2018 IEEE Confs on Internet of Things, Green Computing and Communications, Cyber, Physical and Social Computing, Smart Data, Blockchain, Computer and Information Technology, Congress on Cybermatics

A Homomorphic Searching Scheme For Sensitive Data In NoSQL Database

Baohua Huang
School of Computer and Electronic Information
Guangxi University
Nanning 530004, Guangxi, China
bhhuang66@gxu.edu.cn

Sheng Liang
School of Computer and Electronic Information
Guangxi University
Nanning 530004, Guangxi, China
3104516237@qq.com

Dongdong Xu
School of Computer and Electronic Information
Guangxi University
Nanning 530004, Guangxi, China
dongdxu@foxmail.com

Zhuohao Wan
School of Computer and Electronic Information
Guangxi University
Nanning 530004, Guangxi, China
355060761@qq.com

***Abstract*—NoSQL database system is widely used in Big Data environment for its high performance and scalability. Due to the width of usage of NoSQL database, it is inevitable to store sensitive data in it. Utilizing homomorphic encryption, sensitive data can be encrypted and calculated in the form of ciphertext to preserve privacy. Although calculating on homomorphically encrypted data is possible, searching on these data is not simple for the calculating results are encrypted and the search engine doesn't know what data should be sent back to the requester. A homomorphic searching scheme is proposed to handle this problem. In this scheme, the searching function is divided into two parts named homomorphic comparing and homomorphic decryption. The homomorphic comparing is executed by the encrypted data holder, and the homomorphic decryption is executed by the key holder. With this scheme, encrypted data holder can only get encrypted data and calculate the encrypted differences of these data, and the key holder can only get the encrypted differences and decrypt them. A sample implementation of the scheme with MongoDB is given. Analysis and experimental results show that the proposed scheme is secure and practical.**

Keywords—NoSQL Database, Big Data, Privacy, Homomorphic, Encrypted Difference

I. INTRODUCTION

With the blossom of Big Data and Cloud Computing, NoSQL database has been widely used in personal health record management[1], spatial data store[2], security log store[3], storing and analyzing the internet of things[4], etc. Obviously , some of these data are very sensitive. Generally, privacy-sensitive data in a database must be manipulated under more attentions [5].

Although NoSQL databases are very widely used in Big Data environment, there are not much mechanism to provide security guarantees[6]. For example, HBase is a typical NoSQL database system based on Hadoop and HDFS. Though HBase has some great mechanisms in data access control, but it still has some defects in permission access control[7].

MongoDB is another typical NoSQL database system with cross-platform and document-oriented supporting[8]. When it is running in shared mode, there is no authentication and authorization supported, so MongoDB in shared mode is incapable to store sensitive data. When it is running in standalone or replica-set mode, authentication and authorization are enabled. Even if authentication has been enabled, then the Mongo databases supports only two types of users: read-only and read-write. Read-only users can query everything in the database, while read-write users have full access to all the data in the database on which they are defined[9].

Because NoSQL database systems generally focus on improving performance and scalability to fulfill the requirements of Big Data environment, they pay less attention on the security. With the widespread usage of NoSQL database, there are some works related to enhance the security of NoSQL database emerged. For example, Lai et al propose a fine grained access control framework which extends the access control permissions according to the atomic operations of HBase[7]. Hou et al analyze MongoDB security and design injection defense solution[8]. In our previous work, we developed a transparent middleware for encrypting data in MongoDB[10]. Xu et al propose CryptMDB, a practical encrypted MongoDB over Big Data[11].

The existing works cover a lot of security topics, but there is few works discussing searching over encrypted data in NoSQL database as far as we know. In this paper, we propose a homomorphic searching scheme for sensitive data in NoSQL database. The homomorphic encryption function is used to encrypt sensitive data into calculatable ciphertext. In order to preserve privacy of sensitive data, calculating of searching procedure and calculating results decryption are done by two different parts.

II. PROBLEM MODELING

It is common to encrypt sesitive data, then to store the encrypted data in NoSQL database for privacy preserving. In Big Data environment, searching sensitive data is very necessary for volume of data is so huge that it is not possible to return all data to the data requester. In order to searching over encrypted data without decryption, we can utilize homomorphic encryption.

A. Homomorphic Encryption

The origin of homomorphic encryption is privacy homomorphism introduced by Rivest, Adleman and Dertouzous shortly after the invention of RSA by Rivest, Shamir, and Adleman[12]. The main idea of homomorphic encryption is to encrypt plain data into ciphertext, calculate on ciphertext directly to get result in ciphertext, and decrypt the result ciphertext to get plain result finally.

Formally, if encryption algorithm *enc* and decryption algorithm *dec*,

$$enc: M \rightarrow C \tag{1}$$

$$dec: C \rightarrow M \tag{2}$$

Where *M* is the set of plain data, *C* is the set of ciphertext data.

$$\forall\ op: I \rightarrow O,\ enc: I \rightarrow I_c \tag{3}$$

$$\exists op_c: I_c \rightarrow O_c,\ dec: O_c \rightarrow O \tag{4}$$

Where *I* is the set of input of operation *op*, *O* is the set of output of operation *op*. $I \subset M, O \subset M, I_c \subset C, O_c \subset C$. op_c is the operation on the ciphertext.

Then *enc* with *dec* can be called homomorphic encryption.

There are various types of homomorphic encryption algorithm, e.g partial homomorphic encryption, somewhat homomorphic and full homomorphic encryption[12], etc. The partial homomorphic encryption algorithm can only support limited operations on the ciphertext, e.g multiplication or addition. The somewhat homomorphic encryption algorithm can support all necessary operations but the operation steps are limited. The full homomorphic encryption algorithm can support all operations and unlimited operation steps.

To support searching on ciphertext, the full homomorphic is not necessary for the operation and operation steps are both limited.

B. *Privacy Preserving Searching*

With the help of homomorphic encryption, we can calculate on ciphertext data without decryption. This can conduct to privacy preserving searching.

All searching is the procedure of comparing, so the atomic operation is calculating the difference between the query data and the data from database. So, the privacy preserving searching can based on homomorphic encryption supporting subtraction. Formally, d_c is the ciphertext of data in database, q_c is the ciphertext of query data, the ciphertext of difference between d_c and q_c is

$$r_c = d_c -_c q_c \tag{5}$$

$$\mathrm{dec}: r_c \rightarrow \mathrm{r} \tag{6}$$

where $-_c$ is subtraction on ciphertext, r_c is the ciphertext of difference, and *r* is the plaintext of r_c. If *r* equals zero, then the query data is found in database and vice versa.

In the searching procedure, if d_c and q_c both are not decrypted, the data searching is privacy preserving even if r_c, the ciphertext of the difference between them needs to be decrypted.

C. *The Problem of Searching with Homomorphic Encryption*

In the procedure of searching described in previous section, if r_c can be decrypted, then d_c and q_c can be decrypted also for r_c is calculated from d_c and q_c, and these data are encrypted with the same key.

To tackle this problem, we need to separate the calculating and decryption of the searching procedure to executing on two host in different domain, so as to separate the key from the ciphertext from NoSQL database and ciphertext from the user's query.

III. HOMOMORPHIC SEARCHING SCHEME

The homomorphic searching scheme tries to apply homomorphic encryption to encrypt plain data into calculatable ciphertext, and to deploy calculating on Search Engine, and to place decryption on Decrypter.

A. *Structure of the Scheme*

In order to separate calculating and decryption of searching procedure, we can design the structure of the scheme as showing in Fig. 1.

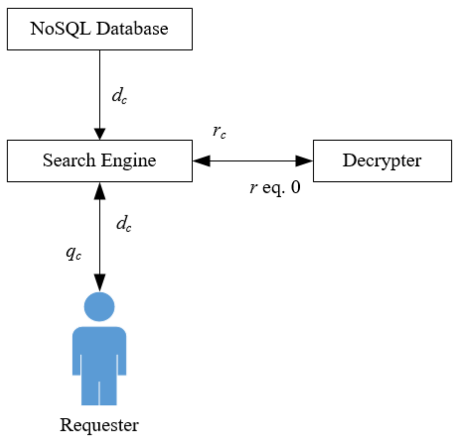

Fig. 1. Structure of Homomorphic Searching Scheme

In the structure showing in Fig. 1., sensitive data were encrypted with homomorphic encryption and stored in NoSQL database. The Search Engine and Decrypter are two services running separately on two hosts in different security domain. Thus, the ciphertext data and its decryption key are separated. None of Search Engine and Decrypter has both ciphertext data and key, so the data privacy is preserved.

B. *Searching procedure*

As described above, the searching procedure consists of two main phases: calculating and decryption. The calculating phase gets the ciphertext of difference between database data and user's query; the decryption phase gets the plaintext difference. The detail of searching procedure is showing in Fig. 2.

```
1:  Requester send $q_c$ to Search Engine
2:  Get $d_c$ from NoSQL Database
3:  While $d_c$ is not null
4:      $r_c = d_c - q_c$
5:      Send $r_c$ to Decrypter
6:      $r$ = dec(rc)
7:      Send $r$ == 0 to Search Engine
8:      If $r$ then
9:          Send $d_c$ to Requester
10:         $d = dec(d_c)$
11:     End if
12:     Search Engine get $d_c$ from NoSQL Database
13: End while
```

Fig. 2. Searching Procedure

```
1:  Requester send $q_c$ to Search Engine
2:  Do
3:      $D_c = \emptyset$
4:      $R_c = \emptyset$
5:      $R = \emptyset$
6:      For $i$=1 to $bs$
7:          Get $d_c$ from the NoSQL Database
8:          If $d_c \neq null$
9:              $r_c = d_c - q_c$
10:             Put $d_c$ into $D_c$
11:             Put $r_c$ into $R_c$
12:         Else
13:             Break for
14:         End if
15:     End for
16:     If $D_c \neq \emptyset$
17:         Send $R_c$ to Decrypter
18:         For each $r_c \in R_c$
19:             $r$ = dec($r_c$)
20:             Put $r$ == 0 into R
21:         End for
22:         Send $R$ to Search Engine
23:         i=1
24:         For each $r \in R$
25:             If $r$ then
26:                 $d_c = D_c(i)$
27:                 Send $d_c$ to Requester
28:                 $d$ = dec($d_c$)
29:             End if
30:             i = i +1
31:         End for
32:     End if
33: While $D_c \neq \emptyset$
```

Fig. 3. Improved Searching Procedure

In the searching procedure showing in Fig. 2, the statements are executed by the Requester, Search Engine or Decrypter. If it is not specified clearly, the executer of a statement is the data target of the previous statement.

For the Search Engine and the Decrypter are running on different host, the communication between them is very time-consuming. We can reduce the communication number between them by batch data sending. Let *bs* denote the batch size, the searching procedure can be improved as showing in Fig. 3.

IV. Sample Implementation with MongoDB

MongoDB is a widely used NoSQL database in Big Data environment and we had developed a transparent middleware[10] for encrypting data in it in our previous work, so we implemented the homomorphic searching scheme with MongoDB on the basis of our previous work.

A. MongoDB Access and System Architecture

We use our previous transparent middleware for encrypting data in MongoDB to access the database and do homomorphic searching. Classes in the middleware are showing in Table I.

TABLE I. Classes in the Middleware

Original Classes	**Newly-defined Classes**
Mongo	EncryptedMongo
DB	EncryptedDB
DBCollection	EncryptedDBCollection
BasicDBObject	EncryptedBasicDBObject
DBCursor	EncryptedDBCursor
	TobeEncrypted

These newly-defined classes are modified to encrypt data with homomorphic encryption algorithm and to support searching over encrypted data.

The architecture of the implemented sample system is showing in Fig. 4.

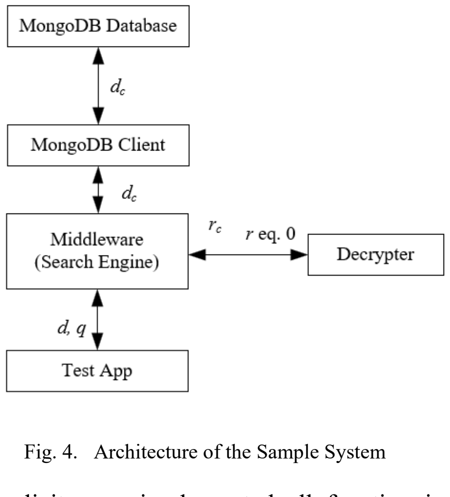

Fig. 4. Architecture of the Sample System

For simplicity, we implemented all function including encryption and searching in the middleware, so the middleware is also the Search Engine.

B. Homomorphic Encryption Algorithm

In the sample implementation of the homomorphic searching scheme, we use the sample secret key homomorphic encryption scheme proposed by Craig Gentry[12]. The key is an odd integer $p > 2N$. An encryption of a bit b is simply a random multiple of p, plus a random integer with the same parity as b. Formally, The ciphertext of b *is*

$$c = b + 2x + kp \tag{7}$$

where x is a random integer in $(-n/2, n/2)$, and k is an integer chosen from some range. In decryption,

$$b = (c \bmod p) \bmod 2 \tag{8}$$

where $(c \bmod p)$ is the number in $(-p/2, p/2)$ that equals c modulo p. Actually, $(c \bmod p)$, which is the noise parameter in this scheme, will be in $[-n, n]$, since $b + 2x$ is in that range.

The decryption would have worked correctly as long as

$$b + 2x \in [-N, N] \subset (-\frac{p}{2}, \frac{p}{2}) \tag{9}$$

According to formula (9), multi steps of calculation on ciphertext may lead to decryption error. For homomorphic searching needs only one step of subtraction, so we can choose parameters carefully to make sure the decryption working correctly.

C. Experimental Result

Performance is the main concern of the searching scheme, so we test the implemented sample system with a small volume of data to observe the actual performance. The test is running on a PC with 16GB Ram, one Intel i7 CPU with 8 cores. The operating system is Fedora 24.

Firstly, we test time needed in searching under various conditions. According to the searching scheme, the searching time must relate to number of encrypted data in database and the size of batch. The test results are showing in Fig. 5.

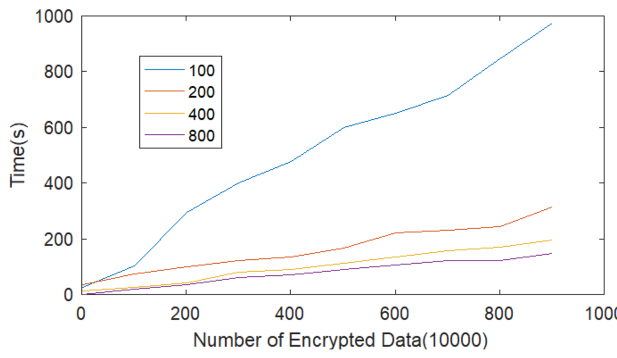

Fig. 5. Searching Time of Different Batch Size

From Fig. 5 we can see that (1) the searching time and number of encrypted data have a linear correlation. The searching time linearly increase with the number of encrypted data. (2) The batch size affect the rate of increase. The small the batch size is, the high the rate of increase will be. (3) There is a upper limitation of batch size, to outnumber it will not get lower increase rate of time.

The searching time may be related to the number of query result item count. We test this condition and the results are showing in Fig. 6.

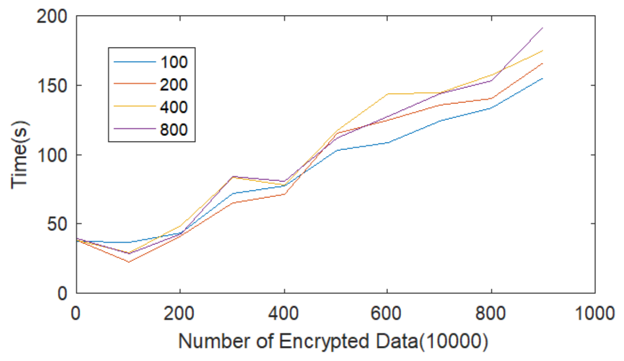

Fig. 6. Searching Time of Different Result Item Count

From Fig. 6 we can see that searching time is not very closely related to the result item count.

V. Related Works

With the rapid development of Big Data and Cloud Computing, data management becomes a hot topic of researching and engineering for the volume of data increase exponentially. There are works focus on power efficiency[13] , Scalable and Cost-efficient[14] of low level data storage, etc. There are also a lot of works focus on high level data management. NoSQL database is a kind of data management system running at the high level, which directly support application.

NoSQL database is widely used in Big Data and Cloud Computing environment. For example, Sarkar et al use Hadoop to manage personal health record [1]. Zhang et al propose HBaseSpatial, a scalable spatial data storage based on HBase[2]. Jeong et al propose a NoSQL-based method to collect and integrate security logs using MapReduce[3]; Lehmann et al design a data structure for storing time-series data in a MongoDB document for optimal query performance of large datasets, and implement Office Analysis as a Service using MongoDB[4]. Obviously , there are a lot of sensitive data managed by NoSQL database.

Although NoSQL database is widely used in Big Data environment and stores a lot of sensitive data that must be manipulated under more attentions [5], there are not much mechanism to provide security guarantees[6]. HBase and MongoDB are two typical example.

HBase is an open source project of Apache, and is built on Hadoop and HDFS. In HBase, the granularity of access is column. Although Apache Accumulo project says it can ensure cell control and provide the key-value pair of security, the deployment and configuration for Apache Accumulo is not very convenient. Additionally, permissions of HBase is coarse-grained. It only has 5 different permissions, such as READ, WRITE, CREATE, EXEC and ADMIN[7].

MongoDB is a document database managing collections of JSON-like documents. In MongoDB, data-files are unencrypted, and Mongo doesn't provide a method to automatically encrypt these files. It heavily utilizes JavaScript as an internal scripting language. Internal commands available to the developer are actually short JavaScript scripts. It is even possible to store JavaScript functions in the database in the db.system.js collection that are made available to the database users. This makes MongoDB potential for injection attacks. Mongo databases supports only two types of users: read-only and read-write. Read-only users can query everything in the database, while read-write users have full access to all the data in the database on which they are defined. As for auditing, MongoDB doesn't provide any facilities for auditing actions performed in the database[9].

Driving by the requirement of high security in NoSQL database, some works related to enhance the security of NoSQL database emerged. Lai et al propose a fine grained access control framework which extends the access control permissions according to the atomic operations of HBase[7]. Hou et al analyze MongoDB security and design injection defense solution[8]. In our previous work, we developed a transparent middleware for encrypting data in MongoDB[10]. Xu et al propose CryptMDB, a practical encrypted MongoDB over Big Data. It utilizes an additive homomorphic asymmetric cryptosystem to encrypt user's data and achieve strong privacy protection[11]. These existing works don't discuss searching over encrypted data in NoSQL database.

VI. Conclusion

There are few works focus on searching encrypted data in NoSQL database as far as we know, but it is very necessary for the widely used NoSQL database have to contain sensitive data in ciphertext. Homomorphic encryption makes it possible to calculate on ciphertext and get encrypted result. Searching need to know the calculating result in plaintext to decide if the compared data should send to requester. The proposed homomorphic searching scheme separates the calculation and decryption to achieve privacy preserving of sensitive data. For NoSQL database and search engine, they only have encrypted data; for Decrypter, even if it has the key of encrypted data, it can only access the encrypted difference of database data and query data and decrypt this difference into plaintext.

In the next step, we are going to use more complex homomorphic encryption scheme in the searching scheme, and design encrypted index to improving the searching performance.

Acknowledgment

This work was financially supported by the National Natural Science Foundation of China (No. 61262072)

References

[1] B. B. Sarkar, S. Paul, B. Cornel, N. Rohatinovici, and N. Chaki, "Personal health record management system using hadoop framework: An application for smarter health care," in *7th International Workshop on Soft Computing Applications(SOFA 2016)*, Arad, Romania, 2016, pp. 385-393.

[2] N. Zhang, G. Zheng, H. Chen, J. Chen, and X. Chen, "HBaseSpatial: A scalable spatial data storage based on HBase," in *13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications(TrustCom 2014)*, Beijing, China, 2014, pp. 644-651.

[3] H. Jeong, X. Piao, J. Choi, J. Shin, and P. Kim, "Efficient integration method of large-scale heterogeneous security logs Using NoSQL in cloud computing environment," *Journal of Internet Technology,* vol. 17, pp. 267-275, 2016.

[4] M. Lehmann, A. Biorn-Hansen, G. Ghinea, T.-M. Gronli, and M. Younas, "Data analysis as a service: an infrastructure for storing and analyzing the internet of things," in *12th International Conference on Mobile Web and Intelligent Information Systems(MobiWis 2015)*, Rome, Italy, 2015, pp. 161-169.

[5] G. Livraga, *Protecting Privacy in Data Release*. Springer Cham Heidelberg New York Dordrecht London: Springer International Publishing AG Switzerlan, 2015.

[6] A. M. Chacko, M. Fairooz, and S. D. Madhu Kumar, "Provenance-aware NoSQL databases," in *4th International Symposium on Security in Computing and Communications(SSCC 2016)*, Jaipur, India, 2016, pp. 152-160.

[7] Y.-Y. Lai and Q. Qian, "HBase fine grained access control with extended permissions and inheritable roles," in *16th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing(SNPD 2015)*, Takamatsu, Japan, 2015, pp. 1-5.

[8] B. Hou, Y. Shi, K. Qian, and L. Tao, "Towards Analyzing MongoDB NoSQL Security and Designing Injection Defense Solution," in *3rd IEEE International Conference on Big Data Security on Cloud(BigDataSecurity 2017)*, Beijing, China, 2017, pp. 90-95.

[9] L. Okman, N. Gal-Oz, Y. Gonen, E. Gudes, and J. Abramov, "Security issues in NoSQL databases," in *10th IEEE Int. Conf. on Trust, Security and Privacy in Computing and Communications(TrustCom 2011)*, Changsha, China, 2011, pp. 541-547.

[10] X. Tian, B. Huang, and M. Wu, "A Transparent Middleware for Encrypting Data in MongoDB," in *2014 IEEE Workshop on Electronics, Computer and Applications*, Ottawa, Canada, 2014, pp. 906-909.

[11] G. Xu, Y. Ren, H. Li, D. Liu, Y. Dai, and K. Yang, "CryptMDB: A practical encrypted MongoDB over big data," in *2017 IEEE International Conference on Communications(ICC 2017)*, Paris, France, 2017, pp. 1-6.

[12] C. Gentry, "A Fully Homomorphic Encryption Scheme," Doctor, STANFORD UNIVERSITY, 2009.

[13] L. Zhang, Y. Deng, W. Zhu, J. Zhou, and F. Wang., "Skewly Replicating Hot Data to Construct a Power efficient Storage Cluster," *Journal of Network and Computer Applications,* vol. 50, pp. 168-179, 2015.

[14] J. Xie, Y. Deng, G. Min, and Y. Zhou, "An Incrementally Scalable and Cost-efficient Interconnection Structure for Datacenters," *IEEE Transactions on Parallel and Distributed Systems,* vol. 28, pp. 1578-1592, 2017.