Towards Differential Query Services in Cost-Efficient Clouds

Qin Liu, Chiu C. Tan, *Member, IEEE,* Jie Wu, *Fellow, IEEE* and Guojun Wang, *Member, IEEE*

Appendix A
Additional Examples

In this section, we provide examples for the Ostrovsky scheme, EIRQ-Efficient, and EIRQ-Privacy, to better illustrate their working processes. In all examples, we assume that $Dic = \langle A, B, C, D\rangle$ and that sample files stored in the cloud are as shown in Table 1.

A.1 The Ostrovsky Scheme

Suppose that Alice, whose public key is pk, wishes to retrieve files with keywords $\{A, B\}$. The working process of the Ostrovsky scheme is as follows:

Step 1: Alice runs the *GenerateQuery* algorithm to send an encrypted query $Q = \langle E_{pk}(1), E_{pk}(1), E_k(0), E_k(0)\rangle$ to the cloud.

Step 2: the cloud runs the *PrivateSearch* algorithm to generate c-e pairs, as shown in Table 2, and maps the c-e pairs into an encrypted buffer, as shown in Fig. 1. For example, F_1 contains keywords A and B, which corresponds to the first and second bits in Q. Therefore, the cloud will multiply them together to form $c_1 = E_{pk}(1) \cdot E_{pk}(1) = E_{pk}(2)$, and powers $|F_1|$ to c_1 to form $e_1 = c_1^{|F_1|} = E(2 \cdot |F_1|)$. Then, the cloud maps/multiplies (c_1, e_1) to the first and second entries of an encrypted buffer, where each entry is initialized with $(E_{pk}(0), E_{pk}(0))$. Each entry has one of the three statuses: survival, collision, and mismatch. As shown in Fig. 1, if only one matched file is mapped, the entry state is survival, e.g., Entry 1 and Entry 3; if more than one matched file is mapped, the entry state is collision, e.g., Entry 2; if no matched files are mapped, the entry state is mismatch, e.g., Entry 4, Entry 5, and Entry 6.

Step 3: Alice runs the *FileRecover* algorithm to recover files. For example, after decryption, Alice can obtain plaintext c-e pair $(2, 2|F_1|)$ from Entry 1, whose state is survival. Thus, Alice can compute $2|F_1|/2$ to recover $|F_1|$.

- *Qin Liu and Guojun Wang are with the School of Information Science and Engineering, Central South University, Changsha, Hunan Province, P. R. China, 410083. E-mail: gracelq628@yahoo.com.cn, csgjwang@mail.csu.edu.cn*
- *Chiu C. Tan and Jie Wu are with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122, USA. E-mail: {cctan, jiewu}@temple.edu*

TABLE 1
Sample files in the cloud

File name	Keywords	File name	Keywords
F_1	$\{A, B\}$	F_4	$\{C\}$
F_2	$\{B, C\}$	F_5	$\{D\}$
F_3	$\{C, D\}$		

TABLE 2
Sample c-e pairs

	c value	e value
F_1	$E_{pk}(1) \cdot E_{pk}(1) = E_{pk}(2)$	$E_{pk}(2)^{\lvert F_1\rvert} = E_{pk}(2 \cdot \lvert F_1\rvert)$
F_2	$E_{pk}(1) \cdot E_{pk}(0) = E_{pk}(1)$	$E_{pk}(1)^{\lvert F_2\rvert} = E_{pk}(1 \cdot \lvert F_2\rvert)$
F_3	$E_{pk}(0) \cdot E_{pk}(0) = E_{pk}(0)$	$E_{pk}(0)^{\lvert F_3\rvert} = E_{pk}(0)$
F_4	$E_{pk}(0)$	$E_{pk}(0)^{\lvert F_4\rvert} = E_{pk}(0)$
F_5	$E_{pk}(0)$	$E_{pk}(0)^{\lvert F_5\rvert} = E_{pk}(0)$

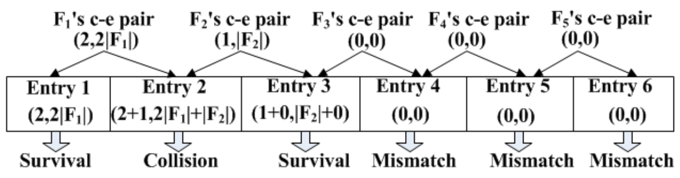

Fig. 1. Mapping files twice to a buffer of six entries. $E_{pk}(a) \cdot E_{pk}(b)$ is abbreviated to $a + b$.

A.2 The EIRQ-Efficient Scheme

To make it easier to understand EIRQ-Efficient, we will use the following example to describe its working process. Suppose that queries are classified into $0 \sim 4$ ranks, where Alice wishes to retrieve files with keywords $\{A, B\}$ and Rank-0 query, and Bob wishes to retrieve files with keywords $\{A, C\}$ and Rank-1 query.

EIRQ-Efficient works as follows: (1) Alice and Bob send their queries to the ADL. (2) The ADL generates a mask matrix M, as shown in Fig. 2, where M is a 4-row and 4-column matrix. Keywords A, B, C, and D correspond to the first, second, third, and fourth row in M, respectively. According to our rules, keywords A and B are Rank-0 keywords, C is a Rank-1 keyword, and D is a Rank-4 keyword. Thus, $M[1, 1], \ldots, M[1, 4]$, $M[2, 1], \ldots, M[2, 4]$, and $M[3, 1], \ldots, M[3, 3]$ are set to 1; the remainder elements are set to 0. (3) For each file F_j where $1 \leq j \leq 5$, the cloud generates c_j, as shown in Fig. 2, then powers the file content to c_j to obtain e_j, and maps (c_j, e_j) to a buffer. For example,

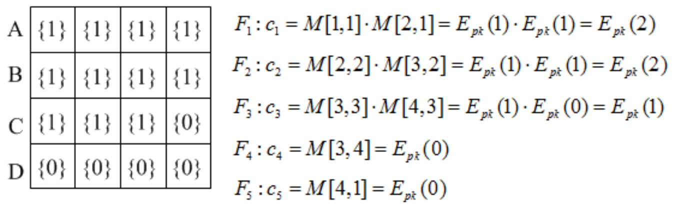
A {1} {1} {1} {1}
B {1} {1} {1} {1}
C {1} {1} {1} {0}
D {0} {0} {0} {0}
$F_1 : c_1 = M[1,1] \cdot M[2,1] = E_{pk}(1) \cdot E_{pk}(1) = E_{pk}(2)$
$F_2 : c_2 = M[2,2] \cdot M[3,2] = E_{pk}(1) \cdot E_{pk}(1) = E_{pk}(2)$
$F_3 : c_3 = M[3,3] \cdot M[4,3] = E_{pk}(1) \cdot E_{pk}(0) = E_{pk}(1)$
$F_4 : c_4 = M[3,4] = E_{pk}(0)$
$F_5 : c_5 = M[4,1] = E_{pk}(0)$

Fig. 2. Mask matrix in EIRQ-Efficient. Symbol "{}" denotes Paillier encryption under the ADL's public key pk.

the first and second rows of M correspond to F_1's keywords A and B, respectively; the cloud calculates $M[1,k] \cdot M[2,k] = E_{pk}(1) \cdot E_{pk}(1) = E_{pk}(2)$ to obtain c_1, where $k = 1$. According to our rules, F_1 and F_2 are Rank-0 files, which should be returned with probability 1, F_3 and F_4 are Rank-1 files, which should be returned with probability 75%, and F_5 is a Rank-4 file, which should not be returned. After this step, c_4 and c_5 are processed to encrypted 0s, and thus F_4 and F_5 are filtered out before mapping. (4) The ADL decrypts the buffer to obtain file contents $\{|F_1|, |F_2|, |F_3|\}$, and distributes files $\{|F_1|, |F_2|\}$ to Alice, and files $\{|F_1|, |F_2|, |F_3|\}$ to Bob.

A.3 The EIRQ-Privacy Scheme

Suppose that queries are classified into $0 \sim 4$ ranks, where Alice wishes to retrieve files with keywords $\{A, B\}$ and Rank-0 query, and Bob wishes to retrieve files with keywords $\{A, C\}$ and Rank-1 query. Therefore, A and B are Rank-0 keywords, C is a Rank-1 keyword, and D is a Rank-4 keyword.

We provide the sample mask matrix and the sample c-e pairs as follows: If $\gamma_0 = 7$ and $\gamma_1 = 3$, the sample mask matrix M is as shown in Fig. 3, where M is a 4-row and 7-column matrix. For each file, the cloud multiplies the rows that correspond to file keywords, element by element, to form a resulting row, as shown in Fig. 3. For example, F_1 contains keywords A and B which correspond to the first and second rows of M, respectively. After multiplying these two rows together, for F_1, all elements in the resulting row are an encryption of 2. Therefore, we have the following c-e pairs: for F_1, 7 pairs are in the form of $(E_{pk}(2), E_{pk}(2 \cdot |F_1|))$; for F_2, 3 pairs are in the form of $(E_{pk}(2), E_{pk}(2 \cdot |F_2|))$, and 4 pairs are in the form of $(E_{pk}(1), E_{pk}(1 \cdot |F_2|))$; for F_i, 3 pairs are in the form of $(E_{pk}(1), E_{pk}(1 \cdot |F_i|))$, and 4 pairs are in the form of $(E_{pk}(0), E_{pk}(0))$, where $i = 3, 4$; for F_5, 7 pairs are in the form of $(E_{pk}(0), E_{pk}(0))$.

Appendix B
Additional Discussions

In this section, we will provide discussions from four aspects: the overhead at the ADL, the querying delays, no trusted ADL, and an additional performance comparison. The notations are shown in Table 3.

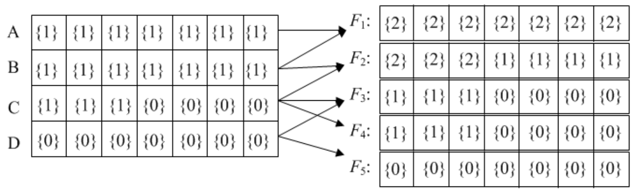
A {1} {1} {1} {1} {1} {1} {1}
B {1} {1} {1} {1} {1} {1} {1}
C {1} {1} {1} {0} {0} {0} {0}
D {0} {0} {0} {0} {0} {0} {0}
F1: {2} {2} {2} {2} {2} {2} {2}
F2: {2} {2} {2} {1} {1} {1} {1}
F3: {1} {1} {1} {0} {0} {0} {0}
F4: {1} {1} {1} {0} {0} {0} {0}
F5: {0} {0} {0} {0} {0} {0} {0}

Fig. 3. Mask matrix in EIRQ-Privacy. Symbol "{}" denotes Paillier encryption under the ADL's public key pk.

B.1 Overhead at the ADL

One concern with our solution is that the use of an ADL may become a performance bottleneck. We will analyze the computation and communication costs on the ADL as follows. In terms of computation cost, the ADL first encrypts every element of a mask matrix with the Paillier cryptosystem, whose running time is $O(\max(\gamma_i) \cdot d)$ in the worst case (EIRQ-Privacy); then, the ADL decrypts a buffer entry by entry, whose running time is $O(f_i \cdot \log(f_i/(i/r + \alpha)))$ in the worst case (EIRQ-Simple).

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. For kind (1), the ADL first receives n queries from n users, the size of which is $O(n \cdot d)$, then distributes searching results to each user, the size of which is $O(\sum_{i=1}^{n} f^i)$. For kind (2), the ADL first sends a combined query to the cloud, the size of which is $O(\max(\gamma_i) \cdot d)$ in the worst case (EIRQ-Privacy), and then receives a buffer from the cloud, the size of which is $O(\sum_{i=0}^{r} f_i \cdot \log(f_i/(i/r + \alpha))$ in the worst case (EIRQ-Simple).

During the interactions with all of the users, all messages are transferred through local area networks, the speed of which is much faster than accessing the Internet, and the communication cost is lower. In practice, even without any cryptographic approach, the messages from each user to the cloud are forwarded by an organization gateway. The communication cost at the ADL is almost the same as the cost at the gateway. In addition, our solution can be easily extended to multiple ADLs.

B.2 Querying Delays

An ADL is analogous to a *query aggregator* maintained by an organization that has thousands of users inside, querying the cloud. To aggregate sufficient queries, the organization may require the ADL to wait for a period of time before running our schemes, which may incur a certain querying delay. However, for the cost-efficient cloud applications, a certain degree of querying delay is tolerable to the users for saving cost. For example, consider that an organization has outsourced many files to the cloud for better sharing by its staff. Each staff member downloads files of his interest from the cloud by performing keyword-based searches. When more than

1. We use subscript i for notations f, p, q, γ, and β to reflect rank, and we use p', f^i, and f'_i to denote threshold failure rate, the number of files that match the *i-th* user's query, and the number of files *matching* Rank-i query but *mismatching* higher ranked queries, respectively.

TABLE 3
Summary of Notations

Notation [1]	Description
d	Number of keywords in the dictionary
r	The lowest query rank
f	The estimated number of files matching Q
p, q	Failure rate, survival rate
α, β, γ	Threshold value, buffer size, mapping times

one user wishes to download file F_j, the querying cost will be reduced if the ADL combines their queries to download F_j once.

Note that the degree of aggregation can be controlled through a time-out mechanism to meet a given querying delay requirement, based on different policies. One policy is to let the organization set a parameter T during system setup, as to determine the time that the ADL will wait before querying the cloud. When T is set to zero, this is degraded to a normal sequential search. An alternative policy is to allow each user to personally specify the tolerable delay. Let T_i denote the tolerable delay for user i. User i will send T_i together with his query to the ADL, which will record T_i in an *alarming* table. If any of the time in the alarming table is overdue, the ADL will generate a mask matrix with all of the queries received so far, and send it to the cloud.

B.3 No Trusted ADL

The ADL as an aggregator will collect all messages between the users and the cloud, and may become the biggest attacking target. To avoid possible information leaking, it may be required to hide user privacy from the ADL. We will discuss how our schemes work without a trusted ADL, based on the techniques used in our previous work [1]. The system model is shown in Fig. 4, where the *Combiner* and the *Distributor* substitute the ADL. As in [1], three kinds of secure functions are used, the secret seeds of which are shared by the users and the cloud: *Shuffle function* is used to shuffle a dictionary; *Pseudonym function* is used to calculate file pseudonym; *Obfuscate function* is used to obfuscate file content.

For ease of exposition, we use the following example to describe the working process of EIRQ-Efficient without a trusted ADL: suppose that $Dic = \langle A, B, C, D\rangle$ and queries are classified into $0 \sim 4$ ranks. Alice uses keyword $\{A\}$ with Rank-0 query, and Bob uses keyword $\{B\}$ with Rank-1 query. The sample files stored in the cloud are as shown in Fig. 4.

Step 1. Alice and Bob send their mask matrices (see Fig. 5), denoted as M_{Alice} and M_{Bob}, to the Combiner, and send their shuffled queries, denoted as Q_{Alice} and Q_{Bob}, to the Distributor. Each user sets the values of mask matrix elements as the *MatrixConstruct* algorithm in EIRQ-Efficient, and encrypts each element under the Distributor's public key pk. The shuffled queries are 0-1 bit strings of d bits. Each user first shuffles the dictionary to Dic', then sets the *i-th* bit of the query to 1 only if the

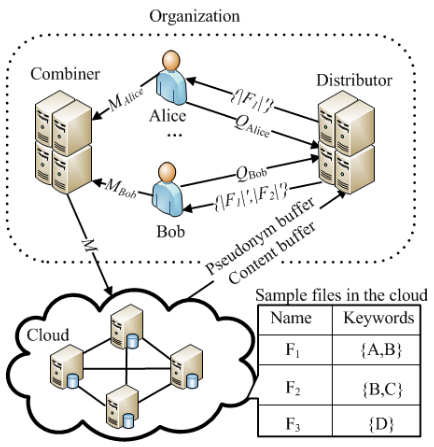

Fig. 4. System model without a trusted ADL.

i-th keyword in Dic' is chosen. If $Dic' =< B, C, A, D >$, $Q_{Alice} =< 0, 0, 1, 0 >$ and $Q_{Bob} =< 1, 0, 0, 0 >$.

Step 2. The Combiner generates a combined mask matrix M by performing multiplications on M_{Alice} and M_{Bob} element by element (see Fig. 5), and sends M to the cloud.

Step 3. The cloud processes M on the whole file collection, and returns two buffers, *pseudonym buffer* and *content buffer*, to the Distributor. The pseudonym buffer is constructed as follows. Each keyword $Dic[i]$ is associated with a c-e pair, denoted as (c_{w_i}, e_{w_i}). The cloud chooses the element in the *i-th* row and the first column of M as c_{w_i}, and then powers ξ_{w_i} to c_{w_i} to form e_{w_i}, where ξ_{w_i} is the concatenation of all pseudonyms of files that contain $Dic[i]$. For example, let η_j denote F_j's pseudonym. $(c_A, e_A) = (E_{pk}(1), E_{pk}(1 \cdot \xi_A))$, where $\xi_A = \eta_1$; $(c_B, e_B) = (E_{pk}(1), E_{pk}(1 \cdot \xi_B))$, where $\xi_B = \eta_1 || \eta_2$; the c-e pairs of other keywords are encrypted to 0s. Finally, the cloud shuffles the dictionary, and uses a series of *map functions* (h-type) to map (c_{w_i}, e_{w_i}) to multiple entries of the pseudonym buffer, where $1 \le i \le d$. The input of these map functions is the position of a keyword in the shuffled dictionary, and the output is a mapping location in the pseudonym buffer. The content buffer is constructed as follows. Each file F_j is associated with a c-e pair, denoted as (c_j, e_j). The cloud generates c_j as the *FileFilter* algorithm in EIRQ-Efficient, and powers $|F_j|'||\eta_j$ to c_j to form e_j, where $|F_j|' = (|F_j| \cdot x_j)$ with x_j being the obfuscating factor generated by the obfuscate function. For example, $(c_1, e_1) = (E_{pk}(2), E_{pk}(2 \cdot (|F_1|'||\eta_1)))$; $(c_2, e_2) = (E_{pk}(1), E_{pk}(1 \cdot (|F_2|'||\eta_2)))$; the c-e pairs of other files are encrypted to 0s. Finally, the cloud shuffles the dictionary, and uses another series of *map functions* (l-type) to map (c_i, e_i) to multiple entries of the content buffer, where $1 \le i \le t$. The input of these mapping functions is the pseudonym of a file, and the output is a mapping location in the content buffer.

Step 4. The Distributor recovers $\{\xi_A, \xi_B\}$ from the pseudonym buffer, and recovers $\{|F_1|'||\eta_1, |F_2|'||\eta_2\}$ from the content buffer, as the *FileRecover* algorithm in the Ostrovsky scheme. Then, it distributes $\{|F_1|'\}$ and

$$
\begin{array}{c}
M_{\text{Alice}} \\
\begin{array}{|c|c|c|c|}
\hline
\{1\} & \{1\} & \{1\} & \{1\} \\ \hline
\{0\} & \{0\} & \{0\} & \{0\} \\ \hline
\{0\} & \{0\} & \{0\} & \{0\} \\ \hline
\{0\} & \{0\} & \{0\} & \{0\} \\ \hline
\end{array}
\end{array}
\quad
\begin{array}{c}
M_{\text{Bob}} \\
\begin{array}{|c|c|c|c|}
\hline
\{0\} & \{0\} & \{0\} & \{0\} \\ \hline
\{1\} & \{1\} & \{1\} & \{0\} \\ \hline
\{0\} & \{0\} & \{0\} & \{0\} \\ \hline
\{0\} & \{0\} & \{0\} & \{0\} \\ \hline
\end{array}
\end{array}
\quad
\begin{array}{c}
M \\
\begin{array}{|c|c|c|c|}
\hline
\{1\} & \{1\} & \{1\} & \{1\} \\ \hline
\{1\} & \{1\} & \{1\} & \{0\} \\ \hline
\{0\} & \{0\} & \{0\} & \{0\} \\ \hline
\{0\} & \{0\} & \{0\} & \{0\} \\ \hline
\end{array}
\end{array}
$$

Fig. 5. Sample mask matrices. Symbol “{}” denotes Paillier encryption under the Distributor’s public key pk.

$\{|F_1|', |F_2|'\}$ to Alice and Bob, respectively. To distribute searching results to each user correctly, the Distributor first uses the positions of 1s in a shuffled query as the inputs of the h-type map functions to locate the pseudonyms of all matched files in the pseudonym buffer. It then uses the pseudonyms of these matched files as the inputs of the l-type map functions as to locate the obfuscated contents of the matched files in the content buffer.

Step 5. Each user recovers file content by dividing the obfuscated file content by the obfuscated factor. For example, Alice recovers $|F_1|$ by calculating $|F_1|'/x_1$, and Bob recovers $|F_1|$ and $|F_2|$ by calculating $|F_1|'/x_1$ and $|F_2|'/x_2$, respectively.

Analysis. User privacy is protected from the Combiner and the Distributor as follows. The messages aggregated by the Combiner are several mask matrices. Since the mask matrices are encrypted under the Distributor’s public key, the Combiner cannot know the users’ access privacy and rank privacy. The messages aggregated by the Distributor are shuffled queries and two buffers. First, the secret seed of the shuffle function is blind to the Distributor, and thus it cannot know the users’ access privacy and rank privacy. Second, the cloud only maps file pseudonyms and obfuscated file contents to the buffers. Since the secret seeds of the pseudonym function and the obfuscate function are blind to the Distributor, it cannot know which files are really returned. Note that, to ensure that our schemes work well without trusted ADL, we assume that the Combiner and the Distributor will not collude with any other entities. In other words, our schemes fail when the following cases happen: (1) the Combiner colludes with the user or the cloud. The Combiner will know the shuffle function, with which it can deduce which keywords each user is searching for; (2) the Distributor colludes with the user or the cloud. The Distributor will know the obfuscate function and the pseudonym function, with which it can deduce which files are returned to each user.

B.4 Additional Comparisons

In this section, we compare the computation and communication costs incurred on the cloud between *no ranked* queries under the ADL (No Rank), the Ostrovsky scheme [2], and the work by Bethencourt et al. [3] (see Table 4). Suppose that there are n users querying the cloud with threshold failure rate p', t files stored in the cloud whose keywords constitute a public dictionary of size d, f^i files matching the *i-th* user’s query, and f files matching the combined query. In terms of computational cost, we only consider the cost of the exponential operation, which is the most expensive. In No Rank, the running time is mainly on the encryption of t files, denoted as $O(t)$. In [2] and [3], the *PrivateSearch* algorithm will be run n times, and each will perform encryptions on t files. Thus, the total computational cost is $O(n \cdot t)$.

TABLE 4
No Rank vs. Ostrovsky scheme vs. Bethencourt scheme

Scheme	Communication	Computation
Ref. [2]	$O(n \cdot d + \sum_{i=1}^{n} f^i \cdot \log(f^i/p'))$	$O(n \cdot t)$
Ref. [3]	$O(n \cdot d + \sum_{i=1}^{n} f^i)$	$O(n \cdot t)$
No Rank	$O(d + f \cdot \log(f/p'))$	$O(t)$

In terms of communication cost, we consider the query size sent to the cloud and the buffer size obtained from the cloud. In No Rank, the ADL sends a combined query of size $O(d)$ to the cloud, which will return a buffer of size $O(f \cdot \log(f/p'))$ to the ADL. In [2], user i sends his query of size $O(d)$ to the cloud, which will return a buffer of size $O(f^i \cdot \log(f^i/p'))$ to the user. In [3], user i sends his query of size $O(d)$ to the cloud, which will return a buffer of size $O(f^i)$ to the user. Thus, the total communication cost of [2] and [3] is $O(n \cdot d + \sum_{i=1}^{n} f^i \cdot \log(f^i/p'))$ and $O(n \cdot d + \sum_{i=1}^{n} f^i)$, respectively.

References

[1] Q. Liu, C. Tan, J. Wu, and G. Wang, “Cooperative private searching in clouds,” *Journal of Parallel and Distributed Computing*, 2012.
[2] R. Ostrovsky and W. Skeith, “Private searching on streaming data,” in *Proc. of CRYPTO*, 2005.
[3] J. Bethencourt, D. Song, and B. Waters, “New techniques for private stream searching,” *ACM Transactions on Information and System Security*, 2009.