Privacy-Preserving Multi-pattern Matching

Tao Zhang, Xiuhua Wang, and Sherman S.M. Chow(✉)

Department of Information Engineering,
The Chinese University of Hong Kong, Sha Tin, N.T., Hong Kong
{zt112,wx015,sherman}@ie.cuhk.edu.hk

Abstract. Multi-pattern matching compares a large set of patterns against a given query string, which has wide application in various domains such as bio-informatics and intrusion detection. This paper presents a privacy-preserving multi-pattern matching system which processes an encrypted query string over an encrypted pattern set. Our construction is a symmetric-key system based on Aho-Corasick automaton. The computation complexity is the same as the basic automaton in the base case, and within a multiplicative cost in the length of the longest pattern in general.

Keywords: Multi-pattern matching · Symmetric searchable encryption

1 Introduction

Storage service witnesses an increasing popularity for satisfying different needs. Some cloud services also provide applications which perform computation over the outsourced data. However, the data is often sensitive. The users may not fully trust the well behaviour of the cloud. A natural solution for the client is to encrypt the data and outsource the ciphertexts instead. Without the key, one learns nothing about the plaintext. The cloud thus cannot perform any useful function over the data. For example, one may want to apply multi-pattern search over the emails or network traffic for virus scanning or intrusion detection.

Chase and Shen [6] propose a queryable encryption scheme for substring queries based on suffix trees. It can find all occurrences of a query string p as a substring of an outsourced string s. This is like the dual of the multi-pattern search problem — a set of pattern is outsourced and the query is a string for checking which pattern appears at which position of the string.

This paper studies multi-pattern matching over encrypted pattern sets and encrypted query. Given a large pattern set M where each pattern consists of the characters from an alphabet set Σ, the multi-pattern matching algorithm can locate all the patterns from M which appear in a query string q (also consisting of the characters in Σ). This allows the owner of a large set of patterns to outsource the set to a cloud server. At the same time, the data owner can allow any client to make queries to this encrypted pattern set. The cloud server is deterred from learning the pattern or the query. Multi-pattern matching is one of the key technologies for string analysis, and has found application in bioinformatics, business analytics, natural language processing, web search engines, *etc.*

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2017
R. Deng et al. (Eds.): SecureComm 2016, LNICST 198, pp. 199–218, 2017.
DOI: 10.1007/978-3-319-59608-2_11

1.1 Related Work

Searchable Encryption and Structured Encryption. Symmetric searchable encryption (SSE) [7] is a symmetric-key encryption scheme which allows any server hosting the ciphertexts to search over them. The search requires a token generated by the client who holds the symmetric-key. The server is usually considered honest-but-curious, except in special schemes such as verifiable SSE proposed by Kurosawa and Ohtaki [10]. In a typical SSE, the server can easily know which parts of the memory have been accessed when the same memory block is accessed again. The security definition of SSE acknowledged the leakage of information about the plaintext due to a query, such as access pattern.

Most of the existing schemes aim to locate where the keyword query is in the outsourced files, if exists. Some SSE schemes support search beyond the basic keyword equality testing. For example, Cash *et al.* [4] proposed an efficient construction for searches involving multiple keywords. Generally speaking, exact keyword search and pattern matching are quite different. A recent scheme by Wang *et al.* [13] uses locality sensitive hash (LSH) with other techniques to achieve fuzzy search. Yet, LSH is for hashing similar keywords to the same output, but is not applicable for pattern matching in general.

Our privacy-preserving multi-pattern matching scheme falls in the scope of structured encryption [5,11]. Structured encryption is a generalization of SSE, which protects the data privacy while preserving the functionality in the original data structure. However, existing instantiations [5,11] only support data structures which are not readily extensible for our multi-pattern matching problem.

Secure Two-Party Computation of Pattern Matching. Pattern matching and other text processing has been studied in the context of secure two-party computation [3,8,9,12]. The motivation is to protect sensitive data such as DNA records from the client. In other words, the client has no knowledge about the data on the server, and should not obtain extra knowledge beyond each query. In contrast, our setting assumes the client knows and prepares the data to be outsourced to the server. It partially explains why efficient query is possible.

Authenticated Multi-pattern Matching. Recently, Zhou *et al.* [14] proposed an authenticated but not privacy-preserving solution for Aho-Corasick automaton [2]. Their scheme use dynamic accumulator (*e.g.*, [1]) to ensure the authenticity of the query result, which features constant-size proof. However, when the data stored on the server is encrypted, it is unclear how to follow their technique and generate accumulator for different nodes dynamically during a query. Instead, our proposed scheme uses symmetric-key encryption for authentication.

1.2 Our Contribution

We propose the first multi-pattern matching algorithm on encrypted pattern which is secure against malicious adversaries. Our scheme is based on

Aho-Corasick automaton (AC automaton) [2] and benefits from its efficiency. The computation complexity is $O(n+m)$ in the best case (the same as the plain AC-automaton with direct *fail* pointers), or $O(n \cdot d)$ in the worst case, where n is the length of the query string, m is the number of matched patterns, and d is the length of the longest pattern. The communication complexity is proportional to the computation complexity, as both are proportional to the nodes processed.

2 Preliminaries

This section reviews the pattern matching algorithm and some cryptographic primitives used in our proposed system.

2.1 Trie

Trie, also known as prefix tree or radix tree, is a $|\Sigma|$-ary tree for storing a large amount of strings formed by characters from the alphabet Σ. Each path from the root to a node represents a string, which is a common prefix of the strings represented by its succeeding (child) nodes. Every string in a trie can be represented by a path from the root to a node, and this path represents a common prefix of some strings. The root node denotes a null string. Any other node represents a prefix that is created by appending the character of the incoming edge, to the prefix that its parent node represents.

Figure 1 shows a sample trie. A trie $\mathcal{T}$ is setup by adding the patterns to it one by one. We traverse $\mathcal{T}$ from the root with respect to the character sequence of a pattern string. During the traversal, the current node may not have an outgoing edge which represents the next character in the string. In such cases, we add a new edge, from the current node to a new child node, to denote this missing character. The node identifier is its timestamp of insertion. We continue the traversal until the current node is the *end of the pattern*. We then mark it as a *gray* ending node, *e.g.*, node 4. Trie can reduce storage by merging all the common prefixes. The dashed and dotted edges (representing *fail* and *sp* pointers respectively) will be used by the AC automaton.

Searching on a trie is performed in a depth-first manner by sequentially taking one character from the query each time. If there is an outgoing edge for the character, it moves along it. If the outgoing edge for the next character does not exist or the query ends on a non-ending node, the search fails.

2.2 Aho-Corasick String Matching Algorithm

A naïve and costly way of multi-pattern matching is to enumerate all the patterns. Aho-Corasick algorithm [2] (AC automaton) is a pattern matching algorithm based on the trie structure which aims to guarantee the correctness without explicitly processing the patterns one by one. Multi-pattern matching algorithm can locate all the pattern occurrences in the query text by checking if the pattern is a prefix of any suffix of the text. The major idea is to sort all the

prefixes of the query string (*e.g.*, "otear") from short to long (*e.g.*, "o", "ot", "ote", "otea", "otear"), then find all the suffixes which are in the pattern set for each prefix (*e.g.*, "tea" and "a" for "otea"). Correctness is guaranteed by conceptually covering all the suffixes.

For the searching process, the automaton traverses the trie from the root according to the query string, except for the following two special treatments.

The AC automaton adds *fail* pointers to the trie for quick traversal when mismatch happens, *i.e.*, if a node does not have an outgoing edge for a character c, it must have a *fail* pointer pointing at a node to which the searcher should go since that represents a suffix of the current node. Specifically, consider the string denoted by the path from the root to a node v, if there exists another path which denotes a suffix of it, and the last node on this path has a child node w denoting the character c, then *fail* pointer for c of node v points at w. If there is no such suffix on the trie, the *fail* pointer for c of node v points at the root.

In Fig. 1, *fail* pointers are shown as dashed edges. For example, consider node 8, it has one *fail* pointer for 'n' pointing at node 5, and one for 'r' pointing at node 3. Note that a null string is always a suffix, we omit the *fail* pointers pointing at the root or the second level nodes because there are too many of them (*e.g.*, *fail* pointers pointing at nodes 1, 4, 6 for 'e', 'a', 't', respectively are omitted). Step 5 in Table 1 is an example of quick traversal via *fail* pointers.

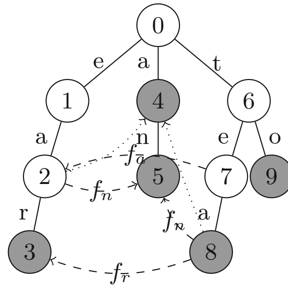

Fig. 1. AC automaton with pattern set {"ear", "a", "an", "tea", "to"}

When the traversal returns to the root via a *fail* pointer, it gives up the current character and moves to the next character in the query string. Step 1 in Table 1 gives an example, which removes 'o' and starts processing 't'.

The second special treatment is the suffix pattern (*sp*) pointers. They are shown as dotted edges in Fig. 1. For a node (may not storing a pattern) which the traversal path from the root to it gives the string s, its *sp* pointer points to an ending node (except itself) which denotes the longest suffix of s in the pattern set. In this way, we can *efficiently output* all the matched patterns for a given query by backtracing along *sp* pointers until the root is reached. (See Step 4 in Table 1 as an example.) Since all the suffixes corresponding to a particular node were accessed, backtracing guarantees that any results will not be missed even though the traversal is processing a particular path.

Table 1. Matching Flow for "otear"

Step	Prefix	Suffix	Movement	Action
1	"o"	""	$0 \rightarrow 0$	The root has no edge for 'o', and its *fail* pointer for 'o' points at the root itself, so stay at the root, and the character in the query to be processed is now 't"
2	"ot"	"t"	$0 \rightarrow 6$	The root has an edge for 't', move to Node 6
3	"ote"	"te"	$5 \rightarrow 7$	Node 6 has an edge 'e', move to Node 7
4	"otea"	"tea"	$7 \rightarrow 8$	Node 7 has an edge 'a', move to Node 8. Output "**tea**" as it is an ending node. Also output "**a**", because Node 4 is linked by the *sp* pointer of Node 8
5	"otear"	"ear"	$8 \rightarrow 3$	Node 8 has no edge 'r', use *fail* pointer f_r to jump to Node 3. Output "**ear**" as it is an ending node. Node 3 has no *sp*/*fail* pointer, so search ends

Each node v stores a set of pointers for its child nodes $\{w_c\}_{c \in \Sigma}$, *fail* pointers, and its *sp* pointer. We require the total number of the child edges and the *fail* pointers of each node is exactly the size of the alphabet $|\Sigma|$, which is a constant. Every nodes thus look the same in this regard which helps us to achieve privacy.

Efficiency. We omit the factor of $|\Sigma|$ in our analysis since it is a small constant for all alphabetic scripts such as English, French, *etc.* Let N denote the number of patterns stored by $\mathcal{T}$, and ℓ is the average pattern length. For setup, the time complexity is $O(N\ell)$, including constructing the trie, the *fail* pointers and the *sp* pointers for each node of the trie. The storage size is also $O(N\ell)$.

Searching accesses $O(n + m)$ nodes where n is for traversal, and m is for outputting result including backtracing.

2.3 Cryptographic Building Blocks

Symmetric-Key Encryption (SKE). An SKE $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ consists of the following three probabilistic polynomial-time (PPT) algorithms:

- $K \leftarrow \mathsf{Gen}(1^\lambda)$: this algorithm takes the security parameter λ and outputs a secret key K of length determined by λ.
- $CT \leftarrow \mathsf{Enc}_K(M)$: this algorithm takes a key K and a message M, outputs a ciphertext CT.
- $(M, \perp) \leftarrow \mathsf{Dec}_K(CT)$: this algorithm takes a key K and a ciphertext CT, outputs a message M, or the symbol $\perp$ indicating CT is invalid.

We require the following security properties for symmetric encryption:

- *Correctness* requires $\mathsf{Dec}_K(\mathsf{Enc}_K(M)) = M$ with probability of 1 for all K and M.

- *CPA (chosen-plaintext attack) security* requires that for any PPT adversary who can adaptively query an encryption oracle, the ciphertexts reveal no information about plaintexts (other than their lengths).
- *Ciphertext integrity* requires that given accesses to an encryption oracle, all PPT adversaries cannot construct a new ciphertext (not output by the encryption oracle) that decrypts successfully. We say that a symmetric encryption scheme is *authenticated* if it has both CPA security and ciphertext integrity.
- *Key hiding* (also known as *which-key concealing*) requires that given two encryption oracles, all PPT adversaries cannot tell whether they encrypt using the same key or different keys.

Pseudorandom Function (PRF) Family. A PRF $F : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^*$ is a family of functions with efficient evaluation. The output of a PRF is computationally indistinguishable from a uniform distribution over the range of F.

2.4 Queryable Encryption

We borrow the notion of queryable encryption introduced by Chase and Shen [6] to model our privacy-preserving multi-pattern matching scheme.

Definition 1 (Queryable Encryption). For message space $\mathbb{M}$, query space $\mathbb{Q}$, and result space $\mathbb{R}$, we define a *queryable encryption scheme* which supports query functionality $\mathcal{F} : \mathbb{M} \times \mathbb{Q} \to \mathbb{R}$ by the PPT algorithms/protocols below.

- $\mathsf{Gen}(1^\lambda) \to \mathsf{sk}$: this algorithm takes a security parameter λ as an input and outputs a secret key sk.
- $\mathsf{Enc}(\mathsf{sk}, M) \to CT$: this algorithm takes a secret key sk and a plaintext message $M \in \mathbb{M}$ as inputs and outputs a ciphertext CT.
- $\mathsf{Query}((\mathsf{sk}, q), CT)$: this is an interactive query protocol between a client and a server. The client input is a secret key sk and a query $q \in \mathbb{Q}$. During the interaction, the client generates a query token to be sent to the server. The server input is a ciphertext CT. The server interacts with the client by receiving query tokens and returning intermediate results. The final output of the client is a query result $R \in \mathbb{R}$. The server has no final output.

Correctness of queryable encryption requires, for all $\lambda \in \mathbb{N}$, $q \in \mathbb{Q}$, $M \in \mathbb{M}$, let $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda)$, $CT \leftarrow \mathsf{Enc}(\mathsf{sk}, M)$, and $R \leftarrow \mathsf{Query}(\mathsf{sk}, q, CT)$, we have that $\Pr[R = \mathcal{F}(M, q)] = 1 - \mathsf{negl}(\lambda)$, where $\mathsf{negl}(\lambda)$ is a *negligible* function of λ.

The security definition of a queryable encryption is parameterized by two leakage functions $\mathcal{L}_1$ and $\mathcal{L}_2$. $\mathcal{L}_1$ denotes the information about the message leaked by the ciphertext. For any j, $\mathcal{L}_2(M, Q_1, \cdots, Q_j)$ denotes the information about the message and all queries made so far that is leaked by the j-th query.

We allow the adversary to be arbitrarily malicious in the protocol. Thus we require that a malicious server either produces the correct output or will be detected. This definition guarantees both privacy and correctness. We recall the definition of security against $(\mathcal{L}_1, \mathcal{L}_2)$-chosen query attack (CQA2) [6].

Definition 2 (Malicious $(\mathcal{L}_1, \mathcal{L}_2)$-CQA2 security). Let $\mathcal{E}$ = (Gen, Enc, Query) be a queryable encryption scheme for message space $\mathbb{M}$, query space $\mathbb{Q}$, result space $\mathbb{R}$, and query functionality $\mathcal{F} : \mathbb{M} \times \mathbb{Q} \rightarrow \mathbb{R}$. We define the following experiments for leakage functions $\mathcal{L}_1$ and $\mathcal{L}_2$, adversary $\mathcal{A}$, and simulator $\mathcal{S}$:

- $\mathbf{Real}_{\mathcal{E},\mathcal{A}}(\lambda)$: The challenger uses $\mathsf{Gen}(1^\lambda)$ to output a secret key K. The adversary $\mathcal{A}$ outputs a message M. The challenger runs $CT \leftarrow \mathsf{Enc}(K, M)$ and sends CT to the adversary. The challenger adaptively makes a polynomial number of queries $Q_1, \cdots, Q_t$. The challenger plays the role of client in the query protocol with input (K, Q_i) and sends the output to adversary. Finally, $\mathcal{A}$ outputs a bit b.
- $\mathbf{Ideal}_{\mathcal{E},\mathcal{A},\mathcal{S}}(\lambda)$: The adversary outputs a message M. Given $\mathcal{L}_1(M)$, the simulator $\mathcal{S}$ outputs CT. The adversary adaptively makes a polynomial number of queries $Q_1, \cdots, Q_t$. For each query Q_i, the simulator is given $\mathcal{L}_2(M, Q_1, \cdots, Q_i)$ and interacts with the adversary. Then the simulator produces a flag f_i. If $f_i = \perp$, then the challenger sends $\perp$ to $\mathcal{A}$. Otherwise it outputs $\mathcal{F}(M, Q_i)$. Finally, $\mathcal{A}$ outputs a bit b.

We say that $\mathcal{E}$ is $(\mathcal{L}_1, \mathcal{L}_2)$-CQA2 secure against malicious adversaries if, for all PPT adversaries $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that:

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

3 Privacy-Preserving Multi-pattern Matching Solution

We first give some intuitions about how to process the AC-automaton for preserving its privacy, then we present the details of our construction.

3.1 Modified AC-automaton

To preserve privacy, we need to hide as much structure as possible. We should allow the recovery of the pointers only when needed. A straightforward approach which works on encrypted trie requires a high number of interactions. It can be as much as the number of traversal steps since the client cannot predict ahead the path of traversal, especially for *fail* or *sp* pointers. Here, we describe how to preprocess the AC-automaton for hiding all the pointers, and encrypt both the query string and the trie, while keeping communication complexity in mind.

Matching the Encrypted Query and the Encrypted Trie. A straightforward way to let the server query over the encrypted trie is to process each character in the query one by one. Our scheme instead generates a sequence of substrings of length $d + 1$ from the query, *i.e.*, one character longer than the longest pattern. We can thus reduce the interaction rounds from $O(n)$ to $O(\frac{n}{d})$ in the best case.

The client prepares the query token by encrypting the query substrings as a series of ciphertexts. The server uses the pre-computed keys stored in the trie

to try decrypting them. The keys are pre-computed by the data owner in the following way. For each node, the data owner derives its *address in the dictionary* and a symmetric key to be stored in its parent node by using two PRFs, both taking as input the string denoted by the path from the root to this node.

During the traversal, if there exists a key which can successfully decrypt the given ciphertext, which happens when the string corresponding to the node matches the sub-string corresponding to the query token, the server can locate the next node with the address obtained from the decryption. Otherwise, the client and the server rely on the *fail* pointer to continue searching.

Handling the *fail* Pointers. Recall that if there is a mismatch happens while searching on the current path, the traversal follows the *fail* pointer which indicates a node to go. When this happens, we say that the search triggers a *fail event*. As described in the previous part, a query is divided into substrings. A fail event ends the current substring at the fail position, and starts the next substring at this fail position. The node pointed at by the *fail* pointer is the *entrance node* for continuing the next query substring.

Whenever a fail event happens, the client can always know the string denoted by the path from the root to the entrance node, when given the level of this node. The reason is that this string is a suffix string of the current substring. The current node indicates the ending position and the level of the next entrance node indicates the length of this suffix string. With this suffix string, the client can compute and send to the server the address of this entrance node (recall how the address in the dictionary is computed from the previous part).

Instead of storing the *fail* pointer directly, a node stores a tuple (c, L_c), denoting that a *fail* pointer for the character c points at a node at level L_c. The client finds the entry for the first mismatching character to get the level of the next entrance node, and recovers the *fail* pointer on spot during the search.

One could omit the *fail* pointers which point to the root. However, this approach leaks the number of fail pointers. Instead, each node stores an entry for every character in the alphabet (*e.g.*, by setting $L_c = -1$ for a child edge denoting c which is not a *fail* pointer). The overhead is reasonable for a small alphabet size.

Handling the *sp* Pointers. Consider the string denoted by the path from the root to current node, recall that the *sp* pointer indicates its longest suffix which is in the pattern set. Instead of storing the *sp* pointers, each node stores all the suffix patterns of a node (located by tracing recursively through the *sp* pointers until the root) with its own pattern. This modification reduces the number of steps during the search to $O(n)$ where n is the length of the query, but increases the server storage to $O(N\ell)$. The maximum number of suffix patterns a node has is the same as the height d of $\mathcal{T}$. We can hide the number by additionally storing at most d dummy patterns for each node. To reduce the storage, we store an encryption of the pattern identities instead of the actual strings.

Handling Malicious Servers. We allow the server to be malicious (returning tampered messages) instead of just being honest-but-curious. We use authenticated symmetric-key encryption such that the client (decryptor) can check the authenticity of the ciphertext returned by the server.

For a node in the trie, the data owner encrypts the information of the path of this node, its child node, its fail pointers, and its patterns. Modification of the path requires the server to modify the encrypted information of one or more nodes. For deletion of any nodes, the server needs to return the ciphertext stored on the parent node of the node it wants to delete, but that is also authenticated. In these ways, any malicious tampering of servers can be detected.

3.2 Our Proposed Construction

Our scheme requires a PRF $F : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$, and an authenticated, key-hiding, CPA-secure symmetric-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ where $\Pi.\mathsf{Gen}$ outputs a λ-bit key. Table 2 lists the major notations. Below we describe the three algorithms/protocols Gen, Enc, and Query of our scheme.

Table 2. Notations

N	The number of all patterns on the trie $\mathcal{T}$
d	The height of $\mathcal{T}$ and the length of the longest pattern
ℓ	The average pattern length
m	The number of matched patterns
w_c	A child node of v that the edge between v and w_c denotes the character c
L_c	The level of the node w_c pointed at by the *fail* pointer for character c
w_{sp}	A node linked by the *sp* pointer of v
p_v	The string denoted by the path from the root to the node v
p_{root}	A special symbol denoting a "null" string corresponding to the string of the root
$\mathcal{P}$	The set of patterns stored in a node
$\mathcal{N}$	The set of decryption keys (f_c for character c) for locating next nodes
$\mathcal{F}$	The set storing the character of the *fail* pointer and the level of the node it points at
$\mathcal{O}$	The set of search results sent from the server to the client
n	The length of the query string q
q^η	The η-th substring of q (division of q into substrings is done in a specific way)
q_i^η	The i-th character in the η-th substring q^η
$\mathbf{\Sigma}$	The alphabet
$\mathbf{\Sigma}_v$	The set of characters corresponding to the existing child edges of node v
$\mathbf{\Sigma}_{fail}$	The set of characters corresponding to the non-existing child edges of v

$\mathsf{Gen}(1^\lambda) \rightarrow \mathsf{sk}$. The data owner randomly chooses K_D, K_1, $K_2 \xleftarrow{\$} \{0,1\}^\lambda$, and sets $\mathsf{sk} = (K_D, K_1, K_2)$.

$\mathsf{Enc}(\mathsf{sk}, M) \rightarrow CT$. Based on the plaintext pattern set M, the data owner sets up the AC-automation $\mathcal{T}$. The data owner encrypts each node of the resulting $\mathcal{T}$ in a specific way (to be detailed below), and stores the encrypted structure as address/value pairs in a dictionary D on the server. Note that the data owner also stores the height d of $\mathcal{T}$ locally. Algorithm 1 shows the pseudocode. Below we explain the pseudocode and illustrate it with an example.

Algorithm 1. Encrypting the pattern set M with the secret key sk

```
1:  procedure Enc(sk, M)
2:      Setup AC-automaton T to store the pattern set M
3:      Initialize a dictionary D and a queue Q
4:      Q.enqueue(root) // first enqueue the root of T
5:      while Q is not empty do
6:          v := Q.dequeue()
7:          v.P := ∅; v.F := ∅; v.N := ∅;
8:          ({w_c}_{c∈Σ_v ∪ Σ_fail}, w_sp) ← v
9:          if p_v is a pattern then v.P := {p_v}
10:         if v has a suffix pattern pointer sp then v.P := v.P ∪ v.w_sp.P
11:         for all c ∈ Σ do
12:             v.N := v.N ∪ {F_{K_2}(p_v||c)}
13:             v.F := v.F ∪ {(c, (c ∈ Σ_fail ? L_c : −1))}
14:         end for
15:         addr_v := F_{K_1}(p_v)
16:         C_v := Π.Enc_{K_D}((addr_v, v.P, v.F, v.N))
17:         D[addr_v] := (v.N, C_v)
18:         for all c ∈ Σ_v do
19:             Q.enqueue(w_c) //child node w of v
20:         end for
21:     end while
22:     Output: D
23: end procedure
```

For each node v in the trie setup by the AC-automation, we define three sets.

1. Set $\mathcal{P}$ stores the pattern of v and all its suffixes which are also patterns.
2. Set $\mathcal{F}$ stores all the *fail* pointers including those pointing at the root or the child nodes of the root (at level 2). Each entry is in the form of (c, L_c):
   - c is the next character in the query which cannot be reached.
   - L_c is the level of the node that the *fail* pointer for c points to.

   If a character $c \in \Sigma_v$, then the entry $(c, -1)$ is stored to make sure that $|\mathcal{F}| = |\Sigma|$.
3. Set $\mathcal{N}$ stores a set of decryption keys used for decrypting the address of next node. For next node w_c, the key is $F_{K_2}(p_{w_c})$ where p_{w_c} denotes the path from the root to node w_c. No matter whether there exists an outgoing edge for the character c, $\mathcal{N}$ contains an entry for all character c in the alphabet.

The client encrypts $(\mathsf{addr}_v = F_{K_1}(p_v), \mathcal{P}, \mathcal{F}, \mathcal{N})$ as C_v under key K_D, and stores the value $(\mathcal{N}, C_v)$ at address addr_v of dictionary D, *i.e.*, $D[\mathsf{addr}_v] = (\mathcal{N}, C_v)$.

Consider the example in Fig. 1. Suppose the alphabet is {a, e, n, o, r, t}. The pattern set is $\{P_1 =$ "ear", $P_2 =$ "a", $P_3 =$ "an", $P_4 =$ "tea", $P_5 =$ "to"$\}$. For node 8, $p_8 =$ "tea" as the traversal from the root passes through edges marked with 't', 'e', 'a'. It has *fail* pointers marked with 'n' and 'r' in particular (*fail* pointers for 'a', ''e', 'o', 't' are not drawn in Fig. 1), *i.e.*,

- $\mathcal{P} = \{P_4, P_2\}$, $\mathcal{F} = \{(a, 2), (e, 2), (n, 3), (o, 1), (r, 4), (t, 2)\}$.
- $\mathcal{N} = \{F_{K_2}(\text{"teaa"}), F_{K_2}(\text{"teae"}), F_{K_2}(\text{"tean"}), \cdots, F_{K_2}(\text{"teat"})\}$, in which all of them are never re-used in the trie as node 8 has no next (child) node.
- $D[\mathsf{addr}_8] = (\mathcal{N}, \Pi.\mathsf{Enc}_{K_D}(\mathsf{addr}_8, \mathcal{P}, \mathcal{F}, \mathcal{N}))$, where $\mathsf{addr}_8 = F_{K_1}(\text{'tea'})$.

The client also stores locally that $\mathcal{T}$ is of height 3. We do not model it as secret state information kept by the client since it is difficult to protect its secrecy unless the client adds many dummy entries and occasionally issues dummy requests.

$\mathsf{Query}(sk, q, CT) \to \mathcal{R}$. This protocol contains two parts: $\mathsf{query}_{\text{client}}()$ (Algorithm 2) on client side, and $\mathsf{query}_{\text{server}}()$ (Algorithm 3) on server side. These two algorithms interactively compute search result $\mathcal{R}$ as the client private output.

The client runs $\mathsf{query}_{\text{client}}()$ with the input of the secret key sk from the data owner, and a query string $q \in \Sigma^n$ where n can be an arbitrary integer. We separate the query string twice instead of processing the entire q at one shot.

The first separation, from line 5 to line 10 of Algorithm 2, is to protect the access sequence for a long query string. We divide the query string q of length n into overlapping sub-query strings $\{q^\eta\}$ of length randomly chosen from the range $[d + 1, n - 1]$, where d is the length of the longest pattern (a state information kept at the client side). Setting the maximum length to $n - 1$ means q is divided into at least 2 sub-query strings when $|q| > d$. If $|q| \leq d$, there will be only one sub-query string which is q itself. We also randomly permute the order of these sub-query strings. We make the overlap to be exactly of length d. The overlap ensures that the division and the order permutation do not affect the match result. Any duplication caused by the overlapping part can be eliminated since the results correspond to not only the matched pattern but also the matched position in the entire query q. Other than duplicate elimination, one can treat each sub-query string as an independent query.

The second separation is for constructing the substrings dynamically for each communication round. Each sub-query string is separated into substrings of length $d+1$. If the total length of the rest of the query is not long enough, the client appends randomly chosen characters (denoted by $c_{\mathsf{pad},j}$ where j denotes the position of the padded character) to the end. The corresponding matching results will be discarded. The length of $d + 1$ ensures that every interaction will raise a transition via a *fail* pointer, so the server is able to send back all the matched results.

For a substring from position i to $i + d$ of a sub-query string q^η, if i is beyond the length of q^η, the client proceeds to search for another unprocessed sub-query. Otherwise, the client performs the following steps to generate the query ciphertexts:

1. The client computes $\mathsf{ind}_j = F_{K_1}(q_i^\eta \cdots q_{i+j-1}^\eta)$ as the search index and $K_j^* = F_{K_2}(q_i^\eta \cdots q_{i+j-1}^\eta)$ as the search key, for the j-th substring $q_i \cdots q_{i+j-1}$ (where $j \in [1, d+1]$) of the query string.
2. The client encrypts ind_j using K_j^* as $T_j \leftarrow \Pi.\mathsf{Enc}_{K_j^*}(\mathsf{ind}_j)$.
3. If the remaining query length is less than $d+1$, the client continues to pad random character c_{pad} and generate T_j for $j \in [|q^\eta|+1, d+1]$.
4. The client sends $(\mathsf{entrance}, \{T_j\}_{j=1}^{d+1})$ to the server where $\mathsf{entrance}$ is the starting entry for searching in D. This $\mathsf{entrance}$ is either $F_{K_1}(p_{root})$ (the address for the root where the input for pseudo-random function F is null string) or decided by the previous communication in the same sub-query string q^η. The latter case will be elaborated later.

To illustrate, suppose the client queries for "otear", *i.e.*, $n = 5$. Recall that the client stores locally the height of tree $d = 3$. The range is $[3+1, 5-1]$, the length of sub-query string can only be 4. Thus the sub-query strings after first separation are "otea" and "tear". Suppose "tear" is permuted as the first, the second separation first picks a substring of length $3+1$ from it, which is "tear" itself in this case and nothing remains.

Then the client performs the second separation and processes in the first round of the query and sends $Q = \{\mathsf{entrance} = F_{K_1}(p_{root}), T_t, T_e, T_a, T_r\}$ to the server. For example, $\mathsf{ind}_t = F_{K_1}(\text{“t”})$, $K_t^* = F_{K_2}(\text{“t”})$, $\mathsf{ind}_r = F_{K_1}(\text{“tear”})$, $K_r^* = F_{K_2}(\text{“tear”})$, $T_t = \Pi.\mathsf{Enc}_{K_t^*}(\mathsf{ind}_t)$, and $T_r = \Pi.\mathsf{Enc}_{K_r^*}(\mathsf{ind}_r)$.

Once received Q, the server runs $\mathsf{query}_{\text{server}}(\mathsf{entrance}, \{T_j\}_{j=1}^{d+1})$.

1. The server sets $\mathsf{addr} = \mathsf{entrance}$ and locates the entry $(\mathcal{N}, C_v) \leftarrow D[\mathsf{addr}]$.
2. The server adds the ciphertext C_v as the value of entry $D[\mathsf{addr}]$ and add the tuple $(0, C_v)$ to the response set $\mathcal{O}$ which 0 is the current position in this sub-query. The server tries $\mathsf{addr}_j \leftarrow \Pi.\mathsf{Dec}_{f_c}(T_j)$ for all $f_c \in \mathcal{N}$ of the node at $\mathsf{entrance}$ where $j = 1$. If there exists f_c which successfully decrypts T_1, updates $\mathsf{addr} = \mathsf{ind}_j$ and repeats the process for $j \in [2, d+1]$; otherwise, breaks the iteration.

Algorithm 2. Part of the Query protocol executed by the Client for querying q

```
1: procedure query_client(sk, q)
2:     R := ∅
3:     Parse q = q_1 q_2 ··· q_n
4:     i := 1, cnt := 0 // i is the starting position of the cnt-th sub-query string
5:     while (i ≤ max(n − d, 1)) ∨ (i > 0) do // divide q into sub-query strings
6:         cnt := cnt + 1
7:         n_cnt ←$ [d + 1, n − 1]
8:         n_cnt := min(n_cnt, n − i + 1) // to set the length of the last sub-query string
9:         q^cnt := q_i q_{i+1} ··· q_{i+n_cnt−1}
10:        i := i + n_cnt − d // to ensure the overlapping length is d
11:    end while
12:    ζ := cnt // the number of sub-query strings
13:    Choose a random permutation P : [ζ] → [ζ]
```

```
14:     for all η′ = 1 to ζ do
15:         η ← P(η′) // {q^η}_{η=1}^ζ is the set of sub-query strings after permutation
16:         i := 1
17:         entrance := F_{K_1}(p_root)
18:         while i ≤ |q^η| do
19:             for all j = i to min(i + d, |q^η|) do
20:                 ind_j := F_{K_1}(q_i^η q_{i+1}^η ··· q_j^η); K_j^* := F_{K_2}(q_i^η q_{i+1}^η ··· q_j^η);
21:                 T_j := Π.Enc_{K_j^*}(ind_j)
22:             end for
23:             if |q^η| < d+i then // pad the substrings to be of the same length d+1
24:                 for all j = |q^η| + 1 to i + d do
25:                     c_{pad,j} ←$ Σ
26:                     ind_j := F_{K_1}(q^η c_{pad,|q^η|+1} ··· c_{pad,j})
27:                     K_j^* := F_{K_2}(q^η c_{pad,|q^η|+1} ··· c_{pad,j})
28:                     T_j := Π.Enc_{K_j^*}(ind_j)
29:                 end for
30:             end if
31:             Client sends Q := (entrance, T_i, T_{i+1}, ··· , T_{i+d}) to the server
32:             Client receives the response O
33:             {(k, C_{k,v})}_{k=0}^{n′} ← O
34:             n* = min(n′, |q^η| − i + 1) // the ending position without padding
35:             for all k = 0 to n* do
36:                 (addr_k, P_k, F_k, N_k) ← Π.Dec_{K_D}(C_{k,v})
37:                 if decryption in the above line returns ⊥ then abort
38:                 i′ := k + i
39:                 if (k > 0 ∧ ind_{i′} ≠ addr_k) then abort
40:                 if (⊥ ← Π.Dec_{f_c}(T_{i′}), ∀f_c ∈ N_k) then abort
41:                 pos := Σ_{j=1}^{η′−1} |q^j| + k + i − 1
42:                 for all p ∈ P_k do
43:                     if (pos, p) ∉ R then R := R ∪ {(pos, p)}
44:                 end for
45:             end for
46:             L := L_{q_{i+n*}^η} − 1, where L_{q_{i+n*}^η} is from F_{n*} for character q_{i+n*}^η
47:             if L == 0 then // fail pointer points to root
48:                 i := i + n* + 1
49:                 entrance := F_{K_1}(p_root)
50:             else if L > 0 then // fail pointer does not point to root
51:                 i := i + n*
52:                 entrance := F_{K_1}(q_{i−L+1}^η ··· q_i^η)
53:             else // invalid L value or the response set is incomplete
54:                 resend Q to the server or abort
55:             end if
56:         end while
57:     end for
58:     Output: R
59: end procedure
```

Algorithm 3. Part of the Query protocol executed by the Server

```
 1: procedure query_server(Q)
 2:     Parse (entrance, T_1, T_2, ..., T_{d+1}) ← Q
 3:     v := (N, C_v) ← D[entrance]
 4:     Initialize an empty set O
 5:     O := O ∪ {(0, C_v)}
 6:     for all i = 1 to d + 1 do
 7:         for all f_c ∈ v.N do
 8:             addr_c ← Π.Dec_{f_c}(T_i)
 9:             if addr_c ≠ ⊥ then
10:                 v := (N, C_v) ← D[addr_c]
11:                 O := O ∪ {(i, C_v)} // i is the occurring position of C_v
12:                 break // quit the inner loop once the matching character c is found
13:             end if
14:         end for
15:         if addr_c == ⊥, ∀f_c ∈ N then
16:             break // quit the outer loop once the query string "disconnected"
17:         end if
18:     end for
19:     Output: O // send O to the client
20: end procedure
```

3. When the server has iterated all the $d+1$ items or the iteration was broken due to failing in decryption for all $f_c \in \mathcal{N}$, the server sends the set $\mathcal{O}$ to the client as the result for this interaction.

To illustrate, the server can use $F_{K_2}(\text{“t”})$ in $\mathcal{N}$ of the root to decrypt $T_t = \Pi.\mathsf{Enc}_{K_t^*}(\mathsf{ind}_t)$, and get $\mathsf{ind}_t = F_{K_1}(\text{“t”})$, which is the address of node 6. Similarly, $\mathcal{N}$ of node 6 has a key to decrypt $T_e = \Pi.\mathsf{Enc}_{F_{K_2}}(\text{“te”})(\mathsf{ind}_e)$ for address of node 7, and $\mathcal{N}$ of node 7 has a key to decrypt T_a for address of node 8. Once the search reaches address $\mathsf{addr} = \mathsf{ind}_a$ (node 8), decryption leads to $\mathsf{ind}_r = F_{K_1}(\text{“tear”})$ but D has no corresponding entry at this index since there exists no path for “tear” in the trie (or node 8 which corresponds to “tea” has no child pointer of ‘r’). The server returns $\mathcal{O}$.

The client continues executing $\mathsf{query}_{\text{client}}()$ after it gets $\mathcal{O} = \{(k, C_{k,v})\}_{k=1}^{n'}$ of size at most d for the last interaction starting at position i. The client then recovers the ending position in the query string where the pattern occurs as $\mathsf{pos} := \sum_{j=1}^{\eta'-1} |q^j| + k + i - 1$, where η' is the number of sub-queries processed so far. This position can be further used to eliminate duplications.

The client first checks whether the size of the received sequence exceeds the query length. If $n' \le |q^\eta| - i + 1$, the received items T_j correspond to characters within the query for all $j \in [1, n']$. In this case, $n^* = n'$ is the number of valid responses. Otherwise, the items T_j for $j \in [|q^\eta| - i + 2, n']$ correspond to the padded random characters. In this case, $n^* = |q^\eta| - i + 1$ is the size of valid response items. The client decrypts all $C_{k,v}$ to get $(\mathsf{addr}_k, \mathcal{P}_k, \mathcal{F}_k, \mathcal{N}_k)$ for $k \in [0, n^*]$, where the tuple for $k = 0$ corresponds to the entrance. If any

decryption returns $\perp$ or $\mathsf{addr}_k \neq \mathsf{ind}_k$ for any k, the client concludes that the server misbehaved and aborts the execution. The client adds the patterns along with their occurring position pos to the result set $\mathcal{R}$.

After obtained the item $(q^\eta_{i+n^*}, L_{q^\eta_{i+n^*}})$ from $\mathcal{F}_{n^*}$ where $q^\eta_{i+n^*}$ is the first mismatched character in the current substring, the client can get the depth of the parent node for the character $q^\eta_{i+n^*}$ linked by this *fail* pointer via $L = L_{q^\eta_{i+n^*}} - 1$. The client generates the starting position for the next substring as follows.

- If $L = 0$, the *fail* pointer points at the root of $\mathcal{T}$. There is no match for the $(i+n^*)$-th character in the current sub-query string. There is no pattern whose prefix is also a suffix of the string from the first to the $(i+n^*)$-th character of this sub-query string. The client needs to skip the character q_{i+n^*} by setting the next substring starting at position $i + n^* + 1$ and $\mathsf{entrance} = F_{K_1}(p_{root})$.
- If $L > 0$, we should find the next node at level $L_{q^\eta_{i+n^*}} > 1$. The client sets this node as the entrance by $\mathsf{entrance} = F_{K_1}(q^\eta_{i+n^*-L+1} \cdots q^\eta_{i+n^*})$. The *fail* pointer of the last matched node is q_{i+n^*}. Hence the next substring starts searching at position $i + n^*$ of the current sub-query string.
- If $L < 0$, which means that there exists a child node for the last tuple in $\mathcal{O}$. By construction L is a positive number and hence an invalid value is returned. It means the obtained response set is incomplete. The client can try resend the query for current substring to the server, or the client simply aborts and issues a complaint against the server.

The client and the server iteratively proceed with the substrings in the above way. When the client finishes querying all the substrings and sub-query strings q^η for a query q, it outputs $\mathcal{R}$.

In our example, after received the response set for the first interaction, the client generates the second round of query as follows.

1. The client reaches the last matched item $(\mathsf{addr}_8, \mathcal{P}, \mathcal{F}, \mathcal{N})$. The client finds that there is a fail pointer ('r', 4) in $\mathcal{F}$ for the mismatching character "r" of the last matched item. The client traces back $4 - 1 = 3$ characters from 'r' to get the starting position of the path the *fail* pointer pointing at and the substring "ear".
2. The client generates the new $\mathsf{entrance}$ for the second round of query where $\mathsf{entrance} = F_{K_1}(\text{"ear"})$. This is exactly the address for node 3. As the former interaction has searched for a substring of "tea", the second substring for sub-query string "tear" is generated as 'r'. The client pads this string with randomly chosen $c_{\mathsf{pad},j}$ until it has the length of 4. Then the client generates the ciphertexts for these padding characters.
3. The client sends new $Q = \{\mathsf{entrance}, T_{c_{\mathsf{pad},1}}, \cdots, T_{c_{\mathsf{pad},4}}\}$ to the server.

After the server receives Q it directly locates the address for 'r' through the new $\mathsf{entrance}$ and adds the tuple $(0, C_r)$ into the new $\mathcal{O}$ set, where 0 is the occurrence position of 'r' in this substring and C_r is the ciphertext of the entry for 'r'. It continues the search until a fail event happens and then sends back the new response set.

During the second separation, the pattern query of sub-query string "tear" can be finished in two rounds of interaction. However, if the sub-query string is long, the interactions will increase accordingly.

After querying all the substrings in sub-query string "tear", the client can get back the patterns and their occurrence from the result set, and delete those patterns matched by padding characters. The client can perform the query for the second sub-query string "otea" with the same method.

Finally, we remark that the result for a sub-query string is constructed by performing all its interactions in a right sequence, but the result for a query is constructed by combining all the results of its sub-query strings regardless of the order. As long as the substrings for one sub-query string are queried in the right order, mixing the substrings from different sub-query strings or even from different queries during communication will not affect the matching results. The client can switch among different sub-query strings q^η of the same query q, or switch among multiple different queries while interacting with the server, and still get the correct answer for each query. The randomized sub-query string order and the mixing of substrings can further hide the information of the queries and the dictionary. The server can only learn the sequence of nodes hit by a substring, but not the sequence of nodes for a whole query.

3.3 Complexity Analysis

We discuss the complexity of our schemes in this section, and further discuss the feasibility to parallelize the schemes.

Encryption. The computational complexity for AC automaton setup is $O(N\ell)$, which is the total length of all the patterns. The data hiding for the small alphabet case accesses every node on the trie once in a breadth-first manner. The number of nodes does not exceed $N\ell$, so the total complexity is $O(N\ell)$. For each node, at most $|\Sigma| + 1$ PRF evaluations (1 for each character and 1 for the dictionary address) and 1 encryption are performed. So Enc contains $N\ell\cdot(|\Sigma|+1)$ PRF evaluations and $N\ell$ encryption operations.

The storage for the original AC automaton is $N\ell|\Sigma|$. After we remove *sp* pointers, and store all the patterns hit by the node, including the patterns traced via *sp*, on every node, the average storage of a node is increased by a multiplicative factor of d. We require $|\Sigma|$ to be a small constant, or we cannot use this implementation of *fail* pointers (as described in Sect. 2.2). The storage complexity is thus $O(N\ell d)$.

Query. We first divide the query string into sub-query strings which overlap with the adjacent substrings for length at most d. In the worst case, all the sub-query strings are of the length $d+1$. In this case, each sub-query string only contains one substring. We have n sub-query strings. For each sub-query string, considering that *fail* event happens for each character, then there will be $O(d)$ generation of T_j for each sub-query. Hence, the worst case complexity is $O(n\cdot d)$. The original query computational complexity of the AC automaton is $O(n+m)$, where the worst case is also $n + m$.

As our algorithm stores the suffix pattern set in each node instead of using an *sp* pointer to trace the suffix patterns, the number of steps is further reduced to $O(n)$, where the worst case is also n. At each step, the server performs at most $|\Sigma|$ decryptions; and the client performs 1 encryption, 1 decryption, and 2 PRF evaluations. The best case computational complexity remains to be $O(n)$. In the worst case, the server performs at most $n \cdot |\Sigma|$ decryptions while every step requires $|\Sigma|$ decryptions. In our scheme, dividing one sub-query string into several substrings of length $(d+1)$ introduces extra computation overhead. The worst case computational complexity is increased to $O(n \cdot d)$ for the client in the case that each step triggers a *fail* event, as each substring involves $d+1$ encryptions. The time complexity on for the server remains the same. This case is the same as that the sub-query strings are of length $d+1$, which is discussed previously.

Similar to the computational complexity, the communication bandwidth between the server and the client is $O(n)$ for the best case, and $O(n \cdot d)$ for the worst case. The number of interactions for the best case is $O(\frac{n}{d})$, while for the worst case is $O(n)$.

3.4 Security

Due to page limitation, we informally define the leakage of our scheme and outline some important parts of its security proof. We first define the information leakage $\mathcal{L}_1$ and $\mathcal{L}_2$. $\mathcal{L}_1(D)$ contains:

- the number of nodes in $\mathcal{T}$,
- the height of $\mathcal{T}$ which is d.

$\mathcal{L}_2(D, Q)$ contains:

- the sequence of entries $(\mathsf{entrance}, \mathsf{addr}_1, \mathsf{addr}_2, \cdots, \mathsf{addr}_{fail})$ in D hit by each substring Q^i including the $\mathsf{entrance}$ entries and the *fail* positions,
- the distribution of *fail* events.

The queries $Q^i \in \mathbf{Q}$ are substrings of length $d+1$, and may be from different queries or different divisions of the same query. The substrings in the set $\mathbf{Q}$ and the corresponding result from the actual server are given to the simulator $\mathcal{S}$ to perform the simulation. This information is leaked to $\mathcal{S}$ by eavesdropping. By our design, the server cannot tell whether two substrings Q^i, Q^j are from the same query or not, where $i \neq j$. $\mathcal{S}$ uses the leaked information from the previous queries in $\mathbf{Q}$ to answer the adversary's query in **Query** phase.

Theorem 1. *The privacy-preserving multi-pattern matching scheme satisfies malicious $(\mathcal{L}_1, \mathcal{L}_2)$-CQA2 security under the random oracle model, if F is a PRF, and Π is an authenticated, key-hiding, symmetric-key encryption scheme.*

Proof. We show that our scheme only leaks $\mathcal{L}_1, \mathcal{L}_2$ by showing that no PPT adversary $\mathcal{A}$ can distinguish an interaction with the real client from one with a simulator $\mathcal{S}$ which is only given the leakages.

Encryption. $\mathcal{S}$ chooses a secret key $K_D \xleftarrow{\$} \{0,1\}^\lambda$, and sets up the dictionary D with the number of entries matching the number of nodes according to $\mathcal{L}_1(D)$. For each entry i, $\mathcal{S}$ randomly chooses $f_{i,c} \xleftarrow{\$} \{0,1\}^\lambda$ for $c \in \Sigma$, and sets $D[f_{i,1}] = (\{f_{i,c}\}_{c\in\Sigma}, C_i)$ where $f_{i,1} \xleftarrow{\$} \{0,1\}^\lambda$ and $C_i = \Pi.\mathsf{Enc}_{K_D}(0, \emptyset_p, \emptyset_{fail}, \emptyset_f)$.

$\mathcal{S}$ maintains two lists R_1, R_2. R_1 stores the visited items in D. For a queried sub-query Q^i, at the j-th position, the string $p_{i,j}$ is the string denoted by the path from the root to the current node. If this is the first time $p_{i,j}$ appears, $\mathcal{S}$ randomly picks an item with address addr in D, and sets $R_1[i,j] = \mathsf{addr}$. R_2 stores the status of each node and its child nodes. $R_2[\mathsf{addr}] = (K^*, \mathsf{flag}, \{\mathsf{flag}_c\}_{c\in\Sigma})$. $\mathsf{flag}, \{\mathsf{flag}_c\}_{c\in\Sigma}$ are the status of the nodes on the simulated trie indicating whether the nodes or their child nodes are accessed by the previous queries. The initial values of flag and flag_c are all 0, indicating that the node denoted by addr and its child node corresponding to the character c are all unvisited. If addr is chosen as the item for $p_{i,j}$ in the substring Q^i at the j-th step, $\mathcal{S}$ finds $\mathsf{addr} = R_1[(i,j)]$, sets $R_2[\mathsf{addr}].\mathsf{flag} = 1$, picks a character c where $R_2[R_1[(i,j-1)]].\mathsf{flag}_c = 0$, sets it to 1, and sets $R_2[\mathsf{addr}].K^* = D[R_1[(i,j-1])]\cdot f_c$. If a *fail* event occurs at the position j, $\mathcal{S}$ will stop the process.

Query. For a new incoming substring $Q' = (\mathsf{entrance}, T_1, \cdots, T_{d+1})$, $\mathcal{S}$ first checks $\mathcal{L}_2(D,Q)$ for an entrance. If Q' has a common entrance with any one of the substrings in Q, $\mathcal{S}$ uses this entrance, otherwise, $\mathcal{S}$ randomly chooses an entry in D (no matter visited or not), updates it as visited, and updates R_1, R_2 accordingly.

At the j-th step of the substring Q', $\mathcal{S}$ checks $\mathcal{L}_2(D,Q)$ for a common prefix string p_j. If $\mathcal{S}$ finds an item $p_{i,j}$ which equals p_j, which means this prefix string p_j has been visited by the previous substrings Q^i, $\mathcal{S}$ sets $\mathsf{ind}_j = R_1[(i,j)]$ and $K_j^* = R_2[F_{K_1}(p_j)].K^*$, and computes $T_j = \Pi.\mathsf{Enc}_{K_j^*}(\mathsf{ind}_j)$. If $p_{i,j}$ found from $\mathbf{Q}$ which equals p_j has triggered a *fail* event, or $j = d+1$, $\mathcal{S}$ generates the rest of T_j with randomly chosen ind_j and K_j^*. Otherwise, if p_j is not a common prefix or common *fail* event for any substrings in Q, $\mathcal{S}$ first flips a coin according to the distribution of *fail* event, and proceed with the following two cases:

- If $\mathcal{S}$ decides that p_j is not a *fail* event, it randomly chooses an item addr in D, updates R_1 and R_2, and computes T_j as described previously.
- Otherwise, it generates the rest of T_j with randomly chosen ind_j and K_j^*.

When a *fail* event occurs or the index $d+1$ is reached, $\mathcal{S}$ sends Q' to $\mathcal{A}$. $\mathcal{A}$ then follows the process in the real scheme to return $\mathcal{O}' = \{k, C_k\}_{k=0}^{n'}$ to $\mathcal{S}$. $\mathcal{S}$ checks whether the returned C_k is the same as the ciphertext stored in $D[\mathsf{ind}_j]$. If not, $\mathcal{S}$ aborts the simulation and concludes that $\mathcal{A}$ returns wrong response.

Here, we show that the real scheme (Game 0) and the simulation above (Game 5) are indistinguishable from $\mathcal{A}$'s view through the transitions below.

Game 0. This game is the real scheme. $\mathcal{A}$ chooses a series of substrings (not necessarily from the same query), and interacts with $\mathcal{S}$ to obtain the outputs from $\mathcal{S}$.

Game 1. This game is the same as Game 0, except that F_{K_1} and F_{K_2} are replaced with two random oracles. $\mathcal{S}$ keeps records of the evaluation of F_{K_1} and F_{K_2} in two tables. Game 0 and Game 1 are indistinguishable due to the pseudorandomness of F.

Game 2. This game is the same as Game 1, except that $\mathcal{S}$ decides whether to output $\perp$ by checking whether the returned ciphertext $C_{k,v}$ equals the one stored in D for each $k \in [1, n']$ instead of decrypting $C_{k,v}$. Game 1 and Game 2 are indistinguishable due to the authenticity (ciphertext integrity) of Π.

Game 3. This game is the same as Game 2, except that

- C_v encrypts two empty sets instead of the real pattern set and the real *fail* information set for each node;
- T_j encrypts randomly chosen ind for $j \geq i$ if *fail* event occurs at position i.

Game 2 and Game 3 are indistinguishable due to the CPA-security of Π.

Game 4. This game is the same as Game 3, except that T_j is generated with randomly chosen symmetric key K^* for $j \geq i$ if *fail* event occurs at position i. Game 3 and Game 4 are indistinguishable due to the key-hiding property of Π.

Game 5. This game is the same as Game 4, except that $\mathcal{S}$ simulates the scheme without the real pattern set or the real substrings. This is easily achieved, because from Game 1 to Game 4, $\mathcal{S}$ has gradually replaced all the parts related to the pattern set or the substrings with either randomness or the information from $\mathcal{L}_1(D)$ and $\mathcal{L}_2(D, Q)$. Game 5 is actually the simulation described previously, and is indistinguishable with Game 4.

Hence, $\mathcal{S}$ can successfully simulate the scheme with only the information leakage provided by $\mathcal{L}_1$ and $\mathcal{L}_2$, if F is a PRF, and Π is an authenticated, key-hiding, symmetric-key encryption scheme. □

4 Conclusion

In this paper, we propose the first privacy-preserving multi-pattern matching scheme. The previous privacy-preserving searching schemes can only support searching for one pattern at a time. Our scheme enables the feature to search for multiple target patterns simultaneously. A data owner can outsource the storage of a large pattern set and the computation of the searching. Our scheme protects the privacy of both the queried string and the target pattern set. The client does not need to download any data other than the matching result in the process. Our design considers the adversary to be malicious. The client can catch any dishonest behavior of the cloud server during the pattern matching process.

Our scheme is a symmetric-key scheme in which the data owner needs to share the secret key with the clients who need to use the multi-pattern matching service. It is also interesting to design a public-key scheme.

Acknowledgement. We thank for the helpful discussions with Russell W.F. Lai, Jiafan Wang, Yongjun Zhao, and Zhe Zhou.

Sherman S.M. Chow is supported in part by General Research Fund Grant No. 14201914 and the Early Career Award from Research Grants Council, Hong Kong; and Huawei Innovation Research Program (HIRP) 2015.

References

1. Acar, T., Chow, S.S.M., Nguyen, L.: Accumulators and U-Prove revocation. In: Financial Cryptography, pp. 189–196 (2013)
2. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM **18**(6), 333–340 (1975)
3. Baron, J., Defrawy, K., Minkovich, K., Ostrovsky, R., Tressler, E.: 5PM: secure pattern matching. In: Visconti, I., Prisco, R. (eds.) SCN 2012. LNCS, vol. 7485, pp. 222–240. Springer, Heidelberg (2012). doi:10.1007/978-3-642-32928-9_13
4. Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.-C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8042, pp. 353–373. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40041-4_20
5. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 577–594. Springer, Heidelberg (2010). doi:10.1007/978-3-642-17373-8_33
6. Chase, M., Shen, E.: Substring-searchable symmetric encryption. PoPETs **2015**(2), 263–281 (2015)
7. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, 30 October–3 November 2006, pp. 79–88 (2006)
8. Hazay, C., Lindell, Y.: Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. J. Cryptol. **23**(3), 422–456 (2010)
9. Katz, J., Malka, L.: Secure text processing with applications to private DNA matching. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, 4–8 October 2010, pp. 485–492 (2010)
10. Kurosawa, K., Ohtaki, Y.: UC-secure searchable symmetric encryption. In: Keromytis, A.D. (ed.) FC 2012. LNCS, vol. 7397, pp. 285–298. Springer, Heidelberg (2012). doi:10.1007/978-3-642-32946-3_21
11. Lai, R.W.F., Chow, S.S.M.: Structured encryption with non-interactive updates and parallel traversal. In: 35th IEEE International Conference on Distributed Computing Systems, ICDCS 2015, Columbus, OH, USA, 29 June–2 July 2015, pp. 776–777 (2015)
12. Mohassel, P., Niksefat, S., Sadeghian, S., Sadeghiyan, B.: An efficient protocol for oblivious DFA evaluation and applications. In: Dunkelman, O. (ed.) CT-RSA 2012. LNCS, vol. 7178, pp. 398–415. Springer, Heidelberg (2012). doi:10.1007/978-3-642-27954-6_25
13. Wang, Q., He, M., Du, M., Chow, S.S.M., Lai, R.W.F., Zou, Q.: Searchable encryption over feature-rich data. IEEE Trans. Dependable Secure Comput. (2016)
14. Zhou, Z., Zhang, T., Chow, S.S.M., Zhang, Y., Zhang, K.: Efficient authenticated multi-pattern matching. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, 30 May–3 June 2016, pp. 593–604 (2016)