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
q3chai@uaterloo.ca

Guang Gong
Department of Electrical & Computer Engineering
University of Waterloo
Waterloo, Ontario N2L 3G1, CANADA
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 of 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 as well. To this end, we implemented and tested the proposed VSSE, with real world data sets, on a laptop (serve as the server) and a mobile phone running `Android` 2.3.4 (serve as the user). The experimental results optimistically suggest that the proposed scheme satisfies all of our design goals.

Categories and Subject Descriptors

E.3 [**Data Encryption**]: Symmetric Cryptography; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval.

General Terms

Data privacy, Algorithms.

Keywords

Symmetric searchable encryption, verifiable searchability, trie

1. 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, e.g., 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., [6, 5, 10, 3, 1, 2, 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 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 as well, 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) in this work.

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/time overhead for the server and the end user, but also provides data privacy as well as the verifiable searchability, both of which are further confirmed by our rigorous security analysis.

2. VSSE is implemented and tested, with real world data sets, on a laptop (serve as the server) and a mobile phone running `Android` 2.3.4 (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 [3, 2, 10, 4, 1] attempt to develop novel cryptographic primitives. One such primitive is the *homomorphic encryption* [3], 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, 2] – $\text{Enc}_K(\mathbf{x})$ and $\text{Enc}_K(\mathbf{y})$ are identical if and only if the underlying plaintext $\mathbf{x}$ and $\mathbf{y}$ are equal. However, deterministic encryption is only able to provide privacy to plaintext with high min-entropy[1]; (2) solutions such as [5, 6, 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 [6], 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. Li *et al.* investigated fuzzy keyword search over encrypted data in [8] and proposed to utilize the *edit distance* to measure the string similarity; (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. Hence, the use is limited.

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

2. PROBLEM FORMULATION

System Model: In this paper, we consider a well-accepted data-outsourcing scenario, which encompasses two roles: a data owner/user and a cloud server. Given a collection of encrypted documents and a keyword, the server performs the search for the user. Without loss of generality, we assume the authentication/authorization between the server and the user is appropriately done.

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

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 search patterns/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, we make use of the following notations: (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 ordered set 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 [6], $\mathbf{p}$ 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, η), produces searchable ciphers for a data set $\mathcal{D}$ and uploads them to the cloud server; (3) querygen produces a privacy-preserving query, given 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 tells the user whether the search outcome from the server is true and whether the server behaves honestly in the current search.

Design Goal: We require a potential scheme to satisfy the following requirements:
Data Privacy [6, 8]: nothing should be leaked to the server from the remotely stored data and the index beyond the search outcome and the (encrypted) search patterns/queries;
Verifiable Searchability: after executing search, the server responses with the search outcome and the proof. 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;
Efficiency: Time complexity of pre-process should be upper bounded by O(size of data set) while search, querygen and verify should be able to finish in constant time. Each operation in querygen and verify should be lightweight for resource-constrained devices, e.g., mobile phones[2].

3. VSSE: VERIFIABLE SSE

In this section, we present the complete scheme, in which the user builds an index, named *PPTrie* (*Privacy-Preserving Trie*), upon a given data set $\mathcal{D}$ before outsourcing it. In parallel to this, documents are separately encrypted by a

[2]Pre-process is also launched by the user. However, it is not likely to be run on a resource-constrained device.

symmetric cipher in a conventional manner. Let us start by reviewing relevant background.

3.1 Preliminary

Trie, abbreviated from "re*trie*val", is an (incomplete) $|\mathcal{E}|$-ary tree to store a set of words. 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 Figure 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.

Property 1. Trie stores a set of words from an alphabetic set $\mathcal{E}$. It supports the search on a query $\mathbf{p}$ with no more than $|\mathbf{p}|$ steps. The space requirement to store n words of length L is usually much less than $O(\frac{|\mathcal{E}|^{L+1}-1}{|\mathcal{E}|-1})$.

Due to its efficiency, trie structure is used in various applications, e.g., storage of a dictionary most commonly, or enabling of the auto-suggest and tab-completion features. However, this data structure cannot be trivially applied to solve the searchable encryption problem, as, even each of its nodes is encrypted, it leaks statistic information of the underlying plaintext characters, e.g., letter frequencies.

3.2 Our Scheme

Our VSSE scheme, as defined above, composes of five algorithms (keygen, pre-process, querygen, search, verify), among which, keygen has obvious meaning thus omitted here.

Pre-process helps the user to create a PPTrie T from the given set of documents. Let: $T_{x,y}$ denote the value of the x-th node from left to right of depth y in T; child$(T_{x,y})$ denote one descendant of a node $T_{x,y}$; and, parent$(T_{x,y})$ denote the predecessor of a node $T_{x,y}$. The PPTrie T is initialized as a full $|\mathcal{E}|$-ary tree, where each node contains three attributes $(r_0, r_1, r_2) = (null, null, null)$ in default: r_0 of each node stores the character in plaintext; r_1 stores a globally unique value – call it *prefix signature* – of the node, which is actually used during the search process; r_2 represents, using bitmap technique, the set of children of the current node if it is an internal node. For example, if the current node has only one child whose r_1 is the i-th character in $\mathcal{E}$, the i-th bit of a bitstream of length $|\mathcal{E}|$ is set to "1" while other bit positions are set to zero. On the other hand, if the current node is a leaf node (whose r_1 = "#"), identifiers of documents in which the associated word appears, is stored in r_2 (in plaintext). When traversing the documents and reading in each character of each word, the algorithm updates the attributes of corresponding nodes. Once all words from the plaintext are stored in T, nodes with empty attributes are either removed permanently or padded with random attributes, depending on one input parameter called "strategy". At last, r_0 of each node is deleted permanently.

Querygen generates a privacy-preserving query, i.e., $\boldsymbol{\pi} = (\pi[1], ..., \pi[m+1])$, in the spirit of a hash chain – the value of $\pi[i]$ depends on the unique signature of the prefix $(p[1], .., p[i-1])$. Search algorithm is basically 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 proof is examined by verify.

Details of pre-process, querygen, search and verify are given in Algorithms 1, 2, 3 and 4 respectively, where we make use of following primitives:

- $g_K : \{0,1\}^* \mapsto \{0,1\}^n$ 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 $(n + \eta)$ bits of plaintext;
- $ord(T_{x,y}[r_0])$ returns the alphabetic order of the character $T_{x,y}[r_0]$ in $\mathcal{E}$; if $r_0 = null$, we say the node $T_{x,y}$ is empty.

Algorithm 1 Pre-process (by the user)

Require:
(1) secret key K and security parameters (n, η)
(2) N documents: D_i, $1 \le i \le N$
(3) strategy: "privacy preferred" or "efficiency preferred"
Ensure:
(1) PPTrie T

create T to be a full $|\mathcal{E}|$-ary tree
$(r_0, r_1, r_2) \Leftarrow (null, null, null)$ for each node
$T_{0,0}[r_0] \Leftarrow$ root; $T_{0,0}[r_1] \Leftarrow 0$; $q_0 \Leftarrow 0$
for each word $\mathbf{w} = (w[1], w[2]...)$ in D_i, $1 \le i \le N$ **do**
  for j from 1 to $|\mathbf{w}|$ **do**
    Find $q_j \in [q_{j-1} \times |\mathcal{E}| + 1, (1 + q_{j-1}) \times |\mathcal{E}|]$ such that $T_{j,q_j}[r_0] = w[j]$; if cannot, find q_j such that T_{j,q_j} is empty
    $T_{j,q_j}[r_0] \Leftarrow w[j]$
    $T_{j,q_j}[r_1] \Leftarrow g_K(j, w[j], \text{parent}(T_{j,q_j})[r_1])$
  end for
  Find $q_{j+1} \in [q_j \times |\mathcal{E}| + 1, (1 + q_j) \times |\mathcal{E}|]$ such that $T_{j+1,q_{j+1}}[r_0]$ = "#"; if cannot, find q_{j+1} such that $T_{j+1,q_{j+1}}$ is empty
  $T_{j+1,q_{j+1}}[r_0] \Leftarrow$ "#"
  $T_{j+1,q_{j+1}}[r_1] \Leftarrow g_K(j+1, \text{"\#"}, \text{parent}(T_{j+1,q_{j+1}})[r_1])$
  $mem \Leftarrow mem || id(D_i)$ since $\mathbf{w} \in D_i$
end for
for each node T_{j,q_j} in T **do**
  if T_{j,q_j} is a termination/leaf node **then**
    $mem \Leftarrow mem || g_K(mem)$
  else
    $mem \Leftarrow 0$
    for each of T_{j,q_j}'s non-empty children **do**
      $mem[ord(\text{child}(T_{j,q_j})[r_0])] \Leftarrow 1$
    end for
    $T_{j,q_j}[r_2] \Leftarrow s_K(T_{j,q_j}[r_1], mem)$
  end if
end for
if strategy = "privacy preferred" **then**
  padding (r_1, r_2) of each empty nodes with random binary streams of same lengths
else
  delete all empty nodes
end if
delete r_0 of each node
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 $\boldsymbol{\pi} = (\pi[1], ..., \pi[m+1])$
$p[m+1] \Leftarrow$ "#"; $\pi[0] \Leftarrow 0$
for each $j \in [1, m+1]$ **do**
  $\pi[j] \Leftarrow g_K(j, p[j], \pi[j-1])$
end for
return $\boldsymbol{\pi}$

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 identifiers if "Yes"
(3) proof of the search outcome
proof $\Leftarrow T_{0,0}[r_2]$; $q_0 \Leftarrow 0$
for j from 1 to $m+1$ **do**
  hit $\Leftarrow$ False
  for $q_j \in [q_{j-1} \times |\mathcal{E}| + 1, (1 + q_{j-1}) \times |\mathcal{E}|]$ **do**
    if $T_{j,q_j}[r_1] = \pi[j]$ **then**
      hit $\Leftarrow$ True; proof $\Leftarrow$ proof $||T_{j,q_j}[r_2]$
      break;
    end if
  end for
  if hit = False **then**
    proof $\Leftarrow$ proof $||j$
    return "No" and proof
  end if
end for
proof $\Leftarrow$ proof $||j$
if T_{j,q_j} has no child **then**
  return "Yes", $T_{j,q_j}[r_2]$ as document identifiers and proof
end if

Algorithm 4 Verify (by the user)

Require:
(1) "Yes" with document identifiers $T_{j,q_j}[r_2]$ 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
if "Yes" $b \Leftarrow \underbrace{1, ...1}_{j-1}, 1$; otherwise $b \Leftarrow \underbrace{1, ...1}_{j-1}, 0$
if "Yes" **then**
  $(\hat{mem}, g_K(mem)) \Leftarrow T_{j,q_j}[r_2]$, where $\hat{mem}$ is the concatenation of identifiers received by the user
  return False if $g_K(\hat{mem}) \neq g_K(mem)$
  $j \Leftarrow j - 1$;
end if
while $j \geq 0$ **do**
  $j \Leftarrow j - 1$;
  decrypt $T_{j,q_j}[r_2]$ to get (x, y)
  if $x \neq \pi[j]$ or $y[ord(p[j+1])] \neq b[j+1]$ **then**
    return False
  end if
end while
return True

3.3 A Live Example

To further exemplify our scheme, we present a toy instance as shown in Figure 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 tuple (r_0, r_1, r_2) as specified, where r_2 represents children set of the current node, 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.

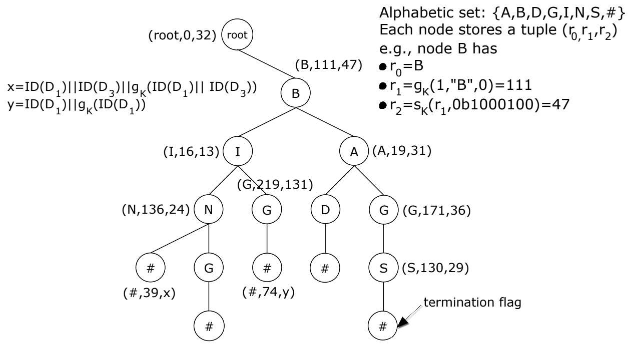

Figure 1: A toy PPTrie constructed by Pre-process containing words "BIG", "BIN", "BING", etc.

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

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

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 (as it has no child). 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 $(32||47||13||3)$.

4. SECURITY/PERFORMANCE ANALYSIS

4.1 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*

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

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

Stated in another way, r_1 (r_2 resp.) of node $T_{j,q}$ is (almost) unique in T.

PROOF. It is only necessary to prove Eq. (1) – as long as it is true, 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 a block cipher in CBC mode, "$T_{j,q}[r_2] = T_{\hat{j},\hat{q}}[r_2]$" happens iff (r_1, mem) of $T_{\hat{j},\hat{q}}$ equals that of $T_{j,q}$, which happens with probability less than $1 - (\frac{2^n-1}{2^n})^{C(C-1)/2}$ due to Eq. (1).

To prove Eq. (1), let us recall that r_1 is defined 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, $b = (1, ...1, 1)$.

	GO96[7]	SWP00[10]	SSE-1[6]	**Our Scheme**
Pre-computation	-	$O(n)$	$O(d)$	$O(d)$
Storage	$O(n \log^2 n)$	$O(n)$	$O(d) + O(n)$	$O(1) + O(n)$
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

Table 1: Comparison of SSE schemes

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 a different $mem' \neq mem$; (2) randomly generates a binary stream of $(n + \eta)$ 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. method (3) seems to be a promising strategy. However, 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.

4.2 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. Besides, 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. Generally speaking, our scheme introduces verifiable searchability without requiring extra commmunication/complexity cost.

5. EMPIRICAL EVALUATION

To validate the efficiency and 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. They were tested using the two (single-file) 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 `Android SDK v2.3.4` together with `javax.crypto.*` and

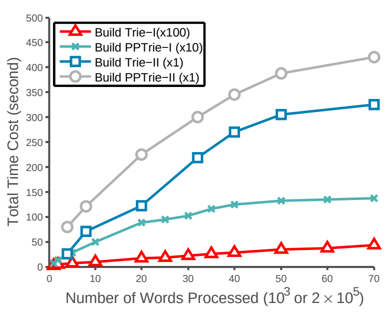

Figure 2: Time cost to build Trie/PPTrie by Pre-process

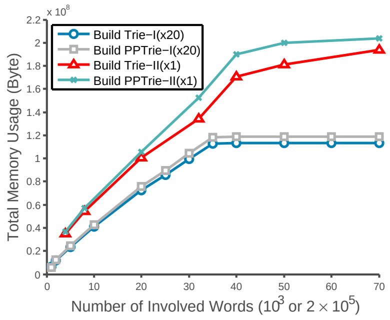

Figure 3: Memory used for Trie/PPTrie by Pre-process

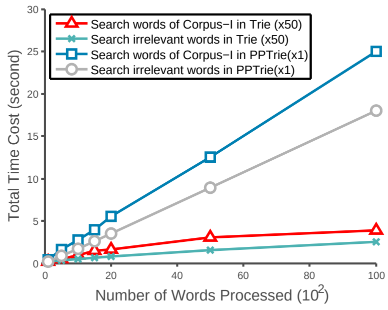

Figure 4: Time cost to search in Trie-I/PPTrie-I by Search

`javax.security.*`.

Testings of Pre-process: Figures 3 and 4 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 may suggest 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 Figure 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.

6. CONCLUSION

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

7. REFERENCES

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

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

[3] D. Boneh, G. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. *Lecture Notes in Computer Science*, 3027:506–522, 2004.

[4] D. Boneh and B. Waters. Conjunctive, subset, and range queries on encrypted data. *Theory of Cryptography*, pages 535–554, 2007.

[5] Y. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. *Lecture Notes in Computer Science*, 3531:442–455, 2005.

[6] 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*, pages 88–92, 2006.

[7] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):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. In *INFOCOM, 2010 Proceedings IEEE*, pages 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*, pages 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*, pages 44–55, 2000.