Towards Verifiable File Search on the Cloud

Fei Chen*, Tao Xiang†, Xinwen Fu‡, Wei Yu§

*Department of Computer Science and Engineering, The Chinese University of Hong Kong. Email:fchen@cse.cuhk.edu.hk.
†College of Computer Science, Chongqing University. Email: txiang@cqu.edu.cn.
‡Department of Computer Science, University of Massachusetts Lowell. Email: xinwenfu@cs.uml.edu.
§Department of Computer & Information Sciences, Towson University. Email: wyu@towson.edu.

***Abstract*—Cloud storage security has attracted a lot of research interest in recent years with the large-scale application of cloud computing. Although there is considerable work on verifying the integrity of the outsourced data in the cloud, research on verifying the file search results returned from the cloud is still lacking. To fill this gap, in this paper, we tackle the verifiable file search problem and propose two protocols to solve the verifiable file search problem. The baseline protocol can achieve the key verifiability property and the enhanced full-fledged protocol can further preserve the user's privacy by protecting the confidentiality of queried filenames while achieving the same level of verifiability as the baseline protocol. We conduct theoretical analysis to prove the correctness of the developed protocols and their security and privacy properties, along with overhead. We implement both the baseline and enhanced protocols and the resulting experimental data validates the effectiveness and efficiency of the proposed protocols.**

***Index Terms*—file search, verifiability, cloud computing, formal language, MAC**

I. Introduction

Recent years have witnessed the large-scale adoption of cloud computing by both individual users and companies. For example, individuals can outsource their data to GoogleDrive, Dropbox, etc. Thus, users can access their data at anytime, anywhere, and through various platforms such as PCs, tablets, and mobile phones. Companies also use Amazon cloud service or Google Engine to host their websites. While cloud computing brings huge benefits to users, the security issue remains as a concern for critical applications [1]. For the long-term sustainable development of cloud computing, addressing the security issues is very important.

There have been considerable research efforts on cloud storage security recently [2], [3], [4], [5], [6]. These research efforts can be generally divided into two categories: cloud storage auditing and encrypted keyword search. For cloud storage auditing, mechanisms were developed to ensure the integrity of the data outsourced to the cloud, including [7], [4], [5], [6]. For encrypted keyword search, research was conducted to search encrypted data outsourced to the cloud [8], [2], [3]. For cloud storage auditing, the correctness of the results returned from the cloud can be verified. Nonetheless, for encrypted search, only the work in [3] considered the challenging issue on verifying the correctness of the cloud. Note that verifiability is very critical and has also been studied in the area of outsourced databases [9], [10]. Nonetheless, the security of the proposed protocols was discussed only heuristically without formal analysis and verification.

In this paper, we focus on cloud storage security and investigate a *novel* security problem, i.e. how to search a file in the outsourced data *verifiably*, which is referred to as verifiable file search. To make verifiable file search feasible in real-world practice, we develop a lightweight, efficient, and secure solution on the verifiability issue. In comparison with the existing study, we focus especially on file search [3], [9], [10].

The setup of the verifiable file search problem is as follows. Suppose a user has outsourced all his/her files to a cloud. In order to find a file, the user sends a query to the cloud to request the return of the file. In some cases, it is correct and legal that the cloud replies to the user indicating that the queried file does not exist. This is because the user could send a wrong query, e.g. the case that the user queries a non-existing file or sends a wrong filename. Nonetheless, this legal answer could be abused by a malicious cloud because the cloud has various incentives to cheat [4], [6]. This type of cloud could lie to the user, claiming that an existing file does not exist or a non-existing file does exist. In this paper, we formulate the problem and develop protocols to enable verifiable file search.

It is worth noting that it is indeed a security concern that the cloud could return unfaithful results in practice. Such abnormal behavior may be due to factors such as the cloud could be hacked, the cloud has software and hardware failures, and/or employees of cloud service providers intend to interrupt the cloud's response under various economical incentives. For *rational* users, the search results returned from the cloud should be verified. From the cloud's viewpoint, it is also rewarding to provide a verifiable search service. On one hand, the operation to search over the data is highly expected by the users. Only by satisfying the needs of users as best as possible can the cloud service provider expect to survive in the market. On the other hand, providing a verifiable service makes the user believe that the cloud is indeed honest. This also helps set up the cloud's reputation. Thus, the cloud could have more clients and earn more market shares. This also helps to eliminate various incentives for the cloud to cheat. Hence, solving verifiable file search benefits both the users and the cloud.

In this paper, we propose a protocol that enables verifiable, secure, and efficient file search over outsourced data. First, we propose a baseline protocol, which can verify whether the file

The corresponding author is Tao Xiang (txiang@cqu.edu.cn).

search result returned by the cloud is correct, i.e. by enabling a verifiability check. This baseline protocol assumes that the queried filename sent to the cloud is the unmasked original filename. Thus, the baseline protocol lacks privacy protection because the query filename is leaked, which may further leak the privacy of users. To address this issue, we develop an enhanced full-fledged protocol that protects the privacy of the user through protecting the queried filenames and the file contents. Our developed protocols which achieve efficient search result verifiability are based on two key principles: one is to classify all possible filenames in a concise way and the other is to embed some secret information in the outsourced data, which is known only by the user. To implement these principles, we use two mechanisms: wildcard representation of strings over an alphabet and keyed message authentication codes [11].

To summarize, in this paper, we make the following paramount contributions:

- *Novel Protocols.* We formalized the problem of *verifiable file search* over outsourced data and proposed a baseline protocol and an enhanced full-fledged protocol which can solve the problem. The baseline protocol enables a user to verify whether or not a cloud cheats when a filename is searched. The enhanced full-fledged protocol preserves the privacy of users who request filenames while still achieving verifiability.
- *Theoretical Foundation.* We formally proved the correctness of our full-fledged protocol and its security and privacy properties using formal methods. We analyzed the privacy and integrity of our designed protocol along with the overhead in terms of computation, storage, and communication costs. Our theoretical analysis contributes to the foundation and understanding of the verifiable file search problem and the design of secure and efficient protocols.
- *Empirical Validation.* We implemented both the baseline and full-fledged protocols. Using a real-world data set, we conducted experiments and measured the computation, storage, and communication costs of these protocols. Our experimental data validates the effectiveness and efficiency of the proposed protocols.

The following paper is organized as follows. In Section II, we formulate the problem of verifiable file search over outsourced data. In Section III, we present a framework to address the verifiable file search problem. In Sections IV and V, we present our solution to the problem in two steps: first, a simple baseline protocol, and second, a full-fledged protocol. In Section VI, we analyze the correctness, security, privacy, and overhead of the proposed full-fledged protocol. In Section VII, we provide experimental results to validate the effectiveness of our proposed protocols. In Section VIII, we review related work. Finally, we conclude the paper in Section IX.

II. Problem Formulation

In this section, we formalize the problem of verifiable file search over outsourced data and state the assumptions. As

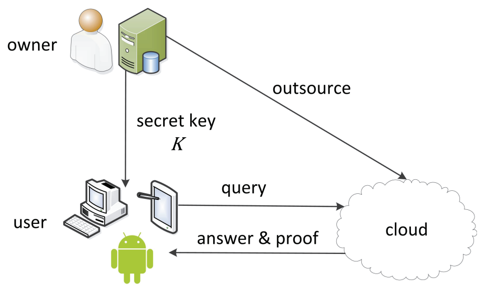

Fig. 1. System Model for Verifiable File Search

shown in Fig. 1, we consider the real-world use of cloud computing architecture, which consists of three parties: a data owner, a data user, and a cloud. A data owner can be a company and a data user can be an employee of the company.

The system works as follows: To reduce management costs, the data owner wants to outsource the data to the cloud. To prevent possible cheating by the cloud, the data owner also embeds some secret information in the outsourced data so that the data user can verify whether or not the cloud is honest on a file query. The data owner and the data user share some common secret information to achieve verifiability. When the data user wants a file, the user sends a query to the cloud that includes the filename. After receiving the query, the cloud searches the outsourced data. If the file exists, the cloud returns the file, along with a proof showing that the returned data is valid. If the file does not exist, the cloud proves to the user that it has no such file.

We consider a malicious cloud in this paper. We say a cloud is malicious if the cloud deviates the protocol in arbitrary ways in order to achieve its goal. Modeling the cloud as malicious is also adopted in the research community [4], [6]. Section I motivates why we need to model the cloud as malicious; although the cloud itself maybe honest. Nonetheless, the returned results could be tampered with due to various attacks, software/hardware failures, just to name a few. We assume that the data owner does not want the cloud to obtain any useful knowledge about the outsourced data. The same assumption applies to the data user too. We assume the file content of each file is well-protected through encryption and authentication mechanisms. This helps protect the privacy and integrity of the file content. In this way, the file content cannot be modified maliciously. Nonetheless, a filename can leak considerable useful information because it is a common practice that a meaningful filename is given to a file according to the file content. In this paper, we aim to protect the privacy of users associated with filenames and file contents.

A protocol that achieves verifiable file search over outsourced data is thus expected to have the following properties:

1) *Correctness*. If all the three parties follow the protocol honestly, the data user can indeed find the wanted file from the cloud.
2) *Security*. If the cloud fools the data user, the user should be able to identify such malicious behavior. For example,

the cloud can indicate that an existing file does not exist or a non-existing file does exist.

3) *Privacy*. The cloud should have no meaningful idea of the data and the queried file, including both the filenames and contents, which may also leak the user's privacy. We aim to achieve this minimum privacy requirement in this paper. We are not focusing on protecting the access pattern and query pattern of the user, as did by existing researches [12], [13].
4) *Efficiency*. The computation, storage, and communication costs of the three parties required by the protocol should be as small as possible.

III. Solution Framework

In this section, we design a framework that can solve the problem of verifiable file search over outsourced data. We also review two mechanisms that will be used in our protocol design.

A. Framework

Formally speaking, a verifiable file search protocol VS = (KeyGen, $\mathsf{Outsource}$, Query, Search, Verify) consists of five components:

- $\mathsf{KeyGen}(1^\lambda)$: After inputting a security parameter λ, the data owner executes this algorithm to output a secret key K, which can protect the security of the protocol and privacy of users. The data owner could share this secret key with the data user as well.
- $\mathsf{Outsource}(F; K)$: After inputting a collection of files F, the data owner embeds some secret information in the data using the secret key K and then outsources the new data F' to the cloud.
- $\mathsf{Query}(f; K)$: After a file f is queried as an input, the data user sends a query q containing the necessary information about searching for f on the cloud. The queried filename depends on the original file f, the secret key, and the privacy requirement of the data user.
- $\mathsf{Search}(q)$: After inputting a query q for some file, the cloud searches for it in the outsourced data. Then, the cloud returns the queried file χ. The cloud also sends a proof Γ to show that χ is indeed the file that the data user requested.
- $\mathsf{Verify}(f, \chi, \Gamma; K)$: After inputting the returned answer χ, correctness proof Γ, the original query filename f, and the secret key K, the data user checks whether or not the cloud has cheated. If the verification succeeds, the answer is accepted; otherwise, the answer is rejected.

A verifiable file search system will work in the following way. The data owner first runs $\mathsf{VS.KeyGen}$ to generate a secret key and then runs $\mathsf{VS.Outsource}$ to outsource the files to the cloud. The data owner also shares the secret key with the data user. Later, when the data user wants to query a file on the cloud, the user runs $\mathsf{VS.Query}$ to send a query to the cloud for searching for the file. Upon receiving the file search query, the cloud runs $\mathsf{VS.Search}$ to search for the queried file and returns the search result. Lastly, the data user runs $\mathsf{VS.Verify}$ to check whether the returned result from the cloud is correct. It is worth noting that the query, search, and verification processes may be run in multiple rounds by the data user and the cloud.

In this paper, we aim to formalize the security of a verifiable file search protocol in a rigorous way, i.e. as in the traditional definition-and-proof approach. Indeed, there have been a number of security breaches in real-world protocols [14], [15] because of the lack of rigorous security design. To define the security of our protocol, we need to understand what security means in practice and the power of a malicious cloud. Intuitively, such a protocol is secure if no real-world malicious cloud can fool the data user with a high probability. We say a tuple (f^*, χ^*, Γ^*) is a forgery if f^* is an existing file, but the cloud (χ^*, Γ^*) says it does not exist, or f^* is a non-existing file, but the cloud (χ^*, Γ^*) says it does exist. Then, we can state that a protocol is secure if the cloud cannot find such a forgery with a high probability. In practice, a cloud can see a lot of interaction between the user and the cloud. We also allow the cloud to know the output of the verification algorithm as the data user may explicitly reject the answer and adopt further actions, e.g. suing the cloud if a false answer is found. With the above considerations, we define the security of the protocol as follows:

Definition 1. *Denote VS = (KeyGen, $\mathsf{Outsource}$, Query, Search, Verify) as a verifiable file search protocol. Suppose $\mathcal{A}$ is a malicious cloud. We define the probability that $\mathcal{A}$ cheats successfully after $\mathcal{A}$ observes many verification results as:*

$$\mathrm{Adv}(\lambda; \mathsf{VS}) = \Pr\left[\begin{array}{c} K \leftarrow \mathsf{KeyGen}(1^\lambda) \\ F' \leftarrow \mathsf{Outsource}(F; K) \\ q_i \leftarrow \mathsf{Query}(f_i; K) \\ (\chi_i, \Gamma_i) \leftarrow \mathsf{Search}(q_i) \\ \delta_i \leftarrow \mathsf{Verify}(f_i, \chi_i, \Gamma_i; K) \\ i = 1, \cdots, \mathrm{poly}(\lambda) \end{array} : \begin{array}{c} \mathcal{A}(F', q_i, \delta_i) \\ \textit{outputs a} \\ \textit{forgery} \\ (f^*, \chi^*, \Gamma^*) \\ \textit{and the data} \\ \textit{user accepts it} \end{array}\right] \quad (1)$$

where $\mathrm{poly}(\lambda)$ is a polynomial function in λ. If $\mathrm{Adv}(\lambda; \mathsf{VS})$ is very "small" for every real-world malicious cloud, the protocol is said to be secure.

The left part before ":" denotes the power of a malicious cloud and the right part denotes the forgery. In practice, "small" $\mathrm{Adv}(\lambda; \mathsf{VS})$ can be interpreted in many ways. In this paper, we require $\mathrm{Adv}(\lambda; \mathsf{VS}) < \frac{1}{\mathrm{poly}(\lambda)}$ to be true asymptotically for any fixed polynomial in λ, in which case it is called negligible [16]. A good example to think about it is $\mathrm{Adv}(\lambda; \mathsf{VS}) = 2^{-\lambda}$.

B. Two Mechanisms

We now briefly introduce two mechanisms that will be used in our protocol design. One is based on formal language theory [17] and the other is based on Message Authentication Codes (MAC).

Strings & wildcard. Denote Σ as an alphabet containing a number of characters. A string over Σ is a finite number of characters in Σ. Denote all possible strings over Σ as $\Sigma^{(*)}$. For example, $\Sigma = \{a, b, c, d, \cdots, z, 0, 1, \cdots, 9, .\}$. Strings over Σ

can be abc, $ab1$, $temp.txt$, $file2.pdf$ etc. A wildcard # is a special character that can be used to denote any string in $\Sigma^{(*)}$. Hence, a string $file\#.pdf$ can be $file.pdf$, $filebcd.pdf$, $file2ab5.pdf$, etc. In this paper, we will use this mechanism to express all possible filenames for the outsourced data.

MAC. Generally speaking, a MAC scheme is mainly used to validate the integrity of a message, preventing illegal modifications of the message. Formally speaking, it contains three algorithms: MAC = (KeyGen, Tag, Verify). After giving a security parameter λ as an input, the key generation algorithm KeyGen outputs a secret key K. After receiving a message m and the secret key as inputs, the secret-embedding algorithm Tag embeds some secret information $\mathsf{Tag}(m; K)$ within the message. The secret information is also referred to as a MAC. After receiving a message m and its authentication code σ as inputs, the verification algorithm Verify outputs ACCEPT if $\sigma = \mathsf{Tag}(m; K)$; else it outputs REJECT. Similarly, let

$$\begin{aligned}&\mathrm{Adv}(\lambda; \mathsf{MAC})\\ &= \Pr\left[\begin{array}{ccc} & & \mathcal{A}(m_i, \sigma_i) \\ K \leftarrow \mathsf{KeyGen}(1^\lambda) & & \text{outputs a} \\ \sigma_i \leftarrow \mathsf{Tag}(m_i; K) & : & \text{different} \\ i = 1, \cdots, \mathrm{poly}(\lambda) & & (m^*, \sigma^*) \text{ and} \\ & & \mathsf{Verify} \text{ accepts it}\end{array}\right] \quad (2)\end{aligned}$$

be the probability of success that an adversary $\mathcal{A}$ is able to fake a legal message and authentication pair (m^*, σ^*). We say a MAC scheme is secure if the probability $\mathrm{Adv}(\lambda; \mathsf{MAC})$ is negligible. In this paper, we use HMAC [11] in our protocol design.

IV. A Basic Protocol for Verifiable File Search

In this section, we present the detailed design of our protocols to address the problem of verifiable file search. Particularly, we first introduce our basic idea. Then, we tackle the technical challenges in the basic idea to solve the verifiable file search problem. We later present our basic protocol that enables a data user to verify whether or not the cloud cheats.

A. Basic Idea

To conquer the verifiable file search problem, the key is to understand the biggest challenge and then find a way to tackle the challenge. The key challenge we find is that a malicious cloud could say an existing file does not exist, i.e. confusing the user about the existence of outsourced files. The reason is that the user cannot differentiate between existing files and non-existing ones. To address this problem, the intuition is to find a way to enable the user to differentiate between existing files and non-existing ones. The basic idea of the proposed protocol is shown as follows:

First, model all possible filenames as a set $\Sigma^{(*)}$ containing all possible strings over an alphabet Σ with some maximal length. In modern operating systems, only a limited number of characters are allowed and the there is a maximal length limit for the filenames.

Second, differentiate between existing files and non-existing ones. The data owner knows all existing filenames, denoted as a set F_1. Then, the non-existing filenames are in the set $\Sigma^{(*)} - F_1$, denoted as F_2. To tell F_1 and F_2 apart, we embed some special character in each filename of F_2. The special character could be # or @, which is not allowed in a legal filename. When outsourcing the files, the data owner outsources both F_1 and F_2.

Third, authenticate the outsourced data. In order to prevent the cloud from modifying F_1 and F_2 which contain the information about existing files and non-existing ones, respectively, we use the MAC scheme to embed some secret information accompanying each file in F_1 and F_2, i.e. authenticating the set. If the cloud desires to indicate that an existing file in F_1 does not exist, in addition to the authentication of the special filename, the cloud also needs to find a special filename containing both information about the queried filename and some special characters # or @. However, the cloud cannot find the correct authentication information because the cloud does not have the secret key of the MAC. The same analysis also applies to the case where the cloud attempts to fool a data user that a non-existing file does exist.

To summarize, our protocol design is based on the following two principles: one is to separate all existing filenames from non-existing ones using the previously introduced string model of filenames, and the other is to authenticate each possible filename, either existing or non-existing.

Nonetheless, one problem still remains open. The size of $\Sigma^{(*)}$ is $\mathsf{SZ}^{\mathsf{LEN}}$ where SZ is the size of the alphabet Σ and LEN is the maximum length of a filename which is allowed by the operating system. Thus, the size of F_2 is also exponential because $F_2 = \Sigma^{(*)} - F_1$ and F_1 to be outsourced has been shown to have a polynomial size. To address this issue, F_2 needs to be expressed in a concise way such that the size of F_2 can be reduced substantially.

We use the following example to clarify our idea to reduce the size of F_2. Suppose $file3$, $file3a$, $file3ab$, $file31$ are non-existing files contained in F_2. They can be expressed using only one string $file3\#$ after the wildcard representation of strings is used. This idea can be formalized using the equivalence relation language. For two strings $s_1, s_2 \in F_2$, we define the relation $\mathrm{rel}(s_1, s_2) = 1$ if they share a common prefix string. We require that the common prefix string is not a prefix for F_1. The reason will be shown in the next subsection. Then, it is easy to show that the relation $\mathrm{rel}(\bullet, \bullet)$ is an equivalence relation and thus it can divide the set F_2 into *finite* equivalence classes. For each equivalence class, we can express it using only one representative element. Thus, the size issue of F_2 is solved.

With the basic idea shown above, in the following subsection, we propose an algorithm to address the key challenge, i.e. explicitly separate all existing filenames from non-existing ones.

B. Separation of All Possible Filenames

We fix some notations first. Let Σ be the alphabet allowed by the data owner's operating system. Denote the length of a filename f as $\mathrm{length}(f)$. Denote α and β as two strings. Then,

$\alpha + \beta$ represents the concatenation of α and β. Denote # and @ as two special characters, which are not in Σ. Semantically, we use # to represent a wildcard character and employ @ to indicate a prefix character. For example, the string $\alpha\#$ can be any string starting with α whereas the string β@ indicates only the string β, which could be a prefix of some other string.

Algorithm 1 Construct All Possible Filenames

```
Input: A set of existing filenames $F_1$
Output: A set of non-existing filenames $F_2$
1: $F_2 \leftarrow \varnothing$
2: for all $f$ in $F_1$ do
3:     for $i = 0$ to length($f$) do
4:         Let $\alpha$ be a prefix of $f$ with length $i$
5:         for all characters $\beta \in \Sigma$ do
6:             $\gamma \leftarrow \alpha + \beta$
7:             if $\gamma$ is not a prefix of all files in $F_1$ then
8:                 $\gamma \leftarrow \gamma + "\#"$
9:                 $F_2 = F_2 \cup \{\gamma\}$
10:            end if
11:        end for
12:        if $\alpha \notin F_1$ and $\alpha$ is not empty then
13:            $\alpha \leftarrow \alpha + "@"$
14:            $F_2 = F_2 \cup \{\alpha\}$
15:        end if
16:    end for
17: end for
18: return $F_2$
```

Using the basic idea introduced in the previous subsection, F_1 and F_2 can be generated using Algorithm 1. Specifically, F_1 contains all the existing filenames and every non-existing filename can be represented using one special string in the set F_2 which has a polynomial size. Algorithm 1 is based on a fact that either a non-existing filename is a prefix of some existing filename or it is totally different from all existing filenames. Hence, by scanning each existing filename, we can find out all possible non-existing filenames. Note that all strings in F_2 have a special symbol, i.e. either # or @. The semantics are listed as follows: if $\alpha\#$ is in F_2, then all filenames starting with α do not exist; if β@ is in F_2, then the filename β does not exist *and* there is some file in F_1, which has β as a prefix. This property is formally summarized in the following proposition.

Proposition 1. *All possible filenames can be represented uniquely using only one string either in F_1 or F_2.*

Proof. Let f be an arbitrary filename. If f does exist, it will be contained in F_1. Thus, it can be represented as an element in F_1. In fact, this element is just f. If f does not exist, there are only two possible cases.

CASE 1. The string f is different from all existing filenames in F_1 and it is not a prefix of some filename in F_1. Suppose that f^* is the string, which is the most similar string to f, i.e. they share a common prefix with the maximal length. Denote this common string as α and the next character of f following α is β. When Algorithm 1 handles the string f^* on the prefix α from lines 2 to 4, it will add $\alpha+\beta+"\#"$ into F_2 as shown in line 8. Note that the string $\alpha+\beta+"\#"$ is a wildcard representation of the string f and "#" is a special character. Thus, the string f can be represented by $\alpha + \beta + "\#"$, which is in F_2.

CASE 2. The string f is a prefix of some existing filename in F_1. Then lines 12 to 15 adds the string $f + "@"$, i.e. the concatenation of f and a special character @, into F_2. Hence, the string f can be represented by the string $f + "@"$, which is in F_2.

Combining the above possible cases, the proof is completed. □

C. A Baseline Protocol

With the basic idea in mind and the key challenged solved, we develop a baseline protocol, as a first step, which can verify whether the cloud returns a false query result based on the separation of all possible filenames constructed in Algorithm 1. This baseline protocol assumes that the data owner, the data user and the cloud all know the true filename. Hence, the privacy of the filename is not protected. Later, we will extend this baseline protocol to handle the privacy issue of file names. In the following, we detail the semantics of each algorithm in our protocol VS = (KeyGen, Outsource, Query, Search, Verify).

KEYGEN. The data owner generates a secret key K_1 for H-MAC, which later will be employed to authenticate a filename.

OUTSOURCE. When the data owner prepares to outsource the data, the following steps will be taken. First, all possible filenames will be separated into two sets F_1 and F_2 as shown in Algorithm 1. Note that the set F_1 contains all existing filenames and all non-existing filenames can be represented by some elements in F_2 with special characters "#" or "@". For every existing filename f in F_1, the data owner sends the tuple (f, HMAC.Tag($f; K_1$), `file content`) to the cloud, where '`file content`' is the file content corresponding to the filename. It is worth noting that the file content is encrypted and authenticated by using a standard encryption algorithm and a standard message authentication algorithm with other independent keys, as we stated in Section II. For every non-existing filename f in F_2, the data owner also gives the tuple (f, HMAC.Tag($f; K_1$), `NULL`) to the cloud, where '`NULL`' indicates that there is no real file corresponding to this f. Later, the data owner shares the secret key K_1 with the data user.

QUERY. When the data user wants to fetch a file f from the cloud, the data user sends the filename to the cloud, i.e. giving f in a plaintext way to the cloud.

SEARCH. When the cloud receives a query filename f, the cloud first searches the data for existing files. If f does exist, the cloud can return the tuple (f, HMAC.Tag($f; K_1$), `file content`) to the data user. If there is no such f in the existing files, f must be a non-existing file. The cloud then first searches $f + "@"$ in the non-existing files, guessing that f is a prefix of some existing file. If the search is successful, the cloud returns the tuple ($f + "@"$, HMAC.Tag($f + "@"; K_1$), `NULL`) back. If the search fails, f must be different from all existing files. In this case, for every possible prefix α of f, the cloud searches $\alpha + "\#"$ in the non-existing files. As a result, the cloud returns the matched one tuple ($\alpha + "\#"$, HMAC.Tag($\alpha + "\#"; K_1$), `NULL`). Note that when the cloud receives the outsourced data, the cloud can store the data in some tree structure so that search operations can be conducted efficiently.

VERIFY. When the data user receives returned tuple from the cloud, the data user first checks the format of the first entry in the tuple. There are only three kinds of correct tuples: f, or $f+$"@", or $\alpha+$"#", where f is the queried filename and α is a prefix of f. If the format validation fails, the data user rejects the result. If the tuple is in a correct format, the data user then verifies whether the MAC is correct by using the secret key K_1. If the MAC is also valid, the data user accepts the result. Otherwise, the result is rejected.

V. A PRIVACY-PRESERVING PROTOCOL FOR VERIFIABLE FILE SEARCH

In this section, we enhance the baseline protocol presented in Section IV to further preserve the privacy of filenames.

A. A Full-fledged Protocol with Privacy Protection

Note that when the data owner outsources the data to the cloud, the tuples $(f, \mathsf{HMAC.Tag}(f; K_1), \bullet)$ is transmitted to the cloud, where '$\bullet$' means '`file content`' or '`NULL`'. The leakage of privacy originates from the clear filename f. If we mask the filename in some secure way, the privacy can be well protected. The basic idea to preserve the filename privacy is to choose another secret key for HMAC, which can mask the filename from the cloud, i.e. we make up a nickname using HMAC for every file. Since the MAC of a filename is a string, the nickname is computed as such a string under a new secret key of HMAC.

OUTSOURCE. All details are the same as the baseline protocol described in Section IV-C except the tuple sent to the cloud. The data owner sends $(\mathsf{HMAC.Tag}(f; K_2)$, $\mathsf{HMAC.Tag}(\mathsf{HMAC.Tag}(f; K_2); K_1)$, $\bullet)$ to the cloud, where K_2 is another independent secret key. The data owner also shares this key with the data user.

QUERY. Suppose the data user wants to query a file f. If f does exist, the data user just needs to send $\mathsf{HMAC.Tag}(f; K_2)$ to the cloud for a query. If f does not exist, the data user can send all possible masked corresponding filenames in the non-existing file set. In this case, the data user can send $\mathsf{HMAC.Tag}(f + \text{"@"}; K_2)$ and all possible $\mathsf{HMAC.Tag}(\alpha + \text{"\#"}; K_2)$'s, where α is a prefix of f. Hence, the data user sends the tuple

$$\begin{pmatrix} \mathsf{HMAC.Tag}(f_1; K_2), \\ \mathsf{HMAC.Tag}(f_2; K_2), \\ \mathsf{HMAC.Tag}(f_3; K_2), \\ \cdots, \\ \mathsf{HMAC.Tag}(f_{\mathrm{length}(f)+2}; K_2) \end{pmatrix} \tag{3}$$

to the cloud to query f, where $f_1 = f$, $f_2 = f + \text{"@"}$, $f_3 = \alpha_1 + \text{"\#"}$, $\cdots$, $f_{i+2} = \alpha_i + \text{"\#"}$ and α_i is the prefix of f with length i for $i = 1, \cdots, \mathrm{length}(f)$.

SEARCH. When the cloud receives a tuple that queries a file, the cloud searches each entry in the tuple until one matched in the outsourced data is found. Suppose the one is with index j in the query tuple. Then, the cloud returns the index j of that entry and the corresponding data tuple $(\mathsf{HMAC.Tag}(f_j; K_2)$, $\mathsf{HMAC.Tag}(\mathsf{HMAC.Tag}(f_j; K_2); K_1)$, $\bullet)$ back.

VERIFY. After receiving an index j and a data tuple $(\mathsf{HMAC.Tag}(f_j; K_2)$, $\mathsf{HMAC.Tag}(\mathsf{HMAC.Tag}(f_j; K_2); K_1)$, $\bullet)$, the data user uses the secret key K_1, K_2 to check whether $\mathsf{HMAC.Tag}(f_j; K_2)$ and $\mathsf{HMAC.Tag}(\mathsf{HMAC.Tag}(f_j; K_2); K_1)$ are expected or not (i.e. whether the MACs are correct or not). If yes, the data user accepts the answer; otherwise the answer is rejected.

B. Summary of the Full-fledged Protocol

We now formally present our full-fledged protocol, which enables verifiable and privacy-preserving file search over outsourced data. Our verifiable file search protocol VS = (KeyGen, Outsource, Query, Search, Verify) contains five algorithms listed as follows:

- $\mathsf{KeyGen}(1^\lambda)$: After receiving a security parameter λ as an input, the data owner runs this algorithm to generate two secret keys K_1 and K_2 for the HMAC algorithm. The data owner also shares them with the data user.
- $\mathsf{Outsource}(F; K_1, K_2)$: After receiving a collection of files F and the secret keys K_1 and K_2 as inputs, the data owner runs Algorithm 1 to get F_1 and F_2, which separate all possible filenames. For each $f \in F_1$, the data owner sends $(\mathsf{HMAC.Tag}(f; K_2)$, $\mathsf{HMAC.Tag}(\mathsf{HMAC.Tag}(f; K_2); K_1)$, `file content`$)$ to the cloud. For each $f \in F_2$, the data owner also sends $(\mathsf{HMAC.Tag}(f; K_2)$, $\mathsf{HMAC.Tag}(\mathsf{HMAC.Tag}(f; K_2); K_1)$, `NULL`$)$ to the cloud.
- $\mathsf{Query}(f; K_2)$: After receiving a queried file f and the secret key K_2 as inputs, the data user sends $q = (\mathsf{HMAC.Tag}(f_1; K_2)$, $\mathsf{HMAC.Tag}(f_2; K_2)$, $\mathsf{HMAC.Tag}(f_3; K_2)$, $\cdots$, $\mathsf{HMAC.Tag}(f_{\mathrm{length}(f)+2}; K_2))$ to the cloud as a query, where $f_1 = f$, $f_2 = f + \text{"@"}$, $f_3 = \alpha_1 + \text{"\#"}$, $\cdots$, $f_{i+2} = \alpha_i + \text{"\#"}$ and α_i is the prefix of f with length i for $i = 1, \cdots, \mathrm{length}(f)$.
- $\mathsf{Search}(q)$: After receiving the query tuple q as an input, the cloud searches each entry in q. The cloud then returns $(\chi, \Gamma) = (j$, $\mathsf{HMAC.Tag}(f_j; K_2)$, $\mathsf{HMAC.Tag}(\mathsf{HMAC.Tag}(f_j; K_2); K_1)$, $\bullet)$ to the data user, which contains the query result and a correctness proof.
- $\mathsf{Verify}(f, \chi, \Gamma; K_1, K_2)$: After receiving the returned answer χ, correctness proof Γ, the original query filename f, and the secret keys K_1 and K_2 as inputs, the data user obtains f_j using χ and f. After that, the data user checks whether the proof has the right MACs or not. If MACs are correct, the data user accepts the result; otherwise, the result is rejected.

VI. ANALYSIS OF THE PROTOCOL

In this section, we analyze the correctness, security, privacy and performance of our full-fledged protocol.

A. Correctness

By inspecting the property of Algorithm 1 which is shown in Proposition 1, we can see that our protocol VS = (KeyGen, Outsource, Query, Search, Verify) is indeed correct. Suppose

TABLE I
THEORETICAL PERFORMANCE

	Computation	Storage	Communication
Data Owner	$O(n^2)$	$O(1)$	$O(n)$
Data User	$O(1)$	$O(1)$	$O(1)$
Cloud	$O(\log n)$	$O(n)$	$O(1)$

all parties follow the protocol and the data user queries an arbitrary file f. The query sent to the cloud is defined by Equation (3). If f does exist, then $\mathsf{HMAC.Tag}(f; K_2)$ must lie in the outsourced data, which can be found in VS.Outsource. If f does not exist, then either f is a prefix, or f can be represented by using the wildcard representation $\alpha + "\#"$, where α is a prefix of f. In both cases, f could be represented as a special string in F_2, which is also outsourced in VS.Outsource. To summarize, the query Equation (3) contains all possible cases for any file f.

B. Security

We argue that our developed full-fledged protocol in Section V is secure according to Definition 1 if HMAC is secure. To the best of our knowledge, there is no effective attack to HMAC [11]. Thus, our protocol should be secure as well.

Theorem 1. *The full-fledge protocol is secure if HMAC is secure.*

Proof Sketch. We briefly introduce the main idea of the proof due to page limits: Suppose that our protocol is not secure, i.e. there is a cloud $\mathcal{A}$ that can fool the data user. We construct another adversary $\mathcal{A}'$ who is capable of breaking the security of HMAC. $\mathcal{A}'$ employs a forgery which is returned by the malicious cloud $\mathcal{A}$ to find a forgery for HMAC. Thus, if HMAC is secure, no forgery exists, which implies the security of the protocol. We plan to include a full proof in an extended version of this paper.

□

C. Privacy and Integrity

In our designed protocol, the privacy of the filenames and file contents is well protected as well. For filenames, if an adversary can find out what the name is, the cloud can forge a valid message-MAC pair. This contradicts with the security of HMAC. For file contents, it is encrypted and authenticated under other independent keys using a secure encryption algorithm and a secure message authentication code algorithm as we assumed in Section II. Thus, it is also well protected.

D. Overhead

We now analyze the theoretical asymptotic performance of our protocol in terms of computation, storage, and communication cost. Table I summarizes the analytical results, which match with our experiment results shown in Section VII.

Detailed analysis. The data owner needs to separate all possible filenames and then outsource the data to the cloud. Algorithm 1 handles the separation. Denote n as the number of existing files, LEN as the maximum length of a filename that is allowed by the file system, and SZ as the size of the alphabet Σ. For a direct implementation of this algorithm, Line 2 takes time $O(n)$; Line 3 takes time $O(\mathsf{LEN})$; Line 5 takes time $O(\mathsf{SZ})$; Line 7 takes time $O(\mathsf{LEN} \cdot n)$; Line 12 takes time $O(\mathsf{LEN} \cdot n)$. Hence, the total time complexity for Algorithm 1 is $O(n) \times O(\mathsf{LEN}) \times [O(\mathsf{SZ}) \times O(\mathsf{LEN} \cdot n) + O(\mathsf{LEN} \cdot n)]$. That is, the time complexity of the separation process is $O(\mathsf{SZ} \cdot \mathsf{LEN}^2 \cdot n^2)$.

Note that only the number of files n changes with the number of files; SZ and LEN are constants. Hence, the asymptotic time complexity is $O(n^2)$. In addition, this operation only needs to be done once and can be *amortized* in the following queries. For storage, because the data owner only needs to store two secret keys, the complexity of the storage is $O(1)$, as a constant. Nevertheless, the cloud needs to store both the data and the MACs for F_1 and F_2. Note that the size of F_1 is $O(n)$, the size of F_2 is also $O(n)$, and the size of a MAC is constant. Then, the communication is $O(n)$, which is caused by outsourcing the data. The same analysis also applies for the data user and the cloud.

Compared with previous researches, most of the existing efforts in the encrypted keyword area assume that the cloud is honest except [3]. In addition, the complexity of [3] is exponential. Differently, the complexity of our developed protocol is much smaller as shown in Table I. Our developed protocol is also light-weight while the authenticated B$^+$-tree employed in the outsourced database area [9], [10] is cumbersome. This is due to the fact that maintaining an authenticated B$^+$-tree is very complicated [18]. Moreover, a practically sound security definition is proposed here and the security of the proposed protocol is shown rigorously, which makes a step forward towards an in-depth understanding of cloud security issues.

VII. PERFORMANCE EVALUATION

We implemented both the baseline and full-fledged protocols. We conducted experiments and measured the computation, storage and communication cost of these protocols. We measure the computation time of each algorithm in the proposed verifiable file search protocol VS = (KeyGen,Outsource, Query, Search, Verify). We also measure the size of all possible filenames $\Sigma^{(*)} = F_1 \cup F_2$, which reflects the storage overhead. For communication overhead, because it is constant and very small, which depends on the MAC size and the speed of the network, we neglect it in our evaluation.

The experimental setup is listed as follows. The baseline protocol and the full-fledged protocol are implemented using Java 1.7 on a PC with an Intel i3 3.1G CPU and 4GB memory. The source code of the protocol implementation and evaluation as well as the experimental data can be found online [19]. We use public data sets to validate the performance of our protocol. Thus, all the performance data shown in this section can be reproduced by using our source code [19] and the public data available online [20], [19]. As a proof of concept to validate the effectiveness and feasibility of our proposed protocol, we use Java to implement our protocol. Note that other more

TABLE II
DATA SETS AND STORAGE COST IN BYTES

	Toy	RFC50	RFC100	RFC200	RFC400	Python
#Files	4	50	100	200	400	92
Basic	42424	1346088	2629080	5283624	9644648	4088272
Full	122592	3610104	7039920	13840144	26716696	10876272

efficient programming language can be used to achieve better performance.

We choose six test data sets. The first is only a toy test set consisting of only four files. This data set helps confirm the correctness of Algorithm 1 for constructing all possible filenames. The following four data sets come from the RFC documents, which can be downloaded online [20]. The test 'RFC50' contains RFCs from 'rfc1.txt' to 'rfc50.txt'. Note that some RFCs are 'pdf' files or 'ps' files. This is also true for 'RFC100', 'RFC200' and 'RFC400'. The RFC data set has similar filenames. To deal with this issue, the sixth data contains all files in the home directory '...\Python27\include' of the Python 2.7 installation [21]. This data set can be obtained in two ways: downloading it from [19] or installing Python 2.7 directly. Table II lists the basic information of these test data sets.

The size of all possible filenames constructed by Algorithm 1 is shown in Table II. As we can see, for the baseline protocol, the minimal storage cost is roughly 41.5 KB for the 'Toy' data set whereas the maximal storage cost is roughly 9.2 MB for the 'RFC400' data set. For the full-fledged protocol with the same data set, the minimum and the maximum are 0.12 MB and 25.5 MB, respectively. It is worth noting that Java stores a character using 2 bytes and Java use some metadata to store an object. This means that in practice the storage cost of our protocols can be reduced at least a half if a proper character encoding is used.

There is a noticeable difference for the 'Python' data set: the storage cost is larger than the 'RFC100' data set. This can be expected as the theoretical analysis focuses on the worst-case and asymptotic performance whereas the experiments measure the exact storage cost. We note that the exact storage cost also depends on the filename structures according to Algorithm 1. Nonetheless, the storage cost grows linearly when more files are outsourced asymptotically. For achieving verifiability of the file search, additional storage is required whereas storage becomes cheap nowadays.

Table III contains the time consumption for the baseline protocol. It indicates that the operation $\mathsf{VS.Outsource}$ takes much more time than other operations. For example, the maximal time is about 1.6 s when the 'RFC400' data set is used. It is reasonable because this data set has more files than others. We can also observe that $\mathsf{VS.Query}$ takes little time.

In our implementation, we randomly generates an existing file and a non-existing file. We then measure the time and compute the mean value after 20 runs. Note that the time cost is a little higher than practice because a filename is generated according to a uniform distribution. In practice, a data user may just needs to send the query filename to the cloud and the time cost is actually constant. Note that for $\mathsf{VS.Search}$, it takes little time too.

TABLE III
BASIC PROTOCOL COMPUTATION TIME IN NANOSECONDS

Benchmark	Outsource	Query	Search	Verify
Toy	9945914	190	139796	13449
RFC50	129859029	157	8272	6174
RFC100	264431126	215	8711	6612
RFC200	625044487	223	11897	9675
RFC400	1595279912	273	12878	10362
Python	249927082	206	10717	10258

TABLE IV
FULL PROTOCOL COMPUTATION TIME IN NANOSECONDS

Benchmark	Outsource	Query	Search	Verify
Toy	17870332	57704	13904	24929
RFC50	272240334	55859	14550	16164
RFC100	540713787	61487	15402	15883
RFC200	1247998618	67959	15576	16429
RFC400	2833943104	62066	356906	16131
Python	589483592	92069	15626	17563

In our implementation, the filenames are stored using the 'TreeMap' class in Java, which is a balanced binary tree data structure. Hence, the search time depends on two parts: one is searching the tree and the other is string comparison, both of which depends on Java detailed implementations of the corresponding classes. Nonetheless, the key is that the time cost is very small. The same also holds for $\mathsf{VS.Verify}$ as we can see from Table III. Lastly, it is worth noting that the time cost also depends on the implementation of the protocol and detailed information about our implementation can be found [19].

Table IV illustrates the time cost for our full-fledged protocol. As we can see, all observations from Table III can be applied here as well. For example, $\mathsf{VS.Outsource}$ takes much more time than other operations and the time consumption for file query, search and verification is quite small. The difference is that all operations takes a little more time than the operations in the baseline protocol as the filename is processed using MAC in order to protect the privacy.

VIII. RELATED WORK

In this section, we review the most recent and important related work in the cloud storage security and cryptography areas.

Closely related to cloud storage security, verifiable fuzzy keyword searches, authenticated data structures, and verifiable outsourced databases have been developed in the past [22], [3]. For example, in terms of fuzzy keyword searches, Li *et al.* in [22] proposed a protocol that enables fuzzy keyword searches in the cloud. More practically, Chai and Gong in [3] considered the verifiable keyword search problem for the first time, which also won the best paper award in the conference. Nonetheless, the complexity of the protocol proposed in [3] is $O(\mathsf{SZ}^{\mathsf{LEN}})$ which is exponential.

Similar to the keyword search problem, how to search in encrypted data has also been studied in the cryptography community [8], [12]. There are some other related research efforts on zero-knowledge set [23] that can be used to solve the verifiable file search problem. Benabbas *et al.* in [24] discussed verifiable polynomial computations over a finite field and also noted a similar idea using prefixes of strings over $0, 1$ to solve the zero-knowledge set problem. However, either a rigorous and in-depth investigation is still lacking, or the efficiency issue is not resolved.

A number of research efforts have been carried out on outsourced databases [9], [10], [25]. For example, Pang *et al.* in [9] proposed to use aggregate signatures [26] to authenticate the search result on an outsourced database. Differently, the computations in our proposed protocols involve only private-key cryptography and thus are highly efficient. Pang *et al.* [10] proposed to use hash authentication trees to ensure the correctness of a text search engine. Hash trees uses only private-key cryptography and thus can be efficient. Nonetheless, the construction and operation of such a hash tree incurs a great cost [18].

IX. Conclusion

In this paper, we addressed the issue of the verifiable file search problem and proposed two lightweight, efficient and secure protocols. The baseline protocol enables a user to verify whether or not a cloud cheats when a file is searched and the enhanced full-fledged protocol further preserves the privacy of users who request filenames. We formally proved the correctness of our developed protocols and their security and privacy-preserving along with the overhead. We implemented both the baseline and full-fledged protocols. Using a real-world data set, we conducted experiments and measured the computation, storage, and communication costs of the proposed protocols. Our experimental data shows the effectiveness and efficiency of the proposed protocols.

Acknowledgments

This work was supported in part by the China Natural Science Foundation Project of CQ CSTC (No. c-stc2013jcyjA40001) and the Fundamental Research Funds for the Central Universities (No. CDJZR13185501), and by the US National Science Foundation grants CNS-1116644 and CNS-1117175. Any opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] C. S. Alliance, "Security guidance for critical areas of focus in cloud computing," https://cloudsecurityalliance.org/research/security-guidance/, 2011.

[2] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, "Privacy-preserving multi-keyword ranked search over encrypted cloud data," in *IEEE INFOCOM*, 2011.

[3] Q. Chai and G. Gong, "Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers," in *IEEE ICC*, 2012.

[4] J. Xu and E.-C. Chang, "Towards efficient proofs of retrievability," in *ACM ASIACCS*, 2012.

[5] C. Wang, S. S. Chow, Q. Wang, K. Ren, and W. Lou, "Privacy-preserving public auditing for secure cloud storage," *IEEE Transactions on Computers*, vol. 62, no. 2, pp. 362–375, 2013.

[6] K. Yang and X. Jia, "An efficient and secure dynamic auditing protocol for data storage in cloud computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 9, pp. 1717–1726, 2013.

[7] A. Juels and B. Kaliski Jr, "Pors: Proofs of retrievability for large files," in *ACM CCS*, 2007.

[8] D. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *IEEE Symposium on Security and Privacy*, 2000.

[9] H. Pang, J. Zhang, and K. Mouratidis, "Scalable verification for outsourced dynamic databases," in *VLDB*, 2009.

[10] H. Pang and K. Mouratidis, "Authenticating the query results of text search engines," in *VLDB*, 2008.

[11] NIST, "The Keyed-Hash Message Authentication Code (HMAC)," http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf, 2008.

[12] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *ACM CCS*, 2012.

[13] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *Financial Cryptography and Data Security*. Springer, 2013, pp. 258–274.

[14] M. R. Albrecht, K. G. Paterson, and G. J. Watson, "Plaintext recovery attacks against SSH," in *IEEE S&P*, 2009.

[15] J. P. Degabriele and K. G. Paterson, "On the (in) security of IPsec in MAC-then-encrypt configurations," in *ACM CCS*, 2010.

[16] O. Goldreich, *Foundations of Cryptography: Basic Tools*. Cambridge University Press, 2000.

[17] M. Sipser, *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[18] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic authenticated index structures for outsourced databases," in *ACM SIGMOD*, 2006.

[19] F. Chen, "Source code for verifiable file search," https://sites.google.com/site/chenfeiorange/verifiable-file-search, 2014.

[20] RFC, "Request for comments database," http://www.ietf.org/rfc.html.

[21] Python, "Python 2.7," http://www.python.org/, 2013.

[22] J. Li, Q. Wang, C. Wang, N. Cao, K. Ren, and W. Lou, "Fuzzy keyword search over encrypted data in cloud computing," in *IEEE INFOCOM*, 2010.

[23] S. Micali, M. Rabin, and J. Kilian, "Zero-knowledge sets," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2003.

[24] S. Benabbas, R. Gennaro, and Y. Vahlis, "Verifiable delegation of computation over large datasets," *CRYPTO'11*, 2011.

[25] W. Cheng and K.-L. Tan, "Query assurance verification for outsourced multi-dimensional databases," *Journal of Computer Security*, vol. 17, no. 1, pp. 101–126, 2009.

[26] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "A survey of two signature aggregation techniques," *RSA Cryptobytes*, vol. 6, no. 2, pp. 1–10, 2003.