practical deployment settings. **Active attacks** are typically carried out in the form of file injection; in this thesis, we use the terms *active attack* and *file-injection attack* interchangeably. File-injection attacks assume a stronger adversarial model, where the attacker can insert crafted files into the target dataset. Most existing injection-based attacks exploit *result patterns* and *volume pattern*.

We adopt the concept of **adaptive** and **non-adaptive** file-injection attacks. In a *non-adaptive file-injection attack*, the adversary injects files only once and subsequently recovers information about future queries or existing query set without further interaction. Representative examples include the *Decoding Attack* in [3], which exploits volume patterns, and the *Binary Attack* proposed by Zhang et al. in [23], which relies on result patterns.

In contrast, *adaptive file-injection attacks* typically require the adversarial ability to observe query results and replay queries after each injection. This iterative process

allows the adversary to gradually narrow down the set of possible keywords associate with a query. For example, the Hierarchical-Search Attack [23] monitors query frequency changes of previous query after each injection step. This method balances the completeness and efficiency of the attack by sacrificing a full recovery rate in exchange for reducing the number of documents injected.

When considering file injection, the fundamental aspects the attacker generally focus on include whether the attacker has update access to the document collection and whether the injection would raise awareness, for example, triggering database audits due to a sudden large volume of injected files. In the latter case, the attacker can inject the crafted files in batches. If the injection frequency and volume match the normal update patterns of the dataset, the injected frequencies may evade detection.

Note that data retrieval ultimately occurs through the secure index. Therefore, successful file injection not only requires inserting/updating new files in the target document set but also updating the secure index accordingly. This means the attacker must evaluate the SSE schemes index update timing when designing a concrete attack.

In this thesis, we propose two non-adaptive file-injection attacks to recover disjunctive queries: **Baseline (naïve) Attack** and **Grid Attack**. The *Baseline Attack* requires less assumption but more files of injection, while the *Grid Attack* requires a stronger assumption but needs fewer files.

1.5 TWo-IN-one SSE

TWo-IN-One SSE (TWINSSE) was proposed by Bag et al.[13] to support both *conjunctive* and *disjunctive* boolean queries, and can be generalized to support queries in *conjunctive normal form (CNF)* and *disjunctive normal form (DNF)*. TWINSSE introduces the concepts of **meta-keywords**, **meta-queries**, and **meta-database**, which we will explain in more detail in Chapter 2.

During the *setup* phase, the client organizes keywords into *meta-keywords*, where each meta-keywords corresponding value is the *disjunction* of a set of original keywords search results. The server stores all possible meta-keywords and their associated values, forming the encrypted meta-database. Such a meta-database must support conjunctive queries, (i.e. such scheme is a conjunctive SSE (CSSE) scheme).

When the client wants to perform a disjunctive query over a set of keywords, it generates a corresponding set of *meta-keywords* according to a predefined rule, forming a *meta-query*. The client then performs a *conjunctive query* over the *meta-keywords* in the *meta-query*. The server returns the result of this conjunctive query, which necessarily includes all expected results of the original disjunctive query along with

additional spurious (noise) results. The client then filters out these irrelevant document identifiers locally to obtain the final expected result.

Since the query result leaked to the server contains extra spurious results, the server cannot always precisely observe the *actual set* of matching documents. Moreover, the server cannot always directly observe the overlapping of queried keywords among different disjunctive queries due to the meta-keywords mechanism.

TWINSSE relies on an underlying **Conjunctive Searchable Symmetric Encryption (CSSE)** scheme to query the conjunction of generated meta-keywords to support disjunctive queries. A specific TWINSSE implementation inherits the conjunctive query capabilities from the underlying CSSE scheme, and the conjunctive queries are directly follow the query protocol of the underlying CSSE scheme. In this paper, we focus primarily on the disjunctive query aspect of TWINSSE.

1.6 Contributions

This paper begins with an *extensional cryptanalysis of TWINSSE* and proposes file-injection attacks and a statistics-based attack for TWINSSE.

1.6.1 Extensional Cryptanalysis of TWINSSE

The original TWINSSE paper [13] lacks a comprehensive security analysis and overlooks potential vulnerabilities. In this work, we extend its security discussion and analyze how the choice of underlying CSSE schemes may enable specific attack opportunities in Chapter 3.

We first identify feasible *passive attack paths* under different leakage patterns of CSSE schemes, particularly focusing on QEP and KEP, which has been given a formal definition in Chapter 2.

To the best of our knowledge, no prior work has proposed a file-injection attack specifically targeting **disjunctive queries**. We fill this gap by presenting a *naïve active attack* named Baseline File-Injection Attack and evaluate its correctness and efficiency under practical settings.

1.6.2 Bucket Targeting and the Grid Attack

During our cryptanalysis, we observe that the **bucketization mechanism**, while crucial for the efficiency of TWINSSE, introduces new security risks. We identify that breaking bucket obfuscation is key to enabling more efficient attacks. To this end, we propose a **Bucket Targeting method** based on file injection.

Building on this, we introduce the **Grid Attack** in Chapter 4, which significantly reduces the number of injected files compared to the Baseline File-Injection Attack. This attack assumes a slightly stronger adversarial knowledge compared to

the baseline approach, trading off assumptions for improved efficiency. A comparison between similar file-injection attacks and our proposed attacks is presented in Table 1.1. The relevant notations will be formally introduced in Chapter 2.

Table 1.1: Comparison of file-injection attacks.

Attack	# Injected Files	Leakage Pattern	Attack Target	Auxiliary Information
Binary Attack [3]	$\log(\lvert\Delta\rvert)$	$\lvert RP\rvert$	Naïve	-
Binary Attack [23]	$\log(\lvert\Delta\rvert)$	RP	Naïve	Keyword space
Baseline Attack	$\lvert\Delta\rvert$	$\widetilde{RP}$	Naïve Disjunctive Queries	Keyword space
Grid Attack	$n_B + N'$	$\widetilde{RP}_{(i)}$ for each $i \in \{1, \ldots, n_B\}$	[13]	Keyword Frequencies

1.6.3 Hypergraph Matching Attack

As a core component of an attack path introduced in Chapter 3, we further investigate *conjunctive query attacks based on KEP leakage*. We design a **naïve Hypergraph Matching Attack**, which leverages auxiliary document information and query observations to construct hypergraphs. We then apply this attack on query recovery towards TWINSSE in Chapter 6.

1.7 Ethical Consideration

Our experiments are conducted on the Enron dataset[1], a public dataset used for simulating attack scenarios. This dataset contains no real private or commercial information. Furthermore, the motivation of our work is to highlight potential security threats, so that such risks can be properly considered when deploying TWINSSE in practice.

[1] `https://www.cs.cmu.edu/~enron/`

2

Preliminaries and Background

In this chapter, we begin by introducing the notation for Searchable Symmetric Encryption (SSE) adopted in this thesis, followed by an explanation of the TWINSSE construction. At the end of this Chapter, we will formally define QEP and KEP leakage under CSSE schemes.

2.1 Searchable Symmetric Encryption

We denote $\Delta = \{w_1, \ldots, w_{|\Delta|}\}$ as the keyword space ranked by keyword frequencies. It contains all the keywords that appear in the database, and $|\Delta|$ denotes the size of the keyword space. A document $\texttt{id}_i$ is the identifier that uniquely refers to the i-th document.

Let `Data` be a mapping from keywords to sets of document identifiers, derived from a plaintext document set. Specifically, if documents with identifiers $\{\texttt{id}_1, \ldots, \texttt{id}_n\}$ all contain keyword w, then $\texttt{Data}(w) = \{\texttt{id}_1, \ldots, \texttt{id}_n\}$. This mapping serves as a plaintext, intermediate data structure that reflects the raw associations between keywords and documents.

We then define `DB` as the encrypted database derived from `Data`, which stores encoded keyword-document associations in a searchable format. While `Data` is a logical representation used only during the index construction phase, `DB` is the final, encrypted structure that is deployed and queried. In other words, `Data` is fully visible and used internally, whereas `DB` is encrypted and supports secure query processing in practice.

We also define an operation $\texttt{DB}(\cdot)$ that takes a conjunctive or disjunctive *expression* containing one or more keywords w as input and returns a *set* of n document identifiers $\{\texttt{id}_1, \ldots, \texttt{id}_n\}$ that match the query. We denote $|\texttt{DB}(w)|$ as the size of the output set for query w on `DB`. We use curly braces { } to denote a set and parentheses ( ) to denote query *expressions* involving keywords.

A *single-keyword query* is a query $q = (w)$ containing only one keyword. In this case, if a client wants to retrieve the documents that contain keyword w_j, it generates the query $q_j = (w_j)$ and sends it to the server. The server computes $\texttt{DB}(q_j) = \texttt{DB}(w_j)$ and returns the resulting document identifiers. Note that while `DB` stores encrypted data, query expressions such as $q = (w)$ are logical representations. In practice, the

client generates encrypted tokens corresponding to these queries to interact with the server.

A *conjunctive query* q_{conj} retrieves documents that **include all** of the specified keywords. It is denoted as

$$q_{\text{conj}} = (w_1 \wedge w_2 \wedge \cdots \wedge w_n),$$

where each w_i for $i \in [n]$ is a keyword in the query. The database processes this query by computing

$$\texttt{DB}(q_{\text{conj}}) = \bigcap_{i=1}^{n} \texttt{DB}(w_i).$$

Similarly, a *disjunctive query* q_{disj} retrieves documents that include **at least one** of the specified keywords. It is denoted as

$$q_{\text{disj}} = (w_1 \vee w_2 \vee \cdots \vee w_n),$$

and the database computes

$$\texttt{DB}(q_{\text{disj}}) = \bigcup_{i=1}^{n} \texttt{DB}(w_i).$$

We also define Conj$(\cdot)$ and Disj$(\cdot)$ as functions that take a set of keywords as input and return their conjunctive or disjunctive expression, respectively.

2.2 TWINSSE

2.2.1 Meta-Keyword

TWINSSE introduces **meta-keywords** (denoted as `mkw`) to support *disjunctive* queries. A meta-keyword represents a *disjunction expression* of certain keywords in the keyword space. More precisely, it is defined as the disjunction of all keywords in the *ordered* keyword space Δ ranked by keywords' frequencies, *excluding a contiguous subset* of keywords.

When the subscript of `mkw` is a single index (e.g., $\texttt{mkw}_i$), it typically refers to an indexable element from a predefined sequence of meta-keywords. When the subscript includes two indices (e.g., $\texttt{mkw}_{i,j}$), it denotes the disjunction over all keywords in Δ *excluding* the continuous subset $\{w_i, w_{i+1}, \ldots, w_j\}$. Formally,

$$\texttt{mkw}_{i,j} = \text{Disj}\Big(\Delta \setminus \{w_i, w_{i+1}, \ldots, w_j\}\Big).$$

2.2.1.1 Constructing Meta-Queries from Disjunctive Queries

In TWINSSE, the disjunction of any set of keywords in Δ can be equivalently represented by a conjunction of a specific set of meta-keywords.

Given a disjunctive query consisting of n keywords:

$$q_{\text{disj}} = (w_{\text{idx}_1} \vee \cdots \vee w_{\text{idx}_n}),$$

we define a transformation function $\texttt{GenMKW}(\cdot)$ that converts a disjunctive expression into a conjunction of meta-keywords via the following steps:

1. **Exclude query terms:**

   Compute the set
   $$\tilde{\Delta} = \Delta \setminus \{w_{\text{idx}_1}, \ldots, w_{\text{idx}_n}\},$$
   which consists of all keywords in Δ not present in the query.

2. **Grouping consecutive keywords:**

   Partition $\tilde{\Delta}$ into *maximal* subsets S_j consisting of *consecutively* indexed keywords (i.e., if $w_i, w_{i+1} \in \tilde{\Delta}$, then they belong to the same subset S_j).

3. **Construct meta-keywords:**

   For each such subset $S_j = \{w_i, w_{i+1}, \ldots, w_k\}$, define
   $$\texttt{mkw}_{i,k} = \text{Disj}(\Delta \setminus S_j),$$
   i.e., the disjunction over all keywords *not* in S_j.

4. **Combine meta-keywords:** The final result is a conjunction of all constructed meta-keywords:
   $$q_{\text{mkw}} = (\texttt{mkw}_1 \wedge \cdots \wedge \texttt{mkw}_m).$$

Such a conjunction of meta-keywords can be called a *meta-query*, denoted as q_{mkw}. We will later show how $\texttt{GenMKW}(\cdot)$ and meta-query is applied in the context of disjunctive queries.

2.3 Setup and Query

As discussed earlier, this thesis focuses on the security of disjunctive queries in the TWINSSE scheme. In this section, we describe how a disjunctive query is processed within TWINSSE.

TWINSSE enables disjunctive queries by transforming them into conjunctive queries over specially constructed *meta-keywords* through $\texttt{GenMKW}(\cdot)$. It integrates with an underlying conjunctive Searchable Symmetric Encryption (CSSE) scheme, such as OXT or HXT [11], [12], to execute the conjunction queries of meta-keywords.

We use TWINSSE to refer to the meta-keyword design and construction mechanism. When discussing query execution using a specific CSSE scheme, we denote the system as $\text{TWINSSE}_{\text{CSSE}}$. For example, if OXT is used as the conjunctive backend, we write $\text{TWINSSE}_{\text{OXT}}$.

2.3.1 Meta-Database Setup

We define the **meta-database**, denoted $\mathtt{MDB}$, shares the same functionality as $\mathtt{DB}$ but retrieves document identifiers from meta-keyword. The setup procedure of $\mathtt{MDB}$ involves:

1. **Data Item Generation:**

   Invoke $\mathtt{GenData}(\cdot)$, which takes the original keyword-document mapping $\mathtt{Data}$ and keyword space Δ, enumerates *all* possible meta-keywords, and computes their associated document sets to form **meta-data** $\widehat{\mathtt{Data}}$:

   $$\widehat{\mathtt{Data}} \leftarrow \mathtt{GenData}(\mathtt{Data}, \Delta),$$

   where the value associated with each meta-keyword $\mathtt{mkw}_i$ can be computed as:

   $$\widehat{\mathtt{Data}}(\mathtt{mkw}_i) \leftarrow \bigcup_{w_i \in \mathtt{mkw}_i} \mathtt{Data}(w_i).$$

2. **Meta-Database Generation:**

   Let $\mathtt{CSSE.Setup}(\cdot)$ be the database *setup* algorithm for a specific CSSE scheme:

   $$(sk, st, \mathtt{DB}) \leftarrow \mathtt{CSSE.Setup}(\mathtt{Data}, 1^\lambda),$$

   where sk represents secret key and st denotes initial state. For an underlying conjunctive SSE scheme of $\text{TWINSSE}_{\text{CSSE}}$, the same algorithm is applied to $\widehat{\mathtt{Data}}$ instead:

   $$(sk, st, \mathtt{MDB}) \leftarrow \mathtt{CSSE.Setup}(\widehat{\mathtt{Data}}, 1^\lambda).$$

3. **Sending the Meta-Database:**

   The client sends the meta-database instance $\mathtt{MDB}$ to the server.

2.3.1.1 Meta-Queries

In a CSSE scheme, the *query* function takes a conjunctive query of keywords and returns matching document identifiers:

$$\{\mathtt{id}_1, \ldots, \mathtt{id}_n\} \leftarrow \mathtt{CSSE.Query}(\mathtt{DB}, q_{\text{conj}}).$$

In $\text{TWINSSE}_{\text{CSSE}}$, we model a meta-database query function $\mathtt{MDB}^{\text{CSSE}}(\cdot)$, which takes a meta-query q_{mkw} as input and outputs a set of document identifiers. For simplicity, when the specific conjunctive SSE scheme is not the focus, we omit the superscript and refer to it as $\mathtt{MDB}(\cdot)$. Thus, the query process is formalized as:

$$\mathtt{CSSE.Query}(\mathtt{MDB}, q_{\text{mkw}}) = \mathtt{MDB}(q_{\text{mkw}}) = \{\mathtt{id}_1, \ldots, \mathtt{id}_n\}.$$

A meta-query

$$q_{\text{mkw}} = (\mathtt{mkw}_1 \wedge \cdots \wedge \mathtt{mkw}_n)$$

is a conjunctive expression over meta-keywords. It is generated from a disjunctive query q_{disj} using the transformation function $\texttt{GenMKW}(\cdot)$, such that

$$q_{\text{mkw}} \leftarrow \texttt{GenMKW}(q_{\text{disj}}, \Delta),$$

which we may abbreviate as $\texttt{GenMKW}(q_{\text{disj}})$ for convenience. Example 1 shows how a conjunctive query of keywords can be transformed into a meta-query by $\texttt{GenMKW}(\cdot)$.

Example 1. As shown in Figure 2.1, consider a keyword space $\Delta = (w_1, w_2, w_3, w_4, w_5)$ and a disjunctive query $q_{\text{disj}} = (w_2 \vee w_3)$. Our goal is to simulate the process of $\texttt{GenMKW}(\cdot)$.

We compute

$$\tilde{\Delta} = \Delta \setminus (w_2, w_3),$$

and form two maximal consecutive subsets as shown in sub-figure 2.1.a:

$$(S_1, S_2) = (\{w_1\}, \{w_4, w_5\}).$$

Then we construct the meta-keywords as:

$$(\texttt{mkw}_1, \texttt{mkw}_2) = (\text{Disj}(\Delta \setminus S_1), \text{Disj}(\Delta \setminus S_2)),$$

which expand to:

$$(w_2 \vee w_3 \vee w_4 \vee w_5), \quad (w_1 \vee w_2 \vee w_3).$$

The set minus operation is illustrated in sub-figure 2.1.b. Finally, we verify that from the conjunction of given meta-keywords, we can retrieve the original queried keywords:

$$q_{\text{mkw}} = (\texttt{mkw}_1 \wedge \texttt{mkw}_2) \Rightarrow (w_2 \vee w_3) = q_{\text{disj}}.$$

This verification is visually demonstrated in sub-figure 2.1.c, where we can observe that each meta-keyword contains both w_2 and w_3, confirming the correctness of the construction.

Formally, when a client wishes to search a disjunctive keyword query $q_{\text{disj}} = (w_1 \vee \cdots \vee w_n)$ and retrieve $\texttt{DB}(q_{\text{disj}})$, it first computes the corresponding meta-query $q_{\texttt{mkw}} \leftarrow \texttt{GenMKW}(q_{\text{disj}}, \Delta)$. Then, the client interacts with the server using the underlying CSSE scheme to execute $q_{\texttt{mkw}}$ and receives:

$$\texttt{MDB}(q_{\texttt{mkw}}) = \bigcap_{i=1}^{m} \texttt{MDB}(\texttt{mkw}_i).$$

According to the original paper [13], the result set returned by the server is a superset (or equal) of the intended disjunctive query result:

$$\texttt{DB}(q_{\text{disj}}) \subseteq \texttt{MDB}(q_{\texttt{mkw}}).$$

Therefore, the client must locally filter $\texttt{MDB}(q_{\texttt{mkw}})$ to derive the exact result $\texttt{DB}(q_{\text{disj}})$

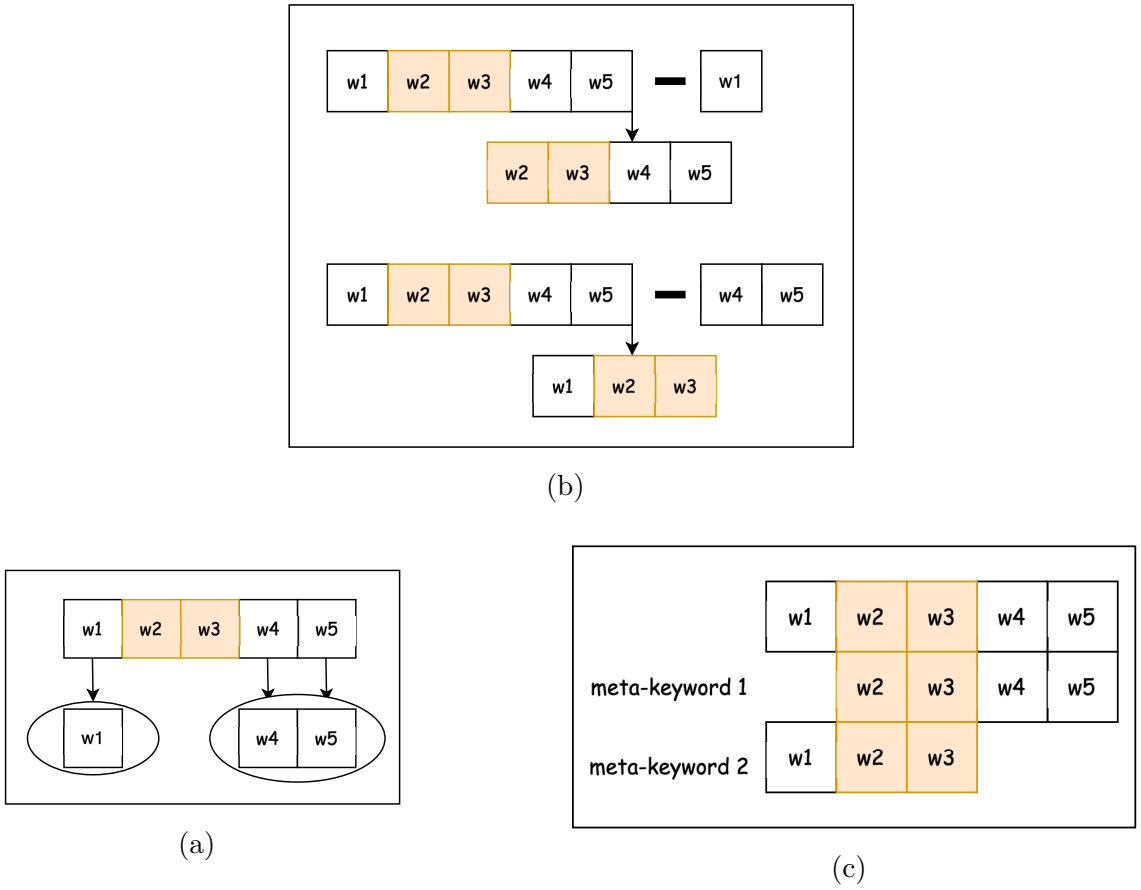


Figure 2.1: Illustration of the process in Example 1: (a) maximal consecutive subsets; (b) set minus operation; (c) verification of meta-keywords.

Since $\texttt{MDB}(q_{\texttt{mkw}})$ may contain *spurious results*i.e., document identifiers that do not match the original disjunctive querythe client must remove the difference:

$$\texttt{MDB}(q_{\texttt{mkw}}) \setminus \texttt{DB}(q_{\text{disj}}).$$

The response correctly adheres to the conjunctive semantics of the constructed meta-query and is guaranteed to include the true result set $\texttt{DB}(q_{\text{disj}})$.

2.3.1.2 Spurious Result

We suspect that the efficiency of TWINSSE may be undermined by the presence of *spurious results*, document identifiers returned to the client that significantly exceed the expected results. To understand how to recognize spurious results, we analyze the origin of spurious results from the perspective of meta-query construction.

Let $F(\cdot)$ be a function, q_{disj_j} is a disjunctive query that derives a meta-query q_{mkw_j} and $F(w, q_{\text{mkw}_j})$ denote the set of meta-keywords in the meta-query $q_{\texttt{mkw}}$ that contain the keyword w, formally defined as:

$$F(w, q_{\text{mkw}_j}) = \{\texttt{mkw}_i \in q_{\texttt{mkw}} \mid w \in \texttt{mkw}_i\}.$$

If all meta-keywords in q_{mkw_j} contain w, then it must be that $w \in q_{\text{disj}_j}$, which implies that $\texttt{DB}(w) \subseteq \texttt{DB}(q_{\text{disj}_j})$.

To identify spurious results, we consider keyword sets C satisfying the following conditions:

1. $|C| > 1$,
2. $C \cap q_{\text{disj}_j} = \emptyset$,
3. $\text{Conj}\left(\bigcup_{w \in C} F(w, q_{\text{mkw}_j})\right) = q_{\text{mkw}_j}$.

Although the keywords in C are unrelated to the original query, they collectively cover all meta-keywords in q_{mkw_j}. As a result, the server may return documents that including keywords belongs to C, which can be represented as $\texttt{DB}(\text{Conj}(C))$, even though the client did not query any of the keywords in C. This leads to the spurious result set:

$$\texttt{DB}(\text{Conj}(C)) \setminus \texttt{DB}(q_{\text{disj}_i}).$$

Example 2 illustrates how such spurious results may arise in practice.

Example 2. Consider a database with keyword space $\Delta = \{w_1, w_2, w_3, w_4, w_5\}$ with order from w_1 to w_5, where each keyword appears in the following document sets respectively:

- w_1: $\{\texttt{id}_1, \texttt{id}_2\}$
- w_2: $\{\texttt{id}_3, \texttt{id}_4\}$

- w_3: $\{\mathtt{id}_5, \mathtt{id}_6, \mathbf{id_{11}}\}$
- w_4: $\{\mathtt{id}_7, \mathtt{id}_8\}$
- w_5: $\{\mathtt{id}_9, \mathtt{id}_{10}, \mathbf{id_{11}}\}$

Suppose a client issues a disjunctive query $q_{\text{disj}} = (w_2 \vee w_4)$, and the server constructs meta-query as:

$$q_{\text{disj}} = (w_2 \vee w_4) \Rightarrow (\mathtt{mkw}_1 \wedge \mathtt{mkw}_2 \wedge \mathtt{mkw}_3).$$

The server evaluates the conjunctive query over the meta-keywords as:

$$\mathtt{MDB}(q_{\mathtt{mkw}}) = \bigcap_{i=1}^{|q_{\mathtt{mkw}}|} \mathtt{MDB}(\mathtt{mkw}_i) = \mathtt{MDB}(\mathtt{mkw}_1) \cap \mathtt{MDB}(\mathtt{mkw}_2) \cap \mathtt{MDB}(\mathtt{mkw}_3).$$

Computing each component based on the disjunctions:

$$\begin{aligned}
\mathtt{MDB}(\mathtt{mkw}_1) &= \mathtt{DB}(w_2) \cup \mathtt{DB}(w_3) \cup \mathtt{DB}(w_4) \cup \mathtt{DB}(w_5) \\
&= \{\mathtt{id}_3, \mathtt{id}_4, \mathtt{id}_5, \mathtt{id}_6, \mathtt{id}_7, \mathtt{id}_8, \mathtt{id}_9, \mathtt{id}_{10}, \mathbf{id_{11}}\}, \\
\mathtt{MDB}(\mathtt{mkw}_2) &= \mathtt{DB}(w_1) \cup \mathtt{DB}(w_2) \cup \mathtt{DB}(w_4) \cup \mathtt{DB}(w_5) \\
&= \{\mathtt{id}_1, \mathtt{id}_2, \mathtt{id}_3, \mathtt{id}_4, \mathtt{id}_7, \mathtt{id}_8, \mathtt{id}_9, \mathtt{id}_{10}, \mathbf{id_{11}}\}, \\
\mathtt{MDB}(\mathtt{mkw}_3) &= \mathtt{DB}(w_1) \cup \mathtt{DB}(w_2) \cup \mathtt{DB}(w_3) \cup \mathtt{DB}(w_4) \\
&= \{\mathtt{id}_1, \mathtt{id}_2, \mathtt{id}_3, \mathtt{id}_4, \mathtt{id}_5, \mathtt{id}_6, \mathtt{id}_7, \mathtt{id}_8, \mathbf{id_{11}}\}.
\end{aligned}$$

The intersection yields the result computed by the server:

$$\mathtt{MDB}(q_{\mathtt{mkw}}) = \{\mathtt{id}_3, \mathtt{id}_4, \mathtt{id}_7, \mathtt{id}_8, \mathbf{id_{11}}\}.$$

However, the actual result expected by the client is:

$$\mathtt{DB}(q_{\text{disj}}) = \mathtt{DB}(w_2) \cup \mathtt{DB}(w_4) = \{\mathtt{id}_3, \mathtt{id}_4, \mathtt{id}_7, \mathtt{id}_8\}.$$

Hence, the spurious result is:

$$\mathtt{MDB}(q_{\mathtt{mkw}}) \setminus \mathtt{DB}(q_{\text{disj}}) = \{\mathbf{id_{11}}\}.$$

This example, from the perspective of executing the query protocol, clearly demonstrates how a document unrelated to any client-issued keyword can still appear in the server's response. From the method we proposed that identifies candidate sets C and computes $\mathtt{DB}(\text{Conj}(C))$, we reach the same outcome. Figure 2.2 offers a visual interpretation of this process. We can observe that the sets $\{w_1, w_3\}$ and $\{w_3, w_5\}$ can both be valid candidates for C, which we denote as C_1 and C_2, respectively. Since the keywords in these sets are unrelated to the original query keywords $\{w_2, w_4\}$,

each candidate set has size greater than one, and every meta-keyword in q_{mkw} includes at least one keyword from these sets. We then find that $\texttt{DB}(\text{Conj}(C_1)) = \emptyset$, so C_1 does not contribute any valid meta-keyword. However, $\texttt{DB}(\text{Conj}(C_2)) = \{\texttt{id}_{11}\}$, and we observe that

$$\texttt{DB}(\text{Conj}(C_2)) \setminus \texttt{DB}(q_{\text{disj}}) = \{\texttt{id}_{11}\}.$$

Thus, we again identify $\texttt{id}_{11}$ as a spurious result.

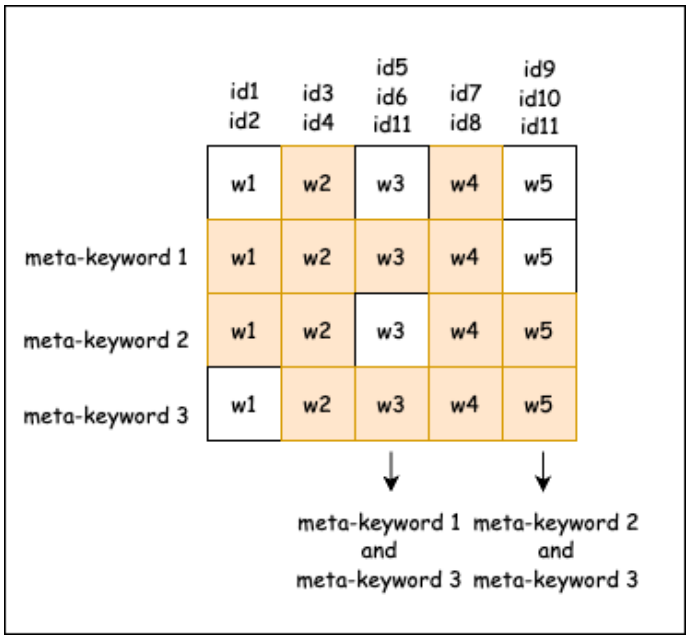


Figure 2.2: Visual representation of candidate set C_2 and its relation to spurious results.

2.3.2 Bucketization

Once all possible sets C are identified, any document that contains all keywords from some C may become a spurious result. Furthermore, the meta-database must store all possible meta-keywords, whose number is roughly $|\Delta|^2$, leading to significant overhead in both storage and query efficiency.

To alleviate spurious results and reduce the number of meta-keywords, **TWINSSE** introduces a *bucketization* technique that partitions the ordered keyword space into multiple *buckets*. The setup and query processes are then performed *bucket-wise*. Assume the system uses n_B buckets, with each bucket containing $N' := |\Delta|/n_B$ keywords. Let Δ_k denotes the k-th bucket.

Under this scheme, meta-keywords are generated *independently* within each bucket during setup, and query processing is restricted to relevant buckets. This ensures that keywords from different buckets do not form the same meta-keyword, effectively limiting database scale, reducing spurious matches, and decreasing the total number of meta-keywords.

2.3.2.1 Meta-Database Generation Under Bucketization

We have previously introduced the function $\texttt{GenData}(\cdot)$, which takes as input a mapping from keywords to their corresponding document identifiers and an ordered keyword space, and outputs a mapping from meta-keywords to their associated document identifiers. Under the bucketization setting, we perform the following steps for each bucket Δ_k:

1. **Extract local data:** From the global keyword-document map $\texttt{Data}$, extract the subset corresponding to keywords in Δ_k, and denote it as $\texttt{Data}_k$.
2. **Generate local meta-data:** Invoke the meta-data generation function within this bucket by computing

$$\widehat{\texttt{Data}_k} \leftarrow \texttt{GenData}(\texttt{Data}_k, \Delta_k).$$

After we have processed the meta-data for each bucket, we:

1. **Merge all local meta-data** into a global meta-data map:

$$\widehat{\texttt{Data}} \leftarrow \widehat{\texttt{Data}_k} \cup \widehat{\texttt{Data}}.$$

2. **Initialize the encrypted database** using the setup function of the underlying **CSSE** scheme:

$$(sk, st, \texttt{MDB}) \leftarrow \texttt{CSSE.Setup}(\widehat{\texttt{Data}}, 1^\lambda).$$

2.3.2.2 Meta-Query Procession Under Bucketization

When a client issues a disjunctive query $q_{\text{disj}} \leftarrow \text{Disj}(S)$, where S is a set of query keywords, the interaction between the client and the server under the bucketization mechanism proceeds as follows:

1. **Client-side sub-meta-query generation:**
   (a) Partition the keyword set S according to their corresponding buckets. Specifically, let S_i denote the subset of S consisting of all keywords that belong to the i-th bucket Δ_i. Note that S_i may be empty, in which case it is ignored.
   (b) For each non-empty S_i, generate a **sub-meta-query** restricted to bucket Δ_i:

$$q_{\texttt{mkw}}^{(i)} \leftarrow \texttt{GenMKW}(\text{Disj}(S_i), \Delta_i).$$

   (c) Collect all sub-meta-queries and send each of them individually to the server in arbitrary order.
2. **Server-side query processing and result aggregation:**

(a) For each received sub-meta-query $q_{\mathtt{mkw}}^{(i)}$, the server evaluates it over the meta-database and merges the results:

$$R \leftarrow R \cup \mathtt{MDB}(q_{\mathtt{mkw}}^{(i)}).$$

(b) After all sub-meta-queries are processed, the server sends the aggregated result R back to the client.

3. **Client-side filtering:**

(a) Upon receiving R, the client filters out spurious results and extracts the final set of document identifiers that satisfy the original query q_{disj}.

Figure 2.3 illustrates the procedure of processing a query into sub-meta-queries. Suppose the client wishes to query the disjunctive keyword query $q_{\text{disj}} = (w_2 \vee w_7 \vee w_9)$ over a meta-database containing the keyword universe $\Delta = \{w_1, w_2, \ldots, w_9\}$, with a predefined bucket size of $N' = 3$. Consequently, the keyword space is partitioned into three buckets $\{\Delta_1, \Delta_2, \Delta_3\}$, each consisting of three keywords.

In the first step, the queried keywords are grouped according to their respective buckets. Specifically, we obtain two disjoint subsets: $S_1 = \{w_2\}$ corresponding to Δ_1, and $S_2 = \{w_7, w_9\}$ corresponding to Δ_3. The client then generates the sub-meta-queries as follows:

$$q_{\text{mkw}}^{(1)} \leftarrow \mathtt{GenMKW}(\text{Disj}(S_1), \Delta_1), \quad q_{\text{mkw}}^{(3)} \leftarrow \mathtt{GenMKW}(\text{Disj}(S_2), \Delta_3),$$

and sends $q_{\text{mkw}}^{(1)}$ and $q_{\text{mkw}}^{(3)}$ to the server individually.

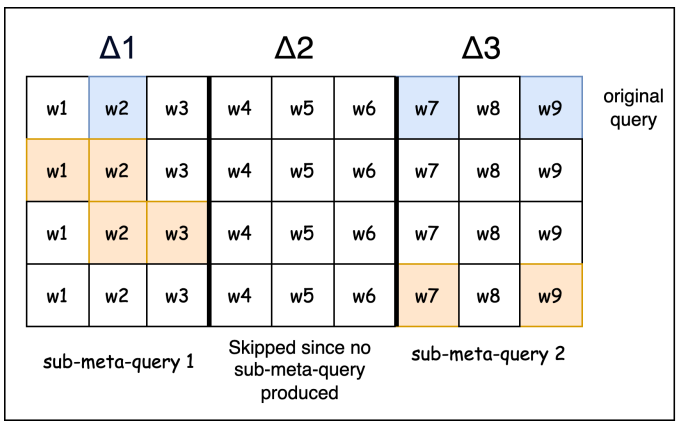


Figure 2.3: Visual representation of sub-meta-query procession.

2.3.2.3 Spurious Result under Bucketization

In Section 2.3.1.2, we introduced a method for identifying potential spurious results in the absence of bucketization, using a candidate set C that satisfies a set of specific conditions.

After introducing bucketization, this method needs to be adapted. Specifically, for each bucket Δ_i accessed by a sub-meta-query $q^{(i)}_{\text{mkw}}$, we define a candidate set C_i that must satisfy the following conditions:

1. $C_i \subset \Delta_i$,
2. $|C_i| > 1$,
3. $C_i \cap q_{\text{disj}_j} = \emptyset$,
4. $\text{Conj}\left(\bigcup_{w \in C_i} F(w, q^{(i)}_{\text{mkw}_j})\right) = q^{(i)}_{\text{mkw}_j}$.

Accordingly, the spurious results generated under the $q^{(i)}_{\text{mkw}_j}$ are given by:

$$\texttt{DB}(\text{Conj}(C_i)) \setminus \texttt{DB}(q_{\text{disj}_j}).$$

2.3.3 Leakage Pattern of TWINSSE

The leakage function of $\text{TWINSSE}_{\text{CSSE}}$ is denoted as $\mathcal{L}_{\text{TWINSSE}} = (\mathcal{L}^{\text{SETUP}}_{\text{TWINSSE}}, \mathcal{L}^{\text{SEARCH}}_{\text{TWINSSE}})$, and is derived from that of the CSSE scheme. In the case **without bucketization**, the leakage functions of $\text{TWINSSE}_{\text{CSSE}}$ are defined as follows:

$$\begin{cases} \mathcal{L}^{\text{SETUP}}_{\text{TWINSSE}}(\texttt{Data}) = \left(\mathcal{L}^{\text{SETUP}}_{\text{CSSE}}(\widehat{\texttt{Data}})\right), \\ \mathcal{L}^{\text{SEARCH}}_{\text{TWINSSE}}(q_{\text{disj}}) = \mathcal{L}^{\text{SEARCH}}_{\text{CSSE}}(q_{\texttt{mkw}}). \end{cases}$$

where $\widehat{\texttt{Data}}$ is the meta-data generated by applying the transformation $\texttt{GenData}(\cdot)$ to the original keyword-document map $\texttt{Data}$, and $q_{\texttt{mkw}}$ is the meta-query corresponding to the original disjunctive query q_{disj}.

Here, the setup leakage corresponds to the meta-database construction, while the search leakage corresponds to the processing of disjunctive queries.

From the servers perspective, query processing in $\text{TWINSSE}_{\text{CSSE}}$ follows the same protocol as in the underlying CSSE scheme: it executes a meta-query that is essentially a conjunction over meta-keywords. Therefore, TWINSSE inherits the leakage behavior of the CSSE scheme during search, and the only difference is that the queried keywords are meta-keywords instead of real keywords (i.e. query-terms).

With bucketization, the leakage pattern of $\text{TWINSSE}_{\text{CSSE}}$ is defined as follows:

$$\begin{cases} \mathcal{L}^{\text{SETUP}}_{\text{TWINSSE}}(\texttt{Data}) = \left(\mathcal{L}^{\text{SETUP}}_{\text{CSSE}}(\widehat{\texttt{Data}}), n_B, N'\right), \\ \mathcal{L}^{\text{SEARCH}}_{\text{TWINSSE}}(q_{\text{disj}}) = \left\{\mathcal{L}^{\text{SEARCH}}_{\text{CSSE}}(q^{(k)}_{\texttt{mkw}})\right\}_{k \in [n_B]}. \end{cases}$$

where n_B denotes the number of buckets and N' the size of each bucket.

In this case, the setup leakage includes both the meta-database construction and the bucketization parameters, whereas the search leakage corresponds to the processing of the disjunctive query decomposed into multiple sub-queries.

During the `setup phase`, the scheme leaks the bucketization parameters, including the number of buckets n_B and the size of each bucket N', in addition to the setup leakage associated with the transformed meta-data $\widehat{\texttt{Data}}$.

During the `query phase`, since the disjunctive query q_{disj} is decomposed into multiple sub-meta-queries $q^{(k)}_{\texttt{mkw}}$, each sub-query is sent independently to the server. For each such sub-query, the server observes the corresponding leakage $\mathcal{L}^{\text{SEARCH}}_{\text{CSSE}}(q^{(k)}_{\texttt{mkw}})$, as determined by the base CSSE scheme. Moreover, although the server knows each sub-query accesses exactly one bucket, it cannot determine which specific bucket (i.e., the value of k) is being queried.

2.4 Leakage Patterns

2.4.1 Result Pattern

Here we formally define the potential leakage patterns of a $\text{TWINSSE}_{\text{CSSE}}$ scheme. In practice, TWINSSE may apply various leakage suppression techniques to hide sensitive information from the server. However, such techniques generally incur non-negligible efficiency overhead. Without suppression, most existing CSSE schemes leak both the query result and the size of the result.

We define the following common leakage patterns:

- $RP \leftarrow \texttt{DB}(q)$: assume the database is correctly organized from the original target dataset (i.e. does not introduce TWINSSE), the document identifiers set database when executing a conjunctive or disjunctive query q.
- $\widehat{RP}_{(t)} \leftarrow \texttt{MDB}(q^{(t)}_{\text{mkw}})$: the set of document identifiers returned from the meta-database when executing a sub-meta-query $q^{(t)}_{\text{mkw}}$.
- $\widehat{RP} \leftarrow \texttt{MDB}(q_{\text{mkw}})$: the set of document identifiers returned from the meta-database in response to the entire meta-query q_{mkw}, where

$$\texttt{MDB}(q_{\text{mkw}}) = \bigcup_{t \in [nB]} \texttt{MDB}(q^{(t)}_{\text{mkw}}).$$

In specific scenarios which has been clearly stated that the client wants to process a set of queries, this labeling method can also be regarded as a vector storing multiple document identifier sets. For example, the user transforms a series of original queries $Q_{\text{disj}} = \{q_{\text{disj}_1}, q_{\text{disj}_2}, ..., q_{\text{disj}_n}\}$ to a series of meta-queries $Q_{\texttt{mkw}} = \{q_{\text{mkw}_1}, q_{\text{mkw}_2}, ..., q_{\text{mkw}_n}\}$. Upon receiving and processing these meta-queries, $\widehat{RP}$ can be considered as a sequence whose i-th element $\widehat{RP}[i]$ denotes the result leakage generated by the i-th meta-query q_{mkw_i}.

2.4.2 Search Equality Pattern

Different $\text{TWINSSE}_{\text{CSSE}}$ schemes may adopt various underlying CSSE constructions depending on efficiency and security trade-offs. We define two granularities of con-

junctive query equality leakage patterns: **Query-level Equality Pattern** (QEP) and **Keyword-level Equality Pattern** (KEP), which capture the assignment of tags to queries and keywords, respectively.

Definition (Keyword-level Equality Pattern and Query-level Equality Pattern)

Let $Q_{\text{conj}} = \{q_{\text{conj}_1}, \ldots, q_{\text{conj}_n}\}$ be a set of conjunctive queries, where each $q_{\text{conj}_i} = \{w_{i,1}, w_{i,2}, \ldots, w_{i,m_i}\}$ is a keyword-ordered tuple. We denote by $q_{\text{conj}_i}[j]$ the j-th keyword of the i-th query.

Note that keywords within each conjunctive query are naturally unordered; however, to formalize keyword-level tag assignments as a matrix, we impose a deterministic ordering (e.g., lexicographical) on the keywords of each query.

We define τ_i as a tag assigned to a query or keyword, and:

- A query-tag vector[1] $\boldsymbol{\tau} = (\tau_1, \tau_2, \ldots, \tau_n)$, where τ_i is the tag assigned to query q_{conj_i}.
- A keyword-tag matrix $\mathbf{T} = [\tau_{i,a}]$ of size $n \times m_{\max}$, where $m_{\max}$ is the size of the longest query in Q_{conj}, and $\tau_{i,a}$ is the tag assigned to the a-th keyword of the i-th query. Positions beyond the length of query i are filled with a null symbol $\varnothing$.

Query-level Equality Pattern (QEP).

The server assigns to each query q_{conj_i} a tag τ_i from a tag set $\mathcal{T}_Q$, and updates the query-tag vector $\boldsymbol{\tau}$ such that $\boldsymbol{\tau}[i] = \tau_i$.

The assignment satisfies:

$$\tau_i = \tau_j \iff q_{\text{conj}_i} = q_{\text{conj}_j}.$$

The leakage output is:

$$QEP \leftarrow (\boldsymbol{\tau}, \mathcal{T}_Q),$$

which reveals which queries are equal based on their tags.

Keyword-level Equality Pattern (KEP).

The server assigns to each keyword $q_{\text{conj}_i}[a]$ a tag $\tau_{i,a}$ from a tag set $\mathcal{T}_K$.

Upon receiving a conjunctive query $q_{\text{conj}_i} = \{w_{i,1}, w_{i,2}, \ldots, w_{i,m_i}\}$, the server updates the keyword-tag matrix $\mathbf{T}$ such that:

$$\mathbf{T}[i][a] = \tau_{i,a}, \quad \text{for } a \in [m_i].$$

The assignment satisfies:

$$\tau_{i,a} = \tau_{j,c} \iff q_{\text{conj}_i}[a] = q_{\text{conj}_j}[c].$$

[1]The terms "vector" and "matrix" here refer only to indexed collections of data and do not imply any linear algebraic operations.

The leakage output is:

$$KEP \leftarrow (\mathbf{T}, \mathcal{T}_K),$$

revealing keyword-level equality across queries.

3

Extensional Cryptanalysis of TWINSSE

It is risky to assert the overall security of an SSE scheme without clearly specifying assumptions such as experimental settings and auxiliary information. SSE security depends heavily on the adversarys capabilities under different threat models. For instance, in some realistic scenarios, a malicious server may be able to inject files into the encrypted database, enabling injection attacks.

In $\text{TWINSSE}_{\text{CSSE}}$, the underlying CSSE construction determines the concrete security guarantees. In this chapter, we aim to analyze the potential vulnerabilities of specific $\text{TWINSSE}_{\text{CSSE}}$ instantiations under various assumptions and adversarial models. While the original paper uses OXT [11] for evaluation, other CSSE schemes may be adopted in practice due to efficiency or storage considerations. Using our previously defined **Query-level Equality Pattern (QEP)** and **Keyword-level Equality Pattern (KEP)**, which characterize different granularities of search pattern leakage, we pose the following research question:

RQ1. *Are there passive attack paths that threaten $TWINSSE_{CSSE}$ under different levels of search pattern leakage (e.g., QEP and KEP)?*

To explore this, we focus on $\text{TWINSSE}_{\text{CSSE}}$ schemes that explicitly leak QEP or KEP information. We assume the existence of a passive attack algorithm capable of recovering conjunctive queries under the KEP setting. A detailed description of this algorithm will be presented in Chapter 5.

We also investigate active (file-injection) attacks in the context of disjunctive queries, which remain insufficiently explored. In this chapter, we introduce a **Baseline Attack** that leverages result pattern leakage. Although the attack operates under relatively weak assumptions and is straightforward to implement, its efficiency is limited. A more optimized strategy will be presented in Chapter 4.

Existing works on file-injection attacks often skip specifying their concrete adversarial capabilities, underlying assumptions, and precise attack settings. We also provide a description of the *attack setting* that underpins all injection-based attacks discussed in this thesis.

Taken together, this chapter outlines practical attack paths under both passive and active adversarial models. Our findings highlight that while bucketization is essential for improving TWINSSEs efficiency in practice, it can significantly weaken the systems security guarantees.

We also show that the effectiveness of these attack strategies is further amplified if the adversary has **bucket targeting** capabilities. In Chapter 4, we will introduce a file-injection-based bucket targeting method to leverage this advantage.

3.1 Passive Attack: Recovering Query from Meta-keywords

3.1.1 Threat Model

We consider a simplified scenario in which the underlying CSSE scheme of a $\mathsf{TWINSSE}_{\mathsf{CSSE}}$ instance leaks the result pattern and follows either the keyword equality pattern (KEP) or query equality pattern (QEP) for search pattern leakage.

To build foundational intuition, we do not include bucketization at this stage; its effects will be discussed in later sections.

1. The client issues a sequence of disjunctive or single-keyword queries $Q_{\text{disj}} = \{q_{\text{disj}_1}, q_{\text{disj}_2}, \ldots, q_{\text{disj}_n}\}$.

2. Each query q_{disj_i} is transformed by the client into a corresponding meta-query q_{mkw_i}, forming the set $Q_{\text{mkw}} = \{q_{\text{mkw}_1}, q_{\text{mkw}_2}, \ldots, q_{\text{mkw}_n}\}$, which is then encoded and submitted to the server individually.

3. During each query interaction:

   (a) Upon receiving encoded q_{mkw_i}, the server observes the search pattern leakage, depending on whether the underlying scheme leaks QEP or KEP:

   - Under **QEP**, the server assigns a tag to each meta-query and updates the query-tag vector $\boldsymbol{\tau}$.
   - Under **KEP**, the server assigns a tag to each meta-keyword in the meta-query and updates the keyword-tag matrix $\mathbf{T}$.

   (b) The server retrieves document identifiers $\mathtt{MDB}(q_{\text{mkw}_i})$ by following the CSSE protocol, associates the result pattern $\widehat{RP}$ with the assigned tag(s), and returns the document identifiers to the client.

   (c) The client locally filters out spurious results to recover the correct document identifiers $\mathtt{DB}(q_{\text{disj}_i})$ matching the original query.

4. After processing all queries in Q_{mkw}, the server aggregates the observed leakage and attempts to launch an inference attack.

We assume the attacker possesses an auxiliary dataset that shares the same keyword space and exhibits similar structural properties to the target dataset.

In order to perform a passive attack, more specifically, statistics-based attack, the server constructs two instances of data structures: one derived from the observed leakage (e.g., co-occurrence volumes over tags), and the other derived from auxiliary data (e.g., co-occurrence volumes over keywords).

The attacker aims to identify a tag-to-keyword mapping that maximizes the structural similarity between the two data structures.

3.1.2 Attack Path

We defer the exploitation of the scheme under the bucketization mechanism to a later section in this chapter. A feasible attack path at this stage is to *first recover the meta-keywords through a passive attack, and then infer the original query keywords from meta-keywords* in a given meta-query q_{mkw_i}.

If the CSSE leaks under the QEP search pattern, the attacker's goal reduces to mapping each observed query tag $\tau_i \in \mathcal{T}_Q$ to a known conjunctive expression over meta-keywords. Each such expression can be treated as a *virtual keyword*, reducing the task to a query recovery problem over a virtual keyword space. The value (e.g., document count) associated with each tag τ_i can be derived from $\widehat{RP}$, and matched against the auxiliary datasets simulated meta-query values in a single-point statistics-based attack.

Assuming the disjunction dimension is bounded by n, and the keyword universe is Δ, the total number of possible meta-queries, which is determined by the possible number of original keyword disjunctive queries, is upper-bounded by

$$\sum_{i=1}^{n} \binom{|\Delta|}{i} \leq 2^{|\Delta|} - 1.$$

which also represents the maximum size of the tag set $\mathcal{T}_Q$.

If the CSSE scheme follows the KEP search pattern, we assume the existence of a **statistics-based conjunctive query recovery attack**. This attack globally attempts to recover the mapping from tags in $\mathcal{T}_K$ to meta-keywords in Q_{mkw}. It operates by matching two structurally similar data representations, for example, hypergraphs storing co-occurrence information in our case:

- The first is constructed from the *observed leakage*, particularly from $\widehat{RP}$ and the KEP search pattern. These enable the adversary to infer tag co-occurrence volumes across queries and organize them into a data structure.
- The other is derived from *auxiliary data*, where the adversary simulates all possible meta-queries using known meta-keywords and builds a corresponding data structure capturing their co-occurrence patterns.

We defer the concrete design of such an attack algorithm to Chapter 5, where we explore how this inference process can be instantiated. By comparing these two structures, the adversary attempts to recover the meta-keywords from received meta-queries.

After meta-keywords are recovered, regardless of whether the scheme follows the QEP or KEP pattern, the attacker can further recover original query keywords from recovered meta-keywords. This inference is enabled by the approximate keyword frequencies provided by the auxiliary data.

Regarding complexity, the number of candidate meta-keywords is upper bounded by $|\Delta|^2$, which also bounds the number of keyword tags $|\mathcal{T}_K|$. However, recovering conjunctions is potentially more costly than recovering single keywords, due to the combinatorial overhead of constructing query candidates.

3.1.3 Impact of Similarity in Auxiliary Data

Document similarity plays different roles at different stages of our proposed attack path. In the *meta-keyword recovery* stage, we focus on semantic features such as co-occurrence patterns between a given keyword and other keywords within the same bucket. During the *recovery from meta-keywords to original keywords*, keyword frequencies and their exact order become critical. This is because meta-keyword generation depends on frequency-based rankings, and the inverse process requires knowledge of this ordering.

As a result, the success of the attack is heavily dependent on the *reliability* of the auxiliary data available to the attacker.

To avoid such a high dependency, an alternative strategy is to *directly recover the original keywords*, bypassing the intermediate meta-keyword stage. This approach, originally adopted in the original paper, is briefly introduced here.

Under the QEP leakage setting of the scheme, each enumerated meta-query corresponds to exactly one original disjunctive query. Therefore, when constructing auxiliary information, the attacker can treat each disjunctive expression as a virtual keyword as introduced in our QEP-based attack, and simulate its query results from auxiliary data directly, without explicitly constructing meta-keywords. The attack objective then shifts to directly recovering disjunctive expressions from the leakage of observed meta-queries, without relying on intermediate abstractions.

The original paper does not explore this direction in detail; although our current work does not include the construction and evaluation of this alternative approach, we identify two key challenges that should be addressed for this strategy to enable a more precise statistics-based attack:

Uncertainty in Modeling Disjunctive Co-occurrence. Most existing statistics-based attacks that rely solely on auxiliary documents, such as those in [4], [5], [17], are based on co-occurrence semantics. However, in the case of recovering disjunctive

expression, it remains unclear whether disjunctive queries exhibit meaningful co-occurrence patterns that can be exploited in a similar way.

Handling Spurious Results.

The presence of spurious results in actual queries further increases the uncertainty of the observed query results, leading to higher entropy, and makes the leakage-derived patterns less reliable. An effective statistics-based attack must account for the noise introduced by such spurious results. Related challenges have been discussed in previous works such as [2], [17], [18].

3.1.4 Threats Under Bucketization

Introducing the bucketization mechanism effectively suppresses both the number of meta-keywords and spurious results, thus alleviating computational overhead during the SSE protocol execution. However, this reduction in the meta-keyword space also constrains the upper bound of possible candidates for matching during query recovery.

Under the bucketization setting, the attacker handles each sub-meta-query $q_{\text{mkw}}^{(i)}$ separately and observes the result leakage $\widehat{RP}_{(i)}$. However, the attacker cannot directly infer which bucket is visited from a sub-meta-query, even though it knows that each $q_{\text{mkw}}^{(i)}$ is derived from a meta-query.

Given n_B buckets, each containing N' keywords, the number of possible sub-meta-queries is determined by the number of possible disjunctive expressions within each bucket, multiplied by the total number of buckets. Denoting the maximum disjunction size as n, the number of such expressions within a single bucket is bounded by:

$$\sum_{i=1}^{n} \binom{N'}{i} \leq 2^{N'} - 1.$$

This also characterizes the size of the virtual keyword space under the QEP search pattern. Consequently, the attacker confronts a single-keyword query recovery problem over a space of size at most $n_B \cdot (2^{N'} - 1)$ virtual keywords.

Similarly, under the KEP search pattern, each bucket contains approximately $(N')^2$ possible meta-keywords, assuming meta-keywords are constructed as keyword pairs. Thus, the attacker faces a conjunctive query recovery problem over a space of $n_B \cdot (N')^2$ candidate meta-keywords.

Once the bucket to which each query keyword $q_{\text{mkw}}^{(i)}$ belongs is determined, the scale of the query recovery problem can be significantly reduced. We refer to this problem as the **Bucket Targeting problem**.

Definition (Bucket Targeting problem).

Let n_B be the number of buckets. Suppose the attacker receives m sub-meta-queries derived from q_{disj} in an arbitrary order. Define a mapping:

$$V_B : \{1, 2, \ldots, m\} \to \{1, 2, \ldots, n_B\}$$

where $V_B(j) = i$ indicates that the j-th received sub-meta-query corresponds to bucket Δ_i.

If the attacker correctly reconstructs such a mapping from the received set of sub-meta-queries, we say that the attacker has successfully solved the **Bucket Targeting problem**.

With the Bucket Targeting problem solved, the attacker can group all received sub-meta-queries associated with a given bucket Δ_i from a query set, and match their corresponding $\widehat{RP}_{(i)}$ values against auxiliary information constructed for that bucket. This enables the attacker to solve smaller subproblems in parallel.

Under the QEP setting, the attacker needs to solve n_B separate single (virtual) keyword recovery problems, each with a reduced space of size at most $2^{N'} - 1$ disjunctive queries. Similarly, in the KEP setting, the attacker needs to solve n_B conjunctive query recovery problems, each over a space of size at most $(N')^2$ meta-keywords.

Solving the Bucket Targeting problem significantly improves the efficiency of the overall query recovery attack. We will present a file-injection-based bucket targeting algorithm in Chapter 4.

Since our passive attack path depends on the problem size, which is typically determined by the number of meta-keywords, it becomes unrealistic to achieve convergence within a limited number of iterations when the problem size is significantly large. In contrast, a smaller meta-keyword space can lead to faster convergence.

We have already observed that under the bucketization setting, the number of potential meta-keywords is reduced from $|\Delta|^2$ to $n_B \cdot (N')^2$. Moreover, once the bucketization problem is solved, each subtask only needs to solve a matching problem with size $(N')^2$. Therefore, bucketization may introduce new vulnerabilities to TWINSSE by reducing the complexity of the meta-keyword recovery task.

3.2 Active Attack: Baseline File-Injection

We introduce a scalable and naïve injection attack towards the disjunctive query setting and state that our Baseline Attack requires the fewest injection files under specific settings. We consider the number of files that need to be injected as our concern of complexity.

3.2.1 Baseline File-Injection Attack Algorithm

The Baseline Attack is a non-adaptive attack, which means that once the attacker finishes injection, it can recover all the rest of the queries.

To inject chosen files into the dataset, the attacker creates a document for each keyword in the keyword space, injects these documents into the target dataset, and records the mapping M between the injected document identifiers and the corresponding keywords.

To recover the queried keywords, the attacker first obtains $\widehat{RP}$ over a query, then queries these identifiers in the mapping M. If a corresponding keyword is successfully retrieved, it is considered one of the queried keywords.

Algorithm 1 has shown how documents are created and injected into the target dataset, and Algorithm 2 has shown how query keywords are recovered.

Algorithm 1 BASELINE_INJECTION(Δ)

```
Initialize an empty mapping M
for each w in Δ do
    Create a file with identifier F_i containing only w
    Inject F_i into the target dataset
    Set M(F_i) ← w
end for
return M
```

Algorithm 2 BASELINE_RECOVER($\widehat{RP}, M$)

```
Initialize an empty result set S ← ∅
for each id in RP^ do
    R ← M(id)
    if R ∈ Δ then
        Add R to S
    end if
end for
return S
```

3.2.2 Attack Setting

File-injection attacks that exploit result patterns, such as the ones proposed in this thesis, rely on a key assumption: the attacker must be able to reliably associate observed returned document identifiers with their corresponding injected documents. This assumption is also adopted in prior work [23], where the authors has discussed its feasibility in practical settings.

Our attack operates in a non-adaptive setting, where the attacker injects a fixed set of files before the meta-database is constructed or updated, and no further updates are made after query processing begins. This design enables full query recovery from a single injection phase.

The same assumptions apply to the improved file-injection method introduced in

Chapter 4. Therefore, we omit repeating these settings and assumptions in that chapter.

For the baseline attack in particular, we also assume that the attacker has knowledge of the complete keyword space.

3.2.3 Correctness

We establish a *lower bound* of $|\Delta|$ on the number of injected files required to guarantee full recovery under non-adaptive attacks against $\text{TWINSSE}_{\text{CSSE}}$ leveraging $\widehat{RP}$, where the attacker is given only the keyword space.

The attacker must prepare a unique and identifiable document combination for each possible query before any queries are issued. Since the attack is non-adaptive and the goal is full recovery, the total number of possible queries, including both single-keyword and disjunctive queries, is:

$$\sum_{i=1}^{|\Delta|} \binom{|\Delta|}{i} = 2^{|\Delta|} - 1.$$

To ensure that each query can be uniquely mapped into a distinct subset of injected documents, the number of possible document combinations must be at least this large. However, if the number of injected files $|D_{\text{injected}}|$ is less than $|\Delta|$, then the total number of possible document combinations is:

$$\sum_{i=1}^{|D_{\text{injected}}|} \binom{|D_{\text{injected}}|}{i} = 2^{|D_{\text{injected}}|} - 1 < 2^{|\Delta|} - 1.$$

In this case, the attacker cannot construct a unique document pattern for every query. Therefore, we conclude that to achieve full recovery, the number of injected files must satisfy:

$$|D_{\text{injected}}| \geq |\Delta|.$$

To ensure the soundness of our baseline recovery method, we must verify that it retrieves exactly the injection documents corresponding to the queried keywords, no more, no less. The following lemma formalizes this guarantee.

Given a meta-query q_{mkw_i} derived from a disjunctive query q_{disj_i}, the baseline recovery will accurately return the document identifiers corresponding to query keywords.

Proof. First, all keywords in q_{disj_i} will lead to the retrieval of their corresponding injection documents. This follows from the correctness guarantee of TWINSSE provided in the original paper.

Second, we show that the injected documents unrelated to q_{disj_i} will not be mistakenly returned as spurious results. Recall the method introduced in Section 2.3.1.2 for

identifying potential spurious results by finding a keyword set C such that $|C| > 1$, $C \cap q_{\text{disj}_i} = \emptyset$, and

$$\text{Conj}\left(\bigcup_{w \in C} F(w, q_{\text{mkw}_i})\right) = q_{\text{mkw}_i}.$$

The spurious result set is then defined as:

$$\text{DB}(\text{Conj}(C)) \setminus \text{DB}(q_{\text{disj}_i}).$$

According to our injection strategy, each injection document contains exactly one keyword. Therefore, no injected document can simultaneously satisfy all keywords in C that $C > 1$, and thus cannot appear in $\text{DB}(\text{Conj}(C))$. As a result, irrelevant injected documents will not be returned as spurious results.

Consequently, all returned injection documents are truly associated with query keywords, enabling accurate intra-bucket index recovery. □

3.2.4 Efficiency

Since this attack requires injecting at least $|\Delta|$ files, it incurs a significantly higher injection cost compared to binary-style single-keyword-based attacks such as those proposed in [3], [23], which only require $\log(|\Delta|)$ injected files. One potential concern is that the sudden injection of a large number of documents may raise suspicion from the client.

The Baseline Attack is scalable in that the attacker only needs to create and inject files containing the keywords they are interested in, rather than injecting $|\Delta|$ files for all possible keywords.

With stronger assumptions in the file-injection attack, such as the attacker having access to keyword frequencies, the bucket parameter, and the keyword space, the number of required injected documents can be significantly reduced. We will introduce an improved file-injection attack in Chapter 4.

4

Bucket-Aware File-Injection

Since Chapter 3 has revealed the potential threat introduced by bucketization, and we proposed the Baseline Attack, this chapter aims to further investigate whether file injection can exploit bucketization to cause more vulnerabilities.

RQ2. *How can we leverage bucketization and the spurious result mechanism to perform a more efficient injection attack?*

We introduce a bucket targeting scheme based on file injection. This approach leverages the spurious result generation mechanism, allowing us to identify accessed buckets through minimal injection within each bucket. Building on this, we propose the **Grid Attack**, a strategy that requires more auxiliary knowledge but enables more fine-grained recovery.

Both injection methods are under stronger auxiliary assumptions, including knowledge of the keyword space and keyword frequencies to construct assignment information of buckets.

4.1 Bucket Targeting

A bucket targeting task refers to the server's ability to determine which bucket is accessed by a given sub-meta-query. Under the auxiliary information that the attacker knows the keywords assigned to each bucket, the attacker constructs a unique document for each bucket. Each document contains the first and last ranked keywords within that bucket and is injected into the target document set. The attacker then records a mapping between the injected document identifiers and their corresponding bucket indices in a map M_B.

To recover the accessed bucket during query execution, the attacker observes the $\widehat{RP}$ from a meta-query q_{mkw} and identifies the injected document identifiers from map the M_B, thus retrieving the associated bucket indexes. This algorithm also supports recovering the bucket accessed by a single sub-meta-query, by mapping each document identifier from $\widehat{RP}_{(i)}$ with M_B. Algorithms 3 and 4 illustrate the pseudo-code of the injection phase and the recover phase of bucket targeting respectively.

Algorithm 3 BUCKET_INJECTION

Initialize map M_B
for each Δ_i in $\{\Delta_1, \Delta_2, \ldots, \Delta_{n_B}\}$ **do**
    Create a document with identifier id containing $\Delta_i[0]$ and $\Delta_i[N'-1]$
    Inject the document id into target document sets
    $M_B(id) \leftarrow i$
end for
return M_B

Algorithm 4 BUCKET_RECOVERY

Input: $\widehat{RP}$, M_B
Instantiate result list R
for each id in $\widehat{RP}$ **do**
    $r \leftarrow M_B(id)$
    if $r \in [n_B]$ **then**
        Append r into R
    end if
end for
return R

4.1.1 Correctness of Bucket Targeting

The attacker includes the first and last ranked keywords from each bucket in the corresponding injected document to ensure that the document is returned as a result whenever any keyword from that bucket is involved in a query.

Let w_1 and w_2 denote the first and last ranked keyword in bucket Δ_i, respectively. Let document d_n be created to contain only w_1 and w_2. Then, if a sub-meta-query $q^{(i)}_{\text{mkw}}$ visits Δ_i, d_n will appear in the result pattern.

Proof. As introduced in Chapter 2, sub-meta-queries are constructed by identifying, within each bucket Δ_i, maximal continuous subsets of keywords that are not included in the queried keywords.

Let $S = \{S_j\}$ denote all such subsets under a sub-meta-query. Each meta-keyword mkw_j in $q^{(i)}_{\text{mkw}}$ is computed as $\Delta_i \setminus S_j$ for some $S_j \in S$, and its value is:

$$\bigcup_{w_k \in \Delta_i \setminus S_j} \texttt{DB}(w_k).$$

Since each S_j is a contiguous interval within Δ_i, excluding both w_1 and w_2 from all $\Delta_i \setminus S_j$ implies that $S_j = \Delta_i$. In this case, no keyword from Δ_i contributes to the query, meaning that $q^{(i)}_{\texttt{mkw}}$ would not access Δ_i, which contradicts our assumption.

Therefore, at least one of w_1 or w_2 must be included in every $\Delta_i \setminus S_j$. As a result,

any document containing both w_1 and w_2, such as d_n, will be returned in the result set under TWINSSE:

$$\texttt{MDB}(q^{(i)}_{\texttt{mkw}}) = \bigcap_{\texttt{mkw} \in q^{(i)}_{\texttt{mkw}}} \texttt{MDB}(\texttt{mkw}) = \bigcap_{S_j \in S} \left(\bigcup_{w_k \in \Delta_i \setminus S_j} \texttt{DB}(w_k) \right).$$

Hence, d_n appears in the result pattern whenever Δ_i is accessed. □

By injecting only two keywords per bucket, we can achieve bucket targeting using the minimal number of keywords in a single injected file.

4.2 Reducing Injected Files

Grid Attack is an attack algorithm that significantly reduces the number of injected files to recover queries keywords.

4.2.1 Column Recovery

The Grid Attack is performed with two dimensions: bucket targeting and column targeting. A column targeting task aims to build a map M_C to recover the positions (i.e., indices) of the queried keywords within their respective buckets, based on the server's observable leakage when processing a (sub-)meta-query. Algorithm 5 and 6 describe how column injection and column recovery works respectively.

Algorithm 5 COLUMN_INJECTION

```
Initialize map M_C
for each i ∈ [N'] do
    Create a document with identifier id
    for each bucket Δ_j ∈ {Δ_1, Δ_2, ..., Δ_{n_B}} do
        Append Δ_j[i] into document id
    end for
    Inject document id into the target dataset
    M_C(id) ← i
end for
return M_C
```

4.2.2 Correctness of Column Recovery

The correctness proof of column recovery algorithm is similar to Lemma 3.2.3.

Given a meta-query q_{mkw_i} derived from a disjunctive query q_{disj_i}, the column recovery can accurately return the document identifiers corresponding to the intra-bucket indices of the queried keywords.

Algorithm 6 COLUMN_RECOVERY

Input $\widehat{RP}$, M_C
Initialize result list R
for each $id \in \widehat{RP}$ **do**
    $r \leftarrow M_C(id)$
    if $r \in [N']$ **then**
        Append r into R
    end if
end for
return R

Proof. Similar to Lemma 3.2.3, we divide the proof into two parts:

1. All relevant injection documents will be correctly returned.
2. No unrelated injection documents will be mistakenly included as spurious results.

Suppose $w_j \in q_{\text{disj}_i}$, and let its intra-bucket index in corresponding bucket Δ_i be k. According to our injection document construction, all keywords at index k in Δ_i are injected into a document with identifier d_l. By the correctness guarantee of TWINSSE, this document d_l will appear in the result set $\texttt{DB}(q_{\text{mkw}_i})$.

To show that no spurious documents are returned, we consider any candidate set C_i introduced in Section 2.3.2.3 that satisfies:

- $|C_i| > 1$,
- $C_i \cap q_{\text{disj}_i} = \emptyset$,
- $\text{Conj}(\bigcup_{w \in C_i} F(w, q_{\text{mkw}_i})) = q_{\text{mkw}_i}$,
- $C_i \subset \Delta_i$.

According to our injection strategy, each injection document contains exactly one keyword from each bucket. Therefore, no injection document contains keywords satisfy such C_i. Consequently, no unrelated injection document will be returned.

□

Hence, all returned injection documents correspond precisely to the queried keywords, enabling accurate column (intra-bucket index) recovery.

4.2.3 Grid Attack

To perform Grid Attack, the attacker constructs a matrix Δ_{B} from the assumed knowledge of keyword space, keyword frequency and bucket parameter n_B and N', where each row corresponds to a bucket and contains the keywords assigned to that bucket, under the assumption of bucket assignment knowledge.

Specifically, $\Delta_B[i]$ stores the keywords of i-th bucket Δ_i, and $\Delta_B[i][j]$ stores the j-th keyword in Δ_i. This matrix explicitly captures the keyword-to-bucket assignment and serves as a basis for subsequent attack steps.

In the injection phase, the attacker consequently performs bucket injection and column recovery injection to the target dataset, as proposed in Algorithms 3 and 5 respectively.

To recover the exact keywords of a received query, Algorithm 7 presents the pseudocode for the recovery phase of the Grid Attack procedure.

Algorithm 7 GRID_ATTACK_RECOVERY

Client:
for each $q_{\text{mkw}}^{(i)}$ derived from q_{disj} **do**
    send $q_{\text{mkw}}^{(i)}$
end for

Server:
Input M_B, M_C, q_{disj}, Δ_B
Initialize result list R
Upon Receiving $q_{\text{mkw}}^{(i)}$:
$\widehat{RP}_{(i)} \leftarrow \text{MDB}(q_{\text{mkw}}^{(i)})$
for each id in $\widehat{RP}_{(i)}$ **do**
    $bucket \leftarrow$ BUCKET_RECOVERY$(\widehat{RP}_{(i)}, M_B)$
    $columns \leftarrow$ COLUMN_RECOVERY$(\widehat{RP}_{(i)}, M_C)$
    for each $index$ in $columns$ **do**
        $r \leftarrow \Delta_B[bucket][index]$
        Append r to R
    end for
end for
return R

4.2.4 Correctness

We now formally state the keyword recovery guarantee provided by Grid Attack.

Upon receiving a set of sub-meta-queries $\{q_{\text{mkw}_i}^{(j)}, \ldots, q_{\text{mkw}_i}^{(k)}\}$ derived from a disjunctive query q_{disj_i}, Grid Attack can deterministically recover the full set of queried keywords in q_{disj_i}.

Proof. According to the TWINSSE query protocol, the disjunctive query q_{disj_i} is split into multiple sub-meta-queries $\{q_{\text{mkw}_i}^{(j)}, \ldots, q_{\text{mkw}_i}^{(k)}\}$, where each sub-meta-query exclusively contains keywords from a single bucket Δ_l.

For each sub-meta-query $q_{\text{mkw}_i}^{(l)}$, the attacker receives the corresponding result pattern $\widehat{RP}_{(l)} \leftarrow \texttt{DB}(q_{\text{mkw}_i}^{(l)})$. By Lemma 4.1.1, the attacker can accurately infer the

corresponding bucket index l based on the unique injection document embedded in $\widehat{RP}_{(l)}$.

Once the bucket Δ_l is determined, Lemma 4.2.2 guarantees the correctness of column recovery: the attacker can recover the in-bucket positions (i.e., column indices) of the queried keywords from $\widehat{RP}_{(l)}$.

Given the auxiliary knowledge of Δ_B, the recovered bucket index l and column index j, the attacker can deterministically recover the exact keywords by querying $\Delta_B[l][j]$.

Since each sub-meta-query $q^{(l)}_{\mathrm{mkw}_i}$ only corresponding to the keyword within Δ_l, by applying this process to each sub-meta-query in $\{q^{(j)}_{\mathrm{mkw}_i}, \ldots, q^{(k)}_{\mathrm{mkw}_i}\}$, the attacker recovers all keywords of q_{disj_i}, thereby fully reconstructing the original disjunctive query. □

4.2.5 Efficiency

The number of files to be injected is $N' + n_B$. Compared to the baseline attack, which requires injecting $|\Delta|$ files, this approach significantly reduces the number of injected files.

However, the total number of keywords contained in all injected documents is $N' \cdot n_B + 2 \cdot n_B = |\Delta| + 2 \cdot n_B$, which exceeds that of the baseline attack, $|\Delta|$, potentially increasing keyword exposure during injection.

5

Hypergraph Matching Attack

In Chapter 3, we assumed an attack to recover meta-keywords from received meta-query set under KEP assumption. In this chapter, we further discuss this *Hypergraph Matching Attack*.

This attack can be divided into two main components:

1. Hypergraph Construction.
2. Simulated Annealing-based hypergraph matching.

We will provide a detailed introduction to the Simulated Annealing algorithm, tailored to different attack objectives.

5.1 Hypergraph Construction

We recall that during the query-interaction phase, the attacker processes a set of conjunctive queries

$$Q_{\text{conj}} = \{q_{\text{conj}_1}, q_{\text{conj}_2}, \ldots, q_{\text{conj}_n}\},$$

along with the observed result pattern RP and keyword-level search equality leakage pattern

$$KEP \leftarrow (\mathbf{T}, \mathcal{T}_K),$$

where:

- $\mathcal{T}_K$ is the set of tags assigned for each keywords in Q_{conj}.
- $\mathbf{T}$ is an $n \times m_{\max}$ matrix of tags.

Each row $\mathbf{T}[i]$ corresponds to the tags produced by query q_{conj_i}, and each tag $\tau \in \mathcal{T}_K$.

5.1.1 Target Hypergraph

We model the observed query leakage as a *target hypergraph*:

$$\mathcal{H}_{\text{T}} = (V_{\text{T}}, \mathcal{E}_{\text{T}}, w_{\text{T}}),$$

where:

1. $V_{\text{T}} = \mathcal{T}_K$;

2. $\mathcal{E}_{\mathrm{T}} = \{e_{\mathrm{T}_i} \mid i \in [n]\}$ is the multiset of hyperedges, where each $e_{\mathrm{T}_i} \in \mathcal{E}_{\mathrm{T}}$ corresponds to the set of non-padding tags in $\mathbf{T}[i]$;

3. w_{T} is the weight function defined as

$$w_{\mathrm{T}}(e_{\mathrm{T}_i}) = \frac{1}{\#\mathrm{docs}_{\mathrm{T}}} \cdot |RP[i]| \quad \text{for each } e_{\mathrm{T}_i} \in \mathcal{E}_{\mathrm{T}},$$

assigning each hyperedge a weight proportional to the size of its query result. Here, $RP[i] \leftarrow \mathrm{DB}(q_{\mathrm{conj}_i})$ is the result pattern as defined in Chapter 3, and $\#\mathrm{docs}_{\mathrm{T}}$ denotes the total number of documents in the target dataset. The scaling factor $\frac{1}{\#\mathrm{docs}_{\mathrm{T}}}$ ensures comparability with weights in the auxiliary hypergraph.

In other words, each conjunctive query q_{conj} corresponds to a hyperedge whose vertices are the tag set by observing KEP, and whose weight reflects the normalized result size. For simplicity, repeated queries are typically represented as a single hyperedge in the hypergraph.

5.1.2 Auxiliary Hypergraph

We next build an *auxiliary hypergraph*

$$\mathcal{H}_{\mathrm{A}} = (V_{\mathrm{A}}, \mathcal{E}_{\mathrm{A}}, w_{\mathrm{A}})$$

based on the attackers side information. Recall from Chapter 2 that `Data` stores the mapping from each keyword to its document-identifier set[1].

After extracting this mapping, the adversary proceeds as follows:

1. Set $V_{\mathrm{A}} = \Delta$ to be the complete keyword space.

2. For each $i = 1, 2, \ldots, m_{\max}$, enumerates all subsets $s \subseteq \Delta$ such that $|s| = i$, and $\mathrm{Conj}(s)$ is a legal meta-query.

3. For each such s:

   - Compute the co-occurrence volume:

$$v_s = \left| \bigcap_{w \in s} \mathtt{Data}(w) \right|.$$

   - Add the hyperedge $e_{\mathrm{A}} = s$ to $\mathcal{E}_{\mathrm{A}}$ and assign its weight as:

$$w_{\mathrm{A}}(e_{\mathrm{A}}) = \frac{1}{\#\mathrm{docs}_{\mathrm{A}}} \cdot v_s.$$

   Here, $\#\mathrm{docs}_{\mathrm{A}}$ is the total number of documents in the auxiliary dataset, and the factor $\frac{1}{\#\mathrm{docs}_{\mathrm{A}}}$ scales the weights to be comparable to those in $\mathcal{H}_{\mathrm{T}}$.

[1] We avoid using `DB` and related terminology here, as `DB` refers to the encrypted index in the real system, which is inaccessible to the attacker. The attackers side information is modeled via the plaintext-level `Data` structure.

Thus, $\mathcal{E}_{\mathrm{A}}$ consists of all keyword subsets of size up to $m_{\max}$ with non-zero co-occurrence volume, and w_{A} records their normalized frequencies.

By construction, we have:

$$|V_{\mathrm{A}}| \geq |V_{\mathrm{T}}| \quad \text{and} \quad |\mathcal{E}_{\mathrm{A}}| \geq |\mathcal{E}_{\mathrm{T}}|.$$

While the theoretical definition of $\mathcal{E}_{\mathrm{A}}$ includes all keyword subsets of size up to $m_{\max}$, its size in practice remains tractable. This is because most conjunctive queries involve only two or three keywords on average [24], and we exclude all subsets with zero co-occurrence volume during actual construction. As a result, the constructed hypergraph is much sparser than its theoretical maximum.

5.2 Hypergraph Matching

After constructing the auxiliary and target hypergraphs, we aim to find a mapping from the tag space to the keyword space. Formally, we seek an injective mapping:

$$M_{\mathrm{HM}} : V_T \hookrightarrow V_A,$$

such that

$$|V_T| < |V_A| \quad \text{and} \quad \forall\, v_T, v'_T \in V_T,\ v_T \neq v'_T \implies M_{\mathrm{HM}}(v_T) \neq M_{\mathrm{HM}}(v'_T).$$

We construct this mapping using a simplified Simulated Annealing (SA)-based stochastic iterative optimization approach.

Algorithm 8 SIMULATED_ANNEALING()

```
mapping ← INITIALIZATION()
score ← SCORING(mapping)
temperature ← initial_temperature
for i = 1 to max_iter do
    new_mapping ← NEIGHBORING(mapping)
    new_score ← SCORING(new_mapping)
    if ACCEPT(score, new_score, temperature) then
        mapping ← new_mapping
        score ← new_score
    end if
    temperature ← temperature × cooling_rate
end for
return mapping
```

A Simulated Annealing (SA) implementation mainly consists of four subroutines: **Initialization**, **Scoring**, **Neighboring**, and **Accept**. The SA framework is flexible and scalable, allowing different subroutine designs to accommodate various tasks—for example, keyword-oriented or meta-keyword-oriented conjunctive query recovery.

In the context of a hypergraph matching attack, the roles of these subroutines are as follows:

- **Initialization**: Provides an initial injective mapping M_{HM} from the target vertex set V_T to the auxiliary vertex set V_A.
- **Scoring**: Evaluates the similarity between the auxiliary and target hypergraphs under the current mapping M_{HM}.
- **Neighboring**: Generates a new candidate mapping based on the current one.
- **Accept**: Determines whether the new mapping should be accepted as the current best mapping.

5.2.1 A Naïve Matching Strategy

We propose a general-purpose matching scheme for hypergraph matching attacks based on the Simulated Annealing (SA) framework. In this scheme, the **Initialization** subroutine produces a randomly generated injective mapping M_{HM}. Specifically, we organize the auxiliary vertex set V_{A} and the target vertex set V_{T} into two indexable sequences: $S_{V_{\mathrm{A}}}$ and $S_{V_{\mathrm{T}}}$. Then, $S_{V_{\mathrm{A}}}$ is randomly shuffled, and a mapping is established by associating each element of $S_{V_{\mathrm{T}}}$ with the corresponding element in the shuffled $S_{V_{\mathrm{A}}}$. Formally:

$$M_{\mathrm{HM}}(S_{V_{\mathrm{T}}}[i]) \leftarrow S_{V_{\mathrm{A}}}[i], \quad i \in \{1, \ldots, |V_{\mathrm{T}}|\}.$$

The **Neighboring** subroutine is defined in the same way as the initialization step, by generating a completely new random injective mapping from V_{T} to V_{A}. To evaluate the similarity between the auxiliary and target hypergraphs under a mapping M_{HM}, we first apply the current mapping to each hyperedge $e_{\mathrm{T}} \in \mathcal{E}_{\mathrm{T}}$ of the target hypergraph. That is, each vertex $v_{\mathrm{T}} \in e_{\mathrm{T}}$ is mapped to $M_{\mathrm{HM}}(v_{\mathrm{T}})$, producing a new set of hyperedges denoted as $\mathcal{E}_{\mathrm{T}}^{\mathrm{mapped}}$, where each mapped hyperedge is formally defined as:

$$e_{\mathrm{T}}^{\mathrm{mapped}} = \{M_{\mathrm{HM}}(v_{\mathrm{T}}) \mid v_{\mathrm{T}} \in e_{\mathrm{T}}\},$$

and we have $e_{\mathrm{T}}^{\mathrm{mapped}} \in \mathcal{E}_{\mathrm{A}}$. Semantically, each hyperedge in $\mathcal{E}_{\mathrm{T}}^{\mathrm{mapped}}$ corresponds to a set of keywords, while the auxiliary hypergraph $\mathcal{H}_{\mathrm{A}}$ covers all the potential combinations of keywords. This allows the scoring function to measure how the mapped target hypergraph deviates from the auxiliary hypergraph in terms of edge presence and weight. The **scoring function** is defined as the sum of squared biases, where each bias is the weight difference between a mapped and original hyperedge:

$$\mathrm{bias} = w_{\mathrm{A}}(e_{\mathrm{T}}^{\mathrm{mapped}}) - w_{\mathrm{T}}(e_{\mathrm{T}}),$$

$$\mathrm{Score}(M_{\mathrm{HM}}) = \sum_{e_{\mathrm{T}} \in \mathcal{E}_{\mathrm{T}}} (\mathrm{bias})^2.$$

In standard SA algorithms, the **Accept** function typically uses a temperature-based probabilistic criterion to occasionally accept worse solutions in order to escape local

minima. However, in our simplified variant, since the neighboring function always generates a random mapping and does not reduce its perturbation over time, we adopt a greedy acceptance policy: we only accept the new mapping if it improves the score.

We also propose a theoretically more semantic-aware approach, which differs from the baseline in both the initialization and scoring strategies. Although this method did not achieve the expected performance in our experiments, we provide a detailed description in Appendix A.

5.2.2 Complexity of Hypergraph Matching

When the maximum number of hyperedges in the target hypergraph is denoted by k, and the number of iterations is m, the total number of hyperedge matching and distance computations is $O(mk)$.

Let the maximum dimension of any sub-meta-query be n, and recall that n_B is the number of buckets and N' is the bucket size, with $n < N'$. In each iteration of the hypergraph matching procedure, the maximum number of hyperedges to be scored (i.e., matched and evaluated by distance) is bounded by

$$k < n_B \cdot \sum_{i=1}^{n} \binom{N'}{i}.$$

Therefore, in the worst case, the total number of hyperedge matchings and distance computations across all iterations is

$$O\left(m \cdot n_B \cdot \sum_{i=1}^{n} \binom{N'}{i}\right).$$

Using the identity $\sum_{i=1}^{n} \binom{N'}{i} = 2^{N'} - \binom{N'}{0} - \sum_{i=n+1}^{N'} \binom{N'}{i}$, and noting that nN', we can upper bound this sum by $2^{N'}$, giving:

$$O\left(m \cdot n_B \cdot 2^{N'}\right).$$

Once the bucket targeting problem is solved, the distance between each target hyperedge and the corresponding auxiliary hyperedge can be computed independently within each bucket. This allows the use of a multi-process approach. In this case, each iteration process handles at most

$$O\left(m \cdot \sum_{i=1}^{n} \binom{N'}{i}\right) = O\left(m \cdot 2^{N'}\right)$$

hyperedge matchings and distance computations.

6

Evaluation

Since our proposed Baseline Attack and Grid Attack is for achieving fully recovery and its number of injected file is deterministic and can be easily computed when given the keyword space and bucket size, this chapter focuses primarily on the passive attack.

In this chapter, we evaluate the effectiveness of our attack path with *Hypergraph Matching Attack* and under the assumption of solving bucket targeting problem. More specifically, we try to recover original queried keywords from disjunctive queries under TWINSSE scheme.

We are proposed to investigate the following research questions:

RQ3. *Is TWINSSE$_{CSSE}$ vulnerable to query recovery attack when the underlying CSSE scheme leaks KEP?*

To answer **RQ3**, we construct a *correct-recovery matrix* to record the recovery outcomes:

- The i-th row corresponds to recovery results for i-dimensional queries;
- The j-th column indicates the rate of **Correctly Recovered Keywords (CRT)**;
- The entry at position $[i, j]$ represents the proportion of i-dimensional queries for which exactly j CRT.

6.1 Experimental Settings

6.1.1 General Settings in Hypergraph Matching Attack

In the *Hypergraph Matching Attack*, various experimental settings may significantly influence recovery performance. We aim to evaluate how the following general settings affect the effectiveness of the attack.

Bucket Setting in TWINSSE

The setup parameter *bucket size* determines the problem scale of meta-keyword recovery within each bucket. We aim to investigate whether a smaller problem scale will improve the performance of convergence under limited and fixed iteration time in our attack.

Keyword Frequencies

The keyword space is defined by the client, and the selection of keywords substantially impacts the recovery rate. A natural factor to consider is the *frequency distribution* of keywords in the dataset.

In this experiment, we use the NLTK toolkit[1] to extract all real words from the target dataset and randomly select some of them to form keyword space. We also introduce an additional setting: selecting the top $n\%$ most frequent words as keywords.

Intuitively, higher-frequency words carry stronger semantic characteristics and are more likely to exhibit identifiable patterns in conjunctive queries. We aim to investigate whether selecting higher-frequency keywords (i.e., using a smaller n) leads to improved recovery performance.

Auxiliary Data

The source and quality of auxiliary data determine its similarity to the target dataset, which directly impacts the success of our Hypergraph Matching Attack.

To simulate different levels of similarity, we consider two sampling strategies introduced in Chapter 1:

- **Internal**: Sampling a fraction of documents from the target dataset itself as the auxiliary dataset. Specifically, the first n documents from every *ten* consecutive documents
- **External**: Splitting the dataset source into two disjoint halves, one for auxiliary data and the other for target data.

By varying the sampling fraction, we evaluate how the similarity between auxiliary and target datasets affects recovery performance.

6.1.2 Hardware Setting

Our experiments were conducted on servers provided by AutoDL[2]. The instance configuration is as follows:

Software	Python 3.12 (Ubuntu 22.04)
CPU	9 vCPUs, Intel(R) Xeon(R) Platinum 8255C @ 2.50GHz
RAM	48 GB

[1] `https://www.nltk.org/`
[2] `https://www.autodl.com/`

6.2 Attack on TWINSSE

To employ hypergraph attack, this experiment is built upon the assumption that the underlying $\text{TWINSSE}_{\text{CSSE}}$ also leaks according to the KEP leakage profile.

We re-implemented the key functionalities of TWINSSE in our experiment code, including the generation of meta-queries and the construction of the meta-database.

We selected *10,000,000 documents* and set the *keyword space size is fixed at 500*. All our experiments on TWINSSE are conducted based on this fixed keyword space.

We generated a total of *3000 disjunctive queries*, assuming that the maximum dimension of disjunctive queries is *3*. The distribution of 1 dimensional, 2 dimensional, and 3 dimensional disjunctive queries is set to *0.3, 0.3, and 0.4*, respectively.

In particular, we also assume that the Bucket Targeting problem has been solved under the Bucket Targeting scheme proposed in Section 4.

6.2.1 Under Different Keyword Frequencies

We evaluate how keyword frequency affects query recovery. In this experiment, we use *50% of the internally extracted documents* as auxiliary data, and set the *bucket size to 10*.

We randomly sample *500 keywords* from the *top 20%, 40%, 60%, and 80%* frequency ranges respectively. As the frequency range broadens, a larger proportion of low-selectivity keywords is included. This allows us to observe how the presence of high- vs. low-frequency keywords impacts query recovery performance.

The recovery performance under these settings is summarized in Tables 6.1, 6.2, 6.3 and 6.4, using top *top 20%, 40%, 60% and 80%* frequency keywords, respectively.

Table 6.1: Recovery of top 20% frequency sampled keywords

#CRT	1-D Query	2-D Query	3-D Query
0 keyword	77.78%	60.33%	43.92%
1 keyword	22.22%	34.44%	41.25%
2 keywords	–	5.22%	13.92%
3 keywords	–	–	0.92%

Table 6.2: Recovery of top 40% frequency sampled keywords

#CRT	1-D Query	2-D Query	3-D Query
0 keyword	79.33%	60.11%	47.25%
1 keyword	20.67%	36.00%	40.58%
2 keywords	–	3.89%	11.25%
3 keywords	–	–	0.92%

Table 6.3: Recovery of top 60% frequency sampled keywords

#CRT	1-D Query	2-D Query	3-D Query
0 keyword	84.11%	69.33%	56.83%
1 keyword	15.89%	27.78%	35.25%
2 keywords	–	2.89%	7.17%
3 keywords	–	–	0.75%

Table 6.4: Recovery of top 80% frequency keywords

#CRT	1-D Query	2-D Query	3-D Query
0 keyword	86.78%	75.78%	63.42%
1 keyword	13.22%	22.22%	31.33%
2 keywords	–	2.00%	5.17%
3 keywords	–	–	0.08%

Analysis. We observe that as the keyword frequency range expands, the overall recovery performance declines. This is likely due to the increasing presence of low-selectivity keywords, which weakens the distinctiveness of co-occurrence semantic features and introduces greater ambiguity in query matching.

6.2.2 Under different Bucket Size

We conduct experiments under a fixed keyword space and a fixed auxiliarytarget document set pair. The entire sampled document set is used as the target dataset, while the auxiliary dataset is constructed by internally extracting *50%* of the documents from the target set. We randomly extract top *50%* most frequent words as our candidate keywords, and the auxiliary documents are sampled *50%* internally. We also fix the bucket size to *10*.

We evaluate the impact of different bucket sizes $N' \in \{20, 10, 5\}$ on recovery performance. Tables 6.5, Table 6.6 and Table 6.7 showed the result under these settings, respectively.

Table 6.5: Correctly recovered keywords under $N' = 5$.

# CRT	1-D Query	2-D Query	3-D Query
0 keyword	82.56%	73.00%	58.33%
1 keyword	17.44%	24.67%	34.42%
2 keywords	–	2.33%	6.67%
3 keywords	–	–	0.58%

Table 6.6: Correctly recovered keywords under $N' = 10$.

# CRT	1-D Query	2-D Query	3-D Query
0 keyword	84.00%	69.33%	54.67%
1 keyword	16.00%	27.56%	37.25%
2 keywords	–	3.11%	7.33%
3 keywords	–	–	0.75%

Table 6.7: Correctly recovered keywords under $N' = 20$.

# CRT	1-D Query	2-D Query	3-D Query
0 keyword	69.33%	49.67%	34.25%
1 keyword	30.67%	42.00%	43.83%
2 keywords	–	8.33%	19.25%
3 keywords	–	–	2.67%

Analysis. Intuitively, we expected that smaller bucket sizes would yield better convergence and matching results under a fixed number of SA iterations. However, our experimental findings reveal the opposite: smaller bucket sizes lead to worse recovery performance.

This result is likely due to the reduced number of queries in each bucket. When fewer queries are present, the hypergraph constructed from leakage information becomes less informative, as fewer meaningful co-occurrences are captured within each bucket. This reduction in semantic density negatively impacts the effectiveness of the matching process.

6.2.3 Under Different Degrees of Similarity

We evaluate how different levels of similarity influence the performance of keyword recovery. In this experiment, we fix the bucket size to 10 and the keywords are extracted from the top *50%* of the words. We consider three document extraction settings: *external sampling*, *50% internal sampling*, *90% internal sampling* and *100% internal sampling*.

Under each extraction method, we use a matrix to provide a rough, indicative measure of the similarity between auxiliary and target data: specifically, the fraction of keywords at the same index position that match between the buckets constructed from auxiliary and target datasets. This measure serves as an indicator of alignment quality. Even if the meta-keywords are completely recovered, poor alignment in keyword order can still lead to incorrect recovery at the corresponding positions.

We emphasize again that our attack is *not* a ground-truth or inference-style attack, even though it adopts an internal sampling strategy for constructing auxiliary information. The purpose of extracting a portion of target documents is solely to obtain an auxiliary dataset that shares a similar distribution with the target dataset.

The results are illustrated in Table 6.8, 6.9, 6.10 and 6.11 representing *external sampling*, *50% internal sampling*, *90% internal sampling* and *100% internal sampling* respectively.

Table 6.8: External extraction with **10.2%** correctly aligned ordered keywords.

# CRT	1-D Query	2-D Query	3-D Query
0 keyword	87.44%	73.44%	61.00%
1 keyword	12.56%	25.00%	32.33%
2 keywords	–	1.56%	6.42%
3 keywords	–	–	0.25%

Table 6.9: Internal extraction with 50% known documents and **16.2%** correctly aligned ordered keywords.

# CRT	1-D Query	2-D Query	3-D Query
0 keyword	82.56%	73.00%	58.33%
1 keyword	17.44%	24.67%	34.42%
2 keywords	–	2.33%	6.67%
3 keywords	–	–	0.58%

Table 6.10: Internal extraction with 90% known documents and **27.6%** correctly aligned ordered keywords.

# CRT	1-D Query	2-D Query	3-D Query
0 keyword	75.78%	59.00%	45.25%
1 keyword	24.22%	36.22%	40.58%
2 keywords	–	4.78%	12.42%
3 keywords	–	–	1.75%

Table 6.11: Internal extraction with 100% known documents and **100%** correctly aligned ordered keywords.

# CRT	1-D Query	2-D Query	3-D Query
0 keyword	59.11%	36.00%	23.42%
1 keyword	40.89%	46.56%	43.08%
2 keywords	–	17.44%	29.17%
3 keywords	–	–	4.33%

Analysis. When the adversary does *not* possess complete knowledge of the target documents, we observe a low proportion of correctly aligned keywords, which limits the overall recovery performance.

Even when the attacker has full knowledge of the target document set—meaning all keywords are correctly aligned—our recovery result remains limited. We examined the auxiliary hyperedges representing Δ_1 and found that *20% of the edges share the same co-occurrence volume*, indicating that low-selectivity keywords appear frequently across documents. This lack of discrimination hinders effective recovery for such keywords.

There are several directions for extending this experiment:

1. We believe that additional metrics can be introduced to better evaluate document similarity. Our current measurement primarily guides the step of recovering keywords from meta-keywords. Ideally, we would like to incorporate a metric that captures *semantic similarity between documents*, enabling a deeper understanding of the extracted auxiliary data and a more robust interpretation of the attack results.

2. In our current setup, keywords and queries are randomly sampled. We hope to obtain a *real-world dataset* of user queries and keywords to better evaluate the attack performance, especially for queries that are currently under-recovered.

3. Since our attack can partially recover the original query keywords, it can be integrated with the QCCP attack [20] to further improve overall performance. Moreover, we can adapt our SA scheme in the Hypergraph Matching Algorithm to obtain better meta-keyword recovery results.

Our attack may not produce entirely satisfactory recovery results. However, compared to the original paper [13], we successfully extended the experiments to higher-dimensional disjunctive queries, and our experimental setting better reflects realistic query scenarios.

6.3 Execution Time

Table 6.12 illustrates the number of files we injected for performing bucket targeting under different bucket sizes in our passive attack flow. Given that the total number of files to be outsourced in our experiment is 10,000,000, which is significantly larger than the number of injected files, we consider our injection to be inconspicuous. As shown, the number of injected files is inversely proportional to the bucket size, since the number of injected files is equal to the number of buckets.

Table 6.12: Number of Injected Files for Bucket Targeting under Different Bucket Sizes

Bucket Size	**# Injected Files**
5	100
10	50
20	25

We implement our matching-based attack using Python, and under the assumption that bucket targeting is resolved, we utilize 8 processors to parallelize the attack. With 10,000,000 iterations per bucket-based sub-task, the average time to complete one full attack instance is approximately 80 minutes. We also observe that the runtime of each sub-task is sensitive to the number of hyperedges in the target hypergraph, which in turn depends on the actual number of queries issued.

Within a reasonable time budget, faster execution allows for more iterations, which may improve the quality of the matching and thus the overall effectiveness of the attack. Therefore, reducing execution time is an important direction for improving attack efficiency.

Currently, our implementation is written in Python, where hypergraphs are represented using dictionaries. If we could represent hypergraphs as tensors, we might be able to leverage CUDA for GPU acceleration. Additionally, reimplementing the attack in C++ may further improve computational efficiency.

7

Conclusion

In this paper, we investigate the security vulnerabilities of TWINSSE.

For passive attacks, we show that if the underlying CSSE scheme in $\text{TWINSSE}_{\text{CSSE}}$ leaks KEP, the system becomes vulnerable once bucket targeting is solved. Similarly, if leaks QEP, the scheme may also become susceptible to inference attacks.

For active attacks, we examine file injection strategies that exploit the spurious result mechanism and bucketization structure. Specifically, we propose a file-injection-based bucket targeting method, a generalized disjunctive query attack we call the Baseline Attack, and a tailored disjunctive query attack for TWINSSE, referred to as the Grid Attack.

7.1 Revisiting Research Questions

We now provide a summary of our responses to the previously stated research questions.

Response to RQ1: *Are there passive attack paths that threaten $TWINSSE_{CSSE}$ under different levels of search pattern leakage (e.g., QEP and KEP)?*

We address this question in Chapter 3, where we define the threat model and analyze the potential threats posed by passive attacks under different leakage patterns.

We propose a two-step attack strategy: the adversary first attempts to recover *meta-keywords* in a meta-query, and subsequently infers the original query keywords from them. This approach is analyzed under two common leakage scenarios in underlying CSSE schemes:

- Under **QEP leakage**, we treat a conjunctive query expression as a *virtual keyword* and apply single-keyword recovery attacks to the observed queries.
- Under **KEP leakage**, we consider adversaries aiming to recover all meta-keywords. To this end, we describe a conjunctive CSSE attack that leverages

co-occurrence information. The concrete implementation of this attack is deferred to Chapter 5.

We further observe that the *bucketization* mechanism in TWINSSE weakens security by reducing the number of meta-keywords stored in the database, thus narrowing the attack space and simplifying the adversary's task.

Additionally, we formally define the *bucket targeting* problem and demonstrate that solving this problem significantly improves the effectiveness of passive attacks.

Response to RQ2: *How can we leverage bucketization and the spurious result mechanism to perform a more efficient injection attack?*

We address this question in Chapter 4. We begin by proposing a *bucket targeting algorithm*, which injects only two keywords into each file and only need to inject n_B files. This guarantees that any accessed bucket will be retrieved from the result pattern of a sub-meta-query.

We then exploit the fact that spurious results are generated *within* each bucket. Based on this observation, we design a *column recovery algorithm* to identify the specific keyword index (column position) in a sub-meta-query that leads to access to a particular bucket.

Finally, by combining both the *horizontal dimension* (bucket targeting) and the *vertical dimension* (column targeting), we propose the *Grid Attack*. This strategy significantly reduces the number of injected files required to extract meaningful keyword information.

Depending on the auxiliary information available to the attacker, different attack strategies can be selected. When only the knowledge of the keyword space is known, the attacker is limited to performing the Baseline Attack. However, if the attacker also knows the ordering of keywords (e.g., by frequency), the Grid Attack can be applied.

Moreover, the Grid Attack is specific to TWINSSE due to its reliance on the spurious results and it leverages bucketization mechanism. In contrast, the Baseline Attack is more broadly applicable: it can be used against disjunctive queries that do not employ search result pattern leakage suppression. Furthermore, because it leverages the presence of spurious results, the baseline attack remains compatible with TWINSSE.

Response to RQ3: *Is $TWINSSE_{CSSE}$ vulnerable to query recovery attack when the underlying CSSE scheme leaks KEP?*

The original TWINSSE paper [13] conducted a preliminary attack under QEP leakage. In Chapter 6, we extend this line of research by implementing a full hypergraph matching attack under the assumption of KEP leakage in the underlying CSSE scheme.

We evaluate our attack across various parameter settings, including different auxiliary data sampling strategies, keyword frequency distributions, and bucket sizes. All experiments are conducted under the assumption that bucket targeting has already been resolved. We present and analyze the attack results under these varying conditions.

To improve upon prior evaluation methods [13], [20], we propose a more fine-grained metric for assessing Boolean query recovery accuracy.

Although the overall recovery rates are not high in absolute terms, we argue that our attack flow remains valuable due to the following reasons:

1. It supports Boolean query recovery in multiple dimensions, more closely reflecting realistic query scenarios.

2. It relies on a naïve structure analysis (SA) approach, leaving room for further improvement with more advanced techniques.

7.2 Highlights of Contributions

We summarize our contributions from two perspectives: **techniques** and **insights**.

Technical Contributions

1. We formally define the notions of *Query-level Equality Pattern (QEP)* and *Keyword-level Equality Pattern (KEP)*, which model the observable equality patterns in search queries under CSSE schemes.

2. We present a passive attack workflow under both QEP and KEP leakage in Chapter 3. This includes a naïve hypergraph matching attack in Chapter 5, with an optimized variant discussed in Appendix A. In Chapter 6, we evaluate the attack under specific assumptions and propose a more fine-grained metric for Boolean query recovery evaluation.

3. We formally define the *bucket targeting problem* in the context of TWINSSE and propose an optimal file-injection based target bucketing algorithm that minimizes the number of injected keywords. We also prove the correctness of this approach.

4. We propose a series of active attacks in Chapters 3 and 4, including a generalized *Baseline Attack* for disjunctive queries and a TWINSSE-specific *Grid Attack*, which leverages both horizontal and vertical attack dimensions.

Conceptual Insights

1. We demonstrate that TWINSSE exhibits *security risks*, particularly when its search equality pattern is exposed under KEP leakage. In such cases, attackers can partially recover some encrypted queries. Compared to the original cryptanalysis in [13], our extended cryptanalysis offers more detailed and thorough attack workflows.

2. In Chapters 2 and 3, we observe that the *bucketization mechanism* may introduce vulnerabilities by reducing the (meta-)keyword space. This simplification may reduce the keyword set and make mapping and recovery easier for the adversary.

3. In Chapter 2, we introduce a procedure for identifying potential *spurious results*, and we exploit the characteristics of these results to propose injection-based targeting algorithms. We hope this deepens the reader's understanding of how spurious results are generated and encourages the development of more robust attack strategies.

7.3 Limits and Future Work

We present the limitations and potential future directions of our work in the form of open questions:

1. **How to construct a more robust hypergraph matching attack?**

   Our current hypergraph matching attack demonstrated limited robustness in our experiments. To address this, we proposed a potentially more semantic-aware implementation in Appendix A, which we hope offers better insight. Compared to the co-occurrence graph matching attack [4], a hypergraph can encode higher-dimensional Boolean search and co-occurrence results, which capture richer semantic co-occurrence features.

   However, a known weakness of hypergraph-based structures lies in their inherent complexity. Achieving convergence more efficiently and accurately motivates the development of more suitable and adaptive neighbor-selection subroutines. We also expect that such improvements can be applied to attacks targeting TWINSSE. Additionally, we are curious whether the co-occurrence of meta-keywords can effectively capture semantic features of the queries.

2. **Is there a statistical bucket targeting algorithm to accurately match as many sub-meta-queries into their targets as possible?**

   Our current bucket targeting algorithm requires adversarial file-injection capabilities and relies on the attacker's knowledge of keyword ordering. We leave it as an open problem to design a statistical bucket recovery algorithm that can operate under different search equality patterns, without relying on file injection or exact keyword order knowledge.

3. **Are OXT and HXT secure enough?**

   Although our attacks are built upon QEP and KEP leakage models, there is currently no formal proof that OXT [11] conforms to either model. Nevertheless, if OXT is vulnerable to statistical attacks, then its use within the TWINSSE framework (i.e., $\text{TWINSSE}_{\text{OXT}}$) may also be susceptible to similar attack strategies.

4. **Is there an active attack that leverages volume patterns to exploit TWINSSE?**

   Our previously proposed active attacks rely on the ability of the adversary to identify their injected documents in the query results. However, this assumption may restrict their applicability in real-world scenarios. We are interested in exploring active attack strategies that instead exploit Volume Pattern to infer query structure or keyword relationships. Such attacks could potentially avoid the need for explicit document identification, thereby increasing their practicality and stealth.

In Chapter 3, we also discussed some potential directions to improve the attack that directly recovering received meta-keywords to disjunction of known keywords.

Bibliography

[1] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006, pp. 79–88.

[2] S. Oya and F. Kerschbaum, "Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 127–142.

[3] L. Blackstone, S. Kamara, and T. Moataz, *Revisiting leakage abuse attacks*, Cryptology ePrint Archive, 2019.

[4] D. Pouliot and C. 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, pp. 1341–1352.

[5] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 668–679.

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

[7] R. Bost, "$\sum$Oφoς: Forward secure searchable encryption," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1143–1154.

[8] I. Miers and P. Mohassel, *Io-dsse: Scaling dynamic searchable encryption to millions of indexes by improving locality*, Cryptology ePrint Archive, 2016.

[9] S.-F. Sun, R. Steinfeld, S. Lai, *et al.*, "Practical non-interactive searchable encryption with forward and backward privacy," in *USENIX Network and Distributed System Security Symposium (NDSS)*, 2021.

[10] Y. Wang and D. Papadopoulos, "Multi-user collusion-resistant searchable encryption with optimal search time," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 252–264.

[11] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rou, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 1822, 2013. Proceedings, Part I*, Springer, 2013, pp. 353–373.

[12] S. Lai, S. Patranabis, A. Sakzad, *et al.*, "Result pattern hiding searchable encryption for conjunctive queries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 745–762.

[13] A. Bag, D. Talapatra, A. Rastogi, S. Patranabis, and D. Mukhopadhyay, "Two-in-one-sse: Fast, scalable and storage-efficient searchable symmetric encryption for conjunctive and disjunctive boolean queries," *Proceedings on Privacy Enhancing Technologies*, 2023.

[14] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre, "Seal: Attack mitigation for encrypted databases via adjustable leakage," in *29th USENIX Security Symposium (USENIX Security 20)*, `https://www.usenix.org/conference/usenixsecurity20/presentation/demertzis`, USENIX Association, 2020, pp. 2433–2450.

[15] S. Kamara and T. Moataz, "Computationally volume-hiding structured encryption," in *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019. Proceedings, Part II*, Springer, 2019, pp. 183–213.

[16] S. Patel, G. Persiano, K. Yeo, and M. Yung, "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*, 2019, pp. 79–93.

[17] Z. Gui, K. G. Paterson, and S. Patranabis, "Rethinking searchable symmetric encryption," in *2023 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2023, pp. 1401–1418.

[18] S. Kamara and T. Moataz, *Bayesian leakage analysis: A framework for analyzing leakage in encrypted search*, IACR Cryptology ePrint Archive, 2023:813, 2023.

[19] A. Boldyreva, Z. Gui, and B. Warinschi, *Understanding leakage in searchable encryption: A quantitative approach*, IACR Cryptology ePrint Archive, 2024.

[20] H. Liu, L. Xu, X. Liu, L. Mei, and C. Xu, "Query correlation attack against searchable symmetric encryption with supporting for conjunctive queries," *IEEE Transactions on Information Forensics and Security*, 2025.

[21] S. Oya and F. Kerschbaum, "{Ihop}: Improved statistical query recovery against searchable symmetric encryption through quadratic optimization," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2407–2424.

[22] C. Liu, L. Zhu, M. Wang, and Y.-a. Tan, "Search pattern leakage in searchable encryption: Attacks and new construction," *Information Sciences*, vol. 265, pp. 176–188, 2014.

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

[24] B. J. Jansen, A. Spink, and T. Saracevic, "Real life, real users, and real needs: A study and analysis of user queries on the web," *Information Processing and Management*, vol. 36, no. 2, pp. 207–227, 2000.

A

Semantic-Aware Hypergraph Matching SA Scheme

We introduce a more specific semantic-aware hypergraph matching SA scheme in this appendix.

In our improved SA implementation, we consider weighted contributions from different query dimensions, which provide varying levels of *semantic reliability* for conjunctive queries. However, this assumption does not hold for `TWINSSE`, as its meta-queries represent disjunctions over multiple keywords, and the attacker cannot determine the true dimensionality of the original query from a (sub-)meta-query alone.

The approach begins with an **initialization phase** that pre-matches certain tags to keywords. These pre-matched tags are excluded from the random shuffling in the neighboring phase, effectively reducing the search space for matching. This allows the algorithm to produce more accurate matches within a limited number of iterations.

A.1 Initialization and Neighboring

In the **initialization phase**, we focus on *unary hyperedges*, each of which consists of a single vertex. The weight of each unary hyperedge represents the *frequency of an individual keyword*. The procedure is as follows:

1. Enumerate all unary hyperedges $e_{\mathrm{T}_i} = \{v_{\mathrm{T}_i}\}$ in $\mathcal{H}_{\mathrm{T}}$.
2. For each e_{T_i}, search $\mathcal{H}_{\mathrm{A}}$ for candidate unary hyperedges $e_{\mathrm{A}_j} = \{v_{\mathrm{A}_j}\}$. Define the matching cost as:

$$\mathrm{cost}(i,j) = \Big(w_{\mathrm{A}}(e_{\mathrm{A}_j}) - w_{\mathrm{T}}(e_{\mathrm{T}_i})\Big)^2.$$

   Select the candidate v_{A_j} with the lowest cost and assign the mapping:

$$M_{\mathrm{HM}}(v_{\mathrm{T}_i}) \leftarrow v_{\mathrm{A}_j}.$$

3. If the best candidate v_{A_j} has already been mapped to another v_{T_k}, continue through the candidate list and choose the next available unmapped vertex.

4. Once matched, remove the corresponding vertices from S_{V_A} and S_{V_T}.

In the **neighboring phase**, we perform a *random shuffle* on the remaining S_{V_A}. The vertices in S_{V_T} are then matched to the shuffled sequence of S_{V_A} in order, forming new neighboring candidates.

A.2 Scoring

Intuitively, *higher-dimensional conjunctive queries* exhibit stronger structural constraints, and a successful match in such queries typically implies *higher semantic consistency*. Therefore, we place greater trust in their matching results.

However, in the **scoring phase**, we acknowledge that conjunctive queries of different lengths contribute *unequally* to the overall bias. High-dimensional queries carry *greater uncertainty* in their influence on the final score due to:

- Their relative scarcity in the query set;
- Their tendency to return *fewer results* under the same set of queried keywords.

To address this, we introduce a **dimension-based weighting mechanism** that balances the *trust* in structural matches against the *volume* of actual results. Specifically, the weight for a conjunctive query of dimension n is defined as:

$$\mathrm{Weight}_n = \frac{n^\gamma}{R_n + \epsilon},$$

where:

- $\gamma \geq 0$ is a hyperparameter that controls the preference for structurally constrained queries. When $\gamma = 0$, all query dimensions are treated equally; higher values of γ increase the emphasis on higher-dimensional queries;
- $R_n = \sum_{q_{\mathrm{conj}_i} \in Q_{\mathrm{conj}},\ |q_{\mathrm{conj}_i}|=n} \left|\mathrm{DB}(q_{\mathrm{conj}_i})\right|$ is the total number of results returned by all n-dimensional conjunctive queries;
- ϵ is a small positive constant for smoothing.

The final scoring function is:

$$\mathrm{bias} = w_\mathrm{A}(e_\mathrm{T}^{\mathrm{mapped}}) - w_\mathrm{T}(e_\mathrm{T}),$$
$$\mathrm{Score}(M_\mathrm{HM}) = \sum_{e_\mathrm{T} \in \mathcal{E}_\mathrm{T}} \mathrm{Weight}_{|e_\mathrm{T}|} \cdot (\mathrm{bias})^2,$$

where $\mathrm{Weight}_{|e_\mathrm{T}|}$ is selected from a pre-defined table based on the dimension of each hyperedge.