Research Article
Achieve Efficient and Privacy-Preserving Compound Substring Query over Cloud

Fan Yin,1,2 Rongxing Lu,2 Yandong Zheng,2 and Xiaohu Tang1

1The Information Security and National Computing Grid Laboratory, Southwest Jiaotong University, Chengdu 611756, China
2Faculty of Computer Science, University of New Brunswick, Fredericton, Canada

Correspondence should be addressed to Rongxing Lu; rlu1@unb.ca

Received 2 August 2021; Revised 27 September 2021; Accepted 12 October 2021; Published 3 December 2021

1. Introduction

The rapid development of information techniques has been promoting the explosive growth of data. In order to mitigate the local storage and computing pressure, an increasing number of individuals and organizations tend to store and process their databases in the cloud [1, 2]. However, since the cloud server may not be fully trustable, those databases with some sensitive information (e.g., electronic health records) have to be encrypted before being outsourced to the cloud. Although the encryption technique can preserve database privacy, it also hides some critical information such that the cloud server cannot well support some users’ query functionality over the encrypted database, e.g., attribute query, which returns a collection of records containing a specific queried attribute.

To deal with the above challenge, the concept of searchable symmetric encryption (SSE) [3] was introduced, which enables the cloud server to search on encrypted records in a very efficient way. Over the past year, in order to improve the query efficiency of SSE, a series of secure index techniques have been designed to match the attributes to corresponding records, such as inverted index [4–7] and tree-based index [8]. Since these index techniques are built with exact attributes, the corresponding SSE schemes can only support the exact attribute query; i.e., the queried attribute must be exactly the same attribute as that stored in cloud.

Recently, to solve the situation that a user only remembers a substring of an attribute rather than the exact attribute, some studies [9–11] designed SSE schemes to support substring queries. However, they just considered a substring query on a single attribute, which cannot be used to achieve a compound substring query on multiple attributes, i.e., the queried records match multiple substring queries for multiple attributes at the same time. Considering...
an example of the compound query that a database DB includes \( n \) records \( \{f_1, f_2, \ldots, f_n\} \) and each record in it has \( \rho \) attributes \( \{A_1, A_2, \ldots, A_\rho\} \), a user can send a compound substring query on two attributes: select \( f \) from DB where \( A_j \) like \( *s_1* \) and \( A_k \) like \( *s_2* \), to query records whose attributes \( A_j \) and \( A_k \) contain substring \( s_1 \) and \( s_2 \), respectively, where \( j \) and \( k \) belong to \([1, \rho] \). A straightforward solution to support the compound query is that users query substrings separately and then calculate the intersection of results. Unfortunately, this solution is inefficient because it leads to a large number of communication overheads and computational costs for the data user.

To address the above problems, in this paper, we propose a privacy-preserving SSE scheme, which can efficiently support compound substring queries on multiple attributes. In specific, the main contributions of this paper are threefold:

(i) First, based on the position heap technique, we design a tree-based index to support substring queries on a single attribute. This tree-based index can support two types of substring patterns: \( *s* \) and \( s_1* s_2 \), where \( s \) and \( s_1 \) and \( s_2 \) represent queried substrings and \( * \) represents any string of any length. In addition, we employ pseudorandom functions and fully homomorphic encryption techniques to encrypt this tree-based index, which can well preserve the privacy of records.

(ii) Second, based on the homomorphism of fully homomorphic encryption, we design an algorithm (see Section 4.2.3) to calculate the intersection of search results for different attributes and therefore achieve the compound substring query on multiple attributes.

(iii) Finally, we analyze the security of our proposed scheme and conduct extensive experiments to evaluate its performance. The results show that our proposed scheme is efficient in terms of computational cost and storage overhead.

The remainder of the paper is organized as follows. We formalize the system model, security model, and design goals in Section 2. Then, we introduce some preliminaries including the position heap technique [12] and the security notion of substring-of-attribute query in Section 3. After that, we present our proposed scheme in Section 4, followed by security analyses and performance evaluation in Section 5 and Section 6, respectively. Some related works are discussed in Section 7. Finally, we draw our conclusions in Section 8.

### 2. Models and Design Goals

In this section, we formalize the system model, security model, and identify our design goals.

#### 2.1. System Model

In our system model, we consider two entities, namely, a data user and a cloud server, as shown in Figure 1.

### (i) Data user: the data user has a database DB with \( n \) records \( \{f_1, f_2, \ldots, f_n\} \) and \( \rho \) attributes \( \{A_1, A_2, \ldots, A_\rho\} \). Each record \( f_i (1 \leq i \leq n) \) in DB includes a unique identifier \( \text{id}_f \), and a set of string-type attributes \( \{a_{i1}, a_{i2}, \ldots, a_{ip}\} \). Due to the limited storage space and computational capability, the data user intends to outsource the database DB and its index, i.e., \( I \), to the cloud server. Later, the data user submits a compound substring query token \( Q = \{\Omega, q_1, \ldots, q_\rho\} \) to the cloud server to retrieve a set of records \( \mathcal{F} \) matching \( \Omega \), where \( q_j (1 \leq j \leq \rho) \) is a substring query for attribute \( A_j \), and \( \Omega \) is a compound formula consisting of conjunctive expressions (i.e., \( \cap \)) and disjunctive expressions (i.e., \( \cup \)) on \( \{q_1, \ldots, q_\rho\} \).

For example, \( \Omega = q_1 \cap \cdots \cap q_\rho \) means matching records’ attribute \( A_j \) contains substring \( q_j \) for \( 1 \leq j \leq \rho \) at the same time.

#### (ii) Cloud server: the cloud server is considered to be powerful in storage space and computational capability. The duties of the cloud server include the following: (i) efficiently store database DB and index \( I \) and (ii) process compound substring query token \( Q \) and respond a set of matching records \( \mathcal{F} \subseteq \text{DB} \) to the data user.

### 2.2. Security Model

In our security model, the data user is considered as trusted, while the cloud server is assumed as honest-but-curious, which means that the cloud server will (i) honestly execute the query processing, return the query results without tampering it and (ii) curiously infer as much sensitive information as possible from the available data. The sensitive information could include the database DB, the index \( I \), and the compound substring query token \( Q \). The formal simulated-based definition for this security model is described in Section 3.4.

### 2.3. Design Goals

In this work, our design goal is to achieve an efficient and privacy-preserving SSE scheme supporting compound substring queries for the database. In particular, the following two requirements should be achieved.

(i) **Privacy Preservation.** In the proposed scheme, all the data obtained by the cloud server, i.e., \( \{DB, I, Q\} \), should be privacy-preserving during the outsourcing and query phases. Formally, the proposed scheme needs to satisfy security Definition 1 in Section 3.4.

(ii) **Efficiency.** In order to achieve the above privacy requirement, additional computational costs will inevitably be incurred. Therefore, in this work, we also aim to reduce the query time to be linear with the length of the query token plus the size of matching results.

### 3. Preliminary

In this section, we give some preliminaries including position heap [12], symmetric key encryption scheme, fully homomorphic encryption, and the security definition of our
Given a string data:edge and pos, which presents the label of the node’s edge considered each node in the position heap stores two types of structure to achieve substring search, such as suffix tree [9].

In the following, we formally describe the PHBuild and PHSearch algorithms of the position heap, which will be efficient in both storage and query time.

3.1. The (Original) Position Heap Technique. Intuitively speaking, the (original) position heap P(t) is a trie built from all the suffixes of t and can be used to achieve efficient substring search for t. To construct the position heap P(t) from a string t = c₁c₂...cₚ, a set of suffixes t[i: p] = cᵢ...cₚ (i ∈ {p, ..., 1}) are chosen and inserted to the P(t), which is initialized as a root node. To do this, for each suffix t[i: p] (i ∈ {p, ..., 1}), its longest prefix t[i: j] (i ≤ j ≤ p) that is already represented by a path in P(t) is found and a new leaf child is added to the last node of this path. The new leaf child is labeled with i and its edge is labeled with t[j + 1] (see Figure 2). Compared to other data structures to achieve substring search, such as suffix tree [9] and suffix array [13], the position heap [12] can achieve high efficiency in both storage and query time.

In the following, we formally describe the PHBuild and PHSearch algorithms of the position heap, which will be used to build a position heap and search on it. Note that we consider each node in the position heap stores two types of data: edge and pos, which present the label of the node’s edge and the label of the node, respectively.

3.1.1. PHBuild Algorithm. Given a string t = c₁c₂...cₚ, the PHBuild (i.e., Algorithm 1) visits the t from the right to left and inserts each position i ∈ {p, p − 1, ..., 1} to the position heap P(t). In particular, for each position i, the algorithm first finds the longest path from the root node of P(t), where its path label is a prefix of cᵢ...cₚ (lines 4–7). Assume that the last node of this longest path is N. Then, the algorithm appends a new leaf child N’ to the N, where N’ · edge = cᵢ₊₁ and N’ · pos = i (lines 8–11). Figure 2 depicts an example to build such a position heap for a string t = bbabbbaba. During inserting position i = 1, this algorithm first finds the longest path (shown in bold) in P(t) and appends a new leaf child N’ to the last node of this path, where N’ · edge = a and N’ · pos = 1.

3.1.2. PHSearch Algorithm. Given a substring s and a position heap P(t), the PHSearch (i.e., Algorithm 2) is supposed to find all the positions in t that are occurrences of s. The time complexity of this algorithm is O(|s|^2 + dₜ), where |s| is the length of the queried substring and dₜ is the number of matching occurrences. The details are as follows:

(i) The algorithm first finds the longest path from the root node of P(t), where its path label denoted by s’ is a prefix of s. We refer to this longest path as a search path in the rest of the article. Then, the algorithm lets L₁ be the set of positions stored in the intermediate nodes along the search path and L₂ be the set of positions stored in the descendants of the last node of the search path (lines 3–18). In particular, if s’ ≠ s, the position stored in the last node of the search path is included in L₁. Otherwise, it is included in L₂.

(ii) After completing the previous step, elements in L₂ must be the matching positions, and elements in L₁ may or may not be the matching positions. Next, the algorithm reviews each position i ∈ L₁ in the string t to filter out unmatching positions and removes them from the L₁. Finally, this algorithm returns L₁ ∪ L₂ (lines 19–23).

Take an example with Figure 2. Given a substring s = bb, the PHSearch algorithm first finds the search path labeled with bb. In this way, L₁ and L₂ are equal to [9] and {5, 1, 4}. Then, this algorithm reviews the string t and makes sure i = 9 ∈ L₁ is not an occurrence of s. Therefore, position 9 is removed from L₁, and L₁ is an empty set now. Finally, this algorithm returns all the positions in L₁ ∪ L₂ = {5, 1, 4}.
3.2. Symmetric Key Encryption Scheme. A symmetric key encryption scheme (SKE) consists of the following three polynomial-time algorithms (KeyGen, Enc, Dec).

(i) $K \leftarrow \text{KeyGen}(1^\lambda)$: it takes a security parameter $\lambda$ as input and outputs a secret key $K$

(ii) $C \leftarrow \text{Enc}(K, M)$: it takes a key $K$ and a message $M$ as inputs and then outputs a ciphertext $C$

(iii) $M \leftarrow \text{Dec}(K, C)$: it takes a key $K$ and a ciphertext $C$ as inputs and then outputs $M$

3.2.1. Correctness. For any message $M$ in plaintext space, it holds that $\text{Dec}(K, \text{Enc}(K, M)) = M$.

3.2.2. Security. In this paper, we consider that the SKE is indistinguishable under a chosen-plaintext attack (IND-CPA) [14], which guarantees that the ciphertext does not leak any information about the plaintext even an adversary can query an encryption oracle. We note that common private-key encryption schemes such as AES in counter mode satisfy this definition.

3.3. (Leveled) Fully Homomorphic Encryption. A leveled fully homomorphic encryption (FHE) scheme [15] consists of four polynomial-time algorithms (KeyGen, Enc, Dec, Eval). The details are described as follows:

(i) $(pk, sk) \leftarrow \text{KeyGen}(1^\lambda, L)$: it takes a security parameter $\lambda$ and a maximum multiplicative depth $L$ as inputs and outputs a public key $pk$ and a secret key $sk$. We assume that a public key $pk$ specifies both the plaintext space $\mathcal{P}$ and the ciphertext space $\mathcal{C}$.

(ii) $m \leftarrow \text{Enc}(pk, m)$: given the public key $pk$ and a plaintext $m$, it outputs a ciphertext $m$. For simplicity, we omit the randomness used for encryption.

![Figure 2: An example of building position heap $P(t)$ for string $t = \text{bbabbbaaba}$. The solid edges in $P(t)$ reflect the insertion for suffix $t[1: 10]$.](image-url)
(iii) \( m \leftarrow \text{Dec}(sk, m) \): given the secret key sk and a ciphertext \( m \), it outputs a message \( m \). 
\( \overline{m}_p \leftarrow \text{Eval}(pk, \phi, \overline{m}_1, \ldots, \overline{m}_l) \). It takes the public key pk, a function \( \phi: P \rightarrow P \), and a set of ciphertexts \( \{\overline{m}_1, \ldots, \overline{m}_l\} \) as inputs and outputs a ciphertext \( \overline{m}_p \).

### 3.3.1. Correctness

For any \( m_i \in \mathcal{P} (1 \leq i \leq l) \) and any function \( \phi: \mathcal{P} \rightarrow \mathcal{P} \) which can be evaluated by a circuit with depth at most \( L \), if \( (pk, sk) \leftarrow \text{KenGen}(1^k) \), \( m \leftarrow \text{Enc}(pk, m_i) \), and \( \overline{m}_p \leftarrow \text{Eval}(pk, \phi, \overline{m}_1, \ldots, \overline{m}_l) \), then it holds that \( \text{Dec}(sk, \overline{m}_p) = \phi(m_1, \ldots, m_l) \).

### 3.3.2. Security

In this work, we consider an FHE scheme is indistinguishable under a chosen-plaintext attack (IND-CPA), which is described in [15].

### 3.3.3. Homomorphic Operations

In general, an FHE scheme can directly support homomorphic bitwise addition (+) and multiplication (·). Other advanced homomorphic operations can be realized by arithmetic circuits based on · and +. In this paper, we consider three types of advanced homomorphic operations: bitwise AND, bitwise OR, and integer equality. In specific, if the FHE ciphertexts of two \( \mu \)-bit integers \( x = x_{\mu-1} \ldots x_0 \) and \( y = y_{\mu-1} \ldots y_0 \) are \( \overline{x} = \{\overline{x}_{\mu-1}, \ldots, \overline{x}_0\} \) and \( \overline{y} = \{\overline{y}_0, \ldots, \overline{y}_{\mu-1}\} \), these arithmetic circuits are defined as follows:

(i) Bitwise AND &: \( \overline{x} \& \overline{y} = \{s_0, \ldots, s_{\mu - 1}\} \), where \( s_i = \overline{x}_i \cdot \overline{y}_i \) for \( i \in [0, \mu - 1] \).

(ii) Bitwise OR |: \( \overline{x} | \overline{y} = \{s_0, \ldots, s_{\mu - 1}\} \), where \( s_i = \overline{x}_i + \overline{y}_i \) for \( i \in [0, \mu - 1] \).

(iii) Integer equality: \( \text{EQ}(\overline{x}, \overline{y}) = \prod_{i=0}^{\mu-1} (1 + \overline{x}_i \cdot \overline{y}_i) \). The output of \( \text{EQ}(\overline{x}, \overline{y}) \) is \( \overline{1} \) in the case of \( x = y \) and \( \overline{0} \) otherwise, where \( \overline{1} \) and \( \overline{0} \) are FHE ciphertexts of 1-bit message 1 and 0, respectively.

### 3.4. Security Definition of Our Proposed Scheme

In this section, we follow the security definition in [5] to formalize the simulated-based security definition of our proposed scheme by using the following two experiments: \( \text{Real}_{\mathcal{A}, \mathcal{E}}(\lambda) \) and \( \text{Ideal}_{\mathcal{A}, \mathcal{E}}(\lambda) \). In the former, the adversary \( \mathcal{A} \), who represents the cloud server, executes the proposed scheme with a challenger \( \mathcal{C} \) that represents the data user. In the latter, \( \mathcal{A} \) also executes the proposed scheme with a simulator \( \mathcal{S} \) that simulates the output of the challenger \( \mathcal{C} \) through the leakage of the proposed scheme. The leakage is parameterized by a leakage function collection \( \mathcal{L} = (\mathcal{L}_D, \mathcal{L}_Q) \), which describes the information leaked to the adversary \( \mathcal{A} \) in the data outsourcing phase and query phase, respectively. If any polynomial adversary \( \mathcal{A} \) cannot distinguish the output information between the challenger \( \mathcal{C} \) and the simulator \( \mathcal{S} \), then we can say there is no other information leaked to the adversary \( \mathcal{A} \), i.e., the cloud server, except the information that can be inferred from the \( \mathcal{L} \). More formally,

(i) \( \text{Real}_{\mathcal{A}, \mathcal{E}}(1^k) = b \in \{0, 1\} \): given a database DB chosen by the adversary \( \mathcal{A} \), the challenger \( \mathcal{C} \) outputs
Definition 1. Our proposed scheme is \( \mathcal{L} \)-secure against adaptive attacks (i.e., \( \mathcal{L} \)-adaptively secure) if, for any probabilistic polynomial-time adversary \( \mathcal{A} \), there exists an efficient simulator \( \mathcal{S} \) such that \( \Pr[\text{Real}_{\mathcal{A},\mathcal{S}}(1^k) \rightarrow 1] = \Pr[\text{Ideal}_{\mathcal{A},\mathcal{S}}(1^k) \rightarrow 1] \leq \text{negl}(k) \).

4. Our Proposed Scheme

In this section, we will present our SSE scheme. Before delving into the details, we first introduce our basic index structure, which is the basic building block of our proposed scheme.

4.1. Basic Index Structure. In order to process efficient compound substring queries, we build an index \( I_{A_j} \) for each attribute \( A_j \) (1 ≤ \( j \) ≤ \( \rho \)) in database DB. The index \( I_{A_j} \) is a modified position heap and can support two types of substring patterns: \( * * \) and \( s_1 * s_2 \). Next, we introduce algorithm Index Build and Index Search, which are used to build index \( I_{A_j} \) and search on it.

4.1.1. Index Build Algorithm. Given an attribute column \( A_j = \{a_{1j}, a_{2j}, \ldots, a_{nj}\} \) of database DB, the IndexBuild algorithm outputs an index \( I_{A_j} \) as follows. It first transforms \( A_j \) to a string \( t_{A_j} = a_{1j} \# a_{2j} \# \cdots \# a_{nj} \), where \( \# \) denotes a character that does not appear in \( A_j \). In this part of the paper, we call \( t_{A_j} \) attribute string. Then, it follows PHBuild algorithm to insert all the positions in \( t_{A_j} \), except positions of character \( \# \), to a position heap \( P(t_{A_j}) \). Finally, this algorithm builds an index \( I_{A_j} \) from \( P(t_{A_j}) \) by replacing its nodes’ position data (i.e., pos) to corresponding identifiers (i.e., id).

Figure 3 gives an example of building an index \( I_{A_j} \) for the first attribute column of database DB.

4.1.2. Index Search Algorithm. Given a substring \( s \) and an index \( I_{A_j} \), the Index Search algorithm follows the PHSearch algorithm to search and outputs a set of identifiers.

4.2. Description of Our Proposed Scheme. In this subsection, we will describe our proposed scheme, which mainly consists of three phases: (i) system initialization; (ii) data outsourcing; (iii) compound substring query. To make the description simple, we first introduce a basic scheme which only supports substring query with \( * * \) pattern, and then extend it to support substring query with \( s_1 * s_2 \) pattern.

4.2.1. System Initialization. Given a security parameter \( \lambda \), the data user first initializes a pseudorandom function (PRF) \( H : \{0, 1\}^* \times \{0, 1\}^\lambda \rightarrow \{0, 1\}^\lambda \), an IND-CPA secure SKE \( \Pi = (\text{KeyGen}, \text{Enc}, \text{Dec}) \), and an IND-CPA secure FHE \( \Sigma = (\text{KeyGen}, \text{Enc}, \text{Dec}, \text{Eval}) \). Then, the data user generates keys \( k_1 \leftarrow \{0, 1\}^\lambda \), \( k_2 = \Pi \cdot \text{KeyGen}(1^\lambda) \), and \( (pk, sk) = \Sigma \cdot \text{KeyGen}(1^\lambda) \).

4.2.2. Data Outsourcing. Assume that the data user has a database DB with \( n \) records \( \{f_1, f_2, \ldots, f_n\} \), where each record \( f_i \) (1 ≤ \( i \) ≤ \( n \)) includes \( \rho \) string-type attributes \( \{a_{1ij}, a_{2ij}, \ldots, a_{njj}\} \). Then, the data user generates a secure index \( I \) and an encrypted database as the following steps:

Step 1: the data user first uses the Index Build algorithm to build \( I_{A_j} \) (1 ≤ \( j \) ≤ \( \rho \)) for each attribute \( A_j \) and then encrypts it as follows:

(i) For each node \( N \) (except the root), the data user encrypts its \( N \cdot \text{id} \) to \( \bar{N} \cdot \bar{\text{id}} = \sum \text{Enc}(pk, N \cdot \text{id}) \)

(ii) For each node \( N \) (except the root), the data user concatenates all the edge labels, i.e., \( N \cdot \text{edge} \), along the path from the root to this node, and calculates the PRF output of the concatenation through pseudorandom function \( H(k_1, \_*) \)

Consider the example in Figure 4, which is encrypted from the index \( I_{A_j} \) in Figure 3(d).

Step 2: the data user encrypts each record \( f_j \in DB \) through \( \Pi \cdot \text{Enc}(k_2, \_*) \) and sends these encrypted records to the cloud server with an encrypted index \( I = \{I_{A_1}, I_{A_2}, \ldots, I_{A_\rho}\} \).

4.2.3. Compound Substring Query. Given a set of substrings \( S = \{s_1, s_2, \ldots, s_k\} \) and a compound formula \( \Omega \) on \( S \), where \( \Omega \) consists of conjunctive expressions (i.e., \( \cap \)) and disjunctive expressions (i.e., \( \cup \)), the data user launches a compound substring query with the cloud server as follows:

Step 1: for each substring \( s_j = c_{1j} \cdots c_{lj} \) with 1 ≤ \( j \) ≤ \( \rho \), the data user calculates \( q_j = (H(k_1, c_{1j}), H(k_1, c_{2j}), \ldots, H(k_1, c_{lj})) \) and sends a compound substring query token \( Q = \{\Omega, q_1, \ldots, q_k\} \) to the cloud server.

Step 2: for each \( q_j \in Q \), the cloud server performs algorithm Index Search to search over \( I_{A_j} \) and outputs a set \( R_j = \{\text{id}_{j1}, \ldots, \text{id}_{j|R_j|}\} \).

Step 3: the cloud server generates a function \( \phi : \{0, 1\}^\rho \rightarrow \{0, 1\} \) according to the compound
idin 1, and every two adjacent attributes in 0 otherwise.

\[ H(k_1, a) \]
\[ id_3 \]
\[ H(k_1, b) \]
\[ id_1 \]
\[ H(k_1, ab) \]
\[ id_2 \]
\[ H(k_1, ba) \]
\[ id_2 \]
\[ H(k_1, bab) \]
\[ id_1 \]
\[ H(k_1, bba) \]
\[ id_1 \]
\[ H(k_1, aba) \]
\[ id_1 \]
\[ H(k_1, ab) \]
\[ H(k_1, ba) \]
\[ H(k_1, b#) \]
\[ H(k_1, bb) \]
\[ H(k_1, b) \]
\[ id_2 \]
\[ id_2 \]
\[ id_1 \]
\[ id_3 \]

Figure 3: An example of building an index \( I_{A_1} \). (a) \( A_1 \) is the first attribute column of database DB, where \( a_{11} = bba \), \( a_{12} = bba \), and \( a_{13} = aba \). (b) To get attribute string \( t_{A_1} \), concatenate all the attributes in \( A_1 \) with character \#. (c) Build a position heap \( P(t_{A_1}) \) for \( t_{A_1} \). (d) For each node \( N \) in \( P(t_{A_1}) \), replace its \( N \cdot \text{pos} \) with the corresponding identifier, called \( N \cdot \text{id} \).

\[ H(k_1, a) \]
\[ id_3 \]
\[ H(k_1, b) \]
\[ id_1 \]
\[ H(k_1, ab) \]
\[ id_2 \]
\[ H(k_1, ba) \]
\[ id_2 \]
\[ H(k_1, bab) \]
\[ H(k_1, bba) \]
\[ H(k_1, aba) \]
\[ H(k_1, ab) \]
\[ H(k_1, ba) \]
\[ H(k_1, b#) \]
\[ H(k_1, bb) \]
\[ H(k_1, b) \]
\[ id_2 \]
\[ id_2 \]
\[ id_1 \]
\[ id_1 \]
\[ id_1 \]
\[ id_1 \]
\[ id_1 \]
\[ id_1 \]
\[ id_1 \]
\[ id_1 \]

Figure 4: An example of the encrypted index \( I_{A_1} \), which is encrypted from Figure 3(d).

Formula \( \Omega \). In specific, all the \( \cap \) and \( \cup \) in \( \Omega \) are replaced by bitwise AND & and bitwise OR |, respectively. For example, if compound formula \( \Omega = q_1 \cup (q_2 \cap \cdots \cap q_{k+1}) \), then \( \phi = t_1 | (t_2 & \cdots & t_{k+1}) \), where \( t_j \in \{0, 1\} \) for \( j \in [0, k] \).

Step 4: the cloud server performs Algorithm 3 to filter matching elements in \( \{R_1, \ldots, R_k\} \) according to function \( \phi \). In specific, assuming that \( R_{\text{min}} \) is the minimum-size element in \( \{R_1, \ldots, R_k\} \), for each encrypted identifier \( \text{id} \) in \( R_{\text{min}} \), this algorithm traverses all encrypted identifiers in \( \{R_1, \ldots, R_k\} \backslash R_{\text{min}} \) to calculate an encrypted flag \( \tilde{f} \in \{0, 1\} \), which is 1 if \( \text{id} \) matches the compound formula \( \Omega \) and 0 otherwise.

Then, the cloud server inserts all the (\( \text{id}, \tilde{f} \)) to an empty set \( \mathcal{F} \) and returns it to the data user. Figure 5 depicts an example of this step where the compound formula \( \Omega = q_1 \cup q_2 \cap q_3 \). In this example, \( \text{id}_1 \) is the only identifier that matches the formula \( \Omega \). Note that, since an EQ algorithm consumes \( \log(\mu) \) multiplicative depth and an AND/OR operation consumes 1 multiplicative depth, this step requires at most \( \lceil \log(\mu) \rceil + \lceil \log(nu - 1) \rceil + \lceil \log(\rho - 1) \rceil \) multiplicative depth, where \( \mu \) is the average attribute length of database and \( n \) is the number of records in the database.

Step 5: for each (\( \text{id}, \tilde{f} \)) \( \in \mathcal{F} \), the data user calculates \( t = \sum \cdot \text{Dec}(sk, \tilde{f}) \). If \( t = 1 \), then the data user calculates \( \text{id} = \sum \cdot \text{Dec}(sk, \text{id}) \) and requests a corresponding encrypted record from the cloud.

4.2.4. Query with \( s_1 \ast s_2 \) Substring. We extend our scheme to support substring queries with \( s_1 \ast s_2 \) pattern. The extension is very simple, which only makes some small changes in the processes of the outsourcing phase and compound substring query phase. In specific, when the data user builds index \( I_{A_j} \) for the attribute column \( A_j = \{a_{1j}, a_{2j}, \ldots, a_{kj}\} \), the attribute string \( t_{A_j} = a_{1j} \# a_{2j} \# \cdots \# a_{kj} \) is replaced by \( t_{A_j} = a_{1j} \# a_{2j} \# a_{2j} \# \cdots \# a_{kj} \# a_{kj} \). In other words, each attribute in \( A_j \) is copied once and concatenated to its replica with character \#. While every two adjacent attributes in \( A_j \) are concatenated with character \#, where \# and \# denote two separate characters that do not appear in \( A_j \). In this way, the substring query with \( s_1 \ast s_2 \) pattern can be transferred to \( s \ast \# \# \ast \# \# \ast s \) pattern where \( s = s_2 \ast s_1 \). Meanwhile, this method leads to double the storage cost of index \( I_{A_j} \) since the size of the attribute string \( t_{A_j} \) is twice longer than before.

5. Security Analysis

In this section, we prove the security of our proposed scheme based on the security definition described in Section 3.4.

5.1. Leakage Function Collection. We first define the leakage function collection \( \mathcal{L} = \{\mathcal{L}_O, \mathcal{L}_Q\} \) of our proposed scheme.

(i) Outsourcing Phase: given the index \( I = \{I_{A_1}, I_{A_2}, \ldots, I_{A_k}\} \) and the encrypted database DB, the leakage \( \mathcal{L}_O \) consists of the following information:

(a) \( n \): the number of records in DB
(b) \( |f_i| (1 \leq i \leq n) \): the size of record \( f_i \in \text{DB} \)
(c) \( \rho \): the number of attributes in DB
(d) \( |I_{A_j}| (1 \leq j \leq \rho) \): the number of nodes in \( I_{A_j} \)
(e) \( |\mathcal{S}_j| (1 \leq j \leq \rho) \): the structural dependencies between nodes in \( I_{A_j} \)

(ii) Query Phase: given the index \( I = \{I_{A_1}, I_{A_2}, \ldots, I_{A_k}\} \) and a compound substring query token \( Q \), the leakage \( \mathcal{L}_Q \) consists of the following information:
(1) let $R_{\min}$ be the minimum-size element in $\{R_1, \ldots, R_n\}$.
(2) let $\mathcal{F}$ be an empty set.
(3) for each $i \in R_{\min}$ do
   (4) $\hat{t}_{\min} \leftarrow T$
   (5) for each $j \in \{1, \ldots, \rho\}/\{\min\}$ do
      (6) $\hat{t}_j \leftarrow \emptyset$
      (7) for each $\mathcal{I}_j \in R_j$ do
         (8) $\hat{t}_j^c \leftarrow \text{EQ}(\mathcal{I}_j, \mathcal{I}_f)$
         (9) $\hat{t}_j = \hat{t}_j^c \cup \hat{t}_{\min}$
      end for
   end for
(11) end for
(12) $\mathcal{I} \leftarrow \sum \cdot \text{Eval}(pk, \phi, \hat{t}_1, \ldots, \hat{t}_\rho)$
(13) insert $(\mathcal{I}, \mathcal{F})$ to $\mathcal{F}$
(14) end for
(15) return $\mathcal{F}$

**Algorithm 3**: Filter elements in $\{R_1, \ldots, R_n\}$ according to function $\phi$.

![Figure 5: An example of Algorithm 3, where the number of attributes $\rho = 3$ and the compound formula $\Omega = q_1 \cap q_2 \cap q_3$.](image)

(a) path pattern: search paths in $I_{A_i}$ $(1 \leq j \leq \rho)$ corresponding to the query token $Q$
(b) $\Omega$: compound formula in $Q$

### 5.2. Security Proof

Now, we prove the security of our proposed scheme based on the leakage function collection $\mathcal{L} = \{\mathcal{L}_O, \mathcal{L}_Q\}$. Intuitively, we first define a simulator $\mathcal{S}$ based on the leakage function collection $\mathcal{L}$ and then analyze the indistinguishability between the output of the $\mathcal{S}$ in the ideal world and the challenger $\mathcal{C}$ (i.e., the data user) in the real world. Finally, we conclude that our proposed scheme does not reveal any information beyond the leakage function collection $\mathcal{L}$ to the server. The details are as follows.

**Theorem 1.** Let $H$ be a pseudorandom function (PRF), let $\Pi$ be an IND-CPA secure symmetric key encryption scheme (SKE), and let $\sum$ be an IND-CPA secure fully homomorphic encryption (FHE). Then, our proposed scheme is $\mathcal{L}$-adaptively secure.

**Proof.** Based on the leakage function collection $\mathcal{L}$, we can build a simulator $\mathcal{S}$ as follows:

(i) Data outsourcing: given the leakage function $\mathcal{L}_O = \{n, |f_i| (1 \leq i \leq n), \rho, |I_{A_i}| (1 \leq j \leq \rho), \Gamma_j (1 \leq j \leq \rho)\},$ the simulator $\mathcal{S}$ is supposed to generate a simulated index $I' = \{I_{A_i}', I_{A_i}, \ldots, I_{A_i}'\}$ and a simulated encrypted database $DB'_i$. To build $I_{A_i}' \in I'$, the simulator $\mathcal{S}$ first generates $|I_{A_i}|$ empty nodes and constructs these nodes to a tree (i.e., $I_{A_i}'$) based on $\Gamma_j$, which means that $I_{A_i}'$ has the same tree structure as $I_{A_i}$. Then, for each node $N$ in $I_{A_i}'$, the simulator $\mathcal{S}$ randomly chooses $N \cdot \text{edge} \in [0,1]^\lambda$ and $N \cdot \text{id} \in [0,1]^d$. Since the outputs of $H$ and $\sum \cdot \text{Enc}$ are pseudorandom, the adversary $\mathcal{A}$ cannot distinguish between $I'$ and $I$. To build $DB'_i$, the simulator $\mathcal{S}$ chooses a random value $r_i \leftarrow [0,1]^{\ell(f)}$ for each record $f_i \in DB$ and lets $DB'_i = [r_1, \ldots, r_n]$. Since $\Pi$ is an IND-CPA secure SKE, the adversary $\mathcal{A}$ cannot distinguish between $DB'_i$ and $DB$.

(ii) Compound substring query: given the leakage function $\mathcal{L}_Q$ for a compound substring query token $Q = \{\Omega, q_1, \ldots, q_\rho\}$, the simulator $\mathcal{S}$ is supposed to generate a simulated encrypted compound substring query token $Q' = \{\Omega, q_1', \ldots, q_\rho\}$. Note that, at this moment, the simulator $\mathcal{S}$ has not only $\mathcal{L}_Q$ but also $\mathcal{L}_O$ and $I'$ from the data outsourcing phase.
Therefore, for each $I_{A_j}' \in \mathcal{I}'$, the simulator $\mathcal{S}$ can follow the corresponding path pattern in $\mathcal{O}_Q$ to find its search path and output all the $N \cdot d$ edge stored in the nodes along the search path as $Q'$. Since $H$ is a pseudorandom function, the adversary $\mathcal{A}$ cannot distinguish between $Q'$ and $Q$.

In summary, as the adversary $\mathcal{A}$ cannot distinguish between the outputs from the simulator $\mathcal{S}$ in the ideal world and the challenger $\mathcal{C}$ in the real world, we can conclude that our proposed scheme is $\mathcal{L}$-adaptively secure.

\section{Performance Evaluation}

In this section, we evaluate the performance of our proposed scheme from both theoretical and experimental perspectives. To the best of our knowledge, we are the first to discuss the compound substring query. Although some existing works [9–11] focus on the substring query, they cannot directly support the compound substring query. Therefore, a fair comparison is quite difficult and we just evaluate our proposed scheme in this section.

\subsection{Theoretical Analysis}

First, we theoretically analyze the query computational cost and storage overhead of our proposed scheme.

For the query computational cost, we analyze the server and user separately. In specific, the query computational cost of the server comes from two steps: search over index $I$ to get a set of collections $\{R_1, \ldots, R_p\}$ and filter matching elements in $\{R_1, \ldots, R_p\}$. The former consumes at most $\sum_{j=1}^{p}|s_j|$ integer comparison operations and the latter consumes at most $\rho \cdot |R_{\min}| \cdot |R_{\max}|$ FHE multiplication operations, where $R_{\min}$ and $R_{\max}$ are the minimum-size and maximum-size elements in $\{R_1, \ldots, R_p\}$. Meanwhile, the query computational cost of the user also comes from two steps: generate query token $Q$ and decrypt the returned FHE ciphertexts, which consumes $\sum_{k=1}^{\log(n)}$ hash operations and $|R_{\min}|$ FHE decryption operations, respectively. Compared with the above operations, we can see that $\rho \cdot |R_{\min}| \cdot |R_{\max}|$ FHE multiplication operations on the server dominate the query computational cost.

For the storage overhead, most of the storage overhead in our proposed scheme comes from the encrypted identifiers in $I = \{I_{A_j}, I_{A_j'}, \ldots, I_{A_j''}\}$, which consumes $\sum_{j=1}^{p}|I_{A_j}| \cdot \log(n)/d$ FHE ciphertexts, where $|I_{A_j}|$ is the number of nodes in $I_{A_j}$ and $d$ is the decomposition parameter in FHE (see next subsection).

\subsection{Experimental Analysis}

Then, we experimentally analyze our proposed scheme. In specific, we implement our proposed scheme in C++ (our code is open source [16]) and conduct experiments on a 64-bit machine with an Intel(R) Core(TM) i5-4300M CPU at 2.66GHz and 4 GB RAM, running Ubuntu 18.4. Note that we implement the data user and cloud server on the same machine, which means there is no network delay between them. The underlying database in our experiment was extracted from Kaggle [17]. It contains 31,087 movies and 153,584 movie tags. The average length of all these movie tags is about 10. We treat each movie as a record with corresponding tags as its attributes.

\subsection{FHE Implementation}

In order to reduce the storage overhead of FHE, we use the SIMD technique [18] in experiments. Before describing our packing method, we first review the underlying structure of FHE. Specifically, the plaintext space of FHE is $\mathcal{P} = F_p[x]/(\Phi_m(x))$ for a positive integer $p \geq 2$ where $\Phi_m(x)$ is the $m$th cyclotomic polynomial. When the plaintext modulus $p$ is prime and not divisible by $m$, the $\Phi_m(x)$ decomposes into $\ell$ irreducible factors $f_1(x), \ldots, f_\ell(x)$ of degree $d$ modulo $p$. This induces an isomorphism, via the Chinese Remainder Theorem, between the algebra of the plaintext space $\mathcal{P} = F_p[x]/(\Phi_m(x))$ and the product of $\ell$ finite fields $L_i \equiv F_{p^{l_i}}$ for $i \in \{1, \ldots, \ell\}$.

With this decomposition, the plaintext of compatible FHE schemes can be regarded as a length $\ell$ vector $m = (m_1, \ldots, m_\ell)$, where $m_i \in F_{p^{l_i}}$ for $i \in \{1, \ldots, \ell\}$. Addition and multiplication on ciphertext $\overline{m}$ correspond to component-wise addition and multiplication over the $m_i$ for $i \in \{1, \ldots, \ell\}$.

Now we give our packing method. In our proposed scheme, each index $I_{A_j}$ for $j \in \{1, \rho\}$ includes a set of identifiers $I_{A_j} = \{id_{j_1}, id_{j_2}, \ldots, id_{j_{|I_{A_j}|}}\}$, where each identifier $id_k$ in it can be seen as a bit-array $\{id_{k,1}, id_{k,2}, \ldots, id_{k, \log(n)}\}$. In the experiments, we pack each index $I_{A_j}$ to $(|I_{A_j}|/\ell) \times \log(n)$ FHE ciphertexts, which means each $id_k$ for $k \in \{1, \ldots, |I_{A_j}|\}$ and $l \in \{1, \ldots, \log(n)\}$ is encoded to an element in $F_{p^l}$ and therefore an FHE ciphertext contains $\ell$ bits in $I_{A_j}$.

\subsection{HElib Parameters}

We utilize the HElib library [19] to implement the FHE described above. For the parameters, we choose $p = 2$ and $m = 18631$, which lead to $\varphi(m) = 18000$, $d = 25$, and $\ell = 720$. Meanwhile, the recent version of the HElib library (commit c74ffab in [19]) uses a concept of ciphertext capacity instead of depth. The ciphertext capacity of a ciphertext is defined in [20] as $\log(Q/\eta)$, where $Q$ is the current modulus, and $\eta$ is the current noise bound. In practice, an EQ algorithm costs about 60-bit capacity and an AND/OR operation costs about 25-bit capacity when $p = 2$ and $m = 18631$. Since $n$ in our experiments does not exceed 25000, $\rho$ does not exceed 5, and the average attribute length $\mu = 10$, we set the initial ciphertext capacity to $60 + \lceil \log(2500 \cdot 10 - 1) \rceil \cdot 25 + \lceil \log(5 - 1) \rceil \cdot 25 = 485$, which leads to 84-bit security.
In the following, we evaluate the computational cost and storage overhead of our proposed scheme in terms of two phases: data outsourcing and compound substring query.

### 6.2.3. Data Outsourcing

First, we consider the computational cost and storage overhead of the data outsourcing phase, which mainly comes from the building runtime and storage overhead of index \( I_{A_1}, I_{A_2}, \ldots, I_{A_{\rho}} \). Figures 6 and 7 plot the building runtime and storage overhead of the data outsourcing versus the number of records \( n \) and the number of attributes \( \rho \). From these figures, we can see that both computational cost and storage overhead increase linearly with \( n \) and \( \rho \).

### 6.2.4. Compound Substring Query

Next, we consider the computational cost of the compound substring query phase. As mentioned in the last subsection, most of the query computational cost is from \( \rho \cdot |R_{\text{min}}| \cdot |R_{\text{max}}| \) FHE multiplication operations on the server, where \( R_{\text{min}} \) and \( R_{\text{max}} \) are the minimum-size and maximum-size elements in \( \{R_1, \ldots, R_{\rho}\} \). As shown in Figures 8 and 9, the query time is indeed linear with \( R_{\text{min}} \cdot R_{\text{max}} \) and \( \rho \). Meanwhile, it is not affected by the number of records \( n \).

### 7. Related Work

A searchable encryption scheme can be realized with optimal security via powerful cryptographic tools, such as fully homomorphic encryption (FHE) [21, 22] and Oblivious
Random Access Memory (ORAM) [23, 24]. However, these tools are extraordinarily impractical. Another set of works utilize property-preserving encryption (PPE) [25–28] to achieve searchable encryption, which encrypts messages in a way that inevitably leaks certain properties of the underlying message. For balancing the leakage and efficiency, many studies focus on searchable symmetric encryption (SSE). Song et al. [3] first used symmetric encryption to facilitate attribute queries over the encrypted data. Then, Curtmola et al. [5] gave a formal definition of SSE and proposed an efficient SSE scheme. Later, Kamara et al. [29] proposed the first dynamic SSE scheme, which uses a deletion array and a homomorphically encrypted pointer technique to securely update files. Unfortunately, due to the use of fully homomorphic encryption, the update efficiency is very low. In a more recent paper [7], Cash et al. described a simple dynamic inverted index based on [5], which utilizes the data unlinkability of the hash table to achieve secure insertion. Meanwhile, to prevent the file-injection attacks [30], many works [31–34] focused on forward security, which ensures that newly updated attributes cannot be related to previous queried results.

Nevertheless, these above works only can support the exact attribute query. If the queried attribute does not match a preset attribute, the query will fail. Fortunately, the fuzzy query can deal with this problem as it can tolerate minor typos and formatting inconsistencies. Li et al. [35] first proposed a fuzzy query scheme, which used an edit distance with a wildcard-based technique to construct fuzzy attribute sets. For instance, the set of CAT with 1 edit distance is \{CAT, *CAT, *AT, C*AT, C*AT, T, CA*T, CA*T, CAT*\}. Then, Kuzu et al. [36] used LSH (Local Sensitive Hash) and Bloom filter to construct a similarity query scheme. Since an honest-but-curious server may only return a fraction of the results, Wang et al. [37] proposed a verifiable fuzzy query scheme that not only supports fuzzy query service but also provides proof to verify whether the server returns all the queried results. However, these fuzzy query schemes only support single fuzzy attribute queries and address problems of minor typos and formatting inconsistency, which cannot be directly used to achieve substring queries.

In [9], Chase and Shen designed an SSE scheme based on the suffix tree to support substring queries. Although this scheme can be used to implement the substring query and allows for substring query in \(O(|s| + d_s)\) time, its storage cost \(O(m)\) has a big constant factor. The reason is that the suffix tree only stores position data in leaf nodes and does not utilize the space of inner nodes effectively. This makes the number of nodes in the suffix tree can be up to \(2m\), where \(m\) is the size of the dataset. In order to reduce the storage cost as much as possible, Leontiadis and Li [13] leveraged Burrows-Wheeler Transform (BWT) to build an auxiliary data structure called a suffix array, which can achieve storage cost \(O(m)\) with a lower constant factor. However, its query time is relatively large. Later, Mainardi et al. [11] optimize the query algorithm in [13] to achieve \(O(|s| + d_s)\) at the cost of higher index space, i.e., \(O(|\Sigma| \cdot m)\), where \(|\Sigma|\) is the number of distinct characters in the dictionary. In addition to suffix tree and suffix array, there are some other auxiliary data structures that can be used to support substring queries. In 2018, Hahn et al. [38] designed an index based on k-grams. When a user needs to perform a substring query, the cloud performs a conjunctive keyword query for all the k-grams of the queried substring. In the same year, Moataz et al. [10] proposed a new substring query scheme based on the idea of letter orthogonalization, which allows testing of string membership by performing an efficient inner product. Although the above schemes can support substring queries, they can only solve the substring query problem for a single attribute, which cannot be used to achieve compound substring queries efficiently.

8. Conclusion

In this paper, we have proposed a novel efficient and privacy-preserving compound substring query scheme. Specifically, based on the position heap technique, we first designed a tree-based index to support a substring query on a single attribute and then applied PRF and FHE techniques to protect its privacy. In addition, based on the homomorphism of fully homomorphic encryption, we designed an algorithm to support compound substring queries on multiple attributes. Detailed security analysis and performance evaluation show that our proposed scheme is indeed privacy-preserving and efficient. In our future work, we will consider extending the proposed scheme to support wildcard queries.

Data Availability

Data will be available at the following link https://www.kaggle.com/rounakbanik/the-movies-dataset.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported in part by NSERC Discovery Grants (Rgpin 04009) and National Natural Science Foundation of China (U1709217) and NSFC Grant (61871331).

References


