Generalized Pattern Matching String Search on Encrypted Data in Cloud Systems

Dongsheng Wang†, Xiaohua Jia‡, Cong Wang‡, Kan Yang§, Shaojing Fu†, Ming Xu‡
†College of Computer, National University of Defense Technology, Changsha, China
‡Department of Computer Science, City University of Hong Kong, Hong Kong SAR
§Department of Electrical and Computer Engineering, University of Waterloo, ON, Canada

Abstract—Searchable encryption is an important and challenging issue. It allows people to search on encrypted data. This is a very useful function when more and more people choose to host their data in the cloud and the cloud server is not fully trustable. Existing solutions for searchable encryption are only limited to some simple functions of search, such as boolean search or similarity search. In this paper, we propose a scheme for Generalized Pattern-matching String-search on Encrypted data (GPSE) in cloud systems. GPSE allows users to specify their search queries by using generalized wildcard-based string patterns (such as SQL-like patterns). It gives users great expressive power in specifying highly targeted search queries. In the framework of GPSE, we particularly implemented two most commonly used pattern matching search functions on encrypted data, the substring matching and the longest-prefix-first matching. We also prove that GPSE is secure under the known-plaintext model. Experiments over real data sets show that GPSE achieves high search accuracy.

I. INTRODUCTION

Cloud storage has become a very popular service in cloud systems nowadays. Many organizations, institutions or companies choose to outsource data services to cloud servers. Data privacy is always the major concern for people to host their data in the cloud. To protect privacy, data are usually encrypted before they are outsourced to the cloud. However, data encryption makes the search on encrypted data extremely difficult. Some searchable encryption (SE) schemes have been proposed to perform search on encrypted data. Existing SE schemes can be classified into two categories, boolean search (e.g. [1]–[4]) and similarity search (e.g. [5]–[8]). Boolean search is to find the exact match between the query strings and the index strings. Similarity search can return index strings that are similar to the query strings. Similarity search can tolerate typos or user’s guesses in the query strings. But, it still requires users to input full strings for search. It cannot support the common search functions such as substring matching, longest-prefix-first (LPF) matching, or string pattern matching.

String pattern matching is an advanced and powerful search function that allows users to use complex matching rules in search queries, such as wildcard character matching and substring matching. It gives users great flexibility to specify their targeted search. However, it is very challenging to design pattern matching string search on encrypted data. A naive solution is to let the user first retrieve all the encrypted indexes from the cloud. The user then decrypts the indexes and conducts the pattern matching search on the plaintext of the indexes. This solution suffers heavy communication cost for transfer of index data between the cloud and the user, and high computation cost at the user side. It cannot be used in cloud systems. There are some works [9]–[15] that use secure multi-party computation to achieve string pattern matching without revealing each party’s own information to others. These methods are not suitable for the cloud systems due to the high computation and communication cost, which is also recognized in [5], [16]. To the best of our knowledge, despite its importance, the issue of pattern matching string search over encrypted cloud data has not been properly investigated and the problem remains open in the literature.

In this paper, we propose a scheme for Generalized Pattern-matching String-search on Encrypted data (GPSE for short) for cloud systems. GPSE allows users to specify their search queries by using generalized wildcard-based string patterns (such as SQL-like patterns). It gives users great expressive power in specifying highly targeted search queries. There are two grand challenges in the design of generalized pattern matching search over encrypted data: 1) Design algorithms to match a query-string that is only part of the index strings and the index strings are encrypted as whole; 2) Design search algorithms for various pattern matching rules over encrypted indexes. In our scheme, we first transform strings (index strings or string patterns) into fingerprint vectors. Each element in a fingerprint vector represents the information of part of a string. We design a weight matrix for each vector according to the matching rule. Then, we multiply the vector with its weight matrix and compare the weighted Euclidean distance between the query vector and the index vectors. For privacy preservation, all the vector-matrix computation and comparison are conducted under two-tier protection: one-way transformation and extended usage of secure kNN computation [17]. We implement GPSE over real data sets and the experiment results show that GPSE achieves high search accuracy with low overhead. We also prove that GPSE is KPA-secure.

The original contributions of this paper can be summarized as follows:

1) We propose a framework for generalized pattern matching string search over encrypted data.

2) We propose the method that constructs fingerprint vectors and uses weight matrix to define the matching rules, so that we are able to achieve search with various matching rules by adjusting the weight matrix.

3) We implement two most commonly used search functions of pattern matching, the substring matching and the longest-prefix-first (LPF) matching over encrypted data.

The rest of this paper is organized as follows: Section II introduces the formalization of the problem and necessary preliminaries. Section III presents the method that converts the process of pattern matching into vector-matrix computation. In section IV, we describe the details of GPSE. Security
analysis and performance evaluation are presented in Section V and Section VI respectively. We summarize related works in Section VII and conclude the paper in Section VIII.

II. PROBLEM FORMULATION

A. System and Threat Model

In this paper, we consider a cloud storage system supporting secure information retrieval. As shown in Fig. 1, there are three entities: data owner, cloud server and data user. We simply call them the owner, the server and the user respectively in the rest of this paper. Before outsourcing data (documental files or structural database) to the server, the owner needs to encrypt the data, construct secure indexes, and upload them to the cloud. It also generates secret keys and distributes them to authorized users. When the user wants to retrieve data from the cloud, it generates a secure trapdoor from a query string pattern and submits it to the server. The server will perform a pattern matching search over the encrypted data and return the matched indexes (or data) to the user.

In this paper, we consider a honest-but-curious cloud [2], [3], [18] in our threat model. That is, the server honestly conducts designated operations, but it is curious to learn useful information from the outsourced data or protocol messages. Our goal is to design privacy-preserving generalized pattern matching string search with high search accuracy. In section IV, we will formalize the privacy threats according to the background knowledge acquired by the server.

B. Preliminaries

In the following, we present some basic terminologies and algorithms that are adopted in GPSE.

String Pattern. A sequence of characters and predefined wildcards. A string pattern is used to represent a subset of the string universe and it is usually generalized into a regular expression. In a string pattern, the subsequences separated by wildcards are called basic substrings. For example, the string universe and it is usually generalized into a regular

n-Gram Counting Order. A vector generated from a string using a hash function [19]. An n-gram is a contiguous sequence of n characters from a string. For example, the word “cloud” has six 2-grams: \{#c, cl, lo, ou, ud, d#\}. \# is a position indicator and we do not consider it as a basic wildcard throughout this paper. Given a string \textit{str} with length \textit{l}_{str}, it has \((l_{str} + n - 1)\) n-grams. All the n-grams of \textit{str} constitute an n-gram list \(L_{n_{str}}^{str}\) in the left-to-right order. Let \(h_{hk}()\) be a cryptographic hash function with the secret key \(hk\) and \(V_{gco}^{n}\) be the n-gram counting order of \textit{str}. The \(i\)-th element of \(V_{gco}^{n}\) is generated as

\[
V_{gco}^{n}(i) = \sum_{g \in L_{gco}^{n}_{str}} is(h_{hk}(g) \mod N = i),
\]

where \(g\) is an n-gram in \(L_{str}^{n}\), \(is()\) is a truth function that only outputs 0 or 1, and \(N\) is the size of \(V_{gco}^{n}\).

In plaintext search scenario, n-gram counting order is used to efficiently evaluate the similarity between two strings based on their n-gram counting information [19]. The hash function used in plaintext design does not have to be secret. In this paper, for security consideration, we introduce the secret hash function \(h_{hk}()\) in Eq. (1) to construct secure n-gram counting order which is one-way transformation against the adversaries. In the following, we denote by \(GCO(L_{str}^{n}, N, hh)\) the algorithm to generate secure n-gram counting order of \textit{str}.

III. VECTOR-MATRIX-BASED PATTERN MATCHING STRING SEARCH

The process of string pattern matching can be transformed into other computation forms such as convolution with Fast Fourier Transform [14] and linear matrix manipulation [15]. However, they cannot be directly conducted on pre-encrypted cloud data. In this section, we introduce a method that converts generalized pattern matching string search into vector-matrix computation and our conversion can be conducted on encrypted data through extended usage of secure kNN computation. Here we first present the main idea of the conversion method. Then we give the details of two key algorithms used in it.

A. Main Idea

Given a string \textit{str}, it can be described by its n-grams and their positions in the string, i.e., \(str \Leftrightarrow \{<g, i> | g \in L_{n_{str}}^{str}, L_{str}^{n}(i) = g\}\), where \(L_{str}^{n}\) is the n-gram list of \textit{str}. Each pair of \(<g, i>\) represents part of the fingerprint of \textit{str} [19] [20]. String search based on pattern matching is to compare the fingerprint of the string pattern against that of the indexes [21]. In our method, we first transform a string (index string or string pattern) into a fingerprint vector. All the fingerprint vectors are of the same size. Each element in the vector represents part of the fingerprint (i.e., \(<g, i>\)) of the string and the identical n-grams (i.e., \(g\)) will be mapped to the same position in the vector. Given an index string \(si\) and a string pattern \(sp\), their fingerprint vectors are denoted by \(V_{si}\) and \(V_{sp}\) respectively. To evaluate the matching degree between \(si\) and \(sp\), we introduce a metric called weighted Euclidean distance that can compare the fingerprint of two strings. Let \(H_{1}\) and \(H_{2}\) be two diagonal matrixes, that is, \(H_{k} = diag(h_{k1}, h_{k2}, ..., h_{kd})\), \(k = 1, 2\). The weighted Euclidean distance between \(si\) and \(sp\) is defined as:

\[
\Omega(si, sp) = -dis(H_{1} \cdot V_{si} , H_{2} \cdot V_{sp})^{2}
= -\sum_{i=1}^{d} (h_{1i}V_{si}(i) - h_{2i}V_{sp}(i))^{2},
\]

where \(d\) is the size of fingerprint vectors and \(dis()\) is to calculate the Euclidean distance between two vectors. Through careful design of \(H_{1}\) and \(H_{2}\), we can specify which elements in the fingerprint vectors need to be compared with, and how much weight they have in the value of \(\Omega(si, sp)\). In such a
way, we can check whether an index string matches a string pattern in a fine-grained manner. In the following, we will present the details of two algorithms that generate fingerprint vectors \( V_{si} \) and \( V_{sp} \), and weight matrices \( H_1 \) and \( H_2 \): 1) Fingerprint vector extraction \( FVE(\cdot) \) that generates fingerprint vectors; 2) Vector-matrix based (VM) string pattern definition \( SPD(\cdot) \) that generates matrixes \( H_1 \) and \( H_2 \) according to the string pattern.

**B. Fingerprint Vector Extraction \( FVE(\cdot) \)**

The algorithm \( FVE(\cdot) \) takes an input string \( str \) and outputs a fingerprint vector \( V \) of size \( d \), i.e., \( V = FVE(str) \). \( str \) can be an index string or a string pattern. \( V \) is generated from n-grams of \( str \). For string search that uses grams, the gram length dictates the accuracy of search [20]. The longer of gram length is, the more accurate of the search will be, but incurring high complexity of search. It has been observed that the gram length of 3 is accurate enough for string search and usually it is selected as 2 for efficiency [19] [20]. In our method, we only consider gram length up to 3. For input string \( str \), we first generate 1-gram list, 2-gram list, and 3-gram list, denoted by \( L_{1str} \), \( L_{2str} \) and \( L_{3str} \), respectively. Note that \( str \) may be a string pattern and it may contain wildcards. In string pattern, wildcards are not considered as normal semantic characters of strings. Thus, if an n-gram (\( n = 1, 2, 3 \)) contains wildcards, it is deleted from \( L_{str} \). \( V \) is made of:

\[
V = (V^1, V^2, V^3, l_{str})
\]

where \( V^1, V^2 \) and \( V^3 \) are three vectors generated from \( L_{1str} \), \( L_{2str} \) and \( L_{3str} \), respectively and \( l_{str} \) is the length of \( str \). \( V^1, V^2 \) and \( V^3 \) are generated as below.

**Generation of \( V^n \) (\( n = 1, 2, 3 \)).** For an n-gram list \( L^n_{str} \), \( V^n \) contains both the counting information and position information of n-grams in it. Specifically, \( V^n \) is the sum of two vectors, i.e., \( V^n = V^n_{cnt} + V^n_{pos} \), where \( V^n_{cnt} \) and \( V^n_{pos} \) represent the counting information and position information of n-grams in \( L^n_{str} \), respectively. Let \( d_n \) denote the size of \( V^n \). For \( i = 1, 2, \ldots, d_n \), we define:

\[
V^n_{cnt}(i) = \begin{cases} \sqrt{V^n_{gco}(i)}, & \text{if } V^n_{gco}(i) \leq CT \\ \sqrt{CT}, & \text{otherwise.} \end{cases}
\]

\[
V^n_{pos}(i) = \begin{cases} \mu^{pos}(i), & \text{if } V^n_{gco}(i) > 0 \\ 0, & \text{otherwise.} \end{cases}
\]

In the above equations, \( V^n_{gco} = GCO(L^n_{str}, d_n, h_k) \), which is defined in Equation (1). \( CT \) is a gram counting threshold, which is introduced to cap the value of elements in \( V^n_{gco} \). \( pos(i) \) is the position of an n-gram in \( L^n_{str} \), defined as:

\[
pos(i) = \min_{j \in N^*} \left\{ j \left| L^n_{str}(j) \xrightarrow{GCO(\cdot)} V^n_{gco}(i) \right. \right\}.
\]

It can be seen that \( pos(i) \) is the first position of the n-gram in \( L^n_{str} \) that is hashed into \( V^n_{gco}(i) \). \( \mu \) is a number that is greater than \( \sqrt{CT} \). By adjusting value \( \mu \), we can adjust the weight of position information, i.e., \( V^n_{pos} \) in vector \( V^n \), which can help reducing the false negative in the search.

Now we obtain the fingerprint vector \( V \). Note that the function \( GCO(\cdot) \) always maps the identical n-grams to the same position in \( V \). To ensure the security of \( V \), we do some permutations of elements of \( V \) as the following.

**Element Permutation.** Let \( \pi \) be a strong pseudo-random permutation (SPRP) and \( pk \leftarrow \{0, 1\}^k \) be a secret permutation key.

\[
\pi : \{0, 1\}^k \times \{0, 1\}^{\log_2(d)} \rightarrow \{0, 1\}^{\log_2(d)}.
\]

All the positions of elements in \( V \) are realigned as \( V'(\pi(pk)(i)) \leftarrow V(i) \). Finally, we assign \( V' \) back to \( V \): \( V \leftarrow V' \).

Based on the secret hash function and secret SPRP function, \( FVE(\cdot) \) is a secure one-way transformation function for the strings.

**C. Vector-matrix-based String Pattern Definition \( SPD(\cdot) \)**

Given the string pattern \( sp \), we use the function \( FVE(sp) \) to generate its fingerprint vector \( V_{sp} = FVE(sp) \). The string pattern \( sp \) may contain wildcards. For each kind of wildcards contained in \( sp \), the algorithm \( SPD(\cdot) \) produces a 3-tuple \( \langle V_{sp}, H_1, H_2 \rangle \). We now discuss the generation of the two weight matrixes \( H_1 \) and \( H_2 \). There are four commonly-used wildcards in the expression of generalized search patterns, i.e., "*", "?", "[ ]" and "\*". Due to space limitation, we only present the solution for the most commonly-used wildcard: "*" (representing any number of characters). The other three wildcards "?", "[ ]" and "\*" can be easily dealt with in the similar way and we leave them in future work.

We consider three basic types of string patterns using "*":

- \( \%bstr\% \) (substring matching), \( bstr\% \) (prefix matching) and \( \%bstr\% \) (suffix matching), where \( bstr \) is the basic substring in the pattern. We discuss the generation of \( H_1 \) and \( H_2 \) for each type of string patterns using "*" as follows:

1) **VM Pattern definition of Substring Matching.**

Given a string pattern \( sp \) for substring matching, i.e., \( \%bstr\% \), the length of a matched index string for \( sp \) is not known and it varies for different index strings. Thus we do not consider the length of strings during matching. We simply set \( l_{sp} \) to 0 in \( V_{sp} \). Note that all the n-grams of \( sp \) are mapped into the nonzero elements of \( V_{sp} \). Denote the positions of nonzero elements in \( V_{sp} \) by a set \( N_e = \{ n_1, n_2, \ldots, n_c \} \), where \( n_i \in N_e \) (\( i = 1, 2, \ldots, c \)). \( V_{sp}(n_i) \neq 0 \). For substring matching, we check whether the n-grams of \( sp \) are contained in that of the index string. Specifically, we can just look at the elements that appear in the positions of \( N_e \) for both the index string and \( sp \). Therefore, \( H_1 \) and \( H_2 \) can be designed as (\( i = 1, 2, \ldots, d \)):

\[
h_{1i} = \begin{cases} \omega, & \text{if } i \in N_e \\ 0, & \text{otherwise} \end{cases}
\]

\[
h_{2i} = \beta \cdot h_{1i},
\]

where \( \omega \) (\( \omega > 1 \)) is a weight parameter that is used to select a certain nonzero element in the fingerprint vector. \( \beta \) is the average value of all the non-zero elements in all fingerprint vectors, which is used to reduce the false negative. The design of \( H_2 \) will be substantially different from \( H_1 \) for other wildcards.

Now we study the weighted Euclidean distance for evaluating substring matching. Given an index string \( si \), \( V_{si} = FVE(si) \). As defined, their weighted Euclidean distance is:

\[
\Omega(si, sp) = -\sum_{n_i \in N_e} \omega^2 [V_{si}(n_i) - \beta V_{sp}(n_i)]^2.
\]

Note that all the n-grams of \( sp \) are mapped to the positions in \( N_e \) and the identical n-grams will be mapped to the same position in their fingerprint vectors. Intuitively, if \( si \) has more
common n-grams with sp, Ω(si, sp) will have a greater value. In other words, for a greater Ω(si, sp), the probability that si contains sp is higher. Therefore, the substring matching can be achieved.

2) VM Pattern definition of Prefix Matching. Given a string pattern sp for prefix matching, i.e., bstr%, we also set the element mapped from l.sp to 0. Since bstr must appear in the head of a matched index string, the position information of n-grams is critical. Specifically, if an n-gram g1 precedes g2 in L.sp, we should consider g1 in priority. Therefore, the weight of g1 should be greater than that of g2 when calculating the weighted Euclidean distance. Here we assign the n-grams of sp with weight values in exponentially decreasing form according to the n-gram positions in L.sp. Let l.n be number of n-grams in L.sp, Nc is the position set of nonzero elements in V.sp. H1 and H2 are designed as (i = 1, 2, ..., d),

\[ h_{1i} = \begin{cases} 
\mu^{l.n(i) - \text{pos}(i) - n(i) + 2}, & \text{if } i \in N_c \text{ and } n(i) > 1 \\
0, & \text{otherwise},
\end{cases} \]

where \( \mu \) is a number greater than \( \sqrt{CT} \). If an n-gram is mapped to \( V_{sp}(i), n(i) = n, \text{ i.e., } n(i) \in \{1, 2, 3\} \), pos(i) is a position number in L.sp. Considering the i-th element in \( V_{sp}(i) \) may be mapped from different n-grams, we set pos(i) as the first position in \( L_{sp} \) that are mapped to \( V_{sp}(i) \):

\[ \text{pos}(i) = \min_{j \in N^+} \left\{ j \mid L_{sp}(j) \xrightarrow{\text{FVE}} V_{sp}(i) \right\}. \]

Since the weight values of n-grams are in exponentially decreasing form, the n-grams appearing at the front in the list \( L_{str}^{n} \) will be given greater weight. Thus they will be considered in higher priority when evaluating the weighted Euclidean distance. Meanwhile, if an index string si has longer common prefix with sp, the matching degree Ω(si, sp) will be greater. This is because the fingerprint vector \( V_{si} \) will have more nonzero elements that appear in the positions of \( N_c \). In such a way, the matched indexes will be ranked according to how many common prefixal n-grams they have with sp in the search results. We claim that the definition of bstr% can achieve not only the prefix matching but also another commonly used search types: LPF matching. The matched indexes of prefix matching are just those that have the longest prefix against sp in the search results of LPF matching. Our experimental results over real data sets also show the high search accuracy of LPF matching under such definition.

3) VM Pattern definition of Suffix Matching. The definition of %bstr is simply a reverse case to that of bstr%. The n-grams appearing at the back in L.sp will be given greater weight values. We omit the details here.

D. Accuracy Analysis on Our Method

In our method, the indexes will be returned according to the matching degree in a top-k ranked manner. For a top-k ranked search scheme, the search accuracy can be analyzed by two parameters: false positive and false negative [21]. The two parameters both indicate the error rate of the search results. Given a string pattern sp with a basic substring whose length is m, si (length l) is an index string to be matched with. Let \( \Sigma \) be the alphabet and an n-gram belongs to \( \Sigma^n \).

1) False positive: If si does not match sp but our method reports that si matches sp, i.e., \( \Omega(si, sp) = 0 \), then a false positive happens. A false positive is mainly caused by the hash collisions in \( FVE(\cdot) \). That is, an n-gram \( g \in L^n_{si}, g \notin L^n_{sp} \) may be hashed to the position in \( N_c \) of V.sp. We take the substring matching for example and ignore the position component since it is much smaller than the counting component. The false positive rate \( fp \) of substring matching is:

\[
fp = \prod_{n=1}^{3} \sum_{k=1}^{t} C_{s}^{t} \cdot t! \cdot (d_n - k)^{-t} P(k) / d_n^t,
\]

where \( s = l + n - 1, t = m - n + 1 \) and \( P(k) \) is the probability that \( t \) n-grams of sp are hashed into \( k \) different positions in V.sp. Here we suppose all the n-grams are independent since the probability that two arbitrary n-grams are the same ones is \( 1/|\Sigma|^m \). According to Equation (2), \( fp \) is smaller when \( d_n \) or \( m \) is larger. That is because larger \( d_n \) will cause less hash collisions and larger \( m \) requires that si has more n-grams mapped to certain positions in the fingerprint vector.

2) False negative: If si matches sp but our method reports that si does not match sp, i.e., \( \Omega(si, sp) \neq 0 \), then a false negative happens. A false negative may happen when more than certain number of n-grams (\( \in L^n_{sp} \)) are hashed to the same position in \( N_c \) of sp. Even though there are no hash collisions, the false negative still happens because si may have multiple copies of a certain n-gram that also appears in L.sp. In our method, we introduce \( CT \) to restrict the false negatives. Take substring matching for example, the false negative rate fn of our method is:

\[
fn = 1 - \prod_{n=1}^{3} \sum_{k=1}^{t} (d_n - k)^{s-t} P(k) / d_n^t,
\]

From Equation (3), we can also see that larger \( d_n \) or \( t \) can reduce the false negative. Therefore, we claim that a longer string pattern has higher search accuracy since the corresponding \( fp \) and \( fn \) are both smaller. For position-sensitive string patterns (such as LPF matching), the position component in fingerprint vector can be used to distinguish the multiple copies of a certain n-gram that appears in si. Therefore the false negative will be reduced to some extent.

IV. THE FRAMEWORK OF GPSE

Our proposed GPSE can conduct the vector-matrix-based pattern matching string search under two-tier protection: one-way transformation and extended usage of secure kNN computation. In this section, we present the framework of GPSE. GPSE has three phases: 1) System setup phase which is conducted by the owner to generate encrypted data sets that will be outsourced to the cloud; 2) Trapdoor generation phase where the user generates query trapdoors and initiates search sessions with the server. 3) Search phase which is performed by the server to search for the expected data sets and return them to the user. The details are as follows:

1) System Setup Phase: Let D be the original data set collected by the owner, \( W = \{w_1, w_2, ..., w_N\} \) is an index string set extracted from D. \( w_i \) could be a keyword or the attribute value in the column of structural database. For the purpose of privacy preservation, the owner first runs the function KeyGen(1^n) to generate secret keys.

KeyGen(1^n): The key generation algorithm takes as input
a security parameter $\kappa$ and outputs a set of secret keys $SK = \{M_1, M_2, K_{spar}, h_k, pk, sskk\}$, where $M_1$ and $M_2$ are two $(d + 2) \times (d + 2)$ invertible transformation matrices, $K_{spar}$ is a binary vector of size $(d + 2)$, $h_k$ is a secret hash key, and $sskk$ is a symmetric secret key to encrypt the original data set $D$.

Then the owner generates searchable secure indexes $I = \{I_1, I_2, ..., I_N\}$ through the algorithm $\text{IndexBuild}()$.

$\text{IndexBuild}()$: The secure index construction algorithm takes as input $W$ and $SK$ to generate the set of secure indexes $I$. For each $w_i \in W$, the algorithm first calls $FV(E(w_i, h_k, pk))$ to generate a fingerprint vector $\vec{w}_i$. $\vec{w}_i$ is then extended into $\vec{w}_i = (\frac{w_i}{2}, -0.5||\vec{w}_i||^2, 1)$. Then $\vec{w}_i$ is further split into two vectors $\{\vec{w}_i', \vec{w}_i''\}$ in such a way that $K_{spar}(j) = 1$ (j = 1, 2, ..., d + 2), $\vec{w}_i(j)$, $\vec{w}_i''(j)$ are randomly generated so that $\vec{w}_i(j) + \vec{w}_i''(j) = \vec{w}_i(j)$. Finally, we get $I_i = \{I_i', I_i''\} = \{M_1^T \vec{w}_i', M_2^T \vec{w}_i''\}$.

After that, the owner encrypts $D$ as $C = \{C_i | C_i = \text{Enc}(D_i, sskk)\}$, where $\text{Enc}()$ is a standard symmetric encryption algorithm such as AES. In the end, the owner outsources $C$ with secret indexes $I$ to the cloud server and distributes secret keys $SK$ to authorized data users.

2) Trapdoor Generation Phase: In order to retrieve expected data sets, the user first selects a string pattern $sp$ and runs the algorithm $\text{TrapdoorGen}(\cdot)$ to generate a query trapdoor.

$\text{TrapdoorGen}(\cdot)$: The query trapdoor generation algorithm calls $SPDsp$ to generate a 3-tuple $Q = (\vec{v}_p, H_x, H_y)$ to define $sp$, where $\vec{v}_p$ is the fingerprint vector of $sp$, $H_x$ and $H_y$ are two weight matrices. The algorithm computes $\vec{v}_p = (H_x)^T H_y \vec{v}_p$ and generates two random numbers: $R > 0$, $r > 0$. $\vec{v}_p$ and $H_n = (n = 1, 2)$ are extended as $\vec{v}_p = (R\vec{v}_p', \vec{v}_p, r)$. After that, $\vec{v}_p$ is split into $\{\vec{v}_p', \vec{v}_p''\}$ in such a way that if $K_{spar}(j) = 1$ (j = 1, 2, ..., d + 2), $\vec{v}_p(j) = \vec{v}_p''(j) = \vec{v}_p(j)$; if $K_{spar}(j) = 0$, $\vec{v}_p'(j)$ and $\vec{v}_p''(j)$ are randomly generated so that $\vec{v}_p'(j) + \vec{v}_p''(j) = \vec{v}_p(j)$. Then the algorithm calculates $\frac{1}{2} (H_1^T H_1(\vec{v}_p'))^2$ to the inner product of two vectors. We extend it to preserve the value of vector-matrix multiplication. Let $H_x(j) = 1, 2, ..., d + 2$ be the j-th column of $H_x$, $H_y(j)$ is also split into two parts $\{H_x(j)', H_x(j)''\}$ under $K_{spar}$ just like $v_p$. Thus the algorithm generates two matrices $\{H_x, H''_x\}$, where $H_x = \{H_x(1)', H_x(2)', ..., H_x(d + 2)\}$ and $H''_x = \{H_x(1)', H_x(2)', ..., H_x(d + 2)\}$ is generated similarly. The same way, the algorithm generates $\{H_y, H''_y\}$ by splitting $H_y$. The correctness of our extension on secure KNN computation is designed to be verified and we omit the details here. All the split vectors and matrices are further transformed using $M_1$ and $M_2$:

$$
\vec{v}_p' = M_1^{-1} \vec{v}_p', \quad \vec{v}_p'' = M_1^{-1} \vec{v}_p'';
H_x' = M_1^{-1} H_x', \quad H_x'' = M_1^{-1} H_x'';
H_y' = M_1^{-1} H_y', \quad H_y'' = M_1^{-1} H_y''.
$$

Finally, $\text{TrapdoorGen}(sp)$ outputs the query trapdoor:

$$
Q_{sp}^T = \{\{\vec{v}_p', \vec{v}_p''\}, \{H_x', H_x''\}, \{H_y', H_y''\}\}.
$$

The user then submits $Q_{sp}^T$ to the server and initiates a search session.

3) Search Phase: After receiving $Q_{sp}^T$ from the user, the server runs the search algorithm $\text{Search}(\cdot)$ to find out the indexes that have the highest matching degree against $sp$ defined by $Q_{sp}^T$.

$\text{Search}(\cdot)$: The search algorithm scans each secure index $I_i \in I$ and calculates the matching degree between $I_i$ and $Q_{sp}^T$ as:

$$
\tilde{\Omega}(I_i, Q_{sp}^T) = \left( I_i^T \vec{w}_i + (I_i'')^T \vec{v}_p'' - (H_x I_i' + H_x'' I_i'' )^T (H_y I_i' + H_y'' I_i'') \right)
$$

$$
\tilde{\Omega}(I_i, Q_{sp}^T) = \left( \vec{w}_i^T \vec{v}_p + \vec{w}_i''^T \vec{v}_p'' - (H_x \vec{w}_i + H_x'' \vec{w}_i'')^T (H_y \vec{w}_i + H_y'' \vec{w}_i'') \right)
$$

$$
R(\vec{w}_i) T (H_x T H_x' H_y T H_y' \vec{w}_i) + r - (H_x \vec{w}_i) T (H_y \vec{w}_i) = R(H_1 \vec{w}_i) T (H_2 \vec{v}_p) + r - \frac{Rr^2}{2}.
$$

Then it compares the matching degree of $I_i$ against $Q_{sp}^T$ to that of $I_j$:

$$
\tilde{\Omega}(I_i, Q_{sp}^T) - \tilde{\Omega}(I_j, Q_{sp}^T)
$$

$$
= -\frac{R}{2} \left( \frac{2}{R} \left( \frac{2}{H_x \vec{w}_i} T (H_2 \vec{v}_p) - \frac{2}{R} \left( \frac{2}{H_x \vec{w}_i} T (H_2 \vec{v}_p) + \frac{2}{H_2 \vec{v}_p} \right) \right) \right)
$$

$$
= -\frac{R}{2} \left( \frac{2}{R} \left( \frac{2}{H_x \vec{w}_i} T (H_2 \vec{v}_p) + \frac{2}{H_2 \vec{v}_p} \right) \right)
$$

$$
= -\frac{R}{2} \left( \frac{2}{R} \left( H_x \vec{w}_i T (H_2 \vec{v}_p) - \frac{2}{R} \left( \frac{2}{H_x \vec{w}_i} T (H_2 \vec{v}_p) + \frac{2}{H_2 \vec{v}_p} \right) \right) \right)
$$

$\tilde{\Omega}(w_i, sp) = \Omega(w_j, sp).
$$

After that, the algorithm selects the top-k secure indexes that have the $k$ biggest matching degree against $Q_{sp}^T$ and gathers them into a result set $I_R$. Finally, the server returns the user with a set of items (data files or rows in a structural database) that are associated with indexes in $I_R$.

V. Security Analysis

In [18], Curtmola et al. formalized a rigorous security definition for searchable encryption schemes. In this section, we first provide our simulation-based framework to analyze the security strength of GPSE. In GPSE, a string is mapped to encrypted vectors through one-way transformation and extended secure KNN computation. We first prove that such cryptography mapping is secure under known-plaintext model. Then we construct the simulation-based framework to prove that GPSE is secure when conducted in the information retrieval scenario.

In most cases, attackers only have the scope of ciphertext in an SE scheme [8]. This is corresponding to ciphertext-only attack. However, we still investigate the security of GPSE under stronger known plaintext model where the attacker may obtain certain tuples of plaintext and corresponding ciphertext through special tunnel. As remarked in [17], KPA is rare in practice since it is not easy for someone who does not have the secret keys to associate the plaintext with their encrypted values. Under known plaintext model, the knowledge of an attacker
can be formalized as \( A = \{C, I, T, <W, I>, <P, T>\} \) where \( C, I, T \), collected by the attacker, are the encrypted data sets, secure indexes and query trapdoors respectively. \( W \) and \( P \) are the sets of plaintext index strings and string patterns respectively. \( I = \text{IndexBuild}(W) \) and \( T = \text{TrapdoorGen}(P) \).

The security strength of GPSE is based on two-tier cryptography mapping: one-way transformation and extended secure kNN computation. Given a string \( str \), the mapping is:

\[
\text{str} \xrightarrow{\pi_{hk}\circ h_{vk}} \text{V}_{\text{str}} \xrightarrow{\text{kNN}(K_{spt}, M_1, M_2)} \{\hat{\text{V}}_{\text{str}}\}.
\]

Intuitively, if the attacker can recover \( \text{str} \) from the encrypted vectors \( \{\hat{\text{V}}_{\text{str}}\} \) within probabilistic polynomial time (P.P.T.), there will be a P.P.T simulator that can break the secret one-way transformation of \( \pi_{hk} \circ h_{vk} \) through simulating the function \( \text{kNN}(K_{spt}, M_1, M_2) \). This is contradictory with the one-way property of hash functions and the pseudorandomness of SPRP functions.

**Theorem 1.** For the two-tier cryptography mapping in GPSE, there is no P.P.T. adversary that can break the data confidentiality of secret keys, index strings and string patterns under known plaintext model.

**proof:** We just have to prove that the probability for the P.P.T. adversary to break data confidentiality is negligible. Let \( str \) be a string (index string or string pattern) and \( N_{e} \) be the position set of nonzero elements in \( FVE(\text{str}) \). Suppose \( d \) is the size of the fingerprint vector and \( \nu \) is the number of possible values for the element in fingerprint vectors. Given \( str \), without the hash key \( hk \) and permutation key \( pk \), the possibility for the adversary to guess out the same vector as \( FVE(\text{str}) \) is \( \varepsilon \) where \( \varepsilon \leq \left( \frac{C_d}{N_{e}}\right)^{N_{e}} \). To break the data confidentiality, the adversary has to analyze its knowledge \( A \) using the three involved algorithms \( \text{IndexBuild} \), \( \text{TrapdoorGen} \), \( \text{Search} \) in GPSE. Here we study \( \text{Search} \) for example and only list the analysis results of the other two algorithms since the analysis is easy to conduct accordingly.

1) **Analyze \( \text{Search} \).** Observing the correlation between the input and output of \( \text{Search} \), the adversary can simulate \( \text{Search} \) over \( \{I_i < P, T > \} \subset A \). Given the secure index \( I_i \in I/V, I_i = \{I'_i, I''_i\} \) and its plaintext index string is denoted by \( w_i \). Let \( p_j \in P \) be a known string pattern. The adversary can set up two equations as \( (j = 1, 2, ..., |T|) \):

\[
\langle I'_i, I''_i \rangle^T \tilde{v}_{p_j} = R_j \langle \tilde{w}_i \rangle^T \cdot (H_1)^T H_2 \tilde{v}_{p_j} + r_j,
\]

\[
(\tilde{H}_x \langle I'_i + \tilde{H}_x \rangle_{I''_i})^T (\tilde{H}_y \langle I'_i + \tilde{H}_y \rangle_{I''_i}) = \frac{R_j}{2} (H_1 \tilde{w}_i)^T (H_1 \tilde{w}_i) + \frac{r_j^2}{2},
\]

where \( T_j = \{\tilde{v}_{p_j}, \langle I'_i, I''_i \rangle, \tilde{H}_x, \langle I'_i, I''_i \rangle, \tilde{H}_y \} \). The attacker first guesses the fingerprint vector of \( p_j \) with the probability of \( \varepsilon \). Then he can set up \( 2|T| \) equations with \( d + 2|T| \) unknown variables. Since \( \tilde{w}_i \) has \( d \) unknown variables, he has to construct at least \( d \) equations with the probability of \( \varepsilon^{d/2} \). Therefore, the attacker can not solve the equations and the probability to set up enough exact equations is negligible (suppose \( d \log d > 80 \)). Also the adversary can simulate \( \text{Search} \) over \( \{<W, I>, T_j\} \) in a similar way. He can construct \( 2|W| \) equations with at least \( 2d \) unknown variables. Therefore, the possibility for the attacker to set up enough equations to break data confidentiality is no more than \( \varepsilon^d \).

2) **Analyze \( \text{IndexBuild} \) over \( \{I, <W, I>\} \), the adversary can set up \( 2d|W| \) equations that contain \( 2d|W| + 2d^2 \) unknown variables with the probability of \( \varepsilon^{|W|} \).

3) **Analyze \( \text{TrapdoorGen} \) over \( \{<P, T, >\} \), the adversary can set up \( 2d|P| \) equations that contain \( 2d|P| + 2d^2 \) unknown variables with the probability of \( \varepsilon^{|P|} \).

Therefore, through analyzing all the involved mapping algorithms in GPSE, there is no P.P.T. adversary that can break the confidentiality of plaintext strings or secret keys within non-negligible probability.

**Discussion:** The security of GPSE benefits from its two-tier security architecture: one-way transformation and extended secure kNN computation. Recently, the secure kNN computation [17] is reported to be susceptible under known plaintext model [22] [23]. In [22], the authors conducted their method by constructing equations just in the same way that we analyze \( \text{search} \) over \( \{I_i < P, T >\} \). As proved in Theorem 1, GPSE is secure in such case since the fingerprint vectors corresponding to \( P \) are concealed from the attacker by the secret keys: \( \{hk, pk\} \) and thus the attacker can not solve the equations in non-negligible probability.

In information retrieval scenario, the access pattern and search pattern can be inherently build up by the cloud server in the interaction with users [18]. Therefore, GPSE should guarantee that nothing beyond such patterns can be learned by attackers. We will follow the simulation-based framework of [18] to investigate the security of GPSE.

**Theorem 2.** GPSE is secure under known plaintext model.

Before presenting the proof details, we introduce some notions used in [18].

- **History:** Suppose \( D \) is an original data set. A \( k \)-query history over \( D \) is a tuple \( \mathbb{H}_k = (D, Q) \), where \( Q \) is a vector of \( k \) string patterns, i.e., \( Q = \{q_1, q_2, ..., q_k\} \). \( \mathbb{H}_k \) is the plaintext knowledge of the attacker. Under known plaintext model, \( D \) and \( Q \) are not arbitrarily chosen by the attacker. Yet they are randomly sampled from a natural plaintext oracle and given to the attacker beforehand.

- **View:** Given a history \( \mathbb{H}_k \), the trace \( TR(\mathbb{H}_k) \) is the information that the scheme may leak about the history. It contains the size of data item \( \{D_1, |D_2|, ..., |D_k|\} \), the access pattern and the search pattern. The access pattern \( \alpha(\mathbb{H}_k) \) is the search results that can be derived when querying \( Q \) upon \( D \). Considering GPSE is a top-k ranked SE scheme, the relevancy between \( D_i \) and \( q_j \) is comparable. In particular, the attacker can meaningfully estimate the matching degree between \( q_j \) and index strings in \( D_i \). Here we give a stronger assumption that the attacker can estimate the attributes (such as the order or even the precise value) of all the intermediate variables which will be leaked out in search phase. Let \( W \) be the set of the index strings extracted from \( D \). All the intermediate variables can be represented by a \( |W| \times (2d + 1)k \) matrix \( \delta \), where \( \delta(i,j) = \langle R_j \tilde{w}_i H_1 H_2 \tilde{q}_j + r_j, \tilde{w}_i H_2 \rangle (\tilde{w}_i \in W) \) is a block sub-matrix in \( \delta \). Based on the estimation of matching degree, the attacker can construct a set \( I(q_j) \) containing the top-k index strings for \( q_j \). Let \( D(q_j) \) be the set of identifiers of all the data items that contain the index strings in \( I(q_j) \). The access pattern \( \alpha(\mathbb{H}_k) = \{D(q_1), D(q_2), ..., D(q_k), \delta\} \). The
and extended weighted matrixes, the rank of it is no more
rows in
Since
\(X\) s.t. \(XY = \delta\) where \(X\) is a \(|W| \times d\) matrix and \(Y\) is a \(d \times (2d + 1)|k|\) matrix. It can set up \((2d + 1)|W|\) equations with \(|W|d + (2d + 1)|k|\) unknown variables. If \(d > |W|\) or \(d > (2d + 1)|k|\), it can always find a solution for the equations. If \(d < \min\{|W|, (2d + 1)|k|\}\), \(S\) can still construct the solution.

Since \(\delta\) is the multiplication using extended fingerprint vectors and extended weighted matrices, the rank of it is no more than \(d\). Without loss of generality, suppose the first \(rank(\delta)\) rows in \(\delta\) are linearly independent vectors. Let \(\delta(i,:)\) be the \(i\)-th row vector in \(\delta\) and \(\delta(i,:)=j\) be a sub-matrix where \(\delta(i,:)=j\) \(=\delta(i,:),\delta(i+1,:),...,\delta(j,:)^T\). Since \(d \geq rank(\delta)\), the vectors in \(\delta(d,:)=W\) can be the linear combination of vectors in \(\delta(1:d)\). Thus there is a \(|[W] - d|\times d|\) matrix \(L\) s.t. \(\delta(d,:)=W=L\delta(1:d)\). Denote the unit matrix by \(E\). Let \(X = (E,L)^T\), \(Y = \delta(1:d)\), then \(XY = \delta\). \(S\) randomly generates a \(d \times d|\) matrix \(M\), and computes \(X' = MX, Y' = M^{-1}Y\). Then \(S\) randomly generates secret keys: \(\{M_1^*, M_2^*, K_{spd}\}\) and simulates \(V^*(\mathbb{H}_k)\) as follows:

- (Simulating \(I^*\)) \(S\) treats each row vector \(X'(i,:)\) as an extended fingerprint vector and simulates the splitting and transforming process of \(\text{IndexBuild}\) to encrypt \(X'(i,:)\) with the keys \(\{M_1^*, M_2^*, K_{spd}\}\). \(X'(i,:)\) are encrypted into two vectors \(\{X'_a(i,:), X'_b(i,:)\}\). All the encrypted vectors constitute the set of encrypted indexes \(I^* = \{X'_a(1,:), X'_b(1,:)\}\).

- (Simulating \(T^*\)) \(S\) splits \(Y'\) as \(Y' = (Y'_1, Y'_2, ..., Y'_{k})\) where \(Y'_{j}\) is a \(d \times (2d + 1)|k|\) block submatrix. Let \(Y'_{j}(l,:)=i\) be the \(i\)-th column of \(Y'_{j}\) and \(Y'_{j}(m|i)=Y'_{j}(l,m),Y'_{j}(m+1),...,Y'_{j}(l,m+1)\). For \(j = 1, 2, ..., k\), \(S\) treats \(Y'_{j}(l,1)\) as an extended fingerprint vector \(\tilde{v}_{pj}, Y'_{j}(2|d+1)\) and \(Y'_{j}(d|d+2|d+1)\) as two extended weight matrixes \(H'_{l}^2\) and \(H'_{l}^3\) respectively. Then it simulates the splitting and transforming process of \(\text{TrappedGen}\) to encrypt \(\tilde{v}_{pj}, H'_{l}^2, H'_{l}^3\) with \(\{M_1^*, M_2^*, K_{spd}\}\). The encryption output is \(T'_j = \{\tilde{p}_{pj}, \tilde{g}_{pj}\}, \{H'_{l}^{ga}, H'_{l}^{ga}\}, \{H'_{l}^{ya}, H'_{l}^{ya}\}\). Finally, it outputs \(T^* = \{T'_1, T'_2, ..., T'_k\}\).

- (Simulating \(C^*\)) \(S\) randomly generates a simulated data set \(C^* = \{C_i^* = \{0,1\}^{|\mathcal{D}|}\}_{i = 1, 2, ..., n}\). For \(j = 1, 2, ..., k\), \(S\) calculates \(\Omega(I^*, T^*_j)\) by simulating \(\text{Search}\) over the intermediate indexes in \(\delta\). Then, it selects the top-k secure indexes for \(T^*_j\) and associates such secure indexes with data items in \(D(q_c) \subset T^*(\mathbb{H}_k)\). After that, if there still exist secure indexes that are not associated with any items, \(S\) randomly associates them with encrypted items.

In the end, \(S\) outputs the view \(V^*(\mathbb{H}_k) = \{C^*, I^*, T^*\}\).

The correctness of such construction is easy to demonstrate by querying \(T^*\) over \(I^*\). In the construction of \(V^*(\mathbb{H}_k)\), the underlying fingerprint vectors and weight matrixes are random values based on the random generator \(H\). However, as proved in Theorem 1, there is no P.P.T adversary that can break data confidentiality of GPSE under known plaintext model. Therefore, we claim that there is no P.P.T adversary that can distinguish \(\{I,T\}\) from \(\{I^*, T^*\}\). Since \(C\) is encrypted under standard symmetric encryption such as AES which is considered as pseudo-randomness and \(C^*\) is randomly generated, the adversary can not distinguish \(C\) from \(C^*\). Thus there is no P.P.T adversary that can distinguish \(V^*(\mathbb{H}_k)\) from \(V^*(\mathbb{H}_k)\) although their underlying histories have the same trace.

VI. PERFORMANCE EVALUATION

We build a prototype of GPSE scheme to evaluate its performance. We conduct GPSE on real data set: an ACM publication database which contains 38469 papers on the topic of information systems. All the papers are published from 2002 to 2011 and included in the ACM Digital Library. To build the indexes, we randomly extract 12000 plaintext keywords from the papers and the average length of all the keywords is 8.19. We implement GPSE on a workstation equipped with a 4GB RAM and an Intel Core Duo CPU running at 2.93GHz. All the algorithms are programmed with C language and each element in the fingerprint vector is chosen as a 32-bit float variable.

Referring to most information retrieval systems, we evaluate the performance of GPSE through three parameters: search accuracy, search time complexity and memory overhead.

A. Search Accuracy

Precision rate and recall rate are two indicators that are widely used to evaluate the search accuracy of an information retrieval system [3] [8] [21]. Precision is the percentage of matched indexes among all the returned indexes. Recall is the ratio between the matched indexes in the returned results and those in the original data sets. Considering they are inter-constrained parameters, the precision-recall curve is a commonly-used method to estimate the search accuracy.

1) Search accuracy of Substring Matching: We randomly generate 6000 substrings (bstr) from the 12000 keywords and the length of each substring varies from 3 to 5. The frequency of the sampled substrings varies from 1 to 825. The parameters for fingerprint vector extraction and weight matrixes definition are selected as \(CT = 2, \mu = 8\) and \(\omega = 10\). We use \(d_1 - d_2 - d_3\) to denote the size of fingerprint vectors. Then we conduct 6000 encrypted pattern matching queries over the secure indexes. GPSE is a top-k ranked information retrieval...
scheme and different values of \( k \) result in different precision rate and recall rate. For comparison, we also introduce an ideal scheme that conducts traditional pattern matching string search algorithms (such as KMP algorithm [24]) on the plaintext data set. However, the ideal scheme is also required to return \( k \) indexes for each query. Fig. 2(a) shows the average precision rate and recall rate under different \( k \). We can see that the search accuracy of GPSE is very close to the ideal scheme. It demonstrates GPSE has high search accuracy. Fig. 2(b) shows the average precision-recall curve. In the curve, we sample the recall rate from 0.1 to 1 by the step of 0.1 and investigate the average precision of the 6000 queries. We can see that GPSE has a very high accuracy for substring matching and the average precision is not affected much by the recall rate. It means that the matched indexes are the top ones in \( I_R \) with a high probability.

2) Search accuracy of LPF Matching: We randomly generate 6000 string prefixes with the length of 6 from the 12000 keywords. The parameters are configured as those in the experiment of substring matching. Then we conduct 6000 LPF matching queries over the secure indexes. For LPF matching, the matched index strings should be arranged in \( I_R \) according to the length of their prefixes. Here we introduce a metric to evaluate the search accuracy of LPF matching. Given a query pattern \( q \) with the length \( l_q \), we use \( W_q(l) \) to denote all the index strings that have more than \( l \) common prefixal characters with \( q \) in the plaintext index string set \((1 \leq l \leq l_q)\). Let \( n_q = |W_q(l)| \). Among the top-\( n_q \) indexes of returned \( I_R \), suppose there are \( n_q^- \) indexes generated from \( W_q(l) \). For an ideal search algorithm, \( n_q^- / n_q \) will always be 1. We use the probability distribution function \( p(x) = P(n_q^- / n_q \geq x) \) to evaluate the search accuracy of the 6000 queries. Fig. 3 shows the search accuracy under different fingerprint vector size and prefix length \( l \) (preLen in the figure). If the vector size is \( 27 - 128 - 64 \), 98% of the queries have the ideal search results (i.e., \( n_q^- / n_q = 1 \)) for \( l = 6 \) and that is 95% for \( l = 5 \), 90% for \( l = 4 \). When \( l = 3 \), the probability of ideal search results falls down to 64%. Actually, if the string pattern query has less characters, the false positive and false negative will both become larger (see the mathematical analysis in section III-D). As shown in Fig. 3(b), the larger vector size can increase the search accuracy since it can reduce hash collisions.

**B. Search Efficiency**

In this section, we investigate the search efficiency of GPSE through two parameters: search time cost and memory overhead. Referring to the design of the Search algorithm, both parameters depend on two factors: the size of fingerprint vector and the number of secure indexes. Moreover, they are not relevant to the type of matching rules that are all defined by fixed-scale weight matrices. Without loss of generality, we can just use substring matching to evaluate the two parameters. For a certain number of secure indexes, we randomly conduct 1000 queries over it and evaluate the average time cost of the 1000 queries. As shown in Fig. 4(a), we can see that the time cost is linear to the number of secure indexes and positively correlated to the vector size. It is because the algorithm should scan the whole index set with \( O(N) \) complexity and the computation complexity is positively correlated to the vector size. Note that the magnitude of search time cost for GPSE is practical. Fig. 4(b) presents the memory overhead of GPSE. Note that the memory overhead is linear to both the two parameters and independent of the matching rules. It demonstrates that we can use the same underlying secure indexes to support various pattern matching rules. Besides, the memory overhead is not relevant to the scale of original data set \( (D) \). This feature is conducive to the scalability of SE schemes for big data sets.

**VII. RELATED WORKS**

**A. Searchable Encryption Schemes**

Most searchable encryption schemes can be classified into two categories according to the search functionality: boolean search and fuzzy search. For boolean search, such SE schemes aim to check the existence of the whole exact query string in the secure indexes. Until now, there have been solutions for conjunctive keyword search [1], single ranked keyword search [2], multi-keyword search [3], and practical implementation over large data sets [4], etc. For fuzzy search, existing solutions mainly focus on the specific problem of secure similarity search. They can capture similarity based on edit distance [5] and gram tokens [7]. Recently, Bing Wang et al. [8] first proposed an SE scheme supporting multi-keyword similarity search. Different from above solutions, Yasuda et al. [25] proposed a solution to calculate multiple Hamming distance on encrypted data using somewhat homomorphic encryption. However, their solution only supports binary text and cannot support wildcard-based string patterns. Thus the search functionalities of existing SE schemes can not support the generalized pattern matching string search on encrypted cloud data.
B. Pattern Matching based on Secure Multiparty Computation

Secure multiparty computation (SMC) can conduct a function between multiple parties privately, i.e., each party should not learn sensitive information from the others’ data. Pastoriza et al. [9] first consider pattern matching using secure two-party computation. Their solution adopted finite automata to conduct the well-known KMP algorithm [24] which is used for pattern matching on plaintext data. To further improve the efficiency, Jonathan et al. [10] modified Yao’s garbled circuit methodology [11] to conduct secure pattern matching (SPM). Then Gennaro [12] first proposed an SPM scheme with full security in the face of malicious adversaries. In other SMC-based solutions, SPM is transformed into linear character comparison [13], secure fast fourier transformation [14] and matrix manipulation [15]. However, SMC-based solutions can not be directly used for the data outsourcing scenario in cloud [5] [16]. The first reason is the security model. In SMC, each party knows its own plaintext data and may directly use it for the intermediate computation [13] [15], this is not available for the cloud server. Meanwhile, each party in SMC can act as another peer party and start a session of secure computation with itself. However, the cloud server running SE solutions should not generate valid query trapdoors and start a query session with itself. The second reason is the too heavy user-side complexity. In SMC, the parties are equivalent peers and have comparable computation burdens. To query on a data set of N scale, the user based on SMC usually suffers $O(N)$ computation complexity, $O(N)$ bandwidth cost or even $O(N)$ rounds communication [13] [14]. By contrast, the user-side computation and communication complexity of GPSE are both $O(1)$.

VIII. Conclusion

In this paper, we proposed a generalized pattern matching string search scheme GPSE over encrypted cloud data. GPSE is the first searchable encryption scheme for generalized pattern matching search in cloud systems. By using GPSE, users can flexibly specify their search queries with privacy preservation. To achieve this, GPSE transforms pattern matching into vector-matrix computation. By introducing the weight matrices, various string matching patterns can be defined in a fine-grained manner. The one-way transformation and extended secure kNN computation are used to preserve the privacy during the computation. We proved that GPSE is KPA-secure. The experiment results over real data set have shown that GPSE achieves high search accuracy with practical search time.

For the future work, we will extend GPSE to support general SQL-Like queries over encrypted text data. GPSE will support string patterns of various combination of wildcards and logic operations.

ACKNOWLEDGMENT

The work is supported in part by the National Natural Science Foundation of China (NSFC) under grants 61379144 and 61402513 and the Research Grants Council of Hong Kong [Project No. CityU 11205014 and CityU 138513].

REFERENCES