Range Dynamic Searchable Symmetric Encryption: Combating Volume Leakage and Enabling Non-interactive Deletion

Jinguo Li, *Member, IEEE*, Qiang Lv, Junqin Huang*, Linghe Kong, *Senior Member, IEEE*

***Abstract*—Most existing range Searchable Symmetric Encryption (SSE) schemes are static, making them vulnerable to volume pattern leakage attacks. While Volume-Hiding Range SSE (VH-RSSE) was proposed to mitigate such leakage, it does not support dynamic data updates. Dynamic SSE (DSSE) addresses this limitation, but directly combining VH-RSSE with DSSE for range queries presents significant challenges, including: (i) lack of backward privacy support, (ii) inaccurate search results, and (iii) high storage and communication overhead. Moreover, current schemes typically require multiple client-server interactions to perform deletions. In this paper, we introduce a volume-hiding range DSSE scheme that is the first to support non-interactive deletion, while concealing the identifier volumes associated with keyword values during query execution. Specifically, we propose the Revocable Inverted Index (RII), which integrates an order-accumulated inverted index and bitmap structures to efficiently handle range queries. By combining RII with DSSE and symmetric revocable encryption, we enable the client to directly manage ciphertext deletions, thus achieving non-interactive deletion and eliminating the need for additional communication overhead. We provide a formal analysis of the leakage functions to demonstrate that our scheme offers the desired security guarantees. Extensive experimental results show that our approach significantly enhances efficiency when compared to existing solutions.**

***Index Terms*—Searchable symmetric encryption, Non-interactive deletion, Symmetric revocable encryption, Volume pattern leakage component.**

I. Introduction

With the rapid growth of cloud computing, individuals and organizations are increasingly outsourcing their data storage and computation needs. PayPal, Netflix, and HealthShare [1] rely on Amazon Web Services, while Yahoo has deployed its services on Verizon Cloud. However, despite the convenience, public cloud services remain vulnerable to both internal and external threats. For instance, Yahoo suffered a massive data breach in which all user data was compromised [2] (approximately 3 billion records). Encrypting sensitive data is a fundamental measure to mitigate data leakage risks; however, traditional symmetric encryption often sacrifices data availability and search functionality.

To address these challenges, advanced cryptographic techniques such as fully homomorphic encryption [3] and functional encryption [4] have been proposed, albeit at the cost of significant computational overhead. Searchable Symmetric Encryption (SSE) [5, 6] offers a more efficient alternative, balancing data confidentiality with availability and searchability. Nevertheless, traditional SSE schemes lack support for data updates, a limitation addressed by Dynamic SSE (DSSE) [7]. Despite this advancement, most DSSE schemes remain susceptible to leakage during search and update operations, thereby enabling file-injection attacks [8], in which adversaries infer access patterns by injecting crafted files.

To counter such attacks, the security goals of forward and backward privacy have been introduced. Forward privacy prevents newly added documents from being linked to previous queries, while backward privacy ensures that deleted or modified records remain inaccessible after removal. Bost et al. [9] categorized backward privacy into three types, with Type-I offering the strongest guarantee. Building on this, Zuo et al. [10] proposed a Type-I$^-$-secure DSSE scheme. Nevertheless, most existing solutions support only single-keyword queries, limiting their practical applicability. Furthermore, directly extending prior work on forward and backward privacy to support more expressive queries may introduce new leakages, including search pattern, access pattern, and volume pattern leakages.

As mentioned above, SSE has evolved to support complex query types, including fuzzy keyword [11–13], Boolean [14–16], range [17–20], and geometric queries [21, 22]. Among these, range queries are particularly expressive and widely applicable. A common approach is to convert range queries into a disjunction of multiple keyword queries, where efficiency and communication cost depend on the query range size [23, 24]. Similar to earlier methods, range queries can be realized using either SSE-based [25] or DSSE-based [18, 26] schemes. However, many such schemes suffer from volume pattern leakage, which reveals the number of matched documents per query. Recent studies [27–30] have shown that even volume leakage alone can enable adversaries to reconstruct entire datasets. As shown in Fig. 1, assume there are three

This work was supported in part by the National Natural Science Foundation of China under Grants 62172276 and 62372285; in part by Yunnan Key Research Program grant 202402AD080004; in part by the Shanghai Action Plan for Science, Technology and Innovation grant 24BC3201300; and in part by Startup Fund for Young Faculty at SJTU (SFYF at SJTU) grant 25X010502613.

J. Li and Q. Lv are with the College of Computer Science and Technology, Shanghai University of Electric Power, Shanghai 201306, China (e-mail: lijg@shiep.edu.cn; lvqiang5878@163.com)

J. Huang and L. Kong are with the School of Computer Science, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: junqin.huang@sjtu.edu.cn; linghe.kong.sjtu.edu.cn; Corresponding author: Junqin Huang)

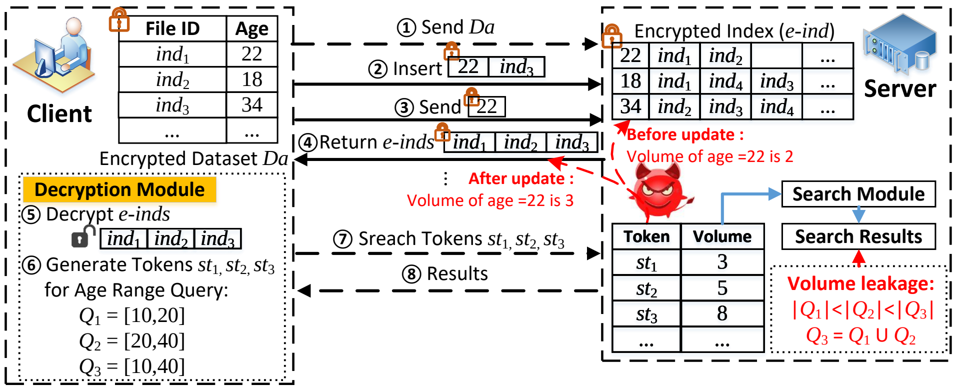


Fig. 1: Volume leakages in the DSSE scheme.

queries, Q_1, Q_2. Owing to the additive property of range query volumes, the volume of [10, 40] equals the sum of the volumes of [10, 20] and [20, 40]. Consequently, the server can learn the number of matching documents for each query (i.e., $|Q_1| < |Q_2| < |Q_3|$). If $|Q_3| = |Q_1| + |Q_2|$, the server might infer that $Q_3 = Q_1 \cup Q_2$. Over time, by accumulating such observations, the server could reconstruct the dataset via known attacks [31].

Although recent solutions such as VH-RSSE [20] leverage bitmap-based inverted indexes to obscure volume information, they lack update support, limiting their practicality. Moreover, most existing schemes do not scale well. In large datasets (e.g., millions of records), storing identifiers for each keyword incurs significant communication and storage overhead. To maintain privacy, these schemes must pad each keyword's volume to a uniform length, further exacerbating inefficiency. In dynamic settings, updates introduce additional leakages that adversaries can exploit through file-injection attacks. This raises a fundamental question: *Can we design a scalable, update-friendly range DSSE scheme that guarantees forward and backward privacy while preventing volume pattern leakage?*

To achieve this design goal, we propose Range DSSE (RDSSE), a novel scheme that achieves volume leakage resistance, supports efficient updates, and guarantees both forward and backward privacy. Furthermore, it enables non-interactive deletion, significantly reducing client-server interactions during record revocation. Our main contributions are summarized as follows:

- To obscure volume patterns, we design a hybrid index that combines bitmap structures with order-accumulated indexes. This approach hides keyword volume patterns, as the volume information embedded in bitmap structures is ineffective for volume pattern attacks. By partitioning the entire range using a local search tree and our hybrid index, each range query requires at most two search tokens, maintaining an O(1) query token size. Additionally, the design avoids full padding for every keyword, enabling efficient deployment even over large datasets.
- To enable non-interactive deletion, we leverage bloom filters to track record states. Based on these states, symmetric revocable encryption manages update and search operations using real and dummy tags. By controlling these tags, clients can perform deletions locally without server interaction. Experimental results show that our scheme reduces deletion latency by approximately 634.64× to 740.78× compared to existing approaches.

II. Related Work

A. Expressive Queries Enabled SSE

To balance privacy protection and search efficiency for expressive queries, SSE has been widely adopted to simultaneously achieve data security and searchability. In 2000, Song et al. [5] first introduced the concept of SSE. Later, Curtmola et al. [6] formalized the security definition of SSE and employed an inverted index structure to achieve sub-linear search time. Since then, various studies have explored rich query expressions, including boolean queries [14–16], range queries [18–20], and geometric queries [21, 22], etc. For instance, Cash et al. [14] proposed the OXT protocol, enabling sub-linear search time for boolean queries. Following this, several SSE schemes have been optimized to support conjunctive keyword queries, each offering a different trade-off between leakage and efficiency [16]. Faber et al. [23] studied range queries in SSE using binary tree indexes, performing disjunctive queries by searching tree nodes that cover the query range. Demertzis et al. [24] proposed six range SSE schemes based on different tree indexing techniques to achieve a general solution. Wang et al. [25] and Liu et al. [20] designed novel inverted index structures to reduce range queries to two keyword searches. In this paper, we primarily focus on range queries.

However, once a database is encrypted, traditional SSE schemes typically do not support updates. To overcome this limitation, DSSE schemes [32] were proposed. Nevertheless, these schemes remain vulnerable to file-injection attacks [33]. To address such vulnerabilities, Stefanov et al. [34] introduced the notions of forward and backward privacy. Building on this, Bost et al. [9] formalized the concept of forward privacy and classified backward privacy into three distinct types: Type-I (strongest), Type-II, and Type-III (weakest).

B. Volume Pattern Leakage-Resistant SSE

Although numerous DSSE schemes satisfy forward and/or backward privacy requirements, most of them are limited to supporting only single-keyword queries. To enable secure range queries in DSSE, Zuo et al. [10] proposed two schemes, FBDSSE-RQ-I and DSSE-RQ-II. Both require constructing a tree before each query, with one scheme lacking forward privacy. Moreover, as they are based on the Paillier cryptosystem, their message space is limited, supporting only a finite number of files. Subsequently, Wang et al. [26] extended FBDSSE-RQ-I to achieve a generic forward-private DSSE. It integrates Bost's backward privacy framework [9] with a two-round transformation, thereby achieving Type-II backward privacy. However, these range DSSE schemes inevitably leak some information, collectively referred to as leakage profiles, which can be categorized into three types: (1) search pattern leakage; (2) access pattern leakage; and (3) volume pattern leakage.

Recent studies have shown that most range SSE schemes remain vulnerable to certain attacks, such as volume pattern leakage attacks. For instance, Zuo et al. [18] proposed a forward and backward secure DSSE range query (FBDSSE-RQ) scheme using a bitmap index to mitigate volume pattern leakage. Nevertheless, other types of leakage, such as the total number of updates, still persist [24]. In addition, Wang et al. [26] observed that the bitmap index used in [18] limits file size, making FBDSSE-RQ impractical for large datasets. To address this, Liu et al. [20] introduced a range SSE scheme that combines bitmap indexes with an order-accumulated inverted index, mitigating volume leakage while better supporting large datasets. However, it does not address other types of leakage (e.g., search or access patterns) and does not support updates to encrypted databases.

Padding strategies mitigate volume pattern leakage [28, 35], with databases like SEAL [28] and ShieldDB [35]. However, these introduce significant storage and communication overhead due to redundant data. Recent work on volume-hiding encrypted multi-maps [36] claims near-optimal efficiency, but Ando et al. [37] noted that padding query responses to maximum length makes the response size of volume-hiding multi-map range SSE at least as large as the dataset. Hence, padding is incompatible with range SSE, where each keyword maps to every document. A comparison with prior work is shown in Table I.

III. RDSSE Scheme

The client partitions the query range $[w_1, w_m]$ into separate, non-overlapping subranges, each containing m_{sub} keywords. For each subrange, starting from an inverted index $\mathbf{I} = (w_i, \mathrm{I}_i)_{i=1}^{m_{\text{sub}}}$, we construct a revocable inverted index $(w_i, \mathrm{I}_i^*)_{i=1}^{m_{\text{sub}}}$, where $\mathrm{I}_i^* = \bigcup_{j=1}^{i} \mathrm{set}(\mathrm{I}_j)$. Here, the set contains all document identifiers whose values do not exceed w_i. Document identifiers are encoded using a bitmap index, ensuring that the size of each subrange fits within the storage capacity of a bit string. For instance, consider an inverted index $(w_i, \mathrm{I}_i)_{i=1}^{m_{\text{sub}}}$, where each I_i is represented by a bit string bs. If a document identifier ind_j is included in I_i, the j-th bit of bs is set to 1; otherwise, it remains 0. Fig. 2 illustrates the setup, addition, and deletion of document identifiers within the bitmap index. Specifically, Fig. 2(a) shows a bit string bs of length 5, where only files ind_1 and ind_3 are present in the database. Fig. 2(b) demonstrates the addition of file ind_2 and the removal of file ind_1. Additionally, we introduce a set $C = \{C^{(0)}, C^{(1)}, \ldots\}$ to record subrange information, where each $C^{(i)}$ stores details of the i-th subrange, including boundary values $[C_{\min}^{(i)}, C_{\max}^{(i)}]$.

As shown in Algorithm 1, the client takes set C as input and constructs a binary tree BT to facilitate local search. Each node N corresponds to a specific subrange, defined by $[N_{\min}, N_{\max}]$, where $N_{\min} = N_{\text{left.min}}$ and $N_{\max} = N_{\text{right.max}}$. Once BT is built, the client creates a revocable inverted index for each subrange. Fig. 3(a) illustrates the structure and functionality of the revocable inverted index. Specifically, when deleting the pair $(18, ind_6)$, the client revokes the associated encryption key using Symmetric Revocable Encryption (SRE) [38]. It eliminates the need to manually set the corresponding bit in the revocable inverted index to 0. In contrast, for the order-accumulated inverted index shown in Fig. 3(b), deleting $(18, ind_6)$ requires explicitly resetting the corresponding bit to 0. Although the client stores the keyword list, document identifier list, and the boundary information $[C_{\min}^{(i)}, C_{\max}^{(i)}]$ for each subrange $C^{(i)}$, this information alone is insufficient for direct bit-level operations. The offset value indicates the starting bit position of each keyword's identifiers within the bitmap. For example, as shown in Fig. 3(b), the offset for keyword 18 is 3. However, the client cannot determine precisely which bit corresponds to keyword 18 or whether the bit corresponding to ind_6 for keyword 22 should be reset to 0. Compared to the order-accumulated inverted index, our revocable inverted index enables more efficient deletion of keyword/document identifier pairs on the client side, achieved simply by adding a bloom filter to store distinct states.

To search for documents within the range $Q = [w_\ell, w_r]$, the client first executes a LocalSearch(Q, BT). If a node's range in BT perfectly matches the query range, the client directly accesses the corresponding bitmap index locally, without needing to transmit search tokens to the server. When the ranges do not align, the client generates two search tokens: one corresponding to $w_{\ell-1}$ and the other to w_r, which are then sent to the server. Upon receiving the search tokens, the server performs the search operation, computes the result, and sends it back to the client. After decrypting the result, the client retrieves the bitmap index from BT. This entire process is divided into four stages: setup, build, search, and update. Finally, the obtained results are given by $\mathrm{I}_r^* \backslash \mathrm{I}_{\ell-1}^*$.

TABLE I: Comparison between our scheme and previous schemes.

Schemes	Forward Privacy	Backward Privacy	Number of Query Tokens	Storage Cost (Server's Side)	Volume Pattern Privacy	Non-interactive Deletion
FPDSSE-RQ-I [17]	✓	✗	$O(\log\hat{S})$	$O(F\cdot\lceil\log W\rceil)$	✗	✗
DSSE-RQ-II [17]	✗	Unknown	$O(\log\hat{S})$	$O(F\cdot N)$	✓	✗
Range SSE-I [25]	✗	✗	$O(S)$	$O(F+W)$	✗	✗
Range SSE-II [25]	✗	✗	$O(1)$	$O(F+W_2)$	✗	✗
FBDSSE-RQ [18]	✓	Type-R	$O(\log S)$	$O(F\cdot N)$	✓	✗
VH-RSSE [20]	✗	✗	$O(1)$	$O(W\cdot L)$	✓	✗
Our Proposed Scheme	✓	Type-II	$O(1)$	$O(N\cdot L)$	✓	✓

Note that N, W, and F denote the total number of keyword/document pairs, the total number of keywords, and the total number of documents, respectively. To simplify storage, we use a unique document identifier (denoted by *ind*) to refer to each document in the database. L represents the length of the bitmap, $\hat{S}$ denotes the range query size, and S is the number of distinct values within a range query. Typically, $\hat{S}\geq S$. Backward privacy is classified into three levels: Type-I, Type-II, and Type-III, with the special case Type-R, which is an extension from [9]. "Unknown" indicates that the detailed backward privacy analysis was not provided.

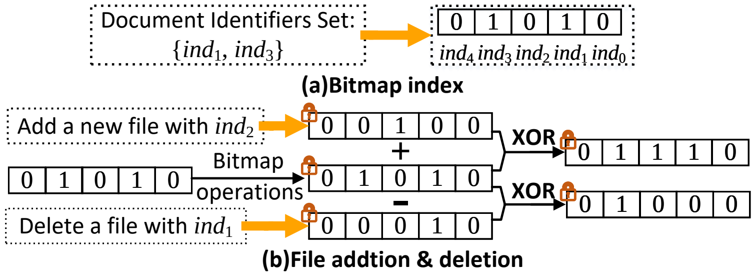


Fig. 2: Operations of document identifiers in the bitmap index.

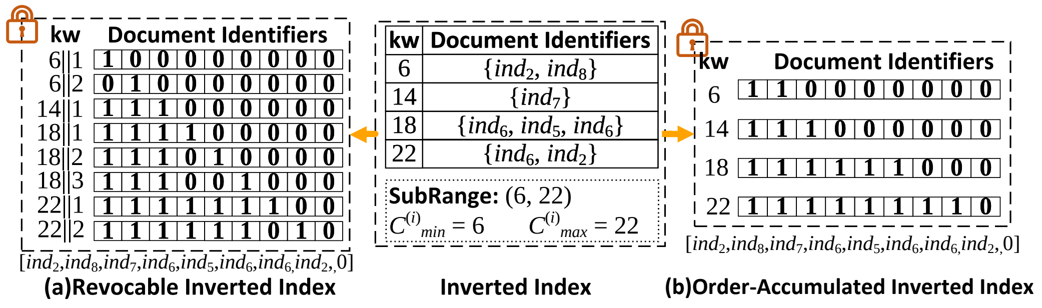


Fig. 3: Revocable and order-accumulated inverted index.

Fig. 4 illustrates the Update and Search processes in our scheme, where green and purple represent deletion and repetition revocations, respectively. Distinct states are stored in a BF, and FP-DSSE Σ_{add} uploads modified keyword-value pairs. Tags for redundant or deleted values in the encrypted database (EDB) are revoked using SRE. During the Update phase, the BF determines whether an input is new or repeated, generating real or dummy tags accordingly. For example, when sequentially adding $(18, ind_6)$, $(18, ind_5)$, $(18, ind_6)$ and $(18, ind_5)$, the BF tracks their states: real tags are created for the first two pairs, while dummy tags and SRE key revocations are triggered for the repeated ones. If a pair such as $(18, ind_6)$ is deleted, its real tag is revoked via SRE. In the Search phase, only unrevoked SRE ciphertexts are decrypted using constrained sub-keys, followed by local decryption of symmetric ciphertexts. Therefore, searching for keyword 18 retrieves only $(18, ind_5)$, which yields ind_5 after decryption.

A. Setup Stage

As shown in Algorithm 2, the bitmap length L is determined by the maximum database volume V, ensuring that $L\geq V$ to prevent volume-related information leakage. An empty EDB is initialized, along with its internal state σ, a secret key K_σ derived from the FP-DSSE scheme Σ_{add}, and other components. These components include secret keys K_s, K_t, K_c, a BF hash set $\mathbf{H}$, and a BF bit array $\mathbf{B}$. A secure Pseudorandom Function (PRF) $\mathbf{F}$ and a hash function $\mathbf{H_1}$ are used to process bit strings. Additionally, four mappings **MSK**, **UpCnt**, **C**, and **D** are initialized in an empty state. A binary tree BT is constructed for range-related data storage. The key K_c encrypts values, while K_s generates cache tokens tkn to retrieve previous search results from $\text{EDB}_{\text{cache}}$. Tags are generated using K_t, and the update counter **UpCnt** is managed. Mapping **C** tracks search counts for each keyword w, **MSK** holds the master key for each keyword, and **D** maintains the structure of revoked keys. These components together ensure both security and efficiency during database

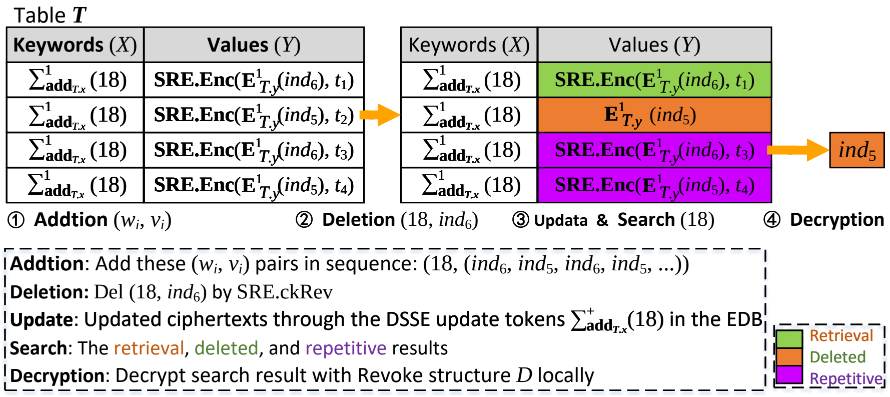


Table T
Keywords (X)
Values (Y)
$\Sigma^1_{\mathbf{add}_{T,x}}(18)$
SRE.Enc($E^1_{T,y}(ind_6)$, t_1)
SRE.Enc($E^1_{T,y}(ind_5)$, t_2)
SRE.Enc($E^1_{T,y}(ind_6)$, t_3)
SRE.Enc($E^1_{T,y}(ind_5)$, t_4)
$E^1_{T,y}(ind_5)$
ind_5
① Addtion (w_i, v_i)
② Deletion (18, ind_6)
③ Updata & Search (18)
④ Decryption
Addtion: Add these (w_i, v_i) pairs in sequence: (18, (ind_6, ind_5, ind_6, ind_5, ...))
Deletion: Del (18, ind_6) by SRE.ckRev
Update: Updated ciphertexts through the DSSE update tokens $\Sigma^+_{\mathbf{add}_{T,x}}(18)$ in the EDB
Search: The retrieval, deleted, and repetitive results
Decryption: Decrypt search result with Revoke structure D locally
Retrieval
Deleted
Repetitive

Fig. 4: Non-interactive deletion based on SRE.

Algorithm 1 Local Binary Tree

```
BinaryTree.BuildTree($C$):
    Inputs: A Collection $C$
    Outputs: A Binary Tree $BT$
1:  if $|C| = 0$ then
2:      Return $\perp$
3:  else if $|C| = 1$ then
4:      Generate a root node and set this node as $BT$
5:      Associate $C^{(1)}$ to this node
6:      Return $BT$
7:  else
8:      Generate $2^{(\ell)}$ leaf nodes where $2^{(\ell-1)} < |C| \leq 2^{(\ell)}$
9:      Assign each element in $C$ to each leaf node
10:     for $i = \ell - 1$ to 0 do
11:         Generate $2^i$ leaf nodes
12:         for each node $N$ do
13:             Set its left and right child to two consecutive unassigned nodes of previous level
14:             $N_{min} := N.left_{min}$
15:             $N_{max} := N.right_{max}$
16:         end for
17:     end for
18:     Set the root node as $BT$
19: end if
20: Return $BT$

BinaryTree.LocalSearch($Q, BT$):
    Inputs: Range Query $Q = [w_\ell, w_r]$ and binary tree $BT$
    Outputs: $C_s$
1:  Temp $\leftarrow$ Empty Set
2:  for $w \in \{w_\ell, w_r\}$ do
3:      Let $N$ be the root node
4:      while $N$ is not a leaf node do
5:          if $w \leq N.left$ then
6:              $N := N.left$
7:          else
8:              $N := N.right$
9:          end if
10:     end while
11:     Temp := Temp $\cup$ $N$
12: end for
13: Parse Temp as $\{C^{(left)}, C^{(right)}\}$ where $left \leq right$
14: $C := \{C^{(left)}, C^{(left+1)}, \ldots, C^{(right)}\}$
15: Return $C_s$
```

Algorithm 2 Setup (EDB is generated through encryption inverted index)

```
    Inputs: Security Parameter $\lambda$, $V = \max\{|\mathrm{set}(\mathrm{DB}(w))|, \forall_w \in W\}$
    Outputs: System Parameters
1: Initialize $L$ ($L \geq V$), hash function $\mathbf{H_1}$ and a PRF $\mathbf{F}$
2: Initialize DDSE scheme $(\mathrm{EDB}, \sigma, K_\Sigma) \leftarrow \Sigma_{add}.\mathrm{Setup}(\lambda, \mathrm{DB})$, $\Sigma_{add}$ is FP DSSE scheme
3: Initialize the Bloom Filter $\mathbf{H}, \mathbf{B} \leftarrow \blacksquare.\mathrm{Gen}(\lambda)$, original distinct state is null
4: Initialize empty maps MSK, UpCnt, C, D, $\mathrm{EDB_{cache}}$, a binary tree $BT$
5: Randomly generate keys $K_s, K_t, K_c \xleftarrow{\$} \{0,1\}^\lambda$
6: Send EDB and the cache $\mathrm{EDB_{cache}}$ (cache of EDB) to the server
```

operations.

B. Index Generation Stage

In Algorithm 3, lines 1-4, the database $\mathrm{DB} = (ind_i, w_i)_{i=1}^{d}$ is provided as input. The client initially parses DB into an inverted index $\mathbf{I} = (w_i, \mathrm{I}_i)_{i=1}^{m}$, where I_i represents the set of identifiers associated with keyword w_i, and $\mathbf{I}^*$ contains identifiers with values less than or equal to w_i. At this stage, the client initializes the offset θ and the subrange set C. By leveraging the offset and the size of the document identifiers within $\mathbf{I}^*$, the client can reconstruct the bitmap when necessary. This design minimizes storage requirements while preserving essential data. In lines 5-19, the client iterates over $\mathbf{I}$. At lines 8 and 11, for each pair (w_i, I_i), a bitmap is created using the offset and document identifiers. This bitmap is then encrypted using the `Update` algorithm and sent to the server's EDB. In lines 7-15, if the number of documents in the current subrange is less than or equal to L (corresponding to w_i), it is placed in one subrange; otherwise, it is placed in a different one. Lines 16-18 update and save subrange information. Finally, at line 20, the binary tree BT is constructed using the set C. The client then splits the range $[w_1, w_m]$ into disjoint subranges, constructs BT and the EDB. L is the he length of bitstring bs, e is the cyphertext of bs.

C. Update Stage

As shown in Algorithm 4, the EDB is updated based on the input operation op = ('add', 'del') and the new internal state. By transferring the deletion operation from the server to the client, the scheme significantly reduces the overhead associated with multiple rounds of interaction between the server and client. At lines 1-7, the client begins by initializing the internal state associated with the keyword w_j. Then, at lines 8-28, operations such as addition or deletion are executed according to the input op. At line 8, the client processes each (w_j, v) within I_j. At line 9, the client generates tags t and l using a PRF function. Specifically, t serves as the real tag marking the first occurrence of (w_j, v), while l acts as a dummy tag for duplicate occurrences. At line 11, the retrieval ciphertext s is created by applying symmetric encryption to the combination of the value v and a unique counter cnt. In lines 12-26, the operation type ("add" or "del") determines whether ciphertext is generated or the SRE key is revoked. For addition operations, the client generates a key K_1 for w_j using the PRF at lines 13-15, derives a bitmap bs based on the offset θ and the size of the document identifiers $\mathbf{I}^*$, and encrypts bs to produce an encrypted bitmap e. At line 16, the client checks whether $w_j||v$ is a first-time occurrence by running the distinct classifier $\Phi.\mathbf{Check}(\mathbf{H}, \mathbf{B}, t)$. If so, the

Algorithm 3 Invert-Index Generation ($\mathbf{I}$ is inverted index of DB)

```
    Inputs: Database DB = $\{(ind_i, w_i)\}_{i=1}^{d}$
    Outputs: Encrypted Database EDB, Local Search Tree $BT$
 1: Parse DB (A plaintext database) into an inverted index $\mathbf{I} = \{(w_i, \mathrm{I}_i)\}_{i=1}^{m}$
 2: Initialize an empty set $C$, let $cnt$ = 1
 3: Initialize an empty set $O$, an offset value $\theta = 0$
 4: Initialize an empty inverted index $\mathbf{I}^*$ and set $\mathbf{I}^*[w_0] = \emptyset$
 5: for $i \leftarrow 1$ to $m$ do
 6:     $temp \leftarrow$ Initialize an empty set
 7:     if $|C^{(cnt)}| + |\mathrm{set}(\mathrm{I}_i)| \leq L$ then
 8:         $temp \leftarrow$ Update($K_\Sigma$, st, 'add', $\theta$, $\mathbf{I}^*[w_{i-1}]$, $(w_i, \mathrm{I}_i)$; EDB)
 9:         $\mathbf{I}^*[w_i] \leftarrow temp \cup \mathbf{I}^*[w_{i-1}]$ where $temp = \mathrm{set}(\mathrm{I}_i)$
10:     else
11:         $\theta := 0$
12:         $temp \leftarrow$ Update$(K_\Sigma, \mathbf{st}, \text{'add'}, \theta,$
13:             $\mathbf{I}^*[w_{i-1}], (w_i, \mathrm{I}_i)$; EDB)
14:         $\mathbf{I}^*[w_i] = temp$
15:         Append $C^{(cnt)}$ into $C$
16:         $cnt = cnt + 1$
17:     end if
18:     $\theta := \theta + |\mathbf{I}^*[w_i]|$
19:     Let $C_{info}^{(cnt)} := (|\mathbf{I}^*[w_i]|, \theta)$
20:     Append $\theta$ into $O$
21: end for
22: Return $BT$
```

Algorithm 4 Update ($\mathscr{E}, \mathscr{D}$ is symmetric encrypt and decrypt algorithms)

```
Update($K_\Sigma, \mathbf{st}, op, \theta, \mathbf{I}^*, (w_j, \mathrm{I}_j)$; EDB):
 1: Get $msk$, $D$, $i$, and $cnt$ from $\mathbf{MSK}[w_j]$, $\mathbf{D}[w_j]$, $\mathbf{C}[w_j]$, and $\mathbf{UpCnt}[w_j]$, respectively.
 2: if $msk$ is not initialized then
 3:     Set $msk \leftarrow \mathbf{SRE}.KGen(\lambda)$, where $msk = (sk, D)$
 4:     Set $\mathbf{MSK}[w_j] \leftarrow msk$, $\mathbf{D}[w_j] \leftarrow D$
 5:     Set $\mathbf{UpCnt}[w_j] \leftarrow 0$, $\mathbf{C}[w_j] \leftarrow 0$
 6:     Set $cnt \leftarrow 1$, $\mathbf{UpCnt}[w_j] \leftarrow cnt$
 7: end if
 8: for each $v$ in $\mathrm{I}_j$ do
 9:     Gen. the real/dummy tags $t \leftarrow \mathrm{F}(K_t, w_j||v||0)$
10:     $l \leftarrow \mathrm{F}(K_t, w_j||v||cnt)$
11:     Gen. the retrieval $s \leftarrow \mathscr{E}(K_c, v||cnt)$
12:     if $op == add$ then
13:         $K_1 \leftarrow \mathrm{F}(K_c, w_j)$
14:         Generate $bs$ based on $\mathbf{I}^*$ and $\theta$
15:         $e \leftarrow bs \oplus \mathbf{H_1}(K_1, w_j)$
16:         if $\Phi.\mathbf{Check}(\mathbf{H}, \mathbf{B}, t)$ is false then
17:             Update BF $\Phi.Upd(\mathbf{H}, \mathbf{B}, t)$
18:             $ct \leftarrow \mathbf{SRE}.\mathrm{Enc}(msk, s, t)$
19:             Insert EDB $\Sigma_{add}$.Update $(K_\Sigma, w_j||i, (ct, t, e), add$;EDB)
20:             Puncture dummy tag $D' \leftarrow \mathbf{SRE.Comp}(D, l)$
21:         end if
22:     else
23:         Puncture the real tag $D' \leftarrow \mathbf{SRE.Comp}(D, t)$
24:         $\mathbf{D}[w_j] \leftarrow D'$
25:     end if
26:     $\mathbf{UpCnt}[w_j] \leftarrow cnt + 1$
27: end for
```

client updates the BF, generates SRE ciphertext ct tagged with t, and uploads it via the DSSE system. Otherwise, dummy SRE ciphertext is created, the corresponding key structure D is revoked, and the dummy ciphertext is uploaded instead. For deletion operations, at line 24, the client revokes the SRE key structure associated with the initial tag t of $w_j||v$. Finally, at line 26, the keyword count $\mathbf{UpCnt}[w_j]$ is updated.

Notably, by combining the revocable inverted index and SRE, the deletion of $(w_j, v)_{v \in bs}$ can be achieved entirely on the client side, without the need for interaction with the server, unlike previous dynamic range query schemes [18]. First, the client obtains the real tag t for $(w_j, v)_{v \in bs}$. Then, using the method provided by SRE, $D' \leftarrow \mathbf{SRE.Comp}(D, t)$, the real tag to be deleted is added to the revocation data structure D, thus completing the deletion of $(w_j, v)_{v \in bs}$. In the next query, using $sk_R \leftarrow \mathbf{SRE.ckRev}(sk, D')$, the revoked secret key sk_R for $(w_j, v)_{v \in bs}$ is obtained. As a result, the search results in the EDB will no longer include $(w_j, v)_{v \in bs}$ (logical deletion). Since the EDB still stores revoked or deleted records, the server cannot determine that $(w_j, v)_{v \in bs}$ has been deleted, thus naturally ensuring backward privacy.

D. Search Stage

In Algorithm 5, the system retrieves the ciphertexts using keyword w and returns distinct values from the search results. This process conceals the actual pattern of result lengths. For a range query $Q = [w_\ell, w_r]$, at line 2, the client obtains the subrange set C_s for Q through the algorithm LocalSearch(Q, BT). If w_ℓ and w_r exactly match the lower and upper bounds of C_s, as specified in lines 3-22, the client can locally generate bitmaps for the range Q. Otherwise, the client generates search tokens and sends them to the cloud server. At lines 23-51, the algorithm retrieves all encrypted bitmap indexes EBS based on the list of search tokens generated for Q. In lines 24-27, the client processes the internal state and applies the distinct constraint program, represented as $sk_R \leftarrow \mathbf{SRE}.ckRev(sk, D)$. During decryption, sk_R filters out revoked SRE ciphertexts and isolates distinct values, thereby obscuring the query response's length. The revoked SRE key sk_R and the cache token tkn are then sent to the server. At lines 28 and 29, the client refreshes the internal state in preparation for subsequent searches involving w. At line 30, the client collaborates with the server to execute the Σ_{add} search protocol, enabling the server to retrieve a list of results, $List$. In lines 31-39, each ciphertext ct within $List$ is decrypted by the server using sk_R. Only symmetric ciphertexts from non-revoked SRE ciphertexts are retained. At line 40, prior results are fetched from the encrypted database cache $\mathrm{EDB_{cache}}$ using tkn. Finally, at line 41, the final search results, EBS, are delivered to the client. At lines 42-50, the client decrypts the search results and restores all plaintext bitmap indexes, utilizing C_s in the process.

IV. Security Analysis

Since the security definition of our proposed scheme is based on leakage functions, it is essential to define these functions, as well as forward and backward privacy, before analyzing the security of the scheme.

Definition 1 (**Leakage Function**). *To define the leakage function, we first introduce a range query* $q = (t, [a, b]) =$

Algorithm 5 Search ($w||v$ is the string concatenation of w and v)

```
    **Inputs:** Search Range $Q = [w_\ell, w_r]$
    **Outputs:** Range Search Result
    Client:
 1: Initialize two empty set $ST, I_{RS}$
 2: $C_s \leftarrow$ BinaryTree.LocalSearch($Q, BT$), $C_s = \{C^{(s_1)}, C^{(s_2)}, \ldots, C^{(s_t)}\}$
 3: **if** $v_\ell = C^{(s_1)}_{\min}$ and $v_r = C^{(s_t)}_{\max}$ **then**
 4:     **for** $j \leftarrow 1$ to $t$ **do**
 5:         $(v, \theta) \leftarrow C^{(s_j)}_{info}$
 6:         $bs_j = 1^v || 0^{L-v}$
 7:         Append $(bs_j, \theta)$ into $I_{RS}$
 8:     **end for**
 9:     **Return** $I_{RS}$
10: **else**
11:     Initialize an empty set $W$,Let $C_{s'} := C_s$
12:     **if** $w_\ell \neq C^{(s_1)}_{\min}$ **then**
13:         $K_1 \leftarrow \mathbf{F}(K_c, w_{\ell-1})$
14:         Append $w_{\ell-1}$ into $W$
15:         $C_{s'} \leftarrow C_{s'} \cdot C^{(s_1)}$
16:     **end if**
17:     **if** $w_r \neq C^{(s_t)}_{\max}$ **then**
18:         $K_1 \leftarrow \mathbf{F}(K_c, w_{\ell-1})$
19:         Append $w_r$ into $W$
20:         $C_{s'} \leftarrow C_{s'} \cdot C^{(s_t)}$
21:     **end if**
22: **end if**
    Client and Server:
23: **for** $w$ in $W$ **do**
24:     Read $i$, $sk$, and $D$ from ($\mathbf{C}[w]$, $\mathbf{MSK}[w]$, $\mathbf{D}[w]$)
25:     **if** $i$ is not valid, **then** $\perp$
26:     $sk_R \leftarrow \mathbf{SRE}.ckRev(sk, D)$
27:     Send $(sk_R, D)$ and $tkn = \mathbf{F}(K_s, w)$ to the server
28:     $msk = (sk, D) \leftarrow \mathbf{SRE}$.KGen$(\lambda)$
29:     Renew $\mathbf{C}[w] \leftarrow i+1$, $\mathbf{MSK}[w] \leftarrow msk$, $\mathbf{D}[w] \leftarrow D$
30:     Run $\Sigma_{add}$.Search$(K_\Sigma, w||i, \sigma;$EDB$)$, and the server gets the list $List =$
      $((ct_1, t_1), (ct_2, t_2) \ldots, (ct_l, t_l))$
31:     Set the new value list $NE \leftarrow \emptyset$
32:     **for** $j \in [1, l]$ **do**
33:         Get the encrypted value $Ebs_j \leftarrow \mathbf{SRE}$.Dec$((sk_R, D), ct_j, t_j, e_j)$
34:         **if** $Ebs_j$ is valid **then**
35:             Update $NE \leftarrow NE \bigcup Ebs_j$
36:         **else**
37:             Delete this ciphertext in EDB
38:         **end if**
39:     **end for**
40:     Retrieve cache $OE \leftarrow$ EDB$_{\text{cache}}[tkn]$
41:     Send $EBS \leftarrow NE \bigcup OE$ to Client, and update EDB$_{\text{cache}}[tkn] \leftarrow S$
    Client:
42:     $K_1 \leftarrow \mathbf{F}(K_c, w)$, $bs \leftarrow EBS \oplus \mathbf{H_1}(K_1, w)$
43:     **if** $w = w_{\ell-1}$ **then**
44:         $(v, \theta) \leftarrow C^{(s_1)}_{\text{info}}$
45:         Append $(b_S \oplus 1^v \| 0^{L-v}, \theta)$ into $I_{RS}$
46:     **else**
47:         $(v, \theta) \leftarrow C^{(s_2)}_{\text{info}}$
48:         Append $(b_S, \theta)$ into $I_{RS}$
49:     **end if**
50:     $I_{RS} := I_{RS} \cup \{(1^v \| 0^{L-v}, \theta)\}$,it meets
51:     $\forall C^{(s_i)} \in C_{s'}, (v, \theta) \leftarrow C^{s_i}_{info}$
52: **end for**
53: **Return** $I_{RS}$
```

$\{t, w\}_{w \in \{a-1, b\}}$, *where t is the timestamp. An update query is denoted as $u = (t, op, (w, bs))$, where w specifies the keyword to be updated, op denotes the type of update operation, and bs is a list of document identifiers involved in the update. For a set of search queries Q, the search pattern is defined as $sp(q) = \{t : (t, w)\}_{w \in \{a-1, b\}}$, where $q \in Q$.*

As stated in [9], FP requires that newly added file-keyword pairs remain unlinkable to prior query tokens. However, adversaries may still infer equality among values, revealing more than just their distinction. To address this, d-DSE must ensure additional protection of value information. The following definition is adapted from [39]:

Definition 2 (**Forward Privacy**). *A $\mathscr{L}$-adaptively-secure d-DSE is forward private if its update leakage function, $\mathscr{L}^{Updt}_D$, can be expressible as:*

$$\mathscr{L}^{Updt}_D(in, op) = \mathscr{L}'(op, \{(ind_i, \mu_i)\}), \tag{1}$$

where $\mathscr{L}'$ is a stateless function, and in the set $\{(ind_i, \mu_i)\}$, ind_i represents the modified document's identifier ind_i and μ_i denotes the corresponding number of keywords associated with that document.

Note that the leakage function is defined as $\mathscr{L}^{Updt}_D(w, bs, op) = \mathscr{L}'(op)$.

BP ensures that no information is leaked about files that were added and later deleted. To model BP in d-DSE, Liu et al. [39] introduced two leakage functions: TimeDB(w) and Update(w). The function TimeDB(w) defines the list of all encrypted values v associated with keyword w, excluding any deleted or duplicate entries, and includes the corresponding timestamp t. In contrast, Update(w) reveals the timestamps of update queries for w, returning the set of timestamps when updates were made. Type-II backward privacy is then defined for single keyword queries in d-DSE.

$$\begin{aligned}\text{TimeDB}(w) = \{(t, ind) \mid (t, w, ind, \text{add}) \in Q \wedge \forall t', (t', w, ind, \text{del}) \notin Q \\ \wedge \forall (t, w, ind_i, \text{add}) \in Q, (t, w, ind_j, \text{add}) \in Q, \\ \text{s.t. } ind_i = ind_j \iff i = j\}.\end{aligned}$$

$$\text{Update}(w) = \{t \mid (t, w, ind, add) \in Q \vee (t, w, ind, del) \in Q\}.$$

To handle range queries, we map a range query to two keywords. Using the above functions, we define the BP of RDDSE as follows:

Definition 3 (**General Type-II Backward Privacy**). *A DSSE scheme Σ is considered general Type-II backward secure under $\mathscr{L}$-adaptively if its update and search leakage functions, $\mathscr{L}^{Updt}$ and $\mathscr{L}^{Srch}$, satisfy the following conditions. (Note that the TimeDB(w) used here includes only content before the second '$\wedge$', excluding deleted entries.)*

$$\mathscr{L}^{Updt}(w, ind, op) = \mathscr{L}'(w, op)$$

$$\mathscr{L}^{Srch}(w) = \mathscr{L}''(sp(w), \text{TimeDB}(w), \text{Update}(w))$$

where both $\mathscr{L}'$ and $\mathscr{L}''$ are stateless functions.

Definition 4 (**Backward Privacy**). *A $\mathscr{L}$-adaptively secure RDDSE scheme is Type-II backward privacy if its Update and Search leakage functions, $\mathscr{L}^{Upt}_D$ and $\mathscr{L}^{Srch}_D$ can be expressed as:*

$$\mathscr{L}^{Updt}_D(w, bs, op) = \mathscr{L}'(w, op), w \in \{a-1, b\}$$

$$\mathscr{L}^{Srch}_D(w) = \mathscr{L}''(sp(q), \text{TimeDB}(q), \text{Update}(q)),$$

where the range query is $q = [a,b]$, the bit string bs represents multiple document identifiers for the keyword w to be updated, $\mathscr{L}'$ and $\mathscr{L}''$ are stateless functions.

As described, BP is compatible with d-DSE. Since Type-III backward privacy allows multiple (w, ind) additions to be linked to deletions, revealing the count of identical values could lead to volume pattern leakage. Our RDDSE prevents this leakage through its Distinct with Volume-Hiding Security (DwVH) [39] and the bitmap index, which inherently conceals volume patterns. To demonstrate volume-hiding, we create two signatures of equal size, each with a maximum volume upper bound. The leakage function $\mathscr{L}_D^{\text{Updt}}$ handles additions, while $\mathscr{L}_D^{\text{Srch}}$ involves value types and update timestamps. $\mathscr{L}_D^{\text{Srch}}$ operates independently of input keyword/value pairs, ensuring that both real and dummy tags generate identical responses for value types across both signatures. As a result, the leakage function does not reveal any volume-related information.

Theorem 1 (Adaptive Security of RDSSE) Let F be a secure PRF and H a hash function modeled as a random oracle. Let $\mathscr{L}_D = (\mathscr{L}_D^{\text{Srch}}, \mathscr{L}_D^{\text{Updt}})$ be the leakage function, where $\mathscr{L}_D^{\text{Updt}}(w, bs, \text{op}) = \mathscr{L}'(\text{op})$, for $w \in \{a-1, b\}$, and $\mathscr{L}_D^{\text{Srch}}(w) = \mathscr{L}''(\text{sp}(w)$, TimeDTS$(w)$, Update$(w))$. The scheme Π = (Setup, BuildIndex, Update, Search) is adaptively secure under $\mathscr{L}_D$.

Proof. Similar to the proofs in [18, 20], we construct a sequence of games transitioning from the real experiment $\text{Real}_{\mathscr{A}}^{\text{RDSSE}}(\lambda)$ to the ideal experiment $\text{Ideal}_{\mathscr{A},\mathscr{S},\mathscr{L}}^{\text{RDSSE}}(\lambda)$, bounding the adversary's advantage at each step.

Game G_0 (Real Experiment). This represents the real execution, hence: $\Pr[\text{Real}_{\mathscr{A}}^{\text{RDSSE}}(\lambda) = 1] = \Pr[G_0 = 1]$.

Game G_1 (PRF Replacement). Replace PRF outputs with random values stored in tables for $F(K_s, w)$, $F(K_t, w||v)$, $F(K_t, cnt)$ and $F(K_c, w)$. If a probabilistic polynomial-time (PPT) adversary $\mathscr{A}$ can distinguish G_0 from G_1, we can construct an adversary $\mathscr{B}_1$ against F's security. The distinguishing advantage is bounded by: $\Pr[G_1 = 1] - \Pr[G_0 = 1] \leq 4\text{Adv}_{F,\mathscr{B}_1}^{\text{PRF}}(\lambda)$, where $\text{Adv}_{\mathscr{A},F}^{\text{PRF}}(\lambda)$ is negligible.

Game G_2 (Random Oracle Substitution):. Replace the hash function H_1 with a random oracle $\mathbb{H}_1$, modifying the generation of bs to $bs' \leftarrow \mathbb{H}_1(K_1, w_i)$, and the encryption function e to $e \leftarrow \mathbb{H}_1(K_1, w_i)$. By the random oracle assumption, for a polynomial number $p(\lambda)$ of queries issued by $\mathscr{A}$, we have: $\Pr[G_2 = 1] - \Pr[G_1 = 1] \leq 2p(\lambda)/2^L$, where L is the length of bs', and the probability of $\mathscr{A}$ making a correct guess for bs' is $1/2^L$.

Game G_3 (Forward Privacy Simulation). Replace the Forward Privacy DSSE instance Σ_{add} with its simulator $\mathscr{S}_{\text{add}}^{\text{DSSE}}$. The simulator involves tracks updates through a history list *Uphist*, delaying additions/deletions until the next query. This is valid due to Σ_{add}'s forward privacy and the server's obliviousness to deletions. The distinguishing advantage between G_1 and G_2 is now reduced to the $\mathscr{L}_{FS}$-adaptive forward privacy of Σ_{add}. Hence, there exists a PPT adversary $\mathscr{B}_2$ such that: $\Pr[G_3 = 1] - \Pr[G_2 = 1] \leq \text{Adv}_{\Sigma_{\text{add}},\mathscr{S}_{\text{add}}^{\text{DSSE}},\mathscr{B}_2}^{\mathscr{L}_{FS}}(\lambda)$.

Game G_4 (Ciphertext Deletion). Puncture the previously inserted values in ciphertexts and replace them with 0s. This adjustment impacts only the ciphertexts associated with revoked tags. Consequently, the difference between games G_4 and G_3 can be attributed to the IND-sREV-CPA security of the SRE scheme. A reduction algorithm $\mathscr{B}_3$ can be constructed to derive the revoked tags from Uphist(w), and to simulate encryption for non-deleted values using revoked keys, yielding: $\Pr[G_4 = 1] - \Pr[G_3 = 1] \leq \text{Adv}_{SRE,\mathscr{B}_3}^{\text{IND-}s\text{REV-CPA}}(\lambda)$.

Game G_5 (Update List Construction). The addition list L_{add} holds the addition entries and corresponds to the update history *Uphist*. It is also used as the input for the simulator $\mathscr{S}_{\text{add}}^{\text{DSSE}}$. We first extract leakage information, including *TimeDTS* and *Update* from *Uphist*. Based on this, L_{add} is constructed, and the compressed data structure $\mathscr{D}$ is updated accordingly. This modification does not alter the distribution of G_4, ensuring that: $\Pr[G_5 = 1] = \Pr[G_4 = 1]$.

Game G_6 (Tag Randomization). Instead of deriving tags from keyword-value pairs and saving them in the map *Tags*, we replace them with random strings. The main difference between G_6 and G_5 lies in whether tags repeat. Since each (keyword, value, count) combination is inserted/deleted only once, maintaining a record of every unique tag is unnecessary for consistency. As a result, it ensures that: $\Pr[G_6 = 1] = \Pr[G_5 = 1]$.

Game G_7 (Encryption Replacement). Replace symmetric encryption outputs with random strings. The distinguishing factor between G_7 and G_6 arises from the advantage: $\Pr[G_7 = 1] - \Pr[G_6 = 1] \leq \text{Adv}_{E,\mathscr{B}_4}^{\text{IND-CPA}}(\lambda)$.

Simulator. To construct a simulator from G_7, we replace the keyword w with min sp(w). When building L_{add} and constructing $\mathscr{D}$, the leakage information *TimeDTS* and *Update* can be used as inputs for *Search*. Therefore, the simulator does not need to keep track of the updates anymore. In this condition, G_7 can be efficiently simulated by the simulator with the leakage function $\mathscr{L}$, ensuring that: $\Pr[G_7 = 1] = \Pr[\text{Ideal}_{\mathscr{A},\mathscr{S},\mathscr{L}}^{\text{RDSSE}}(\lambda) = 1]$.

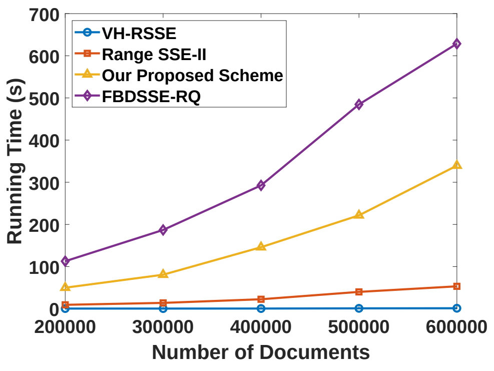
Conclusion. Summing the advantages across all games, it confirms the security of the proposed scheme. the PPT adversary's total advantage is bounded by:

$$\begin{aligned}&\left|\Pr[\text{Real}_{\mathscr{A}}^{\text{RDSSE}}(\lambda) = 1] - \Pr[\text{Ideal}_{\mathscr{A},\mathscr{S},\mathscr{L}}^{\text{RDSSE}}(\lambda) = 1]\right| \leq \\ &\quad 4\text{Adv}_{F,\mathscr{B}_1}^{\text{PRF}}(\lambda) + 2p(\lambda)/2^L + \text{Adv}_{\Sigma_{\text{add}},\mathscr{S}_{\text{add}}^{\text{DSSE}},\mathscr{B}_2}^{\mathscr{L}_{FS}}(\lambda) \\ &\quad + \text{Adv}_{\text{SRE},\mathscr{B}_3}^{\text{IND-sREV-CPA}}(\lambda) + \text{Adv}_{E,\mathscr{B}_4}^{\text{IND-CPA}}(\lambda).\end{aligned}$$

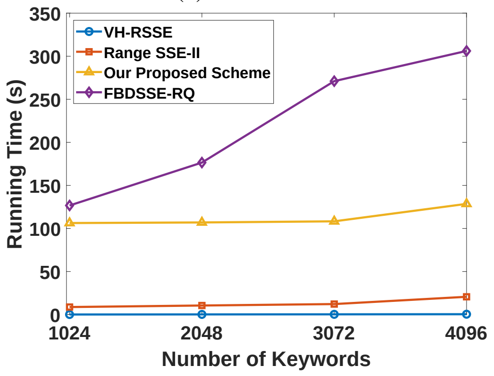
□

V. Performance Evaluation

This section details the implementation of our scheme using Python (v3.6) and C++ on Ubuntu 16.04 with 16GB memory. The client and server are simulated on the same machine to eliminate network instability effects. The symmetric revocable encryption component is implemented in C++, while other



(a) $m = 4096$



(b) $d = 400k$

Fig. 5: Comparison of Running Time for Index Generation

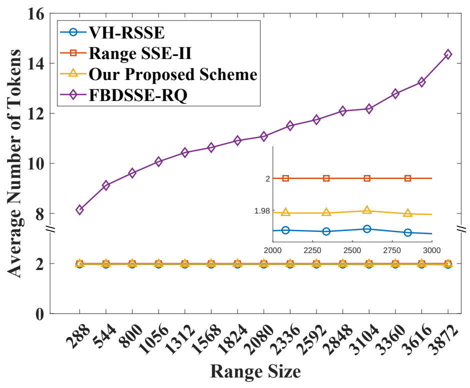


Fig. 6: Required tokens.

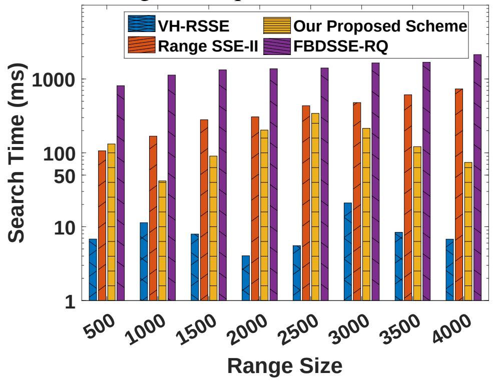


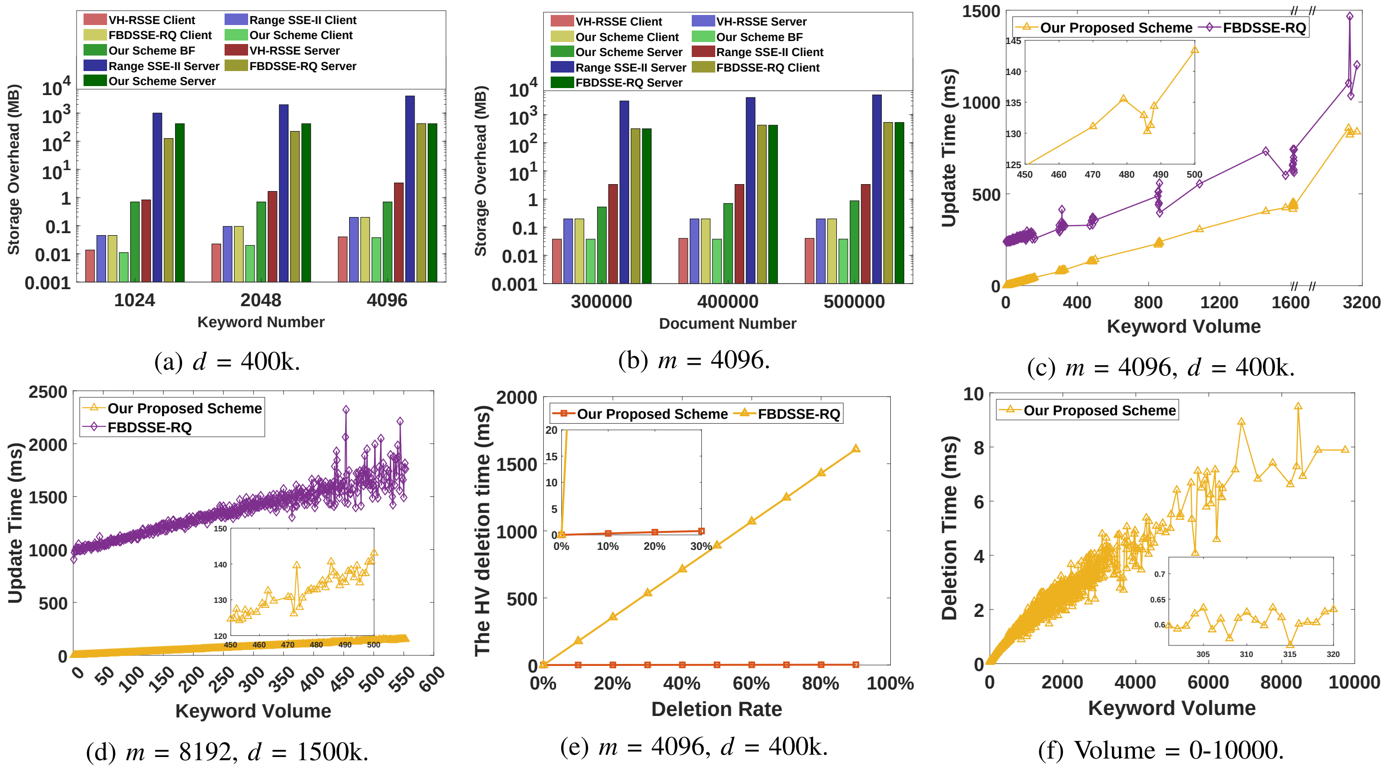
Fig. 7: Compare search time.

components are in Python. We use AES (16-byte key) for encryption, SHA-256 for hashing, and an AES-based pseudorandom function (PRF). We compare our approach with recent schemes [18, 20, 25] and evaluate its performance. The Wikipedia dataset [40], containing 4,565,948 entries, is used for testing, and distinct word-document pairs are extracted using the Wikipedia extractor and Python NLTK package. The default bitmap length L is 6,264 unless otherwise specified.

Table I shows that FBDSSE-RQ [18] outperforms FPDSSE-RQ-I and DSSE-RQ-II [10], while Range SSE-II [25] improves on Range SSE-I [25]. VH-RSSE [20] is the most advanced range SSE scheme. We compare our scheme with Range SSE-II, VH-RSSE, and FBDSSE-RQ using datasets with varying document (d) and keyword (m) counts. First, we compare the time to generate encrypted indexes and local binary trees (Fig. 5). VH-RSSE and Range SSE-II exhibit the fastest index generation times, followed by our scheme, while FBDSSE-RQ is the slowest due to its more complex keyword-document indexing. We use DSSE instead of FBDSSE-RQ's homomorphic encryption, which reduces index generation time. Additionally, FBDSSE-RQ's disorganized keyword identifiers and long bit strings slow down bitmap creation, while our inverted index improves efficiency. When the number of keywords is fixed at 4,096, index generation time increases with the number of documents. VH-RSSE grows slowest, Range SSE-II fastest, and our scheme performs best among range DSSE (Fig.5(a)). With 400k documents, our scheme's runtime increases more slowly than FBDSSE-RQ but slightly faster than VH-RSSE and Range SSE-II (Fig. 5(b)), due to its dependence on both document count and bitmap length.

Fig. 6 presents the analysis of how the average number of search tokens varies with different range sizes. In the experiment, we set $d = 400k$ documents and $m = 4,096$keywords. The range size, defined as the number of distinct values within a range, is denoted as q. A naive approach generates $|q|$ search tokens, corresponding to each distinct value. To optimize this, FBDSSE-RQ [18] employs the Best Range Cover (BRC) strategy, reducing the token count to $O(\log q)$. Range SSE-II [25], utilizing an order-weighted inverted index, further minimizes the token count to 2. Building on this, VH-RSSE [20] and our scheme restrict the token count to a maximum of 2. This is achievable by ensuring that if the query range aligns perfectly with a node's range in the binary tree, the client can directly access the bitmap index locally without transmitting tokens to the server. This optimization significantly enhances search efficiency and conserves computational resources, especially when dealing with large-scale datasets.

Fig. 7 illustrates the search time comparison among different schemes. Due to the prolonged search times of Range SSE-II [25] and FBDSSE-RQ [18] with large datasets, we conduct experiments with 4,096 keywords and 400k docu-



(a) d = 400k.

(b) m = 4096.

(c) m = 4096, d = 400k.

(d) m = 8192, d = 1500k.

(e) m = 4096, d = 400k.

(f) Volume = 0-10000.

Fig. 8: Comparison of storage overhead, update time cost, and deletion time cost.

ments. The results indicate that, apart from FBDSSE-RQ, which exhibits the longest search time, the remaining range query schemes demonstrate efficient performance. Notably, VH-RSSE [20] and our scheme exhibit minimal sensitivity to range size. This efficiency is attributed to our scheme's restriction of search tokens to a maximum of 2, irrespective of range size, and the segmentation of the range into non-overlapping subranges, which mitigates the impact of large document identifiers. Consequently, each keyword in the inverted index is associated with only a small subset of document identifiers. In contrast, search times for Range SSE-II and FBDSSE-RQ increase as the range size grows, owing to their extensive encrypted indexes and elevated communication overhead. Furthermore, FBDSSE-RQ generates additional search tokens with larger ranges, rendering the upload process impractical for real-world applications.

We evaluate the storage overhead across different schemes, where overhead includes both client and server storage. As shown in Fig. 8(a), when the number of documents $d = 400k$, the client storage overhead for all schemes increases with the number of keywords. This is due to the growing number of nodes stored in the client's tree structure, with our scheme also incorporating a bloom filter to track various states. For VH-RSSE [20], Range SSE-II [25], and FBDSSE-RQ [18], the server stores an encrypted bit string for each keyword. In contrast, our scheme also stores symmetric revocable encryption ciphertext for each encrypted record. When the number of keywords m reaches 4,096, our scheme requires less server storage compared to Range SSE-II and FBDSSE-RQ. This is because, in both Range SSE-II and FBDSSE-RQ, each keyword inevitably corresponds to the entire document set. Fig. 8(b) demonstrates that even as d increases, the storage overhead remains manageable when the number of keywords is fixed at $m = 4{,}096$. This suggests that the additional storage incurred by our scheme is justified, as it facilitates volume-hiding and non-interactive deletion functions, while maintaining overall performance. Therefore, this trade-off proves to be both reasonable and advantageous.

Additionally, since Range SSE-II [25] and VH-RSSE [20] do not support data updates (i.e., additions), we limit the comparison to FBDSSE-RQ [18]. Experiments are conducted on a smaller dataset with m = 4,096 keywords and d = 400k documents, testing the update time at different keyword volumes. Fig. 8(c) shows that update time increases with keyword volume, with FBDSSE-RQ being the most affected, while our scheme experiences the least impact. To evaluate scalability, we test a larger dataset with m = 8,192 keywords and d = 1,500k documents (Fig. 8(d)). As the dataset size increases, the update time for FBDSSE-RQ grows significantly (from 356.67 ms to 1,605.03 ms) under the same keyword volume (e.g., keyword volume = 400). In contrast, our scheme's update time remained stable at around 143 ms. This is due to the fact that FBDSSE-RQ stores document identifiers for the entire dataset, and its BRC technique results in a higher number of tree nodes and increased computational overhead as the number of keywords grows. In contrast, our scheme uses a unique constant bit string for each keyword, making it less sensitive to changes in keyword volume or dataset size. The results highlight that our scheme is particularly well-suited for large-scale datasets.

Since Range SSE-II [25] and VH-RSSE [20] do not support data updates (deletions), and since Range SSE-II cannot distinguish values to be deleted due to the weights in the ordered-weighted inverted index, we focus on comparing the deletion performance of FBDSSE-RQ [18]. In Fig. 8(e), we compare the deletion times for the Highest-Volume (HV) keyword/document pairs between FBDSSE-RQ and our proposed scheme, using a dataset with $d = 400k$ documents and $m = 4,096$ keywords. The results show that at a 10% deletion ratio, the deletion times for FBDSSE-RQ and our scheme are 177.7 ms and 0.28 ms, respectively. At a 90% deletion ratio, the times increase to 1,607.5 ms and 2.17 ms, making our scheme approximately 740.78× faster. To further validate the efficiency of our deletion operation, we conduct tests on the real-world Wikipedia dataset (4,565,948 records), as shown in Fig. 8(f). The deletion time increases gradually with keyword volume but remains reasonable ($\leq$ 9.49 ms). When the keyword volume is $\leq 5,000$, the deletion time does not exceed 6 ms, demonstrating the efficiency of our scheme. Compared to existing techniques, as shown in Fig. 8(e), the deletion time of our scheme is markedly lower, a result primarily attributed to the advantages of non-interactive deletion.

VI. Conclusion

This paper presents a dynamic searchable symmetric encryption scheme optimized for range queries over large datasets. By integrating a bitmap structure with an order-accumulated inverted index and symmetric revocable encryption, the scheme simultaneously ensures forward and backward privacy, minimizes token size, mitigates volume pattern leakage, and supports efficient non-interactive deletions. Our security analysis confirms that the scheme achieves the desired security properties, while experimental results demonstrate its practical efficiency and scalability. In future work, we plan to further enhance update efficiency and explore support for more complex query types under dynamic settings.

References

[1] A. Steve, "H1, 2024 healthcare data breach report," 2024. [Online]. Available: https://www.hipaajournal.com/h1-2024-healthcare-data-breach-report/

[2] Yahoo, "Yahoo! inc. customer data security breach litigation settlement," 2023. [Online]. Available: https://yahoodatabreachsettlement.com/

[3] C. Gentry, S. Halevi, and N. P. Smart, "Fully homomorphic encryption with polylog overhead," in *Proc. Annu. Inter. Conf. Theory Appl. Cryptogr. Techn.* Springer, 2012, pp. 465–482.

[4] D. Boneh, A. Sahai, and B. Waters, "Functional encryption: Definitions and challenges," in *Proc. Theory Cryptography: 8th Theory Cryptogr. Conf.* Springer, 2011, pp. 253–273.

[5] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proc. IEEE Symp. Secur. Privacy (S&P).* IEEE, 2000, pp. 44–55.

[6] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *Proc. 13th ACM Conf. Comput. Commun. Secur.*, 2006, pp. 79–88.

[7] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," *Cryptology ePrint Archive*, 2014.

[8] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: the power of {File-Injection} attacks on searchable encryption," in *Proc. USENIX Secur. Symp.*, 2016, pp. 707–720.

[9] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 1465–1482.

[10] C. Zuo, S.-F. Sun, J. K. Liu, J. Shao, and J. Pieprzyk, "Dynamic searchable symmetric encryption with forward and stronger backward privacy," in *Proc. Euro. Symp. Res. Comput. Secur.* Springer, 2019, pp. 283–303.

[11] B. Wang, S. Yu, W. Lou, and Y. T. Hou, "Privacy-preserving multi-keyword fuzzy search over encrypted data in the cloud," in *IEEE. INFOCOM Conf. Comput. Commun.* IEEE, 2014, pp. 2112–2120.

[12] Z. Fu, X. Wu, C. Guan, X. Sun, and K. Ren, "Toward efficient multi-keyword fuzzy search over encrypted outsourced data with accuracy improvement," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 12, pp. 2706–2716, 2016.

[13] L. Chen, Y. Xue, Y. Mu, L. Zeng, F. Rezaeibagha, and R. H. Deng, "Case-sse: Context-aware semantically extensible searchable symmetric encryption for encrypted cloud data," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1011–1022, 2022.

[14] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *Proc. Cryptology–CRYPTO: 33rd Annu. Cryptol. Conf.* Springer, 2013, pp. 353–373.

[15] S. Kamara and T. Moataz, "Boolean searchable symmetric encryption with worst-case sub-linear complexity," in *Proc. Cryptology–EUROCRYPT: 36th Annu. Inter. Conf. Theory Appl. Cryptogr. Techn.* Springer, 2017, pp. 94–124.

[16] S. Lai, S. Patranabis, A. Sakzad, J. K. Liu, D. Mukhopadhyay, R. Steinfeld, S.-F. Sun, D. Liu, and C. Zuo, "Result pattern hiding searchable encryption for conjunctive queries," in *Proc. ACM SIGSAC Conf.*

Comput. Commun. Secur., 2018, pp. 745–762.

[17] C. Zuo, S.-F. Sun, J. K. Liu, J. Shao, and J. Pieprzyk, "Dynamic searchable symmetric encryption schemes supporting range queries with forward (and backward) security," in *Proc. Comput. Secur. 23rd Eur. Symp. Res. Comput. Secur. ESORICS.* Springer, 2018, pp. 228–246.

[18] C. Zuo, S.-F. Sun, J. K. Liu, J. Shao, J. Pieprzyk, and L. Xu, "Forward and backward private dsse for range queries," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 01, pp. 328–338, 2022.

[19] J. Wang and S. S. Chow, "Forward and backward-secure range-searchable symmetric encryption," *Proceedings on Privacy Enhancing Technologies*, vol. 1, pp. 28–48, 2022.

[20] F. Liu, K. Xue, J. Yang, J. Zhang, Z. Huang, J. Li, and D. S. Wei, "Volume-hiding range searchable symmetric encryption for large-scale datasets," *IEEE Transactions on Dependable and Secure Computing*, 2023.

[21] B. Wang, M. Li, and L. Xiong, "Fastgeo: Efficient geometric range queries on encrypted spatial data," *IEEE transactions on dependable and secure computing*, vol. 16, no. 2, pp. 245–258, 2017.

[22] G. Xu, H. Li, Y. Dai, K. Yang, and X. Lin, "Enabling efficient and geometric range query with access control over encrypted spatial data," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 4, pp. 870–885, 2018.

[23] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner, "Rich queries on encrypted data: Beyond exact matches," in *Proc. Computer Security–ESORICS: 20th Euro. Symp. Res. Comput. Secur.* Springer, 2015, pp. 123–145.

[24] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis, "Practical private range search revisited," in *Proc. Inter. Conf. Manag. Data.*, 2016, pp. 185–198.

[25] B. Wang and X. Fan, "Search ranges efficiently and compatibly as keywords over encrypted data," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 6, pp. 1027–1040, 2016.

[26] J. Wang and S. S. Chow, "Forward and backward-secure range-searchable symmetric encryption," *Cryptology ePrint Archive*, 2019.

[27] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia, "Response-hiding encrypted ranges: Revisiting security via parametrized leakage-abuse attacks," in *Proc. IEEE Symp. Secur. Privacy (S&P).* IEEE Computer Society, 2021, pp. 1502–1519.

[28] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre, "{SEAL}: Attack mitigation for encrypted databases via adjustable leakage," in *Proc. USENIX Secur. Symp.*, 2020, pp. 2433–2450.

[29] J. Ning, X. Huang, G. S. Poh, J. Yuan, Y. Li, J. Weng, and R. H. Deng, "Leap: leakage-abuse attack on efficiently deployable, efficiently searchable encryption with partially known dataset," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2021, pp. 2307–2320.

[30] L. Xu, L. Zheng, C. Xu, X. Yuan, and C. Wang, "Leakage-abuse attacks against forward and backward private searchable symmetric encryption," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2023, pp. 3003–3017.

[31] P. Grubbs, M.-S. Lacharité, B. Minaud, and K. G. Paterson, "Pump up the volume: Practical database reconstruction from volume leakage on range queries," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 315–331.

[32] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proc. ACM Conf. Comput. Commun. Secur*, 2012, pp. 965–976.

[33] X. Zhang, W. Wang, P. Xu, L. T. Yang, and K. Liang, "High recovery with fewer injections: Practical binary volumetric injection attacks against dynamic searchable encryption," in *Proc. 32nd USENIX Secur. Symp.*, 2023, pp. 5953–5970.

[34] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," *Cryptology ePrint Archive*, 2013.

[35] V. Vo, X. Yuan, S.-F. Sun, J. K. Liu, S. Nepal, and C. Wang, "Shielddb: An encrypted document database with padding countermeasures," *IEEE Transactions on Knowledge & Data Engineering*, vol. 35, no. 04, pp. 4236–4252, 2023.

[36] A. Bienstock, S. Patel, J. Y. Seo, and K. Yeo, "{Near-Optimal} oblivious {Key-Value} stores for efficient {PSI},{PSU} and {Volume-Hiding}{Multi-Maps}," in *Proc. 32nd USENIX Secur. Symp.*, 2023, pp. 301–318.

[37] M. Ando and M. George, "On the cost of suppressing volume for encrypted multi-maps," *Proceedings on Privacy Enhancing Technologies*, vol. 1, p. 22.

[38] S.-F. Sun, R. Steinfeld, S. Lai, X. Yuan, A. Sakzad, J. K. Liu, S. Nepal, and D. Gu, "Practical non-interactive searchable encryption with forward and backward privacy," in *Proc. Usenix Net. Distri. Syst. Secur. Symp.* The Internet Society, 2021.

[39] D. Liu, W. Wang, P. Xu, L. T. Yang, B. Luo, and K. Liang, "d-dse: Distinct dynamic searchable encryption resisting volume leakage in encrypted databases," *arXiv preprint arXiv:2403.01182*, 2024.

[40] Wikimedia, "Wikimedia downloads," 2025. [Online]. Available: https://dumps.wikimedia.org/