TWo-IN-one-SSE: Fast, Scalable and Storage-Efficient Searchable Symmetric Encryption for Conjunctive and Disjunctive Boolean Queries

Arnab Bag
IIT Kharagpur, India

Debadrita Talapatra
IIT Kharagpur, India

Ayushi Rastogi
IIT Kharagpur, India

Sikhar Patranabis
IBM Research India*

Debdeep Mukhopadhyay
IIT Kharagpur, India

August 24, 2022

Abstract

Searchable Symmetric Encryption (SSE) supports efficient yet secure query processing over outsourced symmetrically encrypted databases without the need for decryption. A longstanding open question has been the following: can we design a fast, scalable, linear storage and low-leakage SSE scheme that efficiently supports arbitrary Boolean queries over encrypted databases? In this paper, we present the design, analysis and prototype implementation of the first SSE scheme that efficiently supports conjunctive, disjunctive *and* more general Boolean queries (in both the conjunctive and disjunctive normal forms) while scaling smoothly to extremely large encrypted databases, and while incurring *linear* storage overheads and supporting extremely fast query processing in practice. We quantify the leakage of our proposal via a rigorous cryptographic analysis and argue that it achieves security against a well-known class of leakage-abuse and volume analysis attacks. Finally, we demonstrate the storage-efficiency and scalability of our proposed scheme by presenting experimental results of a prototype implementation of our scheme over large real-world databases.

1 Introduction

The advent of cloud computing potentially allows individuals and organizations to outsource storage and processing of large volumes of data to third party servers. However, this leads to concerns surrounding the confidentiality of the data.

Consider, for instance, a client that offloads an encrypted database of (potentially sensitive) emails to an untrusted server. At a later point of time, the client might want to issue a query

*Part of the work was done while the author was affiliated with ETH Zürich and Visa Research.

of the form: *retrieve all emails received from xyz@foobar.org* OR *abc@foobar.org* AND *with "research" in the subject field*. Ideally, the client should be able to perform this task without revealing any sensitive information to the server, such as the sources and contents of the emails, the keywords underlying a given query, the distribution of keywords across emails, etc. Unfortunately, techniques such as Fully Homomorphic Encryption (FHE) [Gen09] and Oblivious RAM (ORAM) [GO96], that potentially support such an "ideal" notion of privacy, are currently unsuitable for wide-scale practical deployment due to the large performance overheads incurred by them.

Searchable Symmetric Encryption. Searchable Symmetric Encryption (SSE) [SWP00, Goh03,CM05,CGKO06,CK10,KPR12,KP13,CJJ$^+$13,CJJ$^+$14,FJK$^+$15,KM17,Bos16,BMO17, EKPE18, LPS$^+$18, CPPJ18, PM21] is the study of provisioning symmetric-key encryption schemes with search capabilities. The goal of SSE is two-fold: (a) to allow a (potentially untrusted) server to execute keyword search queries directly on a collection of a client's encrypted documents in an efficient manner, and (b) to ensure client privacy by minimising the amount of information "leakage" to the server in the process. Some examples of leakage include the database size, *query pattern* (which queries correspond to the same keyword) and the *access pattern* (the set of file identifiers matching a given query).

SSE for Boolean Queries. The example query over an email database that we outlined above is an instance of what we call a *Boolean query*, in the sense that it can be viewed as a Boolean formula involving certain equality predicates over keywords, connected by AND and OR operators. In this paper, we broadly investigate the following question:

Can we design a fast, scalable, storage-efficient and low-leakage SSE scheme for general Boolean queries?

This seemingly natural question has, somewhat surprisingly, also been a longstanding open question. In particular, while significant progress has been made in designing efficient SSE schemes for simpler sub-classes of Boolean queries (such as atomic equality predicates and conjunctions), the handful of existing SSE schemes supporting disjunctive and general Boolean queries incur extremely large storage overheads (quadratic in the size of the database), which makes them impractical for real-world deployment. We briefly summarise the state-of-the-art on SSE below.

1.1 Background and Related Work

Initial constructions of SSE focused on single keyword search and their related extensions. Recent SSE algorithms now support multi-keyword search. We briefly go over the current status of multi-keyword SSE constructions below.

SSE for Conjunctions. In a seminal work [CJJ$^+$13], Cash et al. proposed Oblivious Cross Tags (OXT) - an efficient, highly scalable and low-leakage (not leaking more than benign information) SSE schemes supporting conjunctive keyword queries over encrypted document collections. Since then, a number of SSE schemes supporting conjunctive keyword queries with a variety of leakage vs efficiency trade-offs have been proposed in different settings [CJJ+14, LPS+18, PM21]. Unfortunately, these schemes are neither efficient nor low-leakage when processing disjunctive or general Boolean queries. For example, when processing a disjunctive query over $(\mathsf{w}_1, \ldots, \mathsf{w}_n)$, these solutions leak the set of documents matching each w_i *individually*, which can be devastating in the face of existing leakage-abuse attacks [CGPR15, ZKP16, BKM20, OK21].

SSE for Disjunctions. While SSE for single and conjunctive keyword queries has been studied quite extensively, SSE for disjunctive queries has received much less attention. To the best of our knowledge, only IEX-2Lev and IEX-ZMF due to Kamara and Moataz [KM17] support reasonably efficient disjunctive query processing without incurring potentially devastating leakage.

The IEX family of schemes [KM17, PPSY21a] from has a few disadvantages that we outline here. First, its performance for conjunctive queries over real-world databases is significantly worse as compared to OXT. Secondly, it is incompatible with OXT and its follow-up schemes [CJJ+14, LPS+18, PM21]; so it does not lead to a common solution that supports both conjunctive and disjunctive queries efficiently. Finally, the IEX family of schemes incurs a (worst-case) storage overhead that grows *quadratically* with the number of keywords in the database. This makes it impractical for deployment over real-world databases.

1.2 Our Contributions

In this paper, we present the design, analysis and prototype implementation of the first SSE scheme that efficiently supports conjunctive, disjunctive *and* more general (and complex) Boolean queries (in both the conjunctive and disjunctive normal forms) while scaling smoothly to extremely large encrypted databases, and while incurring *linear* storage overheads and little query processing overheads in practice. Our scheme is named TWo-IN-one-SSE, or TWINSSE in short. We expand on our contributions below.

Supporting Conjunctive "and" Disjunctive Queries. Our core technical contribution is a novel mechanism for designing SSE schemes that support both conjunctive and disjunctive keyword searches in a fully compatible manner. At a high level, we achieve this as follows. Given *any* conjunctive SSE scheme (i.e., any generic SSE scheme that only supports conjunctive queries), we present a generic black-box transformation that yields an SSE scheme supporting conjunctive, disjunctive and general Boolean queries in the conjunctive normal form (CNF) and disjunctive normal form (DNF). Our transformation does not rely on any special properties of the underlying conjunctive SSE scheme. This allows it to be instantiated from *any* existing conjunctive SSE scheme, including OXT [CJJ+13]. To the best of our knowledge, such a generic transformation from a conjunctive SSE scheme to an SSE scheme for general and complex Boolean queries has not been studied before in the SSE literature[1].

[1]We note here that OXT does support Boolean queries beyond simple conjunctions, albeit where the query must be in a restricted *searchable normal form* (SNF) [CJJ+13]; our transformation is significantly more general in the sense that it extends to any CNF or DNF formula over keywords, well beyond the scope of SNF queries.

A Naïve Approach. The naïve approach for supporting disjunctive queries in a generic way using a system that only supports conjunctive queries is to allow for "negative searches", wherein given a keyword, we can efficiently retrieve the set of documents that *do not* contain the keyword. Then, given a disjunctive query of the form $q = (\mathsf{w}_1 \vee \ldots \mathsf{w}_n)$, we can transform q into the following conjunctive query $\overline{q} = (\overline{w}_1 \wedge \ldots \overline{w}_n)$, where for each i, $\overline{w}_i$ denotes the "negated keyword" that (hypothetically) occurs in every document that does not contain w_i; consequently, $\overline{q}$ can be viewed as the negated counterpart to the original query q.

This approach has two major disadvantages. First, it requires us to design data structures that support efficiently retrieving, for each keyword, not only the set of documents it occurs in, but also the set of documents that it does not occur in, while maintaining data and query privacy. This is likely to lead to massive blowup in storage. Secondly, and most crucially, for disjunctive queries involving less frequent keywords (which is what we expect from a very large proportion of the queries), the overall computational and communication complexity suffers a huge blowup, since it is now proportional to the result set for the negated query, which would be almost the entire set of documents. Our aim is to design a generic transformation mechanism that is significantly more efficient. To this end, we introduce and use a novel concept called *meta-keywords*.

Using Meta-Keywords. The technical centrepiece of our generic transformation is the concept of *meta-keywords*, which we introduce in this paper. At a high level, a meta-keyword mkw_i is a disjunction of certain carefully chosen keywords of the form $\mathsf{mkw}_i = (\mathsf{w}_{i_1} \vee \mathsf{w}_{i_2} \vee \ldots \vee \mathsf{w}_{i_\ell})$, that we pre-process and store at setup in an inverted search index[2].

Our core technical observation is the following: given a database with N keywords, there exists an $O(N)$-sized set $\mathcal{S}$ of meta-keywords such that for *any* disjunctive query of the form $q = (\mathsf{w}_1 \vee \ldots \mathsf{w}_n)$ for $n \leq N$, there exists a *meta-query* which is a *conjunction over meta-keywords* of the form $q' = (\mathsf{mkw}_1 \wedge \ldots \wedge \mathsf{mkw}_n)$ such that $\mathsf{mkw}_1, \ldots, \mathsf{mkw}_n \in \mathcal{S}$, and such that $\text{Result-Set}(q) \subseteq \text{Result-Set}(q')$.

The non-triviality of our approach lies in addressing the following challenges simultaneously:

- **Coverage:** Designing an $O(N)$-sized meta-keyword set that "covers" an $O(2^N)$-sized space of all possible disjunctive queries.
- **Efficiency:** Minimizing the overheads due to filtering of "spurious" documents in the result "meta-set" (i.e. ensuring that $\text{Result-Set}(q')$ is as close to $\text{Result-Set}(q)$ as possible).
- **Security:** Minimizing leakage by ensuring that the meta-keywords reveal as little information as possible about the underlying keywords being actually queried.

While achieving these requirements simultaneously appears challenging at first sight, we develop a systematic and formal approach that allows us to achieve them for *any* database; more formally, given a database, we show how to convert the same into a *meta-database* equipped with a linearly-sized set of meta-keywords that meets all of the aforementioned requirements. We formalise these properties in Sections 3 and 4.

[2] An inverted search index is a data structure popularly used by essentially all SSE schemes that is indexed by the keywords, and stores, for each keyword, the set of documents it contains, albeit in encrypted form.

TWINSSE. We use the aforementioned transformation to design our overall solution, that we call TWo-IN-one-SSE, or TWINSSE in short. As the astute reader might have already guessed, given a conjunctive SSE scheme, our design of TWINSSE uses the following two-step approach:

- **Step-1:** Given any database $\mathbf{DB}$, convert it into the corresponding meta-database $\widehat{\mathbf{DB}}$, where $\widehat{\mathbf{DB}}$ can be viewed as a database equipped with two kinds of keywords – the original keywords and the meta-keywords.
- **Step-2:** Apply the conjunctive SSE scheme to encrypt and query the database $\widehat{\mathbf{DB}}$.

Note that conjunctive query processing over $\widehat{\mathbf{DB}}$ proceeds exactly as it would over $\mathbf{DB}$, and requires no additional query planning on the part of the client. Disjunctive query processing is more involved because it requires the client to plan the meta-query. In Section 4, we formally describe how this can be done using $O(n)$ computation (which is the information-theoretic minimum for any n-word disjunctive query). Finally, in Section 5, we describe a hybrid query planning approach that allows handling general Boolean queries in CNF and DNF expressions in an efficient manner. We note here that the ability to efficiently handle CNF and DNF queries effectively allows TWINSSE to handle complex Boolean queries (involving both conjunctive and disjunctive clauses) by casting them into either CNF or DNF formulae over keywords.

We then present a concrete instantiation of TWINSSE from OXT as the baseline conjunctive SSE scheme. We denote this version of OXT as $\text{TWINSSE}_{\text{OXT}}$. Details of $\text{TWINSSE}_{\text{OXT}}$ and, in particular, its handling of disjunctive queries, are presented in Section 4.3. Additionally, we present an elaborate discussion on executing complex Boolean queries using $\text{TWINSSE}_{\text{OXT}}$ in Section 5.

Leakage Analysis. We formally detail the leakage profile of $\text{TWINSSE}_{\text{OXT}}$ in Appendix B. In order to analyse the impact of this leakage on the security of $\text{TWINSSE}_{\text{OXT}}$, we perform a detailed cryptanalysis of $\text{TWINSSE}_{\text{OXT}}$ in Appendix D. In particular, we show that known leakage-abuse attacks [CGPR15, ZKP16], volume analysis-based attacks [BKM20], and the state-of-the-art SAP attack [OK21] fail against $\text{TWINSSE}_{\text{OXT}}$ in practical adversarial settings.

Experimental Evaluation. We present a C++ implementation of $\text{TWINSSE}_{\text{OXT}}$ along with performance figures in Section 6. We experimented over the Enron email corpus [3] for compatibility with previous SSE literature. The data set contains around 170K keywords, 500K documents and 20 million unique keyword-document pairs. Our experiments validate that $\text{TWINSSE}_{\text{OXT}}$ supports extremely fast conjunctive, disjunctive and more complex Boolean queries (in CNF/DNF expression), and substantially outperforms the IEX family of schemes on two counts: (a) storage requirements for the encrypted database, and (b) practical search performance for conjunctive queries, while also achieving comparable practical search performance for disjunctive queries[4].

[3] `https://www.cs.cmu.edu/~enron/`
`https://www.kaggle.com/wcukierski/enron-email-dataset`

[4] We plan to make our prototype implementation open-source when the paper is accepted.

2 Preliminaries and Background

In this section we introduce the notations used in the rest of the paper, as well as preliminary background material on SSE.

2.1 Notations

We write $x \xleftarrow{R} \chi$ to represent that an element x is sampled uniformly at random from a set/distribution $\mathcal{X}$. The output x of a deterministic algorithm $\mathcal{A}$ is denoted by $x = \mathcal{A}$ and the output x' of a randomized algorithm $\mathcal{A}'$ is denoted by $x' \leftarrow \mathcal{A}'$. For $a, b \in \mathbb{Z}$ such that $a, b \geq 0$, we denote by $[a]$ and $[a, b]$ the set of integers lying between 1 and a (both inclusive), and the set of integers lying between a and b (both inclusive), respectively.

Databases. Let $\Delta = \{\mathsf{w}_1, \ldots, \mathsf{w}_N\}$ be a dictionary of keywords, and let $\mathcal{F} = \{f_1, \ldots, f_D\}$ be a collection of documents, such that each document f_i is associated with a unique identifier id_i and contains keywords from Δ. We assume that standard set operations including union and intersection are allowed over Δ. We denote by $\mathbf{DB}$ a database of identifier-keyword pairs, such that $(\mathsf{id}, \mathsf{w}) \in \mathbf{DB}$ if and only if the document with identifier id contains the keyword w. We denote by $\mathbf{DB}(\mathsf{w})$ the set of all identifiers corresponding to documents containing w. We denote by $|\Delta|$ the number of distinct keywords in $\mathbf{DB}$, by $|\mathbf{DB}|$ the number of distinct $\mathsf{id} - \mathsf{w}$ pairs in $\mathbf{DB}$. Finally, we denote by $|\mathbf{DB}(\mathsf{w})|$ the number of documents containing w.

Conjunctive and Disjunctive Queries. We represent a conjunctive query over n distinct keywords $\mathsf{w}_1, \ldots, \mathsf{w}_n$ as $q = (\mathsf{w}_1 \wedge \ldots \wedge \mathsf{w}_n)$ and define the set $\mathbf{DB}(q)$ as $\mathbf{DB}(q) = \cap_{i=1}^{n} \mathbf{DB}(\mathsf{w}_i)$. Similarly, we represent a disjunctive query over n distinct keywords $\mathsf{w}_1, \ldots, \mathsf{w}_n$ as $q = (\mathsf{w}_1 \vee \ldots \vee \mathsf{w}_n)$ and define the set $\mathbf{DB}(q)$ as $\mathbf{DB}(q) = \cup_{i=1}^{n} \mathbf{DB}(\mathsf{w}_i)$.

Throughout the paper, we use the notation $\mathcal{R}_q = \mathbf{DB}(q)$ to represent the result of searching a query q (irrespective of the query type), unless otherwise specified.

2.2 Searchable Symmetric Encryption

Any (static) SSE scheme [CGKO06,CJJ+13] consists of a polynomial-time algorithm Setup executed by the client, and an interactive protocol Search executed jointly by the client and the server:

- Setup$(1^\lambda, \mathbf{DB})$: Takes as input the security parameter λ and a database $\mathbf{DB}$, and outputs the tuple $(\mathsf{sk}, \mathsf{st}, \mathbf{EDB})$, where sk is the client's secret-key, st is the client's internal state, and $\mathbf{EDB}$ is the encrypted database.
- Search$(\mathsf{sk}, \mathsf{st}, q; \mathbf{EDB})$: The client takes as input the secret-key sk, its state st and a query q, while the server takes as input the encrypted database $\mathbf{EDB}$. At the end of

the protocol, the client outputs $\mathbf{DB}(q)$[5].

Correctness. An SSE scheme is said to be correct if for every database **DB** and for every query q, the output of the SEARCH protocol contains $\mathbf{DB}(q)$ with overwhelming probability.

Security. The adaptive security of any SSE scheme is parameterized by a leakage function

$$\mathcal{L} = \left(\mathcal{L}^{\textsc{Setup}}, \mathcal{L}^{\textsc{Search}}\right),$$

where $\mathcal{L}^{\textsc{Setup}}$ encapsulates the leakage to an adversarial server during the setup phase, and $\mathcal{L}^{\textsc{Search}}$ encapsulates the leakage to an adversarial server during each execution of the search protocol.

Algorithm 1 Experiment $\mathbf{Real}^{\mathsf{SSE}}(\lambda, Q)$

function $\mathbf{Real}^{\mathsf{SSE}}(\lambda, Q)$
  $N \leftarrow \mathbf{Adv}(\lambda)$
  $(\mathsf{sk}, \mathsf{st}_0, \mathbf{EDB}_0) \leftarrow \textsc{Setup}(\lambda, N)$
  for $k \leftarrow 1 \text{to} Q$ **do**
    Let $q_k \leftarrow \mathbf{Adv}(\lambda, \mathbf{EDB}_{k-1}, \tau_1, \ldots, \tau_{k-1})$
    Let $(\mathsf{st}_k, \mathbf{EDB}_k, \mathbf{DB}(q_k)) \leftarrow$
      $\textsc{Search}(\mathsf{sk}, \mathsf{st}_{k-1}, q_k; \mathbf{EDB}_{k-1})$
    Let τ_k denote the view of the adversary after
      the k^{th} query
  $b \leftarrow \mathbf{Adv}(\lambda, \mathbf{EDB}_Q, \tau_1, \ldots, \tau_Q)$
  return b

Algorithm 2 Experiment $\mathbf{Ideal}^{\mathsf{SSE}}(\lambda, Q, \mathcal{L})$

function $\mathbf{Ideal}^{\mathsf{SSE}}(\lambda, Q, \mathcal{L})$
  Parse the leakage function $\mathcal{L}$ as:
  $\mathcal{L} = \left(\mathcal{L}^{\textsc{Setup}}, \mathcal{L}^{\textsc{Search}}\right)$.
  $(\mathsf{st}_{\textsc{Sim}}, \mathbf{EDB}_0) \leftarrow \textsc{SimSetup}(\mathcal{L}^{\textsc{Setup}}(\lambda, N))$
  for $k \leftarrow 1$ to Q **do**
    Let $q_k \leftarrow \mathbf{Adv}(\lambda, \mathbf{EDB}_{k-1}, \tau_1, \ldots, \tau_{k-1})$
    Let $(\mathsf{st}_{\textsc{Sim}}, \mathbf{EDB}_k, \tau_k) \leftarrow \textsc{SimSearch}$
      $(\mathsf{st}_{\textsc{Sim}}, \mathcal{L}^{\textsc{Search}}(q_k); \mathbf{EDB}_{k-1})$
    Let τ_k denote the view of the adversary after
      the k^{th} query
  $b \leftarrow \mathbf{Adv}(\lambda, \mathbf{EDB}_Q, \tau_1, \ldots, \tau_Q)$
  return b

Informally, an SSE scheme is adaptively secure with respect to a leakage function $\mathcal{L}$ if the adversarial server provably learns no more information about **DB** other than that

[5] We also make the implicit assumption that upon obtaining the set of document identifiers corresponding to a query, the client performs an additional interaction with the server to actually retrieve the documents with these identifiers.

encapsulated by $\mathcal{L}$. Formally, an SSE scheme is said to be adaptively secure with respect to a leakage function $\mathcal{L}$ if for any stateful PPT adversary **Adv** that issues a maximum of $Q = \mathsf{poly}(\lambda)$ queries, there exists a stateful probabilistic polynomial-time simulator $\textsc{Sim} = (\textsc{SimSetup}, \textsc{SimSearch})$ such that the following holds:

$$\left|\Pr\left[\mathbf{Real}^{\mathsf{SSE}}_{\mathbf{Adv}}(\lambda, Q) = 1\right] - \Pr\left[\mathbf{Ideal}^{\mathsf{SSE}}_{\mathbf{Adv},\textsc{Sim}}(\lambda, Q) = 1\right]\right| \leq \mathsf{negl}(\lambda),$$

where the "real" experiment $\mathbf{Real}^{\mathsf{SSE}}$ and the "ideal" experiment $\mathbf{Ideal}^{\mathsf{SSE}}$ are as described in Algorithm 1 and Algorithm 2.

3 TWINSSE: Simplified Version

In this section, we introduce TWINSSE. For ease of representation, we first present a simplified version, which we refer to as $\text{TWINSSE}_{\textsc{Basic}}$.

3.1 The Core Tool: Meta-Keywords

We begin by describing the core technical tool for our construction, which we refer to as *meta-keywords*. Let $\Delta = \{\mathsf{w}_1, \ldots, \mathsf{w}_N\}$ be a dictionary of keywords (N is total number of keywords in **DB**), and assume (without loss of generality) that these keywords are arranged in increasing order of frequency, i.e.,

$$|\mathbf{DB}(\mathsf{w}_1)| \leq |\mathbf{DB}(\mathsf{w}_2)| \leq \cdots \leq |\mathbf{DB}(\mathsf{w}_N)|.$$

Super-Keyword. We begin by defining a super-keyword, which is simply a *disjunction* of some subset of the keywords in Δ. Formally, a super-keyword $\overline{\mathsf{w}}$ is represented by a bit-string of the form $\overline{\mathsf{w}} = (b_1, b_2, \ldots, b_N) \in \{0,1\}^N$, such that

$$\mathbf{DB}(\overline{\mathsf{w}}) = \bigcup_{\ell \in [N] \text{ s.t. } b_\ell = 1} \mathbf{DB}(\mathsf{w}_\ell).$$

Note that one could equivalently represent $\overline{\mathsf{w}}$ using the actual constituent keywords; we use the bit-string representation because it makes the description of our strategy easier to follow, and also more efficiently implementable.

Meta-Keyword. We now define a meta-keyword. At a high level, a meta-keyword is a "special" super-keyword with a *single contiguous stretch of 0-entries* in its Boolean representation. Formally, a meta-keyword is defined as follows.

Definition 3.1 (Meta-Keyword)**.** *A meta-keyword* $\mathsf{mkw}_{i,j}$ *is a super-keyword indexed by* $(i,j) \in [N] \times [N]$ *such that* $i \leq j$*, represented as bit-string* $\mathsf{mkw}_{i,j} = (b_1, b_2, \ldots, b_N) \in \{0,1\}^N$*, where for each* $\ell \in [N]$*, we have*

$$b_\ell = \begin{cases} 0 & \text{if } \ell \in [i,j], \\ 1 & \text{otherwise.} \end{cases}$$

In other words, for the meta-keyword $\mathsf{mkw}_{i,j}$, we have

$$\mathbf{DB}(\mathsf{mkw}_{i,j}) = \bigcup_{\ell \in [N] \setminus [i,j]} \mathbf{DB}(\mathsf{w}_\ell).$$

Informally, one can view a meta-keyword $\mathsf{mkw}_{i,j}$ as a disjunction over $\Delta = \{\mathsf{w}_1, \ldots, \mathsf{w}_N\}$ *excluding* a contiguous sequence of keywords $(\mathsf{w}_i, \mathsf{w}_{i+1}, \ldots, \mathsf{w}_j)$.

We also let $\mathsf{mkw}^* = 1^N$ denote the special "all-ones" meta-keyword. Finally, let $\mathcal{S}_{\mathsf{mkw},\Delta} = \{\mathsf{mkw}_{i,j}\}_{i \leq j} \cup \{\mathsf{mkw}^*\}$ be the set of all meta-keywords over the dictionary Δ. It is easy to see that for $|\Delta| = N$, we have $|\mathcal{S}_{\mathsf{mkw},\Delta}| = O(N^2)$.

Using a Meta-Keyword. A reader might wonder why we choose the above definition of a meta-keyword. To begin with, note that while pre-processing and storing an inverted search index consisting of all super-keywords along with the original keywords allows us to trivially answer conjunctive, disjunctive, and more general Boolean queries in a fully compatible manner. However, this would require exponential storage, and therefore is not practically feasible.

Hence, our approach is to look for a poly-sized subset of the set of all possible super-keywords that, if pre-processed and stored as part of the inverted search index, would allow us to "cover" any disjunctive query. It turns that the set of meta-keywords is indeed this set. We make this explicit by stating the following (informal) claim. We subsequently make this claim more formal and prove it.

Claim 3.1 (Informal)**.** *For any disjunctive query of the form* $q = (\mathsf{w}_{\ell_1} \vee \ldots \mathsf{w}_{\ell_n})$ *(where* $n \leq N$*), there exists a* meta-query *which is a* conjunction over meta-keywords *of the form*

$$q_{\mathsf{mkw}} = (\mathsf{mkw}_{i_1,j_1} \wedge \ldots \wedge \mathsf{mkw}_{i_{\overline{n}},j_{\overline{n}}}),$$

such that $\mathsf{mkw}_{i_1,j_1}, \ldots, \mathsf{mkw}_{i_{\overline{n}},j_{\overline{n}}} \in \mathcal{S}_{\mathsf{mkw},\Delta}$*, and such that*

$$\mathbf{DB}(q) = \bigcup_{k \in [n]} \mathbf{DB}(\mathsf{w}_{\ell_k}) \subseteq \bigcap_{k \in [\overline{n}]} \mathbf{DB}(\mathsf{mkw}_{i_k,j_k}) = \mathbf{DB}(q_{\mathsf{mkw}}).$$

Using Claim 3.1 (Overview). As an astute reader might have already observed, this claim allows us to convert *any* disjunctive query over the original set of keywords into a conjunctive query over the set of meta-keywords. Consequently, given a database **DB** over a dictionary Δ, suppose we pre-process **DB** at setup to construct the set of meta-keywords $\mathcal{S}_{\mathsf{mkw},\Delta}$, and build an augmented *meta-database* $\widehat{\mathbf{DB}}$ over the *meta-dictionary* $\widehat{\Delta} = (\Delta \cup \mathcal{S}_{\mathsf{mkw},\Delta})$ (consisting of *both* the original keywords *and* the meta-keywords). We can then use this augmented (plaintext) database together with *any* conjunctive SSE scheme in a black-box manner to build an SSE scheme that supports both conjunctive *and* disjunctive queries. The only price we pay is the $O(N^2)$ storage overhead; we subsequently show how to reduce this to $O(N)$.

3.2 Meta-Keywords as "Covering" Set

We now formalize and prove Claim 3.1. Before detailing the formal proof, we illustrate why this claim is true via a simple toy-example.

Toy-Example. Consider a database $\mathbf{DB}$ with 10 documents (indexed as $\{\mathsf{id}_1, \ldots, \mathsf{id}_{10}\}$) over a four-keyword dictionary

$$\Delta = \{\mathsf{w}_1, \mathsf{w}_2, \mathsf{w}_3, \mathsf{w}_4\},$$

such that

$$\begin{aligned}
\mathbf{DB}(\mathsf{w}_1) &= \{\mathsf{id}_5, \mathsf{id}_{10}\}, \\
\mathbf{DB}(\mathsf{w}_2) &= \{\mathsf{id}_2, \mathsf{id}_5, \mathsf{id}_8, \mathsf{id}_9\}, \\
\mathbf{DB}(\mathsf{w}_3) &= \{\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_4, \mathsf{id}_7, \mathsf{id}_9\}, \\
\mathbf{DB}(\mathsf{w}_4) &= \{\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3, \mathsf{id}_4, \mathsf{id}_5, \mathsf{id}_6, \mathsf{id}_8, \mathsf{id}_{10}\}.
\end{aligned}$$

Now consider the following example disjunctive queries q and the corresponding meta-queries q_{mkw}.

Example-1. Let $q = (\mathsf{w}_1 \vee \mathsf{w}_2 \vee \mathsf{w}_3)$, and $q_{\mathsf{mkw}} = \mathsf{mkw}_{4,4}$, where $\mathsf{mkw}_{4,4} = (1, 1, 1, 0)$.

$$\begin{aligned}
\mathbf{DB}(\mathsf{mkw}_{4,4}) &= \bigcup_{\ell \neq 4} \mathbf{DB}(\mathsf{w}_\ell) \\
&= \{\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_4, \mathsf{id}_5, \mathsf{id}_7, \ldots, \mathsf{id}_{10}\}, \\
\mathbf{DB}(q_{\mathsf{mkw}}) &= \mathbf{DB}(\mathsf{mkw}_{4,4}).
\end{aligned}$$

Now, we have

$$\mathbf{DB}(q) = \{\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_4, \mathsf{id}_5, \mathsf{id}_7, \ldots, \mathsf{id}_{10}\} = \mathbf{DB}(q_{\mathsf{mkw}}).$$

Example-2. Alternatively, suppose that $q = (\mathsf{w}_2 \vee \mathsf{w}_3)$ and $q_{\mathsf{mkw}} = (\mathsf{mkw}_{1,1} \wedge \mathsf{mkw}_{4,4})$, where $\mathsf{mkw}_{1,1} = (0, 1, 1, 1)$ and $\mathsf{mkw}_{4,4} = (1, 1, 1, 0)$.

$$\begin{aligned}
\mathbf{DB}(\mathsf{mkw}_{1,1}) &= \bigcup_{\ell \neq 1} \mathbf{DB}(\mathsf{w}_\ell) = \{\mathsf{id}_1, \ldots, \mathsf{id}_{10}\}, \\
\mathbf{DB}(q_{\mathsf{mkw}}) &= \mathbf{DB}(\mathsf{mkw}_{1,1}) \cap \mathbf{DB}(\mathsf{mkw}_{4,4}) \\
&= \{\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_4, \mathsf{id}_5, \mathsf{id}_7, \ldots, \mathsf{id}_{10}\}.
\end{aligned}$$

Now, we have

$$\mathbf{DB}(q) = \{\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_4, \mathsf{id}_5, \mathsf{id}_7, \mathsf{id}_8, \mathsf{id}_9\} \subseteq \mathbf{DB}(q_{\mathsf{mkw}}).$$

Formal Statement. We now state the following formal version of Claim 3.1.

Lemma 3.1. *Let* $q = (\mathsf{w}_{\ell_1} \vee \ldots \mathsf{w}_{\ell_n})$ *for some* $n \leq N$*, where* $\ell_1 \leq \ldots \leq \ell_n$*, and let*

$$q_{\mathsf{mkw}} = (\mathsf{mkw}_{i_0,j_0} \wedge \mathsf{mkw}_{i_1,j_1} \wedge \ldots \wedge \mathsf{mkw}_{i_n,j_n}),$$

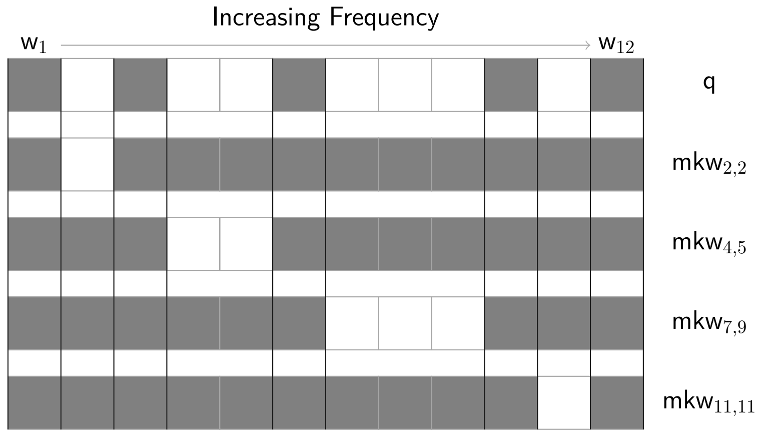

Figure 1: Expressing disjunctive query q in terms of mkw-s with a single stretch of 0s. In this example, $\Delta = \{\mathsf{w}_1, \ldots, \mathsf{w}_{12}\}$, and $q = \mathsf{w}_1 \vee \mathsf{w}_3 \vee \mathsf{w}_6 \vee \mathsf{w}_{10} \vee \mathsf{w}_{12}$ (where $\ell_1 = 1$, $\ell_2 = 3$, $\ell_3 = 6$, $\ell_4 = 10$, $\ell_5 = 12$ and $n = 5$). Note that, each mkw has ws present at the same places where a w is present in q. The stretches of 0s (absence of ws) ensure that when the mkws are ANDed together (searched in a conjunctive manner), only ws in the original query q remain. (Gray and white cells represent 1 and 0, respectively.)

where for each $k \in [0, n-1]$*, we have*

$$\mathsf{mkw}_{i_k,j_k} = \begin{cases} \mathsf{mkw}_{\ell_k+1,\ell_{k+1}-1} & \text{if } \ell_{k+1} > \ell_k + 1, \\ \phi & \text{otherwise,} \end{cases}$$

where we define $\ell_0 := 0$ *and* ϕ *denotes an "empty" meta-keyword, and*

$$\mathsf{mkw}_{i_n,j_n} = \begin{cases} \mathsf{mkw}_{\ell_n+1,n} & \text{if } \ell_n < n, \\ \mathsf{mkw}^* & \text{otherwise.} \end{cases}$$

Then we have

$$\mathbf{DB}(q) = \bigcup_{k \in [n]} \mathbf{DB}(\mathsf{w}_{\ell_k}) \subseteq \mathbf{DB}(q_{\mathsf{mkw}}).$$

Observe that for the specific examples stated above, the conjunctive meta-query q_{mkw} exactly follows the generic conjunctive meta-query laid out in the above lemma. Here, ℓ_k denotes the index of k'th query keyword in Δ, whereas (i_k, j_k) denote the start and end indices of the absent keywords stretch in each mkw. Following the above mkw formulation, we see that mkw^* occurs only if all keywords in Δ are present in the query q - when the value of n (number of keywords in q) is same as the number of keywords in Δ. In this case, the index of the last query keyword in Δ (or ℓ_k) is equal to the number of query keywords n, and only mkw^* is selected. This is a rather unusual case that rarely occurs in real applications.

Note that, at a high level, to prove this lemma it suffices that to prove that for each keyword w_{ℓ_k} in the queried disjunction q, we have $\mathbf{DB}(\mathsf{w}_{\ell_k}) \subseteq \mathbf{DB}(q)$. In more detail, we would like to prove that for each keyword w_{ℓ_k} in the queried disjunction q, we have

$\mathbf{DB}(\mathsf{w}_{\ell_k}) \subseteq \mathbf{DB}(\mathsf{mkw}_{i_{\overline{k}},j_{\overline{k}}})$ for each $\overline{k} \in [0,n]$. The proof follows from the fact that for each $k \in [n]$, the following must be true:

- The index corresponding to the keyword w_{ℓ_k} has a 1-entry in every non-empty meta-keyword in the set $\{\mathsf{mkw}_{i_{\overline{k}},j_{\overline{k}}}\}_{\overline{k}<k}$. This is because the "stretch" of 0-entries in each such meta-keyword ends *before* the index ℓ_k.
- The index corresponding to the keyword w_{ℓ_k} has a 1-entry in every non-empty meta-keyword in the set $\{\mathsf{mkw}_{i_{\overline{k}},j_{\overline{k}}}\}_{\overline{k}\geq k}$. This is because the "stretch" of 0-entries in each such meta-keyword starts *after* the index ℓ_k.
- Finally, the index corresponding to the keyword w_{ℓ_k} has, by default, a 1-entry in mkw^* – the "all-ones" meta-keyword.

Combining these observations, we get that for each keyword w_{ℓ_k} in the queried disjunction q, we have $\mathbf{DB}(\mathsf{w}_{\ell_k}) \subseteq \mathbf{DB}(\mathsf{mkw}_{i_{\overline{k}},j_{\overline{k}}})$ for each $\overline{k} \in [0,n]$, as desired. Figure 1 captures the aforementioned intuition pictorially. We state the formal proof next.

Proof of Lemma 3.1. We show in the formal proof of Lemma 3.1 that each mkw constructed following the description of Lemma 3.1 covers each w in $\mathbf{q}$ ($\mathbf{DB}(q)$ part). Other ws that are not in q are filtered out (due to the intersection in Lemma 3.1).

We start with the following conjunctive meta-keyword expression of q_{mkw} for a particular query q as given in the Lemma 3.1.

$$\mathbf{DB}(q_{\mathsf{mkw}}) = \mathbf{DB}(\mathsf{mkw}_{i_0,j_0} \wedge \ldots \wedge \mathsf{mkw}_{i_n,j_n}) = \bigcap_{k=0}^{n} \mathbf{DB}(\mathsf{mkw}_{i_k,j_k}) \tag{1}$$

By the definition of meta-keywords (Definition 3.1), the following relation holds.

$$\mathbf{DB}(\mathsf{mkw}_{i_k,j_k}) = \bigcup_{l\in[N]\setminus[i_k,j_k]} \mathbf{DB}(\mathsf{w}_l)$$

We rewrite Equation (1) in the following way.

$$\begin{aligned}
\mathbf{DB}(q_{\mathsf{mkw}}) &= \bigcap_{k=0}^{n} \Big(\bigcup_{l\in[N]\setminus[i_k,j_k]} \mathbf{DB}(\mathsf{w}_l) \Big) \qquad (2)\\
&= \bigcup_{r\in[n]} \mathbf{DB}(\mathsf{w}_{\ell_r}) \cup \bigcap_{k=0}^{n} \Big(\bigcup_{\substack{l\in[N]\\ \setminus(\{[i_k,j_k]\}\\ \cup\{\ell_r: r\in[n]\})}} \mathbf{DB}(\mathsf{w}_l) \Big)\\
&= \mathbf{DB}(q) \cup \Big(\bigcap_{k=0}^{n} \Big(\bigcup_{\substack{l\in[N]\\ \setminus(\{[i_k,j_k]\}\\ \cup\{\ell_r: r\in[n]\})}} \mathbf{DB}(\mathsf{w}_l) \Big) \Big)
\end{aligned}$$

Observe that, the union inside the right hand expression in the above expression keeps all ws except a stretch of ws (from index i_k to j_k) for each value of k, inside the outer intersection of $n+1$ terms. Since the intersection of these unions reduces to a small but finite set of id-s, the following relation holds,

$$\mathbf{DB}(q) \subseteq \mathbf{DB}(q_{\mathsf{mkw}})$$

which is exactly what Lemma 3.1 states. □

3.3 $\text{TWINSSE}_{\text{Basic}}$

We now put everything together in our basic scheme $\text{TWINSSE}_{\text{Basic}}$. Let CSSE = (CSSE.Setup, CSSE.Search) be *any* generic conjunctive SSE scheme. Given CSSE, we construct

$$\text{TWINSSE}_{\text{Basic}} = \begin{cases} \text{TWINSSE}_{\text{Basic}}.\text{Setup} \\ \text{TWINSSE}_{\text{Basic}}.\text{Search} \end{cases}$$

as described subsequently. Our description here is slightly informal due to space constraint, but captures the overall idea of our approach. More details are available with the final construction in Section 4. We also present brief details of processing purely conjunctive or purely disjunctive; we defer the discussion on our treatment of general Boolean formulae to Section 5.

$\underline{\text{TWINSSE}_{\text{Basic}}.\text{Setup}(1^\lambda, \mathbf{DB})}$: Given a database $\mathbf{DB}$ over a dictionary Δ, construct the set of meta-keywords $\mathcal{S}_{\mathsf{mkw},\Delta}$ as described above. Let $\widehat{\mathbf{DB}}$ denote the meta-database over $\widehat{\Delta} = (\Delta \cup \mathcal{S}_{\mathsf{mkw},\Delta})$ (consisting of *both* the original keywords *and* the meta-keywords). Output

$$(\mathsf{sk}, \mathsf{st}, \widehat{\mathbf{EDB}}) \leftarrow \text{CSSE.Setup}(1^\lambda, \widehat{\mathbf{DB}}).$$

$\underline{\text{TWINSSE}_{\text{Basic}}.\text{Search}(\mathsf{sk}, \mathsf{st}, q; \widehat{\mathbf{EDB}})}$: Given a query q, proceed as follows:

- If q is a purely conjunctive query, output

$$\mathbf{DB}(q) = \text{CSSE.Search}(\mathsf{sk}, \mathsf{st}, q; \widehat{\mathbf{EDB}}).$$

- If q is a purely disjunctive query, construct the conjunctive meta-query q_{mkw} as described in Lemma 3.1, which allows the client to recover

$$\mathbf{DB}(q_{\mathsf{mkw}}) = \text{CSSE.Search}(sk, \mathsf{st}, q; \widehat{\mathbf{EDB}}),$$

  and locally filter $\mathbf{DB}(q) \subseteq \mathbf{DB}(q_{\mathsf{mkw}})$.

Correctness. Correctness is immediate from Lemma 3.1 (see Section 3.2) and correctness of the CSSE scheme. However, this includes the trivial case of returning entire database upon searching a disjunctive query. To avoid such trivial inclusions, we bound the returned result set size close to the actual result set via a *precision* parameter. We define precision η by the following ratio.

$$\eta = \frac{|\mathbf{DB}(q)|}{|\mathbf{DB}(q_{\mathsf{mkw}})|}$$

At a high level, this precision parameter η is a measure of the fraction of spurious ids present in the obtained result set compared to the actual result set. Thus, the correctness can now be defined by the following statement.

For a functionally correct and exact[6] *conjunctive SSE scheme* CSSE*, a plaintext database* $\mathbf{DB}$*, and a disjunctive query* q *with the corresponding transformed meta-query* q_{mkw}*,* TWINSSE *is functionally correct if the following expressions hold.*

$$\mathsf{sk}, \mathsf{st}; \widehat{\mathbf{EDB}} \leftarrow \text{TWINSSE}_{\text{BASIC}}.\text{SETUP}(1^{\lambda}, \mathbf{DB})$$
$$\bar{\mathcal{R}}_q = \text{TWINSSE}_{\text{BASIC}}.\text{SEARCH}(\mathsf{sk}, \mathsf{st}, q; \widehat{\mathbf{EDB}})$$

where $\mathcal{R}_q \subseteq \bar{\mathcal{R}}_q$ *and* $|\bar{\mathcal{R}}_q| \leq \frac{1}{\eta} \cdot |\mathcal{R}_q|$ *(*$0.85 < \eta \leq 1$*) given that* $\bar{\mathcal{R}}_q$ *is returned by* $\text{TWINSSE}_{\text{BASIC}}$ *(or* $\mathbf{DB}(q_{\mathsf{mkw}})$*) and* $\mathcal{R}_q = \mathbf{DB}(q)$*.*

Note that, the lower bound of η is obtained empirically from experiments over real databases. We present such experimental details in Section 6. The lower bound can be adjusted to accommodate larger $\bar{\mathcal{R}}_q$ for different databases if required.

Storage Overhead. $\text{TWINSSE}_{\text{BASIC}}$ incurs $O(N^2)$ storage overhead to store the meta-keywords, where N is the number of keywords in the original plaintext database. This follows immediately from the fact that the number of meta-keywords is $O(N^2)$. This is undesirable in practice as it affects the scalability of the construction for large real databases. Currently, the schemes designed for disjunctive queries (such as IEX) require quadratic storage often leading to storage blow-up for large databases. Our final construction in Section 4 reduces $O(N^2)$ storage overhead to $O(N)$ - a necessary and significant reduction to use large real databases for deployment.

Search Overhead. The disjunctive search uses a meta-keyword as the least-frequent term for searching with the CSSE search routine. Since each mkw is an "union" of constituent ws, on average the frequency of the least-frequent mkw is smaller compared to a conjunctive query constituting the same ws. As a result, this basic method potentially can result to worst-case linear search overhead, which would be highly undesirable. However, we avoid such overheads by choosing an underlying CSSE scheme with *sub-linear search complexity*.

4 TWINSSE: Final Version

In this section, we present our final scheme – TWINSSE, which improves upon $\text{TWINSSE}_{\text{BASIC}}$ with respect to storage requirements as well as search overheads. At the core of both these

[6] An exact solution returns only the documents belonging to the actual query result.

improvements lies an additional technique that we describe next – "frequency-based bucketization" of keywords. We note that similar techniques have been used in the SSE literature [GPPW20], albeit almost entirely for *frequency padding* and leakage-reduction. To the best of our knowledge, we are the first to show that bucketization can also be used to reduce storage and search overheads in SSE schemes.

4.1 Keyword Bucketization at Setup

We now describe our strategy for frequency-based keyword bucketization and intra-bucket meta-keyword generation at setup. We then use this updated meta-keyword generation strategy to formally describe the new setup algorithm – TWINSSE.SETUP.

Bucketization. Let $\Delta = \{\mathsf{w}_1, \ldots, \mathsf{w}_N\}$ be a dictionary of keywords, and assume that these keywords are arranged in increasing order of frequency. Also, let $n' = O(1)$ be any arbitrarily chosen *constant*. We partition the keyword space into $n_{\mathsf{B}} = N/n'$ "buckets" of size n' each ($n_{\mathsf{B}} = \lceil \frac{N}{n} \rceil$, if N is not a multiple of n), where the k^{th} bucket is defined formally as the keyword subset

$$\Delta_k = \{\mathsf{w}_{(k-1)n'+1}, \ldots, \mathsf{w}_{kn'}\}.$$

Note that since all keywords are arranged in increasing order of frequency, each bucket from Δ_1 through $\Delta_{n_{\mathsf{B}}}$ progressively consists of keywords with increasing frequency ranges. We note that this is similar to the bucketization strategy employed in [GPPW20].

Intra-Bucket Meta-Keyword Generation. Having partitioned the keyword space into frequency-based buckets, we now proceed as follows:

- For each bucket Δ_k, we generate an *intra-bucket* meta-keyword set $\mathcal{S}_{\mathsf{mkw},k}$ of size $O(|\Delta_k|^2) = O\left((n')^2\right)$. This is done exactly as in TWINSSE$_{\text{BASIC}}$, i.e., following the meta-keyword generation strategy in Lemma 3.1.
- We then define the overall set of meta-keywords as the collection of intra-bucket meta-keywords from all buckets, i.e.,

$$\mathcal{S}_{\mathsf{mkw},\Delta} := \bigcup_{k \in [n_{\mathsf{B}}]} \mathcal{S}_{\mathsf{mkw},k}.$$

Observe that

$$|\mathcal{S}_{\mathsf{mkw},\Delta|} = \sum_{k \in [n_{\mathsf{B}}]} |\mathcal{S}_{\mathsf{mkw},k}| = O\left(n_{\mathsf{B}}(n')^2\right) = O(Nn').$$

However, $n' = O(1)$ is a constant, and hence, unlike in the basic solution described in Section 3, now $|\mathcal{S}_{\mathsf{mkw},\Delta}| = O(N)$. In other words, we now have a linear-sized meta-keyword set, which forms the key stepping stone towards avoiding a quadratic storage overhead. We design our proposed TWINSSE to work for any choice of n' (ideally, n' should be a small constant to avoid high storage overheads); we use $n' = 10$ for our prototype implementation and experimentation over real-world databases in Section 6. We present brief discussion and empirical evaluations on the Enron dataset in Section 6 to select a suitable value for n'.

TWINSSE.Setup**:** We now put these ideas together to formally describe TWINSSE.Setup in Algorithm 3, which in turn again uses *any* generic conjunctive SSE scheme

$$\text{CSSE} = (\text{CSSE.Setup}, \text{CSSE.Search})$$

in a black-box way. The key changes from the basic scheme in Section 3 are highlighted in red for ease of exposition (in fact, $\text{TWINSSE}_{\text{Basic}}$.Setup can be viewed as a special case of TWINSSE.Setup where all keywords are placed in the same bucket, i.e., $n_{\mathsf{B}} = 1$ and $n' = N$).

Note that Algorithm 3 uses as a sub-routine Algorithm 4, which formally describes the meta-database generation based on the keyword bucketization and intra-bucket meta-keyword generation procedures described earlier. Overall, the working of Algorithm 3 can be divided into two steps: (a) generate the meta-database with the intra-bucket meta-keywords using Algorithm 4, and (b) generate the client state and the encrypted meta-database using CSSE.Setup in a black-box way (note that this second step is the same as in $\text{TWINSSE}_{\text{Basic}}$; the only alteration is in the generation of the meta-database, which now uses linearly many meta-keywords).

4.2 Updated Query Planning

We now describe the updated query planning strategy that takes into account the above mentioned meta-keyword generation process. We use this updated query planning strategy to build the TWINSSE.Search routine (the query planning for conjunctive queries remains same as in $\text{TWINSSE}_{\text{Basic}}$).

At a high level, we partition a disjunctive query into "regions", where each region consists of the keywords in the query that belong to the *same bucket*. Formally, given a query $q = (\mathsf{w}_{\ell_1} \vee \ldots \vee \mathsf{w}_{\ell_n})$, let $\mathcal{Q} = (\mathsf{w}_{\ell_1}, \ldots, \mathsf{w}_{\ell_n})$ and, for each $k \in [n_{\mathsf{B}}]$, let $\mathcal{Q}_k = \Delta_k \cap \mathcal{Q}$. In other words, $\mathcal{Q}_k$ consists of all the keywords in the disjunction q that belong to the k^{th} bucket. It is easy to see that we can re-write q as a *disjunction over sub-queries* as follows:

$$q = \bigvee_{k \in [n_{\mathsf{B}}]} \left(\bigvee_{\mathsf{w} \in \mathcal{Q}_k} \mathsf{w} \right) := \bigvee_{k \in [n_{\mathsf{B}}]} q_k.$$

Note that for some k, $\mathcal{Q}_k$ could be an empty set; in this case, the sub-query q_k is also empty. Based on the above representation, the query planning strategy in TWINSSE works as follows:

- **Step-1:** Partition q into sub-queries $\{q_k\}_{k \in [n_B]}$ as described above.
- **Step-2:** For each sub-query q_k, construct a conjunctive (sub-) meta-query $q_{\mathsf{mkw},k}$ as described in Section 3, using the intra-bucket meta-keywords corresponding to the k^{th} bucket, i.e., the intra-bucket meta-keywords in $\mathcal{S}_{\mathsf{mkw},k}$.

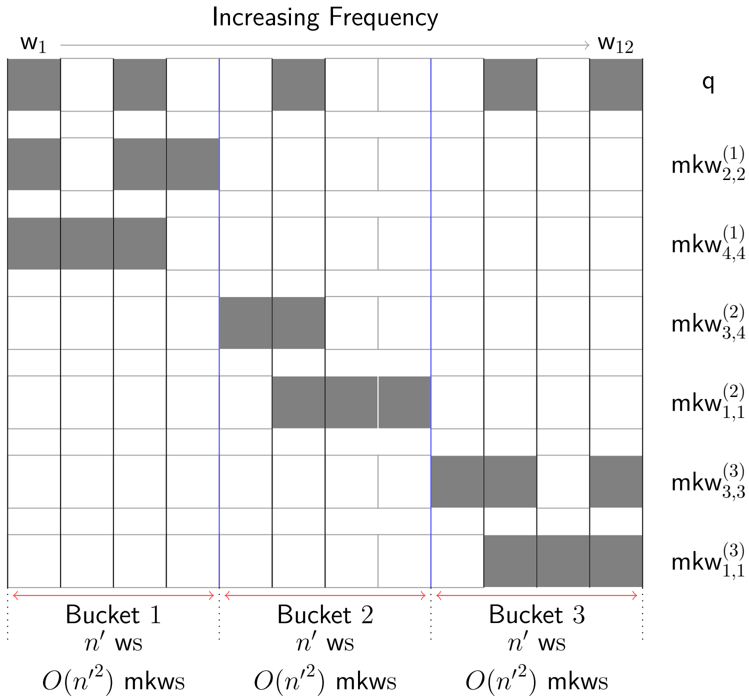

Figure 2: Expressing disjunctive query q in terms of mkws in the improved $O(N)$ construction. We use the same database parameters from Figure 1. The *sub-mkws* (smaller mkws in each bucket) are padded to realise a full mkw and stored in $\widehat{\mathbf{DB}}$. Here, the bucket size n' is four. *The k in $\mathsf{mkw}_{i,j}^{(k)}$ represents the bucket index the $\mathsf{mkw}_{i,j}$ belongs to.* Note that, `AND`ing mkws within a bucket retains only the ws in the q (covered by that bucket). Note that expressing each meta-keyword as a bit-string allows efficient an transformation from the original disjunctive query into the corresponding conjunction of meta-keywords through simple bit-wise set/clear operations. This transformation incurs only negligible additional computational overhead in practice.

Algorithm 3 TWINSSE.Setup

Input: $\mathbf{DB}, 1^\lambda, n', n_\mathsf{B}$
Output: $\mathsf{sk}, \mathsf{st}, \widehat{\mathbf{EDB}}$

function TWINSSE.Setup($1^\lambda, n', \mathbf{DB}$)
    $\widehat{\mathbf{DB}} = \textsc{GenMetaDB}(\mathbf{DB}, n', n_\mathsf{B})$
    $\mathsf{sk}, \mathsf{st}, \widehat{\mathbf{EDB}} \leftarrow \text{CSSE.Setup}(\lambda, \widehat{\mathbf{DB}})$
    return $\widehat{\mathbf{EDB}}, \mathsf{sk}, \mathsf{st}$
Server receives $\widehat{\mathbf{EDB}}$
Client keeps $(\mathsf{sk}, \mathsf{st}, n_\mathsf{B})$

- **Step-3:** Finally, we define the overall meta-query q_mkw as $q_\mathsf{mkw} = \bigvee_{k \in [n_\mathsf{B}]} q_{\mathsf{mkw},k}$, and

Algorithm 4 GenMetaDB

Input: $\mathbf{DB}, n', n_{\mathsf{B}}$

Output: $\widehat{\mathbf{DB}}$

function GenMetaDB($\mathbf{DB}, n', n_{\mathsf{B}}$)
  Extract Δ from $\mathbf{DB}$ and sort ws in Δ in increasing order of frequency
  Partition Δ as $\{\Delta_1, \cdots, \Delta_{n_{\mathsf{B}}}\}$ such that $\Delta_i = \{\mathsf{w}_{(i-1)n'+1}, \cdots, \mathsf{w}_{in'}\}$ ▷ *The last bin may not contain n' ws. Keep only as many are left.*
  Initialise a bucket index $k \leftarrow 1$
  for $\Delta_k \in \{\Delta_1, \cdots, \Delta_{n_{\mathsf{B}}}\}$ **do**
    Set $\ell \leftarrow |\Delta_k|$
    Parse Δ_k as $\{w_1^k, \cdots, w_\ell^k\}$
    for $i \leftarrow 1$ to ℓ **do**
      for $j \leftarrow 0$ to $\ell - i$ **do**
        $\mathsf{mkw}_{i,i+j}^{(k)} \leftarrow \Delta_k \setminus \{w_i^k, \ldots, w_{i+j}^k\}$
        for $\mathsf{w} \in \mathsf{mkw}_{i,i+j}^{(k)}$ **do**
          $\widehat{\mathbf{DB}}(\mathsf{mkw}_{i,i+j}^{(k)}) \leftarrow \widehat{\mathbf{DB}}(\mathsf{mkw}_{i,i+j}^{(k)}) \cup \mathbf{DB}(\mathsf{w})$
    $k \leftarrow k + 1$
  return $\widehat{\mathbf{DB}}$

re-construct $\mathbf{DB}(q_{\mathsf{mkw}})$ as

$$\mathbf{DB}(q_{\mathsf{mkw}}) = \bigcup_{k \in [n_{\mathsf{B}}]} \mathbf{DB}(q_{\mathsf{mkw},k}),$$

where the recovery of each $\mathbf{DB}(q_{\mathsf{mkw},k})$ happens via an independent (and parallel) execution of the same search protocol as in TWINSSE$_{\text{Basic}}$.

The above query planning strategy is summarized pictorially in Figure 2 (note that the superscript k in $\mathsf{mkw}_{i,j}^{(k)}$ represents the bucket index for the meta-keyword $\mathsf{mkw}_{i,j}$). In comparison with the example figure (Figure 1) in Section 3, we note that the meta-keywords are now chosen from a smaller set of size $\approx (4 \times 12) = 48$, as compared to a set of size $\approx 12^2 = 144$ in Figure 1.

TWINSSE.Search**:** We now put these ideas together to formally describe TWINSSE.Search in Algorithm 5, which in turn uses Algorithm 6 as a sub-routine (we only summarize the processing of disjunctive queries since conjunctive queries are processed as in TWINSSE$_{\text{Basic}}$). The key changes from the basic scheme in Section 3 are highlighted in red for ease of exposition (again, TWINSSE$_{\text{Basic}}$.Search can be viewed as a special case of TWINSSE.Search where all of the keywords are placed in the same bucket, i.e., $n_{\mathsf{B}} = 1$ and $n' = N$).

Algorithm 6 formally captures the updated disjunctive query planning strategy based on query partitioning and intra-bucket meta-keywords, as described earlier. Note that in Algorithm 5, each conjunctive sub-meta-query q_k is executed in parallel using the search algorithm CSSE.Search of the underlying conjunctive SSE scheme CSSE, and the final result-set corresponding to the overall meta-query is constructed locally at the client by taking the union over the result-sets corresponding to each conjunctive sub-meta-query.

Algorithm 5 TWINSSE.SEARCH (for disjunctive queries)

Input: $q, \mathsf{sk}, \mathsf{st}, \widehat{\mathbf{EDB}}, n', n_{\mathsf{B}}$
Output: Result set $\mathbf{DB}(q)$

```
function TWINSSE.SEARCH(q, sk, st, EDB^.)
    Client
    Generate q_mkw = (∨_{k∈[n_B]} q_mkw,k) = GENMQUERY(q, n', n_B, Δ).
    For each non-empty q_k (in uniformly random order), the client and server engage in
    the search protocol as below.
    Client+Server
    for each non-empty q_mkw,k (in random order) do
        DB(q_mkw,k) ←
        CSSE.SEARCH(sk, st, q_mkw,k; EDB^).
        At the end of the protocol, client receives DB(q_mkw,k).
    Client
    Initialize DB(q_mkw) ← EMPTY-SET.
    for each DB(q_mkw,k) from search protocol do
        DB(q_mkw) ← DB(q_mkw) ∪ DB(q_mkw,k).
The client locally filters DB(q) ⊆ DB(q_mkw).
```

Correctness. We state the following theorem for the correctness of TWINSSE.

Theorem 4.1 (Correctness of TWINSSE)**.** *Assuming that* CSSE *satisfies correctness of search for conjunctive queries and Lemma 3.1 holds,* TWINSSE *satisfies correctness for both conjunctive and disjunctive queries.*

The proof essentially follows from the same arguments as the proof of correctness for $\text{TWINSSE}_{\text{BASIC}}$ in Section 3 and is detailed in Appendix A.

4.3 Instantiation from the OXT Protocol and Complexity Analysis

In its most general form, our proposed TWINSSE scheme can be concretely instantiated using *any* conjunctive SSE scheme. In this section, we analyze a concrete instance of TWINSSE based on the OXT protocol [CJJ+13], which we call $\text{TWINSSE}_{\text{OXT}}$. We analyze $\text{TWINSSE}_{\text{OXT}}$ asymptotically in terms of storage requirements and search overheads. Our analysis does not require understanding the internal details of OXT beyond what is already stated in this section; the reader may refer to [CJJ+13] for more details. Finally, we refer the reader to Section 6 for experimental validation of the analysis presented here over the Enron email corpus.

Storage Requirements (Server). The (worst-case) server-side storage requirement for $\text{TWINSSE}_{\text{OXT}}$ is $O(n'|\mathbf{DB}|)$, where $|\mathbf{DB}|$ is the number of distinct identifier-keyword pairs in $\mathbf{DB}$, and $n' = O(1)$ denotes the (constant) size of each keyword bucket used. This linearization process through keyword bucketization process incurs an $O(n')$-fold increase in storage overhead over OXT (where $n' = O(1)$ is a constant). We view this as a necessary

Algorithm 6 GenMQuery

Input: $q = \{\mathsf{w}_{i_1} \vee \mathsf{w}_{i_2} \vee \ldots \vee \mathsf{w}_{i_l}\}, n', n_{\mathsf{B}}, \Delta$
Output: Meta-Query q_{mkw}

function GenMQuery($q, n', n_{\mathsf{B}}, \Delta$)
  Parse Δ as $\{\Delta_1, \ldots, \Delta_{n_{\mathsf{B}}}\}$
  Sort ws in q in increasing order of frequency
  Partition query q into set of sub-queries P_q as $q_1||\cdots||q_{n_{\mathsf{B}}}$, such that q_k contains ws only from Δ_k for $k = 1, \cdots, n_{\mathsf{B}}$
  for $q_k \in \mathsf{P}_q$ **do**
    Parse q_k as $\{w^k_{i_1}, \cdots, w^k_{i_{l'}}\}$
    for $j \leftarrow 1$ to l' **do**
      $\mathsf{mkw}^{(k)}_{i_{j-1}+1, i_j-1} \leftarrow \Delta_k \setminus \{w^k_{i_{j-1}+1}, \ldots, w^k_{i_j-1}\}$
      ▷ *Recall that* $\mathsf{mkw}_{i_k, j_k} \leftarrow \phi$ *if* $i_k > j_k$, $i_0 = 0$ *and* w^k_1 *is the first keyword in* Δ_k
    $\mathsf{mkw}^{(k)}_{i_{l'}+1, n'} \leftarrow \Delta_k \setminus \{w^k_{i_{l'}+1}, \ldots, w^k_{n'}\}$
    $q_{\mathsf{mkw},k} \leftarrow \mathsf{mkw}^{(k)}_{1, i_1-1} \wedge \ldots \wedge \mathsf{mkw}^{(k)}_{i_{l'}+1, n'}$
    $q_{\mathsf{mkw}} \leftarrow q_{\mathsf{mkw}} \vee q_{\mathsf{mkw},k}$
  return q_{mkw}

trade-off for the additional ability to support disjunctive queries efficiently yet securely. In comparison, the IEX family of schemes incur (worst-case) quadratic storage overheads, more precisely, $O(|\Delta||\mathbf{DB}|)$, where $|\Delta|$ denotes the number of keywords in $\mathbf{DB}$.

However, n' (or the number of buckets) needs to be chosen carefully to bound the storage overhead to linear (which also keeps the leakage from multiple buckets at minimum). A high value of n' would incur a larger storage overhead with lesser leakage from small number of buckets (as outlined in Section 4.1). Whereas a small value of n' would result in a higher number of buckets leading to lesser storage but increased leakage from more number of buckets. We selected n' in the range 10-15 based on empirical evaluations over real data sets that allows to retain a linear storage overhead. These experimental results are provided in Section 6.

Storage Requirements (Client). The client-side storage requirement for $\text{TWINSSE}_{\text{OXT}}$ is $O(n'|\Delta| \log |\mathbf{DB}|)$. This again represents an $O(n')$-fold increase in storage overhead over the original OXT scheme (where $n' = O(1)$ is a constant), which has a client-side storage requirement of $O(|\Delta| \log |\mathbf{DB}|)$.

We also note here that $\text{TWINSSE}_{\text{OXT}}$ requires $O(1)$ storage for the secret key(s) at the client-end (this is a purely client-side overhead, not associated with the server-side storage). In contrast, the IEX family of schemes require $O(|\Delta|)$ secure storage for secret keys (one key per keyword due to individual multi-map structure required for each keyword index), which is likely to be costly for extremely large databases. We emphasise that this requirement is only for *secure* storage to store secret keys on the client-side. The client-side storage overhead mentioned in the previous paragraph accounts for storing only auxiliary information required during query processing, and this *does not* require secure storage.

Search Complexity. We now present an asymptotic analysis of the search complexity (more concretely, the computational and communication requirements during search query processing) of $\mathrm{TWINSSE}_{\mathrm{OXT}}$. We divide our analysis into two parts – conjunctive queries and disjunctive queries:

Conjunctive queries. Let $q = (\mathsf{w}_1 \wedge \ldots \wedge \mathsf{w}_n)$ be a conjunctive query, where w_1 is the *least frequent* keyword. When processing q using $\mathrm{TWINSSE}_{\mathrm{OXT}}$, the computational costs (at both the client and the server) as well as the communication requirements between the client and the server scale linearly as $O(n|\mathbf{DB}(\mathsf{w}_1)|)$. This is *exactly the same* as in OXT, and is hence worst case sub-linear.

Disjunctive queries. Let $q = (\mathsf{w}_1 \vee \ldots \vee \mathsf{w}_n)$ be a disjunctive query. Also, let $q_{\mathsf{mkw}} = \vee_{k\in[n_{\mathsf{B}}]} q_{\mathsf{mkw},k}$ be the corresponding meta-query, and assume without loss of generality that $\mathsf{mkw}^{(k)}_{i_k,j_k}$ is the least frequent meta-keyword within $q_{\mathsf{mkw},k}$ for each $k \in [n_{\mathsf{B}}]$ (such that $q_{\mathsf{mkw},k}$ is non-empty). When processing q using $\mathrm{TWINSSE}_{\mathrm{OXT}}$, the computational costs (at both the client and the server) as well as the communication requirements between the client and the server scale linearly as $O(\gamma)$, where

$$\gamma = \sum_{k\in[n_{\mathsf{B}}]} |q_k||\mathbf{DB}(\mathsf{mkw}^{(k)}_{i_k,j_k})|,$$

where $|q_k|$ denotes the number of meta-keywords in the conjunctive sub-meta-query q_k ($|q_k| = 0$ when q_k is empty). Note that this is essentially a generalization of the analysis of search query overheads for $\mathrm{TWINSSE}_{\mathrm{BASIC}}$ in Section 3, where all keywords belong to the same bucket (i.e., $n_{\mathsf{B}} = 1$). We provide a comparative summary of storage and search overhead for TWINSSE and IEX in Table 1 for quick reference.

Scheme	**Storage Overhead**	**Search Time**[7]	
		Conjunctive	Disjunctive
$\mathrm{TWINSSE}_{\mathrm{OXT}}$	$O(n'\|\mathbf{DB}\|)$	$O(n\|\mathbf{DB}(w_1)\|)$	$\sum_{k\in[n_{\mathsf{B}}]} \|q_k\|\|\mathbf{DB}(\mathsf{mkw}^{(k)}_{i_k,j_k})\|$
IEX-2LEV	$O(\|\Delta\|\|\mathbf{DB}\|)$	$O(n^2(\|\mathbf{DB}(w^u_M)\| + t\|\mathbf{DB}(\delta_u)\|))$	$O(n^2\|\mathbf{DB}(w_M)\|)$

Table 1: Comparative summary of storage overhead and search complexity of TWINSSE and IEX.

Spurious document identifiers. It turns out that keyword bucketization also significantly reduces the search overheads (both computational and communication) due to spurious document identifiers in $\mathbf{DB}(q_{\mathsf{mkw}})$. In particular, recall our observation with respect to $\mathrm{TWINSSE}_{\mathrm{BASIC}}$ from Section 3: the fraction of spurious identifiers retrieved is directly proportional to the *average number of common documents* over every keyword-pair in the database. However, in our improved solution, the database is partitioned into buckets, and all keywords within the same bucket have essentially similar frequency ranges. This means, in particular, that an overwhelmingly large fraction of buckets either contain all low-frequency keywords (in which case, the spurious document-set is essentially null, since such keywords almost never co-occur across documents [CJJ+13]), or very high-frequency keywords (in which case, such keywords occur in almost all documents, and the proportion of spurious documents is low by default).

We generalise the aforementioned observations into the following (informal) claim about the search complexity incurred by $\text{TWINSSE}_{\text{OXT}}$, which is essentially an extension of our claim for $\text{TWINSSE}_{\text{BASIC}}$.

Claim 4.1 (Informal)**.** $\text{TWINSSE}_{\text{OXT}}$ *incurs (average-case) sub-linear search complexity (in terms of both computational costs and communication overheads) for both conjunctive and disjunctive queries.*

We validate this claim with experimental results over the Enron email corpus in Section 6. We also extend the analysis and experimental evaluations to more general/complicated Boolean queries (CNF or DNF formulae) in Section 5. Our experiments show that searches in $\text{TWINSSE}_{\text{OXT}}$ incur at most 15% overhead due to spurious identifiers in the result set. In order to filter out the set of spurious documents from the final result set, we can resort to the same strategies as used by state-of-the-art volume hiding SSE constructions (e.g. SSE schemes obtained naturally from the encrypted multi-map constructions proposed in [KM19, PPYY19], where the client obtains a mixture of real and "fake" identifiers at the end of the query phase to hide the true query response volume from the server).

Note that, TWINSSE uses separate mkw^* for each bucket where mkw_i^* denotes the mkw^* for the i-th bucket. Following the definition of mkw^*, mkw_i^* represents the disjunction of all ws in the i-th bucket. Using a separate mkw^* per bucket allows TWINSSE to support non-SNF queries more efficiently than OXT. OXT uses a single mkw^* for any non-SNF query into an SNF query. In this process, OXT incurs a worst-case linear search overhead. In contrast, TWINSSE uses mkw_i^* if and only if a query involves *all* keywords from the i-th bucket (in this case, it is the only optimal choice). These specific queries can be considered as corner cases which rarely occur in realistic searches. Consequently, TWINSSE incurs spurious ids, which is typically only $15-20\%$ on average for realistic non-SNF queries. Additionally, this process in $\text{TWINSSE}_{\text{OXT}}$ also allows for parallel execution of independent sub-queries over different buckets. Adopting a similar approach with OXT for parallel execution would incur more leakage from each queried bucket due to the exact result set which reveals the volume pattern. Since $\text{TWINSSE}_{\text{OXT}}$ produces noisy result set due to the spurious ids, the volume pattern leakage is less compared to OXT.

4.4 Security of TWINSSE

We present an informal discussion on the security of our construction here. Detailed formal security discussion and leakage analysis (including experimental evaluations) are available in Appendices B, C and D. Security of TWINSSE is modelled in the semi-honest adversarial setting where the server is assumed to be a honest-but-curious entity (that means, the server follows the algorithmic specifications exactly, but can record information for later analysis).

Informally, TWINSSE inherits security properties and leakage profile from the underlying CSSE construction. We assume that the underlying CSSE construction is an adaptively secure sublinear conjunctive SSE algorithm which is secure against a semi-honest adversary $\mathcal{A}$ and the leakage of CSSE is characterised by the leakage function $\mathcal{L}_{\text{CSSE}}$. The leakage function $\mathcal{L}_{\text{CSSE}}$ is an ensemble of the leakage functions for Setup and Search individually, expressed in the following way.

$$\mathcal{L}_{\text{CSSE}} = \{\mathcal{L}_{\text{CSSE}}^{\textsc{Setup}}, \mathcal{L}_{\text{CSSE}}^{\textsc{Search}}\}$$

Given the above CSSE leakage functions, security of TWINSSE can be analysed using TWINSSE leakage function $\mathcal{L}_{\text{TWINSSE}}$ in the same adaptive semi-honest adversarial model. Similar to $\mathcal{L}_{\text{CSSE}}$, $\mathcal{L}_{\text{TWINSSE}}$ is composed of two separate leakage functions for Setup and Search, as expressed below, that capture the leakage from TWINSSE execution in the meta-keyword setting.

$$\mathcal{L}_{\text{TWINSSE}} = \{\mathcal{L}_{\text{TWINSSE}}^{\textsc{Setup}}, \mathcal{L}_{\text{TWINSSE}}^{\textsc{Search}}\}$$

Concretely, $\mathcal{L}_{\text{TWINSSE}}$ is identical to the $\mathcal{L}_{\text{CSSE}}$ with n_{B} (the number of buckets) as an additional benign component. In other words, we show that $\mathcal{L}_{\text{TWINSSE}}$ is equal to $\bar{\mathcal{L}}_{\text{CSSE}}$ where $\bar{\mathcal{L}}_{\text{CSSE}}$ is $\mathcal{L}_{\text{CSSE}}$ in the context of meta-keywords and n_{B}. At a high level, $\mathcal{L}_{\text{TWINSSE}}^{\textsc{Setup}}$ incorporates $\widehat{\mathbf{DB}}$ instead of $\mathbf{DB}$ generated by the GenMetaDB during setup. Similarly, the search leakage encapsulates leakages from *both* conjunctive and disjunctive queries. We quantify these separately through two individual leakage function instances - one for conjunctive queries, and one for disjunctive queries from meta-keywords, where meta-keywords are generated using GenMQuery routine. We show that the leakage for the conjunctive case is exactly the same as that of the CSSE construction, and for disjunctive queries it incurs a similar leakage profile, but from meta-keywords. We provide a detailed formal analysis of $\mathcal{L}_{\text{TWINSSE}}$ in Appendix B.

Security of $\text{TWINSSE}_{\text{OXT}}$**.** The security analysis of $\text{TWINSSE}_{\text{OXT}}$ follows from the security notions of generic TWINSSE, as informally discussed above (detailed formally in Appendix B and Appendix C). We also present a leakage-based cryptanalysis of the $\text{TWINSSE}_{\text{OXT}}$ scheme via experiments over the Enron email corpus in Appendix D.

Comparison with IEX**.** At a high level, $\text{TWINSSE}_{\text{OXT}}$ avoids two kinds of leakages that the IEX family of schemes incurs for any query. To begin with, IEX leaks to the server the exact size of the result set pertaining to a query (also referred to as the size pattern leakage). As already mentioned, due to the presence of spurious document identifiers in the result set, $\text{TWINSSE}_{\text{OXT}}$ inherently hides the size pattern from the server. More crucially, IEX incurs significant sub-query leakage. For example, given a disjunctive query of the form $q = (\mathsf{w}_1 \vee \mathsf{w}_2)$, where w_1 is the more frequent keyword, it leaks to the server: (a) the frequency of the more frequent keyword, i.e., $|\mathbf{DB}(\mathsf{w}_1)|$, and (b) the number of documents that contain w_2 but not w_1, i.e., $|\mathbf{DB}(\mathsf{w}_2) \setminus \mathbf{DB}(\mathsf{w}_1)|$. Whereas, $\text{TWINSSE}_{\text{OXT}}$ only leaks the frequency of the least frequent meta-keyword (in this example, the meta-keyword corresponding to w_1), and no information about the other meta-keywords in the conjunction (in this example, no information about w_2). In other words, $\text{TWINSSE}_{\text{OXT}}$ incurs less leakage as compared to the IEX family of schemes during search queries.

5 General Boolean Queries (CNF and DNF) in TWINSSE and $\text{TWINSSE}_{\text{OXT}}$

In Section 4, we described how TWINSSE and its instantiation from OXT, namely $\text{TWINSSE}_{\text{OXT}}$, handle purely conjunctive and purely disjunctive queries. In this section, we describe how TWINSSE can be extended to address general Boolean queries in either the conjunctive normal form (CNF) or the disjunctive normal form (DNF).

We note here that OXT does support Boolean queries beyond simple conjunctions, albeit where the query must be in a restricted *searchable normal form* (SNF) [CJJ+13]; our transformation is significantly more general in the sense that it extends to any CNF or DNF formula over keywords, well beyond the scope of SNF queries.

We begin by describing how to handle DNF queries, because, similar to its purely conjunctive and disjunctive counterparts, DNF queries are also handled by TWINSSE (and hence, by extension, $\text{TWINSSE}_{\text{OXT}}$) by making fully black-box usage of the underlying conjunctive SSE scheme. Subsequently, we show how to address CNF queries. This is slightly more involved, and makes non black-box usage of the underlying conjunctive SSE scheme (we describe a specific strategy for $\text{TWINSSE}_{\text{OXT}}$ to handle CNF queries that relies on a special data structure used by the OXT scheme).

5.1 Handling Boolean Queries in DNF Form

In Boolean logic, a disjunctive normal form (DNF) is a canonical normal form of a logical formula consisting of a disjunction of conjunctions (alternatively, OR of AND clauses). Formally, any query q that is a Boolean formula over keywords in DNF takes the form

$$q = \bigvee_{\ell \in [L]} q_\ell = \bigvee_{\ell \in [L]} (\mathsf{w}_{\ell,1} \wedge \ldots \wedge \mathsf{w}_{\ell,t_\ell}),$$

where each $q_\ell = (\mathsf{w}_{\ell,1} \wedge \ldots \wedge \mathsf{w}_{\ell,t_\ell})$ for $\ell \in [L]$ is a conjunctive *clause*. Our approach to handle a DNF query is straightforward, and closely resembles, at a high level, our strategy for handling disjunctive queries via query partitioning in TWINSSE. Let CSSE = (CSSE.Setup, CSSE.Search) be any generic conjunctive SSE scheme. The search algorithm processes q via the following steps (the setup algorithm remains the same as TWINSSE.Setup described in Algorithm 3, Section 4):

- **Client:** Parse a DNF query as $q = \bigvee_{\ell \in [L]} q_\ell$.
- **Client + Server:** For each $\ell \in [L]$ (either in parallel or in uniformly random order), compute $\mathbf{DB}(q_\ell) = \text{CSSE.Search}(q_\ell, \widehat{\mathbf{EDB}})$,
  where $\widehat{\mathbf{EDB}}$ is the encrypted meta-database output by TWINSSE.Setup.
- **Client:** Locally compute at the client

$$\mathbf{DB}(q) = \bigcup_{\ell \in [L]} \mathbf{DB}(q_\ell).$$

Correctness. Correctness of search follows immediately from the correctness guarantees of the underlying conjunctive SSE scheme CSSE.

Search Complexity. We present an (asymptotic) analysis of the complexity of handling DNF search queries (more concretely, the computational and communication requirements during DNF query processing) when we instantiate TWINSSE using the OXT protocol from [CJJ+13], i.e., in $\text{TWINSSE}_{\text{OXT}}$. Let q be a DNF query of the form

$$q = \bigvee_{\ell \in [L]} q_\ell = \bigvee_{\ell \in [L]} (\mathsf{w}_{\ell,1} \wedge \ldots \wedge \mathsf{w}_{t_\ell,\ell}),$$

where we assume, without loss of generality, that for each $\ell \in [L]$, $\mathsf{w}_{\ell,1}$ is the least frequent conjunct in the conjunctive clause q_ℓ. When processing q using $\text{TWINSSE}_{\text{OXT}}$, the computational costs (at both the client and the server) as well as the communication requirements between the client and the server scale linearly as $O(\gamma_{\text{DNF}})$, where

$$\gamma_{\text{DNF}} = \sum_{\ell \in [L]} t_\ell |\mathbf{DB}(\mathsf{mkw}_{\ell,1})|.$$

Note that this is very similar in flavor to the analysis of disjunctive search query overheads for $\text{TWINSSE}_{\text{OXT}}$ in Section 4.

Leakage Analysis. We state the following theorems for the leakage from TWINSSE and $\text{TWINSSE}_{\text{OXT}}$ when processing Boolean queries in DNF form.

Theorem 5.1 (DNF Query Processing in TWINSSE)**.** *Assuming that* CSSE *is an (adaptively) secure SSE scheme with respect to the leakage function* $\mathcal{L}_{\text{CSSE}} = (\mathcal{L}^{\text{Setup}}_{\text{CSSE}}, \mathcal{L}^{\text{Search}}_{\text{CSSE}})$, *the leakage incurred by* TWINSSE *when processing a DNF query as described above is* $\mathcal{L}^{\text{Search,DNF}}_{\text{TWINSSE}})$, *where for any DNF query* $q = \bigvee_{\ell \in [L]} q_\ell$, *we have*

$$\mathcal{L}^{\text{Search,DNF}}_{\text{TWINSSE}}(q) = \left\{\mathcal{L}^{\text{Search}}_{\text{CSSE}}(q_\ell)\right\}_{\ell \in [L]}.$$

Theorem 5.2 (DNF Query Processing in $\text{TWINSSE}_{\text{OXT}}$)**.** *The leakage incurred by* $\text{TWINSSE}_{\text{OXT}}$ *when processing a DNF query as described above is* $\mathcal{L}^{\text{Search,DNF}}\text{TWINSSE}_{\text{OXT}}$, *where for any sequence of DNF queries* $\mathcal{Q} = (q_1, \ldots, q_M)$ *such that* $q_m = \bigvee_{\ell \in [L_m]} q_{m,\ell}$ *for each* $m \in [M]$, *we have*

$$\mathcal{L}^{\text{Search,DNF}}_{\text{TWINSSE}_{\text{OXT}}}(\mathcal{Q}) = [\mathsf{RP}, \mathsf{SP}, \mathsf{EP}, \mathsf{IP}](\{\{q_{m,\ell}\}_{\ell \in [L_m]}\}_{m \in [M]}).$$

where RP, SP, EP *and* IP *leakages for conjunctive queries are as defined in Appendix B.*

The proofs of these theorems are very similar to the proofs of Theorems B.1 and B.2 described earlier in Appendix C, and are hence not detailed separately.

5.2 Handling Boolean Queries in CNF Form

In Boolean logic, a conjunctive normal form (CNF) is a canonical normal form of a logical formula consisting of a conjunction of disjunctions (alternatively, AND of OR clauses).

Formally, any query q that is a CNF Boolean formula over keywords takes the form

$$q = \bigwedge_{\ell \in [L]} q_\ell = \bigwedge_{\ell \in [L]} (\mathsf{w}_{\ell,1} \vee \ldots \vee \mathsf{w}_{\ell,t_\ell}),$$

where each $q_\ell = (\mathsf{w}_{\ell,1} \vee \ldots \vee \mathsf{w}_{\ell,t_\ell})$ for $\ell \in [L]$ is a disjunctive *clause*. Our approach to handle a CNF query is slightly more involved, and makes usage of some specific features of the OXT protocol to ensure sub-linear search overheads in practice. Hence, the subsequent description of how to handle CNF queries is specific to $\text{TWINSSE}_{\text{OXT}}$. We leave it as an interesting open question to investigate a generic solution using any conjunctive SSE scheme in a black-box manner.

We now describe our proposed strategy for handling CNF queries in $\text{TWINSSE}_{\text{OXT}}$. Before delving into the details, we need to recall some details of the original OXT scheme due to Cash *et al.* [CJJ+13]. We refer the reader to [CJJ+13] for details of the OXT scheme; however, we will try to make the description here as self-contained as possible. The OXT protocol maintains on the server (as part of the encrypted database **EDB**) a special data structure called a "cross-tag set" (or XSet in short). The XSet consists of several "cross-tags", where each cross-tag $\mathsf{xtag}_{\mathsf{id},\mathsf{w}}$ corresponds to a document identity-keyword pair $(\mathsf{id}, \mathsf{w})$, where

$$\mathsf{xtag}_{\mathsf{id},\mathsf{w}} \in \mathsf{XSet} \text{ if and only if } \mathsf{w} \in \mathbf{DB}(\mathsf{id}).$$

In our handling of CNF queries in $\text{TWINSSE}_{\text{OXT}}$, we make black-box usage the following sub-functions provided by any implementation of the OXT protocol:

- OXT.GenXTag$(\mathsf{sk}, \mathsf{id}, \mathsf{w})$: The client can use the secret key generated at setup by OXT.Search to generate $\mathsf{xtag}_{\mathsf{id},\mathsf{w}}$ for any document identifier id and keyword w.
- OXT.SearchXTag$(\mathsf{xtag}_{\mathsf{id},\mathsf{w}}; \mathsf{XSet})$: On receipt of a cross-tag $\mathsf{xtag}_{\mathsf{id},\mathsf{w}}$ from the client, the server can look up the XSet efficiently to return a bit $\beta \in \{0, 1\}$, where $\beta = 1$ if $xtag_{\mathsf{id},\mathsf{w}} \in \mathsf{XSet}$, and $\beta = 0$ otherwise.

Given these sub-routines, our proposal for processing a CNF query q proceeds via the steps outlined below (the setup algorithm again remains the same as TWINSSE.Setup described in Algorithm 3, Section 4, albeit for CSSE = OXT). Note that unlike purely conjunctive/disjunctive queries and DNF queries, all of which required a single round search protocol, our processing of CNF queries now requires two rounds of communication between the client and the server.

- **Client:** Parse a CNF query as

$$q = \bigwedge_{\ell \in [L]} q_\ell = \bigwedge_{\ell \in [L]} (\mathsf{w}_{\ell,1} \vee \ldots \vee \mathsf{w}_{\ell,t_\ell}),$$

- **Client:** Identify the candidate disjunctive clause q_ℓ with the smallest result set (this can be computed in a straightforward manner from the client state st output by OXT.Setup, which has the frequency of each keyword in the dictionary).

- **Client+Server (Round-1):** Compute the result-set corresponding to the disjunctive clause q_ℓ as
$$\mathbf{DB}(q_\ell) = \text{TWINSSE}_{\text{OXT}}.\textsc{Search}(q_\ell, \widehat{\mathbf{EDB}}),$$
where $\widehat{\mathbf{EDB}}$ is the encrypted meta-database output by $\text{TWINSSE}_{\text{OXT}}.\textsc{Setup}$, by directly using the disjunctive search protocol described in Algorithm 5 (Section 4) with CSSE = OXT.

- **Client:** For each $\mathsf{id} \in \mathbf{DB}(q_\ell)$ and each $\mathsf{w}_{i,\ell'}$ for $\ell' \neq \ell$ in the query q, compute
$$\mathsf{xtag}_{\mathsf{id},\mathsf{w}_{i,\ell'}} = \text{OXT}.\textsc{GenXTag}(\mathsf{sk}, \mathsf{id}, \mathsf{w}_{i,\ell'}).$$

- **Client+Server (Round-2):** For each $\mathsf{id} \in \mathbf{DB}(q_\ell)$ and each $\mathsf{w}_{i,\ell'}$ (either in parallel or in uniformly random order), the client sends $\mathsf{xtag}_{\mathsf{id},\mathsf{w}_{i,\ell'}}$ to the server and receives in response
$$\beta_{\mathsf{id},\mathsf{w}_{i,\ell'}} = \text{OXT}.\textsc{SearchXTag}(\mathsf{xtag}_{\mathsf{id},\mathsf{w}_{i,\ell'}}; \mathsf{XSet}).$$

- **Client:** For each $\mathsf{id} \in \mathbf{DB}(q_\ell)$, compute
$$\beta_{\mathsf{id}} = \bigwedge_{\ell' \in [L] \setminus \{\ell\}} \left(\beta_{\mathsf{id},\mathsf{w}_{\ell,1}} \vee \ldots \vee \beta_{\mathsf{id},\mathsf{w}_{\ell,t_\ell}}\right).$$
Output the final result set
$$\mathcal{R}_q = \{\mathsf{id} \in \mathbf{DB}(q_\ell) \text{ such that } \beta_{\mathsf{id}} = 1\}.$$

Correctness. Correctness of search follows immediately from the correctness guarantees of $\text{TWINSSE}_{\text{OXT}}$ (Theorem 4.1), and the correctness guarantees of the OXT protocol itself.

Search Complexity. We now present an (asymptotic) analysis of the complexity of handling CNF search queries (more concretely, the computational and communication requirements during CNF query processing) in $\text{TWINSSE}_{\text{OXT}}$. Let q be a CNF query of the form
$$q = \bigwedge_{\ell \in [L]} q_\ell = \bigwedge_{\ell \in [L]} (\mathsf{w}_{\ell,1} \vee \ldots \vee \mathsf{w}_{\ell,t_\ell}),$$
where we assume, without loss of generality, that $q_1 = (\mathsf{w}_{1,2} \vee \ldots \vee \mathsf{w}_{\ell,2})$ is the disjunctive clause with the smallest result set. Let $q_{\mathsf{mkw}} = \vee_{k \in [n_{\mathsf{B}}]} q_{\mathsf{mkw},k}$ be the corresponding meta-query when the disjunctive search query corresponding to q_1 is processed using $\text{TWINSSE}_{\text{OXT}}.\textsc{Search}$, and assume without loss of generality that $\mathsf{mkw}^{(k)}_{i_k,j_k}$ is the least frequent meta-keyword within $q_{\mathsf{mkw},k}$ for each $k \in [n_{\mathsf{B}}]$ (such that $q_{\mathsf{mkw},k}$ is non-empty).

When processing q using $\text{TWINSSE}_{\text{OXT}}$, the computational costs (at both the client and the server) as well as the communication requirements between the client and the server scale linearly as $O(\gamma_0 + \gamma_1)$,
$$\gamma_0 = \sum_{k \in [n_{\mathsf{B}}]} |q_k| |\mathbf{DB}(\mathsf{mkw}^{(k)}_{i_k,j_k})|,$$

where $|q_k|$ denotes the number of meta-keywords in the conjunctive sub-meta-query q_k ($|q_k| = 0$ when q_k is empty), and

$$\gamma_1 = |\mathbf{DB}(q_1)| \cdot (\ell \in [2, L] t_\ell) \,.$$

Note that the term γ_0 is computed exactly as in the analysis of disjunctive search query overheads for $\text{TWINSSE}_{\text{OXT}}$ in Section 4. Moreover, the term γ_1, which represents computational and communication complexities incurred as a result of the round-2 of the CNF query processing (using OXT.GenXTag and OXT.SearchXTag), is independent of the frequencies of any of the disjunctive clauses other than the "least frequent clause" q_1.

Leakage Analysis. We state the following theorems for the leakage from $\text{TWINSSE}_{\text{OXT}}$ when processing Boolean queries in CNF form.

Theorem 5.3 (CNF Query Processing in $\text{TWINSSE}_{\text{OXT}}$)**.** *The leakage incurred by* $\text{TWINSSE}_{\text{OXT}}$ *when processing a CNF query as described above is* $\mathcal{L}^{\text{Search,CNF}}_{\text{TWINSSE}_{\text{OXT}}}$*, where for any sequence of CNF queries* $\mathcal{Q} = (q_1, \ldots, q_M)$ *such that*

$$q_m = \bigwedge_{\ell \in [L_m]} q_{m,\ell} = \bigwedge_{\ell \in [L]} (\mathsf{w}_{m,\ell,1} \vee \ldots \vee \mathsf{w}_{m,\ell,t_{m,\ell}})$$

for each $m \in [M]$*, with* $q_{m,1}$ *being (without loss of generality) the least frequent disjunctive clause for each* $m \in [M]$*, and for any pair of bucketization parameters* (n', n_{B})*, we have*

$$\mathcal{L}^{\text{Search,DNF}}_{\text{TWINSSE}_{\text{OXT}}}(\mathcal{Q}) = \left([\mathsf{RP}, \mathsf{SP}, \mathsf{EP}, \mathsf{IP}](\mathcal{Q}_{\mathsf{mkw}}), \mathcal{L}^*_{\mathsf{xtag}}\right) .$$

where RP*,* SP*,* EP *and* IP *leakages for conjunctive queries are as defined in Appendix B, and where* $\mathcal{Q}_{\mathsf{mkw},1}$ *is a sequence of (sub-)meta-queries of the form*

$$\mathcal{Q}_{\mathsf{mkw}} = \{q_{\mathsf{mkw},k,1}\}_{k \in [n_{\mathsf{B}}]},$$

where for each $\ell \in [M]$*, we have*

$$q_{\mathsf{mkw},\ell} = \left(\bigvee_{k \in [n_{\mathsf{B}}]} q_{\mathsf{mkw},k,1}\right) = \text{GenMQuery}(q_{\ell,1}, n', n_{\mathsf{B}}),$$

and, finally, we have

$$\mathcal{L}^*_{\mathsf{xtag}} = \{|\mathbf{DB}(q_{m,1}) \cap \mathbf{DB}(\mathsf{w}_{m,\ell,\ell'})|\}_{m \in [M], \ell \in [L_m], \ell' \in [t_{m,\ell}]}.$$

The proof of this theorem is again very similar to the proof of Theorem B.2, and is hence not detailed separately.

6 Experimental Results

In this section, we describe a prototype implementation of $\text{TWINSSE}_{\text{OXT}}$ and evaluate its performance over real-world databases. We present experimental results comparing the storage requirements and search performance of $\text{TWINSSE}_{\text{OXT}}$ with that of IEX-2Lev [KM17].

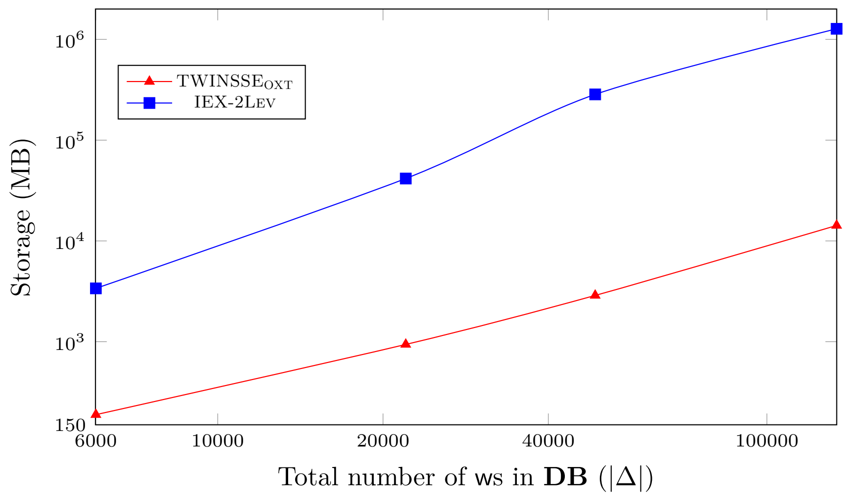

Figure 3: Server storage overhead with database size (|**DB**|) for the Enron database.

6.1 Experimental Results over the Enron Email Data Set

Data Set and Platform. We used the Enron email data set[8] for our experiments. The Enron email data set contained 517,401 documents (emails) and 20 million keyword-document pairs, with a total size 1.9 GB. The complete TWINSSE$_{\mathsf{OXT}}$ implementation was done using C++ (with GCC9 compiler) with native multi-threading support, and we used Redis as the database backend. We ran the experiments on a *single* node with Intel Xeon E5-2690 v4 2.6 GHz CPU with 128 GB RAM and 512 GB SSD storage.

Implementation Details. We created the meta-keyword database (or the transformed database) **MDB** from the parsed Enron database **DB**. The plain Enron database **DB** contains w-s and id-s in inverted index form. The transformed database **MDB** also contains the mkws and the associated id-s in inverted index layout. Since there are a large number of w-s in **DB**, length of each binary string mkw is large. Hence, we hash those strings prior to writing to **MDB**. This **MDB** is further encrypted using the underlying OXT setup to generate the encrypted meta-keyword database $\widehat{\mathbf{EDB}}$.

We report the actual size of $\widehat{\mathbf{EDB}}$ in Figure 3 which is offloaded to the server. The query translation process first generates these mkws in binary string format and we hash those prior to search over the encrypted meta-keyword database $\widehat{\mathbf{EDB}}$.

Evaluation of Storage Overhead. One of the fundamental aspects of our implementation is that TWINSSE$_{\mathsf{OXT}}$ improves upon the quadratic storage overhead of IEX-2Lev and scales linearly with the size of plaintext database. IEX-2Lev exploits the low size of mutual intersections for all pairs of keywords in **DB** and its storage overhead scales with the size of the intersection. In a sparse data set, the size of these intersections for most of the

[8] `https://www.cs.cmu.edu/~enron/`
`https://www.kaggle.com/wcukierski/enron-email-dataset`

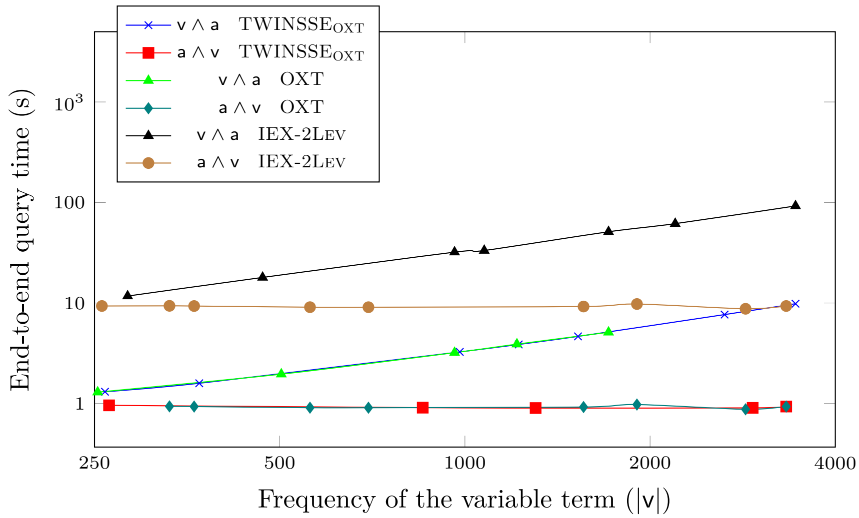

Figure 4: Comparison of end-to-end search latency vs frequency of the variable term ($|\mathsf{v}|$). Observe that, for conjunctive queries TWINSSE$_{\mathsf{OXT}}$ closely follows OXT in practice, and the latency is significantly less than IEX-2Lev. We note here that a fundamental difference between IEX-2Lev and TWINSSE$_{\text{OXT}}$ is that the search frequency of IEX-2Lev scales (by design) with the frequency of the *most* frequent conjunct, while that of TWINSSE$_{\text{OXT}}$ scales with the frequency of the least frequent conjunct; this is the main reason why TWINSSE$_{\text{OXT}}$ outperforms IEX-2Lev by a significant margin for conjunctive queries.

pairs of keywords is very low. However, if the database is not sparse, this results in large intersections for pairs of w-s and the overhead becomes truly quadratic for IEX-2Lev.

Figure 3 compares the storage overhead of TWINSSE$_{\mathsf{OXT}}$ and IEX-2Lev on Enron database (sparse database). It is evident that the storage size scales linearly with the number of keywords in **DB** for TWINSSE$_{\mathsf{OXT}}$ whereas IEX-2Lev becomes quadratic leading to storage blow-up. The storage overhead of IEX-2Lev is 60× more than TWINSSE$_{\mathsf{OXT}}$. Despite the additional storage required for the meta-keywords, TWINSSE$_{\mathsf{OXT}}$ has better storage overhead in worst-case distribution of **DB** as compared to IEX-2Lev.

Effect of linearization. As discussed in Section 4.3, the choice of n' greatly influences the storage overhead. Since the distribution of ws and ids varies across different databases (for example, a medical database's distribution differs from a tax record database), it is quite challenging (and inefficient) to obtain an analytic expression for n' that works for multiple databases. We rely on an empirically chosen value of n' that suitably works for different databases without blowing up the storage. We present experimental results in Figure 5 to illustrate the effect of varying n'. We fix n' at 10 for our final experiments.

Evaluation of End-to-End Search Latency. Figure 4 and 6 compare the end-to-end search latency of TWINSSE$_{\mathsf{OXT}}$ with that of IEX-2Lev for conjunctive and disjunctive queries, respectively.

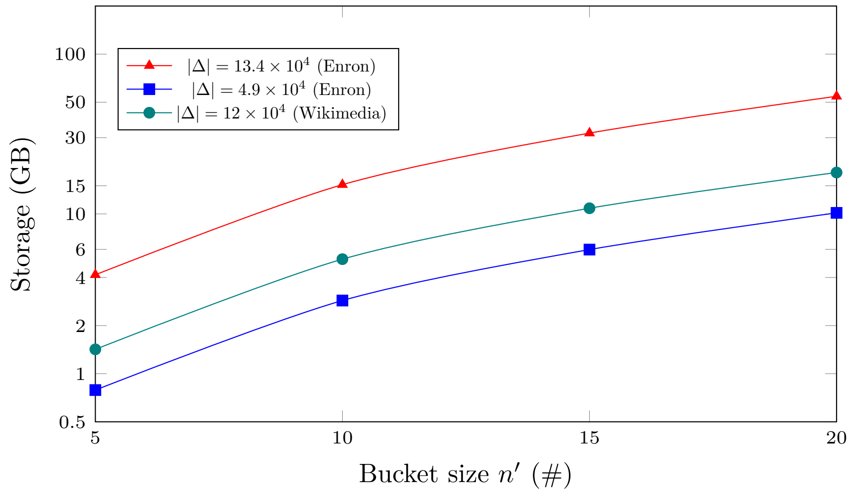

Figure 5: Variation of storage overhead for different choices of the bucket size (n').

Conjunctive queries. Search performance of TWINSSE$_{\mathsf{OXT}}$ is inherited from OXT and is therefore identical to OXT as shown in Figure 4. To validate this, we consider a two-keyword query of the form $q = \mathsf{a} \wedge \mathsf{v}$, where a and v are two keywords from **DB**. Without loss of generality, we consider the first term of q (or a here) to be the least frequent keyword. We vary the frequency of v (referred to as the variable term) with different queries where as the frequency of a is kept constant (constant term). The plot shows constant time overhead for conjunctive queries of this form with TWINSSE$_{\mathsf{OXT}}$, which is identical with OXT. In the same figure, IEX-2Lev conjunctive search time is also plotted which depicts that TWINSSE$_{\mathsf{OXT}}$ is around 10× faster on average.

Disjunctive queries. The plots in Figure 6 compare the end-to-end query time with final result size for disjunctive queries of different hamming weights. Observe that, the query time increases with increasing number of id-s in $\bar{\mathcal{R}}_q$ (the obtained result set, inclusive of the spurious id-s for a disjunctive query) due to the increased frequency of least-frequent mkws in these queries.

As discussed in the main text, frequency of the least-frequent mkw is independent of the frequency of the least-frequent w in a query. Hence, we consider plotting with overall result size that represents the computation overhead. In disjunctive queries, union of the id-s grows with more number of keywords present in the query. Therefore, plotting with the result size provides an accurate measure of computation cost for disjunctive queries. Nonetheless, the OXT sublinear search complexity is maintained, which we verified in our experiments.

The average query time increases with the number of ws in actual disjunctive query q. This increased time attributes to more number of mkws for each query, and the underlying OXT that scales linearly with number of keywords (in this case mkws) in the conjunctive query. The end-to-end disjunctive search latency for TWINSSE$_{\mathsf{OXT}}$ is few hundred milliseconds over the Enron database for queries with moderate result size.

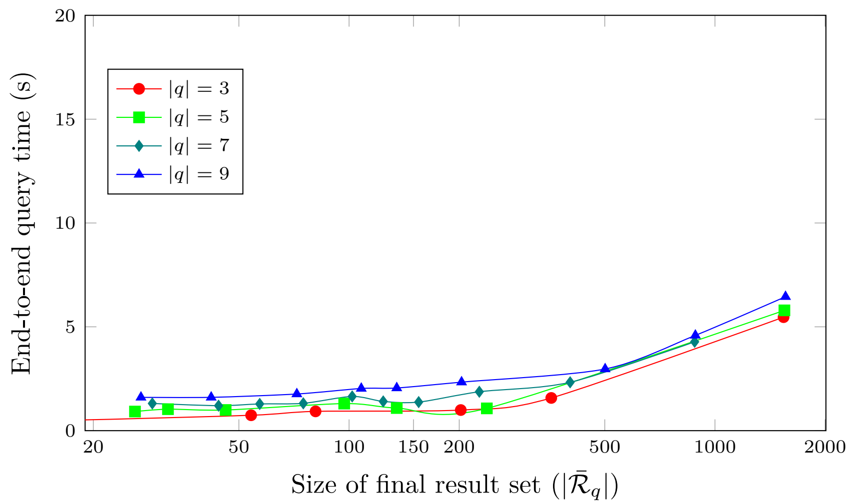

Figure 6: End-to-end search latency vs final result size for disjunctive queries of different number of ws in q ($|q|$).

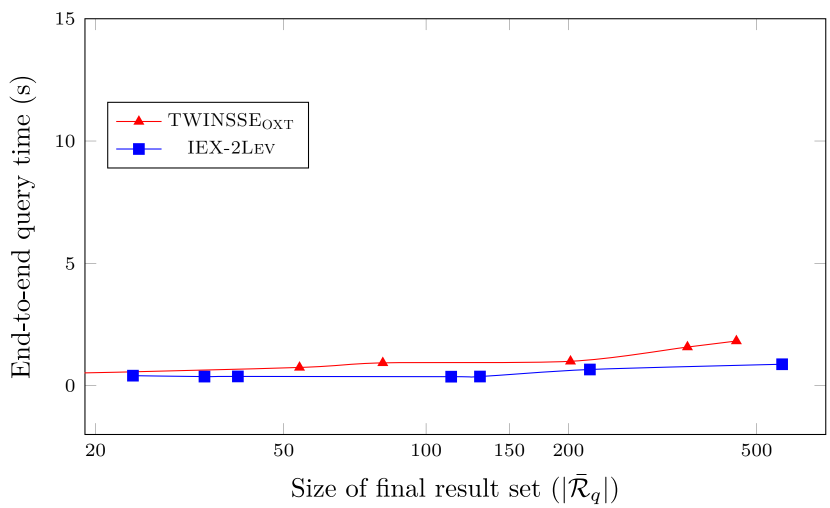

Figure 7: Comparison of end-to-end search latency vs final result size for disjunctive queries of the form $q = w_1 \vee w_2$ over Enron data set.

We provide an end-to-end query performance comparison of TWINSSE$_{\text{OXT}}$ with IEX-2Lev in Figure 7. For queries with smaller result sizes, TWINSSE$_{\text{OXT}}$ achieves almost identical end-to-end query latency as IEX-2Lev. For queries with larger result sizes, IEX-2Lev performs slightly better. This is primarily because of the usage of relatively costly elliptic-curve cryptography-based operations in TWINSSE$_{\text{OXT}}$ (a consequence of using OXT as a black-box, which uses such operations); IEX-2Lev, on the other hand, uses purely symmetric-key crypto-primitives. We view this as an efficiency trade-off; note that TWINSSE$_{\text{OXT}}$ outperforms IEX-2Lev significantly both in terms of storage require-

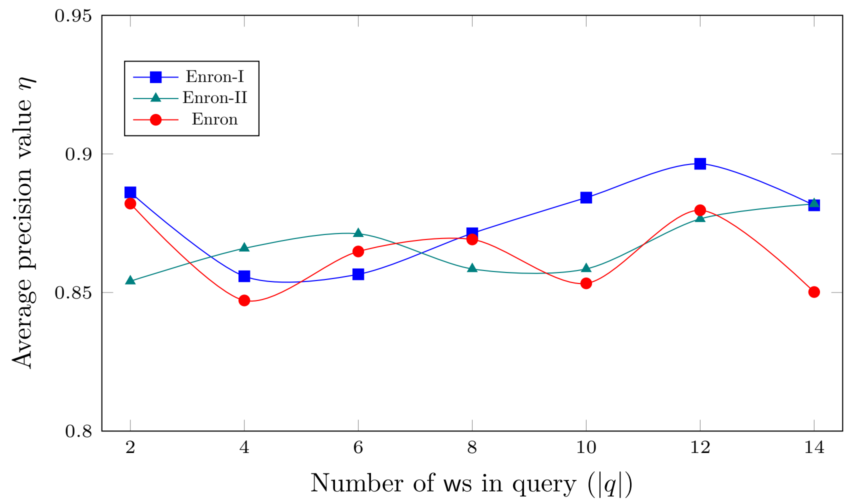

Figure 8: Average precision value vs number of keywords in query for databases of different size.

ments (as demonstrated in Figure 3) and end-to-end latency for conjunctive queries (as demonstrated in Figure 4). Hence, from the point of view of practical performance across a wide class of Boolean queries generally encountered in practice and scalability to extremely large databases, TWINSSE$_{\mathrm{OXT}}$ outperforms IEX-2Lev.

Experiments on the Wiki database. We also present experimental results for performance and storage overhead evaluation over the Wikimedia dataset[9] in Appendix 6.2. We observed similar results with the Wikimedia dataset as of the Enron dataset presented above.

Evaluation of Result Precision. In context of information retrieval *precision* (denoted by η in Section 3.3) is the fraction of relevant documents among the retrieved documents. We compare the average precision values of $\bar{\mathcal{R}}_q$ for disjunctive queries (q) with different number of keywords in Figure 8. Observe that the average precision values for most of the cases is above 85%, which implies that at least 85% documents returned by $\bar{\mathcal{R}}_q$ is relevant to the disjunctive query q (or belongs to the actual result set $\mathcal{R}_q$ without spurious id-s). The plot also illustrates that scaling the database does not affect the average precision of the retrieved documents. Hence, the query result of TWINSSE$_{\mathrm{OXT}}$ does not degrade even for huge databases which is crucial for practical applications.

Note that, IEX is an exact solution that has 100% result precision - it returns the exact result set without spurious ids[10]. However, IEX incurs extremely high storage overhead that makes it impossible to deploy with large real datasets. In contrast, TWINSSE incurs less than 100% result precision (85%-90%, as shown in Figure 8), but TWINSSE outweighs the loss in precision with storage savings (10-50 times less than IEX, as shown in Figure 3).

[9] `https://dumps.wikimedia.org/enwiki/latest/`

[10] State-of-art SSE schemes like OXT [CJJ+13], HXT [LPS+18], ODXT [PM21] or IEX [KM17] are exact solutions. Hence, this precision parameter is defined exclusively for TWINSSE only which produces result set with spurious ids

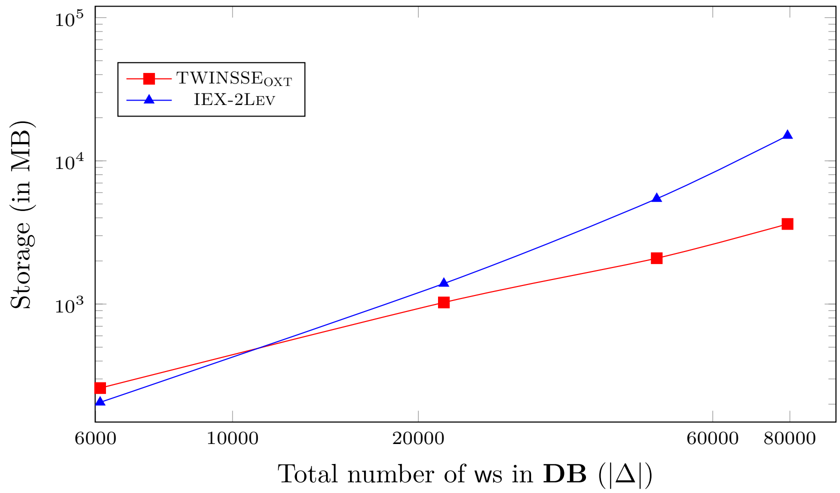

Figure 9: Server storage overhead with plain database size ($|\mathbf{DB}|$) for the Wikimedia dataset.

6.2 Experimental Results over the Wikimedia Dump

We present additional experimental results for TWINSSE$_{\text{OXT}}$ over Wikimedia databases[11] in this section. We varied the database size from $6k$ keywords ($60k$ w-id pairs in the plain index) to $80k$ keywords (8.2 million w-id pairs in the plain index), and we plot the server storage overhead in Figure 9 and performance figures in Figure 10 and Figure 11.

The comparative storage overhead plot (in log scale) in Figure 9 illustrates the quadratic storage overhead for IEX-2Lev; whereas it remains linear for TWINSSE$_{\text{OXT}}$. This storage overhead profile validates our primary contribution of our work, and also illustrates the applicability towards different databases (results on the Enron dataset is presented in the main text Section 6.)

6.3 Evaluation of Storage Overhead with Synthetic Database

We discussed in Section 6 Figure 3 that TWINSSE$_{\text{OXT}}$ improves significantly in terms of storage overhead than IEX-2Lev on the Enron database.

Figure 12 compares the storage overhead of TWINSSE$_{\text{OXT}}$ and IEX-2Lev on a synthetic database that follows Zipf's law and Figure 13 compares the estimated storage overhead of TWINSSE$_{\text{OXT}}$ and IEX-2Lev on a synthetic database that follows a uniform distribution. These databases contain more documents per keyword than the Enron database. This implies that size of the intersections of keyword pairs is much more as compared to the Enron database. Storage overhead of IEX-2Lev hence degrades even more.

To clarify this, the following example of a realistic database can be considered as dense one

[11] `https://dumps.wikimedia.org/enwiki/latest/`

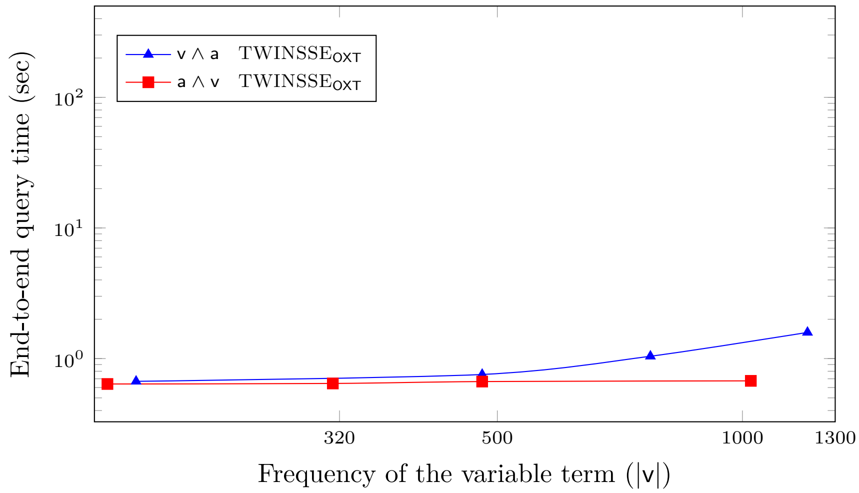

Figure 10: $\text{TWINSSE}_{\text{OXT}}$ end-to-end conjunctive query search latency vs frequency of the variable term ($|\mathsf{v}|$) for Wikimedia dataset.

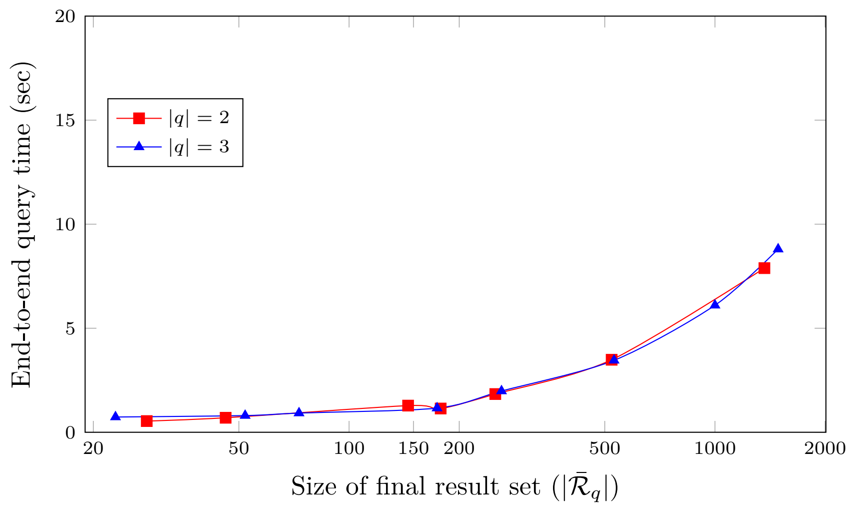

Figure 11: $\text{TWINSSE}_{\text{OXT}}$ end-to-end disjunctive query search latency vs final result size for Wikimedia dataset.

(as we have described above and in Section 6). Note that, any relational-database is dense if each attribute is low-entropy (i.e., takes only a few values), and hence each attribute-value pair (equivalent to keywords) occurs in a very large number of records (equivalent to documents). Consider the following Covid-19 patient-database (Table 2), where each attribute-value-pair likely occurs in a large number of patient-records.

Observe that, querying any of the attributes would return a large number of records from this example database. Our experimental results show that IEX-2Lev incurs 70× higher

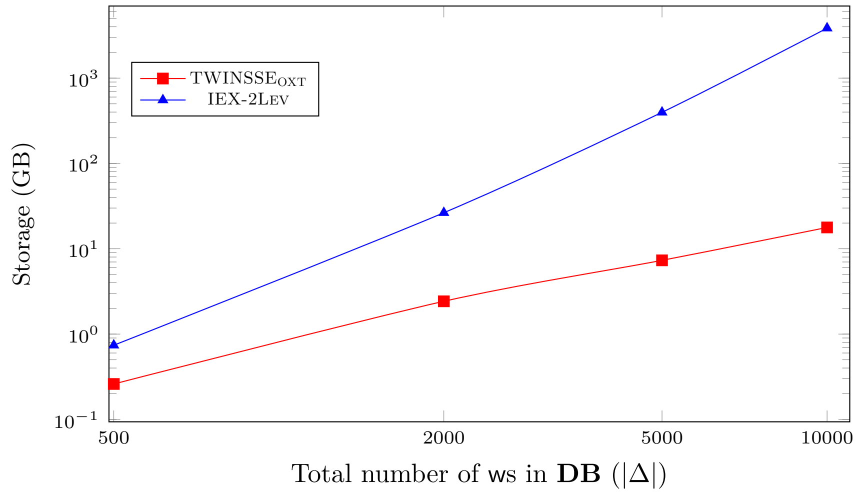

Figure 12: Server storage overhead with number of keywords in synthetic plain database ($|\Delta|$) (prepared following Zipf's distribution).

Table 2: Example of a dense database with possible attribute-value pairs.

Attributes	**Values**
Symptomatic	Yes/No
Not-Vaccinated	Yes/No
Dose 1	Yes/No
Dose 2	Yes/No
Booster Dose	Yes/No

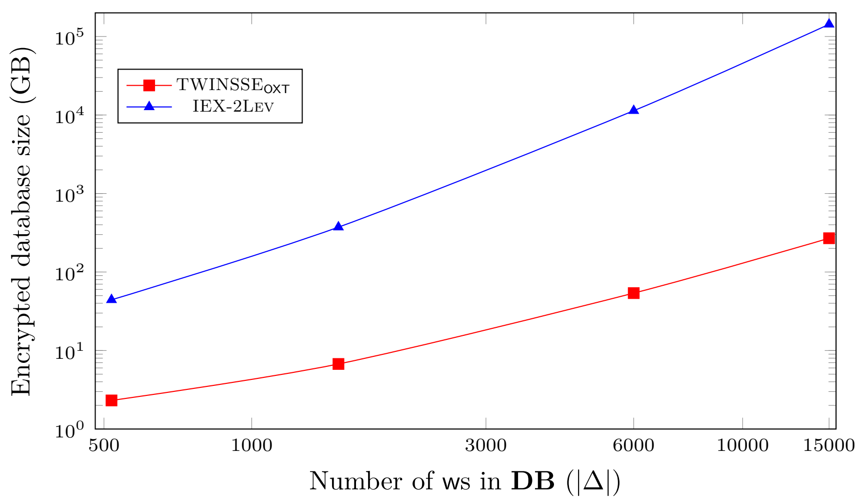

Figure 13: Server storage vs $|\Delta|$ for synthetic DB (following uniform distribution).

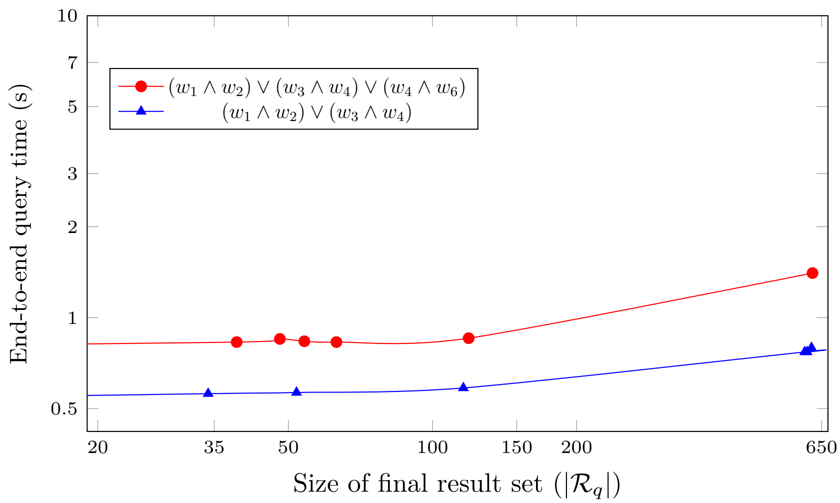

Figure 14: TWINSSE$_{\text{OXT}}$ performance with result set size on Enron dataset for DNF queries.

storage overhead than TWINSSE$_{\text{OXT}}$ for the synthetic database following Zipf's law (Figure 12) and approximately 150× higher storage overhead for the database following uniform distribution (Figure 13). The search time also increases for both the schemes; however, the main advantage of TWINSSE$_{\text{OXT}}$ compared to IEX is in reduced storage, not in search overheads (which still remains sublinear for TWINSSE$_{\text{OXT}}$).

6.4 Experimental Results for CNF and DNF Queries

We provide experimental results for CNF and DNF queries using TWINSSE$_{\text{OXT}}$ in this section. We experimented over the Enron dataset on the same platform (discussed in Section 6) with our implementation of TWINSSE$_{\text{OXT}}$.

DNF queries. We considered multiple queries with two clauses and three clauses with each clause having two keywords. The end-to-end query time is plotted in Figure 14, where the blue curve represents the query time for two clause queries and the red curve represents the query time for three clause queries. Observe that the query time for both two and three clause queries increase with more number of ids in the final result set. This increment can be attributed to large result size of the individual conjunctive clauses. Also note that the query time increases for three-clause queries due more conjunctive clauses and follows the same trend of increased query time with the final result size.

CNF queries. For experimenting with CNF queries, we considered two-clause queries with two keywords and three keywords per clause. Since the Enron dataset is relatively sparse in nature, with higher number of clauses in query it often results in small or empty intersection. We plotted the end-to-end query time in Figure 15 for both cases – two keyword clauses and three keyword clauses with the size of the final result set. The blue curve represents

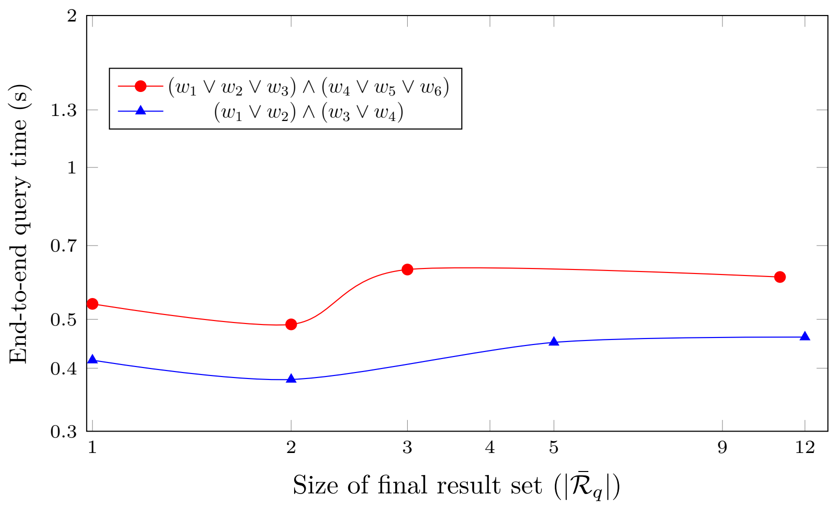

Figure 15: $\text{TWINSSE}_{\text{OXT}}$ performance with result set size on Enron dataset for CNF queries.

the end-to-end query time for the queries with two keywords per clause. Similarly, the red curve represents the end-to-end query time for queries with three-keyword clauses. Observe that, in CNF queries also, the end-to-end query time increases with the final result size, due to the increased size of the initial result set. For the three-keyword clauses, the query time is higher than the two-keyword clauses due to the increased size of the initial result set obtained by disjunctive query.

7 Supporting Dynamic Databases

In this paper, we described $\text{TWINSSE}_{\text{OXT}}$ for static databases. This leaves open the question of extending $\text{TWINSSE}_{\text{OXT}}$ to dynamic databases, and supporting updates efficiently yet securely over these.

We note here that for dynamic databases where the set of keywords across all documents remains fixed (or, more generally, undergoes updates infrequently), the set of meta-keywords also does not change (frequently) over time. In this setting, it is possible achieve an extension of $\text{TWINSSE}_{\text{OXT}}$ to the setting of dynamic databases by simply substituting the underlying OXT scheme with a dynamic conjunctive SSE scheme with desirable efficiency and security guarantees (e.g. ODXT from [PM21]).

However, such an extension becomes challenging for dynamic databases where the set of keywords (and hence, the set of meta-keywords) also gets updated frequently. We leave it as an interesting future question to extend $\text{TWINSSE}_{\text{OXT}}$ to dynamic databases.

Acknowledgements

The authors from IIT Kharagpur would like to thank the grant "Design and Implementation of Efficient and Secure Searchable Encryption" sponsored by MHRD-STARS (Scheme for Transformational and Advanced Research in Sciences), India for partially supporting this work. The authors also thank Kushagra Aggarwal for early discussions on this work.

References

[APP+21] Ghous Amjad, Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. Dynamic volume-hiding encrypted multi-maps with applications to searchable encryption. *IACR Cryptol. ePrint Arch.*, page 765, 2021.

[BKM20] Laura Blackstone, Seny Kamara, and Tarik Moataz. Revisiting leakage abuse attacks. In *NDSS 2020*, 2020.

[BMO17] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *ACM CCS 2017*, pages 1465–1482, 2017.

[Bos16] Raphael Bost. $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In *ACM CCS 2016*, pages 1143–1154, 2016.

[CGKO06] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM CCS 2006*, pages 79–88, 2006.

[CGPR15] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *ACM CCS 2015*, pages 668–679, 2015.

[CJJ+13] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO 2013*, pages 353–373, 2013.

[CJJ+14] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS 2014*, 2014.

[CK10] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In *ASIACRYPT 2010*, pages 577–594, 2010.

[CM05] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *ACNS 2005*, pages 442–455, 2005.

[CPPJ18] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. New constructions for forward and backward private symmetric searchable encryption. In *ACM CCS 2018*, pages 1038–1055, 2018.

[EKPE18] Mohammad Etemad, Alptekin Küpçü, Charalampos Papamanthou, and David Evans. Efficient dynamic searchable encryption with forward privacy. *PoPETs*, 2018(1):5–20, 2018.

[FJK+15] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel-Catalin Rosu, and Michael Steiner. Rich queries on encrypted data: Beyond exact matches. In *ESORICS 2015*, pages 123–145, 2015.

[Gen09] C. Gentry. Fully homomorphic encryption using ideal lattices. In *ACM STOC'09*, pages 169–178, 2009.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[Goh03] Eu-Jin Goh. Secure indexes. *IACR Cryptology ePrint Archive*, 2003:216, 2003.

[GPPW20] Zichen Gui, Kenneth G. Paterson, Sikhar Patranabis, and Bogdan Warinschi. Swissse: System-wide security for searchable symmetric encryption. *IACR Cryptol. ePrint Arch.*, page 1328, 2020.

[IKK12] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS 2012*, 2012.

[KM17] Seny Kamara and Tarik Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *EUROCRYPT 2017*, pages 94–124, 2017.

[KM19] Seny Kamara and Tarik Moataz. Computationally volume-hiding structured encryption. In *EUROCRYPT 2019*, pages 183–213, 2019.

[KP13] Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In *FC 2013*, pages 258–274, 2013.

[KPR12] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *ACM CCS 2012*, pages 965–976, 2012.

[LPS+18] Shangqi Lai, Sikhar Patranabis, Amin Sakzad, Joseph K. Liu, Debdeep Mukhopadhyay, Ron Steinfeld, Shifeng Sun, Dongxi Liu, and Cong Zuo. Result pattern hiding searchable encryption for conjunctive queries. In *ACM CCS 2018*, pages 745–762, 2018.

[OK21] Simon Oya and Florian Kerschbaum. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In *USENIX Security 2021*, pages 127–142, 2021.

[PM21] Sikhar Patranabis and Debdeep Mukhopadhyay. Forward and backward private conjunctive searchable symmetric encryption. In *NDSS 2021*, 2021.

[PPSY21a] Sarvar Patel, Giuseppe Persiano, Joon Young Seo, and Kevin Yeo. Efficient boolean search over encrypted data with reduced leakage. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2021*, volume 13092 of *Lecture Notes in Computer Science*, pages 577–607. Springer, 2021.

[PPSY21b] Sarvar Patel, Giuseppe Persiano, Joon Young Seo, and Kevin Yeo. Efficient boolean search over encrypted data with reduced leakage. *IACR Cryptol. ePrint Arch.*, page 1227, 2021.

[PPYY19] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACMCCS 2019*, pages 79–93. ACM, 2019.

[SWP00] Dawn Xiaodong Song, David A. Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *IEEE S&P 2000*, pages 44–55, 2000.

[SYL+18] Shifeng Sun, Xingliang Yuan, Joseph K. Liu, Ron Steinfeld, Amin Sakzad, Viet Vo, and Surya Nepal. Practical backward-secure searchable encryption from symmetric puncturable encryption. In *ACM CCS 2018*, pages 763–780, 2018.

[ZKP16] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security Symposium 2016*, pages 707–720, 2016.

A Proof of Theorem 4.1 (Correctness of TWINSSE$_{\text{Basic}}$ and TWINSSE)

The proof of correctness for TWINSSE$_{\text{Basic}}$ (and TWINSSE as well) follows from the correctness of CSSE. The correctness of CSSE ensures that a conjunctive query $q = \mathsf{w}_1 \wedge \cdots \wedge \mathsf{w}_n$ over an encrypted database satisfies the following relations.

$$\mathbf{EDB} = \text{CSSE.}\textsc{Setup}(\mathbf{DB}))$$
$$\mathbf{DB}(\mathsf{w}_1) \cap \cdots \cap \mathbf{DB}(\mathsf{w}_n) = \text{CSSE.}\textsc{Search}(q, \mathbf{EDB}))$$

We state the proof for TWINSSE$_{\text{Basic}}$ first. Then we show that this can be simply extended to main TWINSSE scheme (the final bucketized version).

Proof for TWINSSE$_{\text{Basic}}$. Proof of the TWINSSE$_{\text{Basic}}$ directly follows from the proof of Lemma 3.1. Consider a disjunctive query q as stated below.

$$q = \mathsf{w}_1 \vee \cdots \vee \mathsf{w}_n$$

The equivalent conjunctive expression of meta-keywords can be expressed as below.

$$q_{\mathsf{mkw}} = \mathsf{mkw}_{i_0,j_0} \wedge \mathsf{mkw}_{i_1,j_1} \wedge \ldots \wedge \mathsf{mkw}_{i_n,j_n}$$

We write the following relation from Lemma 3.1.

$$\mathbf{DB}(q_{\mathsf{mkw}}) = \mathbf{DB}(q) \cup \Big(\bigcap_{k=0}^{n} \Big(\bigcup_{\substack{l \in [N] \setminus (\{[i_k, j_k]\} \\ \cup \{\ell_r : r \in [n]\})}} \mathbf{DB}(\mathsf{w}_l) \Big) \Big)$$

It easy to notice from the above equation that $\mathbf{DB}(q) \subseteq \mathbf{DB}(q_{\mathsf{mkw}})$. Hence, all *id*s of the actual result set of disjunctive query q is included in the result set obtained from the query using TWINSSE$_{\text{Basic}}$.Search, which proves the correctness of TWINSSE$_{\text{Basic}}$.

Proof of TWINSSE. Recall from Section 4, that in TWINSSE all ws from Δ are partitioned into n_{B} buckets of uniform size, and we execute the basic meta-keyword generation method developed in TWINSSE$_{\text{Basic}}$ over each partition independently. Only those partitions with query meta-keywords are accessed during search.

Assume that the dictionary of ws - Δ is partitioned in the following way,

$$\Delta = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_{n_{\mathsf{B}}}$$

where n_{B} is the number of buckets and each bucket Δ_u can be expressed in the following way.

$$\Delta_u = \{\mathsf{w}_{(u-1)n'+1}, \mathsf{w}_{(u-1)n'+2}, \cdots, \mathsf{w}_{un'}\}$$

The number of ws in each bucket is denoted by n'. The set of mkws in each bucket Δ_k is denoted by $\mathcal{S}_{\mathsf{mkw},k}$. TWINSSE$_{\text{Basic}}$ is executed over each of these bucket individually to generate the encrypted database.

The query expression follows from the TWINSSE construction with the above structure (discussed in Section 4).

$$\mathbf{DB}(q_{\mathsf{mkw}}) = \bigcup_{u \in [n_{\mathsf{B}}]} \mathbf{DB}(q_{\mathsf{mkw},u})$$

and the actual query q can be partitioned as $q = \bigvee_{u \in [n_{\mathsf{B}}]} q_u$. We expand the above expression to individual buckets.

$$\begin{aligned}
\mathbf{DB}(q_{\mathsf{mkw}}) &= \bigcup_{u \in [n_{\mathsf{B}}]} \mathbf{DB}(q_{\mathsf{mkw},u}) \\
&= \bigcup_{u \in [n_{\mathsf{B}}]} \Big(\mathbf{DB}(q_u) \cup \Big(\bigcap_{k=0}^{|q_u|} \Big(\bigcup_{\substack{\in[|\Delta_u|]\setminus \\ (\{[i_k, j_k]\} \\ \cup \{\ell_r : r \in [|q_u|]\})}} \mathbf{DB}(\mathsf{w}_l) \Big)\Big)\Big) \\
&= \bigcup_{u \in [n_{\mathsf{B}}]} \mathbf{DB}(q_u) \cup \bigcup_{u \in [n_{\mathsf{B}}]} \Big(\bigcap_{k=0}^{|q_u|} \Big(\bigcup_{\substack{l \in[|\Delta_u|]\setminus \\ (\{[i_k, j_k]\} \\ \cup \{\ell_r : r \in [|q_u|]\})}} \mathbf{DB}(\mathsf{w}_l) \Big)\Big) \\
&= \mathbf{DB}(q) \cup \bigcup_{u \in [n_{\mathsf{B}}]} \Big(\bigcap_{k=0}^{|q_u|} \Big(\bigcup_{\substack{l \in[|\Delta_u|]\setminus \\ (\{[i_k, j_k]\} \\ \cup \{\ell_r : r \in [|q_u|]\})}} \mathbf{DB}(\mathsf{w}_l) \Big)\Big)
\end{aligned}$$

Clearly, from the above expression $\mathbf{DB}(q) \subseteq \mathbf{DB}(q_{mkw})$, where $|\mathbf{DB}(q)| = \eta \cdot |\mathbf{DB}(q_{mkw})|$ (recall from Section 3.3 that η is the precision parameter), which proves the correctness of result for TWINSSE.

B Detailed Analysis and Discussion on the Leakage of $\mathrm{TWINSSE}_{\mathrm{OXT}}$

In Section 4.4, we informally described the leakage profile for TWINSSE built in a black-box way from any generic conjunctive SSE scheme CSSE. In this section, we formally detail the leakage profile for the specific instantiation of TWINSSE based on the OXT scheme, namely $\mathrm{TWINSSE}_{\mathrm{OXT}}$. We then present a discussion on the leakage profiles for both TWINSSE and $\mathrm{TWINSSE}_{\mathrm{OXT}}$.

B.1 Security of TWINSSE

We present a formal description of the security guarantees of our generic construction TWINSSE. Concretely, we state the following theorem.

Theorem B.1 (Security of TWINSSE)**.** *Assuming that* CSSE *is an (adaptively) secure SSE scheme with respect to the leakage function* $\mathcal{L}_{\mathrm{CSSE}} = (\mathcal{L}_{\mathrm{CSSE}}^{\mathrm{SETUP}}, \mathcal{L}_{\mathrm{CSSE}}^{\mathrm{SEARCH}})$*,* TWINSSE *is an (adaptively) secure SSE scheme with respect to the leakage function* $\mathcal{L}_{\mathrm{TWINSSE}} = (\mathcal{L}_{\mathrm{TWINSSE}}^{\mathrm{SETUP}}, \mathcal{L}_{\mathrm{TWINSSE}}^{\mathrm{SEARCH}})$*, where for any plaintext database* $\mathbf{DB}$*, any search query* q*, and any pair of bucketization parameters* (n', n_{B})*, we have*

$$\mathcal{L}_{\mathrm{TWINSSE}}^{\mathrm{SETUP}}(\mathbf{DB}) = \left(\mathcal{L}_{\mathrm{CSSE}}^{\mathrm{SETUP}}(\widehat{\mathbf{DB}}), n', n_{\mathsf{B}}\right),$$

where $\widehat{\mathbf{DB}} = \textsc{GenMetaDB}(\mathbf{DB}, n', n_{\mathsf{B}})$*, and*

$$\mathcal{L}_{\mathrm{TWINSSE}}^{\mathrm{SEARCH}}(q) = \begin{cases} \mathcal{L}_{\mathrm{CSSE}}^{\mathrm{SEARCH}}(q) \text{ if } q \text{ is conjunctive,} \\ \left\{\mathcal{L}_{\mathrm{CSSE}}^{\mathrm{SEARCH}}(q_{\mathsf{mkw},k})\right\}_{k \in [n_{\mathsf{B}}]} \text{ if } q \text{ is disjunctive,} \end{cases}$$

where

$$q_{\mathsf{mkw}} = \bigvee_{k \in [n_{\mathsf{B}}]} q_{\mathsf{mkw},k} = \textsc{GenMQuery}(q, n', n_{\mathsf{B}}).$$

Proof. We defer the formal proof of this theorem to Appendix C.

B.2 Leakage Profile of $\mathrm{TWINSSE}_{\mathrm{OXT}}$

In this section, we describe the leakage profile of $\mathrm{TWINSSE}_{\mathrm{OXT}}$. We begin by recalling from [CJJ+13] the leakage profile of the original OXT scheme. We then build upon it to describe the leakage profile of $\text{TWINSSE}_{\text{OXT}}$, which is actually very similar in spirit to the leakage profile of OXT.

Setup Leakage. The setup leakage in the OXT scheme consists of the size of the database **DB**, which is nothing but the total number of keyword-document pairs in the database **DB**, formally defined as

$$|\mathbf{DB}| = \sum_{\mathsf{w} \in \Delta} |\mathbf{DB}(\mathsf{w})|,$$

where $\Delta = \{\mathsf{w}_1, \ldots, \mathsf{w}_N\}$ is the dictionary over which the database **DB** is defined.

Search Leakages. Next, we summarize the leakages incurred by OXT during conjunctive keyword search queries.

Result Pattern Leakage: The server learns the final set of document identifiers matching the query. Formally, for a conjunctive query q, the result pattern leakage RP is defined as

$$\mathsf{RP}(q) = \mathbf{DB}(q).$$

Size Pattern Leakage. The server learns the frequency of the s-term (where s-term again refers to the least frequent keyword in the conjunction). Formally, for a conjunctive query $q = (\mathsf{w}_1 \wedge \ldots \wedge \mathsf{w}_n)$, where w_1 is the least frequent keyword in the conjunction, the size pattern SP is defined as

$$\mathsf{SP}(q) = |\mathbf{DB}(\mathsf{w}_1)|.$$

Equality Pattern Leakage. The server learns if two (or more) conjunctive queries have the same s-term (where s-term again refers to the least frequent keyword in the conjunction). Formally, for a sequence of conjunctive queries $(q_1, \ldots, q_M)$, where for each $i \in [M]$, we have

$$q_i = (\mathsf{w}_{i,1} \wedge \ldots \wedge \mathsf{w}_{i,n_i}),$$

where $\mathsf{w}_{i,1}$ is the least frequent keyword in the conjunction, the equality pattern leakage EP is defined as an $M \times M$ matrix where for each $i, j \in [M]$, we have

$$\mathsf{EP}[i,j] = \begin{cases} 1 & \text{if } \mathsf{w}_{i,1} = \mathsf{w}_{j,1}, \\ 0 & \text{otherwise.} \end{cases}$$

Conditional Intersection Pattern Leakage. The server learns if two (or more) conjunctive queries have one or more x-terms in common (where x-term refers any keyword other than the least frequent keyword in the conjunction); more concretely, if two (or more) conjunctive queries have one or more x-terms in common, then the server learns the intersection of the set of documents containing the corresponding s-terms. Formally, for a sequence of conjunctive queries $(q_1, \ldots, q_M)$, where for each $i \in [M]$, we have

$$q_i = (\mathsf{w}_{i,1} \wedge \ldots \wedge \mathsf{w}_{i,n_i}),$$

where $\mathsf{w}_{i,1}$ is the least frequent keyword in the conjunction, the conditional intersection pattern leakage IP is defined as an $M \times M$ matrix of lists, where for each $i, j \in [M]$, we have

$$\mathsf{IP}[i,j] = \begin{cases} \mathbf{DB}(\mathsf{w}_{i,1}) \cap \mathbf{DB}(\mathsf{w}_{j,1}) & \text{if } \overline{\mathsf{IP}}[i,j] = 1, \\ \phi & \text{if } \overline{\mathsf{IP}}[i,j] = 0, \end{cases}$$

where $\overline{\mathsf{IP}}[i,j] = 1$ if and only if there exists at least one pair $(\ell_i, \ell_j) \in [n_i] \times [n_j]$ such that $\mathsf{w}_{i,\ell_i} = \mathsf{w}_{j,\ell_j}$; otherwise, we have $\overline{\mathsf{IP}}[i,j] = 0$.

Security of $\text{TWINSSE}_{\text{OXT}}$**.** We now formalize the security of $\text{TWINSSE}_{\text{OXT}}$ in terms of the leakage profiles described above. We do this using a formal theorem, which may be viewed as a specialization of Theorem B.1 to a specific instantiation of TWINSSE based on OXT. Once again, this theorem is based on the (adaptive) simulation-security definition of SSE in the real world-ideal world paradigm.

Theorem B.2 (Security of $\text{TWINSSE}_{\text{OXT}}$)**.** $\text{TWINSSE}_{\text{OXT}}$ *is an (adaptively) secure SSE scheme with respect to the leakage function* $\mathcal{L}_{\text{TWINSSE}_{\text{OXT}}} = (\mathcal{L}^{\text{Setup}}_{\text{TWINSSE}_{\text{OXT}}}, \mathcal{L}^{\text{Search}}_{\text{TWINSSE}_{\text{OXT}}})$*, where for any plaintext database* $\mathbf{DB}$*, any sequence of conjunctive queries* $\mathcal{Q}_0 = (q_{1,0}, \ldots, q_{M,0})$ *and any sequence of disjunctive queries* $\mathcal{Q}_1 = (q_{1,1}, \ldots, q_{M',1})$*, and any pair of bucketization parameters* (n', n_{B})*, we have*

$$\mathcal{L}^{\text{Setup}}_{\text{TWINSSE}_{\text{OXT}}}(\mathbf{DB}) = (|\widehat{\mathbf{DB}}|, n', n_{\mathsf{B}}),$$

where $\widehat{\mathbf{DB}} = \text{GenMetaDB}(\mathbf{DB}, n', n_{\mathsf{B}})$*, and*

$$\mathcal{L}^{\text{Search}}_{\text{TWINSSE}_{\text{OXT}}}(\mathcal{Q}_0, \mathcal{Q}_1) = [\mathsf{RP}, \mathsf{SP}, \mathsf{EP}, \mathsf{IP}](\mathcal{Q}_0, \mathcal{Q}_{\mathsf{mkw},1}),$$

where $\mathcal{Q}_{\mathsf{mkw},1}$ *is a sequence of (sub-)meta-queries of the form*

$$\mathcal{Q}_{\mathsf{mkw},1} = \{q_{\mathsf{mkw},k,\ell}\}_{k \in [n_{\mathsf{B}}], \ell \in [M']},$$

where for each $\ell \in [M']$*, we have*

$$q_{\mathsf{mkw},\ell} = \left(\bigvee_{k \in [n_{\mathsf{B}}]} q_{\mathsf{mkw},k,\ell} \right) = \text{GenMQuery}(q_{\ell,1}, n', n_{\mathsf{B}}).$$

Proof. We defer the formal proof of this theorem to Appendix C.

B.3 Discussion on the Leakage Profile of $\text{TWINSSE}_{\text{OXT}}$

In this subsection, we present a more in-depth analysis of the leakage profile for $\text{TWINSSE}_{\text{OXT}}$ during conjunctive and disjunctive search queries, and its implications.

Output Leakage. We begin by noting that the output leakage (alternatively, the result pattern leakage) is incurred by nearly all existing SSE schemes, including static and dynamic schemes, in the setting of both single and conjunctive keyword searches (such as in [CGKO06,

$\mathrm{CJJ^+13, LPS^+18, BMO17, CPPJ18, SYL^+18}$]). This is usually considered acceptable in the SSE literature; indeed the few known data/query recovery attacks that manage to exploit this leakage ([IKK12, CGPR15, ZKP16, BKM20]) assume extremely strong adversarial models where the adversary has partial knowledge of the plaintext database/queries.

s-Term Leakages. We focus next on the leakages related to the s-term, namely the size and equality pattern leakages. We begin by noting that these leakages are somewhat inherent to the design paradigm of OXT, which we base our instantiation of TWINSSE on. Even in the simpler setting of single keyword search, most existing schemes [CGKO06, CK10, CJJ[+]14, Bos16, BMO17, CPPJ18, SYL[+]18] also incur size and equality pattern leakages; the only constructions not to incur such leakages seem to rely on the use of ORAM-style data structures [BMO17, CPPJ18]. Fortifying $\mathrm{TWINSSE_{OXT}}$ with such data structures in an attempt to prevent this leakage is an interesting open challenge, although this would probably have to trade-off with some degradation in search performance (mostly in terms of communication complexity and number of rounds of communication during searches).

It is also possible (and perhaps conceptually simpler) to mask this leakage by using volume-hiding techniques such as padding and encrypted multi-maps (EMMs) [CGKO06, KM19, PPYY19, PPSY21b, APP[+]21]. This would incur a degradation in search performance, and it is up to the designer to decide on a suitable trade-off between performance and leakage.

However, we would like to point out that there are no known data/query recovery attacks on SSE schemes that specially exploit leakages related to the s-term. So we believe that *even without the aforementioned fortifications*, it appears that our $\mathrm{TWINSSE_{OXT}}$ scheme is not vulnerable to any known attacks due to the leakages related to the s-term, in realistic/practical adversarial settings.

x-Term Leakages. Next, we focus on the x-term leakages. We again note that these leakages are somewhat inherent to the design paradigm of OXT, which we base our instantiation of TWINSSE on.. The only known attack on conjunctive SSE schemes that exploits a form of x-term leakages is the *file injection attack* proposed by Zhang *et al.* in [ZKP16]. More concretely, the adversarial server must be able to compute $|\mathbf{DB}(\mathsf{w}_1) \cap \mathbf{DB}(\mathsf{w}_i)|$ when processing the search query.

We note however that for file injection attacks to work efficiently, the adversarial server must recover, for every x-term w_i, the result size corresponding to each sub-query of the form $\mathsf{w}_1 \cap \mathsf{w}_i$. However, the x-term leakage profile of $\mathrm{TWINSSE_{OXT}}$ is not sufficient to compute this term, since the set of xtoken values sent to the server is randomly permuted inside the underlying OXT instantiation precisely to mask such inference-style attacks. Further, one could also instantiate our generic construction of TWINSSE from other conjunctive SSE schemes such as HXT [LPS[+]18] that improve upon OXT in terms of provable security against leakage-based cryptanalysis and file-injection attacks.

Finally, fortifying implementations of $\mathrm{TWINSSE_{OXT}}$ by using either ORAM-style data structures or adopting volume-hiding techniques such as padding/EMMs may be useful in masking this leakage even further; however, even without such additional fortifications, it appears that our $\mathrm{TWINSSE_{OXT}}$ scheme is not vulnerable to file injection attacks, or any other known attacks for that matter, due to the leakages related to the s-term, in realistic/practical adversarial settings.

Leakage Cryptanalysis. Looking ahead, in Appendix D, we present a leakage-based cryptanalysis of the $\mathrm{TWINSSE}_{\mathrm{OXT}}$ scheme via experiments over the Enron email corpus. Our experiments help substantiate that the leakages incurred by the disjunctive search protocol in $\mathrm{TWINSSE}_{\mathrm{OXT}}$ are reasonably benign in practice and are quite resistant to even the most powerful leakage-based cryptanalysis techniques in the SSE literature over real-world databases, such as those in [CGPR15, ZKP16]. We leave it as an open question to extend the analysis using the more advanced leakage cryptanalysis techniques, such as those proposed in [BKM20, OK21].

C Security Proofs for TWINSSE and $\mathrm{TWINSSE}_{\mathrm{OXT}}$

In this section, we formally prove the security of TWINSSE and $\mathrm{TWINSSE}_{\mathrm{OXT}}$ with respect to the generic and specific leakage profiles described in Theorems B.1 and B.2, respectively.

C.1 Proof of Theorem B.1 (Security Analysis of TWINSSE)

We provide a simulation-based proof approach for TWINSSE. We assumed that the underlying adaptively secure CSSE has the following leakage profile.

$$\mathcal{L}_{\mathrm{CSSE}} = (\mathcal{L}_{\mathrm{CSSE}}^{\textsc{Setup}}, \mathcal{L}_{\mathrm{CSSE}}^{\textsc{Search}}).$$

We express the leakage of TWINSSE as,

$$\mathcal{L}_{\mathrm{TWINSSE}} = (\mathcal{L}_{\mathrm{TWINSSE}}^{\textsc{Setup}}, \mathcal{L}_{\mathrm{TWINSSE}}^{\textsc{Search}}),$$

where,

$$\mathcal{L}_{\mathrm{TWINSSE}}^{\textsc{Setup}}(\mathbf{DB}) = \mathcal{L}_{\mathrm{CSSE}}^{\textsc{Setup}}(\widehat{\mathbf{DB}}),$$

and, $\widehat{\mathbf{DB}} = \textsc{GenMetaDB}(\mathbf{DB}, n', n_{\mathsf{B}})$, and

$$\mathcal{L}_{\mathrm{TWINSSE}}^{\textsc{Search}}(q) = \begin{cases} \mathcal{L}_{\mathrm{CSSE}}^{\textsc{Search}}(q) & \text{if q is conjunctive,} \\ \{\mathcal{L}_{\mathrm{CSSE}}^{\textsc{Search}}(q_{\mathsf{mkw},k})\}_{k\in[n_{\mathsf{B}}]} & \text{if q is disjunctive,} \end{cases}$$

where

$$q_{\mathsf{mkw}} = \left(\bigvee_{k\in[n_{\mathsf{B}}]} q_{\mathsf{mkw},k} \right) = \textsc{GenMQuery}(q, n', n_{\mathsf{B}}).$$

We show that TWINSSE is secure against an adaptive semi-honest adversary $\mathcal{A}$, which has access to leakages from TWINSSE. We build a simulator SIM $\widehat{\mathbf{EDB}}$ generation by TWINSSE.$\textsc{Setup}$, and transcripts for queries over $\widehat{\mathbf{EDB}}$. The simulator simulates the transcripts τ_i for each query q_i. The simulator has the inputs from the leakage function $\mathcal{L}_{\mathrm{TWINSSE}}$ only, with the setup leakage $\mathcal{L}_{\mathrm{TWINSSE}}^{\textsc{Setup}}$ and the search leakage $\mathcal{L}_{\mathrm{TWINSSE}}^{\textsc{Search}}$.

Simulating TWINSSE.$\textsc{Setup}$**:** The following public parameters are available to SIM_{CSSE} as a part of $\mathcal{L}^{\textsc{Setup}}_{\text{TWINSSE}}$.

$$\{\mathbf{DB}, n', n_{\mathsf{B}}.\}$$

The simulator outputs the its version of $\widehat{\mathbf{EDB}}$ according to the simulation process of CSSE (we assumed that CSSE is provably simulation secure).

$$ct_{\widehat{\mathbf{EDB}}} = SIM^{\textsc{Setup}}_{\text{TWINSSE}}(\mathbf{DB}) = SIM^{\textsc{Setup}}_{\text{CSSE}}(\widehat{\mathbf{DB}}) = SIM^{\textsc{Setup}}_{\text{CSSE}}(\mathbf{DB}, n', n_B).$$

Since, CSSE is proven simulation secure, if follows from the simulation security guarantee of CSSE that $ct_{\widehat{\mathbf{EDB}}}$ is indistinguishable from the one generated in the real experiment.

Simulating TWINSSE.$\textsc{Search}$**:** For conjunctive queries the adversary does not have any advantage from $\mathcal{L}^{\textsc{Search}}_{\text{TWINSSE}}$ compared to $\mathcal{L}^{\textsc{Search}}_{\text{CSSE}}$, which exactly same as CSSE. For disjunctive queries we consider the effect of querying using q_{mkw}.

For disjunctive queries, we argue that the adversary $\mathcal{A}$ does not gain any information about the original disjunctive query with this simulation experiment. The distribution of $\widehat{\mathbf{DB}}$ (hence, also for $\widehat{\mathbf{EDB}}$) is abstracted from $\mathbf{DB}$ by the meta-keywords. The search leakages of CSSE is characterised by the $\mathcal{L}_{\text{CSSE}}$, provided from CSSE construction. Since, CSSE in TWINSSE executes over meta-keyword only, this leakage is expressed in the context of meta-keywords as below.

$$\mathcal{L}'_{\text{CSSE}} = \mathcal{L}_{\text{CSSE}}(meta - keywords).$$

With this leakage information of CSSE, the search leakage of TWINSSE can be expressed as below.

$$\mathcal{L}^{\textsc{Search}}_{\text{TWINSSE}}(q) = \mathcal{L}^{\textsc{Search}}_{\text{TWINSSE}}(q_{\mathsf{mkw},k})_{k \in [n_{\mathsf{B}}]} = \{\mathcal{L}'_{\text{CSSE}}, n_{\mathsf{B}}, n'\}.$$

The parameters n_{B} and n' are derived from N (number of keywords), which is available during setup. Therefore, the search leakage of TWINSSE same as the underlying CSSE, which can be summarised as below.

$$\mathcal{L}^{\textsc{Search}}_{\text{TWINSSE}}(q) = \mathcal{L}^{\textsc{Search}}_{\text{TWINSSE}}(q_{\mathsf{mkw},k})_{k \in [n_{\mathsf{B}}]} = \{\mathcal{L}'_{\text{CSSE}}\}.$$

This same leakage profile for search in TWINSSE and CSSE in the context of meta-keywords ensures that no additional information is leaked beyond CSSE leakage.

C.2 Proof of Theorem B.2 (Security Analysis of TWINSSE$_{\text{OXT}}$)

We resort to a simulation-based security analysis for TWINSSE$_{\text{OXT}}$. We assume a semi-honest adversary $\mathcal{A}$ which has access to the leakage from standard SSE leakages in an adaptive model. Security analysis of TWINSSE relies upon the semantic security notions provided by CSSE. TWINSSE inherits these notions through the core OXT (in case of TWINSSE$_{\mathsf{OXT}}$, the OXT) instance. We assume the following properties of OXT achieves with efficient performance.

1. Primitives used in construction of OXT hold the standard security assumptions.
2. OXT is non-adaptively and adaptively secure with the above assumptions.

We consider the following leakage profile for OXT.

$$\mathcal{L}_{\mathsf{OXT}} = \{\mathcal{L}_{\mathsf{OXT}}^{\textsc{Setup}}, \mathcal{L}_{\mathsf{OXT}}^{\textsc{Search}}\}.$$

Here, $\mathcal{L}_{\mathsf{OXT}}^{\textsc{Setup}}$ captures the leakage from the OXT.$\textsc{Setup}$, and $\mathcal{L}_{\mathsf{OXT}}^{\textsc{Search}}$ encapsulates the leakage from OXT.$\textsc{Search}$. More precisely, these can be expressed as,

$$\mathcal{L}_{\mathsf{OXT}}^{\textsc{Setup}}(\mathbf{DB}) = \{|\mathbf{DB}|\},$$

and

$$\mathcal{L}_{\mathsf{OXT}}^{\textsc{Search}}(\mathbf{EDB}, \{q_k\}_{q_k \in \mathcal{Q}_0}) = \{RP, SP, EP, IP\}$$

where, $\mathcal{Q}_0$ is a set of conjunctive queries. The leakages RP, SP, EP, and IP are the pattern leakages from OXT (see Appendix B.3).

We define the leakage profile of $\text{TWINSSE}_{\mathsf{OXT}}$ with respect to these above definitions and assumptions as below.

$$\mathcal{L}_{\text{TWINSSE}_{\mathsf{OXT}}} = \{\mathcal{L}_{\text{TWINSSE}_{\mathsf{OXT}}}^{\textsc{Setup}}, \mathcal{L}_{\text{TWINSSE}_{\mathsf{OXT}}}^{\textsc{Search}}\}.$$

The leakage functions above can be expressed as

$$\mathcal{L}_{\text{TWINSSE}_{\mathsf{OXT}}}^{\textsc{Setup}}(\mathbf{DB}) = \{|\widehat{\mathbf{DB}}|, n', n_{\mathsf{B}}\}$$

and

$$\mathcal{L}_{\text{TWINSSE}_{\text{OXT}}}^{\textsc{Search}}(\widehat{\mathbf{EDB}}, \mathcal{Q}_0, \mathcal{Q}_1) = [\mathsf{RP}, \mathsf{SP}, \mathsf{EP}, \mathsf{IP}](\mathcal{Q}_0, \mathcal{Q}_{\mathsf{mkw},1}),$$

.

For conjunctive queries,

$$\mathcal{L}_{\text{TWINSSE}_{\text{OXT}}}^{\textsc{Search}}(\widehat{\mathbf{EDB}}, \{q_k\}_{q_k \in \mathcal{Q}_0}) = [\widehat{\mathsf{RP}}, \widehat{\mathsf{SP}}, \widehat{\mathsf{EP}}, \widehat{\mathsf{IP}}].$$

Here, $\{\widehat{\mathsf{RP}}, \widehat{\mathsf{SP}}, \widehat{\mathsf{EP}}, \widehat{\mathsf{IP}}\}$ are the $\{\mathsf{RP}, \mathsf{SP}, \mathsf{EP}, \mathsf{IP}\}$ leakages in the context of meta-keywords. For conjunctive queries, it is exactly the same as OXT.

Since, OXT is simulation secure against these leakages, simulation security of $\text{TWINSSE}_{\text{OXT}}$ for conjunctive queries is straightforwardly implied from OXT.

In disjunctive queries, the query transformation process is carried out locally by the client, and the actual search is completed using $\mathsf{OXT}.\textsc{Search}$ protocol, we can write $\text{TWINSSE}_{\mathsf{OXT}}.\textsc{Search}$ leakage as

$$\mathcal{L}^{\textsc{Search}}_{\text{TWINSSE}_{\text{OXT}}}(\widehat{\mathbf{EDB}}, \{q_{mkw,1,k}\}_{k\in[\mathcal{Q}_1]}) = \{\widehat{\mathsf{RP}}, \widehat{\mathsf{SP}}, \widehat{\mathsf{EP}}, \widehat{\mathsf{IP}}\}.$$

We build a simulator SIM to simulate the $\widehat{\mathbf{EDB}}$ generation by $\text{TWINSSE}_{\text{OXT}}$ from $\mathbf{DB}$, and transcripts for query search over $\widehat{\mathbf{EDB}}$. The simulator simulates the transcripts τ_i for each query $q_i \in \mathcal{Q}$. The simulator has the inputs from the leakage function $\mathcal{L}_{\text{TWINSSE}_{\mathsf{OXT}}}$ only, with the setup leakage $\mathcal{L}^{\textsc{Setup}}_{\text{TWINSSE}_{\mathsf{OXT}}}$ and the search leakage $\mathcal{L}^{\textsc{Search}}_{\text{TWINSSE}_{\mathsf{OXT}}}$.

Simulating Setup: The following public parameters are available to SIM_{OXT} as a part of $\mathcal{L}^{\textsc{Setup}}_{\text{TWINSSE}_{\mathsf{OXT}}}$.

$$\{|\mathbf{EDB}|, |\widehat{\Delta}|\}.$$

The simulator outputs the its version of $\widehat{\mathbf{EDB}}$ according to the simulation process of OXT (we assumed that OXT is provably simulation secure).

$$ct_{\widehat{\mathbf{EDB}}} = SIM_{\mathsf{OXT}}.\textsc{Setup}(|\mathbf{MDB}|, |\widehat{\Delta}|).$$

It follows from the simulation security guarantee of OXT that $\mathsf{ct}_{\widehat{\mathbf{EDB}}}$ is indistinguishable from the one generated in the real experiment.

Simulating Search: For the conjunctive queries, the leakage $\mathcal{L}^{\textsc{Search}}_{\text{TWINSSE}_{\text{OXT}}}$ is exactly the same as $\mathcal{L}^{\textsc{Search}}_{\text{OXT}}$. Hence, we can write the following.

$$\mathcal{L}^{\textsc{Search}}_{\text{TWINSSE}_{\text{OXT}}}(\widehat{\mathbf{EDB}}, \{q_k\}_{k\in[|\mathcal{Q}|]}) = \mathcal{L}^{\textsc{Search}}_{\text{OXT}}(\mathbf{EDB}, \{q_k\}_{k\in[|\mathcal{Q}|]}).$$

By the simulation security guarantee of OXT, $\text{TWINSSE}_{\text{OXT}}$ secure against these leakages.

For disjunctive queries, we argue that the adversary $\mathcal{A}$ does not gain any information about the original disjunctive query except $|q|$. The distribution of $\mathbf{MDB}$ (encrypted to $\widehat{\mathbf{EDB}}$) is abstracted from $\mathbf{DB}$ through the meta-keywords. We resort to a more conservative analysis for this proof, as keywords do not have direct inference from meta-keywords, especially that is applicable over any database in general. The position of each w in an mkw is fixed according to the frequency of w, which is unique for a $\mathbf{DB}$. The lemmas below relate worst cases where an inference can be established between the query keywords and the corresponding meta-keywords without any additional knowledge of the plain database.

Lemma C.1, Lemma C.2, and Lemma C.3 relates the disjunctive q with $\mathsf{w}_i \in \Delta$ to the conjunctive q with $\mathsf{mkw}_i \in \widehat{\Delta}$.

Lemma C.1. *Consider two disjunctive queries of the same length t*

$$q_0 = \mathsf{w}_{1,q_0} \vee \mathsf{w}_{2,q_0} \vee \ldots \vee \mathsf{w}_{t,q_0},\ \mathsf{w}_{i,q_0}$$
$$q_1 = \mathsf{w}_{1,q_1} \vee \mathsf{w}_{2,q_1} \vee \ldots \vee \mathsf{w}_{t,q_1},\ \mathsf{w}_{i,q_1}$$

have the following expressions using mkw*s,*

$$q_0 = q_{0,\mathsf{mkw}} = \mathsf{mkw}_{1,q_0} \wedge \mathsf{mkw}_{2,q_0} \wedge \ldots \wedge \mathsf{mkw}_{t+1,q_0}$$
$$q_1 = q_{1,\mathsf{mkw}} = \mathsf{mkw}_{1,q_1} \wedge \mathsf{mkw}_{2,q_1} \wedge \ldots \wedge \mathsf{mkw}_{t+1,q_1}$$

both of length $t+1$*, and the* mkw*s are placed in the increasing order of the starting index of the* 0*s stretch in each* mkw*. If the* mkw*s at index* k *in* q_0 *and* q_1 *are the same, then* $\mathsf{w}_{k-1,q_0} = \mathsf{w}_{k-1,q_1}$ *and* $\mathsf{w}_{k,q_0} = \mathsf{w}_{k,q_1}$*.*

Proof. The proof of Lemma C.1 is given in Section C.3.1. □

Lemma C.2. *Consider two disjunctive queries* q_0 *and* q_1*, of the same length* t *have the* mkw *expressions as defined in Lemma C.1 - both of length* $t+1$*. If the* mkw*s at indices* k_0 *in* q_0*, and* k_1 *in* q_1 *are the same, then* $\mathbf{x}_{k_0-1,q_0} = \mathsf{w}_{k_1-1,q_1}$ *and* $\mathsf{w}_{k_0,q_0} = \mathsf{w}_{k_1,q_1}$*.*

Proof. The proof of Lemma C.2 is given in Section C.3.2. □

Lemma C.3. *Consider two disjunctive queries of different length* t_0 *and* t_1 *-*

$$q_0 = \mathsf{w}_{1,q_0} \vee \mathsf{w}_{2,q_0} \vee \ldots \vee \mathsf{w}_{t_0,q_0}, \;\; \mathsf{w}_{i,q_0} \in \Delta$$
$$q_1 = \mathsf{w}_{1,q_1} \vee \mathsf{w}_{2,q_1} \vee \ldots \vee \mathsf{w}_{t_1,q_1}, \;\; \mathsf{w}_{i,q_1} \in \Delta$$

have following expressions in the mkw*s*

$$q_0 = q_{0,\mathsf{mkw}} = \mathsf{mkw}_{1,q_0} \wedge \mathsf{mkw}_{2,q_0} \wedge \ldots \wedge \mathsf{mkw}_{t_0+1,q_0}$$
$$q_1 = q_{1,\mathsf{mkw}} = \mathsf{mkw}_{1,q_1} \wedge \mathsf{mkw}_{2,q_1} \wedge \ldots \wedge \mathsf{mkw}_{t_1+1,q_1}$$

which are of lengths t_0+1 *and* t_1+1 *respectively. If the* mkw*s at indices* k_0 *in* q_0*, and* k_1 *in* q_1 *are the same, then* $\mathsf{w}_{k_0-1,q_0} = \mathsf{w}_{k_1-1,q_1}$ *and* $\mathsf{w}_{k_0,q_0} = \mathsf{w}_{k_1,q_1}$*.*

Proof. The proof of Lemma C.3 is given in Section C.3.3. □

Recall that, the query transformation is executed by the client locally. The search is executed as a two-party protocol between the client and the server using the meta-keywords. The server learns $|q|$ trivially from q_{mkw} through of meta-keywords. From Lemma C.1, C.2, and C.3, an adversary can infer the position of the same ws in two queries of same length or different lengths if both queries have a *common* mkw in them.

However, the server can only infer if the least-frequent mkws in q_{mkw} are identical or not in mkw expressions of two qs from $\widehat{SP}$. The mkw expressions in each of the three lemmas require to place mkws in increasing order of the starting index of the 0's stretch. Whereas, the actual query expression for OXT has the least-frequent mkw first. No direct inference can be conjectured for the least-frequent mkw and the query expressions in the lemmas. Hence, an adversary $\mathcal{A}$ can not distinguish between the common meta-keyword and a distinct meta-keyword.

In the case, where the least-frequent of mkws is the first one in the query expression of the lemmas too, the first keyword is also the same for both ws. This is equivalent to the case of two conjunctive queries in keywords having the least-frequent w same.

Therefore, the leakage from $\text{TWINSSE}_{\text{Search}}$ can be limited to the OXT pattern leakages only, as expressed below.

$$\mathcal{L}^{\text{Search}}_{\text{TWINSSE}_{\text{OXT}}}(\widehat{\mathbf{EDB}}, \{q_k\}_{k\in[|\mathcal{Q}|]}) = \{\mathcal{L}'_{\text{OXT}}, |q_k|_{k\in[|\mathcal{Q}|]}\}.$$

Since, OXT is proven simulation secure, if follows from the simulation security guarantee that $\mathcal{A}$ no additional advantage over the real experiment.

C.3 Proofs of the Lemmas

We present the proofs of the lemmas presented earlier in this section. We follow the notations and conventions as used in the main body of the paper.

C.3.1 Proof of Lemma C.1

Proof. By construction, each meta-keyword mkw_i has the original keywords appearing in sorted order in the binary string representation (increasing order of frequency from left to right). Assume, the k'th meta-keyword mkw_k is same for both the queries q_0 and q_1. Without loss of generality, a meta-keyword in the basic $O(N^2)$ ($\text{TWINSSE}_{\text{Basic}}$) method can be formed as

$$\{b_1, b_2, \ldots, b_r, b_{r+1}, \ldots, b_s, b_{s+1}, \ldots, b_n\},\ b_i \in \{0, 1\},$$

where $1 \le r < s \le n$, and $b_i = 0$ for $r < i < s$.

To have an mkw of this form, q must have two keywords at indices r and s, and none in between (for q_0 and q_1 both). Since the mkws are constructed using ws in sorted order, if both queries q_0 and q_1 have the same r and same s (as one mkw is the same), the keywords w_r and w_s in both q_0 and q_1 are also the same. Hence, we have $\mathsf{w}_{k-1,q_0} = \mathsf{w}_{k-1,q_1}$ and $\mathsf{w}_{k,q_0} = \mathsf{w}_{k,q_1}$. □

C.3.2 Proof of Lemma C.2

Proof. We assume the common mkw of q_0 and q_1 can be expressed as

$$\{b_1, b_2, \ldots, b_r, b_{r+1}, \ldots, b_s, b_{s+1}, \ldots, b_n\},\ b_i \in \{0, 1\},$$

where $1 \le r < s \le n$, and $b_i = 0$ for $r < i < s$. The mkw appears at indices k_0 in q_0 and at k_1 in q_1. Since the indices of ws in the mkw strings are in sorted order (increasing frequency) and remains fixed for all mkws, the ws at index r and index s are the same for both q_0 and q_1. However, as the index of mkw is different in q_0 and q_1, the number of preceding ws before index r in q_0 and q_1 are different, equal to $k_0 - 2$ and $k_1 - 2$ respectively. Hence, for q_0, r is equal to $k_0 - 1$, and equal to $k_1 - 1$ in q_1. Following the above argument, we have $\mathsf{w}_{k_0-1,q_0} = \mathsf{w}_{k_1-1,q_1}$ and $\mathsf{w}_{k_0,q_0} = \mathsf{w}_{k_1,q_1}$. □

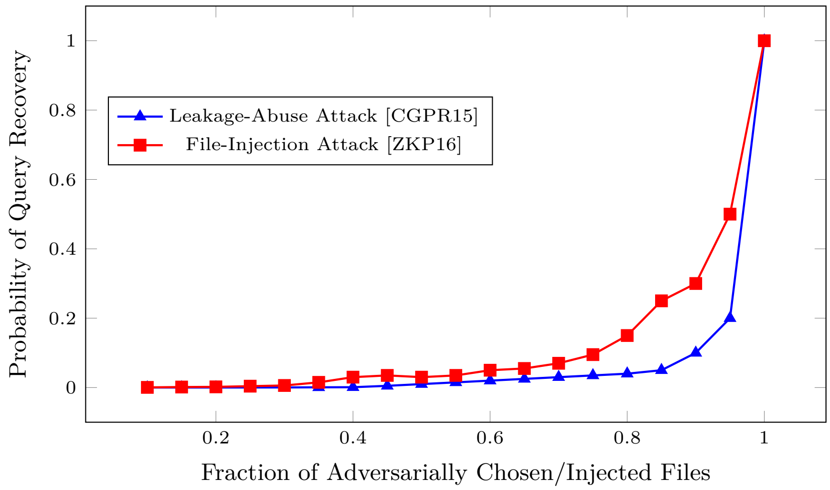

Figure 16: Leakage Analysis of $\mathrm{TWINSSE}_{\mathrm{OXT}}$: Two-Keyword Conjunctive Searches in the "Chosen Files" Setting.

C.3.3 Proof of Lemma C.3

Proof. The proof of Lemma C.3 follows from the proof of Lemma C.2. Essentially, Lemma C.3 is the extension of Lemma C.2 for two different lengths of queries. Intuitively, it can be established in the following way. Recall that in Proof C.3.2, r and s remains same in both q_0 and q_1, as in binary representation all mkws and qs have the same length n. However, the number of ws in q changes, and consequently, number of mkws change. Hence, the range of indices k_0 and k_1 are different for q_0 and q_1. This does not affect r and s which are positions of keywords (not related to number of keywords) in the binary representation of fixed length. Hence, the same argument from the proof of Lemma C.2 holds. □

D Cryptanalysis of $\mathrm{TWINSSE}_{\mathrm{OXT}}$

In this section, we present a leakage-based cryptanalysis of the $\mathrm{TWINSSE}_{\mathrm{OXT}}$ scheme via experiments over the Enron email corpus. Our experiments help substantiate that the leakages incurred by the conjunctive and disjunctive search protocols in $\mathrm{TWINSSE}_{\mathrm{OXT}}$ are benign in practice and are resistant to even the most powerful leakage-based cryptanalysis techniques in the SSE literature over real-world databases, such as *leakage-abuse attacks* [CGPR15] and *file-injection attacks* [ZKP16]. In particular, we experimentally establish the following claim:

Claim D.1 (Informal)**.** *The conjunctive and disjunctive search protocols in* $\mathrm{TWINSSE}_{\mathrm{OXT}}$ *resist leakage-abuse attacks [CGPR15] and file-injection attacks [ZKP16], and are benign in practice.*

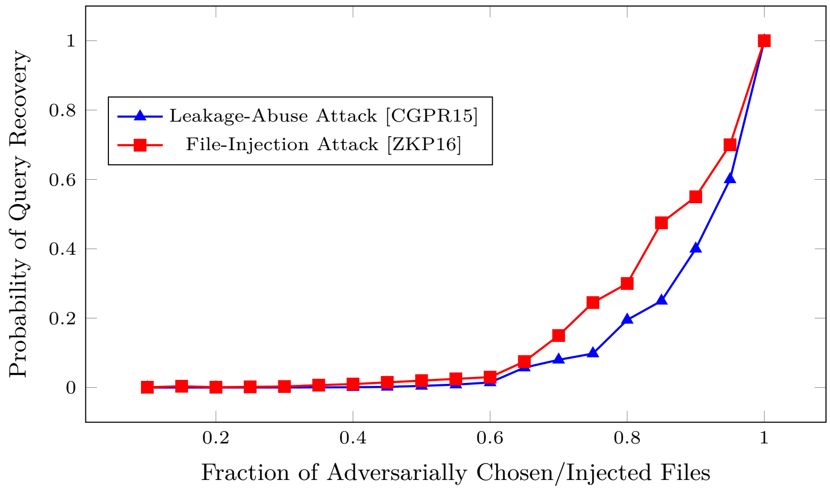

Figure 17: Leakage Analysis of $\text{TWINSSE}_{\text{OXT}}$: Two-Keyword Disjunctive Searches in the "Chosen Files" Setting.

We substantiate this claim by leakage cryptanalysis experiments targeting the conjunctive and disjunctive keyword search protocols in $\text{TWINSSE}_{\text{OXT}}$. We evaluate the probability that the adversary guesses correctly the keywords w_1 and w_2 underlying a two-conjunction query $q = (\mathsf{w}_1 \wedge \mathsf{w}_2)$ (resp., a two-disjunction query $q = (\mathsf{w}_1 \vee \mathsf{w}_2)$) by one of two well-known and extensively studied cryptanalysis methodologies in the SSE literature- the leakage-abuse attack of Cash *et al.* [CGPR15] and the *file-injection attack* of Zhang *et al.* [ZKP16]. The experiments were conducted over the same Enron email corpus as was used for the performance evaluation experiments in Section 6. The attacks operate in the *chosen/injected* file model (the strongest possible attack setting where a certain fraction of the files in the database are adversarially generated.) The corresponding results are plotted in Figure 16 and Figure 17 for conjunctive and disjunctive queries, respectively. Throughout, we use a bucket size $n' = 10$ (same as for the performance evaluation experiments in Section 6) for the disjunctive experiments.

We note here that fortifying implementations of $\text{TWINSSE}_{\text{OXT}}$ by using either ORAM-style data structures or adopting volume-hiding techniques such as padding/encrypted multi-maps [KM19, PPYY19] may be useful in masking leakage even further. However, even without such additional fortifications, $\text{TWINSSE}_{\text{OXT}}$ resists leakage-abuse and file-injection attacks in the strongest possible attacker setting, as demonstrated by the aforementioned experiments.

Volumetric Known-Data Attacks: We further evaluate $\text{TWINSSE}_{\text{OXT}}$ against the known-data volume analysis attacks presented by Blackstone et al. [BKM20], where we analyse $\text{TWINSSE}_{\text{OXT}}$ against the $\mathsf{SelVolAn}$ attack. These specific class of attacks exploits total volume pattern of the queries to recover the keywords, assuming that a fraction of the total data (quantified by "known data rate" δ) is available to the adversary. More precisely, it tries to associate the queried tags available to the server with known keywords. Since $\text{TWINSSE}_{\text{OXT}}$ produces noisy volumes due to the presence of spurious ids, the recovery

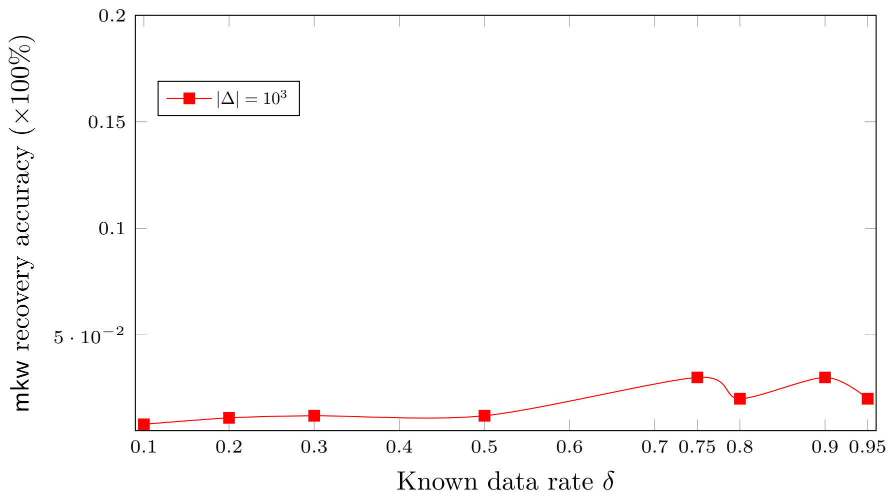

Figure 18: Leakage Analysis of $\text{TWINSSE}_{\text{OXT}}$ - SelVolAn attack. The amount of information available to the adversary (as a fraction of the total data) is varied and plotted on the x-axis. The volume pattern from meta-keyword queries were supplied as the leaked information.

rate is expected to be low in these evaluations. We use the LEAKER[12] framework to execute the SelVolAn attack.

We plot the attack results in Figure 18 which depicts the query recovery accuracy for the SelVolAn attack with varying known data rate δ. Note that, as stated earlier, the recovery rate for the SelVolAn attack is significantly low for our construction due the presence of spurious ids resulting in noisy volume pattern. Furthermore, the server receives the volume pattern information of the meta-keywords - not the actual query keywords. From the adversary's perspective, this requires additional auxiliary information to map recorded encrypted meta-keyword tags to probable meta-keywords pre-computed by the adversary from a partial set of the keyword universe (available as auxiliary data). Precise mapping would require full set of keywords as auxiliary information (indicating a high δ value, an extremely strong assumption) to form actual meta-keywords on the adversary's side.

Search and Access Pattern (SAP) based Attack: We also evaluate $\text{TWINSSE}_{\text{OXT}}$ against the state-of-the-art SAP attack by Oya et al. [OK21]. The SAP attack exploits search pattern (the sequence in which the queries are searched) and the access pattern (the particular address/elements that are "touched" by the server for each queried tag) and combines these two to recover the association among queried tags (recorded by the adversarial server) and the probable keywords available from auxiliary information. Again, in this case too, the server receives noisy access pattern which prevents the adversary to precisely map a recorded tag to a probable keyword (meta-keyword in $\text{TWINSSE}_{\text{OXT}}$) available as auxiliary data. We validate this through experimental evaluations with $\text{TWINSSE}_{\text{OXT}}$. We used the code available from the authors[13] for this evaluation.

[12] `https://github.com/encryptogroup/LEAKER`

[13] `https://github.com/simon-oya/USENIX21-sap-code`

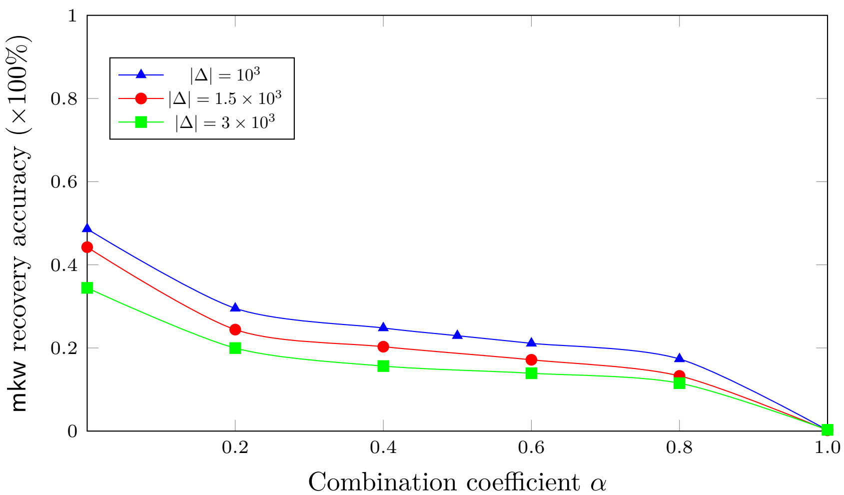

Figure 19: Leakage Analysis of $\text{TWINSSE}_{\text{OXT}}$ - SAP attack. Access pattern from meta-keyword queries and auxiliary information of keywords from Google Trends were provided as the input data.

The attack results are presented in Figure 19 which depicts the attack accuracy (as a fraction of correct "recorded tag"–"probable meta-keyword" associations to all such reconstructed associations) with varying combination coefficient (α). At a high-level, α represents the fraction of frequency information of the total auxiliary information available to the adversary used in the attack. In this case, the adversary recovers the association among queried tags and probable meta-keywords - not tags and actual keywords. Since reconstructing the actual meta-keywords requires the exact same keyword universe available to the adversary, it is unlikely that the adversary would be successful in associating the recovered meta-keywords with the subset of the keyword universe available to her (as auxiliary information).