Toward Full Accounting for Leakage Exploitation and Mitigation in Dynamic Encrypted Databases

Lei Xu, Anxin Zhou, Huayi Duan, Cong Wang, Qian Wang, Xiaohua Jia

Abstract—Encrypted database draws much attention as it provides privacy-protection services for sensitive data outsourced to a third party. Recent studies show that the security guarantee of encrypted databases are challenged by several leakage-abuse attacks on its search module, and corresponding countermeasures are also proposed. Most of these studies focus on static databases, yet the case for dynamic has not been well investigated. To fill this gap, in this paper, we focus on exploring privacy risks in dynamic encrypted databases and devising effective mitigation techniques. To begin with, we systematically study the exploitable information disclosed during the database querying process, and consider two types of attacks that can recover encrypted queries. The first active attack works by injecting encoded files and correlating file volume information. The second passive attack works by identifying queries' unique relational characteristics across updates, assuming certain background knowledge of plaintext databases. To mitigate these attacks, we propose a two-layer encrypted database hardening approach, which obfuscates both search indexes and files in a continuous way. As a result, the unique characteristics emerging after data updates can be eliminated constantly. We conduct a series of experiments to confirm the severity of our attacks and the effectiveness of our countermeasures.

Index Terms—Encrypted search, Cryptographic databases, Leakage abuse attack, Defenses.

1 Introduction

The last few years have witnessed the fast development of encrypted database systems [1], [2], [3], which provide practical and provable confidential service for hosting the outsourced data. For commercial and practicality concerns, many advanced property-preserving encryption schemes are employed, aiming to preserve various flavor of properties for encrypted data, particularly in the area of encrypted search [4], [5], [6], in which the client can retrieve the expected encrypted data without decryption. However, recent studies [7], [8], [9], [10], [11] show that most of these encrypted databases are vulnerable to potential attacks because the fundamental search module is not as secure as they claimed. Here the search module is usually implemented by various searchable encryption (SE) scheme which enables a client to perform search on encrypted data directly through an untrust server without decryption [4].

Leakage abuse attack (LAA) [12], [13], [14] is such an attack mentioned above. Empiric results show that, in LAAs, an adversary with certain prior knowledge can easily leverage the admitted leakage in SE to compromise encrypted databases. A typical example is count attack, proposed by Cash et al. [8], which recovers the query's content by matching the co-occurrence count of the query and the background information. Recently, according to the goal of the attack, most existing LAAs can be classified in two categories: 1) data reconstruction attack, which focuses on recovering the content of encrypted data (numeric data) [15], [16]; 2) query recovery attack that aims to mine the underlying content of the encrypted query [8], [13]. Along with the development of LAAs, many countermeasures are proposed, such as database padding to hide the query result's length [17], [18], [19], in order to sustain the long-term growth of encrypted databases.

While so many LAAs and countermeasures have been actively studied, most of them focus on static databases, few works are discussed on dynamics that supports addition and deletion operations [20], [21], [22]. As known, apart from access pattern, dynamic databases also leaks whether the update encrypted entries contain the keyword corresponding to previous queries, aka "update pattern" [23], but how much this term leaks has not been fully understood. In light of this view, in this paper, we will focus on the investigation of security issues in dynamic databases, especially the file storage database that supports keyword-based search. Specifically, we first revisit LAAs under different settings, including passive and active adversaries, and demonstrate that dynamic databases are more easily to be damaged than static ones. Then based on the understanding of these well-designed attacks, we devise comprehensive countermeasures to hardening the security of existing encrypted databases.

1.1 Motivation and Our Approaches

Following we overview the limitation of prior leakage exploitation on dynamic encrypted databases and illustrate our new approaches on both leakage exploitation and corresponding mitigations.

- *L. Xu is with the School of Science, Nanjing University of Science and Technology, Nanjing, 210094, China.E-mail: xuleicrypto@gmail.com*
- *A. Zhou, H. Duan, C. Wang, X, Jia are with the Department of Computer Science, City University of Hong Kong, Hong Kong SAR, China, and are also with the City University of Hong Kong Shenzhen Research Institute, Shenzhen 518057, China.E-mail: anxizhou-c@my.cityu.edu.hk, hduan2-c@my.cityu.edu.hk, congwang@cityu.edu.hk, csjia@cityu.edu.hk*
- *Q. Wang is with the School of Cyber Science and Engineer, Wuhan University, Wuhan, 430070, China.E-mail: qianwang@whu.edu.cn*

Manuscript received April 19, 2005; revised August 26, 2015.

Limitation of prior LAAs on DSSE and Mitigations. File inject attack is one of the few known attacks against DSSEs suggested by Cash et al. [8] and improved by Zhang et al. [24]. It allows the *active adversary* to inject designated files in the database and then identifies the query by observing the returned document combinations. For example, f_1, f_2, f_3 are three injected files and they have the only one common keyword w. Later, if a query q is responded with the result set containing f_1, f_2, f_3, the adversary can infer that q is w. To ensure the generated file can be injected into the database successfully, they provide a solution for adjusting file volume, because files too large or too small may be recognized as outliers.

An implicit assumption in this attack is that the file IDs in the encrypted index are exactly the plaintext ones. Otherwise, the adversary cannot identify which this injected file is. Zhang et al. argue that this assumption can be removed by assigning these files a distinctive volume and leveraging the volume leakage to identify them. However, how to implement it has not been well-investigated. For this concern, in this paper, we will instantiate this solution and show how to specify admissible volume size for injected files and generate these injected files. The key to our solution is to introduce the total query result volume to help identify the query rather than the individual file size only.

Revisiting LAA from the Update Pattern. In addition to the assumption of knowing file size, file injection attack also requires that the adversary can induce the data owner to inject his generated files into the database. This requirement is too harsh and not suitable for most real-world database applications. Is there any other effective way that can solve the above problem? Prior LAAs on static databases show that allowed leakage in SSE can significantly devastate the database's confidentiality, because some characteristics (e.g., query result length) are invariant in plaintext and encrypted databases. The adversary can easily recover the query by matching above characteristics. For example, the queries and keywords with the same unique result length can be matched. However, such an attack requires that the adversary (possibly) captures the full knowledge of the encrypted database, which is almost impossible in dynamic databases. Can this philosophy still be applied to attack DSSE?

We answer the above problem by proposing a new LAA from update pattern leakage, which reveals the database changes before and after performing update operations. We find that the above assumption of capturing full knowing database knowledge in LAAs can be naturally weakened in dynamic databases, because files in a dynamic database are uploaded one after another and can be distinguished by the query timestamps. Update pattern leakage makes it easy to know which part is the newly updated files. When the background knowledge of encrypted data for this part (not all) is known (trending topic), the passive adversary can efficiently recover the query's content by using prior LAAs. Attack details can be seen in our constructions.

Two-layer Padding Countermeasures. Three observations can be made from the above studies: 1) the leakages in dynamic encrypted databases are diverse, including characteristics derive from update pattern (passive adversary v.s index) and volume of the file (active adversary v.s file); 2) the leakage in dynamic encrypted databases exist in its whole life cycle, the exposure of any updates will leak the content of the query.

In light of the above observations, we devise a two-layer countermeasure from both index and file layers. Consistent with prior studies against LAAs in static's, padding is necessary for dynamic settings unless using costly ORAM. Our goal is to ensure that all generated leakage are hardly be distinguished after doing padding. However, achieving this goal requires executing padding over the whole keyword set and will incur heavy storage overhead. Inspired by Bost et al.'s cluster-based strategy [17], mostly it just requires that the adversary cannot distinguish the exact content of the query in an admissible subset. Thus we leverage cluster padding to boost storage efficiency. In addition, empirical results show that high-frequency padding will also enlarges this overhead, we answer this question by introducing batch padding as as to further reduce the padding overhead.

Our Contributions. We summarize the contributions of this paper as follows:

1) We develop two generic attack models against the dynamic encrypted database, which can break the confidentiality of the query by analyzing its revealed results. They awake our awareness of overlooked update pattern and figure out that the leakage in encrypted databases is two-folder, including both index information and file information.
2) We devise the first two-layer countermeasures to defend both active and passive adversaries for dynamic encrypted databases. More specifically, we doo padding on the index to hide the leakage produced in access/update pattern and do padding on the file to prevent the adversary from learning the encrypted database by volume pattern.
3) We conduct a series of experiments to evaluate the performance of our attacks and countermeasure. The experiment results show that our attacks present superior performances in recovering encrypted queries against dynamic databases. Even though 50% of background knowledge is known, the query recovery rate can still up to 86.5% in a real-world dataset. For countermeasures, experiment results show that we can introduce less than $1.2\times$ total overhead to reduce the recovery rate to below 0.5%.

2 Dynamic SSE and Leakage Profiles

We take the dynamic searchable encryption (DSSE) as an example to illustrate the attack and defense for dynamic encrypted database that leaks access pattern and volume in this work. Here we first review the definition of DSSE and relevant background knowledge.

Let $\mathcal{W} = \{w_1, \ldots, w_m\}$ be the keyword space, a database over $\mathcal{W}$ can be represent by a n-tuple keyword/identifier set $\mathrm{DB} = \{(id_i, W_i)\}_{i=1}^{n}$, where $W_i \subset \{0,1\}^*$ is the set of distinct keywords that appear in the file f_i and id_i denotes the identifier of f_i. For a given keyword $w \in \mathcal{W}$, we use $\mathrm{DB}(w) = \{id_i : w \in f_i\}$ to denote the identifier set of files that contains of keyword w. The size of the set $\mathrm{DB}(w)$ is written as $|\mathrm{DB}(w)|$, the size of the document f_i is denoted as $\#f_i = |W_i|$. More notations can be viewed in Table 1.

Dynamic Searchable Symmetric Encryption. A DSSE [20] consists of three protocols, `Setup`, `Update` and `Search`, between the client and the server. They proceed as follows: `Setup` protocol takes the security parameter λ and database DB as inputs and outputs the encrypted database EDB, a secret key K, and the client's internal state σ; `Search` protocol takes the EDB, secret key K, the internal state σ of client and the keyword $w \in \mathcal{W}$ as inputs, and outputs the query results R; `Update` protocol takes the encrypted database EDB, secret key K, newly added entries in, client's internal state σ and the operation op as input, and outputs an update encrypted database EDB$'$. We say that a DSSE scheme is correct if the `Search` protocol returns all matched results for the query with overwhelming probability, i.e., $\Pr[\texttt{Search}(\mathrm{EDB}, q_w) = \mathrm{DB}(w)] = 1 - \mathrm{neg}(\lambda)$, where q_w denotes the query about w and $\mathrm{neg}(\lambda)$ is the negligible function in security parameter λ.

Formalization of Leakage Profile and Security. Leakage profiles are defined to characterize the leakage of DSSE, which indicates the information revealed from each protocol. Let $\texttt{DSSE} = (\texttt{Setup}, \texttt{Search}, \texttt{Update})$ be a DSSE scheme, their leakage profiles can be parameterized as a triple of stateful functions $\mathcal{L}_{\mathrm{DSSE}} = (\mathcal{L}^{Stp}, \mathcal{L}^{Srch}, \mathcal{L}^{Updt})$, respectively. Before giving the details, we first review a notion $history$, which will be utilized to define leakage profiles later. Specifically, a t-query history H is a tuple $(\mathrm{DB}, \boldsymbol{w})$, where $\boldsymbol{w} = (w_1, \cdots, w_t)$ is a sequence of queried keywords. With this notion, typical leakage profiles of DSSE can be defined as follows:

- `Setup` protocol reveals the total number of keyword occurrences in database DB, i.e., $\mathcal{L}^{stp}(\mathrm{DB}, \sigma) = \sum_{w \in \mathcal{W}} |\mathrm{DB}(w)|$.
- `Search` protocol reveals the search result and the number of the matched result, i.e., $\mathcal{L}^{srch}(\mathrm{DB}, q_1, \ldots, q_t) = \{\mathrm{DB}(q_1), \cdots, \mathrm{DB}(q_t)\}$, also known as access pattern.
- `Update` protocol reveals the discrepancy of the database before and after update, i.e., $\mathcal{L}^{Updt}(\mathrm{DB}, in, \sigma) = \{\mathrm{old}(\mathrm{DB}(w)), \mathrm{new}(\mathrm{DB}(w))\}$.

We say that a DSSE is $\mathcal{L}$-semantic secure against any probabilistic polynomial time (PPT) adversary if it cannot distinguish the view of any two histories which have the same leakage profiles with a non-negligible probability. However, recent studies show that the existing encrypted search schemes are not secure as they claimed. The direct reason resulted in this dilemma are the *invariant characteristics* derived from the above leakages profiles, they may expose the connection between the query and keyword if a certain characteristic is unique. Here we introduce some of the invariant characteristics that will used later. The details are given as follows:

Result length pattern records the number of files (IDs) returned for each query. For given queries $q_1, \ldots, q_t$, the result length pattern is defined as

$$\mathcal{K}_{rlp}(\mathrm{DB}, q_1, \ldots, q_t) = \{|\mathrm{DB}(q_1)|, \cdots, |\mathrm{DB}(q_t)|\}$$

Query volume pattern records the size of files returned for each query. Let $\mathrm{V}(q_i) = \sum_{f \in \mathrm{DB}(q_i)} \#f$. For given queries $q_1, \ldots, q_t$, the query volume pattern is defined as

$$\mathcal{K}_{qvp}(\mathrm{DB}, q_1, \ldots, q_t) = \{\mathrm{V}(q_1), \cdots, \mathrm{V}(q_t)\}$$

TABLE 1
Notations

Notation	Description
w	keyword selected from the file
q	encrypted query of some keyword
W_i	the set of keywords appeared in f_i
id_i	the identifier of the i-th plaintext file
$[n]$	the integer set $\{1, \ldots, n\}$
$\|S\|$	the cardinality of the set S
$\#f$	the size of the file f
$\cap$	the intersection operation of two or more objects
$\mathcal{W}$	dictionary, aka, keyword space
DB	plaintext database
EDB	encrypted database
$\mathrm{K_R}$	relation characteristic
$\mathrm{K_I}$	independent characteristic
$A \preceq B$	all elements in collection in A exits in set B
$\|x\|_p$	Lp norms of vector x

File volume pattern records the size of each file stored in the storage server. For given files $f_1, f_2, \ldots$, the file size pattern is defined as

$$\mathcal{K}_{fvp}(\mathrm{DB}, f_1, f_2, \ldots) = \{\#f_1, \#f_2, \cdots\}$$

Query relation pattern $\mathcal{K}_{qrp}$ records the correlation of the queries, e.g., co-occurrence count, similar coefficient, For given queries $q_1, \ldots, q_t$, the relation pattern is defined as

$$\mathcal{K}_{qrp}(\mathrm{DB}, q_1, \ldots, q_2) = \{r_{i,j} : i, j \in [t]\}$$

Observe that, only the query with unique characteristic can be recovered. Thus, if the characteristic can distinguish each query more finely, then it can recover the content of the query more accurately. Without loss of generality, we classify these characteristics into two categories, *independent* and *relational* characteristics, where the former one describes the feature for one query and the later one describes the relation between queries like co-occurrence count.

3 Attack Model

3.1 Adversary, Knowledge and Target Goals

For achieving a comprehensive understanding of leakage abuse attack, we start with the three elements of the attack, the details are as follows:

Adversary. The adversary is the executor of an attack, which can always be classified into two categories, i.e., active adversary and passive adversary, depending on her capabilities. For an active adversary, she can be proactive in asking the client to perform the required addition, deletion, and search operations for some designated items and then observe the query response. While for the passive adversary, she could only monitor the client and the server's communication channel and learn their communication transcripts, including query tokens and corresponding responses.

Knowledge. Knowledge captures the background information that is exposed to the adversary, including plaintext files and underlying auxiliary data. The plaintext files are the collections of keywords selected from the keyword space, and the auxiliary data is the attribute information of the plaintext database, like keyword distribution, file volume, and search frequency. Generally, the amount of knowledge an adversary captured has a crucial impact on the attack effectiveness.

Target goal. The goal of the attack reports the target objective of the adversary. According to the target goal, existing attacks can be classified into two categories, i.e, query recovery attack and data reconstruction attack. Query recover attack is to mine the content of the encrypted requests issued by the clients, and data reconstruction attack is to reconstruct the encrypted database that supports rich queries. In this work, we mainly focus on the former one.

3.2 Threat Assumption and Attack Models

With an understanding of the three elements of leakage abuse attack, now we illustrate the threat assumptions of this work and give an overview of our attack models.

For the above goals, we first consider a generally persistent attack model in which the adversary can freely access the encrypted data and fetch updated information. Specifically, for passive adversaries, we assume that she knows the all plaintext index information of the encrypted database, and he can continually observe the query transcripts between the cloud and the server. Thus the discrepancy of the database in different stages can be observed. Moreover, we assume plaintext IDs are encrypted and never revealed to the adversary, i.e., she does not know the mapping between the encrypted IDs and plaintext IDs.

While for an active scenario, the adversary is assumed to have some additional abilities. For example, she can induce the client to inject some designated files with specific size into the database (also seen as [8]) adaptively, and she can learn the returned file volume of all queries by observing the communication volume. In particular, she can learn the discrepancy of the file volume between two update stages, which is the key contributing to identifying injected files. Both of the above adversaries are aimed at recovering the content of the encrypted queries.

3.3 Revisiting Leakage Abuse Attacks

Combing with the above description, we will review some classic leakage abuse attack models in this subsection.

IKK and Count Attack. As two state-of-art works in existing passive attacks, IKK attack [7] and count attack [8] are known for their efficiency and theoretical significance on understanding the influence of leakage revealed from the encrypted search problem. Specifically, the IKK attack first shows that the access pattern disclosure may bring devastating damage to the encrypted query while the co-occurrence probabilities of two keywords in the plaintext database are revealed. Cash et al.'s improve their work and point out to use more precise metrics, result length and co-occurrence count, to replace the co-occurrence probabilities. Besides, they also propose to use the result length to recover the queries with the unique result first, which significantly boosts the attack efficiency.

File Injection Attack. File injection is first proposed by Cash et al.'s and optimized by Zhang et al [24]. It's main idea is to encode each keyword with a specified binary index vector, and then observe the query result to rebuild the index vector to mine the content of the query. For example, let $w_1, \cdots, w_8$ be keywords and $q_1, \cdots, q_8$ be the corresponding queries. For each keyword, we first encode the keyword as a 3-bit index vector with its serial number $i-1$, i.e., $r_1 = (0,0,0), r_2 = (0,0,1), \cdots, r_8 = (1,1,1)$, then design the injected file as $f_i = \{w_j : r_{j,i} = 1\}$, where $i \in [\log 8]$. In a query, if a query returns only f_1 of these 3 files, the index vector is $(1,0,0)$, then we know that the content of this query is w_1.

4 Refining Leakage Abuse Attacks

We present two new attack models to refine prior ones. The first is an active attack named structure-based decoding attack. In particular, it leverages individual file volume leakage to encode each keyword, which can be deployed easily. The second attack is the relation-based inference attack, which exploits a more precise metrics, similarity, rather than co-occurrence count, to quantify the relationship between two queries. Therefore, the corresponding candidate keywords of one query can be further distinguished for matching. In the following, we give the technical details of the design.

4.1 Active Attack: Structure-based Decoding

In an active attack, a significant nature is that the adversary can guide the client to execute designated operations like addition, deletion and search. In this part, we mainly discuss how active attack works, in particular file injection attacks. As mentioned before, launching such an attack requires an additional condition that the adversary can identify when the injected files are returned, but how to implement it has not been well-investigated. Following, we will show how to implement it by exploiting individual file volume leakage. We call it structure-based decoding attack because it leverages the binary or volume structure of the query result.

Before that, we extend Zhang et al.'s file generation protocol by adding a constraint on file size so that each generated file has the designated size. Such a protocol is denoted as $\mathrm{F} \leftarrow \mathtt{FileDes}(\mathrm{C}, \boldsymbol{s})$, where $\mathrm{C} \subset \mathcal{W}$ is the set of keywords to be used and $\boldsymbol{s} = (s_1, \ldots, s_{\log|\mathrm{C}|})$ is the size of files $f_1, \ldots, f_{\log|\mathrm{C}|}$ in F. We perform padding on each file to satisfy the file size constraint. Specifically, we set $f_i \leftarrow f_i' \| f_i'$, where $f' \preceq \mathrm{W}_i$, $f_i \preceq \mathrm{W}$ and $\#f_i = s_i$. With the protocol, the refined file injection attack with admissible injected volume is given in Fig. 1. Assume that $\boldsymbol{q} = (q_1, \ldots, q_k)$ are k queries to be recovered, and the underlying keyword space is $\mathcal{W}$. Then the total file-injection attack consists of two steps, following are the technical details.

The first step is to generate the injected files, we implement it with $\mathtt{File-design}$ protocol. Let $\mathrm{C}_1, \cdots, \mathrm{C}_t$ be a partition of the keyword space $\mathcal{W}$, where each group has the same size $\ell = m/t$. For each $i \in [t/2]$, we first specify the size of files generated with keywords in $\mathrm{C}_{2i-1} \cup \mathrm{C}_{2i}$ as $r_i^j = \ell + j$ for $j = 1, \ldots, \log 2\ell$, then call $\mathtt{FileDes}(\mathrm{C}_{2i-1} \cup \mathrm{C}_{2i}, \{r_i^1, \ldots, r_i^{\log(2\ell)}\})$ protocol to generate a file set $\mathbf{G}_i$. Here $\mathbf{G}_i$ satisfies $|\mathbf{G}_i| = r_i^j$. After that, we continue generate another t files with keywords in $\mathrm{W}_i, \ldots, \mathrm{W}_t$, respectively. Specifically, for $i = 1, \ldots, t$, we put all keywords in W_i into file F_i first, and then use these keywords to do padding until the final size of F_i satisfies $|\mathrm{F}_i| = \max\{\hbar, \ell + \log(2\ell)\} + i$. Finally, when all the files are generated, we gradually inject them into the encrypted

Let $\mathcal{W} = \{w_i\}_{i=1}^m$ be the keyword space, let `FileDes` be the binary file design approach before. The goal is to recover the content of $\boldsymbol{q} = (q_1, \ldots, q_k)$.

$\texttt{File} - \texttt{design}(\mathcal{W}, \hbar)$ ▷ $\hbar$ be the largest file volume in EDB

1: Partition $\mathcal{W}$ into t parts $\{\mathrm{C}_i\}_{i=1}^t$ with equal size $\ell := m/t$
2: **for** each $i \in t/2$ **do**
3: Set $r_i^j \leftarrow \ell + j$ for $j = 1, \ldots, \log(2\ell)$
4: Set $\mathbf{G}_i \leftarrow \texttt{FileDes}(\mathrm{C}_{2i-1} \cup \mathrm{W}_{2i}, \{r_i^1, \ldots, r_i^{\log(2\ell)}\})$
5: **end for**
6: **for** each $i \in [t]$ **do**
7: Let $s' \leftarrow \max\{\hbar, \ell + \log(2\ell)\}$, $s_i \leftarrow s + i$
8: Set $f_i \leftarrow \mathrm{W}_i \| \mathrm{C}_i'$, where $\mathrm{W}_i' \preceq \mathrm{W}_i$ and $\# f_i = s_i$
9: **end for**
10: Inject files $\mathcal{F} = \{f_1, \ldots, f_t, \mathbf{G}_1, \ldots, \mathbf{G}_{t/2}\}$ into EDB.

$\texttt{Query} - \texttt{recover}(\boldsymbol{q}, \mathrm{EDB})$

1: Initialize an empty map $\mathrm{M} = (\mathrm{null}, \mathrm{null})$
2: **for** each query $q_i \in \boldsymbol{q}$ **do**
3: Record the received files volume $\boldsymbol{v}_i \leftarrow \{v_i^1, v_i^2, \ldots\}$
4: Set $b \leftarrow \{0\}^{\log(2\ell)}$ and find $k \leftarrow \{j : |f_j| \in \boldsymbol{v}_i\}$
5: **for** each $G_i^t \in \boldsymbol{G}_i$ **do**
6: Set $b_t \leftarrow 1$ if $\exists\ v_i^j \in \boldsymbol{v}_i\ s.t.\ |G_i^t| = v_i^j$
7: **end for**
8: Compute the hex value of binary number b and get d
9: Let w' is the $(d+1)$-th keyword in $\mathrm{C}_{2k-1} \cup \mathrm{C}_{2k}$
10: $\mathrm{M} \leftarrow \mathrm{M} \cup (q_j, w')$
11: **end for**
12: **return** the recovery query/keyword map M

Fig. 1. **Structure-based Decoding Attack**

Let $\mathcal{W} = \{w_i\}_{i=1}^m$ be the keyword space and M^* be the known query/keyword map. Let DB and EDB the plaintext and encrypted database, respectively. Assume $\mathbf{DB} = \{\mathrm{DB}^{(1)}, \mathrm{DB}^{(2)}, \cdots\}$ and $\mathbf{EDB} = (\mathrm{EDB}^{(1)}, \mathrm{EDB}^{(2)}, \ldots)$ are the plaintext and encrypted databases after i-th update, respectively. The goal is to recover $\boldsymbol{q} = (q_1, \ldots, q_k)$.

$\texttt{Candidate\_infer}(\mathrm{DB}, \mathrm{EDB}, q, \mathrm{M}^*)$

1: Initialize an empty set $\mathrm{S} = \emptyset$
2: Set $\mathrm{S} \leftarrow \{w : \mathrm{K_I}(\mathrm{EDB}, q) = \mathrm{K_I}(\mathrm{DB}, w)\}$
3: **if** $\mathrm{M}^* \neq (\mathrm{null}, \mathrm{null})$ **then**
4: **for** $(q^*, w^*) \in \mathrm{M}^*$ and $w \in \mathrm{S}$ **do**
5: $\mathrm{S} \leftarrow \mathrm{S}/\{w\}$ if $\mathrm{K_R}(\mathrm{EDB}, q^*, q) \neq \mathrm{K_R}(\mathrm{DB}, w^*, w)$
6: **end for**
7: **end if**
8: **return** S

$\texttt{Recover\_attack}(\mathbf{DB}, \mathbf{EDB}, \boldsymbol{q})$

1: Initialize an empty map M, a set S and a count $i = 1$
2: **for** $q \in \boldsymbol{q}$ **do**
3: **while** $\mathrm{S} \neq 1$ **do**
4: $\Delta^{(i)} \leftarrow \mathrm{DB}^{(i)} - \mathrm{DB}^{(i-1)}$
5: $\nabla^{(i)} \leftarrow \mathrm{EDB}^{(i)} - \mathrm{EDB}^{(i-1)}$
6: Set $\mathrm{S} \leftarrow \texttt{Candidate\_infer}(\Delta^{(i)}, \nabla^{(i)}, q)$
7: $i \leftarrow i + 1$
8: **end while**
9: **end for**
10: $\mathrm{M} \leftarrow \mathrm{M} \cup (q, w)$ if $|S| = 1$, here $w \in \mathrm{S}$
11: **return** the recovery query/keyword map M

Fig. 2. **Relation-based Inference Attack**

database at different time interval. We give this constraint in order to distinguish $\mathbf{G}_1, \ldots, \mathbf{G}_{t/2}$.

Now we show how to use this injected files to recover an observed query $q_i \in \boldsymbol{q}$. Assume that $\boldsymbol{v}_i = \{v_1, v_k, \ldots\}$ be the set of file volumes corresponding to q_i. Then, we find a k so that $\# f_k \in \boldsymbol{v}_i$. If such k is unique, we can determine that the underlying content of q_i is in $\mathrm{C}_{2k-1} \cup \mathrm{C}_{2k}$. As mentioned before, we are assumed to have the ability to observe the operation of the database, we can know which part of the returned (encrypted) files, as well as their individual volume. Based on this, we check which files in $\mathbf{G}_1$ are returned according to their distinctive volume. Given a $\log 2\ell$-bit string $b = \{0\}^{\log 2\ell}$, we set $b_t = 1$ if there exists a returned file has the same volume with G_k^t. By checking all the volume, we obtain a new string b and compute it value d. The $(d+1)$-th keyword in $\mathrm{C}_{2k-1} \cup \mathrm{C}_{2k}$ is the content we are looking for q_i.

4.2 Passive Attack: Relation-based Inference

As mentioned above, we first consider a baseline scenario that plaintext database DB and later update data sequence $\mathbf{DB} = (\mathrm{DB}^{(1)}, \mathrm{DB}^{(2)}, \ldots)$ are fully exposed to the adversary. Accordingly, let $\mathbf{EDB} = (\mathrm{EDB}^{(1)}, \mathrm{EDB}^{(2)}, \ldots)$ be the encrypted database after every update. A passive adversary (server) receives a sequence of query $\boldsymbol{q} = (q_1, \ldots, q_k)$ from the client, each q_i comes with a corresponding list of returned identifiers. Note that, the returned IDs may be encrypted. The goal is to recover the content of these queries. The core methodology we used to deal with this case is called relation-based inference. That is, exploiting the relational characteristics between the queries in updates to gradually determine the candidates of the target query, and matching the query and the candidate if there is only one candidate remained. In dynamic scenario, more unique characteristics can be revealed in the update procedure while compared with static scenario.

As depicted in Fig. 2, let $\mathrm{K_I}$ and $\mathrm{K_R}$ be the independent and relational characteristics derivation functions mentioned in Section 2. To recover the content of the query, the first step is to determine the possible candidate keyword set of the query. Specifically, for each query q, the adversary first computes its independent characteristic $\mathrm{K_I}(q)$, then set $\mathrm{S} \leftarrow \{w : \mathrm{K_I}(\mathrm{EDB}, q) = \mathrm{K_I}(\mathrm{DB}, w)\}$ as the set of keywords that may be the content of q. Then she further leverages known query/keyword pair and relational function to help refine the candidate set. For example, assume (q^*, w^*) is a pair of known query/keyword pair, for $w \in \mathrm{S}$, the adversary checks whether $\mathrm{K_R}(\mathrm{EDB}, q^*, q) \neq \mathrm{K_R}(\mathrm{DB}, w^*, w)$ holds, where $\mathrm{K_R}$ is the relational function. If yes, she remove w from S.

With the above approach, the adversary observes the status of candidate set of each query in updates. If there is only one keyword w remained in above S, the adversary outputs w as the content of q. Otherwise, the adversary continues to refine the candidate set by updating the dataset in `Candidate-Infer` function, i.e., setting $\Delta^{(i)} = \mathrm{DB}^{(i)} - \mathrm{DB}^{(i-1)}$ and $\nabla^{(i)} = \mathrm{EDB}^{(i)} - \mathrm{EDB}^{(i-1)}$. Since the leakage changes with the updated pattern, the probability of appearing unique characteristics in the dynamic database is more significant than that in statics. Our latter experiment evaluations will confirm this argument.

Remark on Attack with Partially Known Knowledge. We now discuss how to modify the above attack to deal with the case with partially known background knowledge. Note that, in such a case, there is a gap between the character-

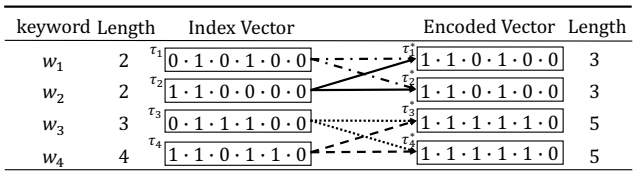


Fig. 3. An example of indistinguishable leakages

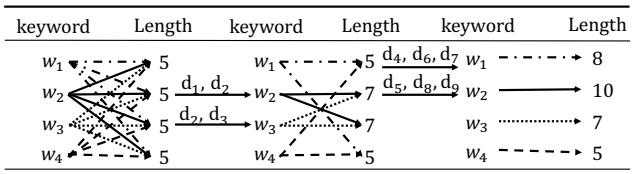


Fig. 4. An example of the leakage in update

istics learned from the encrypted query and background knowledge. It is hard for us to leverage the prior equality test approach to determine the candidate set of the query. In prior works [8], [13], researchers modify the approach to select the candidate set of q as $\mathrm{S} \leftarrow \{w : \mathrm{K_I}(\mathrm{Db}, w) \in [\mathrm{K_I}(\mathrm{EDB}, q) - \delta, \mathrm{K_I}(\mathrm{EDB}, q) + \delta]\}$, where $\mathrm{Db} \subset \mathrm{DB}$ is the known data of EDB and δ is the error term. The error term denotes the discrepancy between the characteristic derived from full background knowledge and that of partial background knowledge. For example, if $\mathrm{K_I}$ denotes the query result length leakage and Db is randomly selected from DB, then we have $\delta = \mathrm{K_I}(\mathrm{EDB}, q)(1 - |\mathrm{Db}|/|\mathrm{EDB}|)$.

We find that the above δ can be naturally narrow down in dynamic databases due to the update pattern leakage. As mentioned before, the data in dynamic data can be identified by the query timestamps. With the help of update pattern leakage, the adversary is easy to find the corresponding encrypted copy $\mathrm{Edb} \subset \mathrm{EDB}$ of the revealed data Db. Accordingly, the candidate set for query q can be narrow down to $\mathrm{S}' \leftarrow \{w : \mathrm{K_I}(\mathrm{Db}, w) \in [\mathrm{K_I}(\mathrm{Edb}, q) - \delta', \mathrm{K_I}(\mathrm{Edb}, q) + \delta']\}$, where $\delta = \mathrm{K_I}(\mathrm{Edb}, q)(1 - |\mathrm{Db}|/|\mathrm{Edb}|)$. It is clear that $\delta' \leq \delta$ and $\mathrm{S}' \subset \mathrm{S}$, namely, the adversary has a higher probability that recovers the query's content successfully.

Summary. In this section, we present two attacks from the perspective of passive and active adversaries, respectively. From Fig. 1 and Fig. 2, we can find that the key factors contributing to leakage abuse attack are those unique characteristics derived from the search results, e.g., length, volume, similarity, and fixed combination of search result. Thus, eliminating or hiding all these unique characteristics in the search results can be considered as a straightforward approach to mitigate the above or more LAAs.

5 New Security Notion and Discussion

5.1 Leakage Indistinguishability

As mentioned before, an adversary can successfully recover the content of the query if and only if only one candidate has the same characteristic with the query. To this end, to protect the content of query, we should guarantee that a large amount of candidates are involved in the candidate set. In other words, the characteristic regarding the query to the encrypted database should be computationally indistinguishable with that of their candidates in the plaintext database. Inspired by the notion of Local Differential Privacy [25] and ϵ-indistinguishable multiple views [26], we suggest the notion as follows.

Definition 1 (ϵ-indistinguishable leakages)**.** *For any PPT adversary, an encrypted search scheme is said to achieve ϵ-leakage indistinguishable security if and only if for any candidate keyword $w_i, w_j \in \mathcal{W}$, the probabilities of recognizing the content of query q through the leakage satisfies*

$$e^{-\epsilon} \leq \frac{\Pr(w_i \; is \; the \; content \; of \; query \; q)}{\Pr(w_j \; is \; the \; content \; of \; query \; q)} \leq e^{\epsilon} \qquad (1)$$

where q denotes an encrypted query randomly selected in the query set. We define $\frac{0}{0}$ to be 1. In the rest of this work, we use $q \triangleq w$ to denote that w may be the content of q.

Example 1. *Figure 3 illustrates an example of access pattern leakage indistinguishability. Let $\tau_1 = (0, 1, 0, 1, 0, 0)$ and $\tau_2 = (0, 1, 0, 1, 0, 0)$ be the index vector of keyword w_1 and w_2, let $\tau_1^* = \tau_2^* = (1, 1, 0, 1, 0, 0)$ be their result vector (aka access pattern leakage) derived from the query result over database equipped with an encode scheme. We say that this encrypted search scheme is indistinguishable secure because access pattern leakage $(1, 1, 0, 1, 0, 0)$ here has the indistinguishable candidates w_1 and w_2, which cannot be identified by the adversary.*

Recall that, in the statics setting, the client can build index vectors for all the keywords first, and then obfuscate these index information (perform padding) to meet the demands of above indistinguishability. However, it is not trivial to realize such a ϵ-indistinguishable security encrypted search scheme that supports data dynamics. Because the update pattern may also leak the information of the raw data, unless all keywords in the keyword universal appear together every time they are updated. Otherwise, the adversaries can exploit the discrepancy generated during the update procedure to infer the content of the query.

Example 2. *Figure 4 is employed to show that why previous countermeasures cannot directly apply to dynamic databases. We take the real-time result query lengths as an example to explain this issue. As seen in Fig. 4, keywords $w_1, \ldots, w_4$ appear with the same length of IDs 5 in the initial stage, after the first update, IDs of w_2 and w_3 are involved and both of them have the IDs length 2. In the second round, IDs of w_1 and w_2 are uploaded, and each of them adds 3 new IDs. After two rounds of update, however, we can see that all of the keywords can be uniquely identified by their distinctive length.*

The above discussion does not indicate that there is no way to provide full protection for dynamic encrypted databases. Padding all the keywords in the keyword universal, including those absented, to the maximum in all update process may be a straightforward approach (The extreme case that only one or few keywords absented seems to be minuscule). Nevertheless, the high costs of maintaining up to above security strength doom that it cannot be applied to the real scenarios. To take one step back, a relaxed version of Definition 1, locally indistinguishable leakages, is required, which also aims at balancing the security and overhead. Following are the details:

Definition 2 (Locally indistinguishable leakages)**.** *An encrypted search scheme is said to be (α, ϵ)-locally indistinguishable against any passive or active PPT adversary if for any query q, there must exist a subset $W \subset \mathcal{W}$ such that for any w_i, w_j in W*

$$e^{-\epsilon} \cdot \Pr(q \triangleq w_j) \leq \Pr(q \triangleq w_i) \leq e^{\epsilon} \cdot \Pr(q \triangleq w_j) \quad (2)$$

holds with overwhelming probability, where $|W| > \alpha$.

Compare with the Definition 1, Definition 2 only requires that the candidates in a designated range are indistinguishable. Under this setting, the practicer can first partition the keyword space into several clusters via the keyword frequency [27], and then run a locally indistinguishable leakages secure protocol upon each update. Empirical results show that doing so can significantly reduce the storage overhead. The above frequency indicates the appearance trends of keywords in data dynamics. The following theorem establishes an upper bound on security guarantees of applying the above local indistinguishability to the encrypted search scheme that supports data dynamics.

Theorem 1 (Sequential Composition)**.** *For a file sequence $\mathbb{F} = \{\mathrm{F}_1, \ldots, \mathrm{F}_k\}$, let $\mathrm{P}_1, \ldots, \mathrm{P}_k$ be k leakage-hiding protocols for encrypted search scheme, where each of them is (α, ϵ_i)-locally indistinguishable secure on the same keyword cluster $W \subset \mathcal{W}$ and $i \in [k]$. Then we have that $\mathrm{P}(\mathrm{F}_1), \ldots, \mathrm{P}_k(\mathrm{F}_k)$ satisfy $(\alpha, \max_{i\in[k]}(\epsilon_i))$-locally indistinguishable security.*

Proof. Let $\mathbb{F} = \{\mathrm{F}_1, \cdots, \mathrm{F}_k\}$ be the total data collection, let $\mathrm{P}(\mathbb{F}) = \{\mathrm{P}_1(\mathrm{F}_1), \cdots, \mathrm{P}_k(\mathrm{F}_k)\}$ be the corresponding image of all P_i on F_i. Let $\Pr[\mathrm{P}(\mathbb{F}) : q \triangleq w_i]$ be the probability that LAA outputs w as the content of q. Since the LAA against dynamic encrypted database is launched along with the updates, then we have $\Pr[\mathrm{P}(\mathbb{F}) : q \triangleq w_i] \leq \max_i \Pr[\mathrm{P}(\mathrm{F}_i) : q \triangleq w_i]$. Combining with that fact each P_i is (α, ϵ_i) locally-indistinguishable secure knowledge-hiding protocols on $\widetilde{\mathcal{W}} \subset \mathcal{W}$ it is clear that

$$\begin{aligned}
\frac{\Pr[\mathrm{P}(\mathbb{F}) : q \triangleq w_i]}{\Pr[\mathrm{P}(\mathbb{F}) : q \triangleq w_j]} &= \frac{\max_{t\in[k]}\{\Pr[\mathrm{P}_t(\mathrm{F}_t) : q \triangleq w_i]\}}{\max_{t\in[k]}\{\Pr[\mathrm{P}_t(\mathrm{F}_t) : q \triangleq w_i]\}} \\
&\leq \frac{e^{\epsilon}\max_{t\in[k]}\{e^{\epsilon_i-\epsilon}\Pr[\mathrm{P}_t(\mathrm{F}_t) : q \triangleq w_j]\}}{\max_{t\in[k]}\{\Pr[\mathrm{P}_t(\mathrm{F}_t) : q \triangleq w_i]\}} \\
&\leq \frac{e^{\epsilon}\max_{t\in[k]}\{\Pr[\mathrm{P}_t(\mathrm{F}_t) : q \triangleq w_j]\}}{\max_{t\in[k]}\{\Pr[\mathrm{P}_t(\mathrm{F}_t) : q \triangleq w_i]\}} \leq e^{\epsilon}
\end{aligned}$$

where $\epsilon = \max\{\epsilon_1, \cdots, \epsilon_k\}$. In the same way, we also have that $\Pr[\mathrm{P}(\mathbb{F}) : q \triangleq w_i] \geq e^{-\epsilon}\Pr[\mathrm{P}(\mathbb{F}) : q \triangleq w_j]$. Thus the property of locally indistinguishability is guaranteed. This completes the proof. □

5.2 Basic cluster-based Encoding Protocol

We describe a cluster-based encoding protocol here to implement the above leakages indistinguishable property. As shown in Fig. 5, the encoding protocol consists of two sub-protocols `EncodeFile` and `EncodeIndex`, where `EncodeFile` aims at protecting the file volume and `EncodeIndex` prefers to hide the information derived from index, aka access pattern.

Let $\mathrm{M} \in \{0,1\}^{t\times l}$ be the index matrix, where $\mathrm{M}[i,j] = 1$ indicates w_i appears in file f_j and otherwise not. Let

Let $\mathcal{W}$ be the keyword space, let $\mathcal{F} = \{f_1, \cdots, f_l\}$ be a set of files and $\mathrm{ID} = \{\mathrm{id}_1, \cdots, \mathrm{id}_l\}$ be their corresponding identifiers, let α and β be security parameters for index encoding and file encoding, respectively. The `EncodeIndex` and `EncodeFile` protocols work as follows:

`EncodeIndex`$(\mathcal{F}, \alpha)$

1: Group $\mathcal{W}$ into k clusters $\mathrm{C}_1, \cdots, \mathrm{C}_k$, s.t. all $|\mathrm{C}| \geq \alpha$
2: **for** each cluster C and $\mathcal{F}$ **do**
3: Set $J \leftarrow \{\mathrm{id}_j : \mathrm{C} \cap W_j \neq \emptyset\}$ be files containing w in C
4: Build the index matrix $\mathrm{M}[i,j] \leftarrow$ `BuiltInd`(C, J)
5: Let $s \leftarrow \max_{w_i\in\mathrm{C}} \|\mathrm{M}[i,:]\|_{p=1}$
6: **for** each $w_i \in W$ **do**
7: $q \leftarrow (s - \|\mathrm{M}[i,:]\|_{p=1})/(|J| - \|\mathrm{M}[i,:]\|_{p=1}))$
8: If $\mathrm{M}[i,j] = 0$, set $\mathrm{M}[i,j] \leftarrow 1$ with probability q
9: **end for**
10: **end for**
11: Set $\mathrm{DS} \leftarrow \{(w_i, \mathrm{id}_j) : \mathrm{M}[i,j] = 1\}$
12: **return** the indexes DS

`EncodeFile`$(\mathcal{F}, \beta)$

1: Group $\mathcal{F}$ into t clusters $\mathrm{F}_1, \cdots, \mathrm{F}_t$ s.t. all $|\mathrm{F}_i| \geq \beta$
2: Let $v_i = \max_{f\in\mathrm{F}_i} \#f$ be the padding size for the i-th cluster
3: **for** each i and $f \in \mathrm{F}_i$ **do**
4: Set $W_f \leftarrow \{w : w$ appears in $f\}$
5: Compute $s \leftarrow v - \#f$
6: Let $f' \leftarrow \{w : w \in W_f\}$ be the collection such that $\#\mathrm{S} = s$
7: Set $f^+ \leftarrow f\|f'$ be the encoded file
8: **end for**
9: **return** the files set $\mathcal{F}' = \{f^+\}_{f\in\mathcal{F}}$

Fig. 5. **Basic Cluster-based Encoding Protocol**

`BuiltInd`(W, F) be the function that takes as input a keyword set W and a file set F and output the index matrix. The following are the details of our encoding protocols.

For `EncodeIndex` protocol, it is designed to protect the aforementioned invariant characteristics derived from the index, so as to break the connection between the plaintext keyword and encrypted query. We fundamentally instantiate this goal by perturbing the plaintext index based on keyword clusters. As depicted in Fig. 5, let $\mathcal{W}$ be the keyword space and ID be the set of identifiers, then `EncodeIndex` protocol proceeds as follows. The client first groups $\mathcal{W}$ into a set of non-intersection subsets $\{\mathrm{C}_1, \cdots, \mathrm{C}_k\}$, where $|C_i| \geq \alpha$ for $i = 1, \cdots, k$. For each cluster W_i, she defines J as the identifiers of files containing keywords in W_i and set $\mathrm{M} =$ `BuiltInd`(C_i, J) as the index matrix of C_i and J. Let $s = \max_{w_i\in W} \|\mathrm{M}[i,:]\|_{p=1}$ be the maximum row hamming distance in the index matrix, which indicates the largest length of ID list of the keyword in the cluster. After that, we show how to perturb the index of the files over this keyword cluster. For each index row of the keyword, the client converts then entry $\mathrm{M}_{\mathrm{ind}}[i,j]$ from "0" to "1" with probability q if $\mathrm{M}[i,j] = 0$, where $q = (s - \|\mathrm{M}[i,:]\|_{p=1})/(|J| - \|\mathrm{M}[i,:]\|_{p=1})$. Finally, the client transforms the new index matrix to the keyword-identifier pairs for future encrypting.

Observe that, by perturbing the rows in such a way, these rows' hamming weights all fluctuate around s. Accordingly, all the corresponding queries will have a similar result length. Thus the result length is hidden. Naturally, all other characteristics will also be obfuscated since they

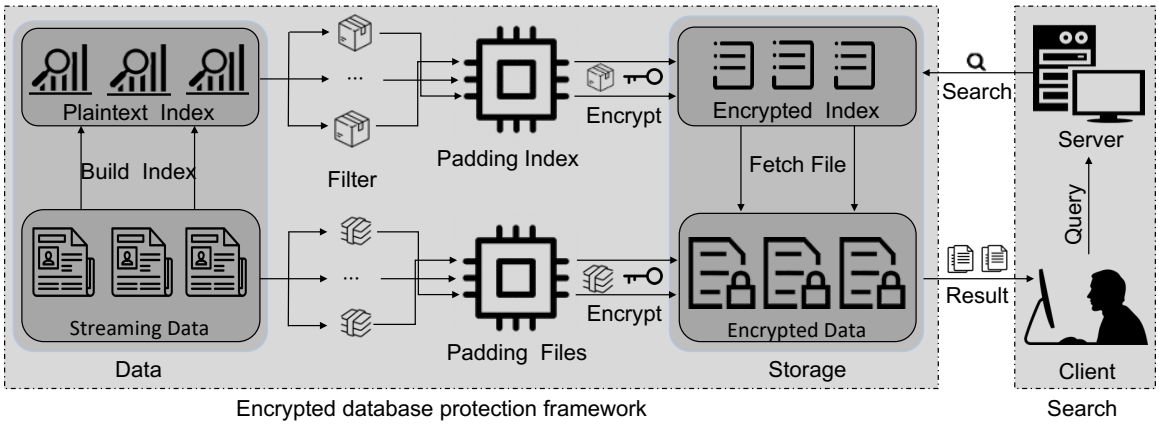


Fig. 6. Two-layer Privacy-preserving Framework

are extracted from these perturbed index vectors (i.e., access pattern). We would like to clarify that we even encode the index vector of the keyword that does not appear in the file because of the security requirements. We write such a non-appear keyword's index vector as $\mathbf{0}$.

For `EncodeFile` protocol, as the confidentiality of files is guaranteed by the cryptographic scheme, we only need to protect their volume. Intuitively, a feasible approach is to pad their volume into the same one. Without loss of generality, we assume that all keywords in $\mathcal{W}$ have the same bit length. Let β be the padding parameter, which determines security strength and the number of required padding keywords for each file. Similar to `EncodeIndex` protocol, we also encode the files by clusters. For example, for a given file set $\mathcal{F} = \{f_1, \cdots, f_l\}$, the client first groups these files into t clusters $\mathrm{F}_1, \cdots, \mathrm{F}_t$. Then for each cluster, she computes $v_i = \max_{f \in \mathrm{F}} \#f$ as the final padding size. Then for file $f_i \in \mathrm{F}_i$, she randomly selects $v_i - \#f_i$ keywords in the set f_i and appends them after the file. After that, the files in the same cluster will keep the same volume, and the adversary cannot exploit the volume information to identify each file. Therefore, the active attack can be denied. Note that, selecting keywords from W_f is to maintain the original index information.

The `EncodeFile` and `EncodeIndex` protocols are designed to mitigate aforementioned privacy risks in DSSE. We provide in Section 6 an detail construction about how it works to hardening prior DSSE schemes, together with their privacy analysis.

6 Two-layer Countermeasures

In this section, we present a two-layer privacy-preserving encrypted database protection mechanism. More specifically, we focus on mitigating the privacy risks from file volume and leakage derived from access pattern simultaneously against both active and passive adversaries.

6.1 Design Intuition

As mentioned before, the passive attacks work relying on the existence of unique knowledge (characteristics) derived from leakage profiles, and active attacks leverage the specific index structure and volume discrepancy to uniquely identify the content of the query. Ground on these results, a natural idea to defeat the above attacks is to eliminate or hide the above unique characteristics.

First, obfuscating the size of the files is necessary, no matter for the passive attack or the active attack. On the one hand, a significant assumption of file-injection attack or our structure-based attack is that the adversary can exploit the volume information to recognize which encrypted files are they inject, so we need to hide the volume by obfuscating every file size. On the other hand, for a passive adversary, once the mapping between the encrypted file IDs and plaintext IDs are determined, she can easily recover the content of the query by comparing the search results. Thus, hiding the file size is vital. For the question of how to obfuscate file size, a simple solution is to inject the existing keywords in that file as described in Fig. 5. Leveraging this strategy can also guarantee that $|f_i^*| = |f_i|$, ensuring the search accuracy from the file perspective, where f_i^* is the encoded file for f_i.

Second, it's also necessary to hide the knowledge derived from the indexes, because the passive (inference) attack can exploit the similarity of these knowledge to infer the content of encrypted queries. Generally, we can achieve this goal by inserting file IDs that are randomly generated or selected from the existed ones. As mentioned in Section 5, we consider to use the existing file IDs because of these two benefits: (1) mitigate the additional storage overhead by avoiding introducing newly bogus files; (2) better obfuscate the access pattern, hiding the similarity of different queries. Furthermore, motivated by Bost et al's clustering padding strategy to reduce storage overhead, we also prefer to employ this strategy to reduce the storage overhead.

6.2 Two-layer Dynamic Encrypted Search Scheme

Now we begin to illustrate our two-layer privacy-preserving approach to secure the encrypted search scheme. As depicted in Fig. 6, the total protection mechanism includes three modules, `Setup`, `Search`, `Update`. Following we will present the details:

Setup: Let $\mathrm{M} = (\mathrm{null}, \mathrm{null})$ be an empty map and IND be an empty set. To build a secure encrypted search system, the client takes the security parameter λ and the file set $\mathbf{F}_0$ as inputs, and runs the `Setup` protocol to output the encrypted file set M, the encrypted index IND with the secret key pair $(\mathtt{ek}, \mathtt{sk})$. Specifically, for a given file set F_0,

Let $\texttt{DSSE} = (\texttt{Setup}, \texttt{Search}, \texttt{Update})$ be a dynamic searchable encryption scheme, let $\texttt{SE} = (\texttt{Encrypt}, \texttt{Decrypt})$ be a secure encryption scheme, let $\mathbb{F} = \mathbf{F}_0, \mathbf{F}_1, \mathbf{F}_2, \cdots$ be the file streaming, where $\mathbf{F}_0$ denotes the files in initial stage, let $\texttt{Encode} = (\texttt{EncodeIndex}, \texttt{EncodeFile})$ be the index and file encoding algorithm. Following are our designs:

$\underline{\texttt{Setup}(\lambda, \mathbf{F}_0, \alpha)}$

Initialize an empty map $\mathrm{M} = \{\mathrm{null}, \mathrm{null}\}$
Encrypt the plaintext files $\mathbf{F}'_0 \leftarrow \texttt{EncodeFile}(\mathbf{F}_0, \beta)$
for $\mathrm{F}'_{0,i} \in \mathbf{F}'_0$ **do**
    Encrypt the files $(\mathrm{E}_{0,i}, \texttt{ek}) \leftarrow \texttt{SE.Encrypt}(\lambda, \mathrm{F}'_{0,i})$
    Store the encrypted file in the map $\mathrm{M}[\mathrm{id}_{0,i}] \leftarrow \mathrm{E}_{0,i}$
end for
Encode the index $\mathrm{DS}'_0 \leftarrow \texttt{EncodeIndex}(\mathbf{F}_0, \alpha)$
for $w \in \mathcal{W}, \mathrm{id} \in \mathrm{DS}'_0$ **do**
    $(\mathrm{IND}, \texttt{sk}) \leftarrow \texttt{DSSE.Setup}(\lambda, \mathrm{DS}'_0)$
end for
Store $(\mathrm{IND}, \mathrm{M})$ at the server and keep the keys $(\texttt{sk}, \texttt{ek})$

$\underline{\texttt{Search}(\texttt{sk}, w, \mathrm{IND}, \mathrm{M})}$

Initialize an empty set R
Run $\mathrm{Id} \leftarrow \texttt{DSSE.Search}(\texttt{ek}, \mathrm{IND})$ to fetch the file IDs
for $\mathrm{id} \in \mathrm{Id}$ **do**
    Invoke $\mathrm{E} \leftarrow \mathrm{M}[\mathrm{id}]$ to fetch the encrypted file
    Set $\mathrm{R} \leftarrow \mathrm{R} \cup \mathrm{E}$
end for
Send R to the client
Decrypt encrypted files $\mathrm{F} \leftarrow \texttt{SE.Decrypt}(\mathrm{R})$

$\underline{\texttt{Update}(\texttt{sk}, \texttt{ek}, \mathbf{F}_j, \mathrm{IND}, \mathrm{M})}$

Encrypt the plaintext files $\mathbf{F}'_j \leftarrow \texttt{EncodeFile}(\mathbf{F}_j, \beta)$
for $\mathrm{F}'_{j,i} \in \mathbf{F}'_j$ **do**
    Encrypt the files $(\mathrm{E}_{j,i}, \texttt{ek}) \leftarrow \texttt{SE.Encrypt}(\lambda, \mathrm{F}'_{j,i})$
    Store the encrypted file in the map $\mathrm{M}[\mathrm{id}_{j,i}] \leftarrow \mathrm{E}_{j,i}$
end for
Encode the index $\mathrm{DS}'_j \leftarrow \texttt{EncodeIndex}(\mathbf{F}_j, \alpha)$
for $w \in \mathcal{W}, \mathrm{id} \in \mathrm{DS}'_j$ **do**
    Run $(\mathrm{IND}, \texttt{sk}) \leftarrow \texttt{DSSE.Update}(\texttt{sk}, \mathrm{DS}'_j, \mathrm{IND})$
end for
Update the map IND and M.

Fig. 7. **Two-layer Database Hardening Approach**

the client first invokes `Encode.File` protocol to encode the files and obtains F'_0. Then she uses `SE.Encrypt` protocol to encrypt each file $\mathrm{F}'_{0,i} \in \mathbf{F}'_0$ and gets the encrypted file $\mathbf{E}_{0,i}$ and encrypted key `ek`. After all the files are encrypted, she stores the $\mathbf{E}$ at the server and builds up the position map M, where $\mathrm{M}[\mathrm{id}_{0,i}] = \mathrm{E}_{0,i}$. To enable search, the client continues to build the inverted index $\mathrm{DS}_0 = \{(w, \mathrm{id})\}_{w \in \mathcal{W}}$ for files $\mathbf{F}_0$, and deploys `Encode.Index` protocol to encode the index information. After that, the client applies the `DSSE.Setup` protocol to the index DS_0 and gets the encrypted index IND and search key `sk`. Finally, the client uploads the encrypted index to the server and keeps the secret key $(\texttt{ek}, \texttt{sk})$.

Search: When the client wants to search the encrypted files with some keyword w, she takes the secret key $(\texttt{ek}, \texttt{sk})$ as inputs, and runs the `Search` protocol to get the search results. Specifically, the client first runs the `DSSE.Search` protocol with her search key and the keyword w to compute the search token `st`, then she submits the query request to the sever with token `st`, the server uses token to find the matched files's position Id and returns the encrypted files R to the client. Once the client gets the encrypted files, she decrypts them to get the plaintext through `SE.Decrypt` protocol. Note that, as the original search index has been obfuscated in the `Setup` stage, some additional files may be returned even they do not contain the keyword w, resulting in communication overhead.

Update: Our encrypted database also supports data dynamics while maintaining search functionality. When new files appear and need to be uploaded, the client will take these newly files $\mathbf{F}_j$ and the key pair $(\texttt{ek}, \texttt{sk})$ as inputs and calls the `Update` protocol to insert them into the encrypted database. As seen in Fig. 7, `Update` protocol proceeds as follows: similar to `Setup` stage, for newly coming file set $\mathbf{F}_j$, the client first encodes the file set with `EncodeFile` protocol and then runs `SE.Encrypt` protocol to encrypt them. Then she stores these encrypted files at the server and renews the position map $\mathrm{M}[\mathrm{id}_{j,i}] = \mathrm{E}_{j,i}$, where $\mathrm{id}_{j,i}$ can be viewed as the identifier (pointer) of the encrypted file $\mathrm{E}_{j,i}$. Following, the client continues to build the index DS_j for $\mathbf{F}_j$ and use `sk` to encrypt the index. It is worth to note that, we need to follow the constraints mentioned in Section 5, i.e., encode the files and index before encrypting them.

6.3 Security Analysis

Given the following two theorems, we show the privacy guarantees brought by `EncodeIndex` and `EncodeFile` protocols. The first one is to protect the file volume and the other one is to protect information derived from access pattern.

Theorem 2. *Let $\boldsymbol{\beta} = (\beta_{\min}, \beta_{\max})$ be the privacy parameter for the* `EncodeFile` *protocl, then for any keywords w_i and w_j whose maximum Hamming distance is d, for any encrypted query q, the probability of recovering the content q under a k-bits structure-based decoding attack satisfies*

$$e^{-\epsilon} \cdot \Pr[q \triangleq w_j] < \Pr[q \triangleq w_i] < e^{\epsilon} \cdot \Pr[q \triangleq w_j] \qquad (3)$$

where $\epsilon = d \log(\beta_{max}/\beta_{min})$.

Proof. Assume that the keyword w can be represented by a well-designed unique k-bits index vector $\tau_w \in \{0,1\}^k$ based on a set of designated file IDs. In a structure-based decoding attack, as the adversary can identify whether an encrypted file is in above set through its file size, so each w can also be denoted as a unique volume vector $\boldsymbol{v}_w$. Knowing that the `EncodeFile` protocol encodes all files with parameter β in each stage, so there are at least β files have the same as the designated ID. Hence, the probability the adversary can view w as the content of q is

$$\Pr[\tau^*_q[1] = \tau_w[1] \wedge \cdots \wedge \tau^*_q[k] = \tau_w[k]] = \frac{1}{|\rho_1| \times \cdots \times |\rho_k|}$$

where $\rho_i = 1/|V_i|$ if $\tau^*_q[i] = 1$ and $\rho_i = 1 - 1/|V_i|$ when $\tau^*_q[i] = 0$. Here V_i is the candidate of IDs which file size is $\boldsymbol{v}_w[i]$ and $\beta_{min} \le |V_i| \le \beta_{max}$. The for any w_i and w_j which have at most d different bits, we have

$$\begin{aligned}\frac{\Pr[q \triangleq w_i]}{\Pr[q \triangleq w_j]} &= \frac{\Pr[\tau^*_q[1] = \tau_{w_j}[1] \wedge \cdots \wedge \tau^*_q[k] = \tau_{w_j}[k]}{\Pr[\tau^*_q[1] = \tau_{w_i}[1] \wedge \cdots \wedge \tau^*_q[k] = \tau_{w_i}[k]} \\ &= \frac{|\rho_{i,1}| \times \cdots \times |\rho_{i,k}|}{|\rho_{j,1}| \times \cdots \times |\rho_{j,k}|} < e^{\epsilon}\end{aligned}$$

where $\epsilon = d \log(\beta_{max}/\beta_{min})$. This completes the proof. □

Theorem 3. *Our two-layer countermeasure is (α, ϵ)-locally indistinguishability secure against passive adversaries.*

Proof. We use a two-step strategy to analyze the privacy of our countermeasure. More specifically, we first show that the `EncodeIndex` in Fig. 5 is (α, ϵ)-indistinguishable secure and then prove that our countermeasure is (α, ϵ)-indistinguishable secure as well.

Let $\mathcal{T}_{\mathrm{W}}$ be the set of index vectors of keywords in $\mathcal{W}$ and $\mathcal{T}_{\mathrm{Q}}$ be the set of output index vectors. We claim the adversary does not know the correlation between the encrypted IDs and plaintext IDs because the volume is hiding. Thus, their relationship can be formulated as a random permutation. Without loss of the generality, the probability that one entry in the index is permuted with another can be assumed to be $1/n$. Combined with the `EncodeIndex` protocol, we can easily get

$$e^{-\epsilon} < \frac{\Pr[\texttt{EncodeIndex}(\tau_{w_i}) = \tau_q]}{\Pr[\texttt{EncodeIndex}(\tau_{w_j}) = \tau_q]} < e^{\epsilon} \tag{4}$$

where τ_{w_i} and τ_{w_j} are arbitrary index vectors in $\mathcal{T}_{\mathrm{W}}$ and τ_q is randomly selected in $\mathcal{T}_{\mathrm{Q}}$.

Recall that, previous discussion has shown that the key factor contributing to query privacy is the characteristic derived from access pattern. Thus the probability that q' can be recovered must satisfy

$$\begin{aligned}\Pr[q \triangleq w_i] &= \sum_{\tau_q \in \mathcal{T}_{\mathrm{Q}}} \Pr[\mathcal{K}_{\tau_q} = o_{q'}] \Pr[\texttt{EncodeIndex}(\tau_{w_i}) = \tau_q] \\ &\leq \sum_{\tau_q \in \mathcal{T}_{\mathrm{Q}}} e^{\epsilon} \Pr[\mathcal{K}_{\tau_q} = o_{q'}] \Pr[\texttt{EncodeIndex}(\tau_{w_j}) = \tau_q] \\ &\leq e^{\epsilon} \Pr[q \triangleq w_j]\end{aligned}$$

where $\mathcal{K}_{\tau_q}$ denotes the leakage of τ_q, o_q is the observed leakage of q, $\tau_{w_i}, \tau_{w_j} \in \mathcal{T}_{\mathrm{W}}$ and $\tau_q \in \mathcal{T}_{\mathrm{Q}}$. The above result directly implies that if the `EncodeIndex` protocol is indistinguishable secure then the encrypted search scheme is (α, ϵ)-indistinguishable secure. This completes the proof. □

Combining the above theorems, we observe that for an encrypted scheme equipped with `EncodeIndex` and `EncodeFile` protocol, no active and passive adversaries can identify the exact content of the query by comparing the characteristic derived from the leakage profiles.

6.4 Balancing Efficiency and Security

While the encoded approach helps hide the privacy risk in leakage profile, it also brings extra storage overhead. As mentioned before, α and β are two parameters that are security parameters. In general, larger α and β can provide stronger security guarantees but will result in heavier storage overhead. Specifically, for a larger cluster, the keywords with various frequencies may be gathered together, padding this keyword to the same frequency requires introducing more overhead than that of the smaller clusters. Thus, how to group the keywords to minimize the storage overhead should be addressed. In addition, it is also hard to tell the clients when to perform encoding on the updated data. According to the principle of padding to the maximum mentioned above, it is easy to find that: (1) high-frequency encoding will result in heavy storage overhead (2) low-frequency encoding makes large amount of files stay at the client-side.

In order to address first problem, for the given α and β, we follow the grouping strategy proposed by Bost et al. [17]. In their work, they formulate the grouping strategy as a optimize problem, which studies how to smooth the keyword with minimum keyword/ID pairs. In their solution, the keywords with a similar frequency will be gathered in the same cluster, thus light overhead is needed. To deal with the second problem and further reduce the storage cost, we propose the batch padding and introduce buffer memory (fixed size) at the client's side to help temporarily store the data to be uploaded and encoded. Doing so can avoid frequently padding and leave the client more space to make padding strategy. Specifically, the client predefine the maximum number of files that can store at local device, i.e., γ. Moreover, to mitigate the local storage overhead simultaneously, three trigger conditions are provided for perform encoding:

(1) Batch size. When number of files stored at client achieves γ, the client runs the padding.
(2) Buffer-full. Once the buffer is full, the client releases all existed keyword-identifier pairs.
(3) Time-out. Considering the freshness issue, we set a time limit for enforcing releasing.

Observe that, once one of the above limits is reached, the client selectively release buffer clusters and then encode them through `EncodeIndex` protocol. In the long run, as more fake entries are gained with the updates, it will bring heavy storage overhead for the server. Deleting the unnecessary ones or rebuild the index may be two possible solutions, but when and how to do this should be further considered.

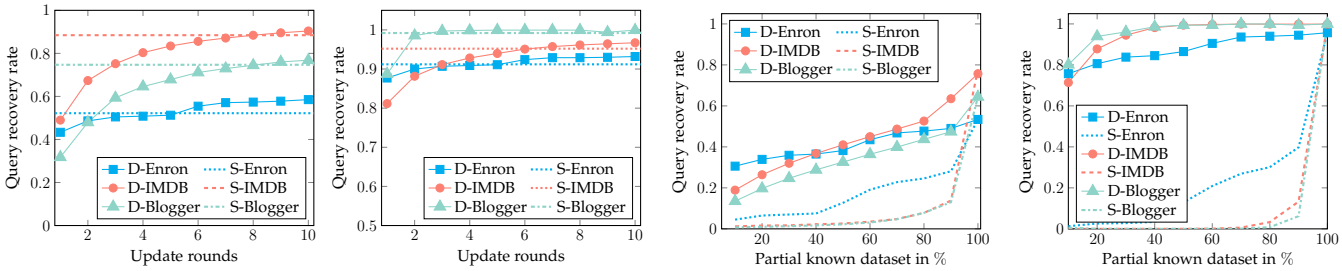
7 Performance Evaluations

Previous sections theoretically analyze the privacy risk of encrypted database supporting data dynamics and present a corresponding countermeasure. In this section, a series of experiments will be conducted on three real-world databases to confirm our theoretical results, including attacks and countermeasures.

7.1 Experiment Setup

To evaluate the performance of the attacks and countermeasures proposed in this work, we conduct a series of experiments with a desktop with the Intel Core i7, 3.2GHz processor and 16GB memory running MacOS. Specifically, we first implement our passive and active attacks on three real-world dataset, Enron email dataset[1], IMDB movie comments dataset[2] and Blogger corpus dataset [3]. For each attack experiment, we randomly choose 10,000 files from the original dataset as the test dataset, the keyword space used to build index consists of 10,000 keywords selected from those 10,000 files. We simulate the update procedures and observe how recovery rate changes as the database updates. For countermeasures, we focus on observing how much storage overhead and time cost are needed for different security strength, including both file and index. For this

1. Enron email dataset: https://www.cs.cmu.edu/~./enron/.
2. IMDB movie comments dataset: https://datasets.imdbws.com/.
3. Blogger Corpus Dataset: http://u.cs.biu.ac.il/~koppel/.



(a) Full: randomly selected queries (b) Full: top frequency queries (c) Partial: random selected queries (d) Partial: top selected queries

Fig. 8. Query recovery rate of relation-based inference attack on fully and partially known dataset. The left two figures show the attack recovery rate against the encrypted database with fully known background and the right two are for the encrypted database with partially known background. Regarding notations, we use "D" to denote the attack on dynamic scenario and "S" to denote that of static scenario.

part, we take the above datasets as the experiment set, and record the storage overhead for index, file and batch padding parameters α, β and γ.

7.2 Implementation of Proposed LAAs

This subsection starts with the passive attack experiments. We exploit the experiments to answer following questions: (1) what is the relationship between leaks and updates in dynamic databases? (2) whether the dynamic database leak more than that of statics?

Passive attack on Fully and Partially Known Dataset. To answer the first question, our first experiment is set as follows: we assume that the encrypted database runs updates every 1,000 files and choose the files in 10 updates, i.e., 10,000 files as the test dataset. With the update goes, we launch our passive attack under the setting that fully and partially dataset are revealed to the adversary, respectively, and record their attack recovery rates. As mentioned before, partially known dataset means that files in some of the updates can be fetched.

As shown in Fig. 8(a) and Fig. 8(b), they report the recovery rate of 10,000 randomly selected queries and 10,000 most frequent queries against the encrypted database with full knowledge known. The plots in the figures show that the recovery rate increases along with the updates. Because with the update operation going, more and more unique characteristics are gradually emerging. For example, in the IMDB database, for randomly selected queries, the recovery rate increases from 49.99% to 90.41%, and for 10,000 most frequent queries, it increases from 81.16% to 96.71%. As for the attack result for encrypted database with partial known background knowledge, it is presented in Fig. 8(c) and Fig 8(d). The similar result can be seen, just like that of full background known attack, more and more queries will be leaked as the update progresses.

To answer the second question, we should demonstrate that the update pattern in dynamic databases does leak more than that of static databases. With regard to the attack for the static case, we run the attack over the overall encrypted database after 10 updates and compare its results with that of dynamic's. The attack results are given by dotted lines in Fig. 8(a), Fig. 8(b), Fig. 8(c) and Fig 8(d). From the experiment results in above four figures, we can find that more queries in dynamic databases are recovered while compared with that in static's, particularly in the case of partial background attack. The most significant is that, according to the update pattern, the adversary can lock the candidates of a unknown query in a certain subset of keywords that appear in some certain update stages (time interval). For instance, the adversary may only know the background of the files in the third update round, then she can fetch the query result in that time for infer the content of the query rather than all of that in the database.

TABLE 2
Storage cost of Structured-based Decoding Attack

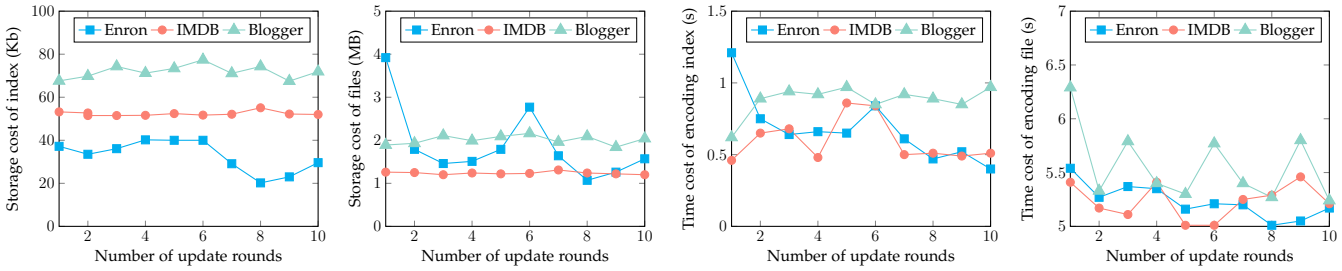
Size of $\mathcal{W}$	Number of files		Totoal volume size	
	T=256	T=512	T=256	T=512
1024	40	22	68.5 KB	74.7 KB
2048	80	44	150.8KB	150.6 KB
3072	120	66	452.8KB	328.6 KB
4096	160	88	4.27MB	422.6 KB
5120	200	110	64.33MB	496.7 KB

File-injection Cost for Active Attack. For the structure-based decoding attack, as the adversary can identify the injected file through its volume size and then decoding them to recover the unknown queries, the recovery rate will be always "1". Regarding this, in this part we only concern the cost the adversary required for launching injection attacks. Table 2 displays the number of files and corresponding volume required for executing structured-based decoding attack. It shows the relationship of the storage cost, number of various keywords in one file (threshold T) and size of keyword space. The results show that, with the size of keyword space increasing, the adversary needs to inject more files for recovering the content of the query. And the total storage overhead will deduce if it allows the file to involve more keyword. For example: when setting $\mathrm{T} = 256$, the adversary just needs to inject 40 files with total volume 68.5 KB for recovering 1024 queries.

7.3 Performance of Our Countermeasures

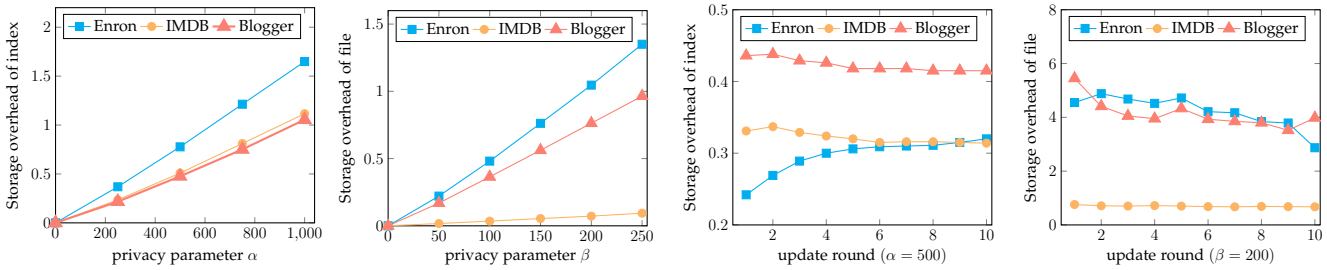
In this section, we evaluate the performance of our two-layer countermeasure from two metrics, i.e., computation cost, communication cost and security strength . In a nut shell, we first equip the database with our countermeasure and measure the storage overhead it cost on real-world datasets, then run both above attacks on those hardening encrypted datasets to check its effectiveness.

Storage Cost of Countermeasures. The storage overhead of our two-layer countermeasure are two folders, including client storage cost and server storage cost. Here, we mainly focus on investigating the influence of update rounds and



(a) Storage cost of index (b) Storage cost of file (c) Time cost of encoding index (d) Time cost of encoding file

Fig. 9. Client-side: storage cost and extra time cost for our two-layer countermeasures on dataset consisting of 10,000 files and 10,000 keywords. The left two figures draw the stroage cost of index and files with privacy parameters $\alpha = 200, \beta = 200, \gamma = 1000$, and the right two figures draw the time cost of encoding index and files.



(a) Storage overhead of index (b) Storage overhead of file (c) Storage overhead of index (d) Storage overhead of file

Fig. 10. Server-side: storage overhead of two-layer countermeasures with different security parameters. The left two figures present the storage overhead on index and the right two figures present the stroage overhead for files.

privacy parameter on the storage overhead. Regarding this, we select different security parameters and run our encoding protocol on the selected datasets, where each dataset consists of 10,000 files and 10,000 most frequent keywords.

From the client perspective, we focus on measuring the communication cost and the computation cost produced from the encoding algorithm. As seen in Fig. 9(a) and Fig. 9(b), when we set $\alpha = 200, \beta = 200, \gamma = 1000$, the overall storage costs for index and files stored at the client-side are about 40.2Kb and 3.92Mb for Enron dataset. It is clear that such storage cost will increase as the size of γ grows. For efficiency, the time costs of executing index and file encoding are presented in Fig. 9(c) and Fig. 9(d). From the figures, we can see that it takes about 0.65s to encode the index and 5.2s to encode the files per update.

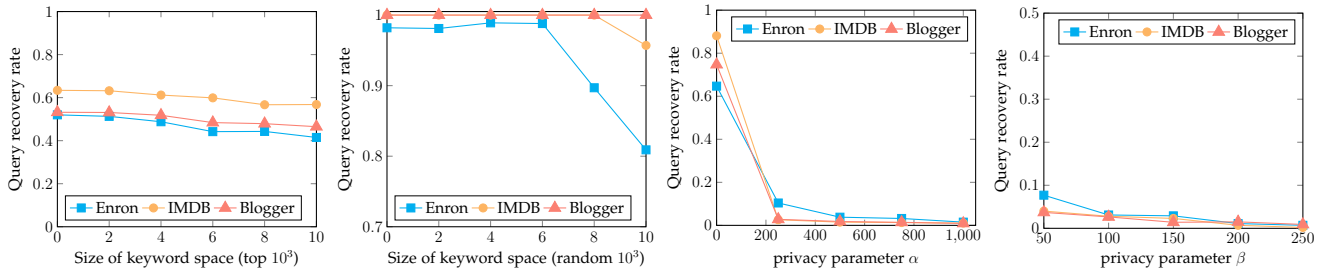
From the server perspective, we mainly compute the storage overhead of countermeasures under different security parameters, especially α and β. Figure. 10(a) reports the relationship between the index overhead and privacy parameters, it shows that the overhead increases as α increases; For example, when we set $\alpha = 250$, it requires to introduce about 36.9% file-keyword pairs in index, and it will increase to 164% while we set $\alpha = 1000$. Figure 10(b) shows the relationship between the extra storage cost of files and parameter β, it also increases with β. When we set $\alpha = 500$ and $\beta = 200$, Figure. 10(c) and Fig. 10(d) display the introduced overhead in each update stage, both of them can be bounded in an admissible ratio.

Effectiveness of Countermeasures. Following we show the effectiveness of the proposed two-layer countermeasure. Specifically, we demonstrate that there should be no trade-off between encoding the index and files. For index, it is clear that encoding index only is not sufficient, because in the active attack the adversary determines the content of the query by files she injects. For files, as the dummy keywords added into the document lacks full accounting for the leakage profiles, it also may be invalid if we only encode the file without obfuscating the index. To verify this argument, we run the passive attack on the database produced by file encoding. From the attack results listed in Fig. 11(a) and Fig. 11(b), we can see that there are still quite a few encrypted queries can be recovered, which validates the necessity of the two-layer solution.

In order to provide a more intuitive grasp of the proposed two-layer solution, we then run our attacks on the databases whose file volumes and the index are both obfuscated. The detailed experimental results are recorded in Fig. 11(c) and Fig. 11(d). Specifically, Fig. 11(c) shows the query recovery rate for encrypted index encoded with privacy parameter α. From the lines in Fig. 11(c), we can see that the recovery rate decreases as α increases. Taking the Enron email dataset as an example, when $\alpha = 250$, 10.4% of selected queries can be recovered, and it decreases to 1.5% when $\alpha = 1000$. Namely, the larger the privacy parameter α is, the more secure the encrypted databases are. Figure. 11(d) draws the attack results for different β's, it also shows that larger β can brings better security performance. Overall, for Enron dataset, our two-layer can introduce less than $1.2\times$ total overhead to reduce the recovery rate to below 0.5%.

8 Related Work

Cryptographic Databases and Encrypted Search. Encrypted databases [28], [29], a type of storage system designed to deal with data disclosure problems [30], [31], have drawn much attention from both industries and academics. Using the encrypted database can save industries, governments or individuals the trouble of having to store



(a) Passive attack v.s encoded file (b) Pasive attack v.s encoded file (c) Passive attack v.s two-layer's (d) Active attack v.s two-layer's

Fig. 11. Recovery rate of our passive attack and active attack on hardening dataset produced by one-layer file encoding countermeasures and two-layer solution

sensitive information in a network-isolated environment, enabling them to enjoy the more cost-effective cloud service. However, for security concerns, the data stored in the encrypted database is always transformed into the ciphertext that is incomprehensible. To preserve the utility, many advanced cryptographic primitives [32], [33], [34] or hardware-assisted techniques [28], [29] are adopted in. Searchable encryption (SE) [4], [35] is one of them which is leveraged to secure the private search of encrypted data without decryption. Early efforts in SE always focus on improving efficiency [20], [21], [36], expressiveness [37], [38], [39], [40] and security [19], [41], [42]. For example, searchable symmetric encryption [32], order-preserving encryption and order-revealing encryption (OPE/ORE) are more flexible to support sorting, performing range queries, and filtering data [34], [38]. A notable feature of most of existing secure search schemes is that they do not protect the search pattern and access pattern, i.e., the adversary can know which query is repeated and which files are returned for a query.

Leakage Abuse Attacks. The problem of how strong security can the encrypted database provide has been investigated for many years. In the early years, researchers always leverage the notion of leakage profile to quantify what will be fundamentally leaked in encrypted databases or cryptographic primitives [20], [32]. But the security implication of such leakage is not well understood in early days. In fact, beyond the above leakage profiles are the invariant characteristics that come along with access pattern and search pattern before and after encryption, e.g., result length pattern, volume pattern. It is first empirically studied by Islam et al. that the access pattern, given some background knowledge, can be utilized to infer the queried keywords and infringe privacy [7]. Later, Cash et al. show improved attacks by combining both co-occurrence of keywords and query result size, and investigate their effectiveness in varying degree of adversary's prior knowledge [8]. Subsequently, a series of leakage-abuse attacks targeting range query [15], [43] and dynamic setting [24] are proposed, amplifying the fact that the existing security models of encrypted search schemes can no longer capture the power of real-world adversaries.

Mitigation Exploration. In the countermeasures front, various padding strategies are widely employed to remove the unique result length by adding extra document/keyword pairs in the database [8], [17], [41]. Hence, the adversary can no longer exploit the volume information to launch the attack. However, such defenses cannot provide privacy protection comprehensively. The effectiveness of the protection on other invariant characteristics like query co-occurrence counts is not guaranteed. Note that though ORAM-based algorithms [44], [45], [46] can be theoretically applicable to hide query leakage, more computation and communication overhead are essentially brought in compared to SSE. Nevertheless, the ORAM cannot serve protection for volume information, which may also result in privacy explosive. Very recently, Chen et al. [47] use the concept of differential privacy to obfuscate the access pattern and Patel et al. [18] exploit hashing on multi-map to realize volume-hiding. However, these countermeasures are not grounded on proper understanding of the intrinsic security limitations of encrypted databases.

9 Conclusion

Privacy risk and defense related to the static database has been explored in past literatures, but few works are concerned on that about dynamics. In this paper, we systematically investigate the leakage exploitation of encrypted database that supports data dynamics and devise a two-layer database hardening approach. Our findings give a comprehensive view of the privacy risk of dynamic databases against both passive and active attacks, and provide a guidance on how to mitigate them. Similar to previous works, security and overhead trade-offs also exist. Reducing system overhead without losing security strength is still an issue for us to study in the future.

References

[1] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: protecting confidentiality with encrypted query processing," in *Proc. of SOSP*, 2011.
[2] B. Fuller, M. Varia, A. Yerukhimovich, E. Shen, A. Hamlin, V. Gadepally, R. Shay, J. D. Mitchell, and R. K. Cunningham, "Sok: Cryptographically protected database search," in *Proc. of IEEE S&P*, 2017.
[3] A. Agarwal, M. Herlihy, S. Kamara, and T. Moataz, "Encrypted databases for differential privacy," *PoPETs*, vol. 2019, no. 3, pp. 170–190, 2019.
[4] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *J. ACM*, vol. 45, no. 6, pp. 965–981, 1998.
[5] M. Chase and S. Kamara, "Structured encryption and controlled disclosure," in *Proc. of ASIACRYPT*, 2010.
[6] S. Kamara, T. Moataz, and O. Ohrimenko, "Structured encryption and leakage suppression," in *Proc. of CRYPTO*, 2018.
[7] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *Proc. of NDSS*, 2012.
[8] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *Proc. of the ACM CCS*, 2015.

[9] V. Bindschaedler, P. Grubbs, D. Cash, T. Ristenpart, and V. Shmatikov, "The tao of inference in privacy-protected databases," *PVLDB*, vol. 11, no. 11, pp. 1715–1728, 2018.

[10] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia, "Data recovery on encrypted databases with k-nearest neighbor query leakage," in *Proc. of IEEE S&P*, 2019.

[11] ——, "The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution," in *Proc. of IEEE S&P*, 2020.

[12] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart, "Leakage-abuse attacks against order-revealing encryption," in *Proc. of IEEE SP*, 2017.

[13] L. Blackstone, S. Kamara, and T. Moataz, "Revisiting leakage abuse attacks," *In Proc. of NDSS*, vol. 2020.

[14] Z. Gui, O. Johnson, and B. Warinschi, "Encrypted databases: New volume attacks against range queries," in *Proc. of ACM CCS*, 2019.

[15] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill, "Generic attacks on secure outsourced databases," in *Proc. of ACM CCS*, 2016.

[16] P. Grubbs, M. Lacharité, B. Minaud, and K. G. Paterson, "Learning to reconstruct: Statistical learning theory and encrypted database attacks," in *Proc. of IEEE S&P*, 2019.

[17] R. Bost and P. Fouque, "Thwarting leakage abuse attacks against searchable encryption - A formal approach and applications to database padding," *IACR Cryptology ePrint Archive*, vol. 2017, p. 1060, 2017.

[18] S. Patel, G. Persiano, K. Yeo, and M. Yung, "Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing," in *Proc. of ACM CCS*, 2019.

[19] S. Kamara and T. Moataz, "Computationally volume-hiding structured encryption," in *Proc. of EUROCRYPT*, 2019.

[20] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proc. of ACM CCS*, 2012.

[21] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *Proc. of NDSS*, 2014.

[22] I. Miers and P. Mohassel, "IO-DSSE: scaling dynamic searchable encryption to millions of indexes by improving locality," in *Proc. of NDSS*, 2017.

[23] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *Proc. of ACM CCS*, 2017.

[24] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in *Proc. of USENIX Security*, 2016.

[25] B. Bichsel, T. Gehr, D. Drachsler-Cohen, P. Tsankov, and M. T. Vechev, "Dp-finder: Finding differential privacy violations by sampling and optimization," in *Proc. of ACM CCS*, 2018.

[26] M. Mohammady, L. Wang, Y. Hong, H. Louafi, M. Pourzandi, and M. Debbabi, "Preserving both privacy and utility in network trace anonymization," in *Proc. of ACM CCS*, 2018.

[27] E. G. Altmann, Z. L. Whichard, and A. E. Motter, "Identifying trends in word frequency dynamics," *Journal of Statistical Physics*, vol. 151, pp. 277–288, 2013.

[28] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *Proc. of IEEE S&P*, 2018.

[29] D. Vinayagamurthy, A. Gribov, and S. Gorbunov, "Stealthdb: a scalable encrypted database with full SQL query support," *PoPETs*, vol. 2019, no. 3, pp. 370–388, 2019.

[30] C. Cadwalladr and E. Graham-Harrison, "Revealed: 50 million facebook profiles harvested for cambridge analytica in major data breach," *The guardian*, vol. 17, p. 22, 2018.

[31] Y. Zou, S. Danino, K. Sun, and F. Schaub, "You 'might' be affected: An empirical analysis of readability and usability issues in data breach notifications," in *Proc. of ACM CHI*, 2019.

[32] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *Proc. of ACM CCS*, 2006.

[33] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang, "Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward," in *Proc. of the ACM CCS*, 2015.

[34] D. Boneh, K. Lewi, M. Raykova, A. Sahai, M. Zhandry, and J. Zimmerman, "Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation," in *Proc. of EUROCRYPT*, 2015.

[35] D. X. Song, D. A. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proc. of IEEE S&P*, 2000.

[36] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *Proc. of FC*, 2013.

[37] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *Proc. of CRYPTO*, 2013.

[38] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Order-preserving encryption for numeric data," in *Proc. of the ACM SIGMOD*, 2004.

[39] C. Wang, N. Cao, J. Li, K. Ren, and W. Lou, "Secure ranked keyword search over encrypted cloud data," in *Proc. of ICDCS*, 2010, pp. 253–262.

[40] C. Wang, K. Ren, S. Yu, and K. M. R. Urs, "Achieving usable and privacy-assured similarity search over outsourced cloud data," in *Proc. of the IEEE INFOCOM*, 2012, pp. 451–459.

[41] S. Kamara and T. Moataz, "Encrypted multi-maps with computationally-secure leakage," in *Proc. of EUROCRYPT*, 2019.

[42] C. Cai, J. Weng, X. Yuan, and C. Wang, "Enabling reliable keyword search in encrypted decentralized storage with fairness," *IEEE Trans. Dependable Secur. Comput.*, vol. 18, no. 1, pp. 131–144, 2021.

[43] P. Grubbs, M. Lacharité, B. Minaud, and K. G. Paterson, "Pump up the volume: Practical database reconstruction from volume leakage on range queries," in *Proc. of ACM CCS*, 2018.

[44] T. H. Chan, K. Chung, B. M. Maggs, and E. Shi, "Foundations of differentially oblivious algorithms," in *Proc. of SODA*, 2019.

[45] E. Stefanov, M. van Dijk, E. Shi, T. H. Chan, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," *Journal of ACM*, vol. 65, no. 4, pp. 18:1–18:26, 2018.

[46] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *Proc. of NDSS*, 2014.

[47] G. Chen, T. Lai, M. K. Reiter, and Y. Zhang, "Differentially private access patterns for searchable symmetric encryption," in *Proc. of INFOCOM*, 2018.