Received January 8, 2018, accepted February 14, 2018, date of publication February 28, 2018, date of current version March 16, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2810297

Enabling Search Over Encrypted Cloud Data With Concealed Search Pattern

JING YAO[1,2], YIFENG ZHENG[2,3], (Student Member, IEEE), CONG WANG[2,3], (Senior Member, IEEE), AND XIAOLIN GUI[1]

[1]School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, China
[2]Department of Computer Science, City University of Hong Kong, Hong Kong
[3]City University of Hong Kong Shenzhen Research Institute, Shenzhen 518057, China

Corresponding authors: Cong Wang (congwang@cityu.edu.hk) and Xiaolin Gui (xlgui@mail.xjtu.edu.cn)

This work was supported in part by the Research Grants Council of Hong Kong under Project CityU 11276816, Project CityU 11212717, and Project CityU C1008-16G, in part by the Innovation and Technology Commission of Hong Kong under Project ITS/168/17, in part by the National Natural Science Foundation of China under Project 61572412 and Project 61472316, and in part by the Key Research and Development Projects of Shaanxi Province under Project 2017ZDXM-GY-011.

ABSTRACT Searchable symmetric encryption (SSE) is a widely popular cryptographic technique that supports the search functionality over encrypted data on the cloud. Despite the usefulness, however, most of existing SSE schemes leak the search pattern, from which an adversary is able to tell whether two queries are for the same keyword. In recent years, it has been shown that the search pattern leakage can be exploited to launch attacks to compromise the confidentiality of the client's queried keywords. In this paper, we present a new SSE scheme which enables the client to search encrypted cloud data without disclosing the search pattern. Our scheme uniquely bridges together the advanced cryptographic techniques of chameleon hashing and indistinguishability obfuscation. In our scheme, the secure search tokens for plaintext keywords are generated in a randomized manner, so it is infeasible to tell whether the underlying plaintext keywords are the same given two secure search tokens. In this way, our scheme well avoids using deterministic secure search tokens, which is the root cause of the search pattern leakage. We provide rigorous security proofs to justify the security strengths of our scheme. In addition, we also conduct extensive experiments to demonstrate the performance. Although our scheme for the time being is not immediately applicable due to the current inefficiency of indistinguishability obfuscation, we are aware that research endeavors on making indistinguishability obfuscation practical is actively ongoing and the practical efficiency improvement of indistinguishability obfuscation will directly lead to the applicability of our scheme. Our paper is a new attempt that pushes forward the research on SSE with concealed search pattern.

INDEX TERMS Searchable symmetric encryption, cloud computing, search pattern leakage, chameleon hashing, indistinguishability obfuscation.

I. INTRODUCTION

Nowadays it is widely popular to outsource data storage to cloud services such as Google Drive, Dropbox, and more. Despite the well-understood benefits, however, data outsourcing to the cloud also naturally raises critical privacy concerns [1]. Indeed data breaches occur frequently in cloud services [2]. For data protection, a plausible approach is to encrypt the data before outsourcing. However, simply applying data encryption will invalidate the fundamentally important search functionality, hindering the effective utilization of the outsourced data and degrading the service experience.

To address the dilemma of data privacy and data utilization, the cryptographic technique of secure searchable encryption has been proposed and has been widely studied in the literature [3], which enables the client to perform search over the outsourced encrypted data. Secure searchable encryption schemes can be either symmetric-key-based or public-key-based. Compared with public-key-based searchable encryption, searchable symmetric encryption (SSE) which builds encrypted index presents much more practical cost efficiency [4], and has attracted wide attention in recent years (e.g., [5]–[7], to just list a few).

However, most of existing SSE schemes are typically built with security trade-offs of access pattern leakage and search pattern leakage. Roughly speaking, the access pattern refers to the search result which indicates which files contain the queried keyword, while the search pattern reveals whether two queries are for the same keyword. Unfortunately, in recent years, it has been shown that the leakages of access pattern and search pattern can be exploited to compromise the confidentiality of the outsourced dataset and queried keywords (e.g., [8]–[12], to just list a few). Therefore, it is of critical importance to address the access pattern leakage and search pattern leakage when SSE is used for encrypted search.

While the access pattern leakage can be well mitigated via introducing dummy data points to the dataset and encrypted index, hiding the search pattern is more challenging and little work has been done before. Prior works act as valuable data points in the design space of hiding the search pattern in SSE, yet they require heuristic parameter tuning for security [12], or suffer from security issues [13], or work under a multi-cloud architecture [14]. Detailed discussion will be given in Section VI. To the best of our knowledge, hiding the search pattern in SSE is still challenging and remains to be fully explored.

In this paper, we present a new SSE scheme which enables the client to search the encrypted cloud data without disclosing the search pattern. As indicated by existing works, the search pattern leakage essentially originates from that the secure search tokens for the queried keywords are generated in a deterministic manner. Hence, our first main idea is to generate the secure search tokens in a randomized way. This means that the secure search tokens are randomized ones even for the same keyword at different queries. Consequently, given two secure search tokens, the cloud is not able to tell whether their underlying plaintext keywords are the same. With this main idea, the challenge is then how to ensure that we can still get the correct search result when secure randomized search tokens are used for encrypted search. To tackle this challenge, our idea is to devise a secure mapping mechanism which should be able to securely map the randomized secure search tokens to the corresponding deterministic versions that can be used to correctly search the encrypted index.

Specifically, our scheme uniquely bridges together the advanced cryptographic primitives of chameleon hashing [15] and indistinguishabilty obfuscation ($i\mathcal{O}$) [16], [17]. At a high level, we mainly rely on the chameleon hashing technique for the generation of secure randomized search tokens. Then, on the cloud side, we rely on the $i\mathcal{O}$ technique to securely map the randomized tokens to deterministic ones for correct encrypted search. In our scheme it is assured that the cloud server is oblivious to both the mapping procedure and the search procedure. This means that the cloud server is not able to obtain the deterministic tokens mapped from the randomized tokens, and also does not observe which entries of the encrypted index are accessed during search. Note that the search pattern may also be leaked if the cloud can observe which entries of the index have been accessed over time [12]. Therefore, our scheme achieves strong protection for the search pattern. In the end, the cloud obtains the search result and returns it to the client. We provide formal security proofs to rigorously justify the security guarantees of our scheme. In addition, we conduct extensive experiments to demonstrate the performance. It is shown that the generation of secure randomized search tokens in our proposed scheme is very efficient, just tens of milliseconds. In our experiments, we also demonstrate the performance of $i\mathcal{O}$. Although currently the $i\mathcal{O}$ technique is inefficient, related research endeavors on practical $i\mathcal{O}$ are actively ongoing. The performance of our scheme relies on the underlying cryptographic technique $i\mathcal{O}$. So, the practical performance improvement of $i\mathcal{O}$ will directly lead to the applicability of our scheme.

The remainder of this paper is organized as follows. Section II presents some preliminaries. Section III gives our problem statement. Section IV elaborates on the details of our scheme. Section V presents the experiment results. Section VI discusses the related work. Section VII concludes the whole paper.

II. PRELIMINARIES

A. NOTATIONS

Given a matrix A, $\{A_i\}$ denotes the i^{th} row in A, and $\{A_{i,j}\}$ denotes the j^{th} element in the i^{th} row of A. Let $|A|$ denote the number of elements in the matrix A. Similarly, given a vector V, V_i denotes i^{th} element in V, and $|V|$ denotes number of elements in the vector V. The concatenation of a string a and a string b is denoted as $a \parallel b$. Let λ be a security parameter. We say that a function $\upsilon : \mathbb{N} \rightarrow \mathbb{N}$ is negligible in λ, if for any positive polynomial $p(\cdot)$ with sufficiently large λ, $\upsilon(\lambda) < \frac{1}{p(\lambda)}$. We take $negl(\lambda)$ as a negligible function in λ. Given a subset $S \subseteq \chi$ and a permutation $\varphi : \chi \rightarrow \chi$, $\varphi(S) = \{\varphi(i) \,|i \in S\}$ means each element $i \in S$ is replaced by $\varphi(i)$.

In addition, we introduce some notations related with searchable symmetric encryption. We informally treat a file as a set of keywords. Therefore, we write $w \in f$ to denote that a file f contains the keyword w. Given a file collection F, we order the files to uniquely identify a file by specifying f_i for $i \in [1, n]$ and order the total keywords W by specifying w_i for $i \in [1, m]$ after removing the duplicate keywords. For the i-th query, we denote T_j as the encryption of the queried keyword w_{i_j}. Here w_{i_j} means that the queried keyword at the i-th query is the j-th one in W. Therefore, the set of tokens generated over a period of time is denoted as $T = \{T_1, \cdots, T_q\}$. The search result for a keyword w is denoted as $C(w)$, which refers to a set of ciphertexts C of files that contain the query keyword w.

B. CHAMELEON HASHING

A chameleon hashing function is a collision-resistant hashing function with a key pair (sk, hk) [15]. The key hk is used to compute the hash value for a message, while the key sk can be used to find a collision for that message. Formally, a chameleon hashing function consists of the following algorithms:

$ParamGen(1^{\lambda})$: This algorithm takes as input a security parameter λ and outputs the system parameters SP.

$KGen(SP)$: This algorithm takes as input the system parameters SP and outputs a key pair (sk, hk).

$CH_H\left(hk, m, r\right)$: This algorithm takes as input the key hk, a message m, and a random integer r, and outputs the hash value $h = CH_H\left(hk, m, r\right)$.

$CH_F\left(sk, m_1, r_1, m_2\right)$: This algorithm takes as input the secret key sk, a message m_1, a random integer r_1, and another message m_2, and outputs another integer r_2 that makes the following equation hold:

$$CH_H\left(hk, m_1, r_1\right) = CH_H\left(hk, m_2, r_2\right)$$

A chameleon hashing function satisfies the following two properties:

- Collision resistance: Without the key sk, there exist no efficient algorithms for finding collision.
- Semantic security: For all pairs of message m_1 and m_2, the random hash values $CH_H\left(hk, m_1, r_1\right)$ and $CH_H\left(hk, m_2, r_2\right)$ are computationally indistinguishable.

C. INDISTINGUISHABILITY OBFUSCATION

An indistinguishability obfuscator for a circuit class $\{Cir_\lambda\}$ is a PPT uniform algorithm satisfying the following conditions [16], [17]:

- $i\mathcal{O}\left(\lambda, Cir\right)$ preserves the functionality of Cir. That is, for any $Cir \in Cir_\lambda$, if we compute $Cir^{'} = i\mathcal{O}\left(\lambda, Cir\right)$, then $Cir^{'}\left(x\right) = Cir\left(x\right)$ for all inputs x.
- For any λ and any two circuits $Cir_0, Cir_1 \in Cir_\lambda$ with the same functionality, the circuits $i\mathcal{O}\left(\lambda, Cir\right)$ and $i\mathcal{O}\left(\lambda, Cir^{'}\right)$ are indistinguishable.

III. PROBLEM STATEMENT

A. SEARCHABLE SYMMETRIC ENCRYPTION DEFINITION

A SSE scheme is a collection of six polynomial-time algorithms:

KeyGen $\left(1^\lambda\right)$: This key generation algorithm takes as input a security parameter λ and outputs the secret key material K.

IndexGen $\left(K, F\right)$: This index building algorithm takes as input the secret key material K and the file collection F, and outputs an inverted index matrix γ.

Token $\left(K, w\right)$: This token generation algorithm takes as input the secret key material K and the query keyword w, and outputs a search token T.

Search $\left(T, \gamma, EDB\right)$: This search algorithm takes as input a search token T, the encrypted index γ, and the encrypted database EDB, and outputs all the encrypted files containing the query keyword, i.e., $C\left(w\right)$.

Enc $\left(K, F\right)$: This encryption algorithm takes as input the secret key material K, the file collection $F = \{f_1, \cdots, f_n\}$, and outputs an encrypted database $EDB = \{i, c_i\}$, where $i \in [1, n]$, and c_i is the ciphertext of file f_i.

Dec $\left(K, c_i\right)$: This decryption algorithm takes as input the secret key material K, a ciphertext c_i and outputs a file f_i.

A SSE scheme is correct, if and only if for all key material K output by *KeyGen*, encrypted index γ output by *IndexGen*, encrypted database EDB output by *Enc*, and any $w \in W$, the following equation should hold:

$$Search\left(Token\left(K, w\right), \gamma, EDB\right) = C\left(w\right)$$

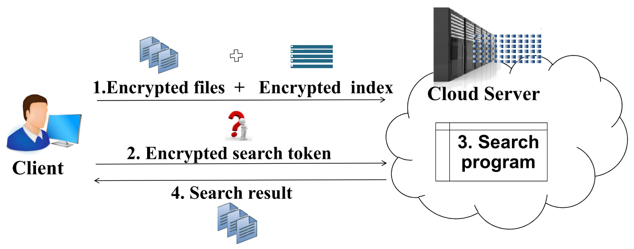

1.Encrypted files + Encrypted index
Cloud Server
Client
2. Encrypted search token
3. Search program
4. Search result

FIGURE 1. Basic paradigm of applying SSE to encrypted search.

and $w \notin Dec\left(K, C\right) \backslash Dec\left(K, C\left(w\right)\right)$ i.e., w should not be contained in the files which are not contained in the search result $C\left(w\right)$.

We now introduce the leakage definitions in SSE. Before giving the leakage definition, we introduce some auxiliary notions which are used to define the leakages. We first define a history which records the interaction between the client and the cloud server. Since the entity in each interaction phase mainly includes a file collection and a sequence of keywords submitted to *Token* and both of them need to be concealed, we use both of them to form the history.

Definition 1 (History): Let F be a file collection. A q-query history over F is a tuple $H = \left(F, \mathbf{w}\right)$ which indicates the file collection with a q-query word vector $\mathbf{w} = \left(w_{i_1}, \cdots, w_{i_q}\right)$

Definition 2 (Access Pattern): Let F be a file collection. The access pattern is a tuple $AP\left(H\right) = \left(F\left(w_{i_1}\right), \cdots, F\left(w_{i_q}\right)\right)$ based on the history $H = \left(F, \mathbf{w}\right)$.

Definition 3 (Search Pattern): Let F be a file collection. The search pattern, based on a q-query history $H = \left(F, \mathbf{w}\right)$, is a symmetric binary matrix $SP\left(H\right)$. For $1 \leq i, j \leq q$, the element in the i^{th} row and j^{th} column is 1, if $w_i = w_j$, and 0, otherwise.

Let F be a file collection. We define the leakage function $\mathfrak{L}$ that describes the leakages in building the index and searching over index based on the history $H = \left(F, \mathbf{w}\right)$.

Definition 4 (Leakages): $\mathfrak{L}_1\left(\gamma, F\right)$: With as input the index γ and the file collection F, the leakage function outputs the $|W|$, $|F|$ and $|f_i|$. $\mathfrak{L}_2\left(\gamma, C, T\right)$: With as input the encrypted index γ, the ciphertext collection C and a sequence of tokens T, the leakage function outputs the search pattern $SP\left(H\right)$ and the access pattern $AP\left(H\right)$.

B. SYSTEM MODEL

The basic paradigm of applying SSE to enable search over encrypted data is shown in Fig. 1. Typically there are two types of entities, *client* and *cloud server*. The client has a set of files to be outsourced to the cloud server. However, the client does not want the cloud server to know the content of the files. Therefore, in a setup phase, the client encrypts the files and produces an encrypted index supporting search by some secret key material K. Then, the client outsources the ciphertext collection C and an encrypted index γ to the cloud server. To search the encrypted outsourced files, the client generates search queries by calling a token generation function which takes as input the secret key material K and a keyword w, and outputs a search token T. Upon receiving the search token T, the cloud server conducts search based on the token and the encrypted index γ, and returns the search result to the client, without knowing any information about either the content of files or the queried keyword. So, after search, the client receives a collection of encrypted files which contain the queried keyword w, and performs decryption to obtain the matched files.

C. THREAT MODEL

Consistent with most of existing works on encrypted search (e.g., [5], [18], [19] to just list a few), we consider the cloud server as a semi-honest adversary. This means that the cloud server honestly follows the designated operations in SSE, yet tries to infer the private file content and queried keywords of the client, based on what it observes along the workflow. In particular, the cloud server may exploit the access pattern leakage and search pattern leakage which are inherent in most of existing SSE constructions. Considering that attacks based on the access pattern leakage can be well mitigated by directly introducing dummy data points [9], we will focus on the threat posed by exploiting the search pattern leakage. In particular, the adversary observes a set of q tokens $T = \{T_1, \cdots, T_q\}$ submitted by the client, and attempts to recover the queried keywords. It has been shown in prior works [12] that if the adversary observes search tokens generated through some deterministic algorithm, the adversary is able to recover the plaintext keywords based on auxiliary information like public query logs. Therefore, we aim to deliver a new SSE construction in which tokens are not generated in a deterministic way, so that the search pattern can hardly be exploitable for potential attacks.

IV. THE PROPOSED SCHEME

In this section, we propose a new SSE scheme, which enables search over encrypted cloud data without disclosing the search pattern. In our proposed scheme, the generation of secure query tokens is performed in a random way, so that even the same keywords will map to different secure tokens. So, tokens are not generated deterministically and the adversary is not able to observe the search pattern, i.e. whether a keyword has been searched before. Then, at the cloud side, we devise a mechanism which can securely map the randomized tokens to deterministic ones for search. Finally, the search result is returned to the client.

Our scheme uniquely bridges together the techniques of chameleon hashing and indistinguishability obfuscation ($i\mathcal{O}$). At a high level, we mainly rely on the chameleon hashing technique for generating randomized tokens and the $i\mathcal{O}$ technique for securely mapping the randomized tokens to deterministic ones at the cloud side for encrypted search. In particular, our construction mainly includes two phases. The first phase is about how to build an encrypted searchable index, and the second phase is about how to generate the

Algorithm 1 Building the Encrypted Searchable Index

Input: Secret keys: (k_1, k_2, hk); File Collection: F.
Output: Encrypted searchable index: γ.

```
 1: for $i = 0$ to $m - 1$ do
 2:     for $j = 0$ to $n - 1$ do
 3:         if $f_j$ contains $w_i$ then
 4:             $\{\delta_{i,j}\} = 1$
 5:         else
 6:             $\{\delta_{i,j}\} = 0$
 7:         end if
 8:     end for
 9: end for
10: for $i = 0$ to $m - 1$ do
11:     $\pi_i = CH_H(hk, G(k_1, w_i), r_i)$
12: end for
13: for $i = 0$ to $m - 1$ do
14:     $\delta_i^{'} = Enc(CH_H(hk, G(k_2, w_i), r_i), \delta_i)$
15: end for
16: $\gamma_i = \{\pi_{\varphi(i)}, \delta_{\varphi(i)}^{'}\}$
```

tokens at the client side and perform search at the cloud side. Note that like most of existing searchable encryption designs (e.g., [5], [18], [19]), the encryption of files can be done by any standard encryption technique, e.g., AES, which is independent to the building of the encrypted index and the search procedure. So, we do not explicitly handle file encryption in our construction. In what follows, we give the details of our construction.

A. THE PROPOSED SCHEME

Phase 1: Algorithm 1 presents the details of building the encrypted index. For practical consideration, we follow the framework in [8] to build an encrypted index that supports keyword search with sub-linear search time. Specifically, we take each keyword as the primary key and associate each keyword with files by a binary vector. Each element in binary vector represents the relationship between a keyword and a file. We use "1" to represent that the file contains the keyword, and "0" otherwise. For each keyword w_i, the client encrypts it as $\pi_i = CH_H(hk, G(k_1, w_i), r_i)$, where k_1 is the private key for a pseudorandom funcion $G(\cdot)$ and r_i is a random number. Then, for each binary vector δ_i associated with the keyword w_i, the client encrypts it as $\delta_i^{'} = Enc(CH_H(hk, G(k_2, w_i), r_i), \delta_i)$, where k_2 is also a secret key and and $Enc(\cdot)$ is a symmetric-key encryption algorithm. At last, for strong protection, the client applies a permutation function to the rows in the encrypted index. In addition, the client also builds a $i\mathcal{O}$ program for later use in the search phase.

Phase 2: In this phase, given a query keyword w_i, the client generates the search token and sends it to the cloud for encrypted search. In order to generate randomized tokens even for the same keyword in each query, we take concatenation of the query keyword and a counter ct representing

Algorithm 2 Generating Secure Search Token

```
Input: Secret keys: $(k_1, k_2, sk)$; Query keyword: $w_i$;
    Counter: $ct_i$.
Output: Secure search token: $T$.
1: $t_1 = G(k_1, w_i \parallel ct_i)$
2: $r_i^1 = CH_F(sk, G(k_1, w_i), r_i, G(k_1, w_i \parallel ct_i))$
3: $t_2 = G(k_2, w_i \parallel ct_i)$
4: $r_i^2 = CH_F(sk, G(k_2, w_i), r_i, G(k_2, w_i \parallel ct_i))$
5: $T = \left(t_1, r_i^1, t_2, r_i^2\right)$
```

the search frequency of the keyword as the input in token generation algorithm. Algorithm 2 presents the details in token generation. Given the keyword w_i to be queried, the client first looks up the local memory to find the random number r_i, and uses it to generate two new random numbers $r_i^1 = CH_F(sk, G(k_1, w_i), r_i, G(k_1, w_i \parallel ct_i))$ and $r_i^2 = CH_F(sk, G(k_2, w_i), r_i, G(k_2, w_i \parallel ct_i))$ by using the chameleon collision-finding function, where sk is a secret key for chameleon hash function and $w_i \parallel ct_i$ is the concatenation of keyword w_i with a counter denoting its searching frequency. Then, the client forms the search token for the query keyword w_i as (t_1, r_i^1, t_2, r_i^2), where $t_1 = G(k_1, w_i \parallel ct_i)$ and $t_2 = G(k_2, w_i \parallel ct_i)$.

Upon receiving the search token, the cloud server then performs search based on the encrypted searchable index and a $i\mathcal{O}$ program. The $i\mathcal{O}$ program is used for obfuscating the search procedure. In particular, it is used to (i) securely map the random search token to the form that can be matched against the encrypted index, and (ii) hide which entry of the encrypted index is accessed during search. Throughout the search procedure, the cloud server is oblivious to the recovered token and also the entry of index that has been accessed. Note that the search pattern may also be leaked if the cloud server can observe which entries of the index have been accessed over time [12]. Therefore, our design achieves strong protection for the search pattern.

Algorithm 3 presents the details in searching encrypted data at the cloud side. The cloud server calls the $i\mathcal{O}$ program, and obtains the search result. Inside the $i\mathcal{O}$ program, the key hk is embedded as a constant in advance by the client. This $i\mathcal{O}$ program first uses the chameleon hash function to revert the hash value $HV_1 = CH_H\left(hk, t_1, r_i^1\right)$ and hash value $HV_2 = CH_H\left(hk, t_2, r_i^2\right)$. Here, $HV_1 = CH_H\left(hk, t_1, r_i^1\right)$ is the recovered token which is able to be matched against the encrypted index. Recall that the encrypted version of the keyword in the index is w_i is $CH_H(hk, G(k_1, w_i), r_i)$, and $CH_H(hk, G(k_1, w_i), r_i) = CH_H\left(hk, t_1, r_i^1\right)$ based on the property of the chameleon hash function. Therefore, the recovered token $HV_1 = CH_H\left(hk, t_1, r_i^1\right)$ can locate the right entry in the index. Subsequently, $HV_2 = CH_H\left(hk, t_2, r_i^2\right)$ is used to decrypt the corresponding $\delta_t^{'}$. Finally, the $i\mathcal{O}$ program outputs the recovered file identifiers and the cloud server returns the corresponding file ciphertexts to the client.

Algorithm 3 Searching Encrypted Data at the Cloud Side

```
Input: Token: $T = \left(t_1, r_i^1, t_2, r_i^2\right)$; Index: $\gamma$.
Output: Search result: $C(w_i)$.
1: Execute a $i\mathcal{O}$ program with inputs $T$ and $\gamma$
   $i\mathcal{O}$ program (with constant $hk$):
2: $HV_1 = CH_H\left(hk, t_1, r_i^1\right)$
3: $HV_2 = CH_H\left(hk, t_2, r_i^2\right)$
4: if $\pi_t = HV_1$ then
5:     Output $Dec\left(HV_2, \delta_t^{'}\right)$.
6: else
7:     Output $\perp$.
8: end if
```

Note that the security of $i\mathcal{O}$ ensures that the cloud server is oblivious to the search procedure and the embedded key hk is also kept confidential. So, the cloud server only obtains the search result through the search procedure. Recall that leakage from the search result is access pattern leakage and can be well mitigated by directly introducing dummy data points [9]. And our focus is on defending against search pattern leakage.

B. SECURITY ANALYSIS

We now provide formal proofs to show the security of our scheme. In particular, we follow the security framework in prior works [5], [12] for analysis. The security framework proposed therein is based on a leakage function $\mathfrak{L}$. While revealing the leakage function to an adversary, the security framework ensures that the adversary should not learn any further information beyond the leakage function itself when the adversary observes a sequence of tokens submitted adaptively. Since our proposed construction doest not disclose the search pattern, we modify the security framework to fit our design. The leakage function $\mathfrak{L} = \{\mathfrak{L}_1, \mathfrak{L}_2\}$ in our proposed scehem is given as follows:

$$\mathfrak{L}_1 = (|W|, |F|, |f_i|)$$
$$\mathfrak{L}_2 = AP(H)$$

Definition 5: Let $\Pi = (KGen, Build, Search)$ be our secure index-based search scheme which uses chameleon hashing to generate secure search tokens and relies on $i\mathcal{O}$ for cloud-side secure search. Given the leakage function $\mathfrak{L}$, we define the following experiments with an adversary $\mathcal{A}$ and a simulator $\mathcal{S}$.

$Real_A(\lambda)$: The client runs KGen to generate the private keys. The adversary $\mathcal{A}$ selects a set of files, and asks the client to generate the index, the $i\mathcal{O}$ program, and the ciphertexts via the algorithm Build. Then $\mathcal{A}$ performs a polynomial number of adaptive q queries, and asks the client for the secure search tokens and the resulting file ciphertexts via the algorithm Search. Finally, $\mathcal{A}$ produces a bit as the output.

$Sim_{A,S}(\lambda)$: The adversary $\mathcal{A}$ selects a set of files, and $\mathcal{S}$ simulates an index, an $i\mathcal{O}$ program, and the ciphertexts for $\mathcal{A}$ based on $\mathfrak{L}_1$. Then, $\mathcal{A}$ performs a polynomial number of

adaptive q queries. From $\mathfrak{L}_2$, $\mathcal{S}$ returns simulated tokens and file ciphertexts. Finally, $\mathcal{A}$ produces a bit as the output.

We say that Π is $\mathfrak{L}$-secure against adaptive chosen keyword attacks if for all polynomial-time adversaries $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that $|\Pr[Real_{\mathcal{A}}(\lambda) = 1] - \Pr[Sim_{\mathcal{A},\mathcal{S}}(\lambda) = 1]| \leq negl(\lambda)$, where $negl(\lambda)$ is a negligible function in λ.

We now prove that our proposed scheme is secure against adaptive chosen keywords attacks with respect to the characterized leakages.

Theorem 1: The proposed encrypted search scheme Π is $\mathfrak{L}$-secure against against adaptive chosen-keyword attacks, if φ is a secure pseudorandom permutation, CH is a secure hash function, iOSearch is a secure indistinguishability obfuscator, and Enc is a PRF-based encryption.

Proof: Given the leakage $\mathfrak{L}_1$, the simulator $\mathcal{S}$ generates the simulated index γ', which is indistinguishable from the real index γ. In particular, the real index γ and the simulated index γ have the same size. The bit length of a real index entry and simulated one is the same. However, $\mathcal{S}$ generates random bit strings for each entry. For the file ciphertexts, $\mathcal{S}$ also generates random bit strings with the same size as the file ciphertexts. In addition, $\mathcal{S}$ generates the $i\mathcal{O}$ program *iOSearch*$'$, which is indistinguishable from *iOSearch*.

Now, we need to show how to simulate q adaptive queries $\{Q_i\}_{i=1}^{q}$ made by the adversary. For each query Q_i, the simulator responds with the simulated search token T^*, where T^* is a 4-tuple of random bit strings, i.e., $T^* = (t_1^*, r^{1*}, t_1^*, r^{2*})$. Then, $\mathcal{S}$ operates the chameleon hashing function CH (inside the *iOSearch*$'$ program) as a random oracle to first randomly point at an entry of the simulated index, so $HV_1 = CH_H\left(hk^*, t_1^*, r^{1*}\right)$. Meanwhile, the simulator sets HV_2 to $CH_H\left(hk^*, t_2^*, r^{2*}\right)$. Then, $\mathcal{S}$ operates a random oracle so that $Dec(HV_2, \delta') = \delta$, where δ is given by the leakage $\mathfrak{L}_2$. This indicates the search result is identical to that from searching the real index. We note that this can be achieved if the encryption algorithm Enc is instantiated as $(r, \mathcal{H}(HV_2||r) \oplus \delta)$, where r is a random bit string and $\mathcal{H}$ is a random oracle. This is actually a widely adopted technique for achieving the adaptive security for SSE [19].

The above analysis shows that the real index and the simulated one, the real tokens and the simulated ones, and the encrypted files and the simulated ciphertexts are computationally indistinguishable. Meanwhile, the search results from the real index and the simulated one are identical, so they are indistinguishable as well. Therefore, it can be concluded that the adversary is unable to distinguish the outputs of the real experiment and the ideal experiment. $\square$

V. EXPERIMENTS

A. EXPERIMENTS SETUP

All experiments were conducted on a cluster and a desktop with 16 cores and 64 GB of memory running Unbantu version 16.04. The cluster contains 1 server as master and 3 servers as slaves. Each slave has 40 cores and 400GB of memory running Linux version CentOS 7.3. We use python version 2.7.6, jdk version 1.8, spark version 2.1.0, HBase version 1.2.5, and Hadoop version 2.7.3. The dataset we use in our experiments is the Enron email dataset,[1] which is collected and prepared by the CALO Project. It contains 150 different users' email files including sent emails, contacts, delete items, etc. We choose three sub-datasets from all the email files of all the users as the experiments datasets. The first dataset contains 10,000 documents. The second dataset contains 100,000 documents. The third set contains all the documents in Enron dataset. Hereafter, for ease of presentation, we will refer to these three datasets as dataset I, dataset II, and dataset III, respectively. In the experiments, we develop a distributed storage system to implement our proposed scheme. Note that we implement the index setup phase on the cluster side, as the number of the email files is huge and it is a bit hard to use desktop to handle the full dataset which has 517, 401 files and 348,935 keywords after removing the stop word.

B. SYSTEM IMPLEMENTATION

1) STORAGE SYSTEM

Storage system is written in Java and is implemented by HBase. We choose HBase for the following reasons. Firstly, it is too large to store the encrypted index and the Enron dataset centrally. HBase however partitions the data into regions controlled by a cluster of RegionServer's. Secondly, when searching over the index, if we store index centrally, it will take $O(m)$ to do text matching operation. When using HBase, however, it takes far less time than that to do text matching. This is mainly due to the addressing mode with 3 levels in HBase (i.e., the root table, the meta table, and region) and the sorting method based on lexicographic order between records. Thirdly, when user submits multi-keyword to cloud, HBase can deal with each token Simultaneously.

2) FILE ENCRYPTION/DECRYPTION

Since the security of file encryption does not affect the security of the token, its security issue is not within our scope. To highlight our important issue, we only use ECB encryption with PKCS7 padding mode in the jar of javax.crypto.Cipher to encrypt the files.

3) INDEX SETUP

Index setup is written in Java with javax.crypto library and JPBC library and runs on Spark to build an inverted index to speed up the document retrieval. We choose Spark for the following reasons. For one hand, we need to pre-process each email file to parse each keyword and remove the stop word. Fortunately, Spark can do these operation with each file in parallel. Therefore, it can dramatically accelerate the index setup phase. For another, since we need to build the index iteratively (i.e., the result of inserting the keywords of the first email file into the index will be used as a reference for the insertion of the keywords of the second email file.)

[1]Enron Email Dataset: http://www.cs.cmu.edu/∼./enron/

Algorithm 4 An Instance of Chameleon Hashing Function

```
Input: Message: $m$; Random number: $r$; Key: $hk = \{g, h = g^x\}$.
Output: Chameleon hash result: $H$.
1: $H = g^m h^r$.
```

Algorithm 5 An Instance of Chameleon Collision-Finding Function

```
Input: Message: $m_1, m_2$; Random number:$r_1$; Key: $sk = x$.
Output: Chameleon collision: $r_2$.
1: $r_2 = log_h\, g^{m_1} h^{r_1} / x g^{m_2}$.
```

and Spark can store the intermediate results in the memory, the setup phase can be accelerated by using Spark. We set the cluster with 1 server as master and 3 servers as workers. For index setup job, we set it with 9 executers and each executer has 5 cores and 20G of memory. We use a pair of RowKey/Data to store the inverted index. "RowKey" stores the encrypted keyword and "value" in "Data" stores an encrypted "binary vector" (i.e., the relationship between keyword with file collection). It first parses each word with space in file collection and used Stanford's database to remove the stop word to establish the inverted index in bitmap style. Then, in order to generate secure index, we use AES encryption with chameleon hash function to encrypt "RowKey" and to generate the private key for symmetric encryption used to encrypt the "Data". The construction of chameleon hash we used is the one in [15], as shown in Algorithm 4 and 5. Since the client generates the token by CH_F function which needs to take $G(w)$ and its corresponding r as input to compute the collision, we need to store the relationship between $G(w_i)$ and r_i at the client side when building an index, where $i = 1, \cdots, m$. It is worth noting that to reduce the storage size at the client side we use the same r_i in both $CH_H(G(k_1, w_i), r_i)$ and $CH_H(G(k_2, w_i), r_i)$. Therefore, we only need to store the relationship between each keyword and r_i. In order to accelerate searching for r_i in token generation phase, we use hash table to store their relationship. Last, it encrypts the "value" with a symmetric encryption AES which takes the "binary vector" as the input message and $CH_H(G(k_2, w_i), r_i)$ as the secret key. To prevent the adversary from looking up the hash value dictionary to get the private key k_1, k_2, we use "salt" in secret key generation for the above mentioned encryptions.

4) TOKEN GENERATION

Token generation is written in java with javax.crypto.Cipher to generate AES encrytion function and jpbc library to generate cyclic group and runs on desktop to generate token. It firstly combines the submitted keyword w_i with the retrieval times ct. Then, it uses AES to encrypt string concatenation $G(w_i \parallel ct)$ with different key k_1 and k_2 and searches the local hash table with pairs of (w, r) to find the r_i corresponding to w_i. And then, it uses CH_F to forge the random r_i^1 with the private key x, the concatenation cipher $G(k_1, w_i \parallel ct)$ and a set of data $(G(k_1, w_i), r_i)$. Similarly, it also forges the random r_i^2 with the same private key x, the concatenation cipher $G(k_2, w_i \parallel ct)$ and a set of data $(G(k_2, w_i), r_i)$. Finally, it combines the output of step 2 with $G(k_1, w_i \parallel ct)$ and $G(k_2, w_i \parallel ct)$ to generate the token $T = \left(G(k_1, w_i \parallel ct), r_i^1, G(k_2, w_i \parallel ct), r_i^2\right)$.

5) RETRIEVAL

Retrieval is written in Java and Python and performed over HBase. It retrieves the files containing the keyword in token. We set the cluster for 1 HMaster and 3 HRegionServer. Each HRegionServer has 40 cores and 400G of memory. It first computes the hash value HV_1 by inputting $\left(G(k_1, w_i \parallel ct), r_i^1\right)$ to chameleon hash function. It then looks up the Zookeeper to find the region containing the "RowKey" corresponding to HV_1. Last, it computes the hash value HV_2 to decrypt the corresponding "value" and returns the file ID which column is "1". The above retrieval procedure should be protected by $i\mathcal{O}$ in our design.

Since $i\mathcal{O}$ currently is still in theoretical stage, we will do the first experiment without *iOSearch* function for demonstration purpose. Then we realize the $i\mathcal{O}$ function [20] only with AND circuit to demonstrate the efficiency of the $i\mathcal{O}$ function and provide some discussion.

The obfuscator implementation consists of the following modules: (1) create a universal circuit for the target functionality, (2) transform the boolean circuit to a branching program according to Barrington's theorem, (3) convert the branching program into a block-diagonal matrix $P_{i,b}$ which pads the branching program to l length, where $l < m + t$ and extent each matrix with $4l$ uniformly random matrices and one uniformly random vector (4) apply multiplicative bundling with $l + 1$ uniformly random non-singular matrices and $2l + 2$uniformly random non-zero scalars, (5) create randomized branching programs by the first step of Kilian's protocol, (6) instantiate multilinear map, (7) encode for multilinear map, (8) circuit encoder into input for universal circuit, (9) transform the input with an input-selection function, (10) zero testing, and (11) post zero testing. Since the most important step proposed is to fix the matrix branching program to $P_{i,b}$ [21], we do not specify the implementation details here except for the third module.

To implement the third module, the obfuscator O is instantiated with two parameters, $t = t(n, \lambda)$ and $s = s(n, \lambda)$ according to Assumption 1 in [20]. Then taking a dual-input matrix branching program BP of length m, width w, and input length n as consideration, O pads BP with identity matrices, if the length of BP is smaller than $l < t + m$. Therefore, it needs to select t elements from the input to make a branching program transformed into one with this input selection. Last, O extends the matrices to $P_{i,b}$ which selects $4l$ uniformly random matrices $Z_{i,b} \in \mathbb{Z}_p^{s \times s}$and one uniformly random vector $Z_{l+1} \in \mathbb{Z}_p^{s \times 1}$. Since we cannot determine the value of t and s, we set t as that in [22] and s as w to ensure the correctness. Since our $i\mathcal{O}$ implementation only fixes the third module based on that of [21], we only test the branching program generation in this experiment.

C. EXPERIMENTS RESULTS

We now present experiments results regarding performance evaluation of our design. For the simplified experiments,i.e., without $i\mathcal{O}$ protection, we first compare index generation time with above mentioned three datasets to illustrate how the size of dataset influences the index generation time. Considering that the index should be stored on the cloud server and the client needs to pay for the storage space, we also test the size of the encrypted index. And then since the token generation time should be one of the most important part, which influences the experience of the client, we compare token generation time with different datasets to compare the token generation times (It is because we need to search r_i in the hash table with pairs of (w_i, r_i) when generating the token). In addition, we compare the search time (without $i\mathcal{O}$) for different datasets for demonstration purpose. For the $i\mathcal{O}$ implementation, we compute the generation time, file size, memory usage and length of branching program for generating a branching program with only AND gate, so as to demonstrate the efficiency of $i\mathcal{O}$. The reason why we choose generation time and memory usage is that we need to estimate the cost for generating branching program. Since it needs to be generated at the client side, if it takes a long time or takes too much computation resources, the client would have to endure for a longer period of time to generate branching program. And since the branching program should be outsourced to the cloud server, the file size should also be taken into consideration. In addition to that, the running *iOsearch* time is correlated to the length of branching program. Therefore, we test it for estimation. It is worth mentioning that the experiment result of the index setup in our paper is the average result of 10 times executions, and in the token generation phase, for each dataset, we randomly select 10% keywords in total keywords collection to generate token and send all the tokens to do the search. We take the average of token generation and search time in each dataset as the experiment results.

1) INDEX SETUP COST

We first present the cost in building an encrypted index. It includes the index setup time and index size. In order to illustrate the relationship between the number of files and the cost in building the index, we compare each time cost and storage cost under the above mentioned three datasets in the Table 1. Table 1 shows the results of index setup time and index size based on three datasets. In principle, the index setup time is mainly dependent on the number of files and the encryption costs. In our experiment, since we use AES with chameleon hash CH_H to encrypt each "RowKey" and "Data", the setup time increases as the number of the files increases. And index size completely depends on the number

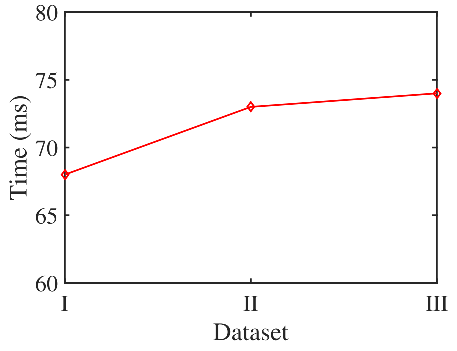


FIGURE 2. Token generation cost.

TABLE 1. Cost of the encrypted index.

Dataset	Setup time (s)	Index size (MB)
I	121.818	8.4
II	1123.656	120
III	1638.115	584

of files and the number of keywords in dataset, the theoretical spatial complexity is $O(mn)$. The experiment results fit the theoretical spatial complexity. The index size increases when the number of files and number of keyword increase. It is worth noting that the index setup phase is a one-time cost.

2) TOKEN GENERATION COST

We present token generation time with different datasets in Fig. 2. Each plot shows the relationship between each dataset with token generation time. Since we use AES to encrypt the keyword and the length of each keyword submitted to *Token* algorithm is less than 128 bits, the generation time of $G(k_1, w_i \parallel ct)$ and $G(k_2, w_i \parallel ct)$ will be almost the same with different keyword submitted to token algorithm each time. And since the calculation of the chameleon collision function also takes almost the same time in each time to generate r_i^1 and r_i^2 (The reason is similar to the reason for assuming almost the same generation time of $G(k_1, w_i \parallel ct)$.), the most important factor affecting token generation time is searching r_i corresponding to the searched keyword from the table with the pairs of (w_i, r_i). Since we use hash table structure to store the table, the generation time with keywords is almost the same. In order to accelerate the token generation time, we store the table with the pairs of (w_i, r_i) in the memory. Therefore, the token generation time is about 87ms.

3) SEARCH COST WITHOUT $i\mathcal{O}$

As mentioned before, since $i\mathcal{O}$ currently is still in theoretical stage, we will first evaluate the search cost without the *iOSearch* function for demonstration purpose. We present the search cost in Fig. 3. From the experiments results, we can see that the search cost is close among three datasets, which is less than 270 ms for all datasets.

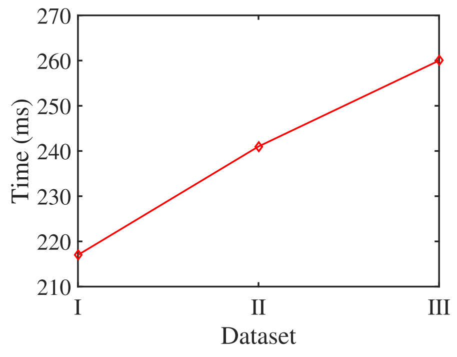


FIGURE 3. Search cost without *iO*.

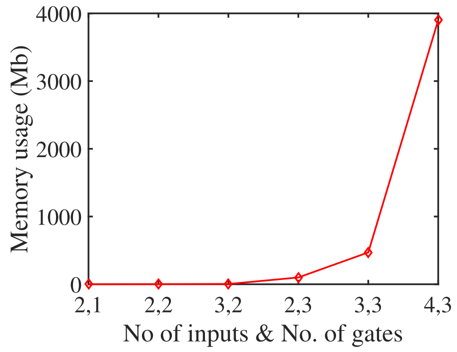


FIGURE 4. Memory usage in branching program generation.

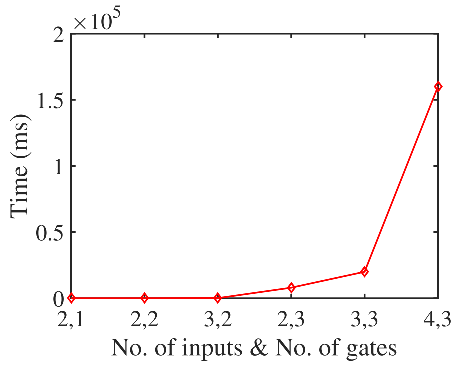


FIGURE 5. Generation time for building a branching program.

4) *iO* BRANCHING PROGRAM GENERATION COST

To demonstrate the resource usage in branching program generation of *iO*, we evaluate the relationship between the resource usage and the parameters in generating a branching program, including the number of inputs and the number of gates of obfuscated circuit. We show the resource usage including memory usage in Fig. 4 and generation time in Fig. 5. A point at location (x_1, x_2, y) in Fig. 4 indicates that it needs y memory to transfer x_1 inputs and x_2 gates into a branching program. A point at location (x_1, x_2, y) in Fig. 5 indicates that the time of transforming x_1 inputs and x_2 gates into a branching program is y. In addition to that, we count the length of branching program and the size of the branching program file with each input. The length of branching program is shown in Fig. 6 and the file size is shown

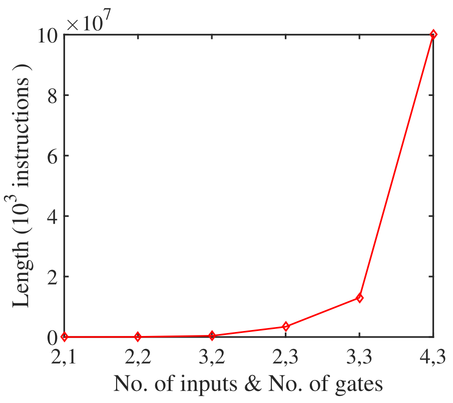


FIGURE 6. Length of generated branching programs.

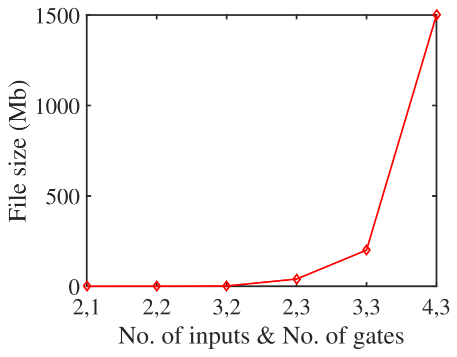


FIGURE 7. File size of a branching program.

in Fig. 7. A point at location (x_1, x_2, y) in Fig. 6 indicates that transforming x_1 inputs and x_2 gates will generate a branching program with the length y. A point at location (x_1, x_2, y) in figure 7 indicates that transforming x_1-length of input and x_2 gates program into a branching program will generate a file with size y. We estimate the length of generating a branching program by applying recursive formula in [21]. Increasing the number of inputs causes a linear increase in each measured output of the experiment. Increasing the number of gates however causes an exponential increase in each measurement. It is because Barrington's theorem tells that every language on $\{0, 1\}$ can be recognized by a family of exponential width and linear length. In addition to that, we could not measure the memory usage reliably due to technical limitations of our memory profiler. We estimate that the memory usage is around a fixed increment of the file size according to [21], since we use the compression algorithm to store BPs.

5) DISCUSSION

Similar to prior works which use *iO* as underlying cryptographic techniques in different application domains (e.g., [23], [24], to just list a few), our design for the time being is not immediately applicable in practice due to the current inefficiency of *iO*. Although program generation and evaluation of *iO* is not fast now, research endeavors on making *iO* is actively ongoing. Hence, it can be envisioned that

$i\mathcal{O}$ with practical performance would probably be achieved in the not-too-distant future. Clearly, the practical efficiency improvement of $i\mathcal{O}$ will directly lead to the applicability of our design.

VI. RELATED WORK

A. SECURE SEARCHABLE ENCRYPTION

Our work is closely related to a research area known as SSE, which studies how to conduct secure search over encrypted data efficiently. We are also aware that public key searchable encryption (PKSE) techniques also support search over encrypted data, yet they are usually very computationally expensive [4]. In terms of efficiency, SSE techniques fits much better into practical realm. So far, a lot of elegant SSE constructions have been proposed (e.g., [5], [7], [25], [26], to just list a few). For performance efficiency, SSE constructions are generally built under the well-known security definitions [5], [6] which allow access pattern leakage and search pattern leakage. In recent years, however, it has been shown that the leakages of access patter and search pattern can be exploited to attack SSE schemes (e.g., [8]–[12]).

While attacks based on the access pattern leakage can be well mitigated by introducing dummy points to the data set and encrypted index [8], [9], hiding the search pattern is more challenging and little work has been done. In [12], Liu *et al.* propose an approach based on the idea of adding some fake query tokens to the real query token. This approach, however, requires heuristic parameter tuning for the number of fake query tokens for strong security, which might not be easy to operate in practice. Secondly, it undesirably brings extra workload to both the user and the cloud server in each query. In [13], Gajek propose to use constrained functional encryption to hide the search pattern. Since the proposed scheme relies on operations in bilinear group to encrypt the query keyword, it might suffer from zeroizing attacks [27]–[33]. Very recently, Li *et al.* [14] study privacy-preserving storage and retrieval in a multi-cloud setting. Their scheme protects the search pattern by relying on secret sharing technique and the limited collusion of multiple cloud service providers. In contrast to these works, our design works under the common setting of a single cloud server, is free of the use of bilinear map, and does not need multiple fake tokens for hiding a real token.

B. INDISTINGUISHABILITY OBFUSCATION

The $i\mathcal{O}$ technique requires that given any two equivalent circuits C_0 and C_1 of similar size, the obfuscations of C_0 and C_1 should be computationally indistinguishable. Barak *et al.* [34] first propose indistinguishability obfuscation ($i\mathcal{O}$) which requires that the two indistinguishable programs have the same size and the same functionality. Recently, Garg *et al.* [16] propose the first candidate construction of an efficient $i\mathcal{O}$ for general circuits. In [35], Apon *et al.* give an implementation of $i\mathcal{O}$ by using the approach in [36] to compile boolean formulas into branching program. So far, many applications based on $i\mathcal{O}$ have been proposed, e.g., multi-party key exchange [37] and deniable encryption [17]. The above mentioned candidate constructions are based on multilinear maps [16], which unfortunately suffer from "zeroizing" attacks, as shown by prior works [27]–[33], [38], [39]. Very recently, Garg *et al.* [20] propose an $i\mathcal{O}$ candidate construction for general programs, which can resist all known polynomial-time attacks. This $i\mathcal{O}$ construction is used in our proposed scheme to achieve strong security.

VII. CONCLUSION

In this paper, we propose a new SSE scheme which can hide the search pattern of the client in searching encrypted cloud data. We leverage the chameleon hashing technique to generate secure search tokens in a randomized way, so for the same keyword the search tokens at different queries are different. On the cloud side, we rely on the $i\mathcal{O}$ technique to securely map the randomly generated search token to the deterministic one for effective encrypted search. We justify the security guarantees of our scheme via rigorous security proofs. In order to demonstrate the performance, we also conduct extensive experiments for evaluation. The performance of our scheme relies on the underlying cryptographic technique $i\mathcal{O}$. Although currently the $i\mathcal{O}$ technique is inefficient, related research endeavors on practical $i\mathcal{O}$ are actively ongoing [40]. The practical performance improvement of $i\mathcal{O}$ will directly lead to the improvement and applicability of our scheme. We emphasize that our scheme presents a new attempt that pushes forward the research on SSE with concealed search pattern.

REFERENCES

[1] K. Liang, C. Su, J. Chen, and J. K. Liu, "Efficient multi-function data sharing and searching mechanism for cloud-based encrypted data," in *Proc. ACM ASIACCS*, 2016, pp. 83–94.

[2] J. Hughes. (2014). *Data Breaches in the cloud: Who's responsible?* [Online]. Available: http://www.govtech.com/security/Data-Breaches-in-the-Cloud-Whos-Responsible.html

[3] C. Bösch, P. H. Hartel, W. Jonker, and A. Peter, "A survey of provably secure searchable encryption," *ACM Comput. Surv.*, vol. 47, no. 2, pp. 18:1–18:51, 2014.

[4] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, "Privacy-preserving multi-keyword ranked search over encrypted cloud data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 1, pp. 222–233, Jan. 2014.

[5] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," in *Proc. ACM CCS*, 2006, pp. 1–10.

[6] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proc. ACM CCS*, 2012, pp. 956–976.

[7] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *Proc. CRYPTO*, 2013, pp. 353–373.

[8] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *Proc. NDSS*, 2012, pp. 1–12.

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

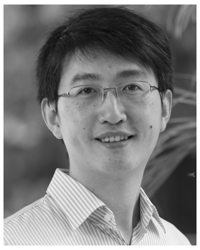
[10] M. Naveed, S. Kamara, and C. V. Wright, "Inference attacks on property-preserving encrypted databases," in *Proc. ACM CCS*, 2015, pp. 644–655.

[11] D. Pouliot and C. V. Wright, "The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption," in *Proc. ACM CCS*, 2016, pp. 1341–1352.

[12] C. Liu, L. Zhu, M. Wang, and Y.-A. Tan, "Search pattern leakage in searchable encryption: Attacks and new construction," *Inf. Sci.*, vol. 265, pp. 176–188, May 2014.
[13] S. Gajek, "Dynamic symmetric searchable encryption from constrained functional encryption," in *Proc. Cryptograph. Track RSA Conf.*, 2016, pp. 75–89.
[14] J. Li, D. Lin, A. C. Squicciarini, J. Li, and C. Jia, "Towards privacy-preserving storage and retrieval in multiple clouds," *IEEE Trans. Cloud Comput.*, vol. 5, no. 3, pp. 499–509, Jul. 2017.
[15] H. Krawczyk and T. Rabin, "Chameleon signatures," in *Proc. NDSS*, 2000, pp. 1–12.
[16] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters, "Candidate indistinguishability obfuscation and functional encryption for all circuits," in *Proc. FOCS*, Oct. 2013, pp. 40–49.
[17] A. Sahai and B. Waters, "How to use indistinguishability obfuscation: Deniable encryption, and more," in *Proc. ACM TC*, 2014, pp. 475–484.
[18] C. Wang, N. Cao, K. Ren, and W. Lou, "Enabling secure and efficient ranked keyword search over outsourced cloud data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 8, pp. 1467–1479, Aug. 2012.
[19] X. Yuan, H. Cui, X. Wang, and C. Wang, "Enabling privacy-assured similarity retrieval over millions of encrypted records," in *Proc. ESORICS*, 2015, pp. 40–60.
[20] S. Garg, E. Miles, P. Mukherjee, A. Sahai, A. Srinivasan, and M. Zhandry, "Secure obfuscation in a weak multilinear map model," in *Proc. Theory Cryptograph. Conf.*, 2016, pp. 241–268.
[21] S. Banescu, M. Ochoa, N. Kunze, and A. Pretschner, "Idea: Benchmarking indistinguishability obfuscation—A candidate implementation," in *Proc. Int. Symp. Eng. Sec. Softw. Syst.*, 2015, pp. 149–156.
[22] S. Badrinarayanan, E. Miles, A. Sahai, and M. Zhandry, "Post-zeroizing obfuscation: New mathematical tools, and the case of evasive circuits," in *Proc. EUROCRYPT*, 2016, pp. 764–791.
[23] C. Guan, K. Ren, F. Zhang, F. Kerschbaum, and J. Yu, "Symmetric-key based proofs of retrievability supporting public verification," in *Proc. ESORICS*, 2015, pp. 203–223.
[24] Y. Zhang, C. Xu, X. Liang, H. Li, Y. Mu, and X. Zhang, "Efficient public verification of data integrity for cloud storage systems from indistinguishability obfuscation," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 3, pp. 676–688, Mar. 2017.
[25] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *Proc. NDSS*, 2014, pp. 72–75.
[26] P. Golle, J. Staddon, and B. Waters, "Secure conjunctive keyword search over encrypted data," in *Proc. ACNS*, 2004, pp. 31–45.
[27] J. H. Cheon, K. Han, C. Lee, H. Ryu, and D. Stehlé, "Cryptanalysis of the multilinear map over the integers," in *Proc. EUROCRYPT*, 2015, pp. 3–12.
[28] J.-S. Coron *et al.*, "Zeroizing without low-level zeroes: New MMAP attacks and their limitations," in *Proc. Annu. Cryptol. Conf.*, 2015, pp. 247–266.
[29] Y. Hu and H. Jia, "Cryptanalysis of GGH map," in *Proc. EUROCRYPT*, 2016, pp. 537–565.
[30] Z. Brakerski, C. Gentry, S. Halevi, T. Lepoint, A. Sahai, and M. Tibouchi, "Cryptanalysis of the quadratic zero-testing of GGH," IACR Cryptol. ePrint Arch., Tech. Rep. 845/2015, 2015.
[31] S. Halevi, "Graded encoding, variations on a scheme," IACR Cryptol. ePrint Arch., Tech. Rep. 866/2015, 2015.
[32] J. H. Cheon, P.-A. Fouque, C. Lee, B. Minaud, and H. Ryu, "Cryptanalysis of the new CLT multilinear map over the integers," in *Proc. Annu. Int. Conf. Theory Appl. Cryptograph. Techn.*, 2016, pp. 509–536.
[33] B. Minaud and P.-A. Fouque, "Cryptanalysis of the new multilinear map over the integers," IACR Cryptol. ePrint Arch., Tech. Rep. 941/2015, 2015.
[34] B. Barak *et al.*, "On the (im) possibility of obfuscating programs," *J. ACM*, vol. 59, no. 2, p. 6, 2012.
[35] D. Apon, Y. Huang, J. Katz, and A. J. Malozemoff, "Implementing cryptographic program obfuscation," IACR Cryptol. ePrint Arch., Tech. Rep. 779/2014, 2014. [Online]. Available: https://eprint.iacr.org/
[36] M. Sauerhoff, I. Wegener, and R. Werchner, "Relating branching program size and formula size over the full binary basis," in *Proc. STACS*, 1999, pp. 57–67.
[37] D. Boneh and M. Zhandry, "Multiparty key exchange, efficient traitor tracing, and more from indistinguishability obfuscation," in *Proc. CRYPTO*, 2014, pp. 480–499.
[38] D. Boneh, D. J. Wu, and J. Zimmerman, "Immunizing multilinear maps against zeroizing attacks," IACR Cryptol. ePrint Arch., Tech. Rep. 930/2014, 2014.
[39] E. Miles, A. Sahai, and M. Zhandry, "Annihilation attacks for multilinear maps: Cryptanalysis of indistinguishability obfuscation over GGH13," in *Proc. Crypto*, 2016, pp. 629–658.
[40] P. Ananth, D. Gupta, Y. Ishai, and A. Sahai, "Optimizing obfuscation: Avoiding barrington's theorem," in *Proc. ACM CCS*, 2014, pp. 646–658.

JING YAO received the B.E. degree from the Xi'an University of Technology, Xi'an, China, in 2009, and the M.E. degree from Shaanxi Normal University, Xi'an, China, in 2012. She is currently pursuing the Ph.D. degree with the School of Electronic and Information Engineering, Xi'an Jiaotong University, and also with the Department of Computer Science, City University of Hong Kong, Hong Kong. Her research interests include cloud computing security, outsourcing storage security, and data privacy.

YIFENG ZHENG (S'16) received the B.E. degree in information engineering from the South China University of Technology, Guangzhou, China, in 2013. He is currently pursuing the Ph.D. degree with the Department of Computer Science, City University of Hong Kong, Hong Kong. From September 2013 to December 2013, he studied at Zhejiang University, Hangzhou, China. His research interests include cloud computing security and multimedia security.



CONG WANG (SM'17) received the B.E. degree in electronic information engineering and the M.E. degree in communication and information system from Wuhan University, China, and the Ph.D. degree in electrical and computer engineering from the Illinois Institute of Technology, USA. He has been an Assistant Professor with the Department of Computer Science, City University of Hong Kong, since 2012. His current research interests include data and computation outsourcing security in the context of cloud computing, network security in emerging Internet architecture, multimedia security and its applications, and privacy-enhancing technologies in the context of big data and IoT. His research has been supported by multiple government research fund agencies, including National Natural Science Foundation of China, Hong Kong Research Grants Council, and Hong Kong Innovation and Technology Commission. He was a recipient of the President's Awards from the City University of Hong Kong in 2016. He was a co-recipient of the Best Student Paper Award of the IEEE ICDCS 2017, the Best Paper Award of the IEEE MSN 2015, and CHINACOM 2009. He is a member of the ACM. He has been serving as the TPC co-chairs for a number of the IEEE conferences/workshops.

XIAOLIN GUI received the B.E., M.E., and Ph.D. degrees from Xi'an Jiaotong University, Xi'an, China, in 1988, 1993, and 2001, respectively. He is currently a Professor and the Deputy Dean of the School of Electronic and Information Engineering, Xi'an Jiaotong University. His research interests include secure computation, data privacy, and the Internet of Things.

• • •