Querying Encrypted Graph Databases

Nahla Aburawi, Alexei Lisitsa and Frans Coenen
Department of Computer Science, University of Liverpool, U.K.

Keywords: Graph Databases, Database Security, CryptDB, Encryption.

Abstract: We present an approach to execution of queries on encrypted graph databases. The approach is inspired by CryptDB system for relational DBs (R. A. Popa et al). Before processing a graph query is translated into encrypted form which then executed on a server without decrypting any data; the encrypted results are sent back to a client where they are finally decrypted. In this way data privacy is protected at the server side. We present the design of the system and empirical data obtained by experimentation with a prototype, implemented for Neo4j graph DBMS and Cypher query language, utilizing Java API. We report the efficiency of query execution for various types of queries on encrypted and non-encrypted Neo4j graph databases.

1 INTRODUCTION

Recently proposed CryptDB approach (Popa et al., 2011) provides a powerful and elegant mechanism for security protection of data against server based attacks. The key idea of CryptDB is that by choosing an appropriate encryption scheme, the data stored on the database server in an encrypted form never need to be decrypted at the server side, not even during the execution of the queries. In this way, neither a curious database administrator, nor an attacker having full access to the server can learn sensitive data. To some extent such a mechanism allows the alleviation security concerns of outsourced computations, such as those in cloud environments. Another advantage of CryptDB approach is that it does not require any changes in the server software. All functionality is implemented in a front-end component which intercepts users' queries in transit, rewrites them and passes to the server for execution.

The original CryptDB system (Popa et al., 2011) was designed for relational data model and database management systems. It demonstrated good performance by imposing reasonable overhead on the server and bringing querying over encrypted data into the realm of practical applications. This could be compared favourably with the generic computations over encrypted data, which, despite the remarkable recent progress in *fully homomorphic encryption* schemes remain computationally expensive and practically challenging. The good performance of CryptDB has been achieved by utilizing a collection of SQL-aware encryption schemes, which is allowed to reveal only the necessary information to execute the various types of SQL-queries, still keeping data itself hidden. One particular challenge was to support Join operations and this required introduction of a new cryptographic primitive.

In this paper, we address the development of CryptDB-like mechanism for graph databases, which have recently become very popular. The graph data model with nodes and links bearing data elements and labels allows for more natural and straightforward data modeling in many contexts. Furthermore graph querying mechanisms could be considerably more efficient than relational ones for some types of queries.

There are several popular implementations of Graph DBMS available (Robinson et al., 2013; Vukotic et al., 2014; tit,). In a work reported here we have used Neo4j, version 2.3.9, by Neo Technology Inc. Neo4j database management system comes with the graph query language Cypher. Cypher is a declarative language with a syntax gives a natural way to express the patterns of nodes and relationships in the graph to be matched during the query execution. It is not require to describe how we can do the select, insert, update or delete from our graph data.

In this paper, we aim to take CryptDB principles as they are implemented for relational databases and transfer them to graph databases. We first elaborate requirements for *CryptGraphDB* component for Cypher query languages and Neo4j database system. It has turned out that SQL-aware encryption schemes can be seamlessly re-used as Cypher-aware schemes,

Aburawi, N., Lisitsa, A. and Coenen, F.
Querying Encrypted Graph Databases.
DOI: 10.5220/0006660004470451
In *Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP 2018)*, pages 447-451
ISBN: 978-989-758-282-0

at the same time preserving the performance advantages of *traversal* graph queries over equivalent relational ones.

In the rest of the paper, we present the design of the system and empirical data obtained by experimentation with a prototype, implemented for Neo4j graph DBMS and Cypher query language, utilizing Java API. We report the efficiency of query execution for various types of queries on encrypted and non-encrypted Neo4j graph databases.

2 PROPOSED DESIGN

In our project we aim to take CryptDB principles as they are implemented in relational databases and apply them to graph databases. We first elaborate requirements for the proposed *CryptGraphDB* component of Cypher query language and Neo4j database system.

2.1 CryptGraphDB

CryptGraphDB works by enabling the execution of Cypher queries over encrypted data. We adopt three ideas originally proposed in CryptDB for relational DBMS: a Cypher-aware encryption strategy, adjustable query-based encryption, and onion encryption.

2.1.1 Cypher-aware Encryption Strategy

CryptGraphDB should enable the applications of different encryption schemes depending on the queries to be executed, so it maps Cypher queries to the encryption schemes to be used.

2.1.2 Adjustable Query-based Encryption

The adjustable query-based encryption is the main feature of CryptGraphDB. Some queries do not require any checking or comparison, so the nodes properties will be encrypted with RND (random level of encryption not revealing any information; different instances of *equal* plain texts are likely mapped to *different* encrypted texts). For the nodes properties that are required to be checked for equality during query execution, DET, that is deterministic level of encryption mapping equal plain texts to equal encrypted texts, is used. As a result, we need to allow CryptGraphDB to adjust the encryption level of each data item based on user queries. This could be done either in advance, if a set of queries is fixed in advance, or during run time, in which case an adjustable onion layered encryption should be used.

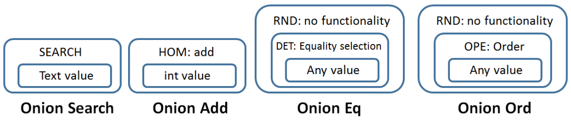

Figure 1: Onion layers of encryption.

2.1.3 Onion Layers of Encryptions

Onion Layers of Encryption allow changes to data encryption levels in an efficient way. To implement adjustable query-based encryption, the onion of encryption allows the encryption of each data item. Each layer of each onion enables some kinds of functionality. As an example, outermost layers such as RND and HOM give maximum security, while the inner layers such as OPE provide more functionality. So, the adjustable query-based encryption dynamically adapts the layer of encryption on the DBMS server as illustrated in Figure 1.

Different cryptographic systems are available to be cascaded into onions layers (here we follow the original CryptDB design):

- **Random (RND).** RND provides the maximum privacy, it was designed to address, that two equal values will be planned to different ciphertexts. RND does not allow any efficient computation to be executed on the ciphertext.
- **Homomorphic Encryption (HOM).** HOM is an encryption scheme that allows the server to perform computations on encrypted data to be secure.
- **Word search (SEARCH).** SEARCH allows the implementation of a cryptographic protocol for keyword searches on encrypted text. SEARCH allows the server to detect repeating words in a given node.
- **Deterministic (DET).** DET only leaks which encrypted values match to the same data value, and no more. DET was implemented to let the equality checks to be performed.
- **Order-preserving Encryption (OPE).** OPE allows to establish order relations and perform comparisons between data values based on their encrypted versions.

2.1.4 Graph vs Relational CryptDB

The main requirement for encryption adjustment is that it has to select an appropriate encryption layer which would 1) reveal enough information about encrypted data for query execution; and 2) do not reveal more information than necessary. It has been noticed in (Popa et al., 2011) that the application of DET layer for execution of SQL queries involving Join operator may lead to unnecessary leaks of information, such as cross-column equalities. In order to address the issue (Popa et al., 2011) proposed a new Join-aware encryption scheme. We notice that graph database querying does not require Join operator and it is absent in the Cypher query language. Nevertheless, under the natural translation of relational databases to graph databases, such as by work flow proposed by Neo4j manual (export relational database instance to CSV file and upload it to Neo4j as a graph DB instance), one may show that the issue of unnecessary leaks remains. Furthermore the original Join-aware encryption scheme can be used as a "property-aware" encryption scheme for Cypher query execution. Related strategies of encryption adjustment are a subject of our ongoing work.

Further improvement in security protection in CryptGraphDB will come from the development of *traversal-aware* encryption adjustment applied dynamically during the execution of graph traversal queries. The development of such schemata and studying the related trade-offs between efficiency and security protection is another topic of our ongoing research.

2.2 CryptGraphDB Principles

The required steps to process the query by CryptGraphDB are illustrated in Figure 2. Processing a query in CryptGraphDB involves the following steps:

1. The application creates a query, which the proxy intercepts by anonymizing [Label, Nodes, Properties] and encrypt the constants.
2. Proxy checks if the DBMS needs to adjust encryption level, if yes, issue an update query to adjust the layer.
3. Proxy sends the encrypted query to the DBMS server, to execute it using standard Cypher and return the encrypted results.
4. Proxy decrypts the query results, and sends them back to the application.

In order to rewrite the query in encrypted format. The syntactical structure of the query remains the same, but its syntactical components are encrypted/replaced as follows:

- **Property Names:** (encrypted/renamed).
- **Property Values:** (comes from the current layer encryption value).
- **Node Names:** (encrypted/renamed).
- **Relationship Names:** (encrypted/renamed).
- **Relationship Values:** (comes from the current layer encryption value).

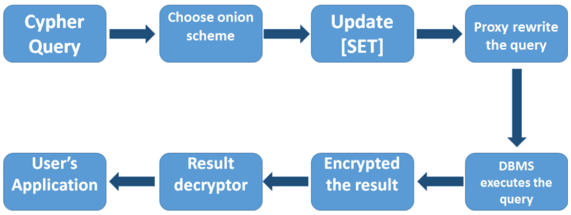

Figure 2: The CryptGraphDB Design.

2.3 Implemented Prototype

The first prototype of a fragment of the above CryptGraphDB design is implemented using Java API. The main focus here is to evaluate the efficiency of execution of Cypher queries over encrypted Neo4j database.

Several ways for integration Neo4j with client applications have been proposed. These include Neo4j Server REST extension, using Neo4j server with JDBC, using Neo4j's Embedded Java API().

Embedded Java API provides with the tight integration of the client Java program with Neo4j server, that is embedding a server into an application. So it does not allow for the actual separation of CryptGraphDB component and the server, as assumed by the design. However, it allows for quick prototyping, where the client, the proxy and the server are all parts of the same application, and experimenting with the execution of the queries on the encrypted store.

For the first prototype only one encryption layer, that is DET has been implemented. To this end, we have used an implementation of AES (Advanced Encryption Standard) algorithm available in Java Cryptographic Architecture (JCA) package.

Neo4j Native Java API is used to create a Neo4j database in the chosen path, as shown here:

```
GraphDatabaseFactory dbFactory = new
GraphDatabaseFactory();
GraphDatabaseService db=
dbFactory.newEmbeddedDatabase("C:/Neo4j");
```

Note that you will need to include this transaction:

```
   try (Transaction tx =
graphDb.beginTx());tx.success();
```

in order to start Neo4j database transaction.

To execute the Cypher query we need to call the function `execute`, as shown in the following example:

```
 ExecutionResult execResult =
execEngine.execute("MATCH (n) WHERE not(
(n)-[ ]-( ) ) RETURN n");
```

In order to illustrate the procedure that required to implement CryptGraphDB principles on the query, we have a simple Cypher query:

```
   MATCH (A:PEOPLE)-[:KNOWS]-(B:PEOPLE)
WHERE A.name= "John"
RETURN B.name;
```

The prototype of CryptGraphDB performs a set of procedures:

1. Proxy informs DBMS to update the properties, DBMS decrypts them to DET layer.
2. Proxy encrypts ”John” to its EQ onion, DET layer encryption, then the proxy generates query and sends it to DBMS:
   ```
   MATCH (AA:PEOPLE1)-[:KNOWS1]-(BB:PEOPLE1)
   WHERE AA.name1="c9Yz1Og1PdfVKBrVnOk46Q"
   RETURN BB.name1;
   ```
3. Proxy receives the encrypted result, then decrypts it. finally sends it back to the application user.

2.4 Experimental Setup

We have used the implemented prototype system to conduct the experiments and to study the efficiency of Cypher queries executed on an encrypted graph database.

For the first group of the experiments we have manually created an instance of a graph database with 250 nodes and 200 relationships. Some nodes were intentionally left orphan, meaning they have neither incoming, nor outgoing edges. Several types of relationships were used and the variety of paths lengths were used from a binary relationship to a path of up to the lengths of ten was available.

2.5 Queries

The set of queries was designed to test some of the common types of queries. For example, traversals are important to define data objects (nodes) originate from or affected by some starting object or node. Another popular operation is searching for particular values within a specific property. The queries were divided into two types: *structural* and *data* queries.

The structural queries:

- **S0:** Find all orphan nodes. Which means, find all nodes in the graph with no incoming edges and no outgoing edges.
- **S1:** Traverse all nodes, transitive closure/ regular expression.
- **S2:** Traverse all the nodes in the graph.
- **S3:** Find all nodes with the incoming relationship, and count them.
- **S4:** Find all nodes with the outgoing relationship, and count them.
- **S5:** Find all nodes with the incoming relationship and outgoing relationship, and count them.
- **S6:** Traverse Nodes depends on their ID.

The data queries:

- **C1:** Count the number of nodes whose data is equal to specific property.
- **C2:** Get the nodes with a path according to specific relationships.
- **C3:** Count the number of nodes in the graph.

2.6 Results/ Analysis

Each query was run 12 times on the database and execution times were collected. All times are in milliseconds (ms). We have dropped the longest times and the shortest times, the remaining ten times were averaged. We have done this to make sure that the caching does not affect the timing. The data values were chosen randomly and in advanced all the same values were implemented for both the non-encrypted databases and the encrypted databases.

A summary information on the execution time for both non-encrypted and encrypted instances of the same database have been tabulated in table 1. For the structure queries, S0, S1, S2, S3, S4, S5, and S6, non-encrypted database was slightly faster, as shown in table 1 (upper part). On the other hand, there was also a small difference in the execution time in the data queries, C1, C2, and C3. Overall slowdown and acceleration, demonstrated for some of the queries, were insignificant.

For the sake of comparison, we have executed the same sets of queries, again over encrypted a non-encrypted instances using the native Neo4j interface. The results can be seen in Table 1 (lower part) . Unlike the case of the Java API, where the *wall time* was measured using Java methods, here *processor time* reported by Neo4j is shown. That explains one or two orders of magnitude difference between the tables. While slowdown is reported for S0 and S2 queries run over encrypted instance, other queries were in fact accelerated.

The results look encouraging for using CryptDB-like approach for graph databases. We need more empirical evidence, though on the larger databases.

Table 1: Querying Non-encrypted and Encrypted Databases with Java Interface and Neo4j Interfaces, time in ms.

Query	S0	S1	S2	S3	S4	S5	S6	C1	C2	C3
Java interface										
Non-encrypted DB	5030.8	5115.7	5256.6	5038.1	4931.1	5326	5455.5	5491.2	5248.8	5216.8
Encrypted DB	5058.4	5092.5	5252.1	5081.8	4881.2	5338.3	5471.9	5478.2	5287.1	5510.4
Slowdown	0.54%	-0.45%	-0.085%	0.86%	-1.012%	0.23%	0.3%	-0.24%	0.72%	-5.33%
Neo4j interface										
Non-encrypted DB	278.5	49	76.3	64.3	68.4	89.1	94.8	63.8	71.8	79.2
Encrypted DB	360.7	46.1	80.3	75.1	41	63	85.2	59.5	40.7	53.8
Slowdown	22.78%	-5.92%	4.98%	14.38%	-40.6%	-29.29%	-10.13%	-6.74%	-43.31%	-32.07%

3 RELATED WORK

Security is an important issue in database management systems and the data in a database management system need to be protected from unauthorized access. In this paper we have proposed CryptGraphDB mechanism inspired by CryptDB (Popa et al., 2011; Popa, 2014), which provides as a provable and practical privacy against any attack on the database server or curious database administrators. CryptDB works by implementing SQL queries as they are implementing on the plain database but on encrypted database. CryptDB does not change the structure of DBMS, in order to CryptDB consists of two parts, the first part is a trusted client-side front-end, which is responsible of non-encrypted database and the secret Master key. Furthermore, the second part is an untrusted DBMS server, it follows the schema of the encrypted database. The database server completely evaluates queries on encrypted data, then the results are sent back to the client to decrypt them.

Then in (Xie and Xing, 2014) authors proposed CryptGraph system, which aims to run graph analytics on the encrypted graph data structures to keep the user graph data and the analytic results secure. So they presented CryptGraph to empower the user of encrypting their graphs before uploading them to the cloud, and get the results back after analysis the encrypted graph in encrypted form, while only the user who can decrypt them to extract the plain text form. They have proposed to query its structure by computing polynomials. In contrast, in our work we aim to develop a system using not a specialized graph data structure and encodings, but rather an extension to existing general purpose graph databases, more in the spirit of the original CryptDB system.

Our particular set of queries used in the experiments was inspired by (Batra and Tyagi, 2012) in which a comparative analysis of relational and graph database querying is presented.

4 CONCLUSIONS

We have proposed CryptGraphDB system allowing to execute the queries over encrypted graph databases. We outlined the design of CryptGraphDB and reported on experiments with an initial prototype system implemented with Embedded Java API for Neo4j graph DBMS. We have shown that as performance concerned CryptGraphDB demonstrates promising results which warrant further development of the system. We have also outlined the future directions in the development of the encryption adjustment schemes specific for graph databases, such as property-aware and traversal-aware schemes. The implementation of these schemes and investigation of related trade-offs between efficiency and security is a topic for our future research.

REFERENCES

Titan: Distributed graph database. http://titan.thinkaurelius.com/.

Batra, S. and Tyagi, C. (2012). Comparative analysis of relational and graph databases. *International Journal of Soft Computing and Engineering (IJSCE)*, 2(2).

Popa, R. A. (2014). *Building practical systems that compute on encrypted data*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA.

Popa, R. A., Redfield, C. M. S., Zeldovich, N., and Balakrishnan, H. (2011). Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA. ACM.

Robinson, I., Webber, J., and Eifrem, E. (2013). *Graph Databases*. O'Reilly Media, Inc.

Vukotic, A., Watt, N., Abedrabbo, T., Fox, D., and Partner, J. (2014). *Neo4J in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition.

Xie, P. and Xing, E. P. (2014). Cryptgraph: Privacy preserving graph analytics on encrypted graph. *CoRR*, abs/1409.5021.