SMPG: Secure Multi Party Computation on Graph Databases

Nouf Aljuaid[1,2], Alexei Lisitsa[2] and Sven Schewe[2]
[1]*Department of Information Technology, Taif University, Saudi Arabia*
[2]*Department of Computer Science, University of Liverpool, Liverpool, U.K.*

Keywords: Graph Databases, Secure Multi-party Computations, Multi-party Querying, Federated Databases, Secure Data Processing.

Abstract: In this position paper, we outline how secure multi-party querying can be brought to graph databases. Such a system will allow multiple users to jointly query federated graph databases that consist of several private parts. We have provided a proof of concept. Our prototype implementation for the SMPG system builds on top of Conclave (Volgushev et al., 2019), which was originally proposed and implemented for multi-party computation and querying on relational databases. We describe the templates of queries that are currently supported by our prototype and discuss current limitations as well as the extensions planned to tap the conceptual benefits.

1 INTRODUCTION

As the relevance of data security does not need much discussion, it is only natural that many techniques have been proposed and developed to increase the security of data, see (Mostafa, 2016) for a survey. Multi-party computation (MPC) is a particularly intriguing example among them; (Cramer et al., 2015) describe it as cryptographic technique that enables a group of people to join together to perform computation without revealing any private data. It differs from standard cryptographic techniques in that it does not focus on the encryption of data or databases, but rather develops protocols that coordinate the analysis of distributed data without joining the data itself. The liberation from the need to share data provides a major benefit that is rather unique to MPC: the opportunity to cooperate and coordinate operations, where collaboration was impeded by a lack of trust (Hemenway et al., 2014). Applications where MPC can solve such trust issues include secure elections (Alwen et al., 2015), auctions (Aly and Van Vyve, 2016), and secret sharing (Evans et al., 2018). Another application of MPC is to secure the query on different types of databases. MPC techniques are currently supported by Conclave (Volgushev et al., 2019) and GOOSE (Ciucanu and Lafourcade, 2020).

Thus far, MPC mainly has been used in to secure relational databases. For example, Conclave uses MPC for securing the queries on a federated database that consists of private individual databases held by multiple parties, where it executes a single query. This raises the question of whether MPC queries are restricted to relational databases, or can transcend the database type. We look in particular at the scope to apply MPC in graph databases, a type of NoSQL database. Graph databases have been created to address the limitations of relational databases (Salehnia, 2017), and found multiple application that particularly benefit from the graph paradigm, such as Instagram, Twitter, and Facebook (Ciucanu and Lafourcade, 2020). These applications benefit from explicitly defined connections that are the thumbmark of graph databases, as opposed to relational databases, where the data is linked implicitly. This guarantees scalability for graph databases where relational databases face scalability and complexity barriers, which makes it cheaper to maintain servers for graph databases (Salehnia, 2017).

While MPC is a useful technique for executing secure queries, its use in graph databases is still in its infancies. In (Ciucanu and Lafourcade, 2020), the use of MPC as the backend for supporting queries on a graph database has been studied. The authors of (Ciucanu and Lafourcade, 2020) suggest distributing a single database over several components that are then handled with the help of MPC, but they did not address multi-party queries at the user level. In this paper, we propose the design of a system that combines the advantages of the MPC technique with those

Aljuaid, N., Lisitsa, A. and Schewe, S.
SMPG: Secure Multi Party Computation on Graph Databases.
DOI: 10.5220/0010876200003120
In *Proceedings of the 8th International Conference on Information Systems Security and Privacy (ICISSP 2022)*, pages 463-471
ISBN: 978-989-758-553-1; ISSN: 2184-4356

offered by graph databases for securing multi-party querying over a graph database.

The remainder of this paper is organised as follows. The following section presents the proposed system for secure multi party computation on graph databases (SMPG). Before presenting the implementation of the prototype in Section 4, we discuss the functionality of the existing Conclave system (Volgushev et al., 2019) we use in our prototype implementation in Section 3. A literature review of existing implementations of MPC and the related work on securing the query on a different type of data model is provided in Section 5. We then close with drawing conclusions and discussing future work in the final section.

2 SMPG ARCHITECTURE

This section provides an overview of the proposed SMPG scheme, introducing its entities and outlining how it works.

2.1 SMPG Entities

The architecture of our SMPG system is illustrated in Figure 1. The system contains three entities, and these entities have different trust levels in the use cases we are interested in.

- Data Owners make their graph database available for SMPG to perform a joint multi-party once the query is agreed and may allow a single user to perform a query using their data. Full trust by the other data owners is required as they need to jointly conduct the query.
- A user may submit a query using the data from Data Owners to get results for a query without sharing any data with the system. She does not gain access to data beyond the answer to the query and can be considered un-trusted.
- A Proxy Server is responsible for authenticating the user to access the system. Proxy Servers are (only) trusted to this end (semi-trusted).

2.2 Two Use Cases for SMPG Systems

The entities are described for the use case of a **single user and multiple data owners**. In this situation, a single user who does not contribute a database that partakes in the query wants to compute something that uses the databases from the data owners. In this case, the user needs to be authenticated to access the system and submit her query. After she gets the permission,

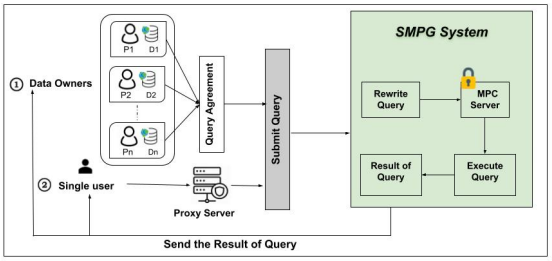

Figure 1: SMPG workflow.

she can submit her query to the SMPG, who then executes a joint query.

In the **joint query**, the data owners, named $p_1, p_2, \ldots, p_n$, who own different graph databases, execute a single query jointly. In more detail, after all parties agree on a joint query, they submit it to the system. Inside the system, the first step is to re-write the query to make it amenable to the execution on the MPC backend. In this stage, the system takes all unnecessary operations to run outside the MPC backend. In our proof-of-concept, we use Conclave (Volgushev et al., 2019) as our backend, which in turn uses Jiff (Albab et al., 2019) for MPC queries. Conceptually, they can be exchanged by other solutions (which then, naturally, need adjusted translations). After receiving the result of the query, the system will send it to the single user who initiated the query. In an alternative scenario, the **multiple data owners** initiate the **joint query** themselves. The translation of the query is then similar, but there is no need to return the result to an external user.

2.3 DB Query Language

While the SMPG scheme can be implemented for any type of graph-based DBs, we have selected the Neo4j environment as our starting point. Neo4j is a popular graph database (López and De La Cruz, 2015), whose graph data model can be represented by a set of nodes, which represent data, and arrows that represent the relationship between them (Miller, 2013).

In order to deal with the data in such a graph, Neo4j uses a query language called Cypher. In our SMPG system, we intend to use a fragment of the Cypher query language, extended with federated query facilities/functionality (Neo4j Fabric) with MPC protocols. The operational principle of Neo4j fabric offers a way to issue Cypher queries that target more than one Neo4j graph database at once.

2.4 Workflow Overview

We first assume that we have a single user, who does not contribute an own database that partakes in the

query and wants to query the federated database from the data owner. Our system will process such a query in eight stages. Figure 2 shows The workflow of the SMPG system when used by an outside user (not a data owner).

1. **Submit Cypher Query:** after the user is authenticated by the system, she can submit her query to the coordinator of the SMPG system through the SMPG Coordinator.
2. **Query Translation:** the Coordinator translates the query into individual queries to the partaking data owners (participating parties) and submits these queries to them.
3. **Process Query:** each participating party processes the query it receives. This includes creating a configuration JSON file.
4. **Query Execution:** each participating party then invokes the MPC Conclave system[1] to execute the workflow specified in the configuration JSON file on its Neo4j DB.
5. **Computation using MPC:** in this stage, the information will be passed to a Jiff server, who provides answers to the individual queries using an MPC protocol.
6. **Send the Result to Parties:** in this stage, the JIFF server will send back the result for the parties involved in the query.
7. **Send the Result to Coordinator:** the overall result of the original query will then be sent to the coordinator, in our implementation as a CSV file.
8. **Final Result:** the SMPG coordinator finally assembles the result and returns it to the user.

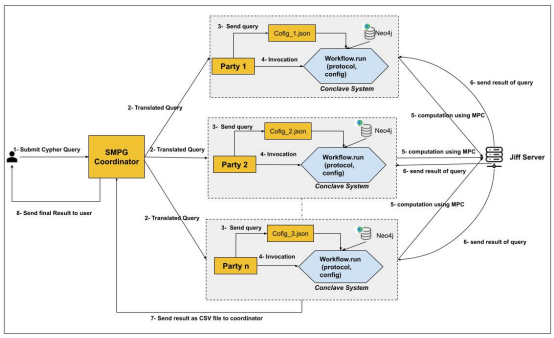

Figure 2: The workflow of the SMPG system when used by an outside user (not a data owner).

In the alternative scenario, where the query is initiated by the data owners, the last two steps can be omitted, as the partaking data owners are already in possession of the result. This leads to a situation where, while the query is assembled by an SMPG Coordinator, this coordinator never receives an answer to the query (cf. Figure 3).

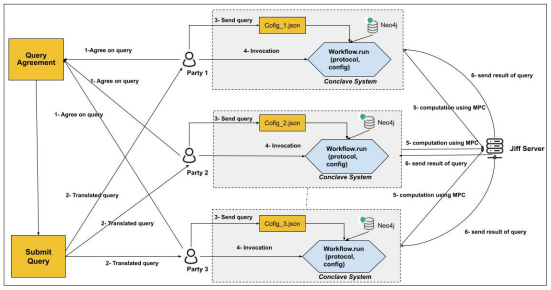

Figure 3: The workflow of the SMPG system with a joint query by the involved data owners.

3 CONCLAVE FUNCTIONALITY

In this section, we describe the functionality of the Conclave system (Volgushev et al., 2019), which uses MPC with a relational database; we use it as a basis to demonstrate that the concept extends to support Neo4j, including a fragment of the Cypher query language.

The Conclave system supports table schema definitions, relational operators (join, aggregate, project, and filter) as well as enumeration, arithmetic on columns and scalars. Inside the system, after the originally submitted query has been re-written, each query is represented as a node within a directed acyclic graph (DAG). As such, each operation on a data set should be stored in a variable as follows:

```
<node> = cc.<query>(<arg1>,<arg2>,..., <argn>)
```

For the aggregate operators, Conclave supports count operation, sum, and mean. Inside the system, the query is written using Language Integrated Query (LINQ) (Bai, 2010). This is a data querying API that provides general-purpose query capabilities to the .NET programming languages with a syntax similar to SQL. The syntax of the aggregate query using LINQ will be:

```
aggregate(input_node, output_name,
group_column_names,aggregated_column_name,
aggregator, output_column_name)
```

The project operator is used to select a list of columns from a data set. A query using the project operator could be:

```
project(input_node, output_name,
selected_column_names)
```

[1] or any alternative backend to process MPC queries

The Join operator is a function that joins two data sets over a list of columns. A query using join could look like this:

```
join(left_node, right_node, output_name,
left_column_names, right_column_names)
```

A filter is a function to filter data in a column with a specified value. A query using filter will be like:

```
filter_by(input_node, output_name,
filter_col_name, by_op)
```

4 PROTOTYPE IMPLEMENTATION

In this section, we describe the implementation of a prototype of a fragment of the above SMPG design, using a Python API. The main objective of this prototype is to prove the effectiveness of a query over a graph database using MPC. Our prototype can use multiple graph databases from multiple data owners and applies MPC to provide a joint query.

Initially, different Neo4j databases are built. Then, for each party, the user has to edit the config file manually to connect to Neo4j. In future implementations, editing the config file will be automatic, relieving the user from the need to edit it manually. Currently, the prototype supports a small fragment of the Cypher query language. In this fragment, we use the `UNWIND`, `CALL`, and `USE` clauses of Cypher to handle querying multiple graph databases. `UNWIND` supports iteration of queries over a list of databases. `CALL` is used to execute a Cypher subquery by each party involved in the joint query. To handle a particular graph database, we utilise the ”USE” clause. Only queries that are instances of the query templates Q1-Q4 discussed below are fully supported, while Q5 is conceptually supported, but needs an adjustment of the backend used (or a workaround, like the mapping of character sequences into the integers, which is possible in principle).

4.1 Query Templates

In the SMPG system, the user submits the query using Cypher query language and this query is used to derive sub-queries depending on the number of parties that want to find the result of this query.The subqueries are then run separately for each party. As a first prototype of our system, we built it on top of the Conclave system. The cypher query from the input is translated into the LINQ language. In the following example queries, we explain the path of executing the query with the implemented prototype.

- **Q1: Aggregation Count**

In this query, the count for node 1 may depend on node 2. The corresponding Cypher query template is:

```
UNWIND example.graphIds() AS graphId
CALL {
USE example.graph(graphId)
MATCH (node1: label1)-[:Relationship]
->(node2:label2)
WITH node2, count(node1) as cnt
RETURN node2, cnt
}
RETURN node2, cnt
```

Running the query will first collect the graph IDs for all remote graphs, then using `UNWIND` to get one record per graph ID. The `CALL` sub-query will be executed separately for each party. As example to use this query, suppose that we have three parties, each equipped with a Neo4j database with the same properties. We want to execute the following query:

```
UNWIND example.graphIds() AS graphId
CALL {
USE example.graph(graphId)
MATCH (c:CAR) -[rel:IN]-> (L:LOCATION)
WITH L, count(c) AS cnt
RETURN L.id AS location_id, cnt
}
RETURN location_id, cnt
```

The return from this query will provide the number (or: count) of cars for each location for all parties involved in the query. The sub-query inside the `CALL` will be executed separately for each party in the list `example` iterated over by `UNWIND` function. The result of this sub-query will be the number of cars for the party. The results from the individual parties are then combined and mapped to the aggregation_count function, which sums up the individual results without exposing the private data of the individual parties. This sub-query is translated to LINQ as:

```
agged = aggregate_count(combined,"heatmap",
["location"],"by_location")
```

In this example query, this will return the overall number of cars grouped by location.

- **Q2: Aggregation Sum**

This query will return the sum for any node in the query. The cypher query is like be:

```
UNWIND example.graphIds() AS graphId
CALL {
USE example.graph(graphId)
MATCH (node1: label1)-[:Relationship]
->(node2:label2)
RETURN sum(node2) as Sum
}
RETURN Sum
```

As an example for this query, suppose that we have two parties that both hold information about scores of students from different courses. We want to find the sum of these scores for a given student on both courses. To do that, we can execute this query:

```
UNWIND example.graphIds() AS graphId
CALL {
 USE example.graph(graphId)
 MATCH (s:student) -[rel:get]-> (C:course)
 RETURN sum(C.score) AS Score
 }
 RETURN Score
```

In SMPG, this query will first gather the graph IDs for all parties involved in the query, and execute the sub-query separately for each party. Then, the individual results of these sub-queries are combined, and their aggregation, as a sum for all individual results, is calculated without exposing the private data of the partaking parties. This aggregation is translated to LINQ as:

```
agged = aggregate(combined,"agged",
["student_id"],"score","sum","score")
```

- **Q3: Project Data**

This query will match and return particular data from node1 or node2. A cypher query would look like this:

```
UNWIND example.graphIds() AS graphId
CALL{
  USE example.graph(graphId)
  MATCH (node1: label1)-[:Relationship]
  ->(node2:label2)
  RETURN node2
  }
RETURN node 2
```

As an example, suppose that we have two parties, and each one of them has a Neo4j database with the same properties. We want to find the scores for all students in both databases in some particular course, without exposing any information about the individual students. To this end, we can execute this query:

```
UNWIND example.graphIds() AS graphId
CALL{
  USE example.graph(graphId)
  MATCH (s:student) -[rel:get]-> (C:course)
  With s.id
  RETURN c.score AS score
  }
RETURN score
```

This query will return the score for all students for each database separately. After that, the results from each user are combined, and the projection returns the complete row of scores from the combined data, without return anything else. To execute this query, it is translated to the LINQ language as:

```
agged = project(combined,"heatmap",
["scores"],"by_scores")
```

- **Q4: Union Two Databases**

We consider as an example that we want to find the list of names of professors, whose students have a grade ≥ 9.0 in either database. Such a query could look like this:

```
CALL{
USE Graph 1
MATCH (P:Professor)-[:guide]->(S:Student)
WHERE S.grade >= 9.0
RETURN P.name as prof_name, S.name as student
}
CALL{
USE Graph 2
MATCH (P:Professor)-[:guide]->(S:Student)
WHERE S.grade >= 9.0
RETURN P.name as prof_name, S.name as student
}
RETURN prof_name, student
```

To execute this query using our system, a query will be translated to a LINQ query as follows:

```
agged = join(in_one,in_two,"heatmap",
"P.name","s_name","P.name","s_name")
```

This query will generate a CSV file, which includes the information that results from the cypher query.

- **Q5: Match Intersection**

This query will calculate the intersection between two (or more) nodes, here node 1 and node 2. The cypher query could look like this:

```
UNWIND example.graphIds() AS graphId
CALL {
  USE example.graph(graphId)
  MATCH (node1: label1)-[:Relationship]
  ->(node2:label2)
  RETURN node2
}
RETURN node 2
```

Assume that we have two different Neo4j databases that each contain a list of people and the names of their friends. We want to determine whether or not there are common friends between them. To do this, we can run the following query:

```
UNWIND example.graphIds() AS graphId
CALL {
   MATCH (n:Person) -[:Friend]-> (m:Person)
  RETURN m.name As Name
  }
RETURN Name
```

After running this query for each party involved, combining the results to determine whether or not there is a common friend between them will be obtained through an MPC protocol. Due to the limitation of the Conclave system to numerical data—not

supporting strings—we will try to extend their system to overcome this shortfall in future.

- **Q6: Correlated Subquery**

Assume that we have two different Neo4j databases (Graph A and Graph B) and we want to query information from Graph B depending on particular data from Graph A, using the following query:

```
CALL {
  USE graphA
  MATCH (node1:label1)
  RETURN max(label1.property) AS MaxValue
}
CALL {
  USE graphB
  WITH MaxValue
  MATCH (node1:label1)
  WHERE label1.property = MaxValue
  RETURN node 1
}
RETURN node1
```

As an example for this type of query, assume that Graph A contains American movies and Graph B contains European movies. We want to find all European movies released in the same year as the latest released American movie. To find this, we can execute the following query:

```
CALL {
  USE graphA
  MATCH (movie:Movie)
  RETURN max(movie.released) AS usLatest
}
CALL {
  USE graphB
  WITH usLatest
  MATCH (movie:Movie)
  WHERE movie.released = usLatest
  RETURN movie
}
RETURN movie
```

This query would first be translated into two subqueries, the first one using Graph A.

```
USE graphA
MATCH (movie:Movie)
RETURN max(movie.released) AS usLatest
```

This query will find all movies that were released in the same year as the latest movie release in the US. A query would look like this:

```
USE graphB
WITH usLatest
MATCH (movie:Movie)
WHERE movie.released = usLatest
RETURN movie
```

This type of query cannot be supported by Conclave due to its limitations. In future, we plan to extend our work to support such queries and, more generally, federated privacy-preserving queries using a combination of Neo4j Fabric with a tailor-made MPC backend.

4.2 Experimental Results

We have run the queries from this section using our implemented prototype SMPG, using three different Neo4j databases from three parties. The total amount of nodes for all the databases is 26 nodes with 13 relationships between nodes. In more detail, database one from the first party involved in the query has 12 nodes with 6 relationships, whereas the second database has 8 nodes with 4 relationships. The third database contains 6 nodes with 3 relationships. We have measured the execution time where all the parties get the results of the query using the *time* function. This function is supported by Python, which returns the query in 66 to 76 seconds for Q1 through Q4.

While this seems much, it is based on translations to relational databases and to Conclave, which does not tap the advantages of graph databases that the concept will—once underpinned with an independent MPC implementation—provide.

It is, however, quite easy to cut the time needed even in the current setting. For example, cutting the sorting that is included by default in Conclave improves the running time by an order of magnitude, keeping all running times below 8 seconds.

5 RELATED WORK

5.1 Implementations of MPC

While MPC was mostly the object of theoretical research unitl recently, there has lately been a considerable effor to bring MPC to real-life applications (Evans et al., 2018). MPC implementations are, for example, used in Jiff (Albab et al., 2019), ObliVM (Liu et al., 2015), and GraphSC (Nayak et al., 2015).

Jiff (Albab et al., 2019) is a JavaScript library that implements MPC. It can perform secure computation on data distributed between several parties. Jiff utilises a server to store and route encrypted messages that are sent between the different participating parties. It assumes the honest-but-curious security model and uses threshold Shamir's Secret Sharing (Shamir, 1979). It is compatible with various browsers and can run as a Node.js application. When Jiff is run as a server, then each parity runs as a client, and after completion of the computation, the result will show for all of the parties without exposing their data that has flown into the computation of the result.

Table 1: MPC for data processing.

Framework	*Parties supported*	*MPC*	*Framework backend*	*Trust Party*	*No.Data owners*	*Data Model*	*Query language/API*	*Available implementation*	*Development language*
Conclave (Volgushev et al., 2019)	>= 2	Secret Sharing	Jiff	Yes	>= 2	Relational DB	SQL/LINQ	Yes	Python
Congregation	>= 2	Secret Sharing	Jiff	No	>= 2	Relational DB	SQL	Yes	Python
SMCQL (Bater et al., 2016)	2	Garbled Circuits/ ORAM	ObliVM	No	2	Relational DB	SQL	Yes	Java
Senate (Poddar et al., 2020)	2	Garbled Circuits	N/A	No	2	Relational DB	SQL	No	-
SAQE (Bater et al., 2020)	2	Garbled Circuits	ObliVM	No	2	Relational DB	SQL	No	-
Shrinkwarp (Bater et al., 2018)	2	Garbled Circuits/ ORAM	ObliVM	No	2	Relational DB	SQL	No	-
Secrecy (Liagouris et al., 2021)	3	Repl.Secret Sharing	-	No	3	Relational DB	SQL	No	C
SDB (Wong et al., 2014) [2]	N/A	Secret Sharing	N/A	No	1	Relational DB	SQL	No	-
GOOSE (Ciucanu and Lafourcade, 2020) [3]	N/A	Secret Sharing	N/A	No	1	GraphDB	SPARQL	Yes	Python
SMPG	>= 2	Secret Sharing	Jiff	No	>= 2	GraphDB	Cypher		Python

*Both [2] and [3] are using MPC as backend over a database; they do not support multi-party user queries

ObliVM (Liu et al., 2015) is a programming framework for secure computation. It compiles a Java-like language called ObliVM-lang and executes a two-party garbled circuit (Evans et al., 2018). The authors of (Liu et al., 2015) show that it is possible to easily and accurately perform complicated arithmetic operations.

The authors of (Nayak et al., 2015) suggest a secure computation framework that supports parallel secure computation. For MPC, they used the ObliVM (Liu et al., 2015) protocol. To evaluate their design's performance, the authors of (Nayak et al., 2015) use four classic data analysis algorithms: a histogram, page-rank, and two versions of matrix factorisation.

5.2 MPC for Data Processing

There are attempts to use MPC with databases for data protection. For example, (Volgushev et al., 2019) proposes Conclave, a query compiler applied to a relational database. It works by transforming the queries into a combination of data-parallel, local cleartext processing, in conjunction with small MPC steps. In this system, the queries are rewritten to minimise expensive processing under MPC in order to improve scalability. They suggested to pass the rewritten query to Jiff (Albab et al., 2019), which is used as a backend MPC system. Furthermore, (Poddar et al., 2020) present the Senate system that allows multiple parties to run analytical SQL queries together, without exposing their private data to each other. The additional advantage of their system over prior work is that it provides security against malicious parties, while older systems adopted a semi-honest model. Moreover, (Liagouris et al., 2021) present a relational MPC framework based on replicated secret sharing called Secrecy. The central idea of this system is to split data into three shares, s1, s2, and s3, where each party will take two of the shares and perform a part of the code to execute the query.

The authors of (Bater et al., 2016) suggest a system called SMCQL that translates SQL queries into secure multi-party computation. The user submits her query to an honest broker, who is considered a trusted third party. An honest broker is responsible for translating the query to a secure cluster and returning the result to the user. A further study of the SMCQL system (Bater et al., 2020) adopts SMCQL and builds a system called SAQE to protect the SQL query on top of it. The query in this system is processed in two parts: a query planning and a query execution part. The client side performs the query plan and optimisation, while the execution of the query happens on the server-side amongst the data owners, using MPC. They jointly execute queries over their databases and return the result to the client. Likewise, Bater et al. build on top of SMCQL (Bater et al., 2018). They use two-party secure computations, which means that they run their experiments with two data owners. While they report an improved performance of SMCQL, this comes at the cost of leaking some information during the process.

On the systems suggested in (He et al., 2015) and (Ciucanu and Lafourcade, 2020), the authors examine using MPC for the implementation of querying by a single party. The SDB system in (He et al., 2015) is a cloud database system on relational tables. It contains two parties: Data owner (DO) and Server provider (SP). Each sensitive data item is split into two shares, one kept at the DO, referred to as the item key, and another at the SP, which is regarded as ciphertext. This system uses MPC and secret sharing between the DO and the SP. After accepting an SQL query from the user, the SDB proxy in the DO part rewrites the query that involves sensitive columns to their corresponding UDFs at SP. It then submits rewritten queries to the SP and sends the encrypted result back to the SDB proxy to decrypt before sending it to the user. Likewise, the GOOSE framework in (Ciucanu and Lafourcade, 2020) is a system to secure the outsourcing of data in the RDF graph database using MPC secret sharing. Here, the graph data is submitted by the data owner to the cloud in a specific form: it is chopped into three different parts, and uploaded in an encrypted form to

various places in the cloud. All these parts are considered multi-party, and each one cannot know the whole graph, or a query, or its result. Moreover, all messages between them are encrypted using the AES algorithm. GOOSE has proven to scale via a large-scale experimental study using a standard query evaluation.

While this previous work has addressed the use of MPC in relational databases and graph databases, multi-party queries over graph databases are novel. Table 1 shows a comparison between all the previous systems and our suggested system, SMPG.

6 CONCLUSIONS

In this position paper, we have proposed SMPG, a system to secure multi-party computation on graph databases. The concept of SMPG is to use MPC protocols to execute queries over graph databases. We have implemented a prototype over the Conclave system to demonstrate the effectiveness of a query over a graph database using MPC. In future work, we will extend this concept to allow for a broader application. These extensions will include 1) an enhancement of the backend system to support operations over string data (where the Conclave system we have used in the proof-of-concept is restricted to numerical data); 2) an extension of the supported fragment of Cypher query language; and 3) the development of a more general system for privacy-preserving federated queries that combine federated query facilities with specialised MPC backends.

REFERENCES

Albab, K. D., Issa, R., Lapets, A., Flockhart, P., Qin, L., and Globus-Harris, I. (2019). Tutorial: Deploying secure multi-party computation on the web using JIFF. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 3–3. IEEE.

Alwen, J., Ostrovsky, R., Zhou, H., and Zikas, V. (2015). Incoercible multi-party computation and universally composable receipt-free voting. In *Advances in Cryptology–CRYPTO 2015: 35th Annual Cryptology Conference,*, volume 9216, pages 763–780. Springer Berlin Heidelberg.

Aly, A. and Van Vyve, M. (2016). Practically efficient secure single-commodity multi-market auctions. In *International Conference on Financial Cryptography and Data Security*, pages 110–129. Springer.

Bai, Y. (2010). *Introduction to Language-Integrated Query (LINQ)*, pages 147–233. Wiley-IEEE Press.

Bater, J., Elliott, G., Eggen, C., Goel, S., Kho, A., and Rogers, J. (2016). SMCQL: secure querying for federated databases. *arXiv preprint arXiv:1606.06808*.

Bater, J., He, X., Ehrich, W., Machanavajjhala, A., and Rogers, J. (2018). Shrinkwrap: efficient sql query processing in differentially private data federations. *Proceedings of the VLDB Endowment*, 12(3):307–320.

Bater, J., Park, Y., He, X., Wang, X., and Rogers, J. (2020). SAQE: practical privacy-preserving approximate query processing for data federations. *Proceedings of the VLDB Endowment*, 13(12):2691–2705.

Ciucanu, R. and Lafourcade, P. (2020). GOOSE: A secure framework for graph outsourcing and sparql evaluation. In *34th Annual IFIP WG 11.3 Conference on Data and Applications Security and Privacy (DBSec'20). Accepté, à paraître*.

Cramer, R., Damgård, I. B., and Nielsen, J. B. (2015). *Secure multiparty computation*. Cambridge University Press.

Evans, D., Kolesnikov, V., and Rosulek, M. (2018). A pragmatic introduction to secure multi-party computation. *Found. Trends Priv. Secur.*, 2:70–246.

He, Z., Wong, W. K., Kao, B., Cheung, D., Li, R., Yiu, S., and Lo, E. (2015). SDB: A secure query processing system with data interoperability. *Proc. VLDB Endow.*, 8:1876–1879.

Hemenway, B., Welser IV, W., and Baiocchi, D. (2014). Achieving higher-fidelity conjunction analyses using cryptography to improve information sharing. Technical report, Rand project, Air Force. Santa Monica, CA.

Liagouris, J., Kalavri, V., Faisal, M., and Varia, M. (2021). Secrecy: Secure collaborative analytics on secret-shared data. *arXiv preprint arXiv:2102.01048*.

Liu, C., Wang, X. S., Nayak, K., Huang, Y., and Shi, E. (2015). ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376.

López, F. M. S. and De La Cruz, E. G. S. (2015). Literature review about Neo4j graph database as a feasible alternative for replacing rdbms. *Industrial Data*, 18(2):135–139.

Miller, J. J. (2013). Graph database applications and concepts with Neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, volume 2324.

Mostafa, A. (2016). Security of database management systems. pages 1–6. https://www.researchgate.net/publication/301613094.

Nayak, K., Wang, X. S., Ioannidis, S., Weinsberg, U., Taft, N., and Shi, E. (2015). Graphsc: Parallel secure computation made easy. In *2015 IEEE Symposium on Security and Privacy*, pages 377–394. IEEE.

Poddar, R., Kalra, S., Yanai, A., Deng, R., Popa, R. A., and Hellerstein, J. M. (2020). Senate: A maliciously-secure mpc platform for collaborative analytics. *arXiv e-prints*, pages arXiv–2010.

Salehnia, A. (2017). Comparisons of relational databases with big data : a teaching approach. pages 1–8. South Dakota State University, Brookings.

Shamir, A. (1979). How to share a secret. *Communications of the ACM*, 22(11):612–613.

Volgushev, N., Schwarzkopf, M., Getchell, B., Varia, M., Lapets, A., and Bestavros, A. (2019). Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–18.

Wong, W. K., Kao, B., Cheung, D. W. L., Li, R., and Yiu, S. M. (2014). Secure query processing with data interoperability in a cloud database environment. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1395–1406.