Verifiable and Dynamic Multi-Keyword Search Over Encrypted Cloud Data Using Bitmap

Feng Li, Jianfeng Ma, *Member, IEEE*, Yinbin Miao, *Member, IEEE*, Qi Jiang, Ximeng Liu, *Member, IEEE*, and Kim-Kwang Raymond Choo, *Senior Member, IEEE*

Abstract—Searchable Symmetric Encryption (SSE), which enables users to search over encrypted data without decryption, has gained increasing attention from both academic and industrial fields. However, existing SSE schemes either have low search efficiency or cannot support multi-keyword search, dynamic updates, and result verification simultaneously. To solve these problems, we propose a Verifiable and Dynamic Multi-keyword Search (VDMS) scheme over encrypted data by using the bitmap and RSA accumulator, which provides multi-keyword search over encrypted data in an efficient, verifiable and updated way. The bitmap is used as a data structure to build the indexes, which improves the search efficiency and reduces the storage space of the indexes. The RSA accumulator and bitmap are combined to verify the correctness of results. Formal security analysis proves that our VDMS is adaptively secure against Chosen-Keyword Attacks (CKA), and empirical experiments using a real-world dataset demonstrate that our VDMS is efficient and feasible in practical applications.

Index Terms—Searchable symmetric encryption, multi-keyword search, bitmap, chosen-keyword attack

1 Introduction

As a computing paradigm, cloud computing has attracted more and more attention from individuals, institutions, and organizations due to its scalability and flexibility, which motivates them to store their data on cloud servers. Cloud users can remotely store and use local data on cloud servers and obtain high-quality on-demand services, which incurs less storage and computation burden. However, security and privacy concerns still impede the deployment of cloud storage systems. Data encryption can eliminate such concerns to some extent but make the search over encrypted data a challenging issue. Thus, the Searchable Symmetric Encryption (SSE), which allows users to retrieve encrypted files of interest, has been extensively explored.

When widely deployed in practical applications, SSE schemes should satisfy the following requirements. (1) *Efficiency*. The search time should be proportional to the number of keywords rather than the total number of files in the database. (2) *Verifiability*. The cloud server may be malicious, which may execute a fraction of search operations or return some forged or tampered results. In order to return the results that the user is interested in, the correctness and completeness of returned results should be guaranteed. (3) *Dynamic*. The cloud users often add or delete data stored in the cloud. Thus, supporting dynamic updates is an indispensable component of an SSE scheme.

Among existing SSE schemes [1], [2], [3], [4], [5], [6], linked lists and vectors are the two most commonly used index structures, but still have some limitations. The linked list-based SSE schemes [5], [7] require cloud servers to traverse all lists to find the required results, and the vector-based SSE schemes [6], [8] also need to traverse the entire indexes, which both incur high search overhead. Besides, some SSE schemes [4], [5] that support dynamic operations are also based on linked lists or vectors to construct indexes but do not improve the search efficiency. Hash function-based result mechanisms such as Merkle hash tree or hash chain will incur lots of extra overhead and cannot support dynamic updates efficiently. The homomorphic MAC [9] supporting result verification still incurs high storage overhead due to tag generations. To solve these problems, in this paper, we propose a Verifiable and Dynamic Multi-keyword Search (VDMS) scheme, which allows users to issue multi-keyword searches efficiently and update on encrypted data in a verifiable way. VDMS achieves the optimal search time, which is proportional to the number of queried keywords rather than the number of entire files/documents. Due to the inherent properties of the bitmap, the cloud server can calculate the indexes corresponding to the keywords without traversing them. Besides, the VDMS scheme also supports the dynamic updates and verification of returned results.

- *Feng Li and Yinbin Miao are with the School of Cyber Engineering, Xidian University, Xi'an, Shaanxi 710071, China, and with the Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin 541004, China, and also with the Department of Computer Science, City University of Hong Kong, Hong Kong 999077, China. E-mail: fli1996@stu.xidian.edu.cn, ybmiao@xidian.edu.cn.*
- *Jianfeng Ma is with the School of Cyber Engineering, Shaanxi Key Laboratory of Network and System Security, Xidian University, Xi'an, Shaanxi 710071, China. E-mail: jfma@mail.xidian.edu.cn.*
- *Qi Jiang is with the School of Cyber Engineering, Xidian University, Xi'an, Shaanxi 710071, China, and also with the Network Communication Research Centre, Peng Cheng Laboratory, Shenzhen, Guangdong 518055, China. E-mail: jiangqixdu@gmail.com.*
- *Ximeng Liu is with the Key Laboratory of Information Security of Network Systems, College of Mathematics and Computer Science, Fuzhou University, Fuzhou, Fujian 350108, China. E-mail: snbnix@gmail.com.*
- *Kim-Kwang Raymond Choo is with the Department of Information Systems and Cyber Security, The University of Texas at San Antonio, San Antonio, TX 78249 USA. E-mail: raymond.choo@fulbrightmail.org.*

Manuscript received 25 Jan. 2021; revised 18 May 2021; accepted 25 June 2021. Date of publication 1 July 2021; date of current version 8 Mar. 2023.
(Corresponding author: Yinbin Miao.)
Recommended for acceptance by C. Rong.
Digital Object Identifier no. 10.1109/TCC.2021.3093304

1.1 Related Work

Many searchable symmetric encryption schemes [1], [10], [11], [12], [13], [14], [15] have been proposed in the past few decades which focus on enriching the functions [1], [7], [11], [16], improving the efficiency [17], [18], [19], [20], and enhancing the security [21], [22], [23].

The first SSE scheme [24] allows users to retrieve ciphertexts by the sequential scan and is indistinguishable against chosen-plaintext attacks. However, this scheme is impractical and inefficient as its search time is proportional to the size of the documents. Curtmola *et al.* [5] proposed another SSE construction based on the inverted index, which achieves the sublinear search time, and defines the notions of non-adaptively secure and adaptively secure, but this scheme does not support dynamic operations. To solve this problem, Kamara *et al.* [7] proposed a dynamic SSE scheme based on [5] and proved that the scheme was adaptively secure against chosen-keyword attacks (CKA2-secure). This scheme can dynamically add or delete files on the database through additional data structures such as deletion array and deletion table and achieves the optimal search time.

SSE scheme with the multi-keyword search is critically crucial for improving search accuracy. There are many works of SSE schemes [6], [10], [25], [26] that support the multi-keyword search. Cao *et al.* [25] proposed an SSE scheme based on the similarity measure of 'coordinate matching' to support ranked multi-keyword search, but its search time is related to the number of documents. Fu *et al.* [27] explored index tree to reduce the search time complexity to $\mathcal{O}(r \log m)$, where r is the number of documents containing queried keywords, and m is the total number of documents.

In order to further improve search efficiency and reduce the storage space of indexes, Hwang *et al.* [28] utilized the bitmap as the index structure of SSE, but failed to support multi-keyword search and verification of results. [29] also proposed an SSE scheme on encrypted bitmap indexes to support multi-keyword search, but requires two rounds of interactions with the cloud server and asks users to choose the keyword with the least frequency in the query. Zuo *et al.* [30] proposed a more secure SSE scheme based on the bitmap index to support dynamic operations with forward and backward security, but this scheme cannot verify the correctness of the results. For verification, [9] proposed a verifiable SSE scheme based on the bitmap index in the presence of a semi-honest-but-curious or malicious cloud server. This scheme supports dynamic operations but does not achieve the multi-keyword search.

The verifiability of SSE scheme is particularly important in environments where cloud servers are assumed to be semi-honest-but-curious or malicious. Pang *et al.* [31] introduced a novel authentication mechanism that utilizes the threshold-based algorithm to accomplish text query processing. This scheme combines the Merkel hash tree and hash chain to verify the correctness of returned results, which is only suitable for plaintext search and does not support dynamic operations. Then, Kurosawa *et al.* [32] extended the above SSE scheme to achieve result verification in the dynamic setting. Their scheme exploits the RSA accumulator [33] to generate verification information for verifying the correctness of results, but its verification time on the user side increases linearly with the total number of documents. The dynamically verifiable SSE scheme [3] is an extension of [7], which uses the RSA accumulator to achieve verification through verifiable matrices and evidence stored locally. However, this method just supports a single keyword search, which cannot be widely deployed in practical applications.

TABLE 1
A Comparative Summary Among Different Schemes

Schemes	$\mathbb{F}_1$	$\mathbb{F}_2$	$\mathbb{F}_3$	$\mathbb{F}_4$	$\mathbb{F}_5$
BSSE [28]	✓	×	×	✓	$\mathcal{O}(\lvert q\rvert)$
BXT [29]	✓	✓	×	×	$\lvert q\rvert \cdot \lvert \omega_1\rvert$
VDMS	✓	✓	✓	✓	$\mathcal{O}(\lvert q\rvert)$

— $|q|$: *the number of keywords contained in* q; ω_1: *the least frequent keyword in the* q.
— $\mathbb{F}_1$: *Single-keyword search;* $\mathbb{F}_2$: *Multi-keyword search;* $\mathbb{F}_3$: *Verification;* $\mathbb{F}_4$: *Dynamic operations;* $\mathbb{F}_5$: *Search time;*

1.2 Our Contributions

In this paper, we design a verifiable and dynamic SSE scheme by utilizing the bitmap and RSA accumulator, which achieves multi-keyword search, results verification, and dynamic updates. We also present a comparative summary of VDMS and the existing schemes based on the bitmap in Table 1. Specifically, the main contributions of our work are summarized as follows:

- Our VDMS utilizes the bitmap technique to achieve dynamic and multi-keyword search, which can reduce the storage space and improve the search efficiency with sublinear search time.
- Our VDMS utilizes the RSA accumulator and bitmap techniques for verifying the correctness of the indexes and results, respectively. Specifically, the RSA accumulator can be used to verify whether the indexes involved in the calculation have been modified by a malicious adversary. Furthermore, the bitmap can be employed to verify whether the results are what users exactly want.
- We formally prove that VDMS is adaptively secure against CKA, and implement empirical experiments to demonstrate the efficiency and feasibility of VDMS in actual applications.

1.3 Organization of This Paper

The remainder of this paper is organized as follows: In Section 2, we introduce the bitmap and RSA accumulator used in VDMS. In Section 3, we describe the system model, the threat model of VDMS, and associated security definitions. Section 4 presents the concrete construction of VDMS. We theoretically analyze the security of VDMS in Section 5. In Section 6, we evaluate the performance of VDMS. Finally, we conclude this paper in Section 7.

2 Preliminaries

2.1 Bitmap

In this paper, we use the bitmap [34] as an index structure to represent file identities, which is also used to represent keyword identities for result verification. Bitmap uses a bit to

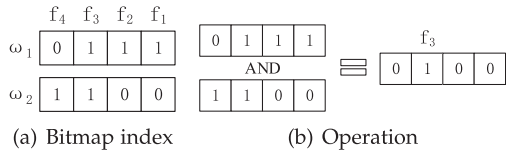


Fig. 1. An example of bitmap index.

store information, so it is a storage-saving data structure and is widely used in the field of data retrieval. Specifically, the bitmap is a bit array of length ℓ, where ℓ is the total number of elements that bitmap can store. If $\mathrm{f_i}$ exists, set the ith position of bit array to 1, otherwise to 0. For example, there exists four files $\{\mathrm{f_1}, \mathrm{f_2}, \mathrm{f_3}, \mathrm{f_4}\}$ and two keywords $\{\omega_1, \omega_2\}$ in Fig. 1, where ω_1 is contained in $\{\mathrm{f_1}, \mathrm{f_2}, \mathrm{f_3}\}$ and ω_2 is contained in $\{\mathrm{f_3}, \mathrm{f_4}\}$. Thus, the two bitmaps are represented as 0111 and 1100, respectively. If we want to retrieve files that contain both keywords ω_1 and ω_2, we need to do AND operation on the two bitmaps, i.e., $0111 \wedge 1100 = 0100$. The bit string 0100 indicates that the file $\mathrm{f_3}$ contains both keywords ω_1 and ω_2.

2.2 RSA Accumulator

The RSA accumulator [35] supports the verification of element membership, similar to the Merkle tree, which can determine whether the element exists. In our scheme, it is used to verify the completeness of the query results. It enables one to accumulate a set of values to a constant-size value (namely accumulator), and to verify whether the value has been accumulated or not with witness efficiently. In addition, the RSA accumulator has an attractive feature, which allows one to dynamically add and delete inputs with constant computational cost. This feature can satisfy our requirements and improve efficiency. In contrast, the Merkle tree cannot flexibly support the dynamic update of elements, and the size of the tree is proportional to the number of elements. Next, we introduce the RSA accumulator.

Let $\lambda \in \mathbb{N}$ be a security parameter, $\ell = \lfloor \lambda/2 \rfloor - 2$, $\mathcal{X} = \{x_1, \ldots, x_n\}$ denotes a prime number set, where $x_i \in \{0,1\}^{\ell}$, and x_i is a prime number. We take $x_i \in \mathcal{X}$ as the input of RSA accumulator[1]. Randomly choose a number $\mathbf{N} \in \{0,1\}^{\lambda}$ with satisfying $\mathbf{N} = pd$, where $p = 2p' + 1, d = 2d' + 1$, and all of them are prime numbers. Therefore, $\varphi(\mathbf{N}) = (p-1)(d-1)$. We define function f as $f(g, x) = g^x \bmod \mathbf{N}$, and the input domain $\mathcal{G}_f$ for f is defined as $\mathcal{G}_f = \{g \in QR_{\mathbf{N}} : g \neq 1\}$, where $QR_{\mathbf{N}}$ is the group of quadratic residues modulo $\mathbf{N}$. For a set of elements $\mathcal{X}$, the RSA accumulator works as follows:

- Randomly choose a number $g \in \mathcal{G}_f$, and for each $x_i \in \mathcal{X}$, the accumulator $Acc(\mathcal{X})$ is denoted as $Acc(\mathcal{X}) = g^{\prod_{i=1}^{n} x_i} \bmod \mathbf{N}$.
- For any subset $\mathcal{X}' \subseteq \mathcal{X}$, we calculate the witness $\chi = g^{\prod_{x_i \in \mathcal{X} - \mathcal{X}'} x_i} \bmod \mathbf{N}$ to check the membership. If and only if $Acc(\mathcal{X}) = \chi^{\prod_{x_i \in \mathcal{X}'} x_i} \bmod \mathbf{N}$ holds, the elements in $\mathcal{X}'$ are the member of $\mathcal{X}$.
- Assume that $\widehat{x}$ is added to the accumulator $Acc(\mathcal{X})$, the new accumulator is computed as $\overline{Acc(\mathcal{X})} = Acc(\mathcal{X})^{\widehat{x}} \bmod \mathbf{N}$. Similarly, if one wants to delete the value $\widehat{x}$ from the accumulator $Acc(\mathcal{X})$, $\overline{Acc(\mathcal{X})} = Acc(\mathcal{X})^{\widehat{x}^{-1} \bmod \varphi(\mathbf{N})} \bmod \mathbf{N}$. It is worth noting that we can add or delete a set of numbers by setting $\widehat{x}$ as the product of all numbers in addition (or deletion) set.

1. If the input domain is an arbitrary length string set, the elements in the set should be mapped to prime numbers before work.

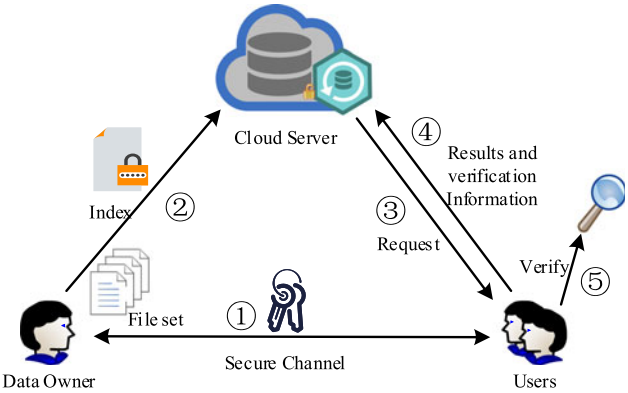


Fig. 2. System model.

3 Problem Formulation

3.1 System Model

In the system model of VDMS, three entities are involved: the data owner, the cloud server, and the data user, as depicted in Fig. 2. The role of each entity is as follows:

- *The data owner*. The data owner builds the indexes based on the files, generates and shares secret keys with the data user, encrypts the indexes and files, and outsources them to the cloud server.
- *The cloud server*. The cloud server stores and manages the received ciphertexts, processes the received request and returns the corresponding results and verification information.
- *The data user*. Each user with secret keys can send a request to the cloud server and decrypt the results returned by the cloud server. Besides, the data user able to verify the returned results with verification information.

The data owner generates the public keys of the RSA accumulator and the secret keys composed of the private keys of the RSA accumulator and the symmetric keys of the scheme, then sends the secret keys to the data user through a secure channel (Step ①). It is worth noting that although the data owner generates the public keys and the secret keys, our scheme is still based on symmetric encryption, since the data owner publishes the public keys and shares the secret keys with the data user. In other words, the data owner and the data user have the same knowledge of the secret keys. To protect the privacy of the files, the data owner first builds the file indexes, and then encrypts the files and indexes and outsources them to the cloud server (Step ②). The data user issues a request to the cloud server to retrieve the files of interest according to specified keywords (Step ③). Upon receiving the request, the cloud server retrieves the matching results on the encrypted data, and then sends the results and verification information to the data user (Step ④). After receiving the results and verification information, the data user can verify the correctness of the returned results, and discard the results if the verification fails (Step ⑤). Note that, in the realistic

system, the data owner can also access and operate the data. Throughout this paper, we do not distinguish between the data owner and the data user, collectively referred to as the client or user.

In the rest of the paper, we will focus our discussion on data retrieval, updating, and verification, instead of describing the encryption and decryption of files in detail.[2] In addition, we assume that the identity associated with each file is independent of its content, thereby allowing the file identity to be exposed to the cloud server.

3.2 Threat Model

Like most verifiable SSE schemes, we assume that the cloud server is malicious [36], which may execute a fraction of search operations or forge some results due to various interest incentives such as saving bandwidth and computation resources. The data owner and the data user are trusted and cannot collude with the cloud server.

To achieve a balance between efficiency and security, adversaries have no access to any information other than access pattern and search pattern. Access pattern refers to the ability to determine whether the returned results are overlapped, and the search pattern refers to whether the queries submitted are the same. [37] can hide the access pattern, which will reduce the efficiency of the scheme at the cost of increasing the interaction and bandwidth between the data user and the cloud server. Therefore, the main goal of VDMS is that the cloud server cannot learn any information about the files or keywords during the search, except for the access pattern and search pattern.

3.3 Algorithm Definitions

VDMS consists of eleven algorithms $\Gamma = (\textbf{KeyGen}, \textbf{BuildIndex}, \textbf{SearchToken}, \textbf{Search}, \textbf{Verify}, \textbf{AddToken}, \textbf{Add}, \textbf{DelToken}, \textbf{Del}, \textbf{Enc}, \textbf{Dec})$, which are as follows:

- $(PK, SK) \leftarrow \textbf{KeyGen}(1^\lambda)$: Given a security parameter $\lambda \in \mathbb{N}$, the algorithm outputs the public key PK and the secret key SK.
- $(\mathcal{I}, Acc(\mathbf{W})) \leftarrow \textbf{BuildIndex}(\mathbf{F}, \mathbf{W}, PK, SK)$: Given a set of files $\mathbf{F}$, a keyword set $\mathbf{W}$, a public key PK and a secret key SK, the algorithm outputs encrypted indexes $\mathcal{I}$ and accumulators $Acc(\mathbf{W})$.
- $\mathcal{T}_q \leftarrow \textbf{SearchToken}(K_1, K_2, q)$: It takes the secret keys K_1 and K_2, a query q as input, and outputs the search token $\mathcal{T}_q$.
- $\mathcal{R} \leftarrow \textbf{Search}(\mathcal{T}_q, \mathcal{I})$: It takes the search token $\mathcal{T}_q$ and encrypted indexes $\mathcal{I}$ as input to return the query results $\mathcal{R}$ corresponding to $\mathcal{T}_q$.
- $\{\textbf{True}, \textbf{False}\} \leftarrow \textbf{Verify}(\mathcal{R}, Acc(\mathbf{W}), q)$: Given the query results $\mathcal{R}$, accumulators $Acc(\mathbf{W})$, and a query q, it outputs **True** if the verification is successful, otherwise, the algorithm outputs **False**.
- $(\tau_a, Acc(\mathbf{W})') \leftarrow \textbf{AddToken}(PK, SK, Acc(\mathbf{W}), \mathrm{w_i}, \mathrm{f_i})$: Given a public key PK, a secret key SK, an added file $\mathrm{f_i}$, a keyword set $\mathrm{w_i}$ contained in file $\mathrm{f_i}$, and accumulators $Acc(\mathbf{W})$, it outputs an addition token τ_a used to update operation and accumulators $Acc(\mathbf{W})'$.
- $\mathcal{I}' \leftarrow \textbf{Add}(\mathcal{I}, \tau_a)$: It takes the encrypted indexes $\mathcal{I}$ and an addition token τ_a as input, and outputs new encrypted indexes $\mathcal{I}'$.
- $(\tau_d, Acc(\mathbf{W})') \leftarrow \textbf{DelToken}(PK, SK, Acc(\mathbf{W}), \mathrm{w_i}, \mathrm{f_i})$: Given a public key PK, a secret key SK, a deleted file $\mathrm{f_i}$, a keyword set $\mathrm{w_i}$ contained in file $\mathrm{f_i}$, and accumulators $Acc(\mathbf{W})$, it outputs a deletion token τ_d used to update operation and accumulators $Acc(\mathbf{W})'$.
- $\mathcal{I}' \leftarrow \textbf{Del}(\mathcal{I}, \tau_d)$: It takes the encrypted indexes $\mathcal{I}$ and a deletion token τ_d as input, and outputs new encrypted indexes $\mathcal{I}'$.
- $\mathrm{c_i} \leftarrow \textbf{Enc}(K_e, \mathrm{f_i})$: It takes the plaintext file $\mathrm{f_i}$ and a secret key K_e as input to generate the encrypted file $\mathrm{c_i}$.
- $\mathrm{f_i} \leftarrow \textbf{Dec}(K_e, \mathrm{c_i})$: It takes the encrypted file $\mathrm{c_i}$ and a secret key K_e as input to generate the plaintext file $\mathrm{f_i}$.

The VDMS scheme is correct if for the security parameter $\lambda \in \mathbb{N}$, the public key PK, and the secret key SK generated by $\textbf{KeyGen}(1^\lambda)$, for all files set $\mathbf{F}$ and keyword set $\mathbf{W}$, for all encrypted indexes $\mathcal{I}$ and accumulators $Acc(\mathbf{W})$ generated by $\textbf{BuildIndex}(\cdot)$, and for all tokens, it can always return correct results $\mathcal{R}$ or perform correctly over encrypted indexes.

3.4 Security Definitions

We use the simulation-based game to prove the security of our proposed scheme, which is defined as follows:

Definition 1. (**negligible function**) *Let f be a negligible function, if for every polynomial $p(\cdot)$, there is an integer N, such that for all integers $n > N$, there is $f(n) < \frac{1}{p(n)}$.*

Definition 2. ($\textbf{VDMS CKA2} - \textbf{Security}$) *Let $\mathcal{L}$ be the leakage function and VDMS $\Gamma = (\textbf{KeyGen}, \textbf{BuildIndex}, \textbf{Search}\ Token, \textbf{AddToken}, \textbf{DelToken}, \textbf{Search}, \textbf{Add}, \textbf{Del}, \textbf{Verify}, \textbf{Enc}, \textbf{Dec})$, and consider the following probabilistic experiment, where $\mathcal{A}$ is the adversary and $\mathcal{S}$ is the simulator:*

- $\textbf{Real}_{\mathcal{A}}(\lambda)$: Given the security parameter λ, the challenger runs $\textbf{KeyGen}(1^\lambda)$ to generate the public key PK and the secret key SK, $\mathcal{A}$ outputs $\mathbf{F}$ and $\mathbf{W}$, then receives $\mathcal{I}$ from the challenger according to $\textbf{BuildIndex}(\mathbf{F}, \mathbf{W}, PK, SK)$. $\mathcal{A}$ generates a polynomial number of queries q, which can be used to search, add, or delete. For each query q, $\mathcal{A}$ receives $\mathcal{T}_q$ from $\textbf{SearchToken}(K_1, K_2, q)$ if q is a search query. Likewise, $\mathcal{A}$ receives $\tau_a \leftarrow \textbf{AddToken}(PK, SK, \mathrm{w_i}, \mathrm{f_i})$ from the challenger if q is an addition query or receives $\tau_d \leftarrow \textbf{DelToken}(PK, SK, \mathrm{w_i}, \mathrm{f_i})$ if q is a deletion query. Finally, $\mathcal{A}$ returns a bit as the output of the probabilistic experiment.
- $\textbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$: $\mathcal{A}$ outputs $\mathbf{F}$ and $\mathbf{W}$, and receives the indexes $\mathcal{I}$ generated by the leakage function $\mathcal{L}_1(\mathbf{F}, \mathbf{W})$ from the simulator $\mathcal{S}$. $\mathcal{A}$ generates a polynomial number of queries $q \in \{\mathrm{w}, \mathrm{f}\}$. If q is a search query, the simulator $\mathcal{S}$ generates $\mathcal{T}_q$ according to $\mathcal{L}_2(\mathbf{F}, \mathbf{W}, \mathrm{w})$ and sends it to $\mathcal{A}$. If q is an addition or deletion query, the simulator $\mathcal{S}$ generates τ_a or τ_d based on $\mathcal{L}_3(\mathbf{F}, \mathbf{W}, \mathrm{w}, \mathrm{f_j})$ or $\mathcal{L}_4(\mathbf{F}, \mathbf{W}, \mathrm{w}, \mathrm{f_j})$ and sends it to $\mathcal{A}$. Finally, $\mathcal{A}$ returns a bit as the output of the probabilistic experiment.

We say VDMS is $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4)$-secure against CKA2, if for all probabilistic polynomial time adversaries $\mathcal{A}$, there exists an efficient simulator $\mathcal{S}$, such that

2. Files can be encrypted using any valid encryption algorithm.

TABLE 2
Summary of Notations

Notations	Descriptions
$\mathbf{F}$	the file set consists of n files
$\mathbf{C}$	the encrypted file set consists of n encrypted files
$\mathbf{W}$	the keyword set consists of m distinct keywords
T_s	search table
T_f	file table
$\mathcal{I}$	encrypted indexes consist of T_s and T_f
$\mathcal{T}_q$	search token of query q
$Acc(\mathbf{W})$	accumulator sequence of m accumulators $Acc(\omega_i)$
$\mathcal{R}$	query results
$\mathcal{B}_{\omega_i}$	a bitmap stores information about files containing ω_i
$\mathcal{B}_{\mathrm{f_i}}$	a bitmap stores information about keywords in $\mathrm{f_i}$
$id(\cdot)$	denote the identity of keyword/file
τ_a	the addition token
τ_d	the deletion token

$$|\Pr[\mathbf{Real}_{\mathcal{A}}(\lambda)=1]-\Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)=1]|\leq f(\lambda),$$

where f is a negligible function and λ is a security parameter.

4 Our Proposed VDMS

In this section, we present the construction of VDMS in detail. We take the bitmap and RSA accumulator as building blocks in VDMS to efficiently search over encrypted data and verify the correctness of results. The bitmap is utilized to build an inverted index to achieve the optimal search time $\mathcal{O}(|q|)$, where q is the set of query keywords, $|q|$ is the number of elements in q. VDMS combines bitmap with the RSA accumulator to verify the correctness of returned results.

Specifically, the search index contains information about file identities corresponding to the keyword ω_i, and the auxiliary verifiable construction bitmap containing all keywords in a file $\mathrm{f_i}$ combines with the RSA accumulator to verify the correctness of results. Thus, our VDMS supports multi-keyword search, dynamic updates, and result verification. Next, we first give some notations used in VDMS, as shown in Table 2, and then present the concrete construction of VDMS.

4.1 Notations

Let $\lambda\in\mathbb{N}$ be the security parameter in the system, $\mathbf{F}$ is the file set containing n files $\{\mathrm{f_1},\ldots,\mathrm{f_n}\}$, $\mathbf{C}$ is the encrypted files corresponding to $\mathbf{F}$, denoted as $\{\mathrm{c_1},\ldots,\mathrm{c_n}\}$. $\mathbf{W}=\mathrm{w_1}\cup\ldots\cup\mathrm{w_n}$ is the keyword set, where $\mathrm{w_i}=\{\omega_1,\ldots,\omega_s\}$, and ω_i is extracted from $\mathrm{f_i}$. We denote $id(\cdot)$ as the keyword/file identity, and T_s,T_f are denoted as search table and file table, respectively. $s_1||s_2$ is the concatenation of strings s_1 and s_2.

In the search process, the data user transforms the multi-keyword query $q=\{\omega_1,\ldots,\omega_t\}$ into search token $\mathcal{T}_q=\{\tau_1,\ldots,\tau_t\}$, then sends $\mathcal{T}_q$ to the cloud server to obtain the results and auxiliary information. $\mathcal{B}_{\omega_i}$ is the bitmap that stores the identities of files containing the keyword ω_i, and $\mathcal{B}_{\mathrm{f_i}}$ stores all keyword identities contained in the file $\mathrm{f_i}$. $Acc(\mathbf{W})$ is the sequence of accumulators $Acc(\omega_i)$, which are computed as $Acc(\omega_i)=g^{\prod_{j=1:\mathcal{B}_{\omega_i}[j]=1}^{n}j}\bmod\mathbf{N}$.

4.2 Proposed Construction

Here we formally present the concrete construction of VDMS, which consists of eleven algorithms $\Gamma=(\mathbf{KeyGen},\mathbf{BuildIndex},\mathbf{SearchToken},\mathbf{AddToken},\mathbf{DelToken},\mathbf{Search},\mathbf{Add},\mathbf{Del},\mathbf{Verify},\mathbf{Enc},\mathbf{Dec})$. Let $P:\{0,1\}^\lambda\times\{0,1\}^*\rightarrow\{0,1\}^*$, $G:\{0,1\}^\lambda\times\{0,1\}^*\rightarrow\{0,1\}^*$, $H:\{0,1\}^*\rightarrow\{0,1\}^n$, $F:\{0,1\}^*\rightarrow\{0,1\}^m$ be four secure Pseudo-Random Functions (**PRFs**). π_1 and π_2 are two Pseudo-Random Permutations (**PRPs**), where π_1 maps each keyword to range 1 to m and π_2 maps each file to range 1 to n. The VDMS scheme is constructed as follows:

$(PK,SK)\leftarrow\mathbf{KeyGen}(1^\lambda)$: Given a security parameter $\lambda\in\mathbb{N}$, the algorithm first generates $(\mathbf{N},\varphi(\mathbf{N}),p,d,g)$, then samples keys from $K_1,K_2,K_3,K_e\xleftarrow{\$}\{0,1\}^\lambda$, where K_e is the key of the encryption algorithm and K_1,K_2,K_3 are keys of **PRFs**. Finally, the algorithm sets the public key $PK=(\mathbf{N},\varphi(\mathbf{N}),g)$ and the secret key $\mathbf{SK}=(p,d,K_1,K_2,K_3,K_e)$.

Algorithm 1. VDMS Build Index

Input: $SK,PK,\mathbf{F},\mathbf{W}$
Output: $\mathcal{I}=(T_s,T_f),Acc(\mathbf{W})$
Initialize the search table T_s and the file table T_f;
for $\omega_i\in\mathbf{W}$ **do**
  $id(\omega_i)\leftarrow P(K_1,\omega_i)$;
  Generate the bitmap $\mathcal{B}_{\omega_i}$;
  $Acc(\omega_i)=g^{\prod_{j=1:\mathcal{B}_{\omega_i}[j]=1}^{n}j}\bmod\mathbf{N}$;
  $x_i=\mathcal{B}_{\omega_i}\oplus h_i$, where $h_i=H(K_{\omega_i}||r_i)$, $K_{\omega_i}=G(K_2,\omega_i)$ and $r_i\xleftarrow{\$}\{0,1\}^*$;
  $T_s[id(\omega_i)]=(x_i||r_i)$;
for $\mathrm{f_i}\in\mathbf{F}$ **do**
  $id(\mathrm{f_i})\leftarrow\pi_2(\mathrm{f_i})$;
  $\mathrm{c_i}=\mathbf{Enc}(K_e,\mathrm{f_i}),hash_i=SHA(\mathrm{f_i})$;
  Generate the bitmap $\mathcal{B}_{\mathrm{f_i}}$;
  $y_i=\mathcal{B}_{\mathrm{f_i}}\oplus u_i$, where $u_i=F(K_{\mathrm{f_i}}||r_i)$, $K_{\mathrm{f_i}}=G(K_3,\mathrm{f_i})$ and $r_i\xleftarrow{\$}\{0,1\}^*$;
  $T_f[id(\mathrm{f_i})]=(y_i||r_i,\mathrm{c_i},hash_i)$;
return $\mathcal{I}=(T_s,T_f),Acc(\mathbf{W})$.

$(\mathcal{I},Acc(\mathbf{W}))\leftarrow\mathbf{BuildIndex}(\mathbf{F},\mathbf{W},PK,SK)$: Given a file set $\mathbf{F}$, a keyword set $\mathbf{W}$, and keys PK,SK, the algorithm outputs the encrypted indexes $\mathcal{I}=(T_s,T_f)$ and auxiliary information $Acc(\mathbf{W})$. Note that the auxiliary information $Acc(\mathbf{W})$ is kept locally. Algorithm 1 shows the **BuildIndex** process. For each $\omega_i\in\mathbf{W}$, the data owner computes $id(\omega_i)=P(K_1,\omega_i)$, and then generates Bitmap $\mathcal{B}_{\omega_i}$, where $\mathcal{B}_{\omega_i}[j]=1$, $j=\pi_2(\mathrm{f_j})$[3], if $\mathrm{f_j}$ contains keyword ω_i for all files in $\mathbf{F}$, and rest positions of $\mathcal{B}_{\omega_i}$ are all 0's. According to the $\mathcal{B}_{\omega_i}$, the data owner computes the accumulator $Acc(\omega_i)=g^{\prod_{j=1:\mathcal{B}_{\omega_i}[j]=1}^{n}j}\bmod\mathbf{N}$, where j should satisfy the input criteria for the RSA accumulator. (for simplicity, we write the equation like this). After obtaining the Bitmap $\mathcal{B}_{\omega_i}$, the data owner computes $x_i=\mathcal{B}_{\omega_i}\oplus h_i$, where $h_i=H(K_{\omega_i}||r_i)$, $K_{\omega_i}=G(K_2,\omega_i)$, and $r_i\xleftarrow{\$}\{0,1\}^*$. Then the data owner stores $(x_i||r_i)$ in $T_s[id(\omega_i)]$.

For each $\mathrm{f_i}\in\mathbf{F}$, the data owner computes $id(\mathrm{f_i})=\pi_2(\mathrm{f_i})$, encrypts $\mathrm{f_i}$ as $\mathrm{c_i}=\mathbf{Enc}(K_e,\mathrm{f_i})$, and calculates the hash value of $\mathrm{f_i}$ using the SHA algorithm as $hash_i=SHA(\mathrm{f_i})$. Similarly, the data owner generates the bitmap $\mathcal{B}_{\mathrm{f_i}}$, where $\mathcal{B}_{\mathrm{f_i}}[j]=1$, $j=\pi_1(\omega_j)$ for all $\omega_j\in\mathrm{w_i}$, and rest positions of $\mathcal{B}_{\mathrm{f_i}}$ are all 0's. Then the data owner computes $y_i=\mathcal{B}_{\mathrm{f_i}}\oplus u_i$, where $u_i=$

3. π_2 does not necessarily map $\mathrm{f_j}$ to its subscript j, so is π_1.

$F(K_{\mathrm{f_i}}||r_i)$, $K_{\mathrm{f_i}} = G(K_3, \mathrm{f_i})$, and $r_i \xleftarrow{\$} \{0,1\}^*$, and stores the entry $(y_i||r_i, \mathrm{c_i}, hash_i)$ in $T_f[id(\mathrm{f_i})]$.

Algorithm 2. VDMS Search

Input: $SK, q, \mathcal{I}$
Output: $\mathcal{R}$
Client :
for $\omega_i \in q$ **do**
  $id(\omega_i) \leftarrow P(K_1, \omega_i)$, where $K_{\omega_i} = G(K_2, \omega_i)$;
  $\tau_i = (id(\omega_i), K_{\omega_i})$;
return $\mathcal{T}_q = (\tau_1, \ldots, \tau_t)$;
Send $\mathcal{T}_q = (\tau_1, \ldots, \tau_t)$ to the cloud server.
Server :
for $\tau_i \in \mathcal{T}_q$ **do**
  Parse $\tau_i = (id(\omega_i), K_{\omega_i})$;
  if *it* $id(\omega_i)$ *doesn't exist in* T_s **then**
    return $\perp$;
  else
    $(x_i||r_i) = T_s[id(\omega_i)]$;
    $\mathcal{B}_{\omega_i} = x_i \oplus H(K_{\omega_i}||r_i)$;
    $Acc(\omega_i) = g^{\prod_{j=1:\mathcal{B}_{\omega_i}[j]=1}^{n} j} \bmod \mathbf{N}$;
$\mathcal{B} = \mathcal{B}_{\omega_1} \wedge \ldots \wedge \mathcal{B}_{\omega_t}$;
return $\mathcal{R} = (\{(y_i||r_i, \mathrm{c_i}, hash_i)\}, \{Acc(\omega_i)\})$.

$\mathcal{T}_q \leftarrow$ **SearchToken**(K_1, K_2, q): The algorithm takes K_1, K_2, and $q = \{\omega_1, \ldots, \omega_t\}$ as input, then outputs the search token $\mathcal{T}_q$. Specifically, for each $\omega_i \in q$, the data user computes $id(\omega_i) = P(K_1, \omega_i)$ and $K_{\omega_i} = G(K_2, \omega_i)$. Then the data user sets $\tau_i = (id(\omega_i), K_{\omega_i})$ and $\mathcal{T}_q = \{\tau_1, \ldots, \tau_t\}$.

Algorithm 3. VDMS Verify

Input: $\mathcal{R}, Acc(\mathbf{W}), q$
Output: {**True**, **False**}
Generate the bitmap $\mathcal{B}_q$ with regard to q;
for $(y_i||r_i, \mathrm{c_i}, hash_i)$ *in* $\mathcal{R}$ **do**
  $\mathcal{B}_{\mathrm{f_i}} = y_i \oplus u_i$, where $u_i = F(K_{\mathrm{f_i}}||r_i)$ and $K_{\mathrm{f_i}} = G(K_3, \mathrm{f_i})$;
  if $\mathcal{B}_{\mathrm{f_i}} \wedge \mathcal{B}_q = \mathcal{B}_q$ *and* $hash_i = SHA(\mathbf{Dec}(K_e, \mathrm{c_i}))$ **then**
    continue;
  else
    return False;
for $Acc(\omega_i) \in \mathcal{R}$ **do**
  if $Acc(\omega_i) = Acc(\mathbf{W})[\omega_i]$ **then**
    continue;
  else
    return False.
return True.

$\mathcal{R} \leftarrow$ **Search**$(\mathcal{T}_q, \mathcal{I})$: Given a search token $\mathcal{T}_q$ and the encrypted indexes $\mathcal{I}$, the cloud server parses $\tau_i = (id(\omega_i), K_{\omega_i})$ for each $\tau_i \in \mathcal{T}_q$ and searches $id(\omega_i)$ in the search table T_s. If $id(\omega_i)$ does not exist in the search table T_s, the cloud server returns $\perp$. Otherwise, the cloud server gets $(x_i||r_i) = T_s[id(\omega_i)]$, and computes $\mathcal{B}_{\omega_i} = x_i \oplus H(K_{\omega_i}||r_i)$. After obtaining the bitmap $\mathcal{B}_{\omega_i}$, the cloud server computes the auxiliary information accumulator $Acc(\omega_i) = g^{\prod_{j=1:\mathcal{B}_{\omega_i}[j]=1}^{n} j} \bmod \mathbf{N}$. Then for all bitmaps $\{\mathcal{B}_{\omega_1}, \ldots, \mathcal{B}_{\omega_t}\}$, the cloud server computes resulting bitmap $\mathcal{B}$ as follows:

$$\mathcal{B} = \mathcal{B}_{\omega_1} \wedge \ldots \wedge \mathcal{B}_{\omega_t}.$$

According to the $\mathcal{B}$, the cloud sever returns entries $(y_i||r_i, \mathrm{c_i}, hash_i)$ stored in file table T_f with regard to $\mathcal{B}[i] = 1$. Finally, the algorithm outputs $\mathcal{R} = (\{(y_i||r_i, \mathrm{c_i}, hash_i)\}, \{Acc(\omega_i)\})$. The **Algorithm** 2 shows the **Search** process.

{**True**, **False**} $\leftarrow$ **Verify**$(\mathcal{R}, Acc(\mathbf{W}), q)$: Given the results $\mathcal{R}$, an auxiliary information $Acc(\mathbf{W})$, and a query q, the data user verifies the correctness of results. The data user employs $q = \{\omega_1, \ldots, \omega_t\}$ to generate the bitmap $\mathcal{B}_q$, where $\mathcal{B}_q[i] = 1$ if $\omega_i \in q$ and $i = \pi_1(\omega_i)$. For each $(y_i||r_i) \in \mathcal{R}$, the data user computes $\mathcal{B}_{\mathrm{f_i}} = y_i \oplus u_i$, where $u_i = F(K_{\mathrm{f_i}}||r_i)$, $K_{\mathrm{f_i}} = G(K_3, \mathrm{f_i})$, and then checks whether $\mathcal{B}_{\mathrm{f_i}} \wedge \mathcal{B}_q = \mathcal{B}_q$ holds and verifies that file has been modified. For each $Acc(\omega_i) \in \mathcal{R}$, the data user checks whether $Acc(\omega_i) = Acc(\mathbf{W})[\omega_i]$ holds. If results $\mathcal{R}$ satisfy the conditions aforementioned, the algorithm outputs **True**, otherwise, **False**. **Algorithm** 3 shows the **Verify** process.

Algorithm 4. VDMS Addition

Input: $\mathrm{w_i}, \mathrm{f_i}, SK, PK, \mathcal{I}, Acc(\mathbf{W})$
Output: $\mathcal{I}', Acc(\mathbf{W})'$
Client :
Generate the bitmap $\mathcal{B}_{\mathrm{f_i}}$, where $\mathcal{B}_{\mathrm{f_i}}[j] = 1$, $j = \pi_1(\omega_j)$ for each $\omega_j \in \mathrm{w_i}$;
$y_i = \mathcal{B}_{\mathrm{f_i}} \oplus u_i$, where $u_i = F(K_{\mathrm{f_i}}||r_i)$, $K_{\mathrm{f_i}} = G(K_3, \mathrm{f_i})$ and $r_i \xleftarrow{\$} \{0,1\}^*$;
$\mathrm{c_i} = \mathbf{Enc}(K_e, \mathrm{f_i})$, $hash_i = SHA(\mathrm{f_i})$, $id(\mathrm{f_i}) = \pi_2(\mathrm{f_i})$;
for $\omega_j \in \mathrm{w_i}$ **do**
  Generate the bitmap $\mathcal{B}_a$, where $\mathcal{B}_a[i] = 1$, $i = \pi_2(\mathrm{f_i})$;
  $x_j = \mathcal{B}_a \oplus h_j$, where $h_j = H(K_{\omega_j}||r_j)$, $K_{\omega_j} = G(K_2, \omega_j)$ and $r_j \xleftarrow{\$} \{0,1\}^*$;
  $id(\omega_j) = P(K_1, \omega_j)$;
  if $Acc(\omega_j)$ *exists in* $Acc(\mathbf{W})$ **then**
    $Acc(\omega_j)' = Acc(\omega_j)^{\prod_{i=1:\mathcal{B}_a[i]=1}^{n} i} \bmod \mathbf{N}$;
  else
    $Acc(\omega_j) = g^{\prod_{i=1:\mathcal{B}_a[i]=1}^{n} i} \bmod \mathbf{N}$;
return $\tau_{\mathrm{s}} = (\{id(\omega_j)\}, \mathcal{B}_a, \{(x_j||r_j)\})$, $\tau_{\mathrm{f}} = (id(\mathrm{f_i}), (x_i||r_i), \mathrm{c_i}, hash_i)$;
Server :
for $id(\omega_j) \in \tau_{\mathrm{s}}$ **do**
  if $id(\omega_j)$ *exists in* T_s **then**
    $(x_j||r_j) = T_s[id(\omega_j)]$;
    $x'_j = x_j \oplus \mathcal{B}_a$;
  else
    $T_s[id(\omega_j)] = (x_j||r_j)$;
$T_f[id(\mathrm{f_i})] = ((x_i||r_i), \mathrm{c_i}, hash_i)$.

$(\tau_a, Acc(\mathbf{W})') \leftarrow$ **AddToken**$(PK, SK, \mathrm{w_i}, \mathrm{f_i}, Acc(\mathbf{W}))$: Given an added file $\mathrm{f_i}$, a keyword set $\mathrm{w_i}$ contained in $\mathrm{f_i}$, accumulators $Acc(\mathbf{W})$, a public key PK, and a secret key SK, then algorithm outputs an addition token τ_a and the verifiable auxiliary accumulators $Acc(\mathbf{W})'$. According to the keyword set $\mathrm{w_i}$, the data user computes $\mathcal{B}_{\mathrm{f_i}}$ with regard to the added file $\mathrm{f_i}$, where $\mathcal{B}_{\mathrm{f_i}}[j] = 1$, $j = \pi_1(\omega_j)$ for each $\omega_j \in \mathrm{w_i}$, and the rest positions of $\mathcal{B}_{\mathrm{f_i}}$ are all 0's. Then the data user computes $y_i = \mathcal{B}_{\mathrm{f_i}} \oplus u_i$, where $u_i = F(K_{\mathrm{f_i}}||r_i)$, $K_{\mathrm{f_i}} = G(K_3, \mathrm{f_i})$ and $r_i \xleftarrow{\$} \{0,1\}^*$. The added file $\mathrm{f_i}$ is encrypted as $\mathrm{c_i} = \mathbf{Enc}(K_e, \mathrm{f_i})$, $hash_i = SHA(\mathrm{f_i})$, and $id(\mathrm{f_i}) = \pi_2(\mathrm{f_i})$. The data user sets τ_{f} as $\tau_{\mathrm{f}} = (id(\mathrm{f_i}), (x_i||r_i), \mathrm{c_i}, hash_i)$. For each $\omega_j \in \mathrm{w_i}$, the data user computes $id(\omega_j) = P(K_1, \omega_j)$, and generates bitmap $\mathcal{B}_a$, where $\mathcal{B}_a[i] = 1$, $i = \pi_2(\mathrm{f_i})$. Then the data user computes $x_j = \mathcal{B}_a \oplus h_j$, where $h_j = H(K_{\omega_j}||r_j)$, $K_{\omega_j} = G(K_2, \omega_j)$, and $r_j \xleftarrow{\$} \{0,1\}^*$. The data user sets $\tau_{\mathrm{s}} = (\{id(\omega_j)\}, \mathcal{B}_a, \{(x_j||r_j)\})$, and $\tau_a = (\tau_{\mathrm{f}}, \tau_{\mathrm{s}})$. The data user also needs to update the verifiable auxiliary accumulators $Acc(\mathbf{W})$, $Acc(\omega_j)' =$

$Acc(\omega_j)^{\prod_{i=1:\mathcal{B}_a[i]=1}^{n} i} \bmod \mathbf{N}$ for each $\omega_j \in \mathrm{w_i}$. If $Acc(\omega_j)$ doesn't exist in $Acc(\mathbf{W})$, the data user needs to add new $Acc(\omega_j) = g^{\prod_{i=1:\mathcal{B}_a[i]=1}^{n} i} \bmod \mathbf{N}$ to accumulators $Acc(\mathbf{W})$.

$\mathcal{I}' \leftarrow \mathbf{Add}(\mathcal{I}, \tau_a)$: Received τ_a, the cloud server parses τ_a as $(\tau_\mathrm{f}, \tau_\mathrm{s})$. According to $\tau_\mathrm{s} = (\{id(\omega_j)\}, \mathcal{B}_a, \{(x_j, r_j)\})$, the cloud server searches $id(\omega_j)$ in T_s, if $id(\omega_j)$ exists in T_s, the cloud server acquires $(x_j||r_j) = T_s[id(\omega_j)]$, $x'_j = x_j \oplus \mathcal{B}_a$, replaces $(x_j||r_j)$ in T_s with $(x'_j||r_j)$. Otherwise, the cloud server adds the new entry $(x_j||r_j)$ to search table $T_s[id(\omega_j)]$. And the cloud server updates the file table T_f by adding the new entry $(id(\mathrm{f_i}), (y_i||r_i), \mathrm{c_i}, hash_i)$ to $T_f[id(\mathrm{f_i})]$. Finally, $\mathcal{I}' = (T_s, T_f)$. Algorithm 4 shows the **Addition** process.

$(\tau_d, Acc(\mathbf{W})') \leftarrow \mathbf{DelToken}(PK, SK, \mathrm{w_i}, \mathrm{f_i}, Acc(\mathbf{W}))$: Given a deleted file $\mathrm{f_i}$, a keyword set $\mathrm{w_i}$ contained in $\mathrm{f_i}$, accumulators $Acc(\mathbf{W})$, a public key PK, and a secret key SK, the data user outputs a deletion token τ_d and the verifiable auxiliary accumulators $Acc(\mathbf{W})'$. For τ_f, the data user computes $id(\mathrm{f_i}) = \pi_2(\mathrm{f_i})$, then sets $\tau_\mathrm{f} = id(\mathrm{f_i})$. For each $\omega_j \in \mathrm{w_i}$, the data user computes $id(\omega_j) = P(K_1, \omega_j)$, and generates the bitmap $\mathcal{B}_d$, where $\mathcal{B}_d[i] = 1$, $i = \pi_2(\mathrm{f_i})$. Finally, the data user sets $\tau_\mathrm{s} = (\{id(\omega_j)\}, \mathcal{B}_d)$ and $\tau_d = (\tau_\mathrm{f}, \tau_\mathrm{s})$. Besides, the data user needs to update $Acc(\mathbf{W})$ as follows:

$$Acc(\omega_j)' = Acc(\omega_j)^{\prod_{i=1:\mathcal{B}_d[i]=1}^{n} i^{-1} \bmod \varphi(\mathbf{N})} \bmod \mathbf{N}.$$

Algorithm 5. VDMS Deletion

Input: $\mathrm{w_i}, \mathrm{f_i}, SK, PK, \mathcal{I}, Acc(\mathbf{W})$
Output: $\mathcal{I}', Acc(\mathbf{W})'$

Client :
$id(\mathrm{f_i}) = \pi_2(\mathrm{f_i})$;
$\tau_\mathrm{f} = id(\mathrm{f_i})$;
Generate bitmap $\mathcal{B}_d$, where $\mathcal{B}_d[i] = 1$, $i = \pi_2(\mathrm{f_i})$;
for $\omega_j \in \mathrm{w_i}$ **do**
  $id(\omega_j) = P(K_1, \omega_j)$;
  $Acc(\omega_j)' = Acc(\omega_j)^{\prod_{i=1:\mathcal{B}_d[i]=1}^{n} i^{-1} \bmod \varphi(\mathbf{N})} \bmod \mathbf{N}$;
$\tau_\mathrm{s} = (\{id(\omega_j)\}, \mathcal{B}_d)$;
Server :
for $id(\omega_i) \in \tau_\mathrm{s}$ **do**
  $(x_i||r_i) = T_s[id(\omega_i)]$;
  $x'_i = x_i \oplus \mathcal{B}_d$;
Delete entry in $T_f[id(\mathrm{f_i})]$.

$\mathcal{I}' \leftarrow \mathbf{Del}(\mathcal{I}, \tau_d)$: Given the encrypted indexes $\mathcal{I}$ and a delete token τ_d, the cloud server can execute the deletion operation. The cloud server parses τ_d as $(\tau_\mathrm{f}, \tau_\mathrm{s})$, and simply deletes the entry $id(\mathrm{f_i})$ in T_f according to τ_f. As for T_s, the cloud server searches $T_s[id(\omega_i)] = (x_i||r_i)$ for each $id(\omega_i) \in \tau_\mathrm{s}$, then replaces x_i with $x'_i = x_i \oplus \mathcal{B}_d$. **Algorithm** 5 shows the **Deletion** process.

4.3 Illustration Example

(Preparation) Assume that there exists a file set $\mathbf{F} = \{\mathrm{f_1}, \mathrm{f_2}, \mathrm{f_3}, \mathrm{f_4}\}$ and a keyword set $\mathbf{W} = \{\omega_1, \omega_2, \omega_3\}$, ω_1 is contained in $\mathrm{f_1}$, $\mathrm{f_2}$, and $\mathrm{f_3}$, ω_2 is contained in $\mathrm{f_1}$ and $\mathrm{f_2}$, ω_3 is contained in $\mathrm{f_3}$ (Note that, file $\mathrm{f_4}$ is used during the update process, so it is not discussed here). Given a security parameter $\lambda \in \mathbb{N}$, $\mathbf{KeyGen}(1^\lambda)$ outputs (PK, SK) and a prime number set $\mathcal{X} = \{x_1, x_2, x_3, x_4\}$ corresponding to the file set $\mathbf{F}$.

Then the data owner can build encrypted indexes $\mathcal{I}$. For all $\omega_i \in \mathbf{W}$, the data owner generates bitmaps $\mathcal{B}_{\omega_1} = 0111$, $\mathcal{B}_{\omega_2} = 0011$ and $\mathcal{B}_{\omega_3} = 0100$[4], then computes the accumulators as follows:

$$Acc(\omega_1) = g^{\prod_{i=1:\mathcal{B}_{\omega_1}[i]=1}^{4} i} \bmod \mathbf{N} = g^{x_1 \cdot x_2 \cdot x_3} \bmod \mathbf{N},$$
$$Acc(\omega_2) = g^{\prod_{i=1:\mathcal{B}_{\omega_2}[i]=1}^{4} i} \bmod \mathbf{N} = g^{x_1 \cdot x_2} \bmod \mathbf{N},$$
$$Acc(\omega_3) = g^{\prod_{i=1:\mathcal{B}_{\omega_3}[i]=1}^{4} i} \bmod \mathbf{N} = g^{x_3} \bmod \mathbf{N},$$

$Acc(\mathbf{W}) = (Acc(\omega_1), Acc(\omega_2), Acc(\omega_3))$. For all $\mathrm{f_i} \in \mathbf{F}$, the data owner generates bitmaps $\mathcal{B}_{\mathrm{f_1}} = 011$, $\mathcal{B}_{\mathrm{f_2}} = 011$, $\mathcal{B}_{\mathrm{f_3}} = 101$ and encrypts them as $\mathrm{c_i} = \mathbf{Enc}(K_e, \mathrm{f_i})$. For all bitmaps $\mathcal{B}_{\omega_i}$ and $\mathcal{B}_{\mathrm{f_i}}$, the data owner encrypts and stores them in search table T_s and T_f, respectively. Finally, the data owner sends $\mathcal{I} = (T_s, T_f)$ to the cloud server and keeps $Acc(\mathbf{W})$ locally.

(Search) Suppose that the data user wants to find files containing both keywords ω_1 and ω_2, he/she first generates the search token in terms of $q = \{\omega_1, \omega_2\}$, $\mathcal{T}_q = ((id(\omega_1), K_{\omega_1}), (id(\omega_2), K_{\omega_2}))$, and sends $\mathcal{T}_q$ to the cloud server. Upon receiving the search token $\mathcal{T}_q$, the cloud server locates the entries $T_s[id(\omega_1)]$ and $T_s[id(\omega_2)]$, then computes $\mathcal{B}_{\omega_1} = x_1 \oplus H(K_{\omega_1}||r_1)$ and $\mathcal{B}_{\omega_2} = x_2 \oplus H(K_{\omega_2}||r_2)$ to recover the bitmaps. According to the bitmaps $\mathcal{B}_{\omega_1}$ and $\mathcal{B}_{\omega_2}$, the cloud server calculates the accumulators

$$Acc(\omega_1) = g^{\prod_{i=1:\mathcal{B}_{\omega_1}[i]=1}^{4} i} \bmod \mathbf{N} = g^{x_1 \cdot x_2 \cdot x_3} \bmod \mathbf{N},$$
$$Acc(\omega_2) = g^{\prod_{i=1:\mathcal{B}_{\omega_2}[i]=1}^{4} i} \bmod \mathbf{N} = g^{x_1 \cdot x_2} \bmod \mathbf{N},$$

and executes the AND operation on $\mathcal{B}_{\omega_1}$ and $\mathcal{B}_{\omega_2}$

$$\mathcal{B} = \mathcal{B}_{\omega_1} \wedge \mathcal{B}_{\omega_2} = 0111 \wedge 0011 = 0011.$$

From the calculation, $\mathrm{f_1}$ and $\mathrm{f_2}$ are the files to be retrieved, the cloud server sends results $\mathcal{R} = (T_f[1], T_f[2], Acc(\omega_1), Acc(\omega_2))$ to the data user. After receiving $\mathcal{R}$, the data user can verify the correctness of $\mathcal{R}$. First, the bitmap $\mathcal{B}_q$ of query q is generated, $\mathcal{B}_q = 011$, and then the bitmaps $\mathcal{B}_{\mathrm{f_1}}$ and $\mathcal{B}_{\mathrm{f_2}}$ in the results $\mathcal{R}$ are recovered by the keys $u_1 = F(K_{\mathrm{f_1}}||r_1)$ and $u_2 = F(K_{\mathrm{f_2}}||r_2)$. Note that, the search results are correct if and only if the following equations hold:

$$\begin{cases} \mathcal{B}_q \wedge \mathcal{B}_{\mathrm{f_1}} = 011 \wedge 011 = \mathcal{B}_q; \\ \mathcal{B}_q \wedge \mathcal{B}_{\mathrm{f_2}} = 011 \wedge 011 = \mathcal{B}_q; \\ Acc(\mathbf{W})[\omega_1] = Acc(\omega_1); \\ Acc(\mathbf{W})[\omega_2] = Acc(\omega_2). \end{cases}$$

If these equations hold, the data user can recover the files from $\mathrm{f_1} := \mathbf{Dec}(K_e, \mathrm{c_1})$ and $\mathrm{f_2} := \mathbf{Dec}(K_e, \mathrm{c_2})$ and verify the files have been modified.

(Update) Suppose the data user wants to add a file $\mathrm{f_4}$ containing keywords ω_2 and ω_3, the bitmap $\mathcal{B}_{\mathrm{f_4}} = 110$ is generated and encrypted with key $u_4 = F(K_{\mathrm{f_4}}||r_4)$, and the file $\mathrm{f_4}$ is encrypted as $\mathrm{c_4} = \mathbf{Enc}(K_e, \mathrm{f_4})$ and $hash_4 = SHA(\mathrm{f_4})$. The data owner also needs to generate a bitmap $\mathcal{B}_a = 1000$ that modifies the search table T_s. Since the file $\mathrm{f_4}$ may contain

4. In the following, we all use subscripts to denote the results of the corresponding permutation functions.

keywords that do not exist in the search table, $\mathcal{B}_a$ should be encrypted with $h_j = H(K_{\omega_j}||r_j)$, and the addition token $\tau_a = (\tau_{\mathrm{s}}, \tau_{\mathrm{f}})$ is sent to the cloud server, where $\tau_{\mathrm{s}} = (\{id(\omega_2), id(\omega_3)\}, \mathcal{B}_a, \{(x_2||r_2), (x_3||r_3)\})$ and $\tau_{\mathrm{f}} = (id(\mathrm{f}_4), (x_4||r_4), \mathrm{c}_4, hash_4)$. Besides, the verifiable auxiliary accumulators $Acc(\mathbf{W})$ need to be updated. For each $\omega_i \in \{\omega_2, \omega_3\}$, the accumulators are updated as follows:

$$Acc(\omega_i)' = Acc(\omega_i)^{\prod_{i=1:\mathcal{B}_a[i]=1}^{n} i} \bmod \mathbf{N} = Acc(\omega_i)^{x_4} \bmod \mathbf{N}.$$

Given the addition token τ_a, the cloud server can update the encrypted indexes $\mathcal{I}$. We found that the search table contains the records of keywords ω_2 and ω_3, so the cloud server only needs to add file f_4 to the corresponding records by using $\mathcal{B}_a$. Finally, the cloud server adds the contents corresponding to τ_{f} to the file table T_f. As for the deletion operation, the process is similar to the addition operation, so that we will omit it.

4.4 Discussion

Compared with the existing SSE schemes based on linked list, using bitmap as an index structure can significantly reduce storage space and improve search efficiency. However, this is not true in all cases. If most of the bit strings have many empty bits (e.g., 0), it will not reduce the storage. To address the obstacle, we consider compressing bitmap to reduce storage costs and minimize CPU usage. [38] proposed a compressed bitmap EWAH (Enhanced Word-Aligned Hybrid), which uses two different types of words, including dirty word and marker word, to improve computation and storage efficiency. In addition, many compressed bitmap algorithms have been proposed [39], [40], [41], [42], [43]. Therefore, we can adopt these compressed bitmaps to replace the traditional bitmap if most of the bit strings have many empty bits.

In this paper, we focus on joint queries such as $\omega_1 \wedge \ldots \wedge \omega_t$, but it can be easily extended to arbitrary boolean queries of the form $\omega_1 \wedge f(\omega_2, \ldots, \omega_t)$ without changing the construction.

5 Security Analysis

In this section, we will analyze the security of VDMS. Since VDMS supports multi-keyword retrieval and update operations, it will leak more information (i.e., keyword identities, correspondence between keywords and documents, etc.) than static SSE schemes. We use the leakage function and simulation-based game to prove that $\mathbf{Real}_{\mathcal{A}}(\lambda)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ are computationally indistinguishable.

Two tables make up the indexes of VDMS, one stores the keyword identity and the file identities containing the keyword, and the other stores the file identities and corresponding ciphertexts. Note that some operations of VDMS will leak some information to the adversary. In the search process, the keyword identities are used as input, and the corresponding file identity set is output. All output files contain the queried keywords. Similarly, in the addition operation, the cloud server takes the addition token as input and modifies the indexes accordingly. In the deletion operation, the deletion token is used as input to delete the file in the corresponding indexes. Therefore, the information mentioned above will be exposed and exploited by the adversary.

Before analyzing the security of VDMS in detail, we first define the following leakage functions:

- The leakage function $\mathcal{L}_1$, given a file set $\mathbf{F}$ and a keyword set $\mathbf{W}$, is defined as

  $$\mathcal{L}_1(\mathbf{F}, \mathbf{W}) = (\{id(\omega_i)\}_{i\in m}, \{id(\mathrm{f_j})\}_{j\in n}, \{|\mathrm{f_j}|\}_{j\in n}),$$

  where $\{id(\omega_i)\}$ is the set of keyword identities, $\{id(\mathrm{f_j})\}$ is the set of file identities, and $|\mathrm{f_j}|$ is the length of file $\mathrm{f_j}$.
- The leakage function $\mathcal{L}_2$, given a file set $\mathbf{F}$, a keyword set $\mathbf{W}$, and a query keyword set w, is defined as

  $$\mathcal{L}_2(\mathbf{F}, \mathbf{W}, \mathrm{w}) = (\{id(\omega_i)\}_{\omega_i\in \mathrm{w}}, \mathbf{AP}(\mathrm{w})),$$

  where $\{id(\omega_i)\}$ is the identity set of keywords contained in w and $\mathbf{AP}(\mathrm{w})$ is the access pattern which consists of the sequence of file identities, such as $(id(\mathrm{f_1}), \ldots, id(\mathrm{f_t}))$.
- The leakage function $\mathcal{L}_3$, given a file set $\mathbf{F}$, a keyword set $\mathbf{W}$, the added file $\mathrm{f_j}$, and the set of keywords it contains, is defined as

  $$\mathcal{L}_3(\mathbf{F}, \mathbf{W}, \mathrm{w}, \mathrm{f_j}) = (\{id(\omega_i)\}_{\omega_i\in \mathrm{w}}, id(\mathrm{f_j}), |\mathrm{f_j}|),$$

  where $\{id(\omega_i)\}$ is the set of keyword identities contained in w, $id(\mathrm{f_j})$ is the identity of the added file $\mathrm{f_j}$, and $|\mathrm{f_j}|$ is the length of $\mathrm{f_j}$.
- The leakage function $\mathcal{L}_4$, given a file set $\mathbf{F}$, a keyword set $\mathbf{W}$, the deleted file $\mathrm{f_j}$, and the set of keywords it contains, is defined as

  $$\mathcal{L}_4(\mathbf{F}, \mathbf{W}, \mathrm{w}, \mathrm{f_j}) = (\{id(\omega_i)\}_{\omega_i\in \mathrm{w}}, id(\mathrm{f_j})),$$

  where $\{id(\omega_i)\}$ is the set of keyword identities contained in w and $id(\mathrm{f_j})$ is the identity of the file $\mathrm{f_j}$.

Theorem 1. *If the encryption algorithm[5] is secure against chosen-plaintext attacks and functions P, H, G, and F are secure pseudo-random, VDMS is $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4)$-secure against CKA2.*

Proof. We describe a probabilistic polynomial time simulator $\mathcal{S}$ to simulate indexes and a series of tokens. Thus, for all probabilistic polynomial time adversaries, $\mathbf{View}_{\mathcal{A},\mathbf{Real}}$ and $\mathbf{View}_{\mathcal{A},\mathbf{Ideal}}$ are computationally indistinguishable. $\mathcal{S}$ utilizes the leakage function $\mathcal{L}_1$ to simulate the encrypted indexes and represents it with $\widetilde{\mathcal{I}}$. In order to make it impossible for the adversary to distinguish the real environment from the ideal environment, $\mathcal{S}$ needs to follow the dependencies among the leakage information from $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{L}_3$, and $\mathcal{L}_4$ when it generates the search, addition, and deletion tokens. The simulation process is as follows:

(Simulating index) Given

$$\mathcal{L}_1(\mathbf{F}, \mathbf{W}) = (\{id(\omega_i)\}_{i\in m}, \{id(\mathrm{f_j})\}_{j\in n}, \{|\mathrm{f_j}|\}_{j\in n}),$$

$\mathcal{S}$ generates the simulated encrypted indexes $\widetilde{\mathcal{I}}$ from leakage information as follows:

- (Simulating $\widetilde{T}_s$) It initializes the empty search table $\widetilde{T}_s$ which is used to store associations between keyword identities and corresponding bitmaps. Then, it randomly chooses a string $\widetilde{r}_i$ and bit string $\widetilde{x}_i$ of size n and stores them in the

5. The encryption algorithm that used to encrypt the files.

search table $\widetilde{T}_s[id(\omega_i)] = (\widetilde{x}_i||\widetilde{r}_i)$ for $i \in m$, where $id(\omega_i)$ is obtained from $\mathcal{L}_1(\mathbf{F}, \mathbf{W})$.

– (Simulating $\widetilde{T}_f$) It initializes the empty file table $\widetilde{T}_f$ to store the file identities, file ciphertexts, and the bitmaps of keyword identities. $\mathcal{S}$ randomly selects a string $\widetilde{\mathrm{f}}_j$ of length $|\mathrm{f_j}|$ and encrypts it as $\widetilde{\mathrm{c}}_j = \mathbf{Enc}(K_e, \widetilde{\mathrm{f}}_j)$ for $j \in n$, where K_e is sampled from $\{0,1\}^\lambda$, and computes hash value $\widetilde{hash}_i = SHA(\widetilde{\mathrm{f}}_j)$. Then $\mathcal{S}$ randomly selects a bit string $\widetilde{y}_j$ of size m. Finally, $\mathcal{S}$ stores the above information as $\widetilde{T}_f[id(\mathrm{f_j})] = (\widetilde{x}_j||\widetilde{r}_j, \widetilde{\mathrm{c}}_j, \widetilde{hash}_i)$, where $id(\mathrm{f_j})$ is obtained from $\mathcal{L}_1(\mathbf{F}, \mathbf{W})$ and $\widetilde{r}_j$ is randomly chosen by $\mathcal{S}$.

According to the leakage information $\mathcal{L}_1(\mathbf{F}, \mathbf{W})$, $\mathcal{S}$ simulates the encrypted indexes $\widetilde{\mathcal{I}} = (\widetilde{T}_s, \widetilde{T}_f)$. The difference between the real environment and the ideal environment lies in the randomly selected bitmap $\widetilde{x}_i$ and the ciphertext $\widetilde{\mathrm{c}}_i$ generated from the encrypted random string. Because the encryption scheme we adopted is secure against CPA, and H and G are secure pseudo-random functions, the probabilistic polynomial time adversary $\mathcal{A}$ cannot distinguish the real environment from the ideal environment with a non-negligible probability.

However, simulating search, addition, and deletion tokens are more complicated than simulating the encrypted indexes. This is because these tokens need to satisfy the dependence between leaked information and be consistent with each other. Otherwise, the adversary can easily distinguish between the real environment and the ideal environment when executing simulated tokens on simulated indexes. For this purpose, $\mathcal{S}$ needs to construct internal data structures (T_L, T_{K_ω}, T_H) to record the leakage information. The table T_L is used to store the information exposed by the leakage function, such as the file identities associated with keyword identity $id(\omega_i)$, and the table T_{K_ω} stores keys K_{ω_i} associated with keyword identities $id(\omega_i)$, and T_H holds oracle queries and associated responses. Thus, $\mathcal{S}$ needs to set up internal data structures before starting the next simulation.

- T_L: Let T_L be an empty table and set $T_L[id(\omega_i)] = L_{x_i}$ for all $id(\omega_i)$ from leakage information $\mathcal{L}_1(\mathbf{F}, \mathbf{W})$. L_{x_i} is an all-zero bitmap of length n, which shows the file identities associated with the keyword identity $id(\omega_i)$. During the next operations, such as search, addition, and deletion, $\mathcal{S}$ will update L_{x_i} with leakage information from $\mathcal{L}_2(\mathbf{F}, \mathbf{W}, \mathrm{w})$, $\mathcal{L}_3(\mathbf{F}, \mathbf{W}, \mathrm{w}, \mathrm{f_j})$, and $\mathcal{L}_4(\mathbf{F}, \mathbf{W}, \mathrm{w}, \mathrm{f_j})$.
- T_{K_ω}: The search token consists of $id(\omega_i)$ and K_{ω_i}, where K_{ω_i} is generated by pseudo-random function G. Here $\mathcal{S}$ uses a random string $\widetilde{K}_{\omega_i}$ to replace the key generated by the pseudo-random function in the real environment, and sets $T_{K_\omega}[id(\omega_i)] = \widetilde{K}_{\omega_i}$ for all $id(\omega_i)$ from leakage information $\mathcal{L}_1(\mathbf{F}, \mathbf{W})$. As for adversary $\mathcal{A}$, the simulated key $\widetilde{K}_{\omega_i}$ is indistinguishable from the real environment under the assumption that G is a pseudo-random function.
- T_H: Here, T_H is simulated to act as a random oracle, just like the pseudo-random function H. $\mathcal{S}$ initializes an empty table, and sets $T_H[(s_1||s_2)] = \widetilde{h}$, where $s_1||s_2$ is the concatenation of random strings s_1 and s_2, and $\widetilde{h}$ is a bit string of length n, corresponding to $s_1||s_2$. When a query $(s_1||s_2)$ is received, the T_H returns the corresponding $\widetilde{h}$. But there are two cases to be considered as follows:
  (1) If the query $(s_1||s_2)$ exists in table T_H, $\mathcal{S}$ returns $\widetilde{h}$ with regard to $T_H[(s_1||s_2)]$.
  (2) If $(s_1||s_2)$ does not exist in table T_H, $\mathcal{S}$ needs to determine whether $(s_1||s_2) \stackrel{?}{=} (\widetilde{K}_{\omega_i}||\widetilde{r}_i)$ holds for each $i \in m$. If the equation holds, $\mathcal{S}$ computes $\widetilde{h} = \widetilde{x}_i \oplus L_{x_i}$, where $\widetilde{x}_i$ exists in $\widetilde{T}_s$ and L_{x_i} exists in T_L. Otherwise, $\mathcal{S}$ returns a random string of size n as $\widetilde{h}$.

(Simulating search tokens) Given

$$\mathcal{L}_2(\mathbf{F}, \mathbf{W}, \mathrm{w}) = (\{id(\omega_i)\}_{\omega_i \in \mathrm{w}}, \mathbf{AP}(\mathrm{w})),$$

where $\mathbf{AP}(\mathrm{w})$ is a set of file identities that contain the keyword set w. In this simulation, $\mathcal{S}$ needs to update the internal data structures according to the leakage information of $\mathcal{L}_2(\mathbf{F}, \mathbf{W}, \mathrm{w})$, calculate the simulated search token, and reflect the search results in the simulated indexes.

The leaked information revealed by $\mathcal{L}_2(\mathbf{F}, \mathbf{W}, \mathrm{w})$ is a set of file identities that contain all the keywords in set w, $\mathcal{S}$ can use this leaked information to update T_L in the following way:

$$T_L[id(\omega_i)] = L_{x_i} \oplus \mathcal{B},$$

for all $\omega_i \in \mathrm{w}$, where $\mathcal{B}$ is a bitmap, whose ith position is set to 1 if $i \in \mathbf{AP}(\mathrm{w})$, otherwise to 0.

Then $\mathcal{S}$ simulates the search tokens $\mathcal{T}_q = (\tau_1, \ldots, \tau_t)$ for the query w, and τ_s is denoted as

$$\tau_s = (id(\omega_i), \widetilde{K}_{\omega_i}),$$

where $id(\omega_i)$ is exposed by the leakage information $\mathcal{L}_2(\mathbf{F}, \mathbf{W}, \mathrm{w})$ and $\widetilde{K}_{\omega_i}$ is obtained from T_{K_ω}. $\mathcal{S}$ uses the simulated search token $\mathcal{T}_q$ to search on the simulated indexes $\widetilde{\mathcal{I}}$, note that the leaked information is the same as $\mathcal{L}_2(\mathbf{F}, \mathbf{W}, \mathrm{w})$. Finally, $\mathcal{S}$ should update the simulated indexes $\widetilde{\mathcal{I}} = (\widetilde{T}_s, \widetilde{T}_f)$ as follows:

$\mathcal{S}$ computes $\widetilde{x}_i = L_{x_i} \oplus \widetilde{h}'$ and sets $\widetilde{T}_s[id(\omega_i)] = (\widetilde{x}_i||\widetilde{r}'_i)$. The search process does not involve the file modification, thus there is no need to update $\widetilde{T}_f$.

(Simulating add tokens) Given

$$\mathcal{L}_3(\mathbf{F}, \mathbf{W}, \mathrm{w}, \mathrm{f_j}) = (\{id(\omega_i)\}_{\omega_i \in \mathrm{w}}, id(\mathrm{f_j}), |\mathrm{f_j}|),$$

where $\mathrm{f_j}$ is an added file and the keyword set w is contained in $\mathrm{f_j}$. In this simulation process, $\mathcal{S}$ needs to update the internal data structures according to the leakage information, calculate the simulated add token, and reflect the addition results in the simulated indexes.

The simulator updates the internal data structures as follows:

- T_L: With the leaked information $\mathcal{L}_3(\mathbf{F}, \mathbf{W}, \mathrm{w}, \mathrm{f_j})$, $\mathcal{S}$ could update the T_L, which reveals the association between file $\mathrm{f_j}$ and the keyword set w. $\mathcal{S}$ generates a bitmap $\mathcal{B}$ of length n, note that the jth position of $\mathcal{B}$ is set to 1, and the rest of positions are set to 0. Then $\mathcal{S}$ updates the T_L for all $id(\omega_i) \in \{id(\omega_i)\}_{\omega_i \in \mathrm{w}}$, but some of the keyword identities in the set $\{id(\omega_i)\}_{\omega_i \in \mathrm{w}}$ do not exist in T_L. If $id(\omega_i)$ does not

exist in T_L, $\mathcal{S}$ adds a new record to T_L as $T_L[id(\omega_i)] = L_{x_i}$, where L_{x_i} is replaced by $\mathcal{B}$. Otherwise, L_{x_i} is updated as follows,

$$T_L[id(\omega_i)] = L_{x_i} \oplus \mathcal{B}.$$

- T_{K_ω}: If there is a keyword identity $id(\omega_i) \in \{id(\omega_i)\}_{\omega_i \in \mathrm{w}}$ but does not exist in T_{K_ω}, then $\mathcal{S}$ randomly selects a string as the $\widetilde{K}_{\omega_i}$ corresponding to the $id(\omega_i)$, and sets $T_{K_\omega}[id(\omega_i)] = \widetilde{K}_{\omega_i}$. Otherwise, there is no change in T_{K_ω}.
- T_H: For each $id(\omega_i) \in \{id(\omega_i)\}_{\omega_i \in \mathrm{w}}$, $\mathcal{S}$ randomly generates the corresponding string $\widetilde{r}_i$ and the bit string $\widetilde{h}_i$ of length n, and sets $T_H[(\widetilde{K}_{\omega_i}||\widetilde{r}_i)] = \widetilde{h}_i$.

After updating the internal data structures, $\mathcal{S}$ simulates the addition token τ_a,

$$\tau_{\mathrm{s}} = (\{id(\omega_i)\}, \mathcal{B}, \{(\widetilde{x}_i||\widetilde{r}_i)\}), \tau_{\mathrm{f}} = (id(\mathrm{f}_\mathrm{j}), (\widetilde{y}_j||\widetilde{r}_j), \widetilde{\mathrm{c}}_j, \widetilde{hash}_i),$$

where $\widetilde{x}_i$ in τ_{s} is computed as $\widetilde{x}_i = \widetilde{h}_i \oplus \mathcal{B}$, $\widetilde{x}_j$ is a random bit string of size m, $\widetilde{r}_j$ is a random string, $\widetilde{\mathrm{c}}_j = \mathbf{Enc}(\widetilde{K_e}, \widetilde{\mathrm{f}}_j)$, $\widetilde{\mathrm{f}}_j$ is randomly chosen by $\mathcal{S}$ of size $|\mathrm{f}_\mathrm{j}|$, and $\widetilde{hash}_i = SHA(\mathrm{f}_\mathrm{j})$. The information disclosed during this simulation is the same as that disclosed by $\mathcal{L}_3(\mathbf{F}, \mathbf{W}, \mathrm{w}, \mathrm{f}_\mathrm{j})$. Finally, $\mathcal{S}$ updates the simulated indexes $\widetilde{\mathcal{I}} = (\widetilde{T}_s, \widetilde{T}_f)$ as follows:

If there is no $id(\omega_i) \in \{id(\omega_i)\}_{\omega_i \in \mathrm{w}}$ in $\widetilde{T}_s$, $\mathcal{S}$ adds a new record to $\widetilde{T}_s$ as $\widetilde{T}_s[id(\omega_i)] = (\widetilde{x}_i||\widetilde{r}_i)$. Otherwise, $\mathcal{S}$ updates the corresponding value $\widetilde{x}_i = \widetilde{x}_i \oplus \mathcal{B}$. For the table $\widetilde{T}_f$, $\mathcal{S}$ sets $\widetilde{T}_f[id(\mathrm{f}_\mathrm{j})] = (\widetilde{y}_j||\widetilde{r}_j, \widetilde{\mathrm{c}}_j, \widetilde{hash}_i)$.

(Simulating delete tokens) Given

$$\mathcal{L}_4(\mathbf{F}, \mathbf{W}, \mathrm{w}, \mathrm{f}_\mathrm{j}) = (\{id(\omega_i)\}_{\omega_i \in \mathrm{w}}, id(\mathrm{f}_\mathrm{j})).$$

Similar to the previous simulation process, $\mathcal{S}$ needs to update the internal data structures, generate a simulated token for deletion, and reflect the simulation results on the simulated encrypted indexes. $\mathcal{S}$ works as follows.

The leakage function $\mathcal{L}_4(\mathbf{F}, \mathbf{W}, \mathrm{w}, \mathrm{f}_\mathrm{j})$ reveals the keyword identity set $\{id(\omega_i)\}_{\omega_i \in \mathrm{w}}$ contained in the deleted file f_j. Therefore, $\mathcal{S}$ updates the L_{x_i} corresponding to the $id(\omega_i)$ in the T_L for all $id(\omega_i) \in \{id(\omega_i)\}_{\omega_i \in \mathrm{w}}$ as follows:

$$T_L[id(\omega_i)] = L_{x_i} \oplus \mathcal{B}.$$

Then $\mathcal{S}$ simulates the deletion token

$$\tau_d = (id(\mathrm{f}_\mathrm{j}), \{id(\omega_i)\}, \mathcal{B}),$$

where $\mathcal{B}$ is a bitmap of size n, note that the jth position of $\mathcal{B}$ is set to 1, and the rest positions are set to 0.

The simulated deletion procedure reveals the same information as $\mathcal{L}_4(\mathbf{F}, \mathbf{W}, \mathrm{w}, \mathrm{f}_\mathrm{j})$. Finally, $\mathcal{S}$ reflects the simulation results on the simulated encrypted indexes as follows:

For each $id(\omega_i) \in \{id(\omega_i)\}_{\omega_i \in \mathrm{w}}$, $\mathcal{S}$ updates the value of $\widetilde{T}_s[id(\omega_i)]$ as $\widetilde{x}_i = \widetilde{x}_i \oplus \mathcal{B}$. And as for table $\widetilde{T}_f$, $\mathcal{S}$ just needs to delete the record of $\widetilde{T}_f[id(\mathrm{f}_\mathrm{j})] = (\widetilde{y}_j||\widetilde{r}_j, \widetilde{\mathrm{c}}_j, \widetilde{hash}_i)$.

In this section, we describe the process of simulator $\mathcal{S}$ using the leakage information $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4)$ to simulate the real world. Under the assumptions that the encryption algorithm is secure against chosen-plaintext attacks and P, H, G, and F are secure pseudo-random functions, the probabilistic polynomial time adversary $\mathcal{A}$ cannot distinguish between the real world and the ideal world with a non-negligible probability. Furthermore, [35] has proved that the RSA accumulator is a secure universal accumulator under the strong RSA assumption. □

6 Performance Evaluation

In this section, we evaluate the performance of VDMS by comparing it with BSSE [28] and BXT-Bitmap (in the following, we refer to it as BXT for short) [29]. The evaluation demonstrates that VDMS is more practical than BSSE and BXT. Since the BSSE scheme does not support multi-keyword search over encrypted data, we compare VDMS with BXT supporting the multi-keyword search. Besides, we compare our VDMS with the BSSE scheme in terms of dynamic operations.

Implementation and Setup. The experiments are simulated with Java language. The bitmap we use in the index structure is the compressed bitmap EWAH, which can help reduce storage cost and improve efficiency when there are many empty bits in strings, and the code is open source.[6] For the system parameters, we set the security parameter $\lambda = 256$ bits, and the secret keys $K = 256$ bits. In the BuildIndex process, the length of the random string r is arbitrary, but for simplicity, we set its length to 64 bits. We use HMAC-SHA-256 for the pseudo-random functions P and G, SHA-256, for the pseudo-random functions H and F. Since the output of H and F depends on the number of files and the number of keywords, respectively, we concatenate or truncate the output to obtain a bit string of a specified length. For the encryption algorithm, we use AES to encrypt the files. Our experiments are performed on an Intel Core i7 2.9GHz with 16GB memory, where the operating system is macOS Catalina. Moreover, each numerical value is an average of 40 trials.

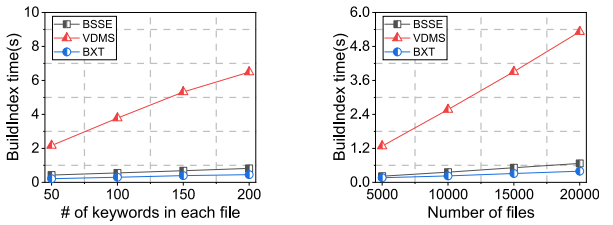
Dataset. To show that VDMS is practical and deployable, we use the real-world dataset 20 Newsgroups[7] to evaluate the performance of VDMS. The dataset consists of 19998 files, and we extract 3000 distinct keywords from the dataset based on tf-idf.[8] In order to understand the impact of the number of keywords contained in each file and the number of files on the performance of VDMS, the number of keywords contained in each file ranges from 50 to 200, and the number of files ranges from 5000 to 19998, respectively.

6.1 Evaluation of BuildIndex

Here, we evaluate and analyze the performance of the BuildIndex. Note that, the indexes generation mainly includes the generation of search table and file table, computation of RSA accumulators, and encryption of files content.

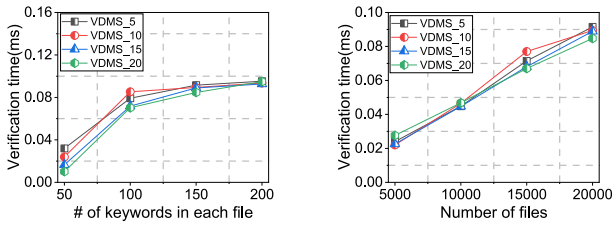
As shown in Fig. 3, we compare the indexes generation time of VDMS with BSSE and BXT. Fig. 3a shows the effect of varying the number of keywords contained in each file on the indexes generation time under 19998 files and Fig. 3b shows the effect of varying the number of files on the indexes generation time under 150 keywords contained in each file. Both figures show that the indexes generation time is affected by the number of files and the number of keywords contained in each

6. JavaEWAH: https://github.com/lemire/javaewah.
7. http://qwone.com/ jason/20Newsgroups/.
8. In information retrieval, tfidf or TFIDF, short for term frequency inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus.



(a) #files = 19998 (b) #keywords = 150

Fig. 3. Performance of the BuildIndex.



(a) #files = 19998 (b) #keywords = 150

Fig. 5. Performance of the verification.

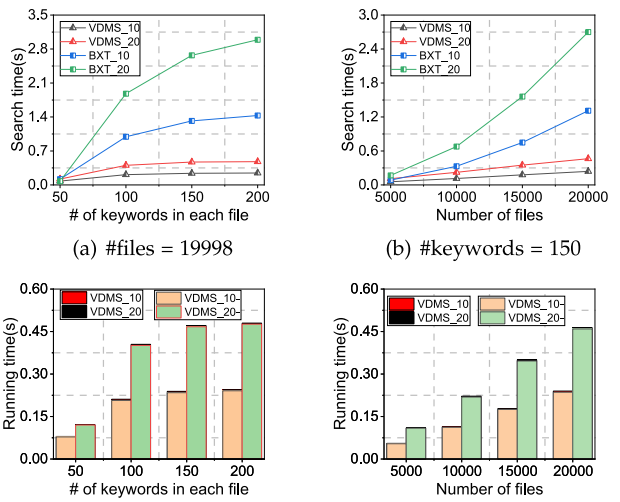
file, respectively. Moreover, the generation time increases linearly concerning the number of files and keywords.

Besides, we notice that VDMS requires more time than the other two schemes as it needs to compute the time-consuming RSA accumulators for verification. However, the process of indexes generation is a one-time computation conducted by the data owner, which does not affect its deployment in practical applications. Therefore, the computational overhead in indexes generation is acceptable.

6.2 Evaluation of Search

The search process includes searching for matching results and calculating RSA accumulators with respect to the queries. For evaluating search performance, we compare VDMS with BXT scheme that provides multi-keyword search but does not support verification, as shown in Fig. 4. It is worth noting that because BXT does not support the verification of the returned results, there are no extra operations in the search process, such as calculating the RSA accumulator. In addition, in order to better evaluate the performance of the search, we perform 10 keywords and 20 keywords contained in the query, respectively. In figures, the different numeric suffixes in the icons indicate the number of keywords in the query. For example, VDMS_10 means that there are 10 keywords in the query under the scheme VDMS.

Figs. 4a and 4b show that the search time of both BXT and VDMS increases as the number of keywords contained in each file or the number of files increases. In addition, the running time grows with the number of keywords in the query. Compared with BXT, VDMS is more efficient when searching the same number of keywords. Since the search time of VDMS is proportional to the number of keywords queried, and the search time of BXT is proportional to the number of keywords and the number of documents queried. Figs. 4c and 4d show the time it takes to execute the search algorithm of the VDMS and the time it takes to calculate the value of the RSA accumulator under different variables. VDMS_10 represents the time it takes to query 10 keywords and VDMS_10- represents the time spent only calculating the value of the RSA accumulator in the process of querying 10 keywords. We can observe that both figures show the same characteristics, that is, the time spent on searching for matching results only accounts for a tiny part of the entire search process.



(a) #files = 19998 (b) #keywords = 150

(c) Search time and calculation time of RSA accumulator with #files = 19998

(d) Search time and calculation time of RSA accumulator with #keywords = 150

Fig. 4. Performance of the search.

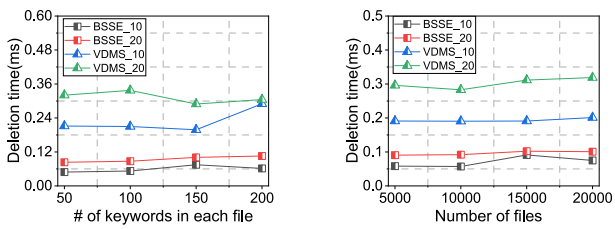
6.3 Evaluation of Verify

Here, we evaluate the impact of the number of keywords and the number of files on the verification. As shown in Fig. 5, the verification time grows with the number of files and the number of keywords contained in the file, respectively. Nevertheless, we noticed that under the same conditions, the running time of different numbers of keywords in the query is relatively similar.

Note that the verification process consists of verifying the correctness of the indexes involved in the calculation and verifying the correctness of the results returned by the cloud server. However, the most time-consuming procedure, calculating the RSA accumulators, has been completed on the cloud server, the data users only need to compare whether the results are equal. Besides, the more keywords are queried, the fewer results are returned. Therefore, with different numbers of query keywords, the running time is similar.

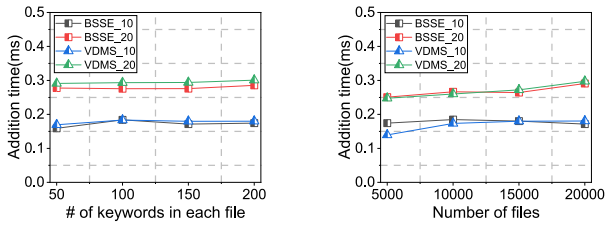
6.4 Evaluation of Update

We present the update evaluation of VDMS by comparing it with that of BSSE. Fig. 6 illustrates the performance of VDMS and BSSE by adding a file. We observe that the two schemes are relatively similar in running time under the



(a) #files = 19998 (b) #keywords = 150

Fig. 6. Performance of the addition operation.



(a) #files = 19998 (b) #keywords = 150

Fig. 7. Performance of the deletion operation.

same conditions. Moreover, the running time is related to the number of keywords contained in the added file since the indexes corresponding to all keywords contained in the file need to be updated.

For the deletion algorithm in VDMS, the user has to delete the element in terms of deleted file from the RSA accumulators by calculating modulo multiplicative inverse. Since BSSE is unverifiable, it lacks the calculation process. As shown in Fig. 7, the running time of BSSE is slightly less than that of VDMS, which is consistent with our theoretical analysis.

7 Conclusion

In this paper, we proposed a VDMS scheme combining bitmap and RSA accumulator, which simultaneously supports multi-keyword retrieval, verification, and dynamic updates in an efficient way. Using bitmap as an index structure simplified the retrieval operation of multi-keyword, which improves search efficiency and reduces the storage space of the indexes. The RSA accumulator and bitmap were used to verify the indexes and the returned results, respectively, to ensure the correctness of the returned results. We used the simulation-based game to prove that VDMS is adaptively secure against chosen-keyword attacks, and the experimental experiments demonstrated that VDMS is practical and deployable.

In future research, we will study the potential of enhancing security while maintaining the efficiency and minimize the information leakage during the search and update process. In addition, improving the search efficiency and enriching the search functions under the asymmetric cryptosystem is also a field worth studying, and the key point is how to design the index structure.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grant 62072361, in part by the Fundamental Research Funds for Central Universities under Grant JB211505, in part by the Guangxi Key Laboratory of Cryptography and Information Security under Grant GCIS201917, in part by the Guangxi Key Laboratory of Trusted Software under Grant KX202028, in part by CCF-Tencent Open Fund WeBank Special Funding under Grant CCF-Webank RAGR 20200102, in part by CCF-NSFOCUS Kunpeng Research Fund under Grant CCF-NSFOCUS 20200004, in part by the Henan Key Laboratory of Network Cryptography Technology and the State Key Laboratory of Mathematical Engineering and Advanced Computing under Grant LNCT2020-A06, and in part by Hong Kong Scholar Program under Grant XJ2019038. The work of Kim-Kwang Raymond Choo was supported by Cloud Technology Endowed Professorship.

References

[1] Y. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in *Proc. Appl. Cryptogr. Netw. Secur.*, 2005, pp. 442–455.

[2] J. Shao, R. Lu, Y. Guan, and G. Wei, "Achieve efficient and verifiable conjunctive and fuzzy queries over encrypted data in cloud," *IEEE Trans. Services Comput.*, early access, Jun. 24, 2019, doi: 10.1109/TSC.2019.2924372.

[3] Q. Liu, Y. Tian, J. Wu, T. Peng, and G. Wang, "Enabling verifiable and dynamic ranked search over outsourced data," *IEEE Trans. Services Comput.*, early access, Jun. 11, 2019, doi: 10.1109/TSC.2019.2922177.

[4] F. Hahn and F. Kerschbaum, "Searchable encryption with secure and efficient updates," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2014, pp. 310–320.

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

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

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

[8] 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 Trans. Inf. Forensics Secur.*, vol. 11, no. 12, pp. 2706–2716, Dec. 2016.

[9] R. Ramasamy, S. S. Vivek, P. George, and B. S. R. Kshatriya, "Dynamic verifiable encrypted keyword search using bitmap index and homomorphic MAC," in *Proc. IEEE Int. Conf. Cyber Security Cloud Comput.*, 2017, pp. 357–362.

[10] D. Cash *et al.*, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *Proc. Annu. Cryptol. Conf.*, 2013, pp. 353–373.

[11] S. Faber *et al.*, "Rich queries on encrypted data: Beyond exact matches," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2015, pp. 123–145.

[12] I. Demertzis, D. Papadopoulos, and C. Papamanthou, "Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency," in *Proc. Annu. Int. Cryptol. Conf.* 2018, pp. 371–406.

[13] S. Kamara and T. Moataz, "Boolean searchable symmetric encryption with worst-case sub-linear complexity," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2017, pp. 94–124.

[14] M. Bellare, A. Boldyreva, and A. O'Neill, "Deterministic and efficiently searchable encryption," in *Proc. Annu. Int. Cryptol. Conf.*, 2007, pp. 535–552.

[15] Y. Miao *et al.*, "Privacy-preserving attribute-based keyword search in shared multi-owner setting," *IEEE Trans. Dependable Secure Comput.*, vol. 18, no. 3, pp. 1080–1094, May/Jun. 2021.

[16] Y. Miao, R. Deng, X. Liu, K.-K. R. Choo, H. Wu, and H. Li, "Multi-authority attribute-based keyword search over encrypted cloud data," *IEEE Trans. Dependable Secure Comput.*, early access, Aug. 14, 2019, doi: 10.1109/TDSC.2019.2935044.

[17] M. Naveed, M. Prabhakaran, and C. A. Gunter, "Dynamic searchable encryption via blind storage," in *Proc. IEEE Symp. Secur. Privacy*, 2014, pp. 639–654.

[18] Y. Wu, X. Lu, J. Su, and P. Chen, "An efficient searchable encryption against keyword guessing attacks for sharable electronic medical records in cloud-based system," *J. Med. Syst.*, vol. 40, no. 12, 2016, Art. no. 258.

[19] I. Demertzis, R. Talapatra, and C. Papamanthou, "Efficient searchable encryption through compression," *Proc. VLDB Endowment*, vol. 11, no. 11, pp. 1729–1741, 2018.

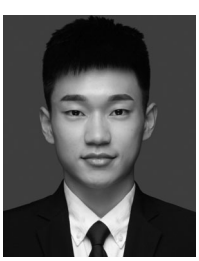
[20] Y. Miao, R. Deng, K.-K. R. Choo, X. Liu, J. Ning, and H. Li, "Optimized verifiable fine-grained keyword search in dynamic multi-owner settings," *IEEE Trans. Dependable Secure Comput.*, early access, Sep. 10, 2019, doi: 10.1109/TDSC.2019.2940573.

[21] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2015, pp. 668–679.

[22] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W. Kim, "Forward secure dynamic searchable symmetric encryption with efficient updates," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2017, pp. 1449–1463.

[23] S. Sun *et al.*, "Practical backward-secure searchable encryption from symmetric puncturable encryption," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2018, pp. 763–780.

[24] D. X. Song, D. A. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proc. IEEE Symp. Secur. Privacy*, 2000, pp. 44–55.
[25] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, "Privacy-preserving multi-keyword ranked search over encrypted cloud data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 1, pp. 222–233, Jan. 2014.
[26] B. Wang, W. Song, W. Lou, and Y. T. Hou, "Inverted index based multi-keyword public-key searchable encryption with strong privacy guarantee," in *Proc. IEEE Conf. Comput. Commun.*, 2015, pp. 2092–2100.
[27] Z. Fu, X. Sun, Q. Liu, L. Zhou, and J. Shu, "Achieving efficient cloud search services: Multi-keyword ranked search over encrypted cloud data supporting parallel computing," *IEICE Trans. Commun.*, vol. 98, no. 1, pp. 190–200, Jan. 2015.
[28] Y. H. Hwang, J. W. Seo, and I. J. Kim, "Encrypted keyword search mechanism based on bitmap index for personal storage services," in *Proc. IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun.*, 2014, pp. 140–147.
[29] M. A. Abdelraheem, C. Gehrmann, M. Lindström, and C. Nordahl, "Executing boolean queries on an encrypted bitmap index," in *Proc. ACM Cloud Comput. Secur. Workshop*, 2016, pp. 11–22.
[30] C. Zuo, S. Sun, J. K. Liu, J. Shao, and J. Pieprzyk, "Dynamic searchable symmetric encryption with forward and stronger backward privacy," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2019, pp. 283–303.
[31] H. Pang and K. Mouratidis, "Authenticating the query results of text search engines," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 126–137, 2008.
[32] K. Kurosawa and Y. Ohtaki, "How to update documents verifiably in searchable symmetric encryption," in *Proc. Cryptol. Netw. Secur.*, 2013, pp. 309–328.
[33] J. Camenisch and A. Lysyanskaya, "Dynamic accumulators and application to efficient revocation of anonymous credentials," in *Proc. Annu. Int. Cryptol. Conf.*, 2002, pp. 61–76.
[34] I. Spiegler and R. Maayan, "Storage and retrieval considerations of binary data bases," *Inf. Process. Manage.*, vol. 21, no. 3, pp. 233–254, 1985.
[35] J. Li, N. Li, and R. Xue, "Universal accumulators with efficient nonmembership proofs," in *Proc. Int. Conf. Appl. Cryptogr. Netw. Secur.*, 2007, pp. 253–269.
[36] A. Soleimanian and S. Khazaei, "Publicly verifiable searchable symmetric encryption based on efficient cryptographic components," *Des., Codes Cryptogr.*, vol. 87, no. 1, pp. 123–147, 2019.
[37] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, no. 3, pp. 431–473, 1996.
[38] D. Lemire, O. Kaser, and K. Aouiche, "Sorting improves word-aligned bitmap indexes," *Data Knowl. Eng.*, vol. 69, no. 1, pp. 3–28, 2010.
[39] J. Chang *et al.*, "PLWAH+: A bitmap index compressing scheme based on PLWAH," in *Proc. ACM/IEEE Symp. Architectures Netw. Commun. Syst.*, 2014, pp. 257–258.
[40] K. Wu, E. J. Otoo, and A. Shoshani, "Compressing bitmap indexes for faster search operations," in *Proc. IEEE Conf. Sci. Statist. Database Manage.*, 2002, pp. 99–108.
[41] S. J. van Schaik and O. de Moor, "A memory efficient reachability data structure through bit vector compression," in *Proc. ACM Int. Conf. Manage. Data*, 2011, pp. 913–924.
[42] F. Fusco, M. P. Stoecklin, and M. Vlachos, "NET-FLi: On-the-fly compression, archiving and indexing of streaming network traffic," *Proc. VLDB Endowment*, vol. 3, no. 1–2, pp. 1382–1393, 2010.
[43] G. Guzun, G. Canahuate, D. Chiu, and J. Sawin, "A tunable compression framework for bitmap indices," in *Proc. IEEE Int. Conf. Data Eng.*, 2014, pp. 484–495.



Feng Li received the BE degree from the School of Cyberspace Security, Hangzhou Dianzi University, Hangzhou, China, in 2018. He is currently working toward the PhD degree with the School of Cyber Engineering, Xidian University. His current research interests focuses on cloud computing security.

Jianfeng Ma (Member, IEEE) received the PhD degree in computer software and telecommunication engineering from Xidian University, Xi'an, China, in 1995. From 1999 to 2001, he was a research fellow with Nanyang Technological University, Singapore. He is currently a professor and a PhD supervisor with the Department of Cyber Engineering, Xidian University. His current research interests include information and network security, wireless and mobile computing systems, and computer networks.

Yinbin Miao (Member, IEEE) received the PhD degree from the Department of Telecommunication Engineering, Xidian University, Xi'an, China, in 2016. From 2018 to 2019, he was a postdoctoral fellow with Nanyang Technological University, Singapore. He is currently a lecturer with the Department of Cyber Engineering, Xidian University. His current research interests include information security and applied cryptography.

Qi Jiang received the BS degree in computer science from Shaanxi Normal University in 2005 and the PhD degree in computer science from Xidian University in 2011. He is currently a professor with the School of Cyber Engineering, Xidian University. His research interests include security protocols and IOT security.

Ximeng Liu (Member, IEEE) received the PhD degree from the Department of Telecommunication Engineering, Xidian University, Xi'an, China, in 2015. He is currently a professor with the Key Laboratory of Information Security of Network Systems, College of Mathematics and Computer Science, Fuzhou University, Fuzhou, China. His current research interests include applied cryptography and big data security.

Kim-Kwang Raymond Choo (Senior Member, IEEE) received the PhD degree from the Queensland University of Technology, Australia, in 2006. He is currently the cloud technology endowed professor with the University of Texas at San Antonio (UTSA). He was the recipient of various awards, including the UTSA College of Business Col. Jean Piccione and Lt. Col. Philip Piccione Endowed Research Award for Tenured Faculty in 2018, and ESORICS 2015 Best Paper Award. He is an Australian ComputerSociety Fellow.