Querying Encrypted Data in Graph Databases

Nahla Aburawi[1], Frans Coenen[1], and Alexei Lisitsa[1]

[1]Department of Computer Science, University of Liverpool, UK
{nahla.aburawi, coenen, A.lisitsa}@liverpool.ac.uk

Abstract. Encryption is an effective way to protect sensitive data in a database from various attacks. Querying encrypted data, however, becomes a challenge. Either the data should be decrypted before the querying, leaving it vulnerable to server-side attacks, or one has to apply computationally expensive methods for querying encrypted data. In this paper, we present a flexible mechanism for the execution of queries over encrypted graph databases. Data privacy is protected at the server side, through the use of multi-layered encryption and encryption layer adjustment, conducted dynamically during the execution of queries. The proposed scheme reveals less information to the adversary than in the case of static adjustment done prior to execution. We report on the implementation of the scheme as applied to a subset of Cypher graph queries (graph traversal queries) directed at a Neo4j graph database. The experimental results show the efficiency of query execution for various types of query on encrypted graph data stores.

Keywords: Graph databases; Encryption Adjustment; Data privacy; Security;

1 Introduction

Database security is attracting considerable interest due to the importance of data that is routinely hosted in enterprise databases, the large amounts of found in organizations of all sizes, from large corporations to small businesses. The goal of database security involves protecting the database from unauthorised or accidental access to data, modification or destruction.

In order to protect data integrity and privacy, data encryption has been used, as an active protection mechanism. One of the most challenging and important aspect of data encryption is how to query the encrypted data. Starting from the influential work of Popa et al. [9] a lot of research has been conducted in the field of querying encrypted database, avoiding full data decryption. Using multi-layered encryption and appropriate encryption adjustment procedures is the key to finding the right balance between security and query execution performance.

In this paper we present different mechanisms for adjusting encryption layers in the context of graph databases [11]. In order to provide a reasonable trade-off between data security protection and data processing efficiency CryptGraphDB [3] utilizes multi-layered encryption and encryption adjustment, inspired by the CryptDB system for relational DBs [9]. The graph query is translated into an encrypted form before processing. The encryption layers of the data are adjusted accordingly at the server side. Subsequently, the query is executed on a server and the encrypted results are sent back to a user where they are finally decrypted. In both, CryptDB and CryptGraphDB approaches various types of encryption are used, organized as encryption onion layers. Notice that in all mentioned works and in this paper only the variants of symmetric encryption as opposed to asymmetric (public key) encryption are used.

At the outermost layer highly secure encryption schemes that leak virtually nothing about the data are typically used. Most common examples of such encryption schemes are random, or randomized (RND) encryption schemes [10], meaning that the same plain text values are likely to be translated into different cipher texts under same encryption key. To make it possible to execute some queries over encrypted data which require, for example, equality checks, the encryption level should be adjusted to become a deterministic layer (DET). This allows for equality checking to be done, without revealing any more information. The DET layer can be easily provided by any deterministic symmetric encryption algorithm (e.g. DES or AES).

In this paper we elaborate a traversal-aware encryption adjustment mechanism for graph databases, first proposed in [2], report on its implementation for a subset of Cypher queries (traversal queries) and empirically evaluate its efficiency. The proposed mechanism reveals only the information required to execute the query. By using this approach, we can observe that not all property values in the graph are adjusted with respect to the DET layer, but only required values are adjusted, while the rest are still in the RND layer. One of the major drawbacks of the traditional way to search an encrypted database is to decrypt all the data to the DET layer, then find the required records. Apart from representing a significant security risk, this traditional approach is resource intensive particularly when considering a large number of records. Our technique shows a clear advantage by dynamically adjusting encryption layers as query execution progresses. In this way less information is revealed to any adversary watching the execution of the query on the encrypted store; while, as demonstrated in the paper, being reasonably efficient. The graph database system used to analyse the approach is Neo4j as developed by Neo Technology in 2007 [14].

The remainder of this paper is organized as follows. The related work is outlined in Section 2. In Section 3 querying of encrypted graph DB, as proposed in [2, 3], is presented. Section 4 then explains the implementation of the proposed traversal-aware encryption adjustment and case studies. The experiments and analyses of the performance of the proposed approach is presented in Section 5. Some conclusions and some suggested areas for future work are presented in Section 6.

2 Related Work

One of the challenges of querying encrypted databases is revealing unnecessary data while performing the adjustment of the encryption layers. In a classical CryptDB paper [9], dealing with such an issue for JOIN queries for relational databases, the authors introduced a special JOIN-aware encryption mechanism. In [3] a CryptDB-like approach, referred to as CryptGraphDB, was proposed for graph databases and it was particularly noted that due to absence of a JOIN operator in the Cypher graph query language, there is no need in use of JOIN-aware encryption; however, the problem of unnecessarily revealing information after encryption layer adjustment continued to exist. To address this problem, in a position paper [2] a novel scheme of traversal-aware encryption adjustment for graph DBs was proposed and theoretically discussed, but no implementation, or empirical evaluation was reported. The work on structured encryption and controllable disclosure presented in [6] provided an interesting alternative to the methods developed in [2, 3, 9] and this paper. In particular [6] uses a different cryptographic scheme under which the whole (graph) data structure is encrypted, not only the data elements. The latter, unlike the approach we present in this paper, could make an implementation of the methods of [6] on the top of an existing graph DBMS challenging. The detailed comparison of both approaches is a topic of future research.

3 Querying of Encrypted Graph DB: Existing Approach

As noted above, there has been little work on querying encrypted graph databases. The only work that the authors are aware of is the work on Graph CryptDB presented in [3] and, the work on Traversal-Aware Encryption Adjustment presented in [2]. In both cases, the implementation was done with respect to Neo4j graph DBMS and Cypher as a query language. Each is discussed in further detail in the following two sub-sections.

3.1 Graph CryptDB

CryptGraphDB [3] works by executing Cypher queries over an encrypted graph database. By translating the query into an encrypted form, executing it over the encrypted data on the server without any decryption and sending the results back to the user where they can be decrypted. In this way data privacy is protected at the server side. At the core of the Graph CryptDB are three ideas espoused by CryptDB [9] : (1) a Cypher-aware encryption strategy which maps Cypher queries to the encryption schemes; (2) Adjustable query-based encryption that lets CryptGraphDB adjust the encryption level of each data item based on the user query; and (3) Onion encryption to change data encryption levels in an efficient manner.

CryptGraphDB is composed of two parts: a trusted client-side frontend, and an untrusted DBMS server. The frontend keeps track of the database schema as seen by the application without encryption, and the current level of onion encryption exposed in the server for each data item. On the other hand, the server keeps track of the encrypted schema; the encrypted format of user data (the lowest level of encryption revealed to the server). It provides confidentiality for data content and for names of labels and properties; also, it does not hide the entire graph structure, the number of nodes, or the approximate size of data in bytes.

Processing a query Q in CryptGraphDB, as shown in Figure 1, involves four steps:

1. The application issues a query, that is rewritten by the proxy by anonymizing each label, node and relationship names; and encrypts each constant in the query with the most private encryption scheme (RND).
2. The proxy checks whether the DBMS needs to adjust the encryption level. If so, it sends an update query to adjust the encryption level.
3. The proxy sends the encrypted query to the DBMS server to be executed using a standard Cypher and returns the encrypted results.
4. The DBMS server returns the encrypted query result which are decrypted by the proxy and sent back to the application.

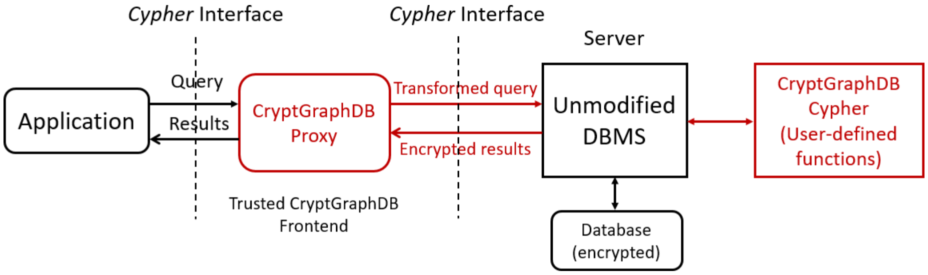

Fig. 1 The typical query flow in Graph CryptDB

In [3] the simple encryption adjustment method is proposed. Initially, each value in the graph is encrypted using RND, the outermost layer. At the second of the above steps checking by the proxy prior to query execution is done using simple syntactical criteria. In particular, if a value of some attribute may be required for the query in a plaintext form, or encrypted at DET level, the corresponding adjustments will be done across the whole data store. As was pointed out in [2] this may lead to unnecessary information leaks and therefore a more advanced Traversal-Aware Encryption Adjustment method for alleviating this issue was proposed in [2] as discussed in the following subsection.

An example of query processing with simple encryption adjustment

Given a query of the form:

```
MATCH (node:person)-[:knows]->( )
WHERE  node.name = {"Tom"}
RETURN node.age
```

In order to allow the equality check, this query needs to pass to the DET layer as detailed in Table 1.

Table 1. Data layout at the server where the application table on the left is created at the server from the table on the right.

person	
name	**age**
Tom	29
Smith	22

person			
name at RND	**age at RND**	**name at DET**	**age at DET**
a4a895a87052	e6ba69bdf08c	UD82Pv8uGNi7	33TPfYgeYDKb
9d60b415e6e7	686097aa7a7a	j39IjDVyx/+	NMtqlsMp8Qaf

Step (1), Proxy sends to the DBMS: `UPDATE person SET name=DECRYPT RND(name)`, DBMS decrypts entire name property to `DET` layer:

`DECRYPT name, RND(a4a895a87052) = UD82Pv8uGNi7`

Next, Proxy updates its internal state to log that name is currently at layer `DET` in the DBMS.

Step (2), the proxy encrypts `"Tom"`, to the `DET` layer encryption value of `UD82Pv8uGNi7`, then, proxy generates query and sends it to DBMS:

```
MATCH (node:person)-[:knows]->( )
WHERE node.name = {""UD82Pv8uGNi7"}
RETURN node.age
```

Step (3), the proxy receives encrypted `RND` level result `e6ba69bdf08c` and decrypts it using: `DECRYPT age,DET(DECRYPT age,RND(e6ba69bdf08c))= 29.` Lastly, proxy sends decrypted result 29 to the application.

3.2 Traversal-Aware Encryption Adjustment

The idea of Traversal-Aware Encryption Adjustment (TAEA) is quite natural and simple. During the query execution the paths starting with nodes with specific names values and progressing alongside specified relationships are traversed. The execution may perform additional checks of some properties of encountered nodes.

So, the adjustment will not be everywhere, but just along the query execution path. The scheme of traversal-aware encryption adjustment is dynamic, and the encryption adjustment happens not before the query execution, but rather it gradually progresses alongside the execution.

The scheme follows the simple principles defined in [2]:

- Encryption adjustments and traversal query execution are interlaced;
- The adjustments happen in-between traversal steps;
- The adjustment is performed to enable one step of traversal using all information accumulated to this step, in particular the set of nodes traversed so far.

By considering a simple case study in [2] (an execution of a simple query “on a paper”) it was shown that indeed TAEA may reveal less information to a potential attacker as compared with simple adjustment. However, the study in [2] was only a proof-of-concept study.

4 Towards Implementation of TAEA

In this section we describe an implementation of TAEA for a subset of the Cypher queries. We start with simple examples first. In general, an execution of a query with TAEA over an encrypted graph data store requires an execution of interlaced partial queries and encryption adjustment updates. While it is possible to compose these partial queries and updates using the WITH construct of Cypher, and thereby to execute all the sequence automatically (in one go), for simplicity, here we present the required sequence of separate queries and updates. The composition is discussed later in sub-section 5.2.

Query with a single relationship. Consider a query `Q` consisting of one link and two search criteria:

```
MATCH (node1: label1)-[:Relationship]->(node2: label2)
WHERE node1.propertyA = {value1} AND
node2.propertyB = {value2} RETURN node2
```

The query `Q`, using the TAEA scheme, is processed, as follows:

1. Each value starts out encrypted with the most private encryption level where data is encrypted using the `RND` scheme.
2. To check the equality for the first part of the `WHERE` clause, `node1.propertyA = value1`, we need to lower the encryption of `propertyA` to level `DET`. The proxy issues this query to the server `UPDATE Label1 SET propertyA = DECRYPT RND(propertyA)`, that use the `DECRYPT RND` UDFs, where `DECRYPT RND` is a user defined function implementing decryption which is discussed in sub-section 4.1.

3. Executing the query Q_1 to allow the initial search node1.propertyA = encrypted value1 for nodes of Q to be executed. Here encrypted value1 is the encryption of value1, when the path required in the main query Q start as:

   ```
   MATCH (node1:label1)-[:Relationship]->(node2:label2)
   WHERE node1.propertyA = {encrypted value1}
   RETURN node2 AS result
   ```

   Where result is used as an alias for the result column name.
4. Lowering the encryption level of node2.propertyB for nodes that are reachable from the outgoing of Q_1 to DET layer.
5. Processing the second part of the query Q:

   ```
   MATCH (node1:label1)-[:Relationship]->(result:label2)
   WHERE result.property2 = {encrypted value2}
   RETURN result
   ```

6. Finally, proxy decrypts the results from the server and returns them to the user.

Query with multiple relationships In the case of having a query Q consisting of multiple statements and two search criteria, as follows:

```
MATCH(node1:label)-[Rel1]->(node2:label)-[Rel2]->(node3:label)
WHERE node1.propertyA={value1} AND node2.propertyB = {value2}
AND node3.propertyC = {value3}
RETURN node3
```

Processing a query Q of the above form under TEAE is as follows:

1. Each value in the graph is encrypted using the RND scheme.
2. According to the first part node1.propertyA=value1 of Q, we need to lower the encryption of propertyA to level DET. By using DECRYPT RND UDF: UPDATE Label SET propertyA = DECRYPT RND(propertyA).
3. As Q has multiple links, we start with the first part R_1 which is (node1)-[Rel1]->(node2), and execute the query Q_1 to allow the initial search node1.propertyA=encrypt(value1) for nodes of Q to be executed. When the path required in the main query Q start as:

   ```
   MATCH (node1:label)-[Rel1]->(node2:label)-[Rel2]-
   >(node3:label)
   WHERE node1.propertyA = {encrypt(value1)}
   RETURN node2 AS result
   ```

   Here, result is used as an alias for result column name of Q_1, while encrypt(value1) is the encryption of value1.
4. In order to implement the second part Q_2 of Q, we need to lower the encryption level of result.propertyB for nodes that have an incoming relationship with the result variable of Q_1 to the DET layer.

5. Processing the second part Q_2 of the query Q:

   ```
   MATCH (result)<-[Rel2]-(node3:label)
   WHERE node3.propertyB = {encrypted value2}
   RETURN result
   ```

6. Finally, proxy decrypts the results and sends them back to the user.

We now consider general case of the *simple traversal query* of the form:

```
MATCH(node1:label_1)-[:Relationship1]->...(node_i:label_i)-
[:Relationship_i]->...(node_k: label_k)
WHERE node_1.property_1={value1} AND ... node_i.property_i
= {value_i}... AND node_k.property_k = {value_k}
RETURN node_k
```

The following is the process for resolving a query Q of the above form using the TEAE scheme:

1. Encrypt all values at RND layer.
2. Lowering the encryption of property _1 to level DET, by using decryptRND UDF: SET property 1 =decryptRND(property 1).
3. Execute Q_1 which is the first part of Q when the path required start as:

   ```
   MATCH(node_1:label_1)-[:Relationship1]->(node_i:label_i)

   WHERE node_1.property_1 = {encrypt(value1)}
   RETURN node_i AS result
   ```

   Here, result is used as an alias for the result column name of Q_1, while encrypt(value1) is the encryption of value1.
4. In order to execute the second part Q_2 of Q, we need to lower the encryption level of result.property-i for nodes that have an incoming relationship with result of Q_1 to the DET layer. Then, execute Q_2 as:

   ```
   MATCH (result)<-[:Relationship_i]-(node_k:label_k)

   WHERE result.property_i={encrypt(value_i)}

   RETURN result_1
   ```

5. For Q_3, lower the encryption of property_k for nodes that have an incoming relationship with result_1 of Q2 to DET, …
6. Finally, proxy decrypts the results and sends them back to the user.

4.1 Implementation

We implemented a prototype system for evaluating the performance of traversal-aware encryption adjustment. To build this prototype, we utilized AES (Advanced Encryption Standard) algorithm [1]. For the RND layer we used AES in CBC mode with an initialization vector (IV) obtained as the hash of ID of the node to which encrypted data belong to. For DET layer we used AES in ECB mode. The security parameter of AES key encryption schemes is 128-bit.

We create a set of User-Defined Functions (UDFs) to be called directly from Cypher queries [15]. The functions *encryptDET*, *encryptRND*, *decryptRND*, and *decryptDET* implement encryption and decryption for the DET and RND layers, respectively. UDFs are written in Java, they are packaged in a Jar-file, deployed into the $NEO4J_HOME/plugins, and then can be called in the same way as any other Cypher function.

4.2 Case studies

In this sub-section we present several examples of the queries executed on a particular graph data store under different encryption adjustment policies using implemented UDFs for encryption and decryption. Suppose we have a graph database of Person and two properties of interest: name and age, and the relationships KNOWS; the scenario is illustrated in Figure 2. Consider the Cypher query as follows:

```
MATCH (node1:person)-[:KNOWS]->(node2:person)
WHERE node1.name = "Tom" AND node2.age = "22"
RETURN node2
```

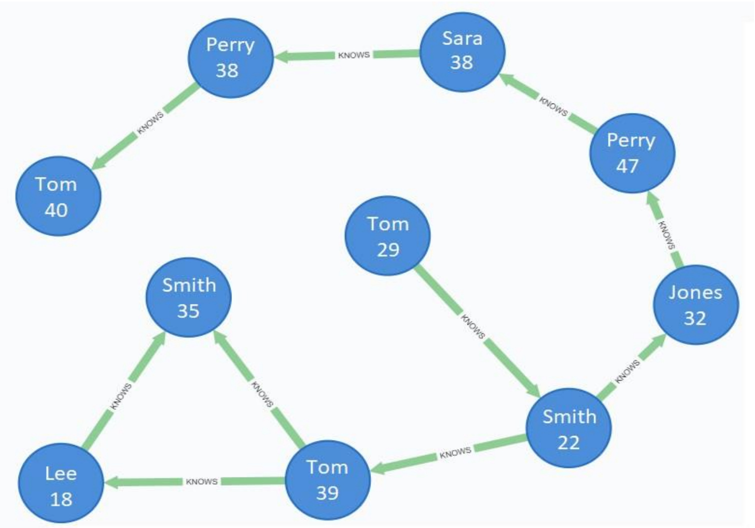

Fig. 2 Example data layout schema at the server of the graph database.

For this study, we considered the execution of the above query in three modes: (1) non-encrypted; (2) encrypted with simple adjustment; and (3) with traversal-aware encryption adjustment.

Non-encrypted The search criteria WHERE clause has two parameters, starts with node1.name = "Tom" when executed gives an output of three nodes {Smith,22}, {Smith,35},{Lee,18} to be traversed (these are reached from the {Tom,29} node in one step via the KNOWS relation). Next, execute the second part of the query node2.age = "22". Lastly, get the {Smith,22} node as the final result.

Encrypted with simple adjustment We observe from Figure 2 that the schema of the graph database at plain status. Initially, each value in the graph is encrypted within the RND layer as the outermost layer, as follows:

```
MATCH (n)
SET n.name =
encryptRND((encryptDET(n.name)),ID(n)),n.age=encryptRND
((encryptDET(n.age)),ID(n))
```

Where encryptRND and encryptDET are a user defined function implementing the encryption as mentioned in sub-section 4.1. Resolution of the query requires the lowering of the encryption of name and age to level DET (as we need to check the equality). To do so, we need to update the data by using SET clauses:

```
MATCH (n)
SET n.name = decryptRND(n.name,ID(n)) AND
n.age=decryptRND(n.age,ID(n))
```

Next, the proxy generates a query and sends it to DBMS:

```
MATCH (node1:person)-[:KNOWS]->(node2:person)
WHERE node1.name = "UD82Pv8uGNi7" AND node2.age =
"NMtqlsMp8Qaf"
RETURN node2
```

Where, "UD82Pv8uGNi7" and "NMtqlsMp8Qaf" are the encryption of "Tom" and "22", respectively. Lastly, proxy receives the encrypted result {j39IjDVyx/+,NMtqlsMp8Qaf}. Proxy sends the decrypted result {Smith,22} to the application.

Encrypted using Traversal-Aware Encryption Adjustment We note from Figure 2 that the graph at plain status. Initially, each value in the database is encrypted with the most secure RND layer, as listed in Table 2. The advantage is that the server can learn nothing about the data values.

Returning to resolving the example query Q:

```
MATCH (node1:person)-[:KNOWS]->(node2:person)
WHERE node1.name = "Tom" AND node2.age = "22"
RETURN node2
```

Subsequently, we need to remove the onion layer, as WHERE node1.name="Tom" requires lowering the encryption of name to level DET, the proxy issues the following query to the server UPDATE Label SET name = DECRYPT RND(name),

Table 2. Plain text data, encryption at the RND layer and encryption at the DET layer (Ciphertexts shown are not full-length).

name	age	name at RND	age at RND	name at DET	age at DET
Tom	29	a4a895a87052	e6ba69bdf08c	UD82Pv8uGNi7	33TPfYgeYDKb
Smith	22	9d60b415e6e7	686097aa7a7a	j39IjDVyx/+	NMtqlsMp8Qaf
Tom	39	9b078f653478	21da9938c098	UD82Pv8uGNi7	Ss67Waxq2n+m
Lee	18	6cf77f7817b1	bcb86ac44437	RQpqwfEE8Kbm	fxEYkxe7g+P27L
Smith	35	e7a86cbc36ff	83e6b8ab0edc	j39IjDVyx/+	5K6xJRUEJ2s+
Jones	32	141a99a21cf4	dfd2e8d1dfa2	ax+/5Q23fEl4	z0sfDuU2mIP/
Perry	47	ca06d68f7c6b	c1051d53aae2	0nPCg1bAxh8R	oSLl00rhMbeZ
Sara	38	a5cb936cd7ed	7ed31f9f083d	Z+NQr9J7iSRi	V01kYVwG13GU
Perry	38	c32f8d5d66a1	49521d4f028e	0nPCg1bAxh8R	V01kYVwG13GU
Tom	40	5f2041a58089	56c26c25e4d	UD82Pv8uGNi7	rXhFoilgAFoO

that use the decryptRND UDFs, where decryptRND is a user defined function implementing decryption which is discussed in sub-section 4.1.

Then we execute the query Q_1 that process the initial search for nodes where the path required to resolve the original query Q may start:

```
MATCH (node1:person)-[:KNOWS]->(node2:person)
WHERE node1.name = "UD82Pv8uGNi7"
RETURN node2 AS output
```

Here the output variable is used as an alias for the result column name of Q_1, and UD82Pv8uGNi7 is the encryption of Tom. As a result of the first stage of the query resolution, there are three nodes as the outgoing of the n.name = "UD82Pv8uGNi7" node.

Before processing the second part of the query Q, WHERE y.age = "22", we need to lower the encryption level of the age property of nodes in the output variable ONLY to the DET layer.

Then we execute the query Q_2, implementing the next step of Q execution:

```
MATCH (n: person)-[:KNOWS]->(output)
WHERE output.age = "NMtqlsMp8Qaf"
RETURN output
```

Lastly, Proxy receives the encrypted result of the above implementation {j39IjDVyx/+q,NMtqlsMp8Qaf} , decrypts the result and sends the decrypted result back {Smith,22} to the application. This solution improves on previous methods by not decrypting all age properties at the DET layer, but only decrypting what the query resolution requires.

Bounded traversal In order to investigate how the traversal-aware adjustment works with a specific variable length path, we return to the database example that is presented in Figure 2, a variable length path of between 1 and 3 relationships from node1 to node2 is considered below. For example, if we assume a query Q:

```
MATCH (node1)-[*1..3]->(node2)
WHERE node1.name = ’Smith’ AND node2.age = ’38’
RETURN node2
```

At the start point all values are held in the `RND` layer. We then move values to the `DET` layer using the function `UPDATE Label SET P = DECRYPT RND`, where `P` corresponds to `name`. Thereafter, we perform the query Q_1 processing the initial search for nodes when the path required in the original query `Q` starts as:

```
MATCH (node1)-[*1..3]->(node2)
WHERE node1.name = "j39IjDVyx/+"
RETURN node2 AS output
```

Again, the output variable is used as an alias for the result column name of Q_1, `j39IjDVyx/+` corresponds to the encryption of `Smith`. Further execution of Q_1 shows that there are six nodes as the outgoing of `node1.name = "j39IjDVyx/+"` condition. Before processing the second part of the query `Q`, `WHERE node2.age = "38"`, we need to lower the encryption level of the age property of nodes in the `output` variable to the `DET` layer.

Next we execute the query Q_2, implementing the next step of `Q` execution:

```
MATCH (node1)-[*1..3]->(output)
WHERE output.age = ’V01kYVwG13GU’
RETURN output
```

Where `V01kYVwG13GU` is the encryption of "38". Finally, Proxy receives the encrypted result of the above implementation `{Z+NQr9J7iSRi,V01kYVwG13GU}`, and sends the decrypted result `{Sara,38}` back to the application.

Unbounded traversal Now, we need to see the affect when the path length between nodes is unbounded; when the variable path length of any number of relationships from node1 to node2 is unlimited. With reference to the example graph in Figure 2, assume the following query `Q`:

```
MATCH (node1)-[*]->(node2)
WHERE node1.name = ’Smith’ AND node2.age = ’38’
RETURN node2
```

To resolve the query the `DET` layer for name is required. We process the query Q_1 to allow the initial search for nodes to be executed, when the path required in the original query `Q` starts as:

```
MATCH (node1)-[*]->(node2)
WHERE node1.name = "j39IjDVyx/+"
RETURN node2 AS output
```

At this stage the output variable is used as an alias for the result column of Q_1, j39IjDVyx/+ corresponds to the encryption of Smith. Further execution of Q_1 indicates that there are seven nodes in output using the filter node1.name = "j39IjDVyx/+".

As soon as the lowering of the encryption level of the age property of the nodes in the output variable to the DET layer has been done, we can process the second part of the query Q, which is WHERE node2.age = "38". Next we execute Q_2, to implement the next step of Q:

```
MATCH (node1)-[*]->(output)
WHERE output.age = 'V01kYVwG13GU'
RETURN node2
```

Where V01kYVwG13GU is the encryption of "38". The Proxy receives the encrypted results from the previous implementation: {Z+NQr9J7iSRi,V01kYVwG13GU} and {0nPCg1bAxh8R,V01kYVwG13GU} ; and decrypts the results {Sara,38} and {Perry,38} and returns them to the user.

5 Experiments and Performance Analysis

In this section we report on the results of experiments conducted to show the validity of the approach and estimate the performance.

5.1 Datasets

In order to study the traversal-aware encryption adjustment concept a variety of datasets have been used. A total of five Neo4j databases were constructed (Table 3). Each database consists of a number of nodes and edges, and each node has a different number of properties, as well as relationships.

For the case study we consider a particular graph database instance. In this example scenario we have nodes with the label Person, and group of properties of interest: name, age, and gender. Graph datasets were created to contain approximately 10, 100, 500, 1000, and 10000 nodes to aid in assessing execution time of queries over non-encrypted data and encrypted data.

The system used for testing ran on Windows, version 10. It has an Intel Core 2 Duo CPU running at 3.40 GHz and has 16 GB of RAM. The benchmarking program was the only application running when the results were created, but the machine was connected to the Internet and standard system processes were running.

5.2 Queries

To test the approach, we executed the queries Q_1 – Q_5 below over non-encrypted data and over encrypted data using traversal-aware encryption adjustment where applicable. We refer to encrypted versions of the queries as Q'_1 – Q'_5. This particular set of queries was selected to test some commonly used in graph databases queries.

Q_1: Find all orphan nodes (no incoming edges and no outgoing edges).

```
MATCH (node)
WHERE not((node)-[ ]-())
RETURN node
```

Q_2: Basic Relationships Matching

```
MATCH (node1)-[:KNOWS]->(node2)
WHERE node1.name = ’Tom’ AND node2.age = ’22’
RETURN node2.name, node2.age
```

Q_3: Adding Relationship Length

```
MATCH (node1)-[:KNOWS]->(node2)-[:KNOWS]->(node3)
WHERE node1.name = ’Jones’ AND node2.age = ’47’ AND
node3.gender = ’Female’
RETURN node3.name, node3.age, node3.gender
```

Q_4: Variable Relationship Length

```
MATCH (node1)-[:KNOWS*1..3]->(node2)
WHERE node1.name = ’Jones’ AND node2.age = ’38’
RETURN node2.name, node2.age
```

Q_5: Infinite Length and Length Limit

```
MATCH (node1)-[:KNOWS*]->(node2)
WHERE node1.name = ’Jones’ AND node2.age = ’38’
RETURN node2.name, node2.age
```

Notice that execution of each of Q'2 – Q'5, requires the execution of several queries/updates (unlike the single query execution of non-encrypted versions). In order to make a fair comparison we composed query/update parts of Q'_i by using `WITH` clauses. Having `WITH` enabled the query parts to be chained together, passing the outputs from one to be used as starting points or criteria in the next. As in these queries the first condition is `WHERE node1.name=’value’` we need to adjust the encryption level of `name` to the `DET` layer to allow equality checking. Take for example, Q'_2, as follows:

```
(1) MATCH (node1)-[:KNOWS]->(node2)
(2) WHERE node1.name=decryptRND(encryptRND
(encryptDET(’Tom’), ID(node1)),ID(node1))}
(3) SET node2.age= decryptRND(node2.age,ID(node2))
(4) with node2
(5) MATCH (node1:Person)-[:KNOWS]->(node2)
(6) WHERE node2.age = decryptRND
(encryptRND(encryptDET(’22’), ID(node2)),ID(node2))
(7) return decryptDET(node2.name), decryptDET(node2.age)
```

In step (1), we have a `MATCH` clause to determine the direction of the relationship and its depth. In step (2), as all values are held in the RND layer, we need to decrypt the name property within the DET layer, in order to allow the equality checking. In step (3), we lower the encryption of the age property for nodes in the previous step. In step (4), by using WITH we can pass the previous result so that it becomes the starting criteria to the next part of the query. In step (5) we determine the direction of the relationship. In step (6) we implement the second condition. In step (7) we return the result in plain text format.

5.3 Results

Each query was run over all five databases and execution times (in milliseconds) for non-encrypted data and encrypted data was collected, as presented in Table 3.

The queries Q_1 and Q'_1, to find orphan nodes, resulted in a similar result for both the non-encrypted and encrypted databases. Those nodes were iterated through, checking each node for the presence of edges. For the queries Q_2 - Q_5 the execution time was clearly faster, this was expected since the queries over non-encrypted databases do not require any encryption layer adjustment.

Table 3. Query results using different graph database sizes (milliseconds).

Database	No. of Nodes	No. of Relationships	Q1	Q2	Q3	Q4	Q5	Q'1	Q'2	Q'3	Q'4	Q'5
DB1	10	9	1	2	2	4	3	1	5	6	9	7
DB2	100	82	1	4	4	3	4	2	20	20	18	15
DB3	500	410	4	2	2	2	4	4	36	32	35	34
DB4	1000	820	4	2	2	2	4	4	63	61	58	55
DB5	10000	8200	4	2	2	2	4	15	555	504	505	655

From an overall perspective, the retrieval time for non-encrypted databases is small and roughly similar for all datasets. For the encrypted case, the execution time clearly grows with the size of the database but remains in a practically feasible range (under a second) for the largest considered dataset.

6 Conclusion

In this paper, we reported on the implementation and evaluation of traversal- aware encryption adjustment mechanism for querying encrypted data in graph databases. The method provides better security protection against server-side attacks while keeping good implement ability and reasonable performance of query execution.

Security The considered case studies have shown the trade-off between simple and traversal-aware encryption adjustment policies. The simple policy requires less queries and updates to be followed, on the other hand, the traversal-aware policy provides better security, as it reveals less information to a possible server- side attacker. With the latter policy, as observed above not all age property values were adjusted to the DET layer, just those required to allow the query execution to progress.

Performance We report on experiments and performance of the proposed schema in the Appendix. To evaluate the proposed mechanism a collection of five databases was created, the proposed approach was tested using five types of Cypher queries. The evaluation was conducted by doing experiments that measure the execution time for a set of queries directed at both non-encrypted data and encrypted data with different dataset sizes. Our results are encouraging, but still, need to be validated using larger data sets.

Implementability Similar to the methods in [2, 3, 9] and unlike the methods in [6, 13] the proposed mechanism does not need to change the inner structure of the DBMS because it is implemented as a set of layers above the DBMS. In particular, the proposed approach is compatible with a concurrency control for multi-user DBMS, but related security aspects and performance evaluation in multi-user environment need to be addressed in future work.

References

1. A. M. Abdullah: Advanced Encryption Standard (AES) Algorithm to Encrypt and Decrypt Data. In: Cryptography and Network Security, pp. 1–12. Proceedings, Lo- cation (2017)
2. N. Aburawi, F. Coenen, and A. Lisitsa: Traversal-Aware Encryption Adjustment for Graph Databases. In: 7th International Conference on Data Science, Technology and Applications, pp. 381–387. Proceedings, Portugal (2018)
3. N. Aburawi, A. Lisitsa, and F. Coenen: Querying Encrypted Graph Databases. In: 4th International Conference on Information Systems Security and Privacy, pp. 447–451. Proceedings, Portugal (2018)
4. S. Ali, A. Rauf, and S. Mahfooz: UPDATE query over encrypted data. In: International Conference on Computer Networks and Information Technology, pp. 279–282. Proceedings, Pakistan (2011)
5. J. Al-Saraireh: An Efficient Approach for Query Processing Over Encrypted Database. Journal of Computer Science 13(10), 548–557 (2017)
6. M. Chase and S. Kamara: Structured Encryption and Controlled Disclosure, in Proc. of Advances in Cryptology - ASIACRYPT 2010, 577–594, 2010
7. N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor: Cypher: An Evolving Query Language for Property Graphs. In: 18th SIGMOD International Conference on Management of Data, pp. 1433–1445. Proceedings, USA (2018)
8. L. Liu and J. Gai: A Method of Query over Encrypted Data in Database". In: International Conference on Computer Engineering and Technology, pp. 23–27. Pro- ceedings, China (2009)
9. R. A. Popa and C.M.S. Redfield, N. Zeldovich and H. Balakrishnan: CryptDB: Protecting Confidentiality with Encrypted Query Processing. In: 23rd ACM Symposium on Operating Systems Principles, pp. 85–100. Proceedings, Portugal (2011)
10. R. L. Rivest, A. T. Sherman: Randomized Encryption Techniques. In: D. Chaum (ed.) Advances in Cryptology, pp 145–163. Springer US, Boston, MA (1983)
11. I. Robinson , J. Webber , and E. Eifrem: Graph Databases. 1st edn. OReilly Media, Inc., United States of America (2013)
12. A. Turu Pi, O. Koroglu, and E. Zimanyi: Graph Databases and Neo4J. University libre de Bruxelles (2017)
13. P. Xie and E. Xing : CryptGraph: Privacy Preserving Graph Analytics on En- crypted Graph. In: CoRR journal, volume: abs/1409.5021, archivePrefix = arXiv, (2014)
14. Neo4j, Inc. (2019) Neo4j Homepage. https://neo4j.com Accessed 20 Feb 2019
15. Neo4j, Inc. (2019) User Defined Procedures and Functions. https://neo4j.com/developer/procedures-functions/ Accessed 18 April 2019
16. openCypher (2019) openCypher Resources. https://www.opencypher.org/resources Accessed 13 June 2019
17. C. Willemsen and M. Bachman (2019) Cypher: Variable Length Relationships by Example. https://graphaware.com/graphaware/2015/05/19/neo4j-cypher-variable- length-relationships-by-example.html Accessed 15 June 2019