Hindawi
Security and Communication Networks
Volume 2021, Article ID 9703969, 12 pages
https://doi.org/10.1155/2021/9703969

Research Article

Database Padding for Dynamic Symmetric Searchable Encryption

Ruizhong Du,[1,2] Yuqing Zhang ,[1] and Mingyue Li[3]

[1]*School of Cyber Security and Computer Science, HeBei University, BaoDing, China*
[3]*School of Cyber Security, NanKai University, TianJin, China*
[2]*Key Laboratory of High-Confidence Information System of Hebei Province, BaoDing, China*

Correspondence should be addressed to Yuqing Zhang; yqzhang@hbu.edu.cn

Received 16 September 2021; Accepted 6 December 2021; Published 31 December 2021

Academic Editor: Qi Jiang

Dynamic symmetric searchable encryption (DSSE) that enables the search and update of encrypted databases outsourced to cloud servers has recently received widespread attention for leakage-abuse attacks against DSSE. In this paper, we propose a dynamic database padding method to mitigate the threat of data leakage during the update operation of outsourcing data. First, we introduce an outlier detection technology where bogus files are generated for padding according to the outlier factors, hiding the document information currently matching search keywords. Furthermore, we design a new index structure suitable for the padded database using the bitmap index to simplify the update operation of the encrypted index. Finally, we present an application scenario of the padding method and realize a forward and backward privacy DSSE scheme (named PDB-DSSE). The security analysis and simulation results show that our dynamic padding algorithm is suitable for DSSE scheme and PDB-DSSE scheme maintains the security and efficiency of the retrieval and update of the DSSE scheme.

1. Introduction

With the rapid development of Internet technology, the concept of the Internet of Things (IoT) has attracted widespread attention once it was proposed. Due to the computing power bottleneck of IoT terminals, IoT terminal users must rely on cloud platforms to store and process private data. Therefore, the client loses control of private data, which will lead to the risk of leakage of private data (such as medical records and identity information) and endanger the security of the data. The most intuitive way to protect private data is to encrypt it and store it on the cloud platform before outsourcing it. However, it is difficult to search ciphertext data, which impairs data availability. To solve this problem, Song et al. [1] first proposed searchable encryption (SE) technology.

Previous SSE solutions have been static and could not support dynamic database updates, such as addition or deletion of outsourced files, limiting the application of SSE. Therefore, dynamic symmetric searchable encryption (DSSE) technologies were proposed [2]. However, the process of dynamic update will cause data leakage, and some existing attacks will pose risks [3]. For instance, file injection attacks can obtain the user's query content by injecting a small part of the updated files [4].

To solve the above problems, Stefanov et al. [5] informally introduced the security concepts of forward privacy and backward privacy. Forward privacy is the security involved when adding a document/keyword pair. It ensures that the newly added text will not be searched by the previous search token. Backward privacy is the security involved in the deletion operation. It guarantees that the deleted documents will not be searched. In other words, if, after searching w, adding a document/keyword pair (w, f) and deleting it before the next query, then searching for the keyword w again in the future will not display f. Bost et al. [5] give the formal definition of backward privacy according to the different degrees of leakage. In recent years, many excellent forward and backward privacy DSSE schemes have been designed [6–17]. However, most of the schemes have low search efficiency and involve a complex index update process. Furthermore, leaks caused by access patterns can easily recover the underlying keywords of the search token, and while many schemes use oblivious RAM (ORAM) to solve this problem, ORAM is less efficient.

Database padding technology is a simple and efficient measure to solve the above problems that is suitable for real large data sets. Nevertheless, studies of padding schemes mainly focus on how to design efficient padding methods [3,18,19] that are not suitable for dynamic databases.

1.1. Our Contributions. In this paper, we design a dynamic database padding algorithm that can be applied to DSSE schemes and give the application scenarios. We have implemented a DSSE scheme with forward privacy and backward privacy (named PDB-DSSE) to prove that the dynamic database padding algorithm proposed in this paper is suitable for the DSSE scheme. In summary, our contributions are as follows:

We introduce outlier detection technology to design a dynamic database padding algorithm that is suitable for DSSE schemes. Specifically, each bogus file is sampled and verified by the outlier detection algorithm that pads the database, so that the generated bogus file and the real file are indistinguishable and finally realize result hiding.

We design a new index structure suitable for the padding database using the bitmap index that consists of two parts that represent real files and bogus files, respectively. In the updating process, the index can be modified by homomorphic addition operations, simplifying the update operation of the index.

We instantiated the application scenario of the dynamic padding algorithm, implemented PDB-DSSE, and tested its performance. Experimental results and security analysis show that the dynamic database padding algorithm proposed in this paper is suitable for the DSSE scheme, and PDB-DSSE maintains the security and search and update efficiency of the DSSE scheme.

1.2. Related Works. Song et al. [1] first proposed SSE. After encrypting the keyword, when searching for files matching the keyword w, the algorithm sends the keyword w and its corresponding key k_i of each position i where the keyword may appear to the server. The server compares it with the encrypted keywords one by one to obtain the result. For a file of length N, the encryption and search algorithm of this scheme requires a stream cipher and block cipher. The time complexity is $O(N)$. Subsequently, research on SSE has been extended in many aspects, such as conjunctive search [15], rank search [11], range query [14], small client storage SSE [16], and verifiable schemes [11,12].

To support dynamic updates, DSSE was proposed in [2]. The early DSSE schemes were vulnerable to potential attacks [3], such as file injection attacks [4]. To reduce the threat posed by this attack, the concepts of forward privacy and backward privacy were proposed in [5]. In 2016, Bost [6] gave the formal definition of forward privacy and a forward privacy DSSE scheme $\sum o\varphi os$. This scheme achieves certain improvements in efficiency and security by using simple cryptographic tools (i.e., pseudorandom functions and trapdoor primitives) without relying on the ORAM structure. In the next year, Bost et al. [7] gave a formal definition of backward security according to different levels of leakage and then gave four forward and backward privacy schemes. Among them, the Fides scheme they designed achieves the second level of backward privacy. The second scheme, Diana, is very efficient but only implements Type-III backward privacy. Diana is modified to a backward privacy scheme $\text{Diana}_{\text{del}}$ that only requires two roundtrips. The third scheme Janus is the first proposed backward privacy scheme with only a single roundtrip, but this scheme also only implements Type-III backward privacy. The last scheme, Moneta, implements Type-I backward privacy. However, the computational overhead and communication overhead of this scheme are very large due to the use of the TWORAM structure. In 2018, Sun et al. [8] constructed a new primitive called symmetric puncturable encryption (SPE). They further use the common primitive to propose a noninteractive backward privacy DSSE scheme Janus++. This scheme is more efficient than Janus. However, to achieve backward privacy, it hands over much work to clients, greatly increasing their storage and computing overhead.

In 2018, Chamani et al. [9] proposed several dynamic symmetric searchable encryption schemes, namely *Mitra*, Orion, and Horus. Among them, Orion achieves the highest level of backward privacy: Type-I backward. It requires $O(\log N)$ rounds of interaction, and the search process requires $O(n_w \log^2 N)$ steps. Horus improves the efficiency of Orion. However, it only reaches the third level of backward privacy. In 2019, Zuo et al. [13] designed a forward privacy and backward privacy DSSE scheme that implements Type-I^- backward privacy that is somewhat stronger than Type-I. To support a larger database, they further extend it to a multiblock setting. Moreover, they extended the first scheme to support range query in [14].

The early studies on database padding focused on the design of an efficient padding method. Cash et al. [3] proposes a heuristic padding method, which pads the number of files corresponding to keywords by multiples. However, this padding method introduces unnecessary padding. In response to the problem of padding overhead, Bost and Fouque [18] divide the clusters according to frequency and pad the clusters with similar frequencies as one cluster. The algorithm achieves the smallest padding cost while preventing counting attacks. Xu et al. [19] proposed the concept of relative entropy to measure the distance between the original keyword distribution and padded keyword distribution and proposed a bogus file generation algorithm to strengthen database padding. As a result, it is almost impossible for us to distinguish a bogus document from a real document.

1.3. Organization. The content structure in the rest of this article is as follows: We give the background knowledge, including the bitmap index structure that we used, the definition of the DSSE and its security model, and the notation used in the work in Section 2. We describe the dynamic padding algorithm in Section 3. We describe the PDB-DSSE scheme in Section 4. The security analysis is given in Section 5. The performance analysis and simulation experiment results are described in Section 6. Finally, Section 7 summarizes the paper.

2. Preliminaries

2.1. Bitmap Index. The index structure of PDB-DSSE is based on the bitmap index [14]. Assuming that the database can store up to N files, each keyword corresponds to an N-bit string S. If the keyword exists in file f_i, the i-th bit of the S is set to 1; otherwise, it is set to 0. As shown in Figure 1(a), the maximum number of files that the database can store is 6, that is, $N = 6$. At this time, the bit string $(000100)_2$ indicates that file f_2 exists. As shown in Figure 1(b), if file f_3 is added, a bit string $(001000)_2$ needs to be generated and added to the initial S. If we perform a delete operation to delete the file f_2, as shown in Figure 1(c), we need to generate $-(000100)_2 = (111100)_2 \bmod 2^6$ and add it to the initial bit string. In our PDB-DSSE scheme, we use the database padding algorithm to pad the database with bogus files and generate the corresponding bitmap index. Assuming that the index length is N, the first k bits represent real files (k is used to indicate the maximum number of files that the real database can store). The $n - k$ bits represent the padded bogus files (as shown in Figure 1(a)). The real database can store up to 4 files, that is, $k = 4$, and the calculation rules are the same as the bitmap index. The detailed algorithm and calculation process are described in Section 3.

2.2. DSSE Definition. In this section, we define the DSSE scheme. We also give its security model and formally define forward privacy and backward privacy.

The database $\mathrm{DB} = (f_i, w_i)_{i=1}^{k}$, where f_i is the file identifier, w_i is the keyword set, and k represents how many real files are stored in DB. |DB| represents the total number of keyword/file identifier pairs. $W = \cup_{i=1}^{k} w_i$ is used to represent the collection of all distinct keywords in DB.

A DSSE scheme DSSE=(Setup, Search, Update) consists of the following.

$\mathrm{Setup}(1^\lambda)$: this algorithm enters a security parameter λ. It outputs a client's state σ and an encrypted database EDB which is uploaded to the cloud server.

$\mathrm{Search}(q, \sigma; \mathrm{EDB})$: this protocol requires interaction between the client and the server. The client has stored the state σ and enters the query q. The server retrieves EDB through the search token and returns the matching result to the client.

$\mathrm{Update}(\sigma, \mathrm{op}, \mathrm{ind}; \mathrm{EDB})$: this protocol requires interaction between the client and the server. The update operations it supports include add and delete. If the client wants to perform an update operation and add (or delete) a bunch of (w, f), the server will update EDB and add (or delete) the corresponding ciphertext. At the same time, the client updates the local state σ.Finally, the server updates EDB.

2.3. Security Model. Given a DSSE scheme defined in Section 2.2, we will define the real game REAL and the ideal game IDEAL to give its security model. REAL reflects the behavior of the original DSSE scheme, and IDEAL reflects what the simulator $\mathcal{S}$ does. $\mathcal{S}$ takes the leaked function of the scheme as input. For adversary $\mathcal{A}$, the leak function is defined as $\mathcal{L} = (\mathcal{L}^{\mathrm{stup}}, \mathcal{L}^{\mathrm{srch}}, \mathcal{L}^{\mathrm{updt}})$, $\mathcal{L}^{\mathrm{stup}}$ being the information that $\mathcal{A}$ can learn when the setup algorithm is executed, $\mathcal{L}^{\mathrm{srch}}$ being the information that $\mathcal{A}$ can learn when executing the search protocol, and $\mathcal{L}^{\mathrm{updt}}$ being the information that $\mathcal{A}$ can learn when executing the update protocol. Games REAL and IDEAL are defined as follows.

$\mathrm{REAL}_{\mathcal{A}}^{\mathrm{DSSE}}(\lambda)$: upon input to a database DB that is chosen by the adversary $\mathcal{A}$, it runs $\mathrm{Setup}(1^\lambda, \mathrm{DB})$ to obtain the EDB. $\mathcal{A}$ initiates a series of search queries s (or update query (op, ind)). Finally, $\mathcal{A}$ returns the experimental result b, $b \in (0, 1)$.

$\mathrm{IDEAL}_{\mathcal{A},\mathcal{S}}^{\mathrm{DSSE}}(\lambda)$: the simulator $\mathcal{S}$ executes the leaked function $\mathcal{L}^{\mathrm{stup}}(\lambda, \mathrm{DB})$ for a series of search queries or update queries for the difference (op, ind) of $\mathcal{A}$, and the simulator $\mathcal{S}$ inputs $\mathcal{L}^{\mathrm{srch}}$ and $\mathcal{L}^{\mathrm{updt}}$, respectively, and returns the output result to $\mathcal{A}$. Finally, $\mathcal{A}$ returns the experimental result b, $b \in (0, 1)$.

Definition 1 (see [14] (adaptive security of DSSE schemes)). A DSSE scheme is $\mathcal{L}$-adaptively secure with respect to leakage function $\mathcal{L}$, iff for any probabilistic polynomial time (PPT) adversary $\mathcal{A}$ issuing a polynomial number of queries q, there exists a stateful PPT simulator $\mathcal{S}$, such that

$$\Pr\left[\mathrm{REAL}_{\mathcal{A}}^{\mathrm{DSSE}}(\lambda) = 1\right] - \Pr\left[\mathrm{IDEAL}_{\mathcal{A},\int}^{\mathrm{DSSE}}(\lambda) = 1\right] \leq \mathrm{negl}(\lambda). \tag{1}$$

2.4. Forward Privacy. Forward security guarantees that the updates that occur cannot be associated with the operations that occurred before.

Definition 2 (see [6] (forward privacy)). If a dynamic searchable encryption scheme D's update leak function $\mathcal{L}^{\mathrm{updt}}$ can be written as

$$\mathcal{L}^{\mathrm{updt}}(\mathrm{op}, w, S) = \mathcal{L}'(\mathrm{op}, S), \tag{2}$$

D is adaptively forward privacy, whhere w is a set of keywords of update operation and S is the update file index.

2.5. Backward Privacy. Zuo et al. [13] defined Type-I$^-$ backward privacy. Type-I$^-$ will reveal the file information containing the keyword w, the total number of updates of w, and the update time of each update on w. Our PDB-DSSE scheme uses "0" and "1" bit string to indicate whether the file exists or not, and its operations of addition and deletion use the same module, so it will not reveal when the file was inserted. In summary, our scheme achieves Type-I$^-$ backward privacy.

For example, consider the following series of update operations in a single-keyword query system, which occur in sequence: search for the files corresponding to the keyword w at time 1, add file f_1 for the keyword w at time 2, add file f_2 for keyword w at time 3, add file f_3 for keyword w at time 4, delete file f_1 at time 5, and finally at time 6, search again

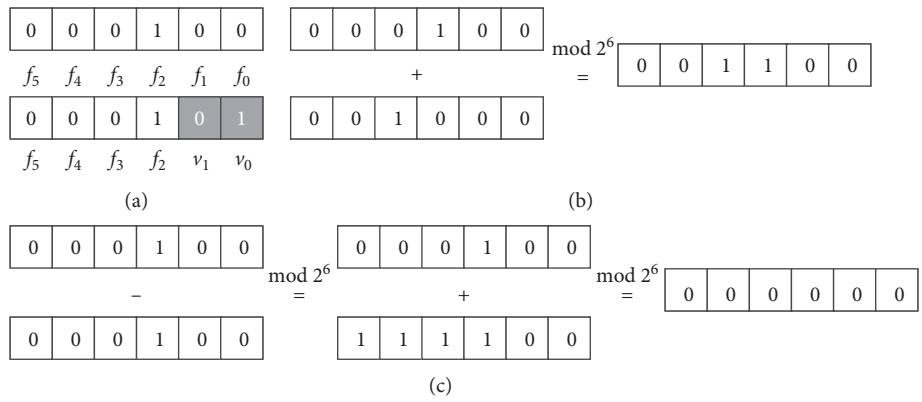

Figure 1: Bitmap index. (a) Bitmap index. (b) Add. (c) Delete.

for the document corresponding to the keyword w. Finally, perform the above operations. Type-I$^-$ backward privacy revealed that searching for w will return files f_2 and f_3. And, a total of 4 updates occurred at the above time, which occurred at times 2, 3, 4, and 5.

To formally define Type-I$^-$, for the search query list Q and the timestamp t, the search pattern is defined as $\mathrm{sp}(w) = t\colon\ (t, w) \in Q'$. The search mode reveals whether the search keyword w is repeated. It is also necessary to define a new leak function TimePDB. For the keyword w, TimePDB (w) lists all the updated timestamps t of w. For update query Q',

$$\mathrm{Time\,PDB}(w) = \{t\colon\ (t, \mathrm{op},\ (w, S)), Q'\}. \tag{3}$$

Definition 3 (see [13] (backward privacy)). If a dynamic searchable encryption scheme D's search and update leak functions $\mathscr{L}^{\mathrm{updt}}$ and $\mathscr{L}^{\mathrm{updt}}$ can be written as

$$\begin{aligned} \mathscr{L}^{\mathrm{updt}}(\mathrm{op}, w, S) &= \mathscr{L}'(\mathrm{op}), \\ \mathscr{L}^{\mathrm{srch}}(w) &= \mathscr{L}''(\mathrm{sp}(w), \mathrm{Time\,PDB}(w)). \end{aligned} \tag{4}$$

D is Type-I$^-$ adaptively backward privacy.

2.6. Notation. The notation used in the work is given in Table 1.

3. Dynamic Database Padding Algorithm

To solve the problem of data leakage during the update of DSSE schemes, we introduced outlier detection to design a dynamic database padding algorithm that can be used in the DSSE scheme. The padding algorithm can hide the information of the files currently matching the query keyword. In this scheme, real files and bogus files are represented by bitmap indexes, simplifying the process of update operations.

The dynamic database padding algorithm includes the padding database generation algorithm PDB-Gen and the padding database update algorithm PDB-Update.

3.1. PDB ⟵ PDB − Gen (DB). Upon input to a database DB, it outputs a padding database PDB, where PDB $= \{V_1, \ldots, V_l\}, 0 \le l \le n - k$. The algorithm is described in Algorithm 1. It initializes two empty sets PDB and G that store the number of bogus files that need to be padded for each keyword. For a database DB, we use the same clustering method as [19]. For cluster $\mathscr{C} = \{w_1, \ldots, w_{|\mathscr{C}|}\}$, the keywords are sorted by their frequency in ascending order. We pad the keyword counts in a cluster to the same size according to the largest one, calculate the number of bogus files that need to be padded for each keyword, and put the result into set G. Subsequently, we randomly generate bogus files, select those that meet the conditions, and add them to PDB. The bogus file generation algorithm is as follows: we use the m dimension vector to represent bogus file, where m represents the size of the W. If we fill the keyword w_i, then set the i-th bit to 1; otherwise, we set it to 0. Then, we randomly generate a bogus file and judge whether it meets the requirements. Select the τ-bit keyword to fill in the bogus file, where τ is selected from the maximum file size and the maximum file size that the dataset can hold. After generating the bogus file, we use the local outlier factor (LOF) algorithm to detect the outliers of the bogus file [19]. As shown in [19], we should choose a bogus file whose LOF value is approximately equal to 1 for padding. Therefore, we use this threshold to filter samples. If LOF(V)<1, we add V to the padding database PDB. Otherwise, we roll back the padding counts and reselect the bogus files for detection until all keywords are padded.

3.2. $\widetilde{PDB}$ ⟵ PDB − Update, $(w, index, k, PDB)$. Upon inputting the keyword w that needs to be updated, the index of the update operation, the number of bits of the real database in the index k, and the padding database PDB, it outputs the updated PDB and the adjusted index. The algorithm is described in Algorithm 2. For the update operation of the padded database, our scheme uses an N-bit bitmap index (Section 2.1) to represent the real files and bogus files, where the first k bits indicate whether the real file exists or not. If file f_i exists, then the i-th bit is set to 1; otherwise, it is set to

Table 1: Notation.

Parameter	Notation
λ	The security parameter
F	A secure PRF
n	The length of S
k	The number of files that the database can hold
W	Keyword space $W = \mathrm{U}_{i=1}^{k} w_i$
f	The i-th real file $(1 \le i \le k)$
V_i	The i-th bogus file $(1 \le i \le n-k)$
DB	A database $\mathrm{DB} = (f_i, w_i)_{i=1}^{k}$
PDB	A padding database $\mathrm{PDB} = \cup_{i=1}^{n-k} V_i$
EDB	Encrypted database (including DB and PDB)
ST	The current search token for a keyword w
STM	A map that stores ST and the times tm of update operations on keywords in the keyword space W
S	The bit string which used to indicate whether the real file and the bogus file exist
ES	The encrypted S
$\mathrm{Sum}_{\mathrm{ES}}$	Sum of the ES
sk	The onetime secret key
$\mathrm{Sum}_{\mathrm{sk}}$	Sum of sk

```
(1) PDB, G ⟵ emptymap
(2) for all cluster C:
(3) for 1 ≤ i ≤ |w_|C|| do
(4)     put(w_i, |DB(|w_|𝒞||)| − |DB(w_i)|)to G
(5) while G do
(7)     v ⟵ 0, ..., 0_m, τ ⟵ [l, u]
(8) select τ different keywords w_1, ..., w_τ randomly
(9)     for i = 1: τ do
(10)        if G.get(wi) > 0 then
(11)        V[j] = 1
(12)        G.update(w_i, G.get(w_i) − 1)
(13)        else if G.get(w_i) = 0 then
(14)            select a new w_i from W
(15)        else
(16)            V[j] = 0
(17)    LOF(V) //validation via LOF detection
(20)    if LOF(V) < 1 then
(21)        DB ⟵ PDB ∪ {V}//put bogus files to PDB
(22) else
(23) rollback
(24) return PDB
```

Algorithm 1: PDB-Gen(DB).

0. For a bogus file, if the file V_i is padded, the $k+i$ bit of the index is set to 1; otherwise, it is set to 0.

Subsequently, we convert the index into an array and initialize a counter to record the number of files that need to be added or deleted. If the operation is addition, the padding files $\mathrm{PDB}(V_w)$ corresponding to the keyword w in PDB need to delete some bogus files corresponding to w. Specifically, we try to change the corresponding position of V_i from 1 to 0. Prior to modifying this position, we perform outlier detection on the changed bogus files V_i. If $\mathrm{LOF}(V_i) < 1$, then we modify it and modify the index at the same time. Otherwise, we roll back to the state before the modification and continue to try to modify the next bogus file.

If the operation is deletion, we try to pad files corresponding to keyword w. First, we count the files that need to be deleted, find the bogus file V_i that is not padded with the keyword w in PDB, and modify them one by one. It is worth noting that we try to modify the file corresponding to the position of 0 to 1 and perform outlier detection. If $\mathrm{LOF}(V_i) < 1$, we modify the bogus file and index. Otherwise, we roll back to the state before the modification and modify the next file. Finally, the modified index is converted into a sequence and returned.

The corresponding index during the execution of the dynamic padding algorithm is shown in Figure 2 that is explained in plain text. Assuming that the maximum capacity

```
(1) UA ⟵ to Array (index)
(2) count ⟵ 0
(3) (1) op = add
(4)    for 1 ≤ i ≤ index.length do
(5)        if UA[i] = 1 then
(6)           count + +
(7)    for 1 ≤ i ≤ |PDB| and count > 0 do
(8)       if V_i[w] = 1 then
(9)          Vi[w] = 0//try to modify bogus file
(10)         if LOF(V_i) < 1 then
(11)            count − −
(12)            Add corresponding bit string to UA
(13)         else
(14)            V_i[w] = 1//roll back the modify
(15) (2) op = delete
(16)   for 1 ≤ i ≤ index.length do
(17)       if UA[i] = 1 then
(18)          count + +
(19)   for 1 ≤ i ≤ |PDB| and count > 0 do
(20)       if V_i[w] = 1 then
(21)          Vi[w] = 1//try to modify bogus file
(22)          if LOF(V_i) < 1 then
(23)             count − −
(24)             Add corresponding bit string to UA
(25)          else
(26)             V_i[w] = 0//roll back the modify
(27)   index ⟵ to Array (UA)
(28)   return index
```

Algorithm 2: PDB-Update (w; index; k; PDB).

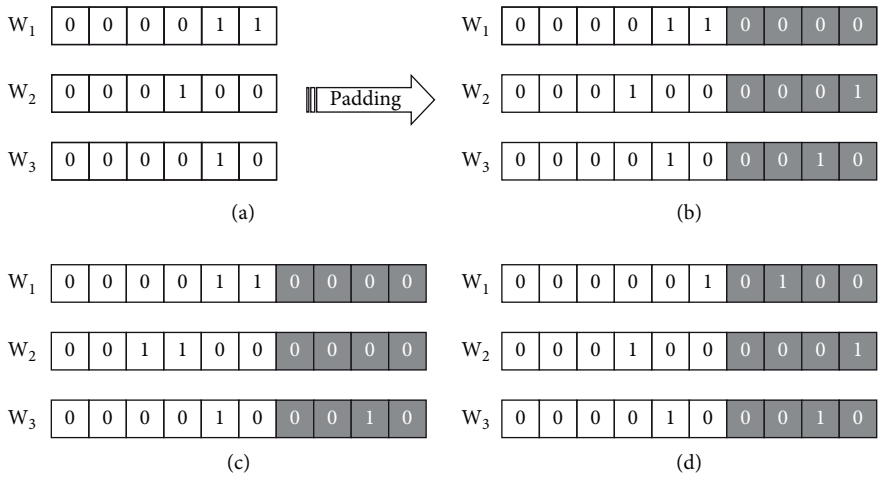

Figure 2: Corresponding index during the execution of the dynamic padding algorithm. (a) Bitmap index. (b) Pad. (c) Add (W_2, f_4). (d) Delete (W_1, f_2).

of the database is 10, the maximum capacity of the real database is 6, that is, $k = 6$, so that the maximum capacity of the PDB is 4. The initial real data are inserted in order (w_1, f_1), (w_1, f_2), (w_2, f_3), and (w_3, f_2). At this time, the database index is as shown in Figure 2(a). After the PDB-Gen(DB) algorithm is executed, the index is shown in Figure 2(b), where (w_2, V_1) and (w_3, V_2) are padded. If we want to insert (w_2, f_4), we need to add the bit string $(0010000000)_2$ according to the adding rules of the bitmap index. We search the PDB for the bogus file PDB (V_{w_2}) corresponding to w_2, and the result is V_1. Then, deleting V_1, we need to subtract the bit string $(0000000001)_2$. According to the deletion rule of the bitmap index, this is equivalent to adding the bit string $(1111111111)_2$ to obtain the final w_2 new index $(0011000000)_2$, as shown in Figure 2(c). If we perform a delete operation, delete (w_1, f_2), we need to subtract the bit string $(0000100000)_2$ from the index of w_1. According to the deletion

rule of the bitmap index, this is equivalent to adding the bit string $(1111100000)_2$ to retrieve all of the files corresponding to the position value of 0 and attempting to modify the bogus files one by one to change the corresponding position 0 to 1. After trying to modify, if outlier detection is performed on V_i and $\mathrm{LOF}(V_i) < 1$, then proceed Modify; otherwise, try to modify the next one. In this example, a bogus file V_3 is added to w_1 and the bit string $(0000000100)_2$ needs to be added to obtain the final new index $(0000010100)_2$, as shown in Figure 2(d). The above calculations require modulo 10.

4. PDB-DSSE Scheme

In this section, we construct the PDB-DSSE scheme using the dynamic database padding algorithm proposed in Section 3, where the real files and bogus files are represented by bitmap indexes and symmetric encryption with additive homomorphism [20] is used to encrypt the indexes. We use the framework of [13] combined with our proposed dynamic padding algorithm to prove the feasibility of the dynamic padding algorithm. It specifically consists of the algorithms Setup, Update, and Search.

4.1. $(EDB, \sigma) \longleftarrow Setup(1^{\lambda}, DB)$. The detailed design of this algorithm is given in Algorithm 3. Input the security parameter λ. The security key K is randomly selected. Generate a security certificate N, where $N = 2^n$. In addition, set the maximum file number k of the database DB, initialize an empty map STM, call the padding database generation algorithm PDB-Gen to generate the padding database PDB, and then initialize a map EDB, where EDB stores encrypted databases (including database DB and padding database PDB) and STM stores ST and the times tm of update operations on keywords in the keyword space W. Finally, send EDB to the server, and the client stores the state σ =(N, k, K, STM) secretly.

4.2. $(\sigma', EDB') \longleftarrow Update(w, S, \sigma; EDB)$. This protocol requires interactive execution between the client and the server. Algorithm 4 gives a detailed algorithm design. First, the client enters the update keyword w, status σ, and bit string S. Then, obtain search token ST and the update times tm on keywords in the keyword space W according to STM and use F_k to obtain the key. If it is the first update, initialize the search token ST and the times of updates tm. The hash function H_1 generates the update token UT, the hash function H_2 is used to mask the previous ST, and the hash function H_3 is used to generate a one-time key. According to the bit string that needs to be updated, the algorithm PDB-Update (w, S, σ, PDB) is called and the bit string S' used to update the index is returned. Then, S' is encrypted with a simple SE algorithm with additive homomorphism to obtain the ES. Then, the client sends UT, ES, and M_{ST} to the server. Finally, the server updates the ES to ES$'$.

4.3. $(\sigma', EDB') \longleftarrow Search(s, \sigma; EDB)$. This protocol requires interaction between the client and the server. Algorithm 5 gives a detailed algorithm design. First, the client enters the keyword w that it wants to search, generates a search token ST and a key k_w, and sends it to the server. The server uses the hash function H_1 to obtain the corresponding update token and accesses the database to obtain the encrypted bit string ES. Then, the server uses simple homomorphic addition to add them to the final result $\mathrm{Sum}_{\mathrm{ES}}$ and sends it to the client. The client uses the hash function H_3 to obtain the key, calculates the key sum $\mathrm{Sum}_{\mathrm{sk}}$, and uses the key to decrypt $\mathrm{Sum}_{\mathrm{ES}}$ to obtain the bit string S'. After removing the $n - k$ stuffing bits, it outputs the final result bit string S.

5. Security Analysis

In this section, we give the security analysis of our schemes.

PDB-DSSE is forward and backward privacy. Because the hash function inversion operation is almost impossible to complete, the adversary cannot associate the future update with the update before the search. This guarantees forward privacy. For backward privacy, due to the bitmap index, the number of bit strings returned is consistent and the adversary cannot obtain the number of documents currently matching the keyword. Besides, since addition and deletion are done through an algorithm, our solution will not leak the update type. This guarantees backward privacy. In summary, PDB-DSSE is forward and backward privacy.

Definition 4 (adaptive forward and Type-I$^-$ backward privacy of PDB-DSSE). Let F be a secure PRF, $\prod$ = (Setup, Enc, Dec, Add) be a perfectly secure symmetric encryption with homomorphism addition, and RO_1, RO_2, and RO_3 be random oracles. We define $\mathscr{L}_{\mathrm{PDB-DSSE}} = (\mathscr{L}^{\mathrm{srch}}_{\mathrm{PDB-DSSE}}, \mathscr{L}^{\mathrm{updt}}_{\mathrm{PDB-DSSE}})$, where $\mathscr{L}^{\mathrm{srch}}_{\mathrm{PDB-DSSE}} = (\mathrm{sp}(w), \mathrm{Time\,PDB}(w))$ and $\mathscr{L}^{\mathrm{updt}}_{\mathrm{PDB-DSSE}} = (\mathrm{op}, w, S)$. PDB-DSSE is $\mathscr{L}_{\mathrm{PDB-DSSE}}$-adaptive forward and Type-I$^-$ backward privacy.

Proof. Similar to [9], we formulate a sequence of games between REAL and IDEAL. Despite the subtle differences between two consecutive games, they are indistinguishable, leading to the conclusion that REAL and IDEAL are indistinguishable. Finally, we use the leakage function defined in Theorem 1 as the input of the simulator to simulate IDEAL:

Game G_0: G_0 is exactly the same as $\mathrm{REAL}^{\mathrm{DSSE}}_{\mathscr{A}}(\lambda)$ defined in Section 2.3.

Game G_1: G_1 is identical to G_0 except we do not use a pseudo-random function F to generate the key but randomly select the key with the uniform probability. If w has never been searched, the key is generated and stored in the table key; otherwise, input the keyword w, and return the corresponding key in the table key. G_1 is indistinguishable from G_0 due to the security of the PRF.

Game G_2: G_2 is the same as G_1, except that we use a random oracle RO_1 instead of hash function H_1. During the update process, the function H_1 is used to generate the update token. Instead of using H_1, we

(1) $K \xleftarrow{\$} \{0,1\}^{\lambda}$, $n \longleftarrow \text{Setup}(1^{\lambda})$
(2) STM $\longleftarrow$ emptymap
(3) PDB $\longleftarrow$ PDB − Gen(DB)
(4) EDB $\longleftarrow$ Encrypt(DB, PDB)
(5) set the max size of DB to k
(6) return (EDB, $\sigma = (n, k, K, \text{STM})$)

Algorithm 3: PDB-DSSE setup (1^{λ}, DB).

Client:
(1) $K_w \parallel K'_w \longleftarrow F_k(w)$
(2) $(\text{ST}_{\text{tm}}, \text{tm}) \longleftarrow \text{STM}[w]$
(3) if $(\text{ST}_{\text{tm}}, \text{tm}) = \perp$ then
(4) $\text{tm} \longleftarrow -1$, $\text{ST}_{\text{tm}} \longleftarrow (0,1)^{\lambda}$
(5) $\text{ST}_{\text{tm+1}} \longleftarrow (0,1)^{\lambda}$
(6) $\text{STM}[w] \longleftarrow (\text{ST}_{\text{tm+1}}, \text{tm} + 1)$
(7) $\text{UT}_{\text{tm+1}} \longleftarrow H_1(K_w, \text{ST}_{\text{tm+1}})$
(8) $M_{\text{ST}_{\text{tm}}} \longleftarrow H_2((K_w, \text{ST}_{\text{tm}}) \oplus M_{\text{ST}_{\text{tm-1}}})$
(9) $\text{sk}_{\text{tm+1}} \longleftarrow H_3(K'_w, \text{tm} + 1)$
(10) $S' \longleftarrow \text{PDB} - \text{Update}(w, S, k, \text{PDB})$
(11) $\text{ES}_{\text{tm+1}} \longleftarrow \text{Enc}(\text{sk}_{\text{tm+1}}, S', n)$
(12) send $(\text{UT}_{\text{tm+1}}, (\text{ES}_{\text{tm+1}}, M_{\text{ST}_{\text{tm}}}))$ to server
Server:
(13) $\text{EDB}[\text{UT}_{\text{tm+1}}] \longleftarrow (\text{ES}, M_{\text{ST}_{\text{tm}}})$

Algorithm 4: (σ', EDB′)⟵Update(w, S, σ; EDB).

Client:
(1) $K_w \parallel K'_w \longleftarrow F_k(w)$, $(\text{ST}_{\text{tm}}, \text{tm}) \longleftarrow \text{STM}[w]$
(2) if $(\text{ST}_{\text{tm}}, \text{tm}) = \perp$ then
(3) return ϕ
(4) send $(K_w, \text{ST}_{\text{tm}}, \text{tm})$ to server
Server:
(5) $\text{Sum}_{\text{ES}} \longleftarrow 0$
(6) for $i = \text{tm}$ to 0 do
(7) $\text{UT}_i \longleftarrow H_1(K_w, \text{ST}_i)$
(8) $(\text{ES}, M_{\text{ST}_{i-1}}) \longleftarrow \text{EDB}[\text{UT}_i]$
(9) $\text{Sum}_{\text{ES}} \longleftarrow \text{Add}(\text{Sum}_{\text{ES}}, \text{ES}_i, n)$
(10) if $M_{\text{ST}_{i-1}} = \perp$ then
(11) Break
(12) $\text{ST}_{i-1} \longleftarrow H_2((K_w, \text{ST}_i) \oplus M_{\text{ST}_{i-1}})$
(13) $\text{EDB}[\text{UT}_{\text{tm}}] \longleftarrow (\text{Sum}_{\text{ES}}, \perp)$
(14) send Sum_{ES} to client
Client:
(15) $\text{Sum}_{\text{sk}} \longleftarrow 0$
(16) for $i = \text{tm}$ to 0 do
(17) $\text{sk}_i \longleftarrow H_3(K'_w, i)$
(18) $\text{Sum}_{\text{sk}} \longleftarrow \text{Sum}_{\text{sk}} + \text{sk}_i \bmod n$
(19) $S' \longleftarrow \text{Dec}(\text{Sum}_{\text{sk}}, \text{Sum}_{\text{ES}}, n)$
(20) $S \longleftarrow$ select first k bits of S'
(21) return S

Algorithm 5: (σ', EDB′)⟵Search(s, σ; EDB).

randomly select a string as the update token. In the search process, use the random oracle RO_1 to generate the update token. We also use the random oracle RO_2 and RO_3 similar to RO_1 instead of hash functions H_2 and H_3. Since a search token is chosen randomly, G_2 is indistinguishable from G_1.

Game G_3: G_3 is the same as G_2, except that we replace the number in each position of the bit string S with 0 and the length of bit string remains the same. Due to the perfect security of simple SE with homomorphic addition, G_3 is indistinguishable from G_2.

The simulator is the same as G_3, except that we use the search mode $\mathrm{sp}(w)$ instead of the keyword w to simulate the ideal world. During the update process, we chose a new random string for each update in the game G_3. During the search process, the simulator starts from ST_c and generates new random strings for the previous ST one by one. For the keyword w, $\mathcal{S}$ uses the first timestamp $\widehat{w} \longleftarrow \mathrm{minsp}(w)$. Then, it uses the random oracle RO_2 to calculate it with the ciphertext c. At the same time, $\mathcal{S}$ calculates S and ST_c through RO_3 and embeds all 0s in the remaining search tokens. Hence, G_3 and simulator are indistinguishable. In addition, what is described in simulator is essentially the game $\mathrm{IDEAL}_{\mathcal{A},\mathcal{S}}^{\mathrm{DSSE}}(\lambda)$ defined in Section 2.3. Hence, G_3 and $\mathrm{IDEAL}_{\mathcal{A},\mathcal{S}}^{\mathrm{DSSE}}(\lambda)$ are indistinguishable.

In summary, game $\mathrm{IDEAL}_{\mathcal{A},\mathcal{S}}^{\mathrm{DSSE}}(\lambda)$ is indistinguishable from G0, that is, $\mathrm{REAL}_{\mathcal{A}}^{\mathrm{DSSE}}(\lambda)$, which completes the proof. □

6. Experimental Analysis

In this section, we present the experimental analysis of our PDB-DSSE scheme, including the comparison with other research results in communication overhead, storage overhead, and the time complexity of query and update.

Table 2 presents the performance comparison between PDB-DSSE and existing research solutions, including security, interaction rounds, and client storage overhead. Here, N is the number of (w, f) in the database, d_w is the number of times the file containing the keyword w is deleted, $|W|$ is the number of all different keywords, and $|D|$ is the number of files stored in the database. Regarding security, as shown in Table 2, PDB-DSSE implements forward privacy and Type-I^- backward privacy. Regarding communication overhead, PDB-DSSE only requires one interaction, greatly reducing the communication overhead of the solution. Furthermore, the client storage overhead is $O(1)$ since the client only needs to store the secret key sk and the state σ. Therefore, it is clear that PDB-DSSE only requires one interaction and less client overhead to achieve stronger backward privacy.

Table 3 shows the comparison of the time complexity of the PDB-DSSE scheme with other schemes with stronger backward privacy, Moneta [7], Orion [9], and FB-DSSE [13], where N is the number of (w, f) in the database, f_w is the number of files containing the keyword w, u_w is the number of update operations performed on the keyword w, and t_{ED} is the time it takes for symmetric encryption to perform encryption and decryption operations. t_{AM} is the calculation time for modular addition.

6.1. Implementation. The simulation experiment of our scheme uses a machine with an AMD Ryzen 7 4800U with a Radeon Graphics 1.8 GHz processor configured with a Windows 10 (64-bit) system and 16 GB RAM. We use the Java language programming to implement our scheme. During the experiments, the number of bits of the bitmap index was adjusted many times to perform experiments to simulate the performance of this scheme under the situation of different maximum file numbers of the database. In our experiments, the update time and search time of the scheme PDB-DSSE were tested when the number of bitmap index bits was 10^2 to 10^9. We compare our scheme with the Orion scheme [9] that uses the ORAM structure and a higher level of backward security and the FB-DSSE scheme [13] that uses bitmap indexing efficiency. The comparison results are as follows.

Figure 3 shows the search time comparison of PDB-DSSE scheme, Orion scheme [9], and FB-DSSE scheme [13] under different bit lengths that indicate the maximum number of files supported in the scheme. In the Orion scheme, this is represented by the size of |DB|. The tested bit length ranges from 10^2 to 10^9 (Orion scheme is $|\mathrm{DB}| = 10^2$ to 10^5). Figure 3 shows that the search time of these three schemes increases with increasing bit length (|DB|). Since the Orion uses the ORAM structure, its search time complexity is $O(n_w \log^2 N) \cdot t_{\mathrm{SKE}}$. When $|\mathrm{DB}| = 10^5$, its search time took 96 888.2 ms. We can observe from the experimental results that the search efficiency of the ORAM structure is extremely low for large datasets. Through test comparison, it can be found that PDB-DSSE maintains the same search efficiency as the FB-DSSE scheme to a certain extent. In summary, PDB-DSSE has high search efficiency while realizing forward privacy and stronger backward privacy.

Figure 4 shows the comparison of the update time of the PDB-DSSE with Orion [7] and FB-DSSE [13] under different bit lengths (or |DB|), where the Orion scheme includes the insertion and deletion times. Since Orion needs to modify the ORAM structure when updating, the insertion time of $|\mathrm{DB}| = 10^2$ requires 51.823 ms and the deletion time requires 29.454 ms. In particular, when $|\mathrm{DB}| = 10^5$, it takes 3.472 days to build a database with a magnitude of 10^5 and it takes at least 35 days to build a database with a magnitude of 10^6. Therefore, it is extremely difficult to test the update time when |DB| is greater than 10^6; this problem does not exist in PDB-DSSE. We tested the update time of the PDB-DSSE and FB-DSSE schemes with a bit length of 10^2 to 10^9. We find that the update time of FB-DSSE is stable at 0.2–0.3 ms and the update time of FB-DSSE is approximately 0.4 ms when the bit length is less than 10^6. For bit lengths greater than 10^6, the update time of PDB-DSSE and FB-DSSE increases with increasing bit length. This is because when the bit length is less than 10^6, the influence of homomorphic addition and modular arithmetic is not significant. The update time complexity of PDB-DSSE is longer than that of the FB-

Table 2: Comparison of research results.

Scheme	FP	BP	Roundtrips	Client storage
FIDES [7]	√	Type-II	2	$O(\|W\|\log\|D\|)$
$\text{DIANA}_{\text{del}}$[6]	√	Type-III	2	$O(\|W\|\log\|D\|)$
Janus [6]	√	Type-III	1	$O(\|W\|\log\|D\|)$
Janus++ [9]	√	Type-III	1	$O(\|W\|\log\|D\|)$
MONETA [6]	√	Type-I	3	$O(1)$
HORUS [9]	√	Type-III	$O(\log d_w)$	$O(\|W\|\log\|D\|)$
ORION [9]	√	Type-I	$O(\log N)$	$O(1)$
FB-DSSE [13]	√	Type-I^-	1	$O(\|W\|\log\|D\|)$
PDB-DSSE	√	Type-I^-	1	$O(\|W\|\log\|D\|)$

Table 3: Comparison of search and update time complexity.

Scheme	Search	Update
MONETA [7]	$\widehat{O}(u_w \log N + \log^3 N) \cdot t_{\text{ED}}$	$\widehat{O}(\log^2 N) \cdot t_{\text{ED}}$
ORION [9]	$O(f_w \log^2 N) \cdot t_{\text{ED}}$	$O(\log^2 N) \cdot t_{\text{ED}}$
FB-DSSE [13]	$O(u_w) \cdot t_{\text{AM}}$	$O(1) \cdot t_{\text{AM}}$
PDB-DSSE	$O(u_w) \cdot t_{\text{AM}}$	$O(f_w + t_{\text{AM}})$

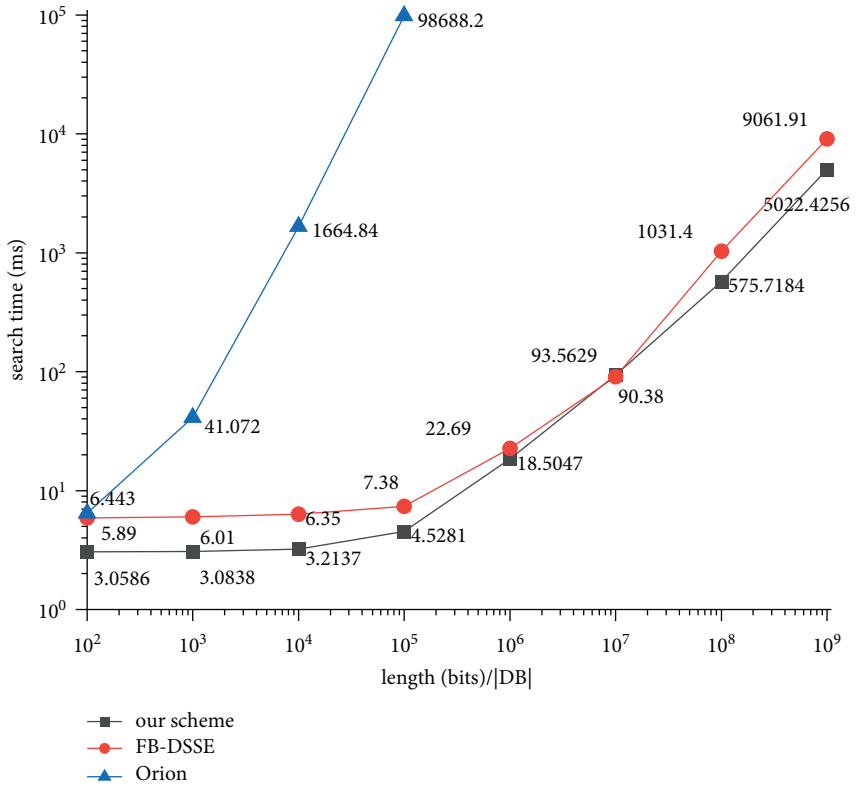

Figure 3: The comparison of search time.

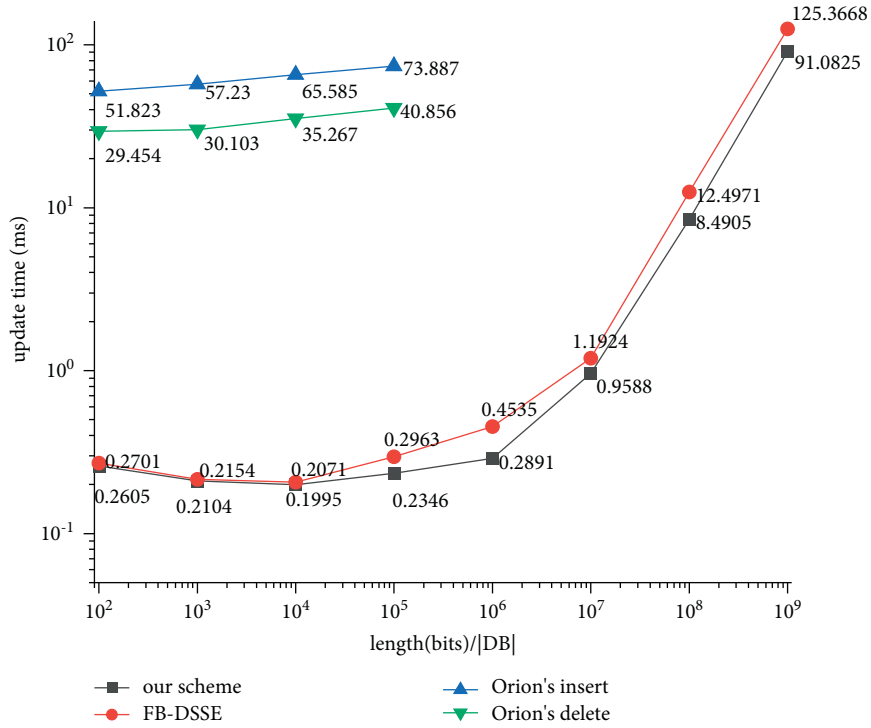

Figure 4: The comparison of update time.

DSSE scheme since it needs to update the PDB, but the update efficiency can still be maintained. Through comparison, it can be found that PDB-DSSE has high update efficiency.

7. Conclusion

In this paper, we construct a dynamic padding method that can be used for DSSE schemes and give an application scenario to realize a DSSE scheme PDB-DSSE with forward privacy and Type-I$^-$ backward privacy. First, we introduce outlier detection technology to design a dynamic database padding algorithm that can be used in DSSE schemes. We use the padded bogus file to confuse the real file to prevent leakage of information about the files that currently matches the search keyword. Furthermore, we design a new index structure based on the bitmap index that is suitable for the padding database that simplifies the process of modifying the index when updating. Finally, we propose an application scenario of the padding method and implement the $\mathscr{L}_{\text{PDB-DSSE}}$-adaptive forward and Type-I$^-$ backward privacy DSSE scheme. Simulation results and comparative experiments show that the dynamic padding scheme proposed in this paper is suitable for DSSE schemes. In addition, the PDB-DSSE scheme, which incorporates the dynamic padding scheme proposed in this paper, still maintains efficient search and update efficiency.

As the size of the database increases, bitmap indexing and padding algorithms face certain limitations. To support a larger number of files (such as billions), in future work, we will consider dividing the index into blocks, padding each block separately to improve retrieval and update efficiency and reduce computational overhead. In addition, we will consider the use of bitmap indexes to implement a search for conjunctive keyword queries, thereby improving the retrieval efficiency of existing solutions.

Data Availability

Previously reported Enron Email Datasets were used to support this study and are available at https://www.cs.cmu.edu/~./enron/.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

[1] DX Song, D Wagner, and A Perrig, "Practical techniques for searches on encrypted data," in *Proceedings of the 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pp. 44–55, IEEE, Berkeley, CA, USA, May 2000.

[2] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," *Journal of Computer Security*, vol. 19, no. 5, pp. 895–934, 2011.

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

[4] Y Zhang, J Katz, and C Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in *Proceedings of the 25th {USENIX} Security*

Symposium ({USENIX}Security 16), pp. 707–720, USENIX Association, Austin, TX, USA, August 2016.

[5] E Stefanov, C Papamanthou, and E Shi, "Practical dynamic searchable encryption with small leakage," *NDSS*, vol. 71, pp. 72–75, 2014.

[6] R. Bost, "Σοφ οζ: forward secure searchable encryption," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1143–1154, ACM, Vienna Austria, October 2016.

[7] R Bost, B Minaud, and O Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1465–1482, ACM, Dallas, TX, USA, October 2017.

[8] SF Sun, X Yuan, JK Liu et al., "Practical backward-secure searchable encryption from symmetric puncturable encryption," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 763–780, ACM, Toronto, Canada, October 2018.

[9] J Ghareh Chamani, D Papadopoulos, C Papamanthou, and R Jalili, "New constructions for forward and backward private symmetric searchable encryption," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1038–1055, ACM, Toronto, Canada, October 2018.

[10] P Rizomiliotis and S Gritzalis, "Simple forward and backward private searchable symmetric encryption schemes with constant number of roundtrips," in *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pp. 141–152, ACM, London, UK, November 2019.

[11] A. Najafi, H. H. S. Javadi, and M. Bayat, "Verifiable ranked search over encrypted data with forward and backward privacy," *Future Generation Computer Systems*, vol. 101, pp. 410–419, 2019.

[12] L Sardar and S Ruj, "Fspvdsse: A forward secure publicly verifiable dynamic sse scheme," in *Proceedings of the International Conference on Provable Security*, pp. 355–371, Springer, Cairns, Australia, October, 2019.

[13] C Zuo, SF Sun, JK Liu, J Shao, and J Pieprzyk, "Dynamic searchable symmetric encryption with forward and stronger backward privacy," in *Proceedings of the European Symposium on Research in Computer Security*, pp. 283–303, Springer, Luxembourg, UK, September 2019.

[14] C. Zuo, S. 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. 99, p. 1, 2020.

[15] X Li, T Xiang, and P Wang, "Achieving forward unforgeability in keyword-field-free conjunctive search," *Journal of Network and Computer Applications*, vol. 166, Article ID 102755, 2020.

[16] I Demertzis, JG Chamani, D Papadopoulos, and C Papamanthou, "Dynamic searchable encryption with small client storage," in *Proceedings of the Network and Distributed System Security Symposium*, DBLP, San Diego, CA, USA, February 2020.

[17] S. Chatterjee, S. K. Parshuram Puria, and A. Shah, "Efficient backward private searchable encryption," *Journal of Computer Security*, vol. 28, no. 2, pp. 229–267, 2020.

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

[19] L Xu, X Yuan, C Wang, Q Wang, and C Xu, "Hardening database padding for searchable encryption," in *Proceedings of the IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pp. 2503–2511, IEEE, Paris, France, April 2019.

[20] C Castelluccia, E Mykletun, and G Tsudik, "Efficient aggregation of encrypted data in wireless sensor networks," in *Proceedings of the Second Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services*, pp. 109–117, IEEE, San Diego, CA, USA, July 2005.