Cooperative Private Searching in Clouds

Qin Liu[a,b], Chiu C. Tan[a], Jie Wu[a,*], and Guojun Wang[b]

[a] *Department of Computer and Information Sciences*
Temple University
Philadelphia, PA 19122, USA
[b] *School of Information Science and Engineering*
Central South University
Changsha, Hunan Province, P. R. China, 410083

Abstract

With the increasing popularity of cloud computing, there is increased motivation to outsource data services to the cloud to save money. An important problem in such an environment is to protect user privacy while querying data from the cloud. To address this problem, researchers have proposed several techniques. However, existing techniques incur heavy computational and bandwidth related costs, which will be unacceptable to users. In this paper, we propose a cooperative private searching (COPS) protocol that provides the same privacy protections as prior protocols, but with much lower overhead. Our protocol allows multiple users to combine their queries to reduce the querying cost while protecting their privacy. Extensive evaluations have been conducted on both analytical models and on a real cloud environment to examine the effectiveness of our protocol. Our simulation results show that the proposed protocol reduces computational costs by 80% and bandwidth cost by 37%, even when only five users query data.

Keywords: Cloud computing, cooperative, privacy preserving, private searching, cost-effective.

*Correspondence Author: Tel: (215) 204-8450, Fax: (215) 204-5082
Email address: jiewu@temple.edu (Jie Wu)

1. Introduction

Cloud computing as an emerging technology has attracted a lot of attention in recent years. The benefits of utilizing the cloud (lower operating costs, elasticity, and so on) come with a tradeoff. Users will have to entrust their data to a potentially untrustworthy cloud provider. As a result, cloud security has become an important problem for both industry and academia [1, 2].

One important security problem is the potential privacy leakages that may occur when outsourcing data to the cloud. For instance, let us consider the application scenario as shown in Fig. 1. When the users want to search for some files, they will query the cloud with certain keywords. The cloud will evaluate the query and return the necessary files to the users. During this process, the cloud will know what files a user is interested in from observing the query and the type of files returned to that user. Preventing a leak of this type of information to the cloud is difficult since the cloud must have access to the information to efficiently return the appropriate files to the users.

A naïve solution to this problem is for the user to request *all* of the files from the cloud. While this does provide the necessary privacy, the overhead will be excessive. More efficient protocols, known as *private searching protocols*, have been proposed [3, 4, 5, 6, 7, 8, 9] to address this problem. In the private searching protocol, files stored in the cloud are in the clear forms. The user will send a special type of query that is encrypted under *Paillier cryptography* [10] to the cloud so that the cloud never learns *what keywords* the users are searching for. Using the *homomorphic properties* of Paillier cryptography, the cloud will be able to return the appropriate files to the user without knowing *which files* have been returned.

However, to protect user privacy, a private searching protocol must require the cloud to process the encrypted query on *every* file in a collection. Omitting any file will inform the cloud that particular file is not related to the user's query and leak privacy. Therefore, it will quickly become a performance bottleneck when the cloud needs to process thousands of queries over a collection of hundreds of thousands of files. Alternatively, if we can combine more than one queries together, we can save the overhead by reducing the number of queries that the server has to process.

In this paper, we propose a new private searching protocol termed *CO-operative Private Searching* (COPS). This protocol reduces the computation and communication costs while providing similar privacy protection as in prior protocols. Our solution introduces an *aggregation and distribution layer*

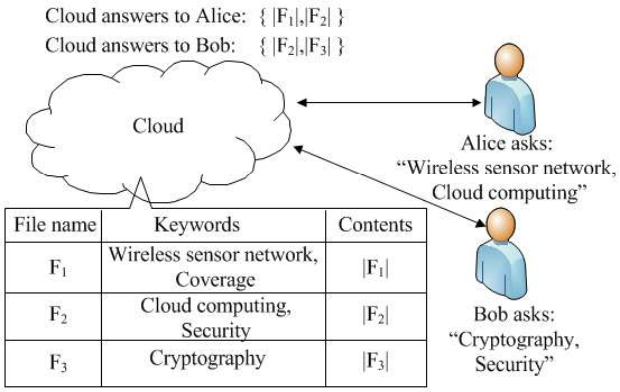

Figure 1: Application scenario. Files F_1, F_2, and F_3 stored in the cloud are described with keywords "Wireless sensor network, Coverage", "Cloud computing, Security", and "Cryptography", respectively. Alice uses keywords "Wireless sensor network, Cloud computing", and Bob uses keywords "Cryptography, Security" to query data from the cloud.

(ADL)-a middleware layer between the users and the cloud. Users will first send their queries to the ADL, which will combine queries and query the cloud on the users' behalf. In this way, the cloud needs to execute query only once to return files matching all users' queries to the ADL. Furthermore, under the ADL, the files interested by many users need to be returned only once. Thus, the communication cost will also be reduced. For example, Alice and Bob first send their queries to the ADL, which will send to the cloud a combined query containing keywords "Wireless sensor network, Cloud computing, Cryptography, Security". Given the combined query, the cloud only needs to execute the private search once to return $\{|F_1|, |F_2|, |F_3|\}$ to the ADL, which will return $\{|F_1|, |F_2|\}$ to Alice and $\{|F_2|, |F_3|\}$ to Bob. Here, the computation and communication costs at the cloud are saved by 50% and 25%, respectively, compared to existing private searching protocols.

We envision that an ADL will be deployed in medium and large organizations that have outsourced their data operations to a cloud. Such organizations have thousands of users querying the cloud, and thus will have an incentive to reduce the querying cost. The ADL is analogous to a web cache or proxy server maintained by the organization and can be configured based on different policies. An organization that is concerned with cost for instance, can require the ADL to wait for a longer period of time to aggregate sufficient queries before querying the cloud. A key feature of the COPS protocol is that every user's privacy is protected from the cloud, the ADL,

as well as from other users.

Our protocol is suitable for a *cost-effective* environment, where the user can tolerate a certain delay while retrieving information from the cloud in order to reduce cost. The COPS protocol will incur some processing delay for aggregating queries. However, the degree of aggregation can be controlled through a time-out mechanism to meet a given processing delay requirement. When the time-out is set to zero, this is degraded to the sequential queries.

The COPS protocol allows us to provide the same privacy protection at a much lower cost. The simulation results show that our protocol reduces computational costs by 80% and bandwidth cost by 37%, even when only five users query data from the cloud simultaneously. We make the following contributions in this paper:

1. To the best of our knowledge, the COPS protocol is the first cooperative private searching protocol for a cloud environment. The COPS protocol introduces an ADL to aggregate multiple queries and to divide results to each user, which outperforms existing private searching protocols while providing the same privacy protection as before.
2. We thoroughly analyze the security and complexity of the COPS protocol. Our protocol protects user query privacy from the cloud, the ADL, and other users.
3. Extensive experiments are performed using a combination of simulations and real cloud deployments to validate our schemes. Simulation experiments indicate that our protocol performs better than the existing scheme, even when there are only few users executing searches simultaneously.

The remainder of this paper is organized as follows: We introduce related work in Section 2 and present technical preliminaries in Section 3. Then, we describe the COPS protocol scheme in Section 4 and theoretically analyze its performance and security in Section 5. Next, we evaluate the performance of the COPS protocol in Section 6 and provide additional discussion in Section 7. Finally, we conclude this paper in Section 8.

2. Related work

Our work is on protecting user privacy while searching data on untrusted servers. User privacy can be classified into search privacy and access privacy [11]. Search privacy means that the servers knows nothing about what the

user are searching for, and access privacy means that the cloud knows nothing about which files are returned to the user. There has been a lot of work conducted in this field including private searching [3, 4, 5, 6, 7, 8, 9], private information retrieval (PIR) [12, 13, 14], and searchable encryption [15, 16, 17, 18, 19, 20], where user privacy can be protected in private searching and PIR, but only search privacy can be protected in searchable encryption.

Private searching was first proposed by [3] (also referred to as the Ostrovsky protocol), where data is stored in the clear form, and the query is encrypted with the Paillier cryptosystem [10] that exhibits the homomorphic properties. The server processes the encrypted query on each file and stores the encrypted file into a compact buffer, with which the user can successfully recover all wanted files with high probability. Since the query and the results are encrypted under the user's public key, the server cannot know the user's interests. The key merit of their work is that the buffer size depends on the number of files matching the query and is independent of the number of files stored on the server. Therefore, private searching can provide the same level of privacy as downloading entire database from the server while incurring significantly less communication costs. Among various extensions, the work by [4, 5] reduced the communication cost by solving a set of linear equations to recover data; the work by [6] and [7] presented an efficient decoding mechanism for private searching; the work by [8] applied private searching to achieve public shuffling; and the work by [9] used private searching to protect user query privacy in MapReduce [21, 22]. The main drawback of existing private searching protocols is that both the computation/communication cost will grow linearly with the number of users executing searches.

PIR was first introduced in [12], where the data is viewed as an n-bit string $x = x_1x_2, \ldots, x_n$, and a user retrieves the bit x_i while keeping the index i private from the database by accessing multiple replicated database copies. The work by [13] provided a single-database PIR protocol to further reduce incurred communication costs. Recently, the work by [14] applied the private information retrieval technique to a relational database by hiding sensitive constants contained in the predicates of a query. There are two main differences between PIR and private searching. First, the communication costs in existing PIR schemes depend on the size of the entire database other than the size of retrieved messages. Second, PIR is first proposed to let a user to retrieve a bit from a database without letting the database know which bit is retrieved. Although some work addressed the problem of retrieval files by keywords, none of them can support multi-keyword search.

Searchable encryption may be viewed as the flip side of private searching and PIR, where the user conducts searches on encrypted data. Searchable encryption was first proposed by [15], where both the user query as well as the data is encrypted under a symmetric key setting. Therefore, only the users with the symmetric key can encrypt data and generate queries. The work by [16] proposed the first public key-based searchable encryption protocol, where anyone with the public key can encrypt data, but only users with the private key can generate queries. The work by [17] first used the Bloom filter to build an index of keywords for each file. The work by [18] also developed a similar per-file index protocol. The work by [19] encrypts files and queries with Order Preserving Symmetric Encryption (OPSE) [23] and utilizes keyword frequency to rank results; the later work by [20] uses the secure KNN technique [24] to rank results based on inner products. The main difference between all these work and ours is that in searchable encryption the cloud will know which files (file identifiers) are returned to each user, even if the file contents are encrypted. Thus, the cloud may deduce whether two users are interested in the same files.

Our protocol is built on top of the private searching technique. We propose a protocol to enable many users to cooperatively execute private searches to reduce both computation and communication costs.

3. Preliminaries

In this section, we will first formulate the problem for this work, and then provide the security requirements. Finally, we will outline the Ostrovsky protocol, which serves as a base of the COPS protocol.

3.1. Problem formulation

We consider a cost-effective cloud environment where a user can tolerate a certain delay while retrieving information from the cloud in order to reduce cost. Many unencrypted files are stored in a potentially untrusted cloud. The users can query the cloud to retrieve files that they are interested in. When the users do not want the cloud to know their interests, the private searching technique can be adopted to protect user query privacy.

Now suppose there are n users, where each user issues a query to the cloud using the private searching protocol. If each user independently requests the data from the cloud, the cloud needs to execute private searches n times, and return results to n users, respectively. Fig. 2 shows that both the

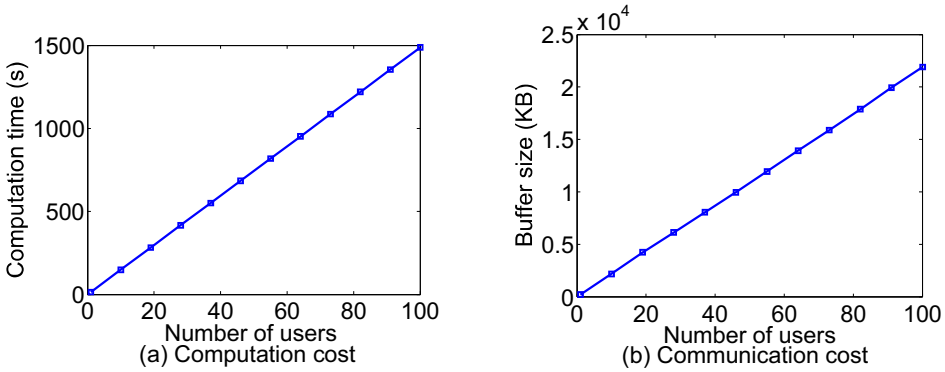

Figure 2: Computation/communication costs in existing protocols. There are 1,000 files stored in the cloud. Each file is described by 1-5 keywords, and each user randomly chooses 1-5 keywords from a dictionary of 1,000 keywords.

computation and communication costs at the cloud grow linearly with the number of users in private searching protocols.

In this paper, we propose the COPS protocol that introduces an aggregation and distribution layer (ADL) that acts like an aggregator and distributor as shown in Fig 3. The ADL is deployed in an medium and large organization that have outsourced their data operations to a cloud. For ease of explanation, in this paper, we only use a single ADL, but multiple ADLs can be deployed as necessary. The users, who want to retrieve files that they are interested in, form a group. Each user will first send their queries to the ADL, which will in turn query the cloud on the users' behalf and return the appropriate files to each user. In this way, the computational cost at the cloud will be largely reduced, since the cloud needs to execute the private search only once, no matter the number of users. The communication cost at the cloud will also be largely reduced, since all users conceivably have common interests and the number of files matching all users' queries will not grow linearly with the number of users.

3.2. Security and privacy requirements

Our goal is to protect each user's privacy while querying data in the cloud. The communication channels are assumed to be secured under existing security protocols such as SSL to protect user privacy during information transferring. There are three types of adversaries: the cloud, the ADL, and other users. The cloud operated by a third party may leak user privacy for making profits. The ADL as an aggregator will collect all messages and may

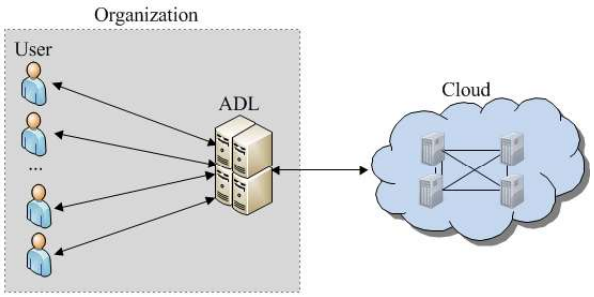

Figure 3: System model of COPS protocol.

become a bigger target. To avoid possible information leaking, it is required to hide user privacy from the ADL. A small number of malicious users may want to know other users' privacy. These adversaries are assumed to be honest but curious. That is, they will obey our protocol, but they still want to know some additional information.

The ADL is assumed to not collude with any other entities. However, malicious users may either work together or collude with the cloud to know other users' interests. This assumption is reasonable, since the ADL, maintained by the organization, is reliable and independent, and thus has no incentive to collude with other entities during our protocol. This assumption has also been made in previous research by other researchers, e.g., the proxy re-encryption systems [25, 26], where the proxy server is assumed to not collude with other entity to ensure system-wide security.

We consider our protocol to fail if any of the following cases is true:

Case 1. The cloud knows keywords or file contents queried by any user.

Case 2. The ADL knows keywords or file contents queried by any user.

Case 3. The user knows keywords or file contents queried by other users.

3.3. Outline of the Ostrovsky protocol

The Ostrovsky protocol [3] relies on a public key cryptosystem, Paillier cryptosystem [10]. Let $E(m)$ denote the encryption of a plaintext m. The Paillier cryptosystem has the following homomorphic properties:

- $E(a) \cdot E(b) = E(a + b)$
- $E(a)^b = E(a \cdot b)$

The homomorphic properties are achieved as follows: the ciphertexts of a and b can be considered as g^a and g^b, where g is a random generator.

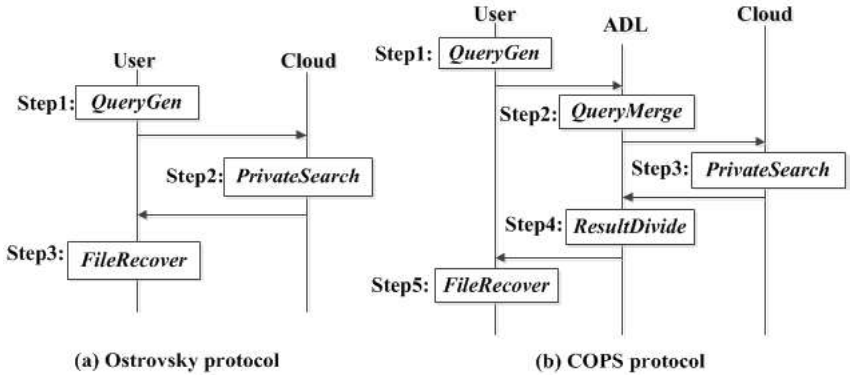

Figure 4: Working processes of the Ostrovsky protocol and COPS protocol.

$g^a \cdot g^b = g^{a+b}$, i.e., the product of ciphertexts of a and b is equal to the ciphertext of $a + b$; $(g^a)^b = g^{a \cdot b}$, i.e., the ciphertext of a to the power of b is equal to the ciphertext of $a \cdot b$.

With the Paillier cryptosystem, the Ostrovsky protocol enables the cloud to perform certain operations, such as *multiplication* and *exponentiation*, on ciphertext directly. Given the resultant ciphertext, the user can obtain the corresponding plaintext that is processed *addition* and *multiplication* operations. We briefly outline the working process of the Ostrovsky protocol while many users are querying data from the cloud, as shown in Fig. 4-(a).

Step 1: Each user runs the QueryGen algorithm to send an encrypted query to the cloud. Based on a public dictionary *Dic* that consists of an array of keywords, user i generates a query Q_i, where each entry is set to 1 if corresponding keyword in *Dic* is chosen, otherwise the entry will be 0. Then, the user encrypts each entry of the query with the Paillier cryptosystem under his public key. The query is an array of encrypted 0s and 1s. Since the Paillier cryptosystem is semantically secure, the ciphertext of every 1 or 0 is different from other 1s or 0s. Therefore, given the encrypted query, the cloud cannot guess the user's interest. The encrypted query as well as the *estimated* number of files matching the query will be sent to the cloud.

Step 2: The cloud runs the PrivateSearch algorithm to return a compact buffer to each user. For each user's query, the cloud needs to execute the private search once as follows: the cloud processes the encrypted query on each file stored on the servers to obtain the encrypted file, and stores the ciphertext into an encrypted buffer, which will be returned to the user. To reduce the communication cost, the number of buffer entries (also referred to

as buffer size) only depends on the number of files matching the query, which is smaller than the number of files stored in the cloud. Thus, it is unavoidable that one entry may store more than one file. Due to the homomorphic properties of the Paillier cryptosystem, each file mismatching the query is encrypted to 0, which has no impact on the matched files, even if they are stored in the same entry. Although a collision will result and all files will be lost if more than one matched file is stored in the same entry, it is proven that all matched files can be recovered with high probability, when each file is mapped multiple times into a sufficiently large buffer. The buffer size is determined as follows: if user i wants to retrieve f_i files with a failure probability that is smaller than p, each file should be mapped $log(f_i/p)$ times *randomly* into a buffer of size $2f_i \cdot log(f_i/p)$.

Step 3: Each user runs the FileRecover algorithm to recover matched files. Given the buffer, each user uses his private key to decrypt the buffer entry by entry to recover files.

Since the query and the results are encrypted under the user's public key, and the cloud processes each file similarly, the user can protect his query privacy from the cloud. This protocol also provides a collision-detection mechanism to let the user get rid of the conflicted file copies. We refer the readers to [3] for more details.

4. COPS protocol

In this section, we will first provide an overview for the COPS protocol, and then describe how the system is set up. Finally, we will describe the working process of the COPS protocol in detail.

4.1. Overview

Our basic idea is to introduce an ADL to combine user queries and divide appropriate results to each user. The users, who want to retrieve files that they are interested in, form a group, where each member shares a group public/private key pair. As in [3], we assume that the ADL is able to estimate the number of files matching the query.

As illustrated previously, the main reason for introducing an ADL to combine queries and divide results is to save both computation and communication costs. We require the ADL to divide appropriate results to each user instead of simply returning everything to protect user privacy. For example, the results contain two files $\{|F_1|, |F_2|\}$, where $|F_1|$ is wanted by Alice, and

$|F_2|$ is wanted by Bob. If the ADL directly passes $\{|F_1|, |F_2|\}$ to Alice and Bob, Alice will know Bob wants $|F_2|$. The same case holds true for Bob.

Our protocol consists of five steps (see Fig. 4-(b)), where the second step (running *QueryMerge* algorithm) and the fourth step (running *ResultDivide* algorithm), executed by the ADL are newly added.

Step 1: Each user runs the QueryGen algorithm *to send a shuffled query to the ADL.* The query is an array of 0s and 1s as in [3]. To protect each user's query from the ADL, our protocol requires each user to shuffle his query with a *shuffle function*. Since the ADL does not know the secret seed of the shuffle function, the ADL cannot deduce the unshuffled query to know what each user is searching for.

Step 2: The ADL runs the QueryMerge algorithm *to send a combined query to the cloud.* The ADL executes OR operations on user queries entry by entry to obtain a merged query. Since each user's query is an array of 0s and 1s, the merged query is also an array of 0s and 1s. Then, the ADL encrypts each entry of the merged query with the Paillier cryptosystem *under its own public key*. The encrypted query, the number of keywords in the merged query, and the *estimated* number of files matching the merged query will be sent to the cloud.

Step 3: The cloud runs the PrivateSearch algorithm *to return two compact buffers to the ADL.* Since the merged query is encrypted under the ADL's public key, after processing the merged query on each file, each file is also encrypted under the ADL's public key. To protect file information from the ADL, we design a new mechanism as follows: The cloud uses a *pseudonym function* to replace the file name with the *file pseudonym*, and uses *obfuscate functions* to add some *obfuscate factors* to the file content. Without the secret seeds of these functions, the ADL cannot deduce the file names or the file contents. To enable the ADL to correctly distribute appropriate file pseudonyms and obfuscated file contents for each user, the cloud first shuffles the dictionary, and then constructs two buffers: *file pseudonym buffer* and *file content buffer*. The positions of the file pseudonym in the file pseudonym buffer are determined by a set of *map functions* and the positions of the file keywords in the shuffled dictionary. The positions of the obfuscated file content in the file content buffer are determined by another set of *map functions* and the file pseudonym. These map functions are publicly available.

Step 4: The ADL runs the ResultDivide algorithm *to divide appropriate results to each user.* The ADL first decrypts each entry of the two buffers sequently using its private key to obtain file pseudonyms and obfuscated file

contents. Given each user's shuffled query and a set of map functions, the ADL can find pseudonyms of files wanted by this user. For this type of map functions, the input is the location of the keyword in the shuffled query and output is the locations of file pseudonym buffer that store pseudonyms of files containing such a keyword. Then, given each file pseudonym and another set of map functions, the ADL can find obfuscated file contents wanted by this user. For this type of map functions, the input is the file pseudonym and output is the locations of file content buffer that stores obfuscated file content with such a pseudonym.

Step 5: Each user runs the FileRecover algorithm *to recover matched files.* After obtaining the obfuscated file contents from the ADL, the user only needs to remove the obfuscate factors to recover the file contents.

4.2. System setup

The system setting of the COPS protocol is as follows: There are t files $\{F_1, \ldots, F_t\}$ stored in the cloud. Each file F_i can be described by keyword set W_i, where $i \in \{1, \ldots, t\}$ denotes a file index. Each keyword w exists in a public dictionary Dic that consists of an array of d keywords $\langle w_1, \ldots, w_d \rangle$. Suppose n users constitute a group that shares a group public/private key pair (PK_G, SK_G). Let (PK_{ADL}, SK_{ADL}) and (PK_{cloud}, SK_{cloud}) denote the public/private key pairs of the ADL and the cloud, respectively. The summary of notations used in the COPS protocol is shown in Table 1.

Then, we describe the properties of functions used in the COPS protocol as follows, the definitions of which will be provided in Appendix A. In summary, there are three kinds of functions can be only executed by the cloud and the user, i.e., shuffle function, pseudonym function, and obfuscate function. Each function has its own unique secret seed. Therefore, there are four secret seeds shared between the users and the cloud. Since all users are in the same organization, the organization will manage and distribute the secrets. When a new user join in, the organization will distribute the secret seed to him off-line. This process can be analogous to an organization providing the password to a new user. The secret seed should be changed periodically. The rate of change is a system-defined parameter which out the scope of this paper.

- *Shuffle function* $F_1(s_1, \rho_w)$ is used to shuffle a query or a dictionary.
- *Pseudonym function* $F_2(s_2, i)$ is used to calculate the pseudonym for a file.

Notation	Description
Dic, Dic'	Original dictionary, Shuffled dictionary
B_1, B_2	File pseudonym buffer, File content buffer
w	Keyword in the dictionary
d	Number of keywords in the dictionary
n	Number of users
p	Failure probability
K_i	User i's keyword set
Q_i,Q'_i	User i's original query, User i's Shuffled query
Q	Merged query
f_i, f	Estimated number of files matching Q'_i, Q
k_i, k	Number of keywords in Q'_i, Q
F_i, F'_i	File name, File pseudonym
$\|F_i\|, \|F_i\|'$	File content, Obfuscated file content
W_i	F_i's keyword set
t	Number of files stored in the cloud
F_w	File set containing keyword w
$\digamma_1$, $\digamma_2$	Shuffle function, Pseudonym function
$\digamma_3$, $\digamma_4$	Obfuscate functions
$\{h_i\}_{1\leq i\leq log(k)}$	Map functions for B_1
$\{g_i\}_{1\leq i\leq log(f)}$	Map functions for B_2
ξ_w	Concatenation of pseudonyms containing keyword w
η_i	F_i's pseudonym
ρ_w	Keyword w's position in the dictionary or query
x_i, y_i	Obfuscate factors for F_i
$\{s_i\}_{1\leq i\leq 4}$	Secret seeds shared by the cloud and the users

Table 1: Summary of Notations

- *Obfuscate functions* $\digamma_3(s_3, \eta_i)$ and $\digamma_4(s_4, \eta_i)$ are used to calculate a obfuscate factor for the occurrence of user keywords in the file and a obfuscate factor for the file content, respectively.

- *Map functions* $\{h_j(\rho'_w)\}_{1\leq j\leq log(k)}$ and $\{g_j(\eta_i)\}_{1\leq j\leq log(f)}$ [1] are used to determine the mapping locations of the file pseudonym in the file pseudonym buffer and the mapping locations of the obfuscated file content in the

[1]We abbreviate $log(f/p)$ and $log(k/p)$ to $log(f)$ and $log(k)$, respectively.

File name	File keywords	File content
F_1	A, B	$\|F_1\|$
F_2	B, C	$\|F_2\|$
F_3	C, D	$\|F_3\|$
F_4	C	$\|F_4\|$
F_5	D	$\|F_5\|$

Table 2: Sample files in the cloud

Algorithm 1 Algorithms run by user i

QueryGen
for $j = 1$ **to** d **do**
  if keyword $w_j \in Dic$ is chosen by user i **then**
    $Q_i[j] = 1$
  else
    $Q_i[j] = 0$
shuffle Q_i to Q'_i with $\digamma_1$
Message from user i ***to the ADL:*** $MSG_{U_i 2ADL} = \{Q'_i\}$

FileRecover
for Each pseudonym-content pair $(\eta_*, |F_*|')$ obtained from the ADL **do**
  set $x_* = \digamma_3(s_3, \eta_*)$ to be a obfuscate factor for keyword occurrence in F_*
  set $y_* = \digamma_4(s_4, \eta_*)$ to be a obfuscate factor for F_*'s content
  execute Eq. 6 to recover file content $|F_*|$

file content buffer, respectively.

4.3. Protocol Description

The COPS protocol shown in Fig. 4-(B) consists of five steps. The functions used in the COPS protocol are defined in Appendix A, and the correctness of the COPS protocol is proven in Appendix B. We use a simple example to illustrate its working process. The example assumes that the original dictionary $Dic = \langle A, B, C, D \rangle$ and that the files stored in the cloud are as in Table 2. Two users, Alice and Bob, wish to retrieve files with keywords "A, B" and "A, C", respectively.

Step 1: Each user runs the QueryGen *algorithm in Alg. 1 to send a shuffled query to the ADL.* The query is generated as follows: if a keyword

$w \in Dic$ is chosen by user i, then the corresponding entry in Q_i is set to 1, otherwise 0; then, Q_i is shuffled with F_1. Actually, this process is equivalent to first shuffling the dictionary to Dic' with F_1, and then checking whether $w_j \in Dic'$ is chosen to determine the value (0 or 1) of $Q_i[j]$.

For example, Alice's original query $Q_{Alice} = \langle 1, 1, 0, 0 \rangle$ and Bob's original query $Q_{Bob} = \langle 1, 0, 1, 0 \rangle$. If the shuffle function $F_1(s_1, 1) = 4$, $F_1(s_1, 2) = 1$, $F_1(s_1, 3) = 2$, $F_1(s_1, 4) = 3$, then Alice's shuffled query $Q'_{Alice} = \langle 1, 0, 0, 1 \rangle$, and Bob's shuffled query $Q'_{Bob} = \langle 0, 1, 0, 1 \rangle$.

Step 2: The ADL runs the QueryMerge *algorithm in Alg. 2 to send an encrypted and combined query to the cloud.* The query is generated as follows: the ADL executes OR operations on user queries entry by entry to obtain a combined query, and then encrypts each entry of the combined query with its public key.

For example, the combined query $Q = Q'_{Alice} \vee Q'_{Bob} = \langle 1 \vee 0, 0 \vee 1, 1 \vee 0, 1 \vee 1 \rangle = \langle 1, 1, 0, 1 \rangle$. The encrypted query is in the form of $Q = \langle E(PK_{ADL}, 1), E(PK_{ADL}, 1), E(PK_{ADL}, 0), E(PK_{ADL}, 1) \rangle$. The ADL also needs to estimate f, the number of files matching Q, and determine k, the number of keywords in Q, to determine buffer size and mapping times. Here, k is actually the number of 1s in Q.

Step 3: The cloud runs the PrivateSearch *algorithm in Alg. 3 to return two buffers to the ADL.* Firstly, the cloud shuffles the dictionary to Dic', so that the position of each keyword in the shuffled dictionary matches its position in the shuffled query. Then, it constructs two buffers B_1 of size $2 \cdot k \cdot log(k)$ and B_2 of size $2 \cdot f \cdot log(f)$, where each entry is initialized with $(E(PK_{ADL}, 0), E(PK_{ADL}, 0))$.

The file pseudonym buffer B_1 is generated as follows: For each keyword $w \in Dic'$, the cloud finds out all files containing w, denoted as F_w, calculates the pseudonym for each file in F_w, and concatenates pseudonyms of all files in F_w (the concatenation is denoted as ξ_w). Let ρ_w denote the position of keyword w in Dic'. The cloud sets c_w with $Q[\rho_w]$, and powers η_w to c_w to obtain e_w. Each *keyword-pseudonym pair* (c_w, e_w) is calculated with Eq. 1:

$$c_w = Q[\rho_w]; e_w = c_w^{\xi_w} \tag{1}$$

where c_w denotes whether w is chosen by at least one user, and e_w denotes c_w raised to the power of ξ_w. Thus, if w is not chosen by any user, then $Q[\rho_w] = 0$, and both c_w and e_w will be encrypted to 0. Otherwise, c_w is an

Algorithm 2 Algorithms run by ADL

QueryMerge
for $j = 1$ **to** d **do**
  $Q[j] = \vee_{i=1}^{n} Q'_i[j]$
  encrypt $Q[j]$ with PK_{ADL}
set f to the estimated number of files matching Q
set k to the number of keywords in Q
Message from the ADL to the cloud: $MSG_{ADL2Cloud} = \{Q, f, k\}$

ResultDivide
Decrypt B_1, B_2 with SK_{ADL}
for $i = 1$ **to** n **do**
  for $j = 1$ **to** d **do**
    if $Q'_i[j] = 1$ **then**
      for $l = 1$ **to** $log(k)$ **do**
        locate $B_1[h_l(j)]$ to obtain plaintext pair (c'_{w_j}, e'_{w_j})
        calculate Eq. 4 to get concatenation of file pseudonyms ξ_{w_j}
      for each $\eta_* = \digamma_2(s_2, *)$ in ξ_{w_j} **do**
        for $l = 1$ **to** $log(k)$ **do**
          locate $B_2[g_l(\eta_*)]$ to obtain plaintext pair (c'_*, e'_*)
          calculate Eq. 5 to get obfuscated file content $|F_*|'$
Message from the ADL to user i***:*** $MSG_{ADL2U_i} = \{(\eta_*, |F_*|')\}$

Keyword	Encrypted occurrence	Encrypted pseudonym
A	$E(1)$	$E(1)^{\xi_A} = E(1 \cdot F'_1)$
B	$E(1)$	$E(1)^{\xi_B} = E(1 \cdot F'_1 \| \| F'_2)$
C	$E(1)$	$E(1)^{\xi_C} = E(1 \cdot F'_2 \| \| F'_3 \| \| F'_4)$
D	$E(0)$	$E(0)^{\xi_D} = E(0)$

Table 3: Keyword-pseudonym pair

encryption of 1, and e_w is an encryption of some value larger than 0. The example pairs are generated as shown in Table. 3.

Finally, in order to return file pseudonyms matching the query, the cloud *multiplies* each pair (c_w, e_w) to the file pseudonym buffer B_1 with Eq. 2:

$$B_1[h_l(\rho_w)] = B_1[h_l(\rho_w)] \cdot (c_w, e_w), (1 \leq l \leq log(k)) \tag{2}$$

Each entry of B_1 is initialized with $(E(PK_{ADL}, 0), E(PK_{ADL}, 0))$. The cloud

Algorithm 3 Algorithms run by cloud

PrivateSearch

shuffle the dictionary to Dic' with $\digamma_1$

initialized each entry of B_1 of size $2 \cdot k \cdot log(k)$ and B_2 of size $2 \cdot f \cdot log(f)$ with $(E(PK_{ADL}, 0), E(PK_{ADL}, 0))$

for $j = 1$ **to** d **do**

  for each keyword $w \in Dic'$ **do**

    set ρ_w to be the position of w in Dic'

    set F_w to be the file set containing keyword w

    for each file $F_* \in F_w$ **do**

      set $\eta_* = \digamma_2(s_2, *)$ to be F_*'s pseudonym

    set ξ_w to be the concatenation of file pseudonyms containing w

    execute Eq. 1 to obtain (c_w, e_w)

    for $l = 1$ **to** $log(k)$ **do**

      execute Eq. 2 to store (c_w, e_w) in the $h_l(\rho_w)$-*th* entry of B_1

for each file F_i **do**

  set $\eta_i = \digamma_2(s_2, i)$ to be the pseudonym of F_i

  set $x_i = \digamma_3(s_3, \eta_i)$ to be a obfuscate factor for keyword occurrence in F_i

  set $y_i = \digamma_4(s_4, \eta_i)$ to be a obfuscate factor for F_i's content

  execute Eq. 3 to obtain (c_i, e_i)

  for $l = 1$ **to** $log(f)$ **do**

    execute Eq. 3 to store (c_i, e_i) in the $g_l(\eta_i)$-*th* entry of B_2

Message from the cloud to the ADL: $MSG_{Cloud2ADL} = \{B_1, B_2\}$

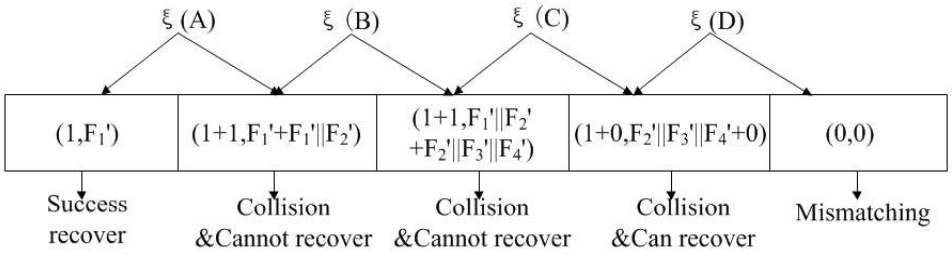

Figure 5: Map concatenation of file pseudonyms two times into B_1 of five entries. a is used to denote $E(a)$; thus, $E(a) \cdot E(b)$ and $E(a)^b$ are replaced with $a + b$ and $a \cdot b$, respectively.

will multiply (c_w, e_w) to the $h_l(\rho_w)$-th entry of B_1, respectively. The example file pseudonym buffer is constructed as in Fig. 5.

The file content buffer B_2 is generated as follows: For each file F_i, the

File name	Encrypted occurrence	Encrypted content
F_1	$(E(1) \cdot E(1))^{x_1}$	$E(2 \cdot x_1)^{\lvert F_1 \rvert'}$
	$= E(2 \cdot x_1)$	$= E(2 \cdot x_1 \cdot y_1 \cdot \lvert F_1 \rvert)$
F_2	$(E(1) \cdot E(1))^{x_2}$	$E(2 \cdot x_2)^{\lvert F_2 \rvert'}$
	$= E(2 \cdot x_2)$	$= E(2 \cdot x_2 \cdot y_2 \cdot \lvert F_1 \rvert)$
F_3	$(E(1) \cdot E(0))^{x_3}$	$E(1 \cdot x_3)^{\lvert F_3 \rvert'}$
	$= E(1 \cdot x_3)$	$= E(1 \cdot x_3 \cdot y_3 \cdot \lvert F_3 \rvert)$
F_4	$E(1)^{x_4}$	$E(1 \cdot x_4)^{\lvert F_4 \rvert'}$
	$= E(1 \cdot x_4)$	$= E(1 \cdot x_4 \cdot y_4 \cdot \lvert F_4 \rvert)$
F_5	$E(0)^{x_5} = E(0)$	$E(0)^{\lvert F_5 \rvert'} = E(0)$

Table 4: Occurrence-content pair

cloud calculates file pseudonym η_i, and obfuscates its content by setting $|F_i|' = |F_i| \cdot \digamma_4(s_4, \eta_i) = |F_i| \cdot y_i$. Let ρ_w denote the position of keyword w in Dic'. The cloud then multiplies the entries in Q that correspond to file keywords to obtain the encrypted occurrence of user keywords in the file. The cloud then obtains c_i by powering the encrypted occurrence by x_i, where $x_i = \digamma_3(s_3, \eta_i)$. Next, the cloud raises the obfuscated file content $|F_i|'$ to c_i to obtain the encrypted obfuscated file content e_i. Each *occurrence-content pair* (c_i, e_i) is calculated with Eq. 3:

$$c_i = \prod_{w \in W_i} E(PK_{ADL}, Q[\rho_w])^{x_i} = E(PK_{ADL}, \sum_{w \in W_i} Q[\rho_w])^{x_i}; e_i = c_i^{y_i \cdot |F_i|}$$

where c_i denotes whether W_i contains at least one keyword chosen by users, and e_i is the encryption of c_i powered by the obfuscated file content $|F_i|$. Thus, if W_i contains no keyword chosen by any user, then $\sum_{w \in W_i} Q[\rho_w] = 0$, and both c_i and e_i are an encryption of 0. Otherwise, both of them are an encryption of some values larger than 0. The example pairs are generated as shown in Table. 4.

Finally, in order to return obfuscated file contents matching the query, the cloud *multiplies* each pair (c_i, e_i) to file content buffer B_2 with Eq. 3:

$$B_2[g_l(\eta_i)] = B_2[g_l(\eta_i)] \cdot (c_i, e_i), (1 \leq l \leq log(f)) \tag{3}$$

Each entry of B_2 is initialized with $(E(PK_{ADL}, 0), E(PK_{ADL}, 0))$. The cloud will multiply (c_i, e_i) to the $g_l(\eta_i)$-th entry of B_2, respectively. The example file content buffer is constructed as in Fig. 6.

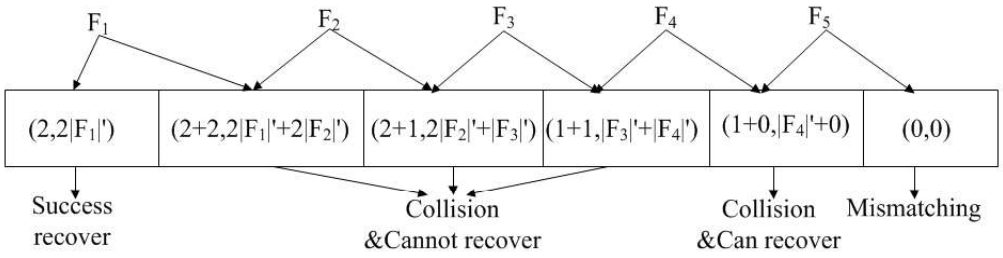

Figure 6: Map obfuscated file content two times into B_2 of six entries. a is used to denote $E(a)$; thus, $E(a) \cdot E(b)$ and $E(a)^b$ are replaced with $a + b$ and $a \cdot b$, respectively.

Step 4: The ADL runs the ResultDivide *algorithm in Alg. 2 to distribute obfuscated file contents to appropriate users.* To ensure that each user only obtains the files related to his query, and no files associated with other users, the ADL has to first correctly locate file pseudonyms matching each user's query in B_1, and then locate obfuscated file contents associated with these file pseudonyms in B_2. The file pseudonym is recovered as follows: The ADL first checks the position of 1 in Q'_i. Suppose that the $w_j \in Dic'$ is chosen by user i, i.e., $Q'_i[j] = 1$. The ADL locates entries $h_1(j), \ldots, h_{log(k)}(j)$ in B_1 to obtain plaintext keyword-pseudonym pair (c'_{w_j}, e'_{w_j}). If $c'_{w_j} \neq 0$, ξ_{w_j} is calculated with Eq. 4:

$$\xi_{w_j} = e'_{w_j}/c'_{w_j} \tag{4}$$

Then, the ADL decomposes ξ_{w_j} to obtain separate file pseudonyms [2]. Suppose that η_* is a file pseudonym containing keyword w_j. The ADL locates entries $g_1(j), \ldots, g_{log(f)}(j)$ in B_2 to obtain plaintext occurrence-content pair (c'_*, e'_*). If $c'_* \neq 0$, the obfuscated file content is calculated with Eq. 5:

$$|F'_*| = e'_*/c'_* \tag{5}$$

In the example, the ADL first decrypts each entry of B_1 and B_2. Suppose that all file pseudonyms and obfuscated file contents matching Q can be recovered. The ADL knows file pseudonyms matching Q_{Alice} will appear in positions $B_1[h_1(1)], \ldots, B_1[h_{log(k)}(1)]$ and $B_1[h_1(4)], \ldots, B_1[h_{log(k)}(4)]$,

[2]Each file pseudonym can be set to $log(t)$ bits equally, and the ADL can divide every $log(t)$ bits in the concatenation to obtain the separate pseudonyms.

and matching Q_{Bob} will appear in positions $B_1[h_1(2)], \ldots, B_1[h_{log(k)}(2)]$, and $B_1[h_1(4)], \ldots, B_1[h_{log(k)}(4)]$. It gets the concatenation of file pseudonyms $\eta_1||\eta_2$ and η_1 for Alice, and $\eta_2||\eta_3||\eta_4$ and η_1 for Bob, and decomposes $\eta_1||\eta_2$ and $\eta_2||\eta_3||\eta_4$ to η_1, η_2 and η_2, η_3, η_4. Then, it determines that the file with pseudonym η_i will be stored in $B_2[g_1(\eta_i)], \ldots, B_2[g_{log(f)}(\eta_i)]$, for $1 \leq i \leq 4$. Finally, it returns $\{(\eta_i, |F_i|')\}_{i=1,2}$ to Alice and $\{(\eta_i, |F_i|')\}_{i=1,2,3,4}$ to Bob, respectively.

Step 5: The user runs the FileRecover *algorithm in Alg. 1 to obtain all files matching his query.* Let $(\eta_*, |F_*|')$ denote a pseudonym-content pair obtained from the ADL.

$$|F_*| = |F_*|' / \digamma_4(s_4, \eta_*) \tag{6}$$

Note that: (1) The user cannot obtain the file index/name from the file pseudonym, since the pseudonym function is a one-way hash function. Instead, the file index/name can be appended to the file content before the cloud obfuscates the file content. After removing the obfuscate factors, the user can extract the file name. (2) Only obfuscate function $\digamma_4$ is used by the user to recover files. The obfuscate function $\digamma_3$ is used to obfuscate the number of common keywords between each file and the query.

5. Analysis

In this section, we will provide the security analysis for the COPS protocol, analyze its performance regarding computation and communication costs, and make performance comparisons with existing works.

5.1. Security analysis

The security and privacy requirements in Section 3 are satisfied if the following three cases are true:

Case 1. The cloud knows neither keywords nor file contents queried by any user.

The messages between the ADL and the cloud are Q, f, k, and two buffers, where k is the only information which we leak more than the work by [3, 5]. Note that the query Q and two buffers are encrypted under the ADL's public key. Therefore, even if the cloud collude with a small number of malicious users, it cannot know what the honest users are searching for nor which files are returned to the honest users.

Since f is an estimated value for the number of files matching the combined query, the cloud cannot know users' interests from f. Given the additional information k, the cloud only knows the number of total keywords specified by the users. It cannot know what these k keywords exactly are. Furthermore, the cloud knows a shuffle function, a pseudonym function, two obfuscate functions, and a set of map functions. Given the shuffle function, the cloud can shuffle the dictionary. Since the shuffle function has nothing to do with user interests, it cannot reveal any valuable information. Given the pseudonym function, obfuscate functions, and map functions, the cloud can store file pseudonyms and obfuscated file contents into two buffers, respectively. Since the cloud runs these functions on every file equally to generate two buffers encrypted under the ADL's public key, it cannot know which files are actually of interest to the users. Therefore, Case 1 is true.

Case 2. The ADL knows neither keywords nor file contents queried by any user.

The messages from each user to the ADL is a shuffled query Q'_i. The secret seed of the shuffle function is only known by the users and the cloud. Therefore, the ADL itself cannot deduce the un-shuffled query. Given a shuffled query Q'_i, the ADL knows k_i, the number of keywords in the shuffled query. The probability for the ADL to guess one keyword is $1/d$, and all k_i keywords is $(1/d)^{k_i}$. Given the combined query Q, the ADL knows k, the number of keywords in the merged query. The probability for the ADL to guess one keyword is $1/d$, and all k keywords is $(1/d)^k$. When the dictionary is large enough, it is hard for the ADL to guess each user's interests.

The messages from the cloud to the ADL are B_1 and B_2. The buffers are encrypted with the ADL's public key, and thus the ADL can decrypt each entry in two buffers to know file pseudonyms and obfuscated file contents. The pseudonyms cannot reveal any valuable information. The secret seeds of the obfuscate functions are only known by the users and the cloud, and thus the ADL cannot remove the obfuscate factors to recover the file contents. Furthermore, the ADL knows a set of map functions $\{h_l\}_{1\leq l\leq log(k)}$ and $\{g_l\}_{1\leq l\leq log(f)}$, which enable the ADL to know where file pseudonyms and file contents will be stored in B_1 and B_2, respectively. These functions take the shuffled keyword position or the file pseudonym as input, which cannot leak any valuable information to the ADL. Therefore, Case 2 is true.

Case 3. The user knows neither keywords nor file contents queried by other users.

Since the ADL can correctly divide results to each user, each user can

only obtain its own results. Even if a small number of malicious users work cooperatively or collude with the cloud, they cannot know other honest users' interests. Therefore, Case 3 is true.

5.2. Performance analysis

Computational cost. The running time of the *QueryGen* algorithm is negligible, since each user only needs to shuffle his query. The running time of the *FileRecover* algorithm is negligible, since each user only needs to execute $O(f_i)$ divisions.

The running time of the *QueryMerge* algorithm is mainly on encrypting the combined query Q with Paillier encryption, which is $O(d)$. The cost is exactly the same as a user generating a query in [3, 5]. More precisely, the *QueryMerge* algorithm requires d exponentiations and k multiplications. Since the cost on exponentiations is 1000 times that of the cost on multiplications, we only consider the cost on exponential operations. The running time of the *ResultDivide* algorithm is to decrypt two buffers, which is $O(f \cdot log(f/p) + k \cdot log(k/p))$.

The running time of the *PrivateSearch* algorithm mainly is on the generation of $t + d$ pairs, which is $O(t + d)$. The running time of the generations of B_1 and B_2 requires the execution of $t \cdot log(f) + d \cdot log(k)$ multiplications, which is much less than the time of the generations of pairs.

Communication cost. We first consider the communication costs at the ADL. The communication costs can be classified into two kinds: (1) The costs between all users and the ADL; (2) The costs between the ADL and the cloud. In terms of the costs between all users and the ADL, the ADL will first receive n queries from all users, which is $O(|Dic|)$. Then, it will distribute results to each user, which is $O(\sum_{i=1}^{n} f_i)$. In terms of the costs between the ADL and the cloud, the ADL will first send a combined query to the cloud, which is $O(|Dic|)$. Then, it will receive two buffers from the cloud, which is $O(f \cdot log(f/p) + k \cdot log(k/p))$.

During the interaction with all users, all messages are transferred through local area networks, the speed of which is much faster than accessing Internet and the communication cost is lower. In practice, even without any cryptographic approach, the messages from each user to the cloud are forward by organization gateway. The communication cost at the ADL is almost the same as the cost at the gateway.

Then, We will consider the communication cost at the cloud. The transfer-in cost of the cloud is to receive a combined query Q from the ADL, which is

Protocl	Communication	Computation
Ostrovsky	$O(\sum_{i=1}^{n} f_i \cdot log(f_i))$	$O(n \cdot t)$
Bethencourt	$O(\sum_{i=1}^{n} f_i)$	$O(n \cdot t)$
COPS	$O(f \cdot log(f) + k \cdot log(k)))$	$O(t + d)$

Table 5: Performance comparison

$O(|Dic|)$, where $|Dic|$ is the size of the dictionary. The transfer-out cost of the cloud is to return two buffers to the ADL, which is $O(f \cdot log(f/p) + k \cdot log(k/p))$.

The buffer size is determined as follows: In the Ostrovsky protocol, it is proven that if m files are stored uniformly at random γ times into a buffer of size $2\gamma m$, the failure probability to recover m files will be smaller than $m/2^{\gamma}$. Therefore, if we want to retrieve file pseudonyms for k keywords with a failure probability that is smaller than p, we should construct a pseudonym buffer with size $2k \cdot log(k/p)$, and map each pseudonym into $log(k/p)$ buffer entries. If we want to retrieve the contents of f files with a failure probability that is smaller than p, we should construct a content buffer with size $2f \cdot log(f/p)$, and map each into $log(f/p)$ buffer entries.

We compare the computation cost and communication cost at the cloud among our protocol, the Ostrovsky protocol, and the work by Bethencourt et al. [5], as shown in Table 5. Suppose that there are n users querying the cloud with failure probability p, t files stored in the cloud, f_i files matching the *i-th* user's query, and f files matching the combined query.

6. Evaluation

In this section, we first conduct simulations to compare the computation/communication costs incurring at the cloud between the Ostrovsky protocol and the COPS protocol. Our simulation results are partially based on the results in [5]. Then, based on the simulation results, we deploy our program in Amazon Elastic Compute Cloud (EC2) to test the transfer-in and transfer-out time at the cloud when executing private searches. The parameters used in the experiments are summarized in Table 6.

6.1. Computation cost

As described in Section 5-(B), the computation cost in the COPS protocol is mainly determined by the number of exponentiations, which is affected by

Notation	Description	Value
$\|F\|$	File content	$1KB$
$\|w\|$	Keyword content	$1KB$
n	Number of users	1-100
d	Number of keywords in the dictionary	100-1,000
k	Number of keywords in each query	1-5
l	Number of keywords in each file	1-5
t	Number of files stored in the cloud	10^3
p	Failure probability	0.1

Table 6: Parameters

two parameters: the number of files stored in the cloud t and the number of keywords in the dictionary d.

Given $t = 1,000$ is fixed, we will first compare the computation costs in the Ostrovsky protocol and the COPS protocol under different dictionary settings ($d = 100$ and $d = 1,000$). In each setting, there are 1 to 100 users randomly choosing 1-5 keywords from the dictionary to execute queries. The comparisons of computation cost at the cloud are shown in Fig. 7. While the number of users ranges from 1 to 100, the computation cost approximately ranges from $14.8881s$ to $1,488.8s$ in the Ostrovsky protocol and from $14.8988s$ to $14.9495s$ in the COPS protocol under a dictionary of 100 keywords; under a dictionary of 1,000 keywords, it ranges approximately from $14.8494s$ to $1,484.7s$ in the Ostrovsky protocol and from $14.9559s$ to $15.0718s$ in the COPS protocol. In both dictionary settings, the most expensive operation is executing searches, which approximately ranges from $14.7190s$ to $1,471.9s$ in the Ostrovsky protocol, and is approximately $14.7190s$ in the COPS protocol.

Therefore, our protocol consumes less computation cost than the Ostrovsky protocol as the number of users increases. Specifically, our protocol performs better than the Ostrovsky protocol, even when there are only few users executing searches. For example, the computation cost is saved by 80% when $n = 5$ and by 96% when $n = 25$.

6.2. Communication cost

As described in Section 5-(B), given p is fixed, the communication cost in the COPS protocol mainly depends on the number of keywords k in Q and the number of files f matching Q. From Eq. 7, we know that e_k, the expected value of k, is different under different dictionary settings:

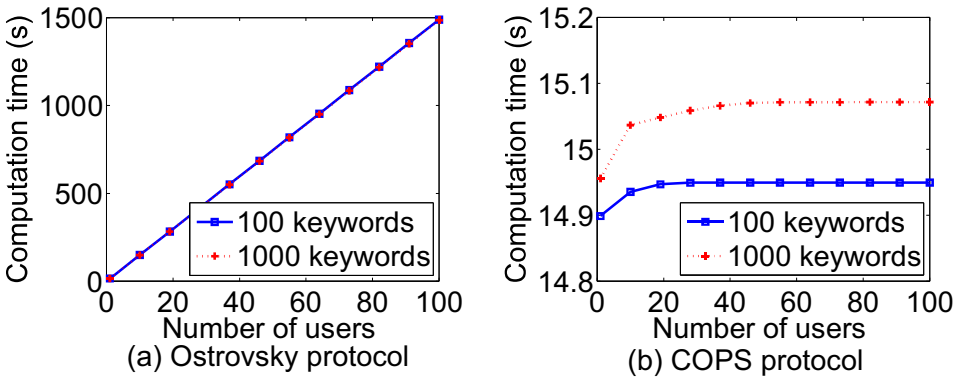

Figure 7: Comparison of computation cost at the cloud.

$$e_k = d \cdot (1 - (1 - \frac{\bar{k}}{d})^n) \tag{7}$$

where $\bar{k} = \sum_{i=1}^{n} k_i/n$ is the average number of keywords chosen by each user.

Thus, we will make comparisons under two dictionary settings ($d = 100$ and $d = 1,000$). Furthermore, the number of files matching queries f is different when users have different common interests. Therefore, in each dictionary setting, we will making comparisons under different common interests as shown in Fig. 8. Here, a common interest can be calculated as follows: Suppose that there are n users, where each user chooses k_i keywords, and the number of keywords after combination is k. Common interests can be calculated with $(\sum_{i=1}^{n} k_i - k)/\sum_{i=1}^{n} k_i$.

From Figs. 9 and 10, we know that our protocol performs better as the common interests between users increase. When the dictionary contains 100 keywords, in the worst cases, where user common interest is less than 70%, the buffer size ranges from $2,478KB$ to $239,840KB$ in the Ostrovsky protocol, and from $2,532KB$ to $36,286KB$ in the COPS protocol; in the best cases, when user common interest is more than 80%, the buffer size ranges from $2,525KB$ to $255,870KB$ in the Ostrovsky protocol, and from $2,577KB$ to $14,846KB$ in the COPS protocol. When the dictionary contains $1,000$ keywords, in the worst cases, where user common interest is less than 40%, the buffer size ranges from $218KB$ to $21,337KB$ in the Ostrovsky protocol, and from $270KB$ to $1,901KB$ in the COPS protocol; in the best cases, when user common interest is more than 80%, the buffer size ranges from $212KB$

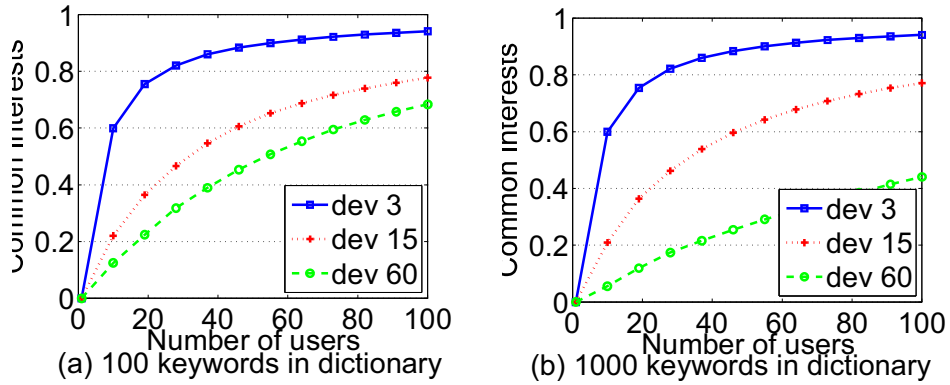

Figure 8: Keyword distribution. Each user chooses 1-5 keywords from the dictionary according to normal distribution with different standard deviations, denoted as $dev = 3$, $dev = 15$, and $dev = 60$, respectively.

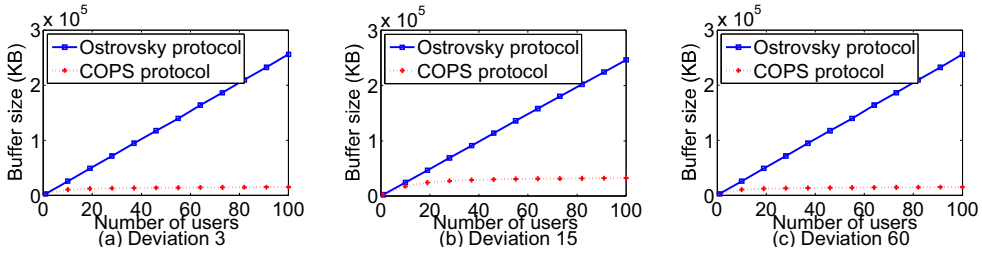

Figure 9: Comparison of communication cost at the cloud under different common interests when the dictionary contains 100 keywords.

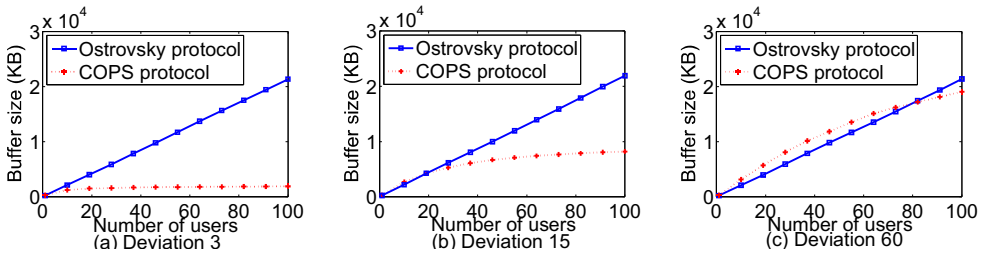

Figure 10: Comparison of communication cost at the cloud under different common interests when the dictionary contains 1,000 keywords.

to $21,369KB$ in the Ostrovsky protocol, and from $265KB$ to $1,904.7KB$. Therefore, *we combine user queries only when they have a certain percentage of common interests.*

Specifically, our protocol performs better than the Ostrovsky protocol,

even when there are only few users executing searches. For example, when the dictionary contains 100 keywords, our protocol can reduce the bandwidth by 4% in the worst cases, and by 37% in the best cases, while only 5 users; our protocol can reduce the bandwidth by 50% in the worst cases, and by 80% in the best cases, while only 25 users.

6.3. Transferring time in real cloud

For the sake of verifying the feasibility of the proposed protocol, we deploy our program in the real cloud environment, Amazon EC2, to test the transfer-in (receiving query) and transfer-out (sending buffer) time at the cloud. The local machine is with Intel Core 2 Duo E8400 3.0 GHz CPU and 8 GB Linux RAM. We subscribe EC2 amzn-ami-2011.02.1.i386-ebs (ami-8c1fece5) AMI and a small type instance with the following specs: 32-bit platform, a single virtual core equivalent to 1 compute unit CPU, and 1.7 GB RAM. The average bandwidth from EC2 to the local machine is 33.43 MB/s, and from the local machine to EC2 is 42.98 MB/s.

First, we will test the transfer-in time in the real cloud under different dictionary settings (100 keywords and 1000 keywords). Since each keyword is $1KB$, the query size is $100KB$ and $1MB$ under a dictionary of 100 keywords and $1,000$ keywords, respectively. The local machine will send a combined query to the EC2 and count the transferring time. We conduct experiment for 1000 times and take the average transferring time to eradicate any discrepancies. In Fig. 11, the transfer-in time is approximately $32.8891s$ when the dictionary size is $100KB$ and is approximately $179.2363s$ when the dictionary size is $1MB$.

Then, based on the results in Figs. 9 and 10, we test the transfer-out time at the cloud in the COPS protocol under different common interests and dictionary settings. In the experiments, each user chooses 1-5 keywords from the dictionary according to normal distribution with different standard deviations (deviation 3, deviation 15, and deviation 60). In Fig. 12, when the dictionary contains 100 keywords, the transfer-out time is approximately from $761s$ to $3,651s$ in the best cases, from $636s$ to $6,928s$ in the medium cases, and from $713s$ to $8,066s$ in the worst cases; when the dictionary contains 1,000 keywords, the transfer-out time is approximately from $107s$ to $509s$ in the best cases, from $138s$ to $1,956s$ in the medium cases, and from $185s$ to $4,420s$ in the worst cases.

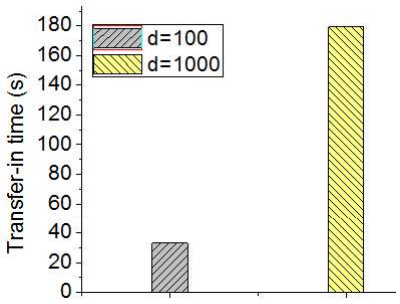

Figure 11: Transfer-in time at the cloud.

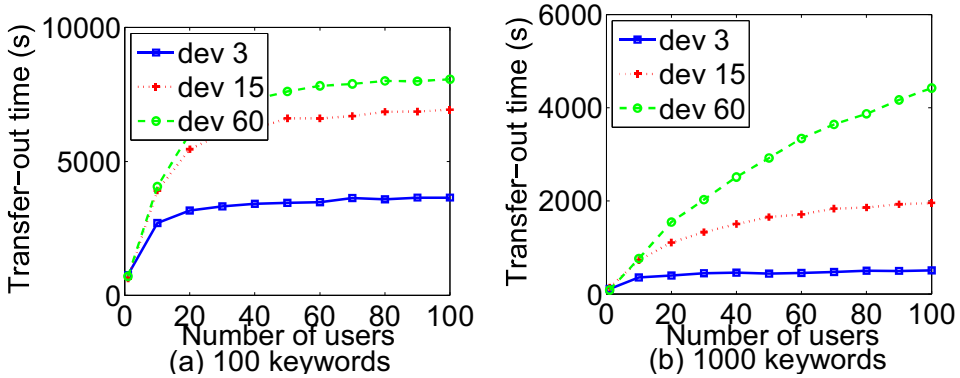

Figure 12: Transfer-out time at the cloud.

7. Discussion

In this section, we will provide additional details about our further work.

Removing the ADL. The ADL in the COPS protocol can be removed by letting users perform the function of the "ADL". The users are first arranged into groups. For each group, a member from a different group will act as the ADL (see Fig. 13). The ADL can be chosen in the following way: For each group, an ID ring is constructed as in Fig. 14, where each node has an ID. Each user is denoted by a set of nodes in the ring, where the number of nodes is to be determined by the capability of a user's client, a user's trusted rank, and so on. Suppose the number of nodes in the ring is n. To elect an ADL, we can randomly generate a number r, and let the user who is denoted by the node with an ID of (r mod n) in the ring to be the ADL. The user is denoted by more nodes, making the probability of him being chosen as the ADL for other groups higher

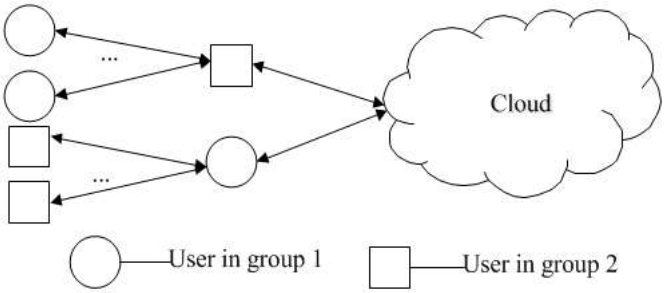

Figure 13: No ADL COPS.

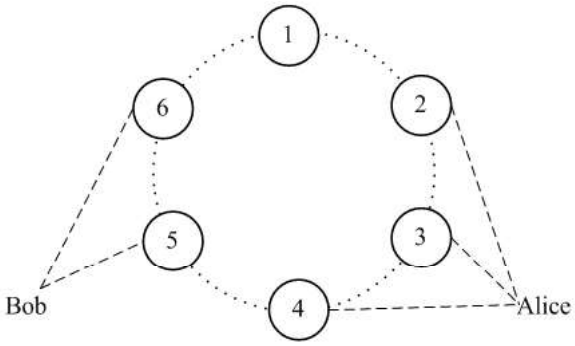

Figure 14: ID ring.

Grouping. In the previous no-ADL protocol, there are two interesting problems: Classifying users into groups and determining how many groups are appropriate. The basic idea is to classify the users with the most common interests into one group. Note that the computation cost at the cloud will grow linearly with the number of groups since the cloud needs to execute private searching once for each group. The transfer-in communication cost at the cloud also grows linearly with the number of groups since each group needs to send a query of size $Q(|Dic|)$ to the cloud. However, the transfer-out communication cost on the cloud in the case of g groups may be less than that in the case of one group, if the following equation is satisfied: $2k \cdot log(k/p) + 2f \cdot log(f/p) > \sum_{i=1}^{g}(2k_i \cdot log(k_i/p) + 2f_i \cdot log(f_i/p))$.

Common noises. Given each user's shuffled query, the ADL can know the number of keywords k_i in each query. After the combination of queries, the ADL further knows the number of keywords in the merged query k as well as common keywords between the users $\sum_{i=1}^{n} k_i - k$. After the decryption

of two buffers, the ADL can know the common files between users, even if it cannot know the exact file name or file content. To solve this problem, all users can add some *common noises* to their queries. The drawback of such a method is that the cloud will return a set of unrelated files matching the common noises to increase the communication cost.

8. Conclusion

In this paper, we propose the COPS protocol for the cost-effective cloud environment to allow multiple users to collaboratively execute private searching on the cloud. Our simulation results show that COPS reduces the computational and bandwidth costs, and hence reduce the bottleneck between the ADL and the cloud. Our future work will avoid the potential bottleneck between the users and the ADL. This bottleneck may occur when too many users have unique queries. One approach is to deploy multiple ADLs, and then map users with similar queries to the same ADL.

Acknowledgements

This research was supported in part by NSF grants ECCS 1128209, CNS 1065444, CCF 1028167, CNS 0948184, CCF 0830289; and National NSF of China under Grant No. 61073037, Hunan Provincial Science and Technology Program under Grant No. 2010GK2003, and National 973 Basic Research Program of China under Grant No. 2011CB302800.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, "Above the clouds: A berkeley view of cloud computing," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28*, 2009.

[2] ——, "A view of cloud computing," *Communications of the ACM*, 2010.

[3] R. Ostrovsky and W. Skeith III, "Private searching on streaming data," in *Proceedings of CRYPTO*, 2005.

[4] J. Bethencourt, D. Song, and B. Waters, "New Constructions and Practical Applications for Private Stream Searching (Extended Abstract)," in *Proceedings of IEEE Symposium on Security and Privacy*, 2006.

[5] ——, "New techniques for private stream searching," *ACM Transactions on Information and System Security (TISSEC)*, 2009.

[6] G. Danezis and C. Diaz, "Improving the decoding efficiency of private search," in *IACR Eprint archive number 024*, 2006.

[7] ——, "Space-efficient private search with applications to rateless codes," in *Proceedings of FC*, 2007.

[8] B. Adida and D. Wikstrom, "How to shuffle in public," in *Proceedings of TCC*, 2007.

[9] H. Zhu and F. Bao, "Private searching on MapReduce," in *Proceedings of TrustBus*, 2010.

[10] I. Damgard and M. Jurik, "A Generalisation, a Simpli. cation and Some Applications of Paillier's Probabilistic Public-Key System," in *Proceedings of PKC*, 2001.

[11] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *Proc. of ACM CCS*, 2006.

[12] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," *Journal of the ACM*, 1995.

[13] E. Kushilevitz and R. Ostrovsky, "Replication is not needed: Single database, computationally-private information retrieval," in *Proc. of IEEE Annual Symposium on Foundations of Computer Science*, 1997.

[14] F. Olumofin and I. Goldberg, "Privacy-preserving queries over relational databases," in *Proceedings of PETS*, 2010.

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

[16] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, "Public key encryption with keyword search," in *Proceedings of EUROCRYPT*, 2004.

[17] E.-J. Goh, "Secure indexes," Cryptology ePrint Archive, Report 2003/216, Tech. Rep., 2003.

[18] Y. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in *Proceedings of ACNS*, 2005.

[19] C. Wang, N. Cao, J. Li, K. Ren, and W. Lou, "Secure ranked keyword search over encrypted cloud data," in *Proc. of IEEE ICDCS*, 2010.

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

[21] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, 2008.

[22] ——, "MapReduce: a flexible data processing tool," *Communications of the ACM*, 2010.

[23] A. Boldyreva, N. Chenette, Y. Lee, and A. Oneill, "Order-preserving symmetric encryption," *Advances in Cryptology-EUROCRYPT*, 2009.

[24] W. Wong, D. Cheung, B. Kao, and N. Mamoulis, "Secure knn computation on encrypted databases," in *Proc. of ACM SIGMOD*, 2009.

[25] M. Green and G. Ateniese, "Identity-based proxy re-encryption," in *Applied Cryptography and Network Security*, 2007.

[26] M. Blaze, G. Bleumer, and M. Strauss, "Divertible protocols and atomic proxy cryptography," *Advances in CryptologyEUROCRYPT*, 1998.

[27] D. Knuth, *Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley Professional, 1997.

[28] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, 1970.

Appendix A. Function definitions

The functions used in the COPS protocol are defined as follows, where the secret seeds $\{s_i\}_{1\leq i\leq 4}$ are shared by the cloud and the users:

Shuffle function. *Shuffle function* $F_1(s_1, \rho_w) \rightarrow \rho'_w$ *takes the secret seed* s_1 *and keyword* w*'s position* ρ_w *in the original query or dictionary as inputs to generate a new position* ρ'_w *in the shuffled query or dictionary. The shuffle function can be considered to generate a random permutation of keywords [27] with a secret seed. Thus, without the secret seed, the ADL cannot deduce* ρ_w *from* ρ'_w*.*

Pseudonym function. *Pseudonym function* $F_2(s_2, i) \rightarrow \eta_i$ *is a secure hash function, such as SHA-1, that takes the secret seed* s_2 *and the file index* i *as inputs to calculate the file pseudonym* η_i*.*

Obfuscate function. *Obfuscate functions* $F_3(s_3, \eta_i) \rightarrow x_i$ *and* $F_4(s_4, \eta_i) \rightarrow y_i$ *are secure hash functions, such as SHA-1.* F_3 *takes the secret seed* s_3 *and the file pseudonym* η_i *as inputs to calculate an obfuscate factor* x_i *for the occurrence of keywords in the file.* F_4 *takes the secret seed* s_4 *and the file pseudonym* η_i *as inputs to calculate an obfuscate factor* y_i *for the file content.*

Map functions. *Map functions* $h_j(\rho'_w) \rightarrow \rho_{w,j}$ *for* $1 \leq j \leq log(k)$ *and* $g_j(\eta_i) \rightarrow \rho_{i,j}$ *for* $1 \leq j \leq log(f)$ *are a set of hash functions as those in Bloom filter[28] and can be publicly available. The* h_j *map function takes keyword* w*'s position* ρ'_w *in the shuffled query or dictionary as inputs to determine* $\rho_{w,j}$ *the positions of keyword-pseudonym pair in the file pseudonym buffer. The* g_j *map function takes file* F_i*'s pseudonym* η_i *as inputs to determine* $\rho_{i,j}$ *the positions of occurrence-content pair in the file content buffer.*

Appendix B. Correctness analysis

We prove the correctness of the COPS protocol using following theorems:

Theorem 1. *The* QueryMerge *algorithm can generate a query containing all users' interests.*

Proof of Theorem 1. To verify the correctness of the *QueryMerge* algorithm, we need to prove that the merged query generated by this algorithm contains all users' interests. Query Q_i denotes that user i wants to retrieve files containing at least one keyword in its keyword set K_i. The merged query Q, generated by executing "OR" operations on all user queries entry by entry, denotes that the users want to retrieve files containing at least one keyword in $\vee_{i=1}^{n} K_i$. Thus, the merged query contains all users' interests. ■

Theorem 2. *The* privateSearch *algorithm can return files matching each user's query with high probability.*

Proof of Theorem 2. To verify the correctness of the *privateSearch* algorithm, we need to prove that this algorithm can return files matching each user's query with high probability. The work by [3] has proved that f files matching the query can be returned with high probability if each file-content pair is mapped $log(f)$ times in a buffer of size $2f \cdot log(f)$, where unrelated information is encrypted to 0, and related information is encrypted to some value larger than 0. Therefore, when we set the buffer size and mapping times as in [3], we need to prove that unmatched information is encrypted to 0, but matched information is encrypted to some value larger than 0.

Note that each pair (c_w, e_w) is generated with Eq. 1:

$$c_w = Q[\rho_w] = E(PK_{ADL}, \vee_{i=1}^{n} Q_i[\rho_w]); e_w = c_w^{\xi_w} = E(PK_{ADL}, \xi_w \cdot \vee_{i=1}^{n} Q_i[\rho_w])$$

If there are no users interested in keyword w, then $\vee_{i=1}^{n} Q_i[\rho_w] = 0$. Thus we have:

$$c_w = E(PK_{ADL}, 0); e_w = E(PK_{ADL}, 0)$$

Therefore, each unmatched keyword-pseudonym pair is encrypted to 0.

In a similar way, each pair (c_i, e_i) is generated with Eq. 3:

$$c_i = \prod_{w \in W_i} E(PK_{ADL}, Q[\rho_w])^{x_i} = E(PK_{ADL}, x_i \cdot \sum_{w \in W_i} Q[\rho_w])$$
$$e_i = c_i^{y_i \cdot |F_i|} = E(PK_{ADL}, y_i \cdot |F_i| \cdot x_i \cdot \sum_{w \in W_i} Q[\rho_w])$$

If no keyword in W_i is interested by any user, then $\sum_{w \in W_i} Q[\rho_w] = 0$. We have:

$$c_i = E(PK_{ADL}, 0); e_i = E(PK_{ADL}, 0)$$

Therefore, each unmatched occurrence-content pair is encrypted to 0.

Now, we need to prove that matched information is encrypted to some value larger than 0. Note that, in Eq. 1, if there is at least one user interested in keyword w, then $\vee_{i=1}^{n} Q_i[\rho_w] = u$, where u is some value larger than 0. Thus, we have:

$$c_w = E(PK_{ADL}, u); e_w = E(PK_{ADL}, u \cdot \xi_w)$$

Therefore, each matched keyword-pseudonym pair is encrypted to some value larger than 0.

In the similar way, in Eq. 3, if at least one keyword in W_i is interested by one user, then $\sum_{w \in W_i} Q[\rho_w] = u$, where $u > 0$. We have:

$$c_i = E(PK_{ADL}, x_i \cdot u); e_i = E(PK_{ADL}, y_i \cdot |F_i| \cdot x_i \cdot u)$$

Therefore, unmatched information is encrypted to 0, but matched information is encrypted to some value larger than 0. ■

Theorem 3. *The* ResultDivide *algorithm can correctly divide files to each user.*

Proof of Theorem 3. To verify the correctness of the *ResultDivide* algorithm, we need to prove that the ADL can correctly divide files to each user, so that the user can only obtain files matching his query. Based on user i's query Q'_i, the ADL knows the positions of user i's keywords in the shuffled query, which are actually the positions of 1s. With Eq. 4, for each position j where $Q'_i[j] = 1$, the ADL can determine the pseudonyms of files, which contain the keyword with position j in Q'_i, will be stored in $B_1[h_1(j)]$, $\ldots$, $B_1[h_{log(k)}(j)]$. In Eq. 2, the pair (c_w, e_w), where $\rho_w = j$ in the shuffled dictionary, is just stored in $B_1[h_1(j)]$, $\ldots$, $B_1[h_{log(k)}(j)]$. Therefore, the ADL can correctly obtain all file pseudonyms for each user.

With Eq. 5, for each pseudonym η_*, the ADL can determine that the blinded file content $|F_*|'$ will be stored in $B_2[g_1(\eta_*)]$, $\ldots$, $B_2[g_{log(f)}(\eta_*)]$. In Eq. 3, the pair (c_*, e_*), is just stored in these locations. Therefore, the ADL can correctly divide results to each user. ■

Theorem 4. *The* FileRecover *algorithm can successfully recover file contents for each user.*

Proof of Theorem 4. To verify the correctness of the *FileRecover* algorithm, we need to prove that each can successfully recover file contents given the results obtained from the ADL. Note that the results sent to each user are in the form of $\{(\eta_*, |F_*|')\}$. In Eq. 5, $|F_*|'$ is generated as follows:

$$|F_*|' = e'_*/c'_*$$

In Theorem 1, we know that:

$$e'_* = |F_*| \cdot y_* \cdot u \cdot x_*; c'_* = u \cdot x_*; e'_*/c'_* = |F_*| \cdot y_*$$

With Eq. 6, the user can recover $|F_*|$ as follows:

$$|F_*|'/y_* = \frac{|F_*| \cdot y_*}{y_*} = |F_*|$$

Therefore, each user can successfully recover all wanted files. ■