d-DSE: Distinct Dynamic Searchable Encryption Resisting Volume Leakage in Encrypted Databases

Dongli Liu and Wei Wang, *Huazhong University of Science and Technology;* Peng Xu, *Huazhong University of Science and Technology, Hubei Key Laboratory of Distributed System Security, School of Cyber Science and Engineering, JinYinHu Laboratory, and State Key Laboratory of Cryptology;* Laurence T. Yang, *Huazhong University of Science and Technology and St. Francis Xavier University;* Bo Luo, *The University of Kansas;* Kaitai Liang, *Delft University of Technology*

https://www.usenix.org/conference/usenixsecurity24/presentation/liu-dongli

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

d-DSE: Distinct Dynamic Searchable Encryption Resisting Volume Leakage in Encrypted Databases

Dongli Liu[1], Wei Wang[1,✉], Peng Xu[1,2,3,4,✉], Laurence T. Yang[1,5], Bo Luo[6], and Kaitai Liang[7]

[1]*Huazhong University of Science and Technology*
[2]*Hubei Key Laboratory of Distributed System Security, School of Cyber Science and Engineering*
[3]*JinYinHu Laboratory*
[4]*State Key Laboratory of Cryptology*
[5]*St. Francis Xavier University*
[6]*The University of Kansas*
[7]*Delft University of Technology*
[✉]*Corresponding authors: viviawangwei@hust.edu.cn, xupeng@hust.edu.cn*

Abstract

Dynamic Searchable Encryption (DSE) has emerged as a solution to efficiently handle and protect large-scale data storage in encrypted databases (EDBs). Volume leakage poses a significant threat, as it enables adversaries to reconstruct search queries and potentially compromise the security and privacy of data. Padding strategies are common countermeasures for the leakage, but they significantly increase storage and communication costs. In this work, we develop a new perspective on handling volume leakage. We start with *distinct search* and further explore a new concept called *distinct* DSE (*d*-DSE).

We also define new security notions, in particular Distinct with Volume-Hiding security, as well as forward and backward privacy, for the new concept. Based on *d*-DSE, we construct the *d*-DSE designed EDB with related constructions for distinct keyword (d-KW-*d*DSE), keyword (KW-*d*DSE), and join queries (JOIN-*d*DSE) and update queries in encrypted databases. We instantiate a concrete scheme BF-SRE, employing Symmetric Revocable Encryption. We conduct extensive experiments on real-world datasets, such as Crime, Wikipedia, and Enron, for performance evaluation. The results demonstrate that our scheme is practical in data search and with comparable computational performance to the SOTA DSE scheme (MITRA*, AURA) and padding strategies (SEAL, ShieldDB). Furthermore, our proposal sharply reduces the communication cost as compared to padding strategies, with roughly 6.36 to 53.14x advantage for search queries.

1 Introduction

Encrypted databases (EDBs) [46] have been developed to support privacy-preserving data storage and search services. They allow clients to outsource sensitive encrypted data to a server and then send search queries to the server who can return the corresponding data. These databases can adopt Searchable Symmetric Encryption (SE) [13, 51] and its Dynamic version [31] (DSE) to offer various types of privacy-preserving queries, such as keyword [16], range [17], SQL [27] queries, and updates. Note more related works for EDBs and DSE are presented in Appendix B and D, respectively.

Although easily inheriting Forward Privacy and Backward Privacy (FP&BP) guarantees from DSE [2,4], those EDBs are still vulnerable to volumetric attacks [1, 64]. The adversary can infer the underlying keywords from search queries [1, 36, 64] or even reconstruct the whole EDB [22, 32] by exploiting *volume leakage* (i.e., the length of the response).

Padding strategies [3, 15, 55] are popularly applied to mitigate the impact of volume leakage in EDBs. They work by adding "dummy" data to the original database to ensure uniform volume for each keyword. Thus, from the viewpoint of adversaries, volume leakage with regard to queries is now indistinguishable. This strategy, however, yields significant costs (see Tab. 1). It requires extra storage and computational resources to establish and update the EDB but also consumes huge communication bandwidth for query response. Note we provide more details on padding in Appendix C.

A new perspective - starting from *distinct search*. In line with the design of secure EDBs [18], we should minimize the leakage during search queries. It is natural to see that the SQL syntax 'SELECT DISTINCT'[1] can be used to eliminate duplicated search responses, which, to some extent, reduces the volume leakage. But we cannot simply apply distinct search in EDBs due to the fact that it still leaks information to the server. In the context of SQL queries, a distinct search retrieves distinct (unique) values from the corresponding columns in a database table, which allows us to eliminate "duplicate" values. The search process traverses only one copy of each value and ignores its duplicates. However, if the number of those "ignored" values (i.e., the number of duplicates) is leaked, it can reveal information about the quantities of all relevant values, leading to volume leakage. To mitigate this threat, it is crucial to design a mechanism that prevents the server from being aware of the repetition of values. We find that we can use the dummy tags to group all relevant values into an

[1]In ISO/IEC 9075-2:2016, the 'DISTINCT' predicate can apply with other syntax like 'SELECT', 'JOIN', 'GROUP BY' to retrieve distinct values from a database table.

Table 1: Comparison on related DSE, padding, and our scheme. N is the total number of keyword/value pairs. W and F are the total number of keywords and files, respectively. In Column **5**, ✓ indicates that padding strategies can be deployed in the schemes. For keyword $\mathbf{w}$, a_w is the total number of update operations, n_w is the current number of pairs, d_w is the number of deletion, d_{max} is the supported maximum number of deletions, and s'_w is the number of legal search tokens. For **SEAL**, x means the adjustable padding's parameter, and n^*_w denotes the number of the pairs currently containing w after padding. For **ShieldDB**, n_r and n_b separately represent the real and padding pairs of the streamed keyword w_i, $|B|$ means the padding dataset length, and f_w and f_{w_i} represent the maximum frequency and the streamed keyword w_i frequency, respectively. Backward privacy (**BP**) contains type-1/2/3 level (I/II/ III), including the special case $\mathrm{II}^{\mathrm{A}}, \mathrm{III}^{\mathrm{R}}, \mathrm{II}^{\mathrm{D}}$ extended from [4].

Schemes	Distinct Search	Computation			Communication			Client Storage	BP
		Search	Update	Padding	Search	Update	Comm. Round		
MONETA [4]	✗	$O(a_w \log N + \log^3 N)$	$O(\log^2 N)$	✓	$O(a_w \log N + \log^3 N)$	$O(\log^3 N)$	3	$O(1)$	I
DIANA$_{del}$ [4]	✗	$O(a_w)$	$O(\log a_w)$	✓	$O(n_w + d_w \log a_w)$	$O(1)$	2	$O(W \log F)$	III
JANUS [4]	✗	$O(n_w \cdot d_w)$	$O(1)$	✓	$O(n_w)$	$O(1)$	1	$O(W \log F)$	III
JANUS++ [54]	✗	$O(n_w \cdot d_{max})$	$O(d_{max})$	✓	$O(n_w)$	$O(1)$	1	$O(W \log F)$	III
AURA [53]	✗	$O(n_w)$	$O(1)$	✓	$O(n_w)$	$O(1)$	1	$O(W \cdot d_{max})$	II^{A}
MITRA [20]	✗	$O(a_w)$	$O(1)$	✓	$O(a_w)$	$O(1)$	2	$O(W \log F)$	II
SD$_\mathbf{a}$ [14]	✗	$O(a_w + \log N)$	$O(\log N)$	✓	$O(a_w + \log N)$	$O(\log N)$	2	$O(1)$	II
SD$_\mathbf{d}$ [14]	✗	$O(a_w + \log N)$	$O(\log^3 N)$	✓	$O(a_w + \log N)$	$O(\log^3 N)$	2	$O(1)$	II
ROSE [62]	✗	$O((n_w + s'_w + 1)\, d_w)$	$O(1)$	✓	$O(n_w)$	$O(1)$	2	$O(W \log F)$	$\mathrm{III}^{\mathrm{R}}$
SEAL* [15]	✗	$O(x \cdot n^*_w)$	✗	$O(x \cdot N)$	$O(x \cdot n^*_w)$	✗	1	$O(x \cdot N)$	✗
ShieldDB's DSE* [55]	✗	$O(n_r + n_b)$	$O(n_r + n_b)$	$O(\lvert B \rvert (f_w - f_{w_i}))$	$O(n_r + n_b)$	$O(n_r + n_b)$	1	$O(\lvert B \rvert)$	–
BF-SRE (Ours)	✓	$O(n_w)$	$O(1)$	✓	$O(n_w)$	$O(1)$	1	$O(W \cdot d_{max})$	II^{D}

*: SEAL only focuses on static database (indicating it cannot support update operations) while providing values retrieval with keyword. In SEAL we store the client's states locally.
⋆: ShieldDB is a dynamic document database. Its deletion is done through re-encryption, which does not clearly specify the backward privacy.

unsearchable dataset so that the server, in the Search stage, can only "see" a single copy of results, thereby concealing the volume of repetitive values and reducing the potential for volume leakage.

A trivial solution to transform DSE for volume-hiding. Recall that DSE focuses on searching file-identifiers instead of values. It may not be straightforward to replace the distinct identifiers with the unique/repeatable values. Here, we attempt to provide a *trivial transformation* to make DSE support "distinct search."

• In the Setup or Update stage, the client maps each value to a unique identifier and then initializes or updates the keyword-value pairs by a DSE instance.

• To perform the distinct search, the client utilizes the query protocol of the DSE to retrieve these identifiers. Afterward, the client restores the values based on the mapping of the identifiers and then extracts the distinct values locally.

We see that this transformation necessitates the client to maintain a local mapping table for translation between identifiers and values during each search. Furthermore, its security and efficiency heavily rely on the underlying DSE instance. We note that a regular DSE scheme could still be vulnerable to volume leakage (e.g., [60]), or it applies expensive padding strategies incurring significant bandwidth and storage costs (e.g., [15, 55]). This observation indicates that a trivial transformation could not be the best solution for both security and efficiency.

In response to this, we propose the *Distinct Dynamic Searchable Symmetric Encryption (d-DSE)* that enables clients to securely search for distinct values with volume-hiding. In this concept, we make use of the Distinct State, the Distinct Classifier, and the Distinct Constraint to determine distinct values and eliminate the volume difference. We instantiate a d-DSE scheme BF-SRE for non-interactive deletion and efficient search. We present a brief comparison between BF-SRE and related (D)SE schemes in Tab. 1.

Extension from "distinct" to "diverse" search. We notice that many works have demonstrated the feasibility of incorporating DSE to support secure queries in SQL syntax, such as keyword [15], range [17], and join [24, 50]. Fortunately, we find out that the distinct search can be integrated into secure SQL as well. Our core idea is to process a query by first obtaining the *distinct values* using the distinct search and subsequently restoring the quantity of each distinct value (i.e., *value's quantity*) by small client storage. Please see Sec. 4 for more details.

The above descriptions briefly introduce our technical roadmap from the beginning by distinct search, constructing secure distinct search (i.e., d-DSE), and fulfilling SQL search for EDBs. We summarize key contributions as follows.

• **Definition and models of *d*-DSE**. We underline the definition and models for secure distinct search. In particular, we propose a so-called Distinct with Volume-Hiding (DwVH) security that captures how a secure distinct search can mitigate volume leakage. We formalize the EDB context of distinct search, analyze security risks, and define the advantage in the models.

• ***d*-DSE designed EDB**. We leverage d-DSE to build EDBs with volume-hiding property. We first illustrate the EDB system enhanced by the d-DSE query model. Then we present three new constructions: d-KW-dDSE (for distinct keyword queries), KW-dDSE (for keyword queries), JOIN-dDSE (for join queries), and the corresponding row addition and deletion operations. We comprehensively expose the leakage functions [6, 26] under these constructions and interpret the pattern leakage from the EDB perspective. Finally, our analysis clari-

fies that they resist both SOTA passive [60] and active [64] volumetric attacks.

• **Concrete scheme for *d*-DSE**. To handle the distinct feature in *d*-DSE designed EDB, we utilize the Bloom Filter (BF) [53], producing Distinct State to control the tag generation for values. In this sense, we can effectively match the distinct values in the search. Based on the above, we propose our scheme, called BF-SRE, by applying Symmetric Revocable Encryption (SRE). The scheme enables the client to process the deletion locally, and thus, it is non-interactive and flexible to row update in encrypted databases. We also provide formal security analysis to demonstrate that the scheme satisfies the FP&BP as well as the DwVH security.

• **Evaluations**. We extensively perform evaluations on our proposals and prior work under diverse real-world datasets, including the Crime reports, Wikipedia, and the Enron email datasets. Concretely, under equivalent security parameters, we compare BF-SRE with the SOTA MITRA* [20], AURA [53], SEAL [15], and ShieldDB [55] in terms of the time and communication costs. Our evaluation demonstrates that BF-SRE stands as a competitive solution compared to the aforementioned schemes. For example, on the Enron dataset, its costs of time and communication for searching the highest-volume keyword are 8.25s and 83.82KB, outperforming others by a factor of approximately 29.27x and 30.54x, respectively.

2 Backgrounds

We first describe the encrypted database that supports recording the keyword/value pairs. In this context, we introduce the parties in distinct search with the corresponding interactions. After that, we propose the threat in distinct search.

2.1 Encrypted Database Description

We use EDB to represent an encrypted database in the context of DSE. This database stores repeatable keyword/value pairs (w,v) (e.g., for relational table structure, an attribute's value in one column as a keyword with a foreign-key in another as a value) in sequence and supports secure addition, deletion, and search operations. Specifically, the client can search keywords to retrieve the matching values, add new keyword/value pairs, and delete all specified repetitive (w,v) pairs. Note that the search operation incorporates the distinct search predicate $TypeDB(w) = \{v_1,...,v_n | v \in (w,v)\ and\ \forall\ i,j \in [1,n]\ s.t.\ v_i = v_j \iff i = j\}$ to retrieve all matched distinct values given a keyword w. We call the number of distinct values in $TypeDB(w)$ as the *value's type*.

2.2 Parties

Two parties involve in the distinct search:

• Client: The client initializes the EDB by security parameter and outsources it to the server. The client sends an update token generated by the input $(w,v,op = \{add,del\})$ to add or delete the keyword/value pairs (w,v) on the EDB. The client sends the search token generated by the keyword w to find the corresponding distinct values.

• Server: The server hosts the EDB outsourced by the client. The server updates the EDB by the update token. In processing a search token, the server searches the distinct values corresponding to the keyword w and returns the result to the client.

2.3 The Threat in Distinct Search

A scheme for secure distinct search should protect the client's outsourced EDB against the probabilistic polynomial time (PPT) adversary from the *passive* [40,41,60] and *active* [64] attacks. Akin to [12], we propose the adversaries' observations during various client events that could reveal volumetric information for the aforementioned attacks.

Table 2: The adversaries' passive and active observations.

Client Event	Passive observations	Active observations
Setup: 1.Set up Secret Key; 2.Outsource the EDB.	1.The initial EDB.	
	1.Prior volume knowledge;	1.The deceptive set of keyword/value pairs; 2. Partial queries volume before injection.
Update: 1.Update ciphertexts in the EDB.	1.The updated EDB.	
	1.Updated queries in ciphertexts.	1. Updated queries generated from deceptive set.
Search: 1.Require distinct search.	1.Client search queries; 2.Search process; 3.The volume of search queries.	1. The after-injection volume of queries.

From Tab. 2, we state that the philosophy of volume-hiding countermeasure is to decouple the volumetric relationship speculated from the Setup and Search stage. The passive and active adversaries first get the baseline of volume through prior knowledge (e.g., outdated keyword frequency [40]) and partial queries observed before injection, respectively. Then they both consult the observation in the Update stage to achieve maximum fit in the Search stage, such as minimizing objective functions [41, 60] and manufacturing binary volume [64]. To mitigate the fitting, padding strategies produce uniform volume through dummy data at initialization. Our perspective is to prevent the search leakage containing predicable volume distribution, which is our security goal for distinct search.

We also require Forward Privacy and Backward Privacy (FP&BP) [2,4] to protect the update and distinct search function from serious privacy disclosure. To capture the leakage from distinct search, the security for volume leakage [42] should be defined.

3 Distinct DSE

We define Distinct Dynamic Searchable Symmetric Encryption (*d*-DSE), which includes its scheme and security guarantees, to tackle the distinct search threat.

3.1 Notations

$\lambda \in \mathbb{N}$ is the security parameter. $r \xleftarrow{\$} \mathcal{R}$ means randomly sampling r from the space $\mathcal{R}$. $a||b$ is the concatenation between a and b. $\{0,1\}^{\lambda}$ is a λ-bit length string. $\{0,1\}^{*}$ is an arbitrary-bit length string. F is a secure Pseudo Random Function (PRF). Σ is the Dynamic Searchable Symmetric Encryption (DSE) scheme. The frequently used notations & concepts, the symmetric encryption, Bloom Filter (BF) [53], and Symmetric Revocable Encryption (SRE) [53], are introduced in Appendix A.

3.2 The *d*-DSE Scheme

Definition 1 (The Distinct DSE) *A Distinct Dynamic Searchable Encryption is a triple* $(\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ *consisting of three protocols:*

- $\mathsf{Setup}(\lambda)$ *is a protocol that takes as input the* security parameter λ. *It invokes the* DSE setup protocol $\Sigma.setup$ *and generates the* Distinct State σ^D, *outputting* $K, \mathbf{st} = \{\sigma, \sigma^D\}$ *for the client and outsourcing* EDB *for the server respectively, where K is the* master secret key, EDB *is the* encrypted database, *and* **st** *is the* client's internal state.

- $\mathsf{Search}(K, \mathbf{st}, w; \mathrm{EDB})$ *is a protocol between the client with inputs K,* **st**, *and a search query restricted to a* keyword w, *and the server with input the* EDB. *It invokes the* DSE search protocol $\Sigma.search$ *with input modified by the Distinct State* σ^D. *At the end, the client gets a search result set included* ***distinct values*** *from the* EDB.

- $\mathsf{Update}(K, \mathbf{st}, op, \mathrm{in}; \mathrm{EDB})$ *is a protocol between the client with K and* **st** *as above, and an* operation op *with its input* in, *where op is from the set {add, del}* (i.e. addition, deletion) *and* in *is parsed as the* keyword/value pair (w, v); *and the server with input* EDB. *It invokes the* DSE update protocol $\Sigma.update$ *with input modified by the Distinct State* σ^D. *At the end, the client updates its internal* $\mathbf{st}'$, *and the server renews the* EDB.

Correctness. *Except with negligible probability, the d-DSE scheme is correct if the* Search *protocol returns (current) correct results for the keyword being searched. For the formalism, we follow the case where the client should not delete a keyword with a retrieval value that is not present in* EDB.

Remark on the *d*-DSE search protocol. The retrieval searched by d-DSE and DSE is different [4]. In DSE, the client eventually needs to obtain the outsourced data through the file-identifier, and the volume leakage is not considered by Forward Privacy and Backward Privacy (FP&BP) [60]. In our context, the client is allowed to search the EDB to retrieve outsourced distinct values by a keyword. To this end, we must refine the leakage on retrieving values.

3.3 Security Notions for Distinct Search

To address the leakage ($\mathcal{L}_D$) in d-DSE, we refer to the DSE security. The security contains FP&BP [2,4] under the Adaptive Security model [13], but it falls short in preventing volume leakage. Therefore, we define the Distinct with Volume-Hiding (DwVH) security. Note that we introduce the Sim-adaptive Security model instead of the Adaptive Security model to perceive FP&BP and the DwVH security.

We assume that the client is *honest* and should prevent the disclosure of sensitive information, e.g., underlying keywords and encrypted data. The server, which is *honest-but-curious*, should follow the protocols' instructions yet passively exposes some information. Similar to the *Real* and *Ideal* formulation [4,7], the Sim-adaptive security notions of d-DSE are defined as follows.

Definition 2 (Sim-adaptive Security of *d*-DSE) *Assume a d-DSE scheme is $\mathcal{L}$-adaptively secure iff for all sufficiently large security parameters $\lambda \in \mathbb{N}$ and PPT adversary $\mathcal{A}$, there is a set of efficient simulators $\mathcal{S}$ with a set of leakage functions $\mathcal{L}$ that has:*

$$\left| \mathbb{P}\left[\mathrm{Real}_{\mathcal{A}}^{\mathrm{d-DSE}}(\lambda) = 1 \right] - \mathbb{P}\left[\mathrm{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}^{\mathrm{d-DSE}}(\lambda) = 1 \right] \right| = negl(\lambda),$$

where the games $\mathrm{Real}_{\mathcal{A}}^{\mathrm{d-DSE}}(\lambda)$ *and* $\mathrm{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}^{\mathrm{d-DSE}}(\lambda)$ *are:*

- *Game* $\mathrm{Real}_{\mathcal{A}}^{\mathrm{d-DSE}}(\lambda)$*: the adversary controls the client to run real protocols. Firstly, it triggers the protocol* Setup *and gets the encrypted database* EDB. *Secondly, with the parameters of its choice, it adaptively triggers* Update, Search, *and then obtains the real transcript list* $Q = (q_1, q_2, ..q_n)$. *Finally, it outputs a bit b decided from the real sensitive information* $(\mathrm{EDB}, q_1, ...q_n)$.

- *Game* $\mathrm{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}^{\mathrm{d-DSE}}(\lambda)$*: all of the real protocols are replaced by the set of simulators $\mathcal{S}$. The adversary first triggers the simulator* $\mathcal{S}_{\mathsf{setup}}$ *and obtains the simulated database* $\mathrm{EDB}^{\mathcal{S}}$. *Secondly, with chosen parameters, it adaptively triggers* $\mathcal{S}_{\mathsf{Update}}$, $\mathcal{S}_{\mathsf{Search}}$, *and then gets the simulated transcript* $Q^{\mathcal{S}} = (q_1^{\mathcal{S}}, q_2^{\mathcal{S}}, ..q_n^{\mathcal{S}})$. *Finally, it outputs a bit b decided from the simulated sensitive information* $(\mathrm{EDB}^{\mathcal{S}}, q_1^{\mathcal{S}}, q_2^{\mathcal{S}}, ..q_n^{\mathcal{S}})$.

Common Leakage Functions. Akin to [4], we use the $sp(w)$ to represent the *search pattern*, i.e., which queries belong to the keyword w, and the $\mathsf{UpHist}(w)$ to represent the history of all updates on keyword w.

Forward Privacy (FP). As d-DSE holds FP, adversaries should not utilize the previous transcript to imply the newly added information before executing the Search protocol. Based on FP in [2], d-DSE must further preserve the value's information. This requirement prevents adversaries from deducing the equality among values, which will expose more information than just the distinct values.

Definition 3 (Forward privacy) *An $\mathcal{L}$-adaptively-secure d-DSE scheme has forward privacy if its* Update *leakage function $\mathcal{L}_D^{Upt}$ can be written as:*

$$\mathcal{L}_D^{Upt}(w,v,op) = \mathcal{L}'(op),$$

where $\mathcal{L}'$ is a state-less function.

Backward Privacy (BP). As d-DSE holds BP, adversaries should not use new transcripts to imply previous information after executing Search.

To capture the BP in d-DSE, we first propose the leakage function TimeDTS(w). TimeDTS(w) defines the list of all encrypted values v matching keyword w, excluding the deleted and repetitive ones, together with the timestamp u of when they were added in the database:

$$\begin{aligned}\text{TimeDTS}(w) = \{(u,v) | (u,w,v,add) \in Q \text{ and } \forall\, u', (u',w,v,del) \notin Q \\ \text{and } \forall\, (u,w,v_i,add) \in Q, (u,w,v_j,add) \in Q, \\ s.t.\ v_i = v_j \iff i = j\}.\end{aligned}$$

Update(w) defines the leakage of which timestamps have update queries on the keyword w. With the input w, Update(w) returns the set of the timestamps, in which each of them represents an update on w:

$$\text{Update}(w) = \{u \mid (u,w,v,add) \in Q \text{ or } (u,w,v,del) \in Q\}.$$

With the above functions, we define the BP of d-DSE:

Definition 4 (Backward privacy) *An $\mathcal{L}$-adaptively-secure d-DSE scheme has backward privacy if its* Update, Search *leakage functions $\mathcal{L}_D^{Upt}$, $\mathcal{L}_D^{Srch}$ can be written as:*

$$\begin{aligned}\mathcal{L}_D^{Upt}(w,v,op) &= \mathcal{L}'(w,op) \\ \mathcal{L}_D^{Srch}(w) &= \mathcal{L}''(sp(w), \text{TimeDTS}(w), \text{Update}(w)),\end{aligned}$$

where $\mathcal{L}', \mathcal{L}''$ are state-less functions.

We say that the aforementioned backward privacy is appropriate for d-DSE. That is inspired by the type-2 level of DSE [4]. Besides, the type-3 level discloses that multiple (w,v) additions can be associated with the corresponding deletion operations, revealing the number of identical values that may cause volume leakage.

3.4 Distinct with Volume-Hiding Security

We analyze the d-DSE leakage in consideration of volumetric attacks from the server. Previous works [42, 57] assume that the number of values associated with any single keyword should not be revealed, excluding the maximum volume. Accordingly, the leakage in *distinct search* should only infer the distinct values searched by keywords, and that cannot be pairwised with prior and current volume knowledge.

Inspired by the volume-hiding [42], we define the DwVH security, which aims to prevent adversaries from distinguishing the "signatures" of EDB. We say that a signature is defined as a sequence of uploaded keyword/value pairs $S = \{w, l(w), t(w)\}$, where $l(w)$ and $t(w)$ denote the number of distinct values (i.e., *value's type*) and the sum of value's quantities associated with keyword w, respectively. Under our security notion, the adversary is allowed to make two signatures S_0, S_1 for the challenger who randomly selects one of them $S_b, b \in \{0,1\}$ to generate the EDB. Afterward, the adversary collects leakages from update and search operations. Finally, it decides which signature is used to construct the EDB. We say that a d-DSE scheme is DwVH secure if this signature (chosen by the challenger) is indistinguishable from the adversaries, i.e., negligible advantages on the leakage from distinct values and d-DSE operations to restore volume information.

We let n denote the initial total number of keyword/value pairs in the EDB, and s represents the step $s = 1, 2, ...poly(\lambda)$. The DwVH security is defined as follows.

Definition 5 (DwVH Security) *An $\mathcal{L}$-adaptive secure d-DSE is called DwVH secure if for all $n \geq 1$ and all adversary $\mathcal{A}$ that execute at most s steps, the probability that $\mathcal{A}$ outputs 1 in* **DwVHGame**$_{\mathcal{A}}^{\mathcal{L}}((n,s),0)$ *is identical to that in* **DwVHGame**$_{\mathcal{A}}^{\mathcal{L}}((n,s),1)$. *The* **DwVHGame**$_{\mathcal{A}}^{\mathcal{L}}((n,s),b)$ *is:*

Game 1 DwVHGame$_{\mathcal{A}}^{\mathcal{L}}((n,s),b)$:

Prepare :

1: $\mathcal{A}$ generates two signatures $S_0 = \{w, l_0(w), t_0(w)\}_{w \in \mathcal{W}}$ and $S_1 = \{w, l_1(w), t_1(w)\}_{w \in \mathcal{W}}$, such that:
$l_0(w)_{w \in \mathcal{W}} < t_0(w)_{w \in \mathcal{W}} \leq n$; $l_1(w)_{w \in \mathcal{W}} < t_1(w)_{w \in \mathcal{W}} \leq n$;
$l_0(w)_{w \in \mathcal{W}} = l_1(w)_{w \in \mathcal{W}}$; $\Sigma_{w \in \mathcal{W}} t_b(w) = n$
2: $\mathcal{A}$ sends S_0 and S_1 to the challenger $\mathcal{C}$.
3: $\mathcal{C}$ computes keyword/value pairs by choosing $t_b(w)$ values for each keyword w, and each keyword w corresponds to $l_b(w)$ distinct values.
4: $\mathcal{C}$ updates all pairs and sends the corresponding leakages $\mathcal{L}_D^{Upt}$ to the adversary $\mathcal{A}$.

Queries :

1: $\mathcal{A}$ adaptively executes s step. In each step, it can perform:
• Update query: $\mathcal{A}$ adaptively performs the s-th update query about the step's keyword w_s, $\mathcal{C}$ computes $\mathcal{L}_D^{Upt}$ and sends it back.
• Search query: $\mathcal{A}$ adaptively performs the s-th search query about w_s, and $\mathcal{C}$ computes and returns $\mathcal{L}_D^{Srch}$.

Guess :

1: $\mathcal{A}$ guess the b input and outputs a bit $b' \in \{0,1\}$.

Compared to the volume-hiding definitions [42], d-DSE can fight against adversaries who enable challengers to adaptively perform update and search operations. The aim of the DwVH security is to show that the leakages from the update and distinct search do not reveal which signature is chosen to establish the EDB. In other words, the adaptive adversaries are consistent with our consideration in the FP&BP d-DSE so that they just leverage the d-DSE leakage of update and distinct search.

4 High Level of d-DSE Designed EDB

This section illustrates the practical application of d-DSE in bolstering foundational queries in EDB systems. We introduce the d-DSE query model, which encompasses update, distinct keyword, keyword, and join queries, effectively covering the

spectrum of fundamental queries pertinent to encrypted relational databases. Based on our d-DSE security model, we also conduct a leakage analysis to identify potential volumetric attacks for EDB. Note that we summarize the frequently used notations in Tab. 3 (see Appendix A).

4.1 Apply to EDB system

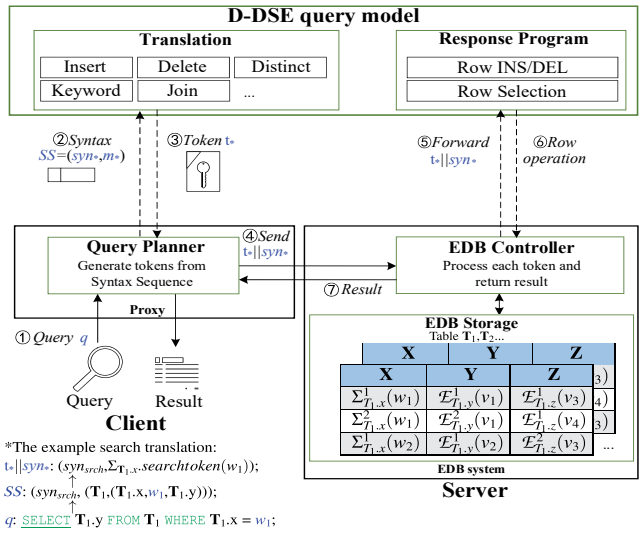


Figure 1: The high-level of d-DSE design EDB. $\Sigma^{c}_{\mathbf{T}_1.x}(w_1)$ and $\mathcal{E}^{c}_{\mathbf{T}_1.y}(v_1)$ denote that the keyword w_1 and value v_1 in table column $\mathbf{T}_1.x$ and $\mathbf{T}_1.y$ are encrypted by Σ and $\mathcal{E}$ with a counter c, respectively.

Like [45, 46], Fig. 1 depicts that d-DSE based EDB leverages the *EDB-proxy* architecture, including:

- **EDB Storage** ($\mathcal{EDB}_S$) organizes encrypted data into the table collection $\mathcal{T} = \{\mathbf{T}_1, ...\mathbf{T}_N\}$ *on disk*. Each $\mathbf{T}_*$ (e.g., $\mathbf{T}_1$.x) is represented by columns $\mathbf{T}_*.\star$ and their respective attribute types (e.g., constant-size string, integer, etc.). Tables include rows, where each row corresponds to the encrypted record $\mathbf{r} = (\Sigma^{c}_{\mathbf{T}_*.\star}(w), \mathcal{E}^{c}_{\mathbf{T}_*.\star}(v), ...)$ manufactured by the d-DSE query model.

- ***d*-DSE query model** includes a series of d-DSE constructions that support the following: (1) the *translation* from the functional equivalent syntax $SS = (syn_*, m_*)$ to the token t_*, where syn_* specifies the construction name, m_* includes the message of $\mathbf{T}_*$, $\mathbf{T}_*.\star$, and numeric or text data, and t_* contains the encrypted data generated from SS; (2) the *response program* for the EDB controller to perform the related construction operations (i.e., row insertion/deletion and selection) on $\mathcal{EDB}_S$ via the concatenation of the encrypted data and the name $t_*||syn_*$. We construct update, (distinct) keyword, and join query for the d-DSE query model (see the following subsections).

- **Query Planner** processes queries and forwards results. For a query q, the planner does the following: ① extracts it to SS (see the example in Fig. 1); ② forwards SS to the translation; ③ receives the t_*; ④ forwards the $t_*||syn_*$ to the EDB Controller and waits for response;

- **EDB Controller** processes the $t_*||syn_*$ and replies the corresponding results. Specifically, it follows operations from the response program to update rows in $\mathcal{EDB}_S$ and to copy encrypted data from $\mathcal{EDB}_S$ to the *controller's memory* for selection. We call the replicated encrypted data as $\mathtt{EDB}_C$. The EDB Controller works as: ⑤ forwards $t_*||syn_*$ to the response program; ⑥ follows the operations to update/select rows; ⑦ returns the operation results.

We leverage the EDB-proxy architecture as described in [45, 46] to map the original SQL syntax into a series of d-DSE tokens. This approach enables us to thoroughly analyze potential leakages within the N-size SS sequence (i.e., $\mathbf{SS} = (SS_1, .., SS_N)$) through the d-DSE query model. Our d-DSE query model can attain a *one-for-all* volume-hiding capability within EDB systems. We note that hereafter, we omit the subscript when describing a query in a table (e.g., $\mathbf{T}$).

4.2 Update Queries

We start with the basic update queries on table $\mathbf{T}$, e.g.

$$\texttt{INSERT INTO}\ \mathbf{T}\ (\mathbf{T}.x, \mathbf{T}.y)\ \texttt{VALUE}\ (w, v) \hookrightarrow SS_{ins} = (syn_{ins}, (\mathbf{T}, (\mathbf{T}.x, w, \mathbf{T}.y, v)))$$

$$\texttt{DELETE FROM}\ \mathbf{T}\ \texttt{WHERE}\ \mathbf{T}.x = w\ \texttt{AND}\ \mathbf{T}.y = v \hookrightarrow SS_{del} = (syn_{del}, (\mathbf{T}, (\mathbf{T}.x, w, \mathbf{T}.y, v))).$$

We can convert this syntax SS_{ins}/SS_{del} to the d-DSE Update protocol, which works for $\mathbf{T}$ (initialized by d-DSE Setup protocol) and receives keyword, value, and 'INSERT'/'DELETE' command as the (w, v, op) input from the Query Planner. We generate the update token through the Update protocol to append/delete the encrypted keyword and value in the corresponding row in $\mathbf{T}$, i.e., $\left(\Sigma_{\mathbf{T}.x}(w), \mathcal{E}_{\mathbf{T}.y}(v)\right)$, for subsequent distinct keyword queries. Note that we use the client's state $\mathbf{st}$ to mark deleted rows on $\mathcal{EDB}_S$ and then process deletion in batch for queries on $\mathtt{EDB}_C$.

Note that one should not separately analyze the leakage from the N-size update sequence (i.e., $\mathbf{op}_N = ((u_1, op_1)..(u_N, op_N))$, where u_* is the timestamp) since it is further refined by the leakage (Update(w), see Sec. 3.3) from search queries in the EDB system (see Sec 4.3).

4.3 Distinct Keyword Queries

A distinct keyword query retrieves distinct values from the column $\mathbf{T}.y$ of table $\mathbf{T}$ that corresponds to keyword w in column $\mathbf{T}.x$, i.e:

$$\texttt{SELECT DISTINCT}\ \mathbf{T}.y\ \texttt{FROM}\ \mathbf{T}\ \texttt{WHERE}\ \mathbf{T}.x = w \hookrightarrow SS_{Dsrch} = (syn_{Dsrch}, (\mathbf{T}, (\mathbf{T}.x, w, \mathbf{T}.y))).$$

Like [15, 27], the d-DSE Search protocol can implement SS_{Dsrch} by viewing w in $\mathbf{T}.x$ as keywords and data in $\mathbf{T}.y$ as values. d-DSE should allocate a local memory as Distinct

State σ^D to control the Update and Search protocols. For generating tokens t_* from SS, the Update and Search protocols should separately employ the Distinct Classifier and Distinct Constraint to pre-process SS in *translation* (Fig. 1):

• *Distinct Classifier* identifies whether the (w,v,op) from update queries is represented as distinct in retrieval, and subsequently, it tags the modified input.

• *Distinct Constraint* generates the constrained key for distinct search queries that limits the retrieval of distinct values from $\mathbf{T}.y$ based on w in $\mathbf{T}.x$.

We refer to this construction as d-KW-dDSE (distinct KeyWord queries from d-DSE). [2]

We notice that finding distinct values in a table requires a huge storage cost to store their state (i.e., distinct or repetitive). At the same time, the rows in a table do not support records that represent delete operations (e.g., the addition & deletion list of file-identifiers [20]). To this end, we apply the Bloom Filter (BF) to minimize local storage costs and the Symmetric Revocable Encryption (SRE) to enable non-interactive deletion. Our scheme for d-KW-dDSE is elaborated in Sec. 5.

Leakage analysis on d-KW-*d*DSE. We represent a N-size sequence of distinct keyword queries as $\mathbf{q}_{dis} = (\mathbf{i}, \mathbf{w})$, including the sequence of table name $\mathrm{i} \in \mathbf{i}$ for $\mathbf{T}_\mathrm{i}$ and searched keyword $w_* \in \mathbf{w}$, respectively. Akin to [24], in the EDB perspective, the leakage function [6, 8, 13] of d-KW-dDSE is:

$$\hat{\boldsymbol{L}}_{d-KW-dDSE}(\mathbf{q}_{dis}) = [(\mathrm{i}_1, \mathrm{t}_{w_1}, \mathcal{L}_D^{Srch}(w_1)) \ldots (\mathrm{i}_N, \mathrm{t}_{w_N}, \mathcal{L}_D^{Srch}(w_N))],$$

where t_{w_*} represents the token from w_*.

Similar to [15], the replacement of the Search protocol is carried out in a black-box manner. We gain the ability to detect and examine all the leakages present in the EDB, which stem from the search leakage in d-DSE.

To the best of our knowledge, passive [60] and active [64] attacks both source patterns revealed from the leakage function. We propose volumetric information [60] for d-KW-dDSE:

- **Update length pattern**, denoted as $ulen(w) = |\mathrm{Update}(\mathrm{w})|$ in $\mathcal{L}_D^{Srch}(w)$, outputs the number of updates made on keyword w.
- **Distinct response length pattern**, denoted as $drlen(w) = |\mathrm{TimeDTS}(w)|$, outputs value's type matching keyword w.
- **Query equality pattern** (a.k.a search pattern), denoted as $qeq(w_i, w_j) = \mathbf{1}(w_i = w_j)$, indicates whether two queries are targeting the same keyword. The predicate $\mathbf{1}(*)$ outputs 1 for $w_i = w_j$ and 0 otherwise.

We state that d-KW-dDSE does not have the insert/delete length pattern due to the type-2 BP of d-DSE. The file volume and response similarity pattern are both related to file retrieval instead of the constant-size distinct values protected by FP&BP.

[2] We give a detailed design for the d-DSE construction in Appendix F.

Potential attacks on d-KW-*d*DSE. d-KW-dDSE does not possess file volume (size) pattern leveraged by BVA [64] in injection attacks. For LAA, we find it difficult to perform similar volumetric attacks as in DSE [60]. Based on DwVH security, the value's quantity can express arbitrary distribution and integrate into the $ulen(w)$ of each keyword. Meanwhile, $drlen(w)$ does not reveal the sum of the value's quantities matching w, leading to the absence of the linear relationship between $drlen(w)$ and $ulen(w)$ [60].

4.4 Keyword Queries

The most common keyword query retrieves values from column $\mathbf{T}$.y for a keyword w in $\mathbf{T}$.x:

$$\texttt{SELECT}\ \mathbf{T}.y\ \texttt{FROM}\ \mathbf{T}\ \texttt{WHERE}\ \mathbf{T}.x = w \hookrightarrow SS_{srch} = (syn_{srch}, (\mathbf{T}, (\mathbf{T}.x, w, \mathbf{T}.y))).$$

The SS_{srch} is implemented by d-KW-dDSE along with certain client computations. Specifically, the client allocates a hash table to map the keyword w with the vector $\mathbf{d}$ constructed from (w,v,op) input. The dimensions of $\mathbf{d}$ record the value's quantities in the value's numerical (or lexicographical for string) order. We can first obtain the encrypted distinct values on column $\mathbf{T}$.y through d-KW-dDSE, decrypt them, and finally restore each value's quantity through $\mathbf{d}$ in their aforementioned order. The order from d-KW-dDSE can also help us to insert/auto-increment/delete a dimension in $\mathbf{d}$ for new/repetitive/deleted (w,v) input when updating encrypted data. We call this construction as KW-dDSE (KeyWord queries from d-DSE), which is a practical approach with $O(W)$ local storage cost to clients [2, 4, 20]. Note that this approach enables record recovery by treating duplicated records copied from multiple columns' data as values.

Leakage analysis on KW-*d*DSE. The KW-dDSE and d-KW-dDSE are identical except for the hash map on the honest client. For a keyword query sequence $\mathbf{q}_{kw} = (\mathbf{i}, \mathbf{w})$, the leakage function of KW-dDSE is:

$$\hat{\boldsymbol{L}}_{KW-dDSE}(\mathbf{q}_{kw}) = \hat{\boldsymbol{L}}_{d-KW-dDSE}(\mathbf{q}_{kw}).$$

Potential attacks on KW-*d*DSE. The potential attacks are identical to those on d-KW-dDSE, as the equation captures the same leakage function.

4.5 Join Queries

We implement the join query [24] between the foreign-key and primary-key, which is the fundamental query type for relational databases. A simple join query of two tables $\mathbf{T}_1/\mathbf{T}_2$ on the foreign/primary key $\mathbf{T}_1.z = \mathbf{T}_2.z$ returns all values on $\mathbf{T}_2.y$ from $\mathbf{T}_1$ and $\mathbf{T}_2$ that agree on $\mathbf{T}_1.z = \mathbf{T}_2.z$ & $\mathbf{T}_1.x = w$, i.e.,

$$\texttt{SELECT}\ \mathbf{T}_2.y\ \texttt{FROM}\ \mathbf{T}_1\ \texttt{JOIN}\ \mathbf{T}_2\ \texttt{ON}\ \mathbf{T}_1.z = \mathbf{T}_2.z\ \texttt{WHERE}\ \mathbf{T}_1.x = w. \hookrightarrow SS_{join} = (syn_{join}, (\mathbf{T}_1, \mathbf{T}_2, (\mathbf{T}_1.x, w, \mathbf{T}_1.z), (\mathbf{T}_2.z, 0, \mathbf{T}_2.y))).$$

It is straightforward to use multi KW-dDSE to capture the SS_{join}. We encrypt $\mathbf{T}_2$ with one KW-dDSE $T_2^{KW-dDSE}$ between column $\mathbf{T}_2$.y and $\mathbf{T}_2$.z and $\mathbf{T}_1$ with another $T_1^{KW-dDSE}$

between $\mathbf{T}_1$.z and $\mathbf{T}_1$.x. For the join query, we first invoke $T_1^{KW-dDSE}$ to get values $\mathbf{v}$ in $\mathbf{T}_1$.z corresponding to w in $\mathbf{T}_1$.x. Then we utilize $T_2^{KW-dDSE}$ to search values on $\mathbf{T}_2$.y by results from $T_1^{KW-dDSE}$. We call this construction as JOIN-dDSE (JOIN queries from d-DSE).

Leakage analysis on JOIN-*d*DSE. We find that the leakage of JOIN-dDSE on the EDB is caused by the process from $T_1^{KW-dDSE}$ and $T_2^{KW-dDSE}$. Since a join query consists of multi KW-dDSE processes (referred to the join equation), the leakage function of JOIN-dDSE for a sequence of join queries $\mathbf{q}_{join} = (\mathbf{i}, \mathbf{j}, \mathbf{w})$ is:

$$\hat{\boldsymbol{L}}_{JOIN-dDSE}(\mathbf{q}_{join}) = \hat{\boldsymbol{L}}_{KW-dDSE}\left((\mathbf{i}, \mathbf{w})||_l^{|\mathbf{v}|}(\mathbf{j}, v_l)\right),$$

where v_l is the value of l dimension in $\mathbf{v}$, $\mathbf{j}$ is the sequence of the second table name (i.e., $\mathbf{T}_2$), and $||_l^{|\mathbf{v}|}$ denotes concatenating the sequence of each value performing KW-dDSE on $\mathbf{T}_\mathbf{j}$.

Different from $\hat{\boldsymbol{L}}_{KW-dDSE}$, $\hat{\boldsymbol{L}}_{JOIN-dDSE}$ has an apparent search sequence between $\mathbf{T}_\mathbf{i}$ and $\mathbf{T}_\mathbf{j}$. To perform KW-dDSE on $\mathbf{T}_\mathbf{j}$, the keyword w in $\hat{\boldsymbol{L}}_{JOIN-dDSE}$ can query the same v, revealing that they have previously searched the same value in $\mathbf{T}_\mathbf{i}$.

Potential attacks on JOIN-*d*DSE. Although JOIN-dDSE resists volumetric attack inherited from KW-dDSE, it additionally leaks a 'generalized' access pattern [6] and reveals the co-occurrence matrix [36, 39] in certain situations (e.g., the plain dataset is partially exposed). To reduce the leakage, one could use the oblivious join [11,35] through indistinguishable join access, which heavily depends on Oblivious RAM (ORAM) and Bitonic-Sorted Network.

5 The BF-SRE Scheme

We describe the d-DSE scheme BF-SRE based on the Bloom Filter (BF), Symmetric Revocable Encryption (SRE), and the Forward Private DSE (FP-DSE) scheme for d-KW-dDSE. Inspired by AURA [53], the retrieval of BF-SRE is exactly the encrypted distinct values, which brings benefits for subsequent token execution. The security analysis depicts that BF-SRE attains FP&BP and DwVH security. The scheme also achieves non-interactive deletion and the sub-linear search.

Symmetric Revocable Encryption (SRE) [53]: (1) SRE.$KGen(\lambda)$ outputs the master secret key $msk = (sk, D)$ from the security parameter λ, where sk and D are the secret key and revoke structure, respectively; (2) SRE.$Enc(msk, s, t)$ outputs the ciphertext ct from msk, message s, and the tag t; (3) SRE.$Comp(D, t)$ compresses t into the revoke structure D and outputs the new one D'; (4) SRE.$ckRev(sk, D)$ takes as input sk and D and outputs sk_R; (5) SRE.$Dec(sk_R, ct)$ decrypts ct via sk_R if the related tag is not in D for sk_R.

Fig. 2 shows the Update and Search protocol of BF-SRE. BF-SRE uses BF for storing the Distinct State and the FP-DSE Σ_{add} to upload modified keyword/value pairs, while the SRE

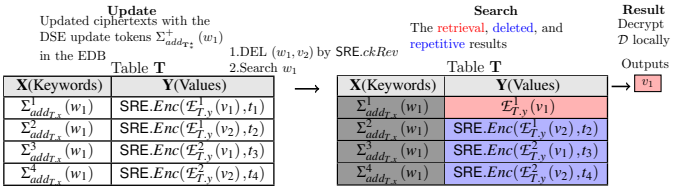


Figure 2: BF-SRE: the scheme overview. We use the blue color to represent the same revocation.

is used to revoke tags associated with deleted or repetitive values in the EDB. In Update protocol, the BF determines the real or dummy tag generation for SRE ciphertexts based on the first or repetitive inputs, respectively. The SRE ciphertext is uploaded with real tags if the BF outputs "false." Otherwise, BF-SRE uploads the SRE ciphertext with a dummy tag manufactured by revoking the corresponding SRE key. In the Search stage, if the SRE ciphertext is unrevoked, it will be decrypted by the constrained sub-key from SRE. Finally, the returned symmetric ciphertexts are decrypted locally.

The combination of BF and SRE can sufficiently deal with large-scale datasets, even for those with table structures. According the BF required bit size $b = -n \ln p / (\ln 2)^2$, where n and p are the maximum count and the tolerated false-positive probability, BF can expense a small storage cost to record whether keyword/value/op pairs appear (e.g., for $n = 2^{20}$ and $p = 10^{-5}$, b size is just 3MB [53]).

5.1 BF-SRE Description

Protocol 1 BF-SRE: Setup, bold lines 2-3 are Distinct State.

Setup(λ):

1: Initialize the DSE scheme $(\text{EDB}, \sigma, K_\Sigma) \leftarrow \Sigma_{add}.Setup(\lambda)$
2: Initialize the Bloom Filter $\mathbf{H}, \mathbf{B} \leftarrow \Phi.Gen(\lambda)$
3: Initialize empty maps MSK, UpCnt, C, D, $\text{EDB}_{\text{cache}}$
4: Randomly generate keys $K_s, K_t, K_c \xleftarrow{\$} \{0,1\}^\lambda$
5: Send EDB and the cache $\text{EDB}_{\text{cache}}$ to the server

Setup. The Setup protocol generates the encrypted database EDB, its internal state σ, secret key K_σ from a FP-DSE scheme Σ_{add}, and constructs secret keys K_s, K_t, K_c, the BF hash collection $\mathbf{H}$, the BF bit array $\mathbf{B}$, and four empty maps **MSK**, **UpCnt**, **C**, **D**. The K_c is used to encrypt values. The K_s generates the cache token tkn to get previous search results from the cache EDB ($\text{EDB}_{\text{cache}}$). The K_t is used to generate tags from inputs and **UpCnt**. The **C** counts the search on each w. The **MSK** and **D** are used to store each master key and revoked key structure associated keyword w, respectively.

Update. This protocol updates the EDB with the new internal state. At lines 1-7, the client loads the internal state on keyword w. At line 8, the client uses PRF to generate tags t and l. The t is the *real* tag for the first input of the $w||v$. The l is the *dummy* tag for the repetitive input. At line 9, the client uses the symmetric encryption to generate the retrieval s from the value v and the unique count cnt. At lines 10 and 22, the

Protocol 2 BF-SRE: Update, bold lines 11-19 are the Distinct Classifier.

Update$(K_\Sigma, \mathbf{st}, op, (w,v)$;EDB):
Client:
1: Read msk, D, i, and cnt from $\mathbf{MSK}[w]$, $\mathbf{D}[w]$, $\mathbf{C}[w]$, and $\mathbf{UpCnt}[w]$, respectively.
2: **if** msk is not initialized **then**
3: Set $msk \leftarrow \mathsf{SRE}.KGen(\lambda)$, where $msk = (sk, D)$
4: Set $\mathbf{MSK}[w] \leftarrow msk$, $\mathbf{D}[w] \leftarrow D$
5: Set $\mathbf{UpCnt}[w] \leftarrow 0$, $\mathbf{C}[w] \leftarrow 0$
6: Set $cnt \leftarrow 1$, $\mathbf{UpCnt}[w] \leftarrow cnt$
7: **end if**
8: Gen. the real/dummy tags $t \leftarrow F(K_t, w||v||0)$, $l \leftarrow F(K_t, w||v||cnt)$
9: Gen. the retrieval $s \leftarrow \mathcal{E}(K_c, v||cnt)$
10: **if** op == add **then**
11: **if** $\Phi.Check(\mathbf{H}, \mathbf{B}, t)$ **is false then**
12: **Update BF** $\Phi.Upd(\mathbf{H}, \mathbf{B}, t)$
13: $ct \leftarrow \mathbf{SRE}.Enc(msk, s, t)$
14: **Insert the EDB** $\Sigma_{add}.Update(K_\Sigma, add, w||i, (ct, t)$;EDB)
15: **else**
16: $ct \leftarrow \mathbf{SRE}.Enc(msk, s, l)$
17: **Insert the EDB** $\Sigma_{add}.Update(K_\Sigma, add, w||i, (ct, l)$;EDB)
18: **Puncture the dummy tag** $D' \leftarrow \mathbf{SRE}.Comp(D, l)$, $\mathbf{D}[w] \leftarrow D'$
19: **end if**
20: **else**
21: Puncture the real tag $D' \leftarrow \mathsf{SRE}.Comp(D, t)$, $\mathbf{D}[w] \leftarrow D'$
22: **end if**
23: $\mathbf{UpCnt}[w] \leftarrow cnt + 1$

client chooses to generate ciphertext or revoke the SRE key according to the input operation ('add' or 'del'). For addition, at line 11, the client executes the Distinct Classifier program to check whether the $w||v$ input appears for the first time. If so, the client updates the BF, generates SRE ciphertexts ct tagged t as the encrypted ciphertexts, and uploads it by the instance of DSE. Otherwise, the client generates dummy SRE ciphertexts, revokes the corresponding revoked key structure D, and uploads the dummy encrypted ciphertexts. For deletion, at line 21, the client revokes the SRE key structure corresponding to the tag t of the first input $w||v$. At line 23, the client updates the count $\mathbf{UpCnt}[w]$ for keyword w.

Protocol 3 BF-SRE: Search, bold lines 3-4 are the Distinct Constraint.

Search$(K_\Sigma, w, \mathbf{st}$;EDB):
Client:
1: Read i, sk, and D from $\mathbf{C}[w]$, $\mathbf{MSK}[w]$, and $\mathbf{D}[w]$, respectively.
2: **if** i is not valid, **then** $\perp$.
3: $sk_R \leftarrow \mathbf{SRE}.ckRev(sk, D)$
4: **Send** $(\mathbf{sk_R}, \mathbf{D})$ **and** $\mathbf{tkn} = \mathbf{F}(\mathbf{K_s}, \mathbf{w})$ **to the server**
5: $msk = (sk, D) \leftarrow \mathsf{SRE}.KGen(\lambda)$
6: Renew $\mathbf{C}[w] \leftarrow i + 1$, $\mathbf{MSK}[w] \leftarrow msk$, $\mathbf{D}[w] \leftarrow D$
Client and Server:
7: Run $\Sigma_{add}.Search(K_\Sigma, w||i, \sigma$;EDB), and the server gets the list $L = ((ct_1, t_1), (ct_2, t_2) ..., (ct_l, t_l))$
Server:
8: Set the new value list $NV \leftarrow \emptyset$
9: **for** $j \in [1, l]$ **do**
10: Get the encrypted value $V_j \leftarrow \mathsf{SRE}.Dec((sk_R, D), ct_j, t_j)$
11: **if** V_j is valid **then**
12: Update $NV \leftarrow NV \bigcup V_j$
13: **else**
14: Delete this ciphertext in EDB
15: **end if**
16: **end for**
17: Retrieve cache $OV \leftarrow \mathrm{EDB}_{\mathrm{cache}}[tkn]$
18: Return $S \leftarrow NV \bigcup OV$, and update $\mathrm{EDB}_{\mathrm{cache}}[tkn] \leftarrow S$
Client:
19: Extract each symmetric cipher s from S, and decrypt the s to get the value v from $v||cnt \leftarrow \mathcal{D}(K_c, s)$ as the search result

Search. The Search protocol finds the ciphertexts by keyword w and returns the distinct values. At lines 1-4, the client reads internal state, computes the Distinct Constraint program to get the SRE revoked key sk_R and the cache token tkn, and sends them to the server. At lines 5-6, the client renews the internal state for the next search on w. At line 7, the client and server run the Search protocol of Σ_{add}, and then the server gets the search list L. At lines 8-16, the server uses the sk_R to decrypt each ct in list L and remains the symmetric ciphertexts from the unrevoked SRE ciphertext. At line 17, the server retrieves previous results from the encrypted database cache $\mathrm{EDB}_{\mathrm{cache}}$ by tkn. At lines 18-19, the client obtains the search result S from the server and decrypts all symmetric ciphertexts to get the distinct values.

Remark on BF-SRE. To simplify the explanation, we illustrate the situation of retrieving distinct values from the specified column $\mathbf{T}.y$ based on the searched keyword in $\mathbf{T}.x$. To retrieve distinct values on other columns in $\mathbf{T}$, we do the following: (1) extend the value input as a vector (i.e., v to $\mathbf{v}$); (2) expand the dimension of $\mathbf{B}$ (i.e., $\mathbf{B}[w]$ to $\mathbf{B}[w].\star$) to record more state on other columns; (3) expand the dimension of $\mathbf{D}$ (i.e., $\mathbf{D}[w].\star$) for their corresponding revoked key structure; and (4) finally extend ct. After that, we switch to the right column's revoked key structure in $\mathbf{D}[w].\star$ for distinct values on arbitrary columns in $\mathbf{T}$. More functional extensions for BF-SRE are possible through replacing Σ_{add} with other FP-DSE schemes like range search [67].

5.2 Analysis on BF-SRE

Correctness. The BF-SRE scheme uses the FP-DSE to upload the modified keyword/value pairs in encrypted databases. With the polynomial-time algorithm of the PRF F, it always outputs a tkn from K_s. The BF's possibility of losing correctness is false positive and acceptable [53]. Hence, our scheme can correctly update and search distinct values.

Security Analysis. BF-SRE securely generates the ciphertexts, search tokens attached to the FP-DSE, the SRE, the PRF, and the symmetric encryption. As for the Update stage, the client modifies the input by **UpCnt** and the Distinct State, and the ciphertext is uploaded by the FP-DSE Σ_{add}. With regard to the Search stage, the client uploads the revoked SRE key to exactly retrieve the values attached to the real tags, which does not show auxiliary information about the volume.

Theorem 1 (Adaptive Security of BF-SRE) *Let F (the PRF) with a specific key be modelled as the random oracle $\mathcal{H}_F$, $\mathcal{L}_D = \left(\mathcal{L}_D^{Upt}, \mathcal{L}_D^{Srch}\right)$ is defined as:*

$$\mathcal{L}_D^{Upt}(w, v, op) = add$$
$$\mathcal{L}_D^{Srch}(w) = sp(w), \mathrm{TimeDTS}(w), \mathrm{Update}(w),$$

BF-SRE is $\mathcal{L}_D$-adaptively-secure.

Proof Sketch. The proof is conducted in the game hop like AURA [53]. We gradually replace the real cryptographic tools

used in the BF-SRE scheme with the bookkeeping tables or the corresponding simulators and obtain the BF-SRE simulator. The addition list L_{add} and the deletion set D are used in the $\mathcal{L}_D^{Srch}$ leakage function to retrieve the search result. We note that the BF determines the workflow of the simulator and does not affect the indistinguishability between each game hop.

Theorem 2 (DwVH Security of BF-SRE) *The leakage function of BF-SRE* $\mathcal{L}_D = (\mathcal{L}_D^{Upt}, \mathcal{L}_D^{Srch})$ *is Distinct with Volume-Hiding.*

Proof Sketch. The proof is analyzed from the leakage in the BF-SRE forward and backward privacy. To prove that $\mathcal{L}_D$ is Distinct with Volume-Hiding, we need to construct two signatures with the same size and the upper bound of maximum volume. Note that $\mathcal{L}_D^{Upt}$ consists the addition operations, and $\mathcal{L}_D^{Srch}$ includes value's type and timestamps of updates. The $\mathcal{L}_D^{Srch}$ is independent of the input keyword/value pairs because the real and dummy tags generation can always maintain identical response on value's type under the two signatures. Therefore, the leakage function reveals nothing about the volume. Both proofs are given in Appendix E.

Complexity. In the Setup stage, BF-SRE spends a constant time initializing Σ_{add}, internal state, and secret keys. In the Update stage, it consumes a constant time to update the internal state, execute Σ_{add}, test the BF, and generate ciphertexts. Hence, the Update protocol computation complexity is $O(1)$ with a one-way interaction and an $O(1)$ length ciphertext. In the Search stage, BF-SRE spends a constant time performing Σ_{add} and spends $O(a_w)$ time on search results, where a_w denotes all ciphertexts found by Σ_{add}. Hence, the Search protocol computation complexity is set as $O(n_w)$ with a one-way interaction, where n_w denotes the number of returned distinct values.

5.3 Optimization

In BF-SRE, the cryptographic tools determine the actual computation and communication costs. Recall that in the Search protocol, BF-SRE uses the DSE scheme to find all matched ciphertexts. A possible solution to improve the efficiency of the Search is to leverage the parallel Forward Private DSE [30, 33].

On the other hand, the SRE decryption affects the efficiency of finding the distinct values. In practice, SRE requires significant computational cost to determine the derived sub-key corresponding to SRE ciphertext. Fortunately, we can identify and implement the *Greedy* and *Default* optimization for the SRE decryption:

• *Greedy*: Temporarily store the sub-keys that are not applied in the previous SRE decryption and try using them in subsequent decryption. In practice, we can use a stack to store unused sub-keys, enabling a systematic approach to ejecting the keys one by one for the decryption of SRE ciphertexts. After decryption, the used sub-keys are discarded, while the (new) derivatives are added to the stack.

• *Default*: Pre-compute the sub-key space size of SRE decryption through the keyword frequency in the target dataset. In practice, we can utilize the estimated keyword frequencies, which are close to the real ones [55], to set the appropriate limitation of revoke operation corresponding to each keyword. After that, SRE decryption reduces the total computation of sub-keys, thereby achieving faster decryption speed.

Note we combine the two optimizations for BF-SRE.

6 Evaluation

We first introduce datasets with different keyword/value distributions and experimental setup. Then, we evaluate the comparison between BF-SRE and MITRA* [20], AURA [53], SEAL [15], and ShieldDB [55]. Our scheme, under equivalent security parameters, exhibits comparable time costs and significant communication improvement. Our codes are publicly available in https://github.com/jd89j12dsa/ddse.

6.1 Dataset

Chicago Crimes Reports.[3] The Crime dataset is suitable for keyword/value pairs (in the context of SQL and EDB) [9, 15]. It includes 7,989,987 keyword/value pairs extracted from the reports (spanning from 1-1-1999 to 2-4-2024). We use street names and the corresponding IUCR codes as keywords and values in the Crime table, respectively. For testing join queries, we extract the whole IUCR table to search the PRIMARY_DESCRIPTION by IUCR codes from the Crime results.

Wikipedia.[4] This is a large-scale document dataset for DSE evaluation [12, 33]. Through the Wikipedia extractor and Python NLTK package [33], we collect 4,565,948 pairs in our table structure, and we aim to find the distinct words by document names.

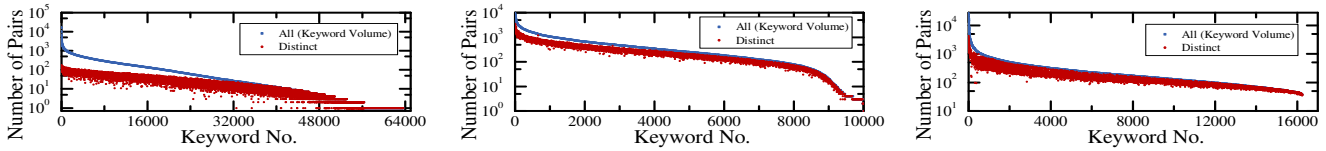
Enron.[5] Enron email dataset [33, 65] stores the texts converted from its email system. We use NLTK package and Porterstemmer [47] to extract 5,190,199 pairs, aiming to identify distinct words through email names.

Fig. 3 illustrates the number of keyword/value pairs associated with each keyword in descending order. We count the number of all keyword/value pairs as *Keyword Volume*. The repetitive-percentage/keyword-space/highest-keyword-volume of the Crime, Wikipedia, and Enron dataset are 80.15%/63,659/16,644, 42.04%/10,000/9,738, and 45.53%/16,241/26,946, respectively.

6.2 Experimental Setup

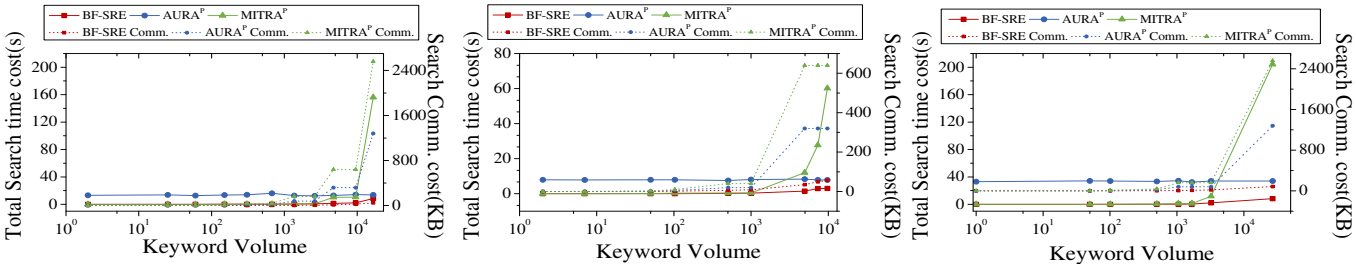
Our test platform is Intel® Xeon Gold 5120 CPU @ 2.20GHz, 128GB, Dell RERC H730 Adp SCSI Disk Device, Windows Server 2016 Standard. In our Python (v3.60) code, we utilize

[3]This dataset is available at: https://data.cityofchicago.org/
[4]The Wikipedia: https://dumps.wikimedia.org/
[5]Enron email dataset:https://www.cs.cmu.edu/~enron/



(a) Crime **(b)** Wikipedia **(c)** Enron

Figure 3: The number of keyword/value pairs associated with each keyword on Crime, Wikipedia and Enron dataset. The blue and red nodes represent the number of all and distinct keyword/value pairs, respectively.



(a) Crime **(b)** Wikipedia **(c)** Enron

Figure 4: Comparison of the **total** search **time and communication** costs of BF-SRE, AURA^P, and MITRA^P **without deletion**.

the PYmysql package to perform at most 100 parallel queries on MySQL (Ver 14.14 Distrib 5.7.33).

We first compare the performance of BF-SRE with MITRA^P and AURA^P. Note BF-SRE uses DIANA [4] as the Forward Private DSE implementation. Our implementation makes Python modules from the SRE [53] source code with the *greedy* and *default* optimization mentioned in Sec. 5.3. We also implement MITRA^P and AURA^P from MITRA* [20] and AURA [53], along with the SEAL's adjustable padding strategy [15] for fair comparisons. Specifically, when initializing the EDB, we ensure that MITRA^P and AURA^P pad each keyword with dummy pairs until the volume of each keyword reaches the exponentiation of $x = 4$, respectively. For correctness, we prepare a translation map between keyword/value pairs and keyword/id pairs to restore search results mentioned in Sec. 1.

Then, we test the time and communication costs of BF-SRE, SEAL, and ShieldDB when processing queries in SQL syntax. SEAL is implemented through the tree-based ORAM [20][6] with the factor $a = 20, x = 4$. For ShieldDB in NL mode ($\alpha = 256$), we transform keyword/value datasets to the related file dataset and further compose keyword clusters uploading the stream data. We say that they can perform the queries via our basic Query Planner established from the PYmysql package and the transformation in Sec. 1.

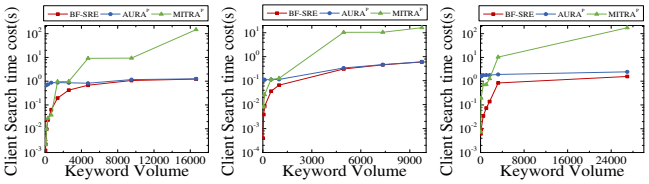
6.3 Compare with DSE

For BF-SRE, AURA^P, and MITRA^P, we document total/client search time, communication, and highest-volume keyword search time costs in deletion. Based on the equation in Sec. 5, the BF's size reaches an appropriate setting to delete on the highest-volume keyword, preserving the correctness of BF-SRE and AURA^P.

[6]The tree-based ORAM: https://github.com/jgharehchamani/SSE

6.3.1 Search performance

Total search time and communication costs. Fig. 4 illustrates that BF-SRE significantly outperforms AURA^P and MITRA^P in time and communication costs when the Keyword Volume > 10^3. Particularly, with regards to time costs, in Fig. 4c, BF-SRE exhibits a slower increase in cost, extending up to 8.25 seconds. This is more efficient, with a 4.12x and 29.27x reduction in time compared to AURA^P and MITRA^P. Meanwhile, the communication costs for BF-SRE are 83.82 KB with 15.27x and 30.54x advantage over AURA^P and MITRA^P. Both compared DSE schemes display significant ladder like mutations (of SEAL) in communication cost. All trends of Fig. 4a,4b,4c have similar increments. This result shows that the padding strategy requires AURA^P to initialize the large size BF to log the deletion of dummy pairs, while MITRA^P has to perform more "clean-up" operations (i.e., removing deleted pairs and re-encrypting the remaining pairs).

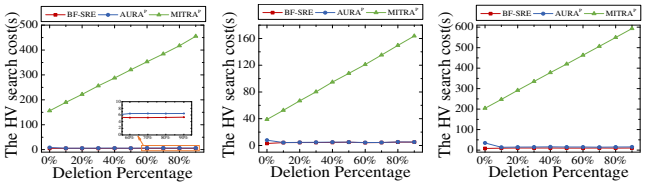


(a) Crime **(b)** Wikipedia **(c)** Enron

Figure 5: Comparison of search **time** costs of BF-SRE, AURA^P, and MITRA^P on **client without deletion**.

Search time cost on client. Fig. 5 shows that BF-SRE outperforms MITRA^P and presents a small advantage over AURA^P. For example, in Fig. 5a, the cost gap between BF-SRE and AURA^P is roughly 0.64-0.03s. We state that BF-SRE does not process dummy data while searching, thereby improving the client's performance.

Highest-Volume (HV) Search time costs. Fig. 6 provides



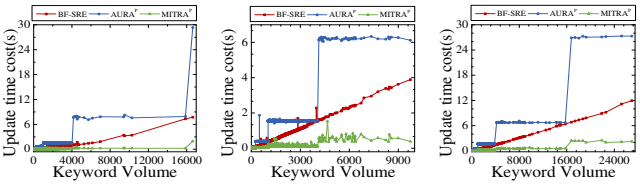
(a) Crime **(b)** Wikipedia **(c)** Enron

Figure 6: Comparison of **search time** costs of BF-SRE, AURA^P, and MITRA^P on the highest-volume (HV) keyword **with deletion** percentage (namely, delete 0-90% pairs).

the fact that BF-SRE is well-perform across all datasets. We notice that the costs associated with BF-SRE exhibit a slight increase in correlation with the rise in the deletion percentage. In Fig. 6b,6c, the costs slowly increase from 2.96s and 8.14s to 4.97s and 10.02s, respectively. The costs of MITRA^P climb up abruptly as its deletion requires traversing and re-encrypting all dummy data (under padding). AURA^P presents a declining cost trend on the costs but still cannot outperform BF-SRE. In Fig. 6c, the cost of AURA^P decreases from 34.31s to 15.33s since the deletion operations indirectly reduce its cost on decryption key generation.

6.3.2 Update performance

We compare update time costs for BF-SRE, AURA^P, and MITRA^P. Due to AURA^P and MITRA^P inheriting the batch update of padding strategies, we record the costs based on Keyword Volume. For BF-SRE, we record the average addition cost of all pairs associated with the keyword at a specific Keyword Volume. We also record the client storage cost when adding keyword/value pairs in the sequence of Keyword No. In other words, for the Crime dataset, we add the corresponding pairs in the order of Keyword No.1 to No.63659 shown in Fig. 3a.

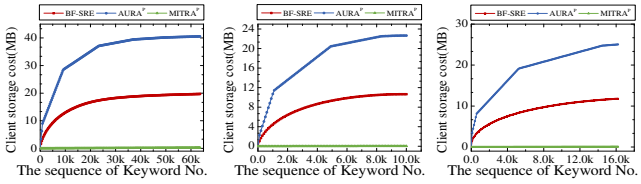


(a) Crime **(b)** Wikipedia **(c)** Enron

Figure 7: Comparison of **update time** costs of BF-SRE, AURA^P, and MITRA^P.

Update time costs. Fig. 7 depicts that the costs of BF-SRE are at the same magnitude in all datasets. BF-SRE yields linear costs that grow in proportion to the Keyword Volume, while the AURA^P and MITRA^P both experience step-wise increases. In detail, for Fig. 7a, the costs of BF-SRE, AURA^P, and MITRA^P climb from $1.52{\cdot}10^{-4}$s, $2.72{\cdot}10^{-4}$s, and $8.8{\cdot}10^{-5}$s to 7.62s, 29.24s, and 1.94s, respectively. Both AURA^P and MITRA^P have the mutation at the Keyword Volume of 4096 and 15868, which is due to the use of SEAL's adjustable padding.

Client storage costs. Fig. 8 illustrates that the costs for BF-SRE fall between those of AURA^P and MITRA^P in all datasets.



(a) Crime **(b)** Wikipedia **(c)** Enron

Figure 8: Comparison of **client storage** costs of BF-SRE, AURA^P, and MITRA^P for update.

Due to the *Default* optimization, the BF-SRE costs eventually trend toward MITRA^P, roughly 1.47x less than AURA^P in Fig. 8a, 8b, 8c, respectively. In Fig. 8a, AURA^P exhibits noticeable inflections when the sequence of Keyword No. reaches 899, 9192, and 23351 (corresponding to Keyword Volumes of 1023, 255 and 63) due to the SEAL's adjustable padding.

6.4 Compare with Padding Strategy

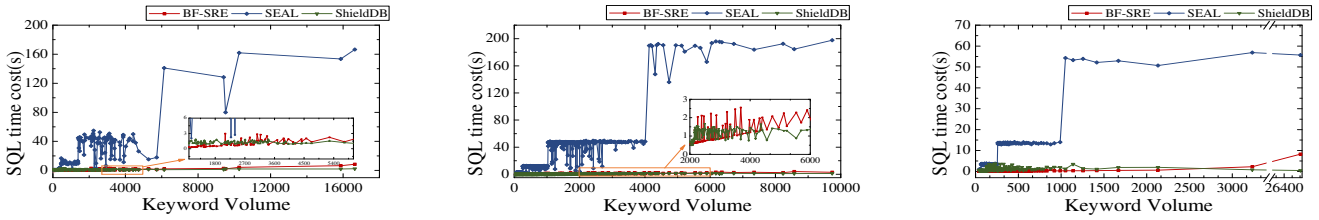
We test the time and communication costs for keyword and join queries in SQL syntax. As mentioned in Sec. 4, we first set up the constructions of BF-SRE, SEAL, and ShieldDB, respectively. Then, we use the Query Planner to extract original queries for each construction to retrieve values from the EDB.

For join queries, we use the Crime dataset to test time and communication costs about its Keyword Volume. We use Query Planner to request retrieving the relevant IUCR code through keyword (street name, see Sec. 6.1) in the Crime table, and then leverage each IUCR code result to find the PRIMARY_DESCRIPTION in the IUCR table. Finally, we let the planner restore the query combined from the two-stage results and log the related costs.

Keyword query time cost. Fig. 9 presents the costs of BF-SRE, SEAL, and ShieldDB. From the query results covering the entire keyword space, we see that BF-SRE provides advantage, nearly 10.89-180x, for SEAL when the Keyword Volume $> 10^3$. Its cost slightly exceeds that of ShieldDB when the Keyword Volume is greater than 4728, 4219, and 2712 in the Crime, Wikipedia, and Enron dataset, respectively. From our analyze, the numbers of keywords below these thresholds are accounted for approximately 95.0%, 96.5%, and 98.5%, revealing that the BF-SRE's cost is competitive among large keyword spaces. Note that some 'spikes' come from batch read operations [20] and cache retrieval [55] in SEAL and ShieldDB, respectively.

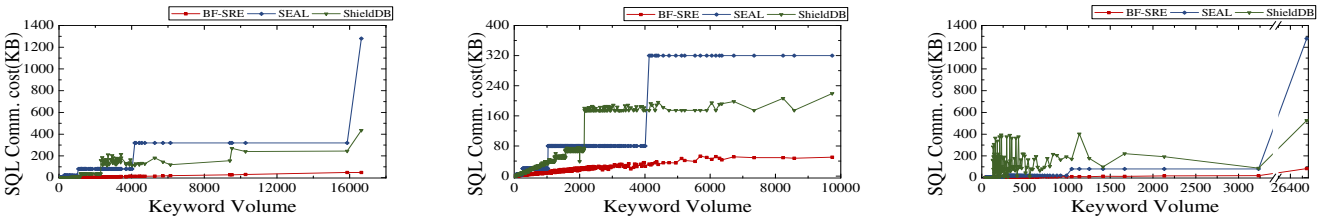
Keyword query communication costs. In Fig. 10, BF-SRE consumes much less communication costs than the padding strategies. Specifically, it reduces the highest-volume keyword cost up to 53.14x, 6.36x, and 15.27x on the Crime, Wikipedia, and Enron dataset, respectively. We recall that BF-SRE mainly concentrates on the retrieval of the values, while the padding has to deal with the dummy data.

Join query cost. Fig. 11 shows that BF-SRE performs the



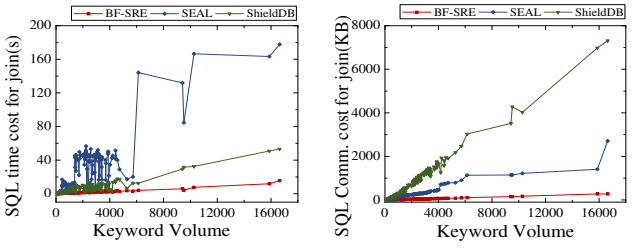
(a) Crime **(b)** Wikipedia **(c)** Enron

Figure 9: Comparison of the SQL **time** cost for keyword queries on BF-SRE, SEAL, and ShieldDB.



(a) Crime **(b)** Wikipedia **(c)** Enron

Figure 10: Comparison of the SQL **communication** cost for keyword queries on BF-SRE, SEAL, and ShieldDB.



(a) Time costs **(b)** Communication costs

Figure 11: Comparison of **time and communication** costs for SQL join queries on Crime.

best in the Crime dataset. Compared to executing SQL keyword queries, BF-SRE does not require a significant rise in the metrics. In contrast, SEAL and ShieldDB repeatedly eliminate the dummy data from the IUCR table locally, thereby sharply increasing their costs. The costs associated with BF-SRE are roughly 26.6x lower than those of ShieldDB w.r.t. the highest-volume keyword.

When considering overall costs, BF-SRE stands as a competitive performer in comparison to trivial-transformed instances from DSE. Moreover, it provides comparable time costs to the padding strategies while delivering significant advantages. It demonstrates the ability to efficiently handle various search queries and offers practical communication costs under the tested security parameters.

7 Discussions

Dataset. We notice that there are still a few datasets that have been used for DSE evaluations, e.g., Apache Lucene [64]. We can apply them to locate distinct words by document names, similar to the approach used in Wikipedia or Enron. We argue that using these datasets will not seriously affect the performance of our proposals because Crime/Wikipedia/Enron has a familiar data distribution to other table/document/email datasets [33].

Highest-volume keyword search in deletion. One may argue that using "highest-volume keyword" may not yield extensive results. It is important to note that the search results obtained from representative keywords exhibit minimal fluctuation. This is because the search process for these keywords consumes sufficient time and ensures accurate results. These keywords are more frequently used in practical databases and we use them to clearly highlight the advantage of our proposal. We note that interested readers may choose to use less-frequent keywords in the evaluation.

Other SQL databases. We use MySQL to store the ciphertexts of the EDB. We say that interested readers may use other SQL databases, such as PostgreSQL and Microsoft SQL Server, to implement the d-DSE schemes via the Python interfaces. The parallel capability of SQL databases is not a crucial aspect to consider when choosing among them, as their performance is largely dependent on disk storage.

The d-DSE storage cost. BF-SRE needs more client storage than MITRA^P, which is required by the underlying SRE revoke structure. We can apply other DSE schemes as building blocks consuming lower client storage.[7]

Threat from access pattern. Similarly to JOIN-dDSE, d-KW-dDSE could potentially be vulnerable to access pattern leakage [1], which is a concern associated with practical DSE constructions [12, 14, 43, 53]. In certain scenarios, the retrieval process could involve accessing memory, such as acquiring a constant-size file-identifier from a column, which could inadvertently reveal access patterns. One straightforward strategy to mitigate this issue is to combine d-KW-dDSE with ORAM [20] to obscure memory access, albeit at extra costs of time and storage. Specifically, we can first retrieve identifiers via d-KW-dDSE and then employ ORAM to re-

[7] In the full version [37], we provide an example based on MITRA and the Inner Product Encryption [34], obtaining distinct values via the distinct inner products between ciphertexts and search tokens.

trieve each identifier's data. When processing queries in batch, we leverage the multi-path ORAM (e.g., OBI [59]) to mitigate the high throughput.

Against frequency attacks. Although we mainly focus on volume leakage, a recent research [60] alerts the frequency(-matching) attack. The success of the attack hinges on the diversity of the query frequency that is exposed in BF-SRE and prior FP&BP DSE schemes [20, 53] through persistent frequency detection. To mitigate the attack, we can apply a general countermeasure - frequency-smoothing - also used in PANCAKE [21], on the top of d-DSE. It is compatible for combining d-DSE and frequency smoothing as: 1) they both leverage keyword/value pairs (KV pairs in their description); 2) their contexts (i.e., static and dynamic frequency distributions) are consistent.

Ciphertext de-duplication. Recall that ciphertext de-duplication [5, 38, 49, 63] is to eliminate the repetitive ciphertexts. In particular, this approach, inherited from convergent encryption [38], emphasizes the elimination of ciphertexts with identical contents across multiple clients in order to minimize storage costs. But this technique does not focus on ensuring secure value searches.

8 Conclusion

We explore the distinct search and propose d-DSE. Following the concept and definition of d-DSE, we propose the d-DSE designed EDB and develop BF-SRE, which satisfies the forward and backward privacy and DwVH security. We conduct extensive experiments to highlight the practical performance of our designs in run time, communication, storage, and effectiveness on volume leakage.

Acknowledgments

We would like to thank the shepherd and the anonymous reviewers for their valuable comments. This work was partly supported by the National Key Research and Development Program of China under Grant No. 2022YFB4501500, the National Natural Science Foundation of China under Grant No. 62372201 and No. 62272186, the Innovation Project of Jinyinhu Laboratory under Grant No. 2023JYH010103, and the European Union's Horizon Europe Research and Innovation Programme under Grant No. 101073920 (TENSOR), No. 101070052 (TANGO), and No. 101070627 (REWIRE).

References

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

[2] Raphael Bost. Sophos: Forward secure searchable encryption. In *CCS*, pages 1143–1154, 2016.

[3] Raphael Bost and Pierre-Alain Fouque. Thwarting leakage abuse attacks against searchable encryption–a formal approach and applications to database padding. *Cryptology ePrint Archive*, 2017.

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

[5] Jan Camenisch, Angelo De Caro, Esha Ghosh, and Alessandro Sorniotti. Oblivious prf on committed vector inputs and application to deduplication of encrypted data. In *FC*, pages 337–356, 2019.

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

[7] 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.

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

[9] David Cash, Ruth Ng, and Adam Rivkin. Improved structured encryption for sql databases via hybrid indexing. In *ACNS*, pages 480–510, 2021.

[10] Javad Ghareh Chamani, Dimitrios Papadopoulos, Mohammadamin Karbasforushan, and Ioannis Demertzis. Dynamic searchable encryption with optimal search in the presence of deletions. In *USENIX Security*, pages 2425–2442, 2022.

[11] Zhao Chang, Dong Xie, Sheng Wang, and Feifei Li. Towards practical oblivious join. In *SIGMOD*, pages 803–817, 2022.

[12] Tianyang Chen, Peng Xu, Stjepan Picek, Bo Luo, Willy Susilo, Hai Jin, and Kaitai Liang. The power of bamboo: On the post-compromise security for searchable symmetric encryption. In *NDSS*, 2023.

[13] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934, 2011.

[14] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. Dynamic searchable encryption with small client storage. In *NDSS*, 2020.

[15] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. Seal: Attack mitigation for encrypted databases via adjustable leakage. In *USENIX Security*, pages 2433–2450, 2020.

[16] Ioannis Demertzis, Rajdeep Talapatra, and Charalampos Papamanthou. Efficient searchable encryption through compression. *Proceedings of the VLDB Endowment*, 11(11):1729–1741, 2018.

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

[18] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K. Cunningham. Sok: Cryptographically protected database search. In *IEEE S&P*, pages 172–191, 2017.

[19] Marilyn George, Seny Kamara, and Tarik Moataz. Structured encryption and dynamic leakage suppression. In *EUROCRYPT*, pages 370–396, 2021.

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

[21] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. Pancake: Frequency smoothing for encrypted data stores. In *USENIX Security*, pages 2451–2468, 2020.

[22] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *CCS*, pages 315–331, 2018.

[23] Florian Hahn and Florian Kerschbaum. Searchable encryption with secure and efficient updates. In *CCS*, pages 310–320, 2014.

[24] Florian Hahn, Nicolas Loza, and Florian Kerschbaum. Joins over encrypted data with fine granular security. In *IEEE ICDE*, pages 674–685, 2019.

[25] Yanxue Jia, Shi-Feng Sun, Hong-Sheng Zhou, Jiajun Du, and Dawu Gu. Shuffle-based private set union: Faster and more secure. In *USENIX Security*, pages 2947–2964, 2022.

[26] Charanjit Jutla and Sikhar Patranabis. Efficient searchable symmetric encryption for join queries. In *ASIACRYPT*, pages 304–333, 2022.

[27] Seny Kamara and Tarik Moataz. Sql on structurally-encrypted databases. In *ASIACRYPT*, pages 149–180, 2018.

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

[29] Seny Kamara, Tarik Moataz, and Olya Ohrimenko. Structured encryption and leakage suppression. In *CRYPTO*, pages 339–370, 2018.

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

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

[32] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. Generic attacks on secure outsourced databases. In *CCS*, pages 1329–1340, 2016.

[33] Kee Sung Kim, Minkyu Kim, Dongsoo Lee, Je Hong Park, and Woo-Hwan Kim. Forward secure dynamic searchable symmetric encryption with efficient updates. In *CCS*, pages 1449–1463, 2017.

[34] Sam Kim, Kevin Lewi, Avradip Mandal, Hart Montgomery, Arnab Roy, and David J Wu. Function-hiding inner product encryption is practical. In *SCN*, pages 544–562, 2018.

[35] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. Efficient oblivious database joins. *Proceedings of the VLDB Endowment*, 13(12):2132–2145, 2020.

[36] Steven Lambregts, Huanhuan Chen, Jianting Ning, and Kaitai Liang. Val: Volume and access pattern leakage-abuse attack with leaked documents. In *ESORICS*, pages 653–676, 2022.

[37] Dongli Liu, Wei Wang, Peng Xu, Laurence T. Yang, Bo Luo, and Kaitai Liang. d-dse: Distinct dynamic searchable encryption resisting volume leakage in encrypted databases. *arXiv preprint, arXiv:2403.01182*, 2024. https://arxiv.org/abs/2403.01182.

[38] Jian Liu, Nadarajah Asokan, and Benny Pinkas. Secure deduplication of encrypted data without additional independent servers. In *CCS*, pages 874–885, 2015.

[39] Jianting Ning, Xinyi Huang, Geong Sen Poh, Jiaming Yuan, Yingjiu Li, Jian Weng, and Robert H Deng. Leap: leakage-abuse attack on efficiently deployable, efficiently searchable encryption with partially known dataset. In *CCS*, pages 2307–2320, 2021.

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

[41] Simon Oya and Florian Kerschbaum. Ihop: Improved statistical query recovery against searchable symmetric encryption through quadratic optimization. In *USENIX Security*, pages 2407–2424, 2022.

[42] 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 *CCS*, pages 79–93, 2019.

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

[44] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on ot extension. In *USENIX Security*, pages 797–812, 2014.

[45] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: An encrypted database using semantically secure encryption. *Proceedings of the VLDB Endowment*, 12(11):1664–1678, 2019.

[46] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP*, pages 85–100, 2011.

[47] Martin F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[48] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *IEEE S&P*, pages 264–278, 2018.

[49] Yanjing Ren, Jingwei Li, Zuoru Yang, Patrick P. C. Lee, and Xiaosong Zhang. Accelerating encrypted deduplication via sgx. In *USENIX ATC*, pages 957–971, 2021.

[50] Masoumeh Shafieinejad, Suraj Gupta, Jin Yang Liu, Koray Karabina, and Florian Kerschbaum. Equi-joins over encrypted data for series of queries. In *IEEE ICDE*, pages 1635–1648, 2022.

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

[52] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *NDSS*, 2014.

[53] Shi-Feng Sun, Ron Steinfeld, Shangqi Lai, Xingliang Yuan, Amin Sakzad, Joseph Liu, Surya Nepal, and Dawu Gu. Practical non-interactive searchable encryption with forward and backward privacy. In *NDSS*, 2021.

[54] Shi-Feng 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 *CCS*, pages 763–780, 2018.

[55] Viet Vo, Xingliang Yuan, Shi-Feng Sun, Joseph K. Liu, Surya Nepal, and Cong Wang. Shielddb: An encrypted document database with padding countermeasures. *IEEE TKDE*, 35(4):4236–4252, 2023.

[56] Jiafan Wang and Sherman S. M. Chow. Omnes pro uno: Practical Multi-Writer encrypted database. In *USENIX Security*, pages 2371–2388, 2022.

[57] Jianfeng Wang, Shi-Feng Sun, Tianci Li, Saiyu Qi, and Xiaofeng Chen. Practical volume-hiding encrypted multi-maps with optimal overhead and beyond. In *CCS*, pages 2825–2839, 2022.

[58] Xingchen Wang and Yunlei Zhao. Order-revealing encryption: file-injection attack and forward security. In *ESORICS*, pages 101–121, 2018.

[59] Zhiqiang Wu and Rui Li. Obi: a multi-path oblivious ram for forward-and-backward-secure searchable encryption. In *NDSS*, 2023.

[60] Lei Xu, Leqian Zheng, Chengzhi Xu, Xingliang Yuan, and Cong Wang. Leakage-abuse attacks against forward and backward private searchable symmetric encryption. In *CCS*, pages 3003–3017, 2023.

[61] Min Xu, Armin Namavari, David Cash, and Thomas Ristenpart. Searching encrypted data with size-locked indexes. In *USENIX Security*, pages 4025–4042, 2021.

[62] Peng Xu, Willy Susilo, Wei Wang, Tianyang Chen, Qianhong Wu, Kaitai Liang, and Hai Jin. Rose: Robust searchable encryption with forward and backward security. *IEEE TIFS*, 17:1115–1130, 2022.

[63] Zuoru Yang, Jingwei Li, and Patrick P. C. Lee. Secure and lightweight deduplicated storage via shielded Deduplication-Before-Encryption. In *USENIX ATC*, pages 37–52, 2022.

[64] Xianglong Zhang, Wei Wang, Peng Xu, Laurence T. Yang, and Kaitai Liang. High recovery with fewer injections: Practical binary volumetric injection attacks against dynamic searchable encryption. In *USENIX Security*, pages 5953–5970, 2023.

[65] 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*, pages 707–720, 2016.

[66] Yongjun Zhao, Huaxiong Wang, and Kwok-Yan Lam. Volume-hiding dynamic searchable symmetric encryption with forward and backward privacy. Cryptology ePrint Archive, Paper 2021/786, 2021.

[67] Cong Zuo, Shifeng Sun, Joseph K. Liu, Jun Shao, and Josef Pieprzyk. Dynamic searchable symmetric encryption schemes supporting range queries with forward (and backward) security. In *ESORICS*, pages 228–246, 2018.

A Summary of Notations & Concepts

Table 3: Notations for d-DSE.

d-DSE designed EDB	Description
$\mathcal{EDB}_S$	The collection of encrypted tables stored on disk
$\mathtt{EDB}_C$	The replicated encrypted data from $\mathcal{EDB}_S$ on memory
q	The query
$\mathbf{q}_s$	The query sequence
SS	The functionally equivalent syntax
$\mathbf{t}_s$	The encrypted data generated from SS for queries
syn_s	The specific d-DSE construction name
$\mathcal{T}$	The table collection in EDB storage
$\mathbf{T}_s$	The table name
$\mathbf{T}_s.\star$	The table column
$\hat{\mathbf{L}}_s$	The leakage function of constructions
$\mathbf{d}$	The vector containing value's quantities
$ulen(w)$	The update length pattern
$drlen(w)$	The distinct response length pattern
$qeq(w)$	The query equality pattern
BF-SRE	**Description**
EDB	The encrypted database
$\text{EDB}_{\text{cache}}$	The cache encrypted database
$\mathbf{C}[w]$	The map counts the number of search on keyword w
$\mathbf{MSK}[w]$	The map records the master secret key about w
$\mathbf{UpCnt}[w]$	The map counts the update on w
$\mathbf{D}[w]$	The map records revoked key structure about w
K_c	The encryption key for values
K_s	The encryption key for $\text{EDB}_{\text{cache}}$
K_t	The encryption key for tag
t	The real tag
l	The dummy tag
$\mathbf{H}$	The Bloom Filter hash collection and bit array
$\mathbf{B}$	The Bloom Filter bit array
Σ_{add}	The Forward Private DSE scheme
Φ	The Bloom Filter scheme
$\mathcal{E}\&\mathcal{D}$	The encryption and decryption algorithm of the symmetric encryption

Symmetric encryption. Given a security parameter $\lambda \in \mathbb{N}$, message space $\mathcal{M} = \{0,1\}^*$, ciphertext space $\mathcal{C} = \{0,1\}^*$ and the key space $\mathcal{K} = \{0,1\}^\lambda$, a symmetric encryption scheme consists of two algorithm $(\mathcal{E}, \mathcal{D})$ the syntax in Algorithm 1.

Algorithm 1 Symmetric Encryption

- $\mathcal{E}(k,m)$: Input a symmetric key $k \in \mathcal{K}$ and a message $m \in \mathcal{M}$, output a ciphertext $c \in \mathcal{C}$.
- $\mathcal{D}(k,c)$: Input a symmetric key $k \in \mathcal{K}$ and a ciphertext $c \in \mathcal{C}$, recover a message $m \in \mathcal{M}$.

* For correctness $\mathcal{E}, \mathcal{D}$ should always make sense. For security, the requirement is INDistinguishability against Chosen Plaintext Attack (IND-CPA).

Bloom Filter [53]. A Bloom Filter (BF) Φ is a probabilistic data structure, which can rapidly and space-efficiently perform set membership test. A BF consists of three polynomial-time algorithms shown in Algorithm 2.

Algorithm 2 Bloom Filter

- $Gen(\lambda)$: It takes λ parsed as two integers $b, h \in \mathrm{N}$, and samples a collection of universal hash functions $\mathbf{H} = \{H_j\}_{j\in[h]}$, where $H_j : \{0,1\}^* \to [b]$, $[b]$ denotes a finite set. Finally, it outputs $\mathbf{H}$ and an initial b-bit array $\mathbf{B} = 0^b$ with each bit $\mathbf{B}[i]$ for $i \in [b]$ set to 0.
- $Upd(\mathbf{H}, \mathbf{B}, x)$: It takes $\mathbf{H} = \{H_j\}_{j\in[h]}$, $\mathbf{B} \in \{0,1\}^b$ and an element $x \in \mathcal{X}$, updates the current array $\mathbf{B}$ by setting $\mathbf{B}[H_j(x)] \leftarrow 1$ for all $j \in [h]$ and finally outputs the updated B. We use $\mathbf{B}_\mathsf{S} \leftarrow Upd(\mathbf{H}, \mathbf{B}, \mathsf{S})$ to denote the final array after inserting all elements in the set S one-by-one.
- $Check(\mathbf{H}, \mathbf{B}, x)$: It takes $\mathbf{H}, \mathbf{B}$ and an element x, and checks if $\mathbf{B}[H_j(x)] = 1$ for all $j \in [h]$. If true, it outputs 1 and 0 otherwise.

For correctness, a BF is *perfectly complete* if for all integers $b, h \in \mathbb{N}$, any set $S \in \mathcal{X}$, and $(\mathbf{H}, \mathbf{B}) \leftarrow Gen(\lambda)$ as well as $\mathbf{B}_\mathsf{S} \leftarrow Upd(\mathbf{H}, \mathbf{B}, \mathsf{S})$, it holds: $\mathbb{P}[\mathsf{Check}(\mathbf{H}, \mathbf{B}_\mathsf{S}, x) = 1] = 1$.

Symmetric Revocable Encryption [53]. Symmetric Revocable Encryption (SRE) is a primitive resembled from Symmetric Puncturable Encryption (SPE). With key space $\mathcal{K}_{SRE}$, message space $\mathcal{M}$ and tag space $\mathcal{T}$, SRE includes four polynomial-time algorithms shown in Algorithm 3.

Algorithm 3 Symmetric Revocable Encryption

- $\mathsf{SRE}.KGen(\lambda)$: It takes a security parameter λ as input and outputs a master secret key $sk \in \mathcal{K}_{SRE}$.
- $\mathsf{SRE}.Enc(sk, m, T)$: It takes as input a sk and a message $m \in \mathcal{M}$ with a list of tags $T \subseteq \mathcal{T}$, and outputs a ciphertext ct for m under tags T.
- $\mathsf{SRE}.KRev(sk, R)$: It takes as input sk and a revocation list $R = \{t_1, t_i, ..., t_\tau\} \subseteq T$, and outputs a revoked secret key sk_R, which can be used to decrypt only the ciphertext that has no tag belonging to R. For *compress revocation*, it takes $D \leftarrow \mathsf{SRE}.Comp(D, t_i)$ to make a revocation data structure D and takes $sk_R \leftarrow \mathsf{SRE}.cKRev(sk, D)$ to make the revoked secret key sk_R.
- $\mathsf{SRE}.Dec(sk_R, ct, T)$: It takes as input sk_R and ct encrypted under tags T, and outputs m or a failure symbol $\perp$.

For correctness, an SRE scheme is correct if the decrypt algorithm returns the correct result for every input of sk_R, ct, T, except with negligible probability. For security, SRE should provide the adaptive security of IND-REV-CPA and the selective security of IND-sREV-CPA [53].

B DSE

We notice that the majority of DSE schemes fail to consider the search and retrieval of the distinct values (which is a fundamental feature in relational databases). And none of existing works adequately address distinct search with well-defined security notions. DSE is commonly used for retrieving file-identifiers by keywords [13], and it is assumed that the client does not query the addition of the same pair of keyword and identifier [7,52]. Kamara et al. [31] conceptualized a database as a collection of files, each represented by a unique identifier. This assumption has been adopted by subsequent works, e.g., [23]. Stefanov et al. [52] used DSE to search the inverted index data structure for keyword/identifier pairs, while Cash et al. [7] revisited definitions from [13] to facilitate the storage of document-type data into databases using DSE. This philosophy has been widely embraced by subsequent works [2, 4, 10, 14, 53, 54, 61]. Xu et al. [62] proposed a robust DSE with an extension of backward security. Chen et al. [12] introduced a solution against key compromise in the context of DSE. Wang et al. [56] proposed a solution for keyword search on multi-writer encrypted databases. Recently, a volume-hiding DSE construction [66] was built on top of a padding strategy called dprfMM [42] (involving a dummy dataset). But its search complexity is proportional to the maximum response length [64].

C Padding Strategies

The padding strategy leverages false positive information as a means of performing obfuscation. Cash et al. [6] initially

proposed padding for keyword search to counter volume leakage. It is important to note that this technique relies on the distribution of the input dataset, making it susceptible to potential leakage even before a search query is initiated. Many works have been proposed to refine the padding [3, 28, 29]. SEAL [15] introduces an adjustable searchable encryption scheme that provides control over the amount of leaked access pattern information. This control is implemented through fine-tuning padding parameters to adjust leakage level. But SEAL does not support update operations, which is impractical in the EDB scenarios. ShieldDB [55] introduces a padding strategy that is explicitly tailored to accommodate a more realistic adversarial model, particularly within the context of databases undergoing continuous updates. This solution requires a large amount of dummy files to perform the padding on the batched data. We highlight that padding strategies, while effectively mitigating volume leakage, come with extra costs, impacting search efficiency and response time of queries. For example, during the initialization stage, they generate dummy data for the encrypted database, which can be computationally intensive, especially for large-scale databases. Padding strategies also tend to increase the amount of data transferred during search operations, thereby raising communication cost.

D Encrypted Databases

Over the past few years, notable efforts have been made in enhancing the security of search operations within encrypted databases. These advancements span a spectrum of dimensions, including bolstering data security, refining security schemes tailored to diverse functionalities, and optimizing the overall design of encrypted databases [18]. Encrypted databases have demonstrated capacity to execute a spectrum of secure functionalities, such as conjunctive search [43], keyword range search [67], and order-revealing encryption [58]. Researchers have leveraged structural encryption (e.g., [27] and [19]), private set intersection [44], private set union [25], and secure hardware [48] to delve into the realm of universal search within encrypted databases.

Table 4: Comparison of related EDBs.

EDB	Volume leakage	Tools for Security	Query mode
CryptDB [46]	✓	SQL aware Encryption	SQL
Arx [45]	✓	DSE	SQL
EnclaveDB [48]	✓	SGX	SQL
ShieldDB [55]	×	DSE + Padding	Keyword based

The SQL aware encryption used in CryptDB [46] is deterministic, which makes the EDB vulnerable to volumetric attacks. At present, researchers have used trusted hardware (SGX) [48] and DSE [45, 55] to safeguard the EDBs. Tab. 4 shows that current EDBs either overlook volume leakage or strongly rely on padding strategies (with significant storage and communication costs).

E Security analysis for BF-SRE

Theorem 1 (Adaptive Security of BF-SRE) *We define* $\mathcal{L}_D = \left(\mathcal{L}_D^{Upt}, \mathcal{L}_D^{Srch}\right)$ *as:*

$$\mathcal{L}_D^{Upt}(w, v, op) = op$$
$$\mathcal{L}_D^{Srch}(w) = sp(w), \text{TimeDTS}(w), \text{Update}(w),$$

BF-SRE is $\mathcal{L}_D$*-adaptively-secure.*

Proof 1 *We analyze the indistinguishability between BF-SRE and simulator* $\mathcal{S}_{BF-SRE}$*, and we use the game hop method to analyze indistinguishability.*

Game G_0**.** *This game is identical with the real BF-SRE:* $\mathbb{P}\left[\text{Real}_{\mathcal{A},\mathcal{S},\mathcal{L}}^{\text{BF-SRE}}(\lambda) = 1\right] = \mathbb{P}[G_0 = 1]$.

Game G_1*. We replace the calls of F with random strings. When each time a previous unseen call is input, we select a random output from this range space, and record it in tables Tokens, Tags, and Counts for* $F(K_s, w)$, $F(K_t, w||v)$*, and* $F(K_t, cnt)$*, respectively. Whenever F is recalled on the same input, the output value is retrieved directly from these tables. The distinguishing advantage between* G_0 *and* G_1 *is equal to that of PRF against an adversary making at most N calls to F:* $\mathbb{P}[G_1 = 1] - \mathbb{P}[G_0 = 1] \leq 3Adv_{F,\mathcal{B}_1}^{prf}(\lambda)$.

Game G_2*. We replace the Forward Private DSE instance* Σ_{add} *with the associate simulator* $\mathcal{S}_{add}^{DSE}$*. To construct this simulator, we use some bookkeeping to keep track of all the Update queries as they come, and postpone all addition and deletion operations to the subsequent Search query. This variant can be done because the additions leak nothing about their contents guaranteed by the forward privacy of* Σ_{add} *and the obliviousness of deletions to server.*

Moreover, a list Uphist is initialized and used in this game. The list Uphist in fact corresponds to the update history on w for the scheme Σ_{add} *and will be taken as the input of the simulator. The distinguishing advantage between* G_1 *and* G_2 *is reduced to the* $\mathcal{L}_{FS}$*-adaptive forward privacy of* Σ_{add}*. Therefor, there exists a PPT adversary* $\mathcal{B}_2$ *such that:* $\mathbb{P}[G_2 = 1] - \mathbb{P}[G_1 = 1] \leq Adv_{\Sigma_{add},\mathcal{S}_{add}^{DSE},\mathcal{B}_2}^{\mathcal{L}_{FS}}(\lambda)$.

Game G_3*. We only modify the generation of the ciphertext deletion. More precisely, we replace the values that were inserted previously and punctured later with constant 0.*

Since the modification above works only on the ciphertexts with revoked tags, we can see that the distinguishing advantage between G_2 *and* G_3 *is the IND-sREV-CPA security of the SRE scheme. The selective security is sufficient for the application here, because the reduction algorithm* $\mathcal{B}_3$ *can obtain the revoked tags from* $\text{Uphist}(w)$ *and simulate the encrypting process of non-deleted values with the revoked secret key. There for, there exists a reduction algorithm* $\mathcal{B}_3$ *such that:* $\mathbb{P}[G_3 = 1] - \mathbb{P}[G_2 = 1] \leq Adv_{SRE,\mathcal{B}_3}^{\text{IND-sREV-CPA}}(\lambda)$.

Game G_4*. We modify the way of constructing addition list* L_{add} *and the way of updating the compressed data structure D.* L_{add} *contains the addition entries and corresponds to the update history Uphist on w for the scheme* Σ_{add} *and is*

taken as the input of the simulator $\mathcal{S}_{add}^{DSE}$. In detail, we first compute the leakage information TimeDTS *and* Update *from the table* Uphist*, and then base the information to construct L_{add} and update D. This has no influence to the distribution of G_3: $\mathbb{P}[G_4 = 1] = \mathbb{P}[G_3 = 1]$.*

Game *G_5. We modify the generation of tags in a different way. In detail, we replace the tags with random strings directly, instead of computing them from keyword-value pairs and storing them in the table Tags. The distinguish between G_4 and G_5 is whether the tags will repeat. It is sufficient because each keyword/value/count pair was inserted/deleted at most once during the updates. We do not need to record every distinct tag for keeping consistence. Therefor, we have: $\mathbb{P}[G_5 = 1] = \mathbb{P}[G_4 = 1]$.*

Game *G_6. We replace the outputs from symmetric encryption to random strings on retrievals. The difference between G_5 and G_6 comes from the advantage: $\mathbb{P}[G_6 = 1] - \mathbb{P}[G_5 = 1] \leq Adv^{\text{IND-CPA}}_{\mathcal{E},\mathcal{B}_4}(\lambda)$.*

Simulator. *When building a simulator from G_6, we need to avoid directly using the keyword w as the protocols input. This can be done by replacing the input w with $\min sp(w)$. To construct L_{add} and collection D, we can properly take the leakage* TimeDTS *and* Update *as the input of* Search*, and the simulator does not need to keep the track of the updates anymore. In this condition, G_6 can be efficiently simulated by the simulator with the leakage function $\mathcal{L}$, so we have: $\mathbb{P}[G_6 = 1] = \mathbb{P}\left[\text{Ideal}^{\text{BF-SRE}}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda) = 1\right]$.*

Conclusion. *By combining all contributions from all games, there exists the adversary such that:*

$$|\mathbb{P}\left[\text{Real}^{\text{BF-SRE}}_{\mathcal{A}}(\lambda) = 1\right] - \mathbb{P}\left[\text{Ideal}^{\text{BF-SRE}}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda) = 1\right]| \leq$$
$$3Adv^{prf}_{F,\mathcal{B}_1}(\lambda) + Adv^{\mathcal{L}_{\mathcal{FS}}}_{\Sigma_{add},\mathcal{S}^{DSE}_{add},\mathcal{B}_2}(\lambda) + Adv^{\text{IND-sREV-CPA}}_{SRE,\mathcal{B}_3}(\lambda) + Adv^{\text{IND-CPA}}_{\mathcal{E},\mathcal{B}_4}(\lambda).$$

Theorem 2 (BF-SRE's DwVH Security) *The leakage function of* BF-SRE *$\mathcal{L}_D = (\mathcal{L}_D^{Upt}, \mathcal{L}_D^{Srch})$ is Distinct with Volume-Hiding.*

Proof 2 *Suppose an adversary $\mathcal{A}_D$ who can the distinguish signature from $\mathcal{L}_D$ and win the DwVH game with advantage $Adv^{DwVH}_{BF-SRE,\mathcal{A}_D}$. Then there exist a PPT algorithm $\mathcal{B}_D$ efficiently breaking all security guarantees from PRF F, Σ_{add},* SRE*, and the symmetric encryption $\mathcal{E}$, i.e.:*

$$MIN(Adv^{prf}_{F,\mathcal{B}_D}, Adv^{\mathcal{L}_{\mathcal{FS}}}_{\Sigma_{add},\mathcal{S}^{DSE}_{add},\mathcal{B}_D}, Adv^{\text{IND-sREV-CPA}}_{SRE,\mathcal{B}_D}, Adv^{\text{IND-CPA}}_{\mathcal{E},\mathcal{B}_D}) \geq Adv^{DwVH}_{BF-SRE,\mathcal{A}_D}.$$

$\mathcal{B}_D$ should perform like the simulator $\mathcal{S}$ with $\mathcal{L}_D$ that $\mathcal{A}_D$ can access in the DwVH game. Specifically, it contains four sub-program $\mathcal{B}_D^1$, $\mathcal{B}_D^2$, $\mathcal{B}_D^3$, and $\mathcal{B}_D^4$ to break the secure primitives from F, Σ_{add}, SRE*, and $\mathcal{E}$, respectively. With the guess from $\mathcal{A}_D$, the sub-programs leverage bookkeeping and embed the challenge message to break the above security primitives. We detail the DwVH security analysis in the full version [37].*

F High-level design for *d*-DSE

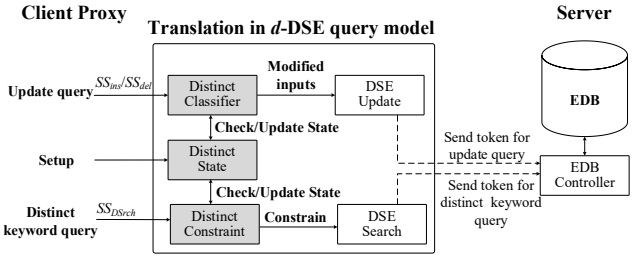


Figure 12: The high-level design of d-DSE construction.

Fig. 12 shows a high-level design for the d-DSE construction. The concrete construction should provide the components of Distinct Classifier, Distinct State, and Distinct Constraint in the Setup, Update, and Search protocols, respectively. The core of achieving secure distinct search is to: *promptly update the state of (w,v,op) inputs at the initial occurrence, and subsequently conceal the repetitive (w,v,op) inputs.*

F.1 Store the state of distinct inputs

The Setup protocol allocates local memory as Distinct State to record the state of whether the input has appeared. In previous work, the state is commonly used to achieve the forward and backward privacy and special properties such as non-interactivity [53] and robustness [62]. To clarify whether the input is distinct, Distinct State should efficiently store all updates that include distinct (w, v, op) pairs.

F.2 Tag distinct values

The Update protocol employs a program called Distinct Classifier to generate real or dummy tags for each input. The tags are used to identify whether the matched value is represented as distinct in the Search protocol. Based on the Distinct State, the inputs attach the real tag as the distinct. The repetitive inputs are replaced with bogus keyword/value pairs and tagged the dummy tag. The processed data will be updated to the EDB system.

We state that traditional dummy data generation like that in [15,55] is applicable to our requirement. The inputs marked by dummy tags will not affect the results of distinct values. Based on the Distinct State, the repetitive inputs can be replaced with random duplicated pairs to construct the arbitrary volume distribution on the EDB system.

F.3 Retrieve distinct values

The Search protocol employs the Distinct Constraint program to generate the constrained key and retrieves the distinct values. Under the requirement from the d-DSE security definitions, the tag can only reveal whether the corresponding value is distinct in the Search protocol. In the full version [37], we further discuss the roadmap to construct d-DSE.