Verifiable Symmetric Searchable Encryption For Semi-honest-but-curious Cloud Servers

Qi Chai
Department of Electrical & Computer Engineering
University of Waterloo
Waterloo, Ontario N2L 3G1, CANADA
Email: q3chai@uwaterloo.ca

Guang Gong
Department of Electrical & Computer Engineering
University of Waterloo
Waterloo, Ontario N2L 3G1, CANADA
Email: ggong@uwaterloo.ca

***Abstract*—Outsourcing data to cloud servers, while increasing service availability and reducing users' burden of managing data, inevitably brings in new concerns such as data privacy, since the server may be *honest-but-curious*. To mediate the conflicts between data usability and data privacy in such a scenario, research of *searchable encryption* is of increasing interest.**

Motivated by the fact that a cloud server, besides its curiosity, may be selfish in order to save its computation and/or download bandwidth, in this paper, we investigate the searchable encryption problem in the presence of a *semi-honest-but-curious server*, which may execute only a fraction of search operations honestly and return a fraction of search outcome honestly. To fight against this strongest adversary ever, a verifiable SSE (VSSE) scheme is proposed to offer *verifiable searchability* in additional to the data privacy, both of which are further confirmed by our rigorous security analysis. Besides, we treat the practicality/efficiency as a central requirement of a searchable encryption scheme. To demonstrate the lightweightness of our scheme, we implemented and tested the proposed VSSE on a laptop (serving as the server) and a mobile phone running `Android` 2.3.4 (serving as the end user). The experimental results optimistically suggest that the proposed scheme satisfies all of our design goals.

I. Introduction

The emergence of cloud computing provides considerable opportunities for academia, IT industry and global economy. Compared to other distributed computing paradigms, one fundamental advantage of the cloud is the enabling of data outsourcing, where end users could enjoy massive data storage/usage with even resource-constrained devices. Despite the tremendous benefits, outsourcing data to cloud servers deprives customers' direct control over their data, which inevitably brings in new concerns on data privacy.

On the other hand, encryption is a well-established technology to boost data privacy. However, classical cryptographic primitives, no matter symmetric-key- or public-key-based, lead data to be unusable and prevent even the authorized users from retrieving segments of data according to certain patterns/keywords. Hence, research of *searchable encryption*, i.e., looking for cryptography primitives and protocols to guarantee data privacy and searchability, is of increasing interest, and has been intensively studied by theorists and practitioners. Various searchable encryption schemes, e.g., [5], [6], [10], [2], [1], [3], [8], have been proposed to fight against a computationally bounded adversary called *honest-but-curious server*, who (1) stores the outsourced data without tampering it; (2) honestly executes every search operation and returns documents associated with the given queries; (3) tries to learn the underlying plaintext of user's data.

However, when experiencing commercial cloud computing services, we noticed that a public cloud server may be selfish in order to save its computation or download bandwidth, which is significantly beyond the conventional honest-but-curious server model. Following this intuition, in this paper, we consider a strongest adversary ever, called *semi-honest-but-curious server*, who may execute only a fraction of search operations honestly and/or return a fraction of search outcome honestly. To fight against it, we introduce one more design rationale – in addition to the data privacy – to the searchable encryption problem, which is named as *verifiable searchability*. Here, by "verifiable searchability", we mean that the server needs to prove to the user (who initiated the query) that the search outcome is correct and complete. Besides, we treat the practicality/efficiency as a central requirement of a searchable encryption scheme, and attempt to answer the following question: *is a searchable encryption scheme feasible even if the end user is a power-constrained device, e.g., mobile phones?* To pursue practicality and efficiency, we restrict ourselves to symmetric searchable encryption (SSE).

Our Contributions: We make following contributions: (1) we propose the first verifiable SSE (VSSE) scheme to the best of our knowledge, which not only enables a constant search complexity (with moderate storage overhead) for the server, but also provides data privacy as well as the correctness/completeness of queries, for the user; (2) VSSE is implemented and tested, with real world data sets, on a laptop (serve as the server) and a mobile phone running `Android v2.3.4` (serve as the user). The experimental results exhibit the efficiency of our scheme.

Related Works: Existing searchable encryption schemes can be categorized into three families: (1) solutions such as [2], [3], [10], [4], [1] attempt to develop novel cryptographic primitives. One such a primitive is the *homomorphic encryption* [2], where a specific algebraic operation performed on the plaintext is equivalent to a different algebraic operation performed on the ciphertext. Nevertheless, many efforts are needed to improve its efficiency. Another primitive is derived from deterministic encryptions [1], [3], i.e., $\mathrm{Enc}_K(\mathbf{x})$ and $\mathrm{Enc}_K(\mathbf{y})$ are identical if and only if the underlying plaintext **x** and **y** are equal. However, deterministic encryption is only able to provide privacy to plaintext with high min-entropy[1]; (2) solutions such as [6], [5], [8], work at data structure level by bringing in a secure index for the given documents. Schemes in this family often achieve more efficiency in search. In [5], a single encrypted hash table is built for the entire document collection, where each entry consists of the keyed hash value of a particular keyword and an encrypted set of document identifiers whose corresponding documents contain the keyword. However, this scheme become less practical with the growing size of the predefined keyword set, and, cannot function well with a semi-honest-but-curious server. (3) as a complementary approach, Raykova *et al.* [9] considered a similar problem – to hide querier's identity as well as the query – from the system level by introducing a trusted proxy, which re-encrypts the user's query to the server. However, the existence of a trusted third party may not be true for every application desiring searchable encryption.

Organization: Section 2 introduces system and threat models. Our scheme is presented in Section 3, while the security and performance analyses are exhibited in Section 4. Implementations and experimental results are reported in Section 5. Section 6 concludes this paper.

II. Problem Formulation

System Model: We consider a well-accepted scenario of data-outsourcing, which encompasses two roles: a data owner/user and a cloud server. Given a secure index and a collection of encrypted documents, the server performs the search according to the query from the user. Without loss of generality, we assume the authentication/authorization between the server and the user is appropriately done.

Threat Model: We consider a computationally bounded adversary, called semi-honest-but-curious server, which satisfies following properties: (1) the server is a storage provider, who does not modify/destroy the stored documents; (2) the server tries to derive sensitive information from the stored documents, user's queries as well as search outcomes; (3) in addition, the server may forge a fraction of the search outcome as it may execute only a fraction of search operations honestly.

Our Definition: In what follows, the following notations are used: (1) let $|X|$ denote the cardinality of a set X and $|\mathbf{x}|$ denote the number of components of a vector $\mathbf{x} = (x_1, ..., x_n)$. Note that we also write $(x_1, ..., x_n)$ as $x_1||...||x_n$ interchangeably; (2) let $\mathcal{E}$ be an alphabetic set of size $|\mathcal{E}|$. Let $\mathcal{D}$ be a set of N documents $\mathcal{D} = \{D_1, ..., D_N\}$, where each document D_i is a vector composed of several words, where each word is an aggregation of characters from the alphabetic set, i.e., $\mathbf{w} = (w[1], ..., w[L])$, $L = |\mathbf{w}|$, $w[i] \in \mathcal{E}$. Note that the unique identifier of each document can be obtained via $id(D_i)$; (3) let a query be $\mathbf{p} = (p[1], ..., p[m])$, $p[i] \in \mathcal{E}$. Unlike [5], **p** is not constrained to a pre-defined set of keywords in our scheme.

Definition 1: **(Verifiable Symmetric Searchable Encryption (VSSE))** A non-interactive verifiable symmetric searchable encryption scheme is a collection of the following polynomial-time algorithms: (1) keygen generates a ψ-bit secret key; (2) pre-process, taking security parameters n, produces a secure index based upon the data set $\mathcal{D}$, separately encrypts documents in $\mathcal{D}$ and uploads them to the cloud server; (3) querygen produces a privacy-preserving query, given the keyword and the secret key; (4) search outputs "Yes" if a queried pattern occurs in $\mathcal{D}$ and "No" otherwise. Additionally, a proof of the search outcome should be attached; (5) verify notifies the user whether the claimed search outcome is true and whether the server behaves honestly in the current session.

Design Goal: We require a potential scheme to satisfy the following requirements: (1) data privacy (as defined in [5], [8]): nothing should be leaked to the server from the remotely stored data and the index beyond the search outcome and the queries; (2) verifiable searchability: if the server behaves honestly in the current search, the probability that the search outcome is incorrect should be negligible; if the server returns incorrect and/or incomplete search outcome, the cheating behavior can be detected by verify with overwhelming probability; (3) efficiency: the running time of pre-process should increase linearly with the size of the data set while search, querygen and verify should be able to finish in a constant time. querygen and verify should be lightweight enough for resource-constrained devices, e.g., mobile phones[2].

III. VSSE: Verifiable SSE

In this section, we present the complete scheme. Let us start by reviewing relevant background.

A. Preliminary

Trie is an (incomplete) $|\mathcal{E}|$-ary tree to store a set of words from an alphabetic set $\mathcal{E}$. The basic idea behind is that all the descendants of a node in the trie have a common prefix associated with that node. An instance of trie is given in Fig. 1 (ignore all numerical notations for the time being). To perform a search in the trie, one starts from the root node and then reads the characters in a query word, following for each read character the outgoing pointer corresponding to that character move to the next node. If such a node does not exist, the search is immediately terminated returning a failure. On the other hand, after all characters in the query are read, one arrives at a node corresponding to the query word as prefix. If one of the children of the current nodes is the termination flag, denoted as "#", the search returns a success indicating that the query word must belong to the trie. Formally, a trie has the following property: *the space required to store n words of length L in a trie is much less than the space required to store a $|\mathcal{E}|$-ary full tree of depth L, .e.g., $O(\frac{|\mathcal{E}|^{L+1}-1}{|\mathcal{E}|-1})$, due*

[1]Min-entropy of a random variable X is defined as $H_{\min}(X) = -\log(\max(\mathrm{Prob}[X = x]))$, where $H(.)$ is Shannon's entropy, and $\mathrm{Prob}[X = x]$ is the probability that X takes value x.

[2]Pre-process is also launched by the user. However, it is less likely to be run on a resource-constrained device.

to the redundancy in the natural language; searching $\boldsymbol{p}$ *in a trie takes no more than* $|\boldsymbol{p}|$ *steps;* However, this data structure cannot be trivially applied to searchable encryption, as, even each of its nodes is encrypted, it leaks statistic information of the underlying plaintext characters, e.g., letter frequencies.

B. Our Scheme

Our VSSE scheme composes of five algorithms (keygen, pre-process, querygen, search, verify), among which, keygen has obvious meaning thus omitted here. Details of the other four algorithms are given in Algorithms 1, 2, 3 and 4 respectively, where we make use of following notations:

- $g_K : \{0,1\}^* \mapsto \{0,1\}^n$ is a keyed hash function such as SHA-256;
- s_K is a block cipher, e.g., AES, in cipher-block chaining (CBC) mode, to encrypt $(n + |\mathcal{E}|)$ bits of plaintext;
- $T_{x,y}$ denotes the x-th node from left to right of depth y in a tree T; child$(T_{x,y})$ denotes a descendant of $T_{x,y}$ while parent$(T_{x,y})$ denotes the direct predecessor of $T_{x,y}$;
- $ord(z)$ returns the alphabetic order of $z \in \mathcal{E}$.

Intuitively, pre-process helps the user to create a trie-like index, called a PPTrie T. First, T is initialized as a full $|\mathcal{E}|$-ary tree, where each node contains three attributes $(r_0, r_1, r_2) =$ (NULL,NULL,NULL). When traversing the documents and reading in each character of each word, pre-process updates the attributes of corresponding nodes: (1) $T_{x,y}[r_0]$ stores the character from a plaintext word; (2) $T_{x,y}[r_1]$ stores a keyed hash value – call it *prefix signature* – of all predecessor nodes of $T_{x,y}$. Thus, $T_{x,y}[r_0]$ is intended to be globally unique in T; (3) $T_{x,y}[r_2]$ represents, using bitmap technique, the set of children nodes of $T_{x,y}$ if it is an internal node. For example, if $T_{x,y}$ has only one child such that $ord(\text{child}(T_{x,y})[r_0]) = i$, the i-th bit of $T_{x,y}[r_2]$ is set to "1" in a $|\mathcal{E}|$-bit binary string, while other bit positions are set to zero. On the other hand, if the current node is a leaf node (whose $T_{x,y}[r_1] =$ "#"), identifiers of documents in which the associated word appears, is stored in r_2. Once all words from the plaintext are processed, nodes with empty attributes could be either removed (while storage efficiency is preferred) or padded with random attributes (while data privacy is strongly desired). At last, r_0 of each node is deleted permanently. In parallel to building the index, documents are separately encrypted by a symmetric cipher in a conventional manner.

Querygen generates a privacy-preserving query, i.e., $\boldsymbol{\pi} = (\pi[1], ..., \pi[m+1])$, in the spirit of the prefix signature – the value of $\pi[i]$ depends on the signature of the prefix $(p[1], .., p[i-1])$. Search is principally to find a path in T according to the components of $\boldsymbol{\pi}$, from the root to one termination flag – the existence of such a path indicates that the queried word happens in at least one of the target documents. During every step of the path exploration, search produces a proof which is later returned to the user. The validity of the search outcome is examined by verify while inputting proof.

Algorithm 1 Pre-process (by the user)

```
Require:
    (1) secret key $K$ and security parameters $n$
    (2) $N$ documents: $D_i$, $1 \le i \le N$
    (3) strategy: "privacy preferred" or "efficiency preferred"
Ensure:
    (1) PPTrie $T$
 1: create $T$ to be a full $|\mathcal{E}|$-ary tree
 2: $(r_0, r_1, r_2) \Leftarrow$ (NULL,NULL,NULL) for each node
 3: $T_{0,0}[r_0] \Leftarrow$ root; $T_{0,0}[r_1] \Leftarrow 0$; $q_0 \Leftarrow 0$
 4: for each word $\mathbf{w} = (w[1], w[2]...)$ in $D_i$, $1 \le i \le N$ do
 5:   for $j$ from 1 to $|\mathbf{w}|$ do
 6:     Find $q_j \in [q_{j-1} \times |\mathcal{E}| + 1, (1 + q_{j-1}) \times |\mathcal{E}|]$ such that
        $T_{j,q_j}[r_0] = w[j]$; otherwise, randomly select $q_j$ such that
        $T_{j,q_j}[r_0]$ is unset
 7:     $T_{j,q_j}[r_0] \Leftarrow w[j]$
 8:     $T_{j,q_j}[r_1] \Leftarrow g_K(j, w[j], \text{parent}(T_{j,q_j})[r_1])$
 9:   end for
10:   Find $q_{j+1} \in [q_j \times |\mathcal{E}| + 1, (1 + q_j) \times |\mathcal{E}|]$ such that
      $T_{j+1,q_{j+1}}[r_0] =$ "#"; otherwise, randomly select $q_{j+1}$ such
      that $T_{j+1,q_{j+1}}[r_0]$ is unset
11:   $T_{j+1,q_{j+1}}[r_0] \Leftarrow$ "#"
12:   $T_{j+1,q_{j+1}}[r_1] \Leftarrow g_K(j+1, \text{"\#"}, \text{parent}(T_{j+1,q_{j+1}})[r_1])$
13:   $T_{j+1,q_{j+1}}[r_2] \Leftarrow T_{j+1,q_{j+1}}[r_2] || id(D_i)$ since $\mathbf{w} \in D_i$
14: end for
15: for each node $T_{j,q_j}$ in $T$ do
16:   if $T_{j,q_j}$ is a termination/leaf node then
17:     $T_{j,q_j}[r_2] \Leftarrow T_{j,q_j}[r_2] || g_K(T_{j,q_j}[r_2])$
18:   else
19:     $mem \Leftarrow 0$
20:     for each of $T_{j,q_j}$'s children do
21:       $mem[ord(\text{child}(T_{j,q_j})[r_0])] \Leftarrow 1$
22:     end for
23:     $T_{j,q_j}[r_2] \Leftarrow s_K(T_{j,q_j}[r_1], mem)$
24:   end if
25: end for
26: if strategy = "privacy preferred" then
27:   padding unset $(r_1, r_2)$s with random binary streams
28: else
29:   delete all empty nodes
30: end if
31: delete $r_0$ of each node
32: return $T$
```

Algorithm 2 Querygen (by the user)

```
Require:
    (1) secret key $K$
    (2) query $\mathbf{p} = (p[1], ..., p[m])$
Ensure:
    (1) privacy-preserving query $\boldsymbol{\pi} = (\pi[1], ..., \pi[m+1])$
1: $p[m+1] \Leftarrow$ "#"; $\pi[0] \Leftarrow 0$
2: for each $j \in [1, m+1]$ do
3:   $\pi[j] \Leftarrow g_K(j, p[j], \pi[j-1])$
4: end for
5: return $\boldsymbol{\pi}$
```

C. A Live Example

To further exemplify our scheme, we present a toy instance as shown in Fig. 1, where a PPTrie, containing "BIG", "BIN", "BING", "BAD" and "BAGS" from the alphabetic set {A,B,D,G,N,S,#}, is constructed by pre-process with strategy="efficiency preferred". Each node in T holds a

Algorithm 3 Search (by the server)

```
Require:
    (1) PPTrie $T$
    (2) privacy-preserving query $\boldsymbol{\pi} = (\pi[1], ..., \pi[m+1])$
Ensure:
    (1) "Yes", if the search is successful; "No", otherwise
    (2) document IDs if "Yes"
    (3) proof of the search outcome
 1: proof $\Leftarrow T_{0,0}[r_2]$; $q_0 \Leftarrow 0$
 2: for $j$ from 1 to $m+1$ do
 3:     hit $\Leftarrow$ False
 4:     for $q_j \in [q_{j-1} \times |\mathcal{E}| + 1, (1 + q_{j-1}) \times |\mathcal{E}|]$ do
 5:         if $T_{j,q_j}[r_1] = \pi[j]$ then
 6:             hit $\Leftarrow$ True; proof $\Leftarrow$ proof $||T_{j,q_j}[r_2]$; break;
 7:         end if
 8:     end for
 9:     if hit = False then
10:         proof $\Leftarrow$ proof $||j$
11:         return "No" and proof
12:     end if
13: end for
14: proof $\Leftarrow$ proof $||j$
15: if $T_{j,q_j}$ has no child then
16:     return "Yes", proof, and $T_{j,q_j}[r_2]$ (containing document IDs)
17: end if
```

Algorithm 4 Verify (by the user)

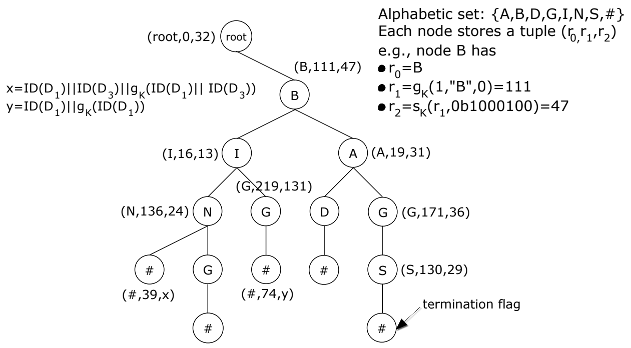
```
Require:
    (1) "Yes" with document IDs $T_{j,q_j}[r_2]$ or "No"
    (2) proof: $T_{1,q_1}[r_2]||...||T_{j,q_j}[r_2]||j$
    (3) privacy-preserving query $\boldsymbol{\pi} = (\pi[1], ..., \pi[m+1])$
    (4) plaintext pattern $\mathbf{p} = (p[1], ..., p[m])$
Ensure:
    (1) True or False
 1: if "Yes" $b \Leftarrow \underbrace{1, ...1}_{j-1}, 1$; otherwise $b \Leftarrow \underbrace{1, ...1}_{j-1}, 0$
 2: if "Yes" then
 3:     $(\hat{mem}, g_K(mem)) \Leftarrow T_{j,q_j}[r_2]$, where $\hat{mem}$ is the concatenation of IDs received by the user
 4:     return False if $g_K(\hat{mem}) \neq g_K(mem)$
 5: end if
 6: while $j \geq 0$ do
 7:     $j \Leftarrow j - 1$;
 8:     decrypt $T_{j,q_j}[r_2]$ to get $(x, y)$
 9:     if $x \neq \pi[j]$ or $y[ord(p[j+1])] \neq b[j+1]$ then
10:         return False
11:     end if
12: end while
13: return True
```

tuple (r_0, r_1, r_2) as specified, e.g., for node "A", $r_2 = s_K(r_1, 00110000) = 31$ where "00110000" represents that both node "D" and node "G" are in its children set. Here we keep r_0 of each node unremoved for clearness.

To search for a pattern "BIG", querygen produces:

$$\begin{aligned}
\pi[1] &= g_K(1, \text{“}B\text{”}, 0) = 111, \\
\pi[2] &= g_K(2, \text{“}I\text{”}, \pi[1]) = 16, \\
\pi[3] &= g_K(3, \text{“}G\text{”}, \pi[2]) = 219, \\
\pi[4] &= g_K(4, \text{“}\#\text{”}, \pi[3]) = 74.
\end{aligned}$$

Upon receiving the pattern, the server does the following operations specified by search: (1) when the depth, denoted as j, is 1, it finds that r_2 of node "B" equals $\pi[1]$ in the query; (2) when $j = 2$, the fact that r_2 of node "I" equals $\pi[2]$ renders the algorithm chooses left branch to explore further; (3) when $j = 3$, the algorithm selects right child because r_2 of node "G" equals $\pi[3]$; (4) when $j = 4$, a termination node is reached. The server thus sends back "Yes" together with the document identifiers, i.e., $y = id(D1)||g_k(id(D1))$, as well as the proof $(32||47||13||131||4)$. On the other hand, providing the pattern to be searched is "BID", the server is incapable to find a child of node "I" equalling to $\pi[3]$. Therefore, it responses "No" with the proof $(32||47||13||3)$. In another scenario, the server dishonestly claims that "BIG" does not exist by returning "No" with the proof $(32||1)$. Verify could inform the user the invalidity of current search as, by decrypting 32, verify could observe that "I" does exist as a child of node "B".


Alphabetic set: {A,B,D,G,I,N,S,#}
Each node stores a tuple (r_0,r_1,r_2)
e.g., node B has
● r_0=B
● $r_1=g_K(1,\text{"B"},0)=111$
● $r_2=s_K(r_1,0b1000100)=47$
(root,0,32)
root
(B,111,47)
B
$x=ID(D_1)||ID(D_3)||g_K(ID(D_1)|| ID(D_3))$
$y=ID(D_1)||g_K(ID(D_1))$
(I,16,13)
I
A
(A,19,31)
(G,219,131)
(N,136,24)
N
G
D
G
(G,171,36)
#
G
#
#
S
(S,130,29)
(#,39,x)
(#,74,y)
termination flag
#
#
Fig. 1. A toy PPTrie constructed by Pre-process

IV. Security/Performance Analysis

A. Security Analysis

Data Privacy: The documents are separately encrypted, and their confidentiality is essentially ensured by the underlying cipher. By using a cryptographic strong cipher, it is sufficient to assume that encrypted documents leaks zero information (except their respective lengths). Besides, the privacy-preserving query can be understood as a collection of $(m+1)$ prefix signatures, the confidentiality/onewayness of which are guaranteed by the underlying hash function.

Instead, more focus should be placed on the confidentiality of the index T. As specified, each node in T has a tuple (r_0, r_1, r_2), where r_0 is deleted after T is created while r_1 (r_2 resp.) is a hashed (encrypted resp.) value. Therefore, direct derivations of plaintext information from (r_1, r_2) seems impossible. Nonetheless, the server may take advantage of the mutual information among nodes in T to learn statistic information regarding (r_1, r_2)s. Due to the following theorem, our scheme is secure in this sense.

Theorem 1: Providing T of depth L has C nodes, $C \leq \frac{|\mathcal{E}|^{L+1}-1}{|\mathcal{E}|-1}$, we have

$$\begin{aligned}
&\text{Prob}[T_{j,q}[r_1] = T_{\hat{j},\hat{q}}[r_1] | (q,j) \neq (\hat{q},\hat{j})] \\
&\qquad \approx 1 - (\frac{2^n - 1}{2^n})^{C(C-1)/2} \qquad (1)
\end{aligned}$$

$$\text{Prob}[T_{j,q}[r_2] = T_{\hat{j},\hat{q}}[r_2] | (q,j) \neq (\hat{q},\hat{j})] < 1 - (\frac{2^n - 1}{2^n})^{C(C-1)/2}. \quad (2)$$

Proof: It is only necessary to prove Eq. (1) – as long as it holds, Eq. (2) follows. This is because r_2 is calculated through

$$T_{j,q}[r_2] \Leftarrow s_K(T_{j,q}[r_1], mem). \quad (3)$$

Since s_K is an ideal cipher, $T_{j,q}[r_2] = T_{\hat{j},\hat{q}}[r_2]$ happens iff (r_1, mem) of $T_{\hat{j},\hat{q}}$ equals that of $T_{j,q}$, which happens with probability $\leq 1 - (\frac{2^n-1}{2^n})^{C(C-1)/2}$ due to Eq. (1).

To prove Eq. (1), let us recall that r_1 is constructed as below

$$T_{j,q}[r_1] \Leftarrow g_K(j, w[j], \text{parent}(T_{j,q})[r_1]). \quad (4)$$

Given two different words $\mathbf{w} = (w[1], w[2]...,)$ and $\mathbf{w}' = (w'[1], w'[2]...,)$ sharing a prefix, i.e., $w[i] = w'[i]$ for $i \leq I$, $I = 0, 1, ...$. It is clear that the shared prefix corresponds to the same set of nodes in T and has no impact on the uniqueness of r_1 of each node. Starting from $w[I+1] \neq w'[I+1]$, we can see that r_1s of the two nodes corresponding to $w[I+1]$ and $w'[I+1]$ are different as $g_K(I+1, w[I+1], X)$ differs from $g_K(I+1, w'[I+1], X)$, where X is the signature of the shared prefix. Thanks to the chained construction, this difference "propagates" all the way to r_1s of other nodes corresponding to the successive characters in $\mathbf{w}$ and $\mathbf{w}'$. Hence, the input of g_K can be understood as a random value, and, the probability the event $T_{j,q}[r_1] = T_{\hat{j},\hat{q}}[r_1]$ happens can be reduced to the well-studied birthday problem: given C integers drawn from $[0, 2^n - 1]$ uniformly at random, what is the probability that at least two numbers are the same? The answer is the right-hand-side of Eq. (1). ■

From theorem 1, it is almost certain that, given a suitable n, each node in T has a unique r_1 (r_2 resp.). In other words, the server is unable to distinguish T from a randomly-padded tree of the same-size without knowing the key.

Another concern is that the "shape" of T could indicate presence of particular words, e.g. a long path from root to the termination node may imply the presence of a word such as "Floccinaucinihilipilification". Fortunately, once the strategy "privacy preferred" is enabled, T is a full $|\mathcal{E}|$-ary tree, which is irrelevant to the set of words stored in it.

Verifiable Searchability: Let us assume j steps are performed by the server. If "No" is returned, we would know that the first $j-1$ characters are matched while $p[j]$ is mismatched, which could be described by a j-bit binary sequence $b = (1, ...1, 0)$; if "Yes" is returned, $b = (1, ...1, 1)$.

Starting from the last (or j-th) step, if "Yes", verify checks the integrity of the concatenation of the document identifiers by computing a keyed hash of it and comparing with the received one. In fact, the completeness of the search outcome is examined here. After that, j is decreased by one. If "No", the above step is skipped. Next, verify validates the correctness of the claimed search outcome by decrypting $r_2 = s_K(T_{j,q_j}[r_1], mem)$ and testing whether: (1) r_1 equals $\pi[j]$; (2) $ord(p[j])$-th position of mem equals $b[j]$. To tamper the search results, the server needs to forge the proof in this step in three possible ways: (1) try to generates a valid r_2 with $mem' \neq mem$; (2) randomly generates a binary stream to replace original r_2; (3) use r_2 of another node, e.g., $T_{\hat{j},\hat{q_j}}$, instead. Due to theorem 1 and Eq. (3), methods (1) and (2) can successfully cheat our algorithm with negligible probability providing the adversary has no knowledge about the key and s_K can be seen as a random oracle. For method (3), r_2 from another node, i.e., $s_K(T_{\hat{j},\hat{q_j}}[r_1], \hat{mem})$, contains a different prefix signature (the uniqueness of which is confirmed by theorem 1), which would be rejected by verify. In addition, the argument above can be applied recursively to the $(j-1)$-th step in verify and so on.

B. Performance Comparison

Table 1 compares our scheme with previous SSE schemes. To make the comparison easier, we assume, for the time being, that n is the total number of words in $\mathcal{D}$ while $d \leq n$ is the number of keywords. Except oblivious RAMs [7], all schemes leak search outcomes and user's access patterns to the server. In addition, both SSE-1 and our scheme work at data structure level and have additional storage costs, i.e., $O(d)$ and $O(1)$ respectively, for the index. However, SSE-1 cannot function well with a semi-honest-but-curious server, e.g., the server could always return NULL as the search outcome while the user has no clue about the correctness/completeness of the claimed result. In all, our scheme introduces verifiable searchability without requiring extra commmunication/complexity cost.

TABLE I
COMPARISON OF SSE SCHEMES

	GO96[7]	SWP00[10]	SSE-1[5]	**Our Scheme**
Pre-computation	-	$O(n)$	$O(d)$	$O(d)$
Storage	$O(n \log^2 n)$	$O(n)$	$O(d) + O(n)$	$O(1) + O(n)$ [1]
Search	$O(\log^3 n)$	$O(n)$	$O(1)$	$O(1)$
Comm. overheads	$O(\log^3 n)$	$O(1)$	$O(1)$	$O(1)$
# of rounds	$O(\log n)$	1	1	1
Hide access pattern	Yes	No	No	No
Verifiable searchability	No	No	No	Yes

[1] In our scheme, the PPTrie grows linearly with respect to the number of words in $\mathcal{D}$ initially, and gets saturated after a certain amount of words are added. Since adding a word, which exists in the PPTrie, does not introduce extra storage cost, the size of the PPTrie actually approaches a constant value $\leq O(\frac{|\mathcal{E}|^{L+1}-1}{|\mathcal{E}|-1})$, where L is the maximum length of the words from $\mathcal{D}$.

V. EMPIRICAL EVALUATION

To validate the efficiency/practicality of our scheme, we implemented keygen, pre-process and search on a laptop (P4 1.8, 2G memory) using `Python v2.6` in conjunction with `Psyco v1.6` and `PyCrypto v2.2`, where strategy="efficiency preferred", $\psi = n = 256$, $\eta = 128$, $g_K =$ HMAC using SHA-256 and $s_K =$AES-256 in CBC mode. The algorithms were tested using the two data sets with different statistic property of plaintext words: (1) Corpus-I is an English novel *Pride and Prejudice* by Jane Austen, which has about 70,000 English words related to literature and life; (2) Corpus-II comes from the DBLP computer science bibliography, which includes about 1.4 million publication records. Title of each record forms Corpus-II. Moreover, querygen and verify are developed on a `Nexus S` mobile phone, using

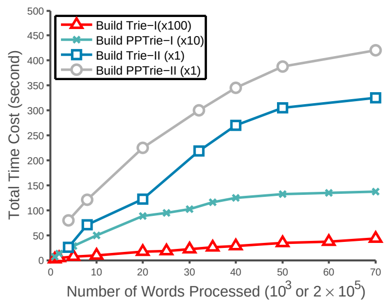


Fig. 2. Time cost to build Trie/PPTrie by Pre-process

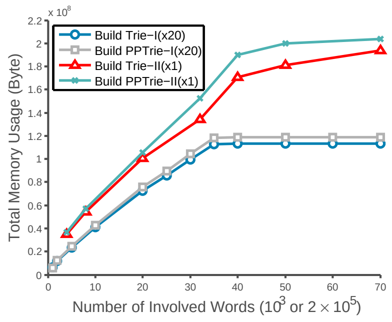


Fig. 3. Memory used for Trie/PPTrie by Pre-process

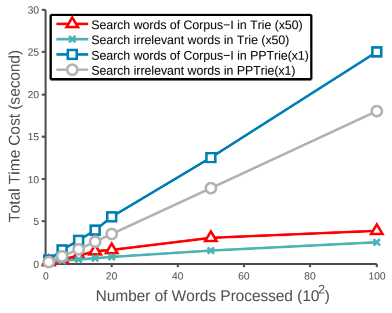


Fig. 4. Time cost to search in Trie-I/PPTrie-I by Search

`Android SDK v2.3.4` together with `javax.crypto.*` and `javax.security.*`.

Testings of Pre-process: Fig. 2 and Fig. 3 display the time and memory costs of building PPTrie-I/II with growing amount of data from Corpus-I/II. For the purpose of comparison, a plaintext Trie-I/II is also built from Corpus-I/II conventionally. Note that time cost of building a Trie-I/PPTrie-I is scaled by 100/10 and the unit of x-axis is 10^3 words for Trie-I/PPTrie-I and 2×10^5 words for Trie-II/PPTrie-II. Our results disclose that: (1) to build PPTrie-I/II only takes several ten/hundred seconds and to store PPTrie-I/II only requires 5.6/200MB memory; (2) the time cost grows linearly with respect to the increasing number of words processed, while the memory cost approach a constant. This is because Trie/PPTrie will eventually be saturated after a certain number of words are added, e.g., Trie-I/PPTrie-I is saturated after 35000 words were added, while Trie-II/PPTrie-II is saturated after 10^7 words were added, which manifests that words related to sciences/technology are more diversified.

Testings of Search: In our experiments, search selected keywords from two different keyword sets and queried Trie-I/PPTrie-I. Keywords in one set are from Corpus-I while keywords in another set are randomly selected from an English dictionary, which may be irrelevant. The obtained timings are shown in Fig. 4, where time cost of searching in the Trie-I is scaled by 50 (which shows that plaintext search using a trie is approximately 50 times faster than encrypted search using a PPTrie). Moreover, we obtained an estimation of throughput of search: 500 words/second. In addition, we noticed that searching for an irrelevant word is slightly faster, which is because search traverses Trie/PPTrie for few steps before a mismatch-and-terminate happens. This "incomplete traversing" saves operating time.

Testings of Querygen and Verify: In our tests, querygen, running on the `Nexus S` phone, generates 50000 privacy-preserving queries, where each query is of L characters and $L \in_R [1, 12]$ is uniformly selected at random. Similarly, verify examines 50000 valid proofs generated by the server-side, where each proof has L, $L \in_R [1, 12]$, components to be checked. The obtained average time costs of these two functions are: 5.34 million second/querygen, 8.01 million second/verify, which suggests that our scheme is quite efficient and practical even for resource-constrained end users.

VI. Conclusion

In this paper, we propose the first verifiable SSE scheme, which offers data privacy, verifiable searchability and efficiency, in the presence of an unusually strong adversarial server in the cloud computing environment. The rigorous security analysis together with our thorough experimental evaluations on a resource-constrained device using real data sets confirms that the proposed VSSE realizes our design goals and is a promising solution to mediate the conflicts between data usability and data privacy in such a scenario .

References

[1] M. Bellare, A. Boldyreva, and A. OŃeill, Deterministic and efficiently searchable encryption, *Advances in Cryptology, CRYPTO'07*, LNCS 4622, pp. 535–552, 2007.

[2] D. Boneh, G. Crescenzo, R. Ostrovsky and G. Persiano, Public key encryption with keyword search, *Advances in Cryptology, EUROCRYPT'04*, LNCS 3027, pp. 506–522, 2004.

[3] M. Bellare, M. Fischlin, A. O'Neill and T. Ristenpart, Derministic encryption: definitional equivalences and constructions without random oracles, *Advances in Cryptology, CRYPTO'08*, LNCS 5157, pp. 360–378, 2008.

[4] D. Boneh and B.Waters, Conjunctive, subset, and range queries on encrypted data, *Theory of Cryptography, TCC'07*, LNCS 4392, pp. 535–554, 2007.

[5] R. Curtmola, J. Garay, S. Kamara and R. Ostrovsky, Searchable symmetric encryption: improved definitions and efficient constructions, *Proceedings of the 13th ACM conference on Computer and Communications Security, CCS'06*, pp. 88–92, 2006.

[6] Y. Chang and M. Mitzenmacher, Privacy preserving keyword searches on remote encrypted data, *Applied Cryptography And Network Security, ACNS'05*, LNCS 3531, pp. 391–421, 2005.

[7] O. Goldreich and R. Ostrovsky, Software protection and simulation on oblivious RAMs, *Journal of the ACM*, vol.43, no. 3, pp. 431–473, 1996.

[8] J. Li, Q. Wang, C. Wang, N. Cao, K. Ren and W. Lou, Fuzzy keyword search over encrypted data in cloud computing, *Proceedings of the 29th Annual IEEE International Conference on Computer Communications, INFOCOM'10*, pp. 1–5, 2010.

[9] M. Raykova, B. Vo, S. Bellovin and T. Malkin, Secure anonymous database search, *Proceedings of the 2009 ACM Workshop on Cloud Computing Security, CCSW'09*, pp. 115–126, 2009.

[10] D. Song, D. Wagner and A. Perrig, Practical techniques for searches on encrypted data, *Proceedings of the 2000 IEEE Symposium on Security and Privacy, S&P'00*, pp. 44–55, 2000.

[11] A. Yun, C. Shi and Y. Kim, On protecting integrity and confidentiality of cryptographic file system for outsourced storage, *Proceedings of the 2009 ACM Workshop on Cloud Computing Security, CCSW'09*, pp. 67–76, 2009.