Computing Data Encrypted by Paillier Encryption Scheme Using Cassandra Database

Jelena Stankovski, Aleksandar Erdeljan, Ivana Kovačević and Nikola Dalčeković

***Abstract*—In this paper, we investigated a way of computing data, encrypted using Paillier's homomorphic encryption scheme and stored in Cassandra database. For computing data, we used Cassandra's User-Defined-Functions (UDF) and User-Defined-Aggregates (UDA). Sensitive data, such as customers' consumed energy was used as an example, where aggregated energy consumption, over a period time, was computed using UDF and UDA. Cassandra database was deployed on Microsoft Azure platform, where six different clusters were set, consisting of two, four and six nodes.**

***Index Terms*—Paillier encryption scheme; Big data; Cassandra database.**

I. Introduction

One of big concerns today is who can see our information, who has access to it, and how that information is handled. More data are moved to clouds every day, which raises certain questions about our privacy. One way of securing our information and data is cryptography. Using cryptography, information becomes illegible and inapplicable, thus making it unavailable for processing. Homomorphic encryption enables certain mathematical operations to be applied on encrypted information, without knowledge of its keys. One of homomorphic encryption usages is electronic voting, in order to preserve privacy in electronic voting [1]. In [2], homomorphic encryption was used in medical purposes, where patient's name, diastolic and systolic pressure numbers were encrypted using homomorphic encryption. Homomorphic encryption may be applied in finance sector, where "functions computed on the data as well as the data itself should remain private" [2]. Combining storing information on clouds, and homomorphic encryption, sensitive information can be stored in one of NoSQL databases, and still enable mathematical operations to be executed over encrypted data. In this paper, we present a way of handling encrypted data with Paillier's homomorphic encryption scheme, in Cassandra NoSQL database.

Jelena Stankovski is with the Faculty of Technical Sciences, University of Novi Sad, 6 Trg Dositeja Obradovića, 21000 Novi Sad, Serbia (e-mail: jelena.stankovski@ uns.ac.rs).

Aleksandar Erdeljan is with the Faculty of Technical Sciences, University of Novi Sad, 6 Trg Dositeja Obradovića, 21000 Novi Sad, Serbia (e-mail: ftn_erdeljan@ uns.ac.rs).

Ivana Kovačević is with the Faculty of Technical Sciences, University of Novi Sad, 6 Trg Dositeja Obradovića, 21000 Novi Sad, Serbia (e-mail: ivana.kovacevic@ uns.ac.rs).

Nikola Dalčeković is with the Faculty of Technical Sciences, University of Novi Sad, 6 Trg Dositeja Obradovića, 21000 Novi Sad, Serbia (e-mail: nikola.dalcekovic@ uns.ac.rs)

The remainder of this paper is organized as follows. Section II reviews homomorphic encryption and Paillier homomorphic encryption scheme in details. Section III gives an overview on Big Data and NoSQL databases, concentrating on User-Defined-Functions and User-Defined-Aggregates in Cassandra database. Section IV describes conducted tests and presents tests' results. Section V summarizes our testing and gives a scope for future work.

II. Security

A. Cryptography

Cryptography transforms readable data into non-readable data, thus aggravating easy data reads. As stated in [3], cryptography is the science of using mathematics for making plain text information (P) into an unreadable cipher text (C) format called encryption and reconverting that cipher text back to plain text called as decryption with the set of cryptographic algorithms (E) using encryption keys (k1 and k2) and the decryption algorithm (D) that reverses and produces the original plain text back from the cipher text. This can be interpreted as cipher text C = E {P, k1} and plain text P = D {C, k2}.

B. Homomorphic encryption

Certain use cases require mathematical operations over encrypted data. One of these use cases is described in [1], where addition needs to be applied on encrypted data. To be able to perform mathematical operations over encrypted data, encryption algorithms need to have homomorphic property. By definition [4]: Given two groups $(G, \lozenge)$ and $(H, \circ)$, a group homomorphism from $(G, \lozenge)$ to $(H, \circ)$ is a function $f: G \rightarrow H$ such that for all g and g' in G it holds that

$$f(g \lozenge g') = f(g) \circ f(g'). \quad (1)$$

One way to classify homomorphic encryption is by operation which scheme supports. In partially homomorphic encryption, only one operation can be performed, addition or multiplication. A more advanced scheme is somewhat homomorphic encryption, which supports more than one operation, but for a limited number of addition and multiplication operations. The most advanced scheme is fully homomorphic encryption, which can compute any function [2].

Another classification of homomorphic encryption differs additive and multiplicative homomorphic encryption. When addition can be performed on encrypted text and decrypting

cipher text matches the original sum, homomorphic encryption is called additive [5]. As shown below, addition of plain text corresponds to multiplication of cipher text

$$Enc(x \oplus y) = Enc(x) \otimes Enc(y) \quad (2)$$

$$Enc(\sum_{i=1}^{l} m_i) = \prod_{i=1}^{l} Enc(m_i). \quad (3)$$

When multiplication can be performed on encrypted text and decrypting cipher text matches the original multiplication, homomorphic encryption is called multiplicative [5], which corresponds to

$$Enc(x \otimes y) = Enc(x) \otimes Enc(y) \quad (4)$$

$$Enc(\prod_{i=1}^{l} m_i) = \prod_{i=1}^{l} Enc(m_i). \quad (5)$$

Some of homomorphic encryptions are: Goldwasser-Micali, Boneh-Goh-Nissim, Paillier, El-Gamal [4], which are partially homomorphic encryptions, and Gentry [6], which is fully homomorphic encryption.

C. Paillier homomorphic encryption

In this paper, focus will be on Paillier's homomorphic encryption scheme, due to its additive characteristic. This encryption scheme has additive homomorphic encryption property and is an asymmetric algorithm. Scheme was proposed by Pascal Paillier in 1999 [7]. Paillier encryption scheme consists of following steps [4]:

- Key generation: since Paillier encryption is an asymmetric algorithm, public and private key needs to be generated. First step of key generation is choosing two prime numbers *p* and *q*, so that greatest common divisor is 1

$$\gcd(pq, (p-1)(q-1)) = 1. \quad (6)$$

Then *n* and λ need to be computed, where $n = pq$, and λ is the least common multiple of $(p-1)$ and $(q-1)$. Next step includes selecting random integer *g*, where $g \in Z^*_{n^2}$ and compute μ as

$$\mu = (L(g^{\lambda} \pmod{n^2}))^{-1} \pmod{n} \quad (7)$$

where L is defined as

$$L(u) = \frac{u-1}{n}. \quad (8)$$

By computing *n*, λ, and μ and selecting *g*, keys are formed as *Public (n, g)* and *Private (λ, μ)*.

- Encryption: to compute cipher text, random *r*, where $r \in Z^*_n$, needs to be generated, and then cipher text can be computed as

$$c = g^m r^n \pmod{n^2}. \quad (9)$$

- Decryption: to decrypt cipher text, formula below is used:

$$m = L(c^{\lambda} \pmod{n^2})\mu \pmod{n}. \quad (10)$$

- Homomorphic properties: as stated before, Paillier encryption scheme has additive homomorphic properties, i.e. multiplication of encrypted data is equivalent to addition of plain data

$$D(E(m_1, pk)E(m_{2,}\, pk) \pmod{n^2}) = m_1 + m_2 \pmod{n} \quad (11)$$

$$E(m_1, pk)E(m_2, pk) = (g^{m_1} r_1^{\,n})(g^{m_2} r_2^{\,n}) \pmod{n^2} \quad (12)$$

$$= g^{m_1+m_2}(r_1 r_2)^n \pmod{n^2} \quad (13)$$

$$= E(m_1 + m_2, pk). \quad (14)$$

Implementation of Paillier encryption scheme which corresponds to described equations and will be used later is written in Java and located in [8].

III. BIG DATA

One of currently trending topics is Big Data. Considering Big Data as large amount of data or information is misleading, Big Data is much more to that. As stated in [9], Big Data can be described with 5 Vs: velocity, variety, volume, veracity and value. Volume is a characteristic of Big Data which describes the size of data, velocity is the high speed of data, variety indicated heterogeneous data types and sources, veracity describes consistency and value provides outputs for gains from large data sets.

Closely related to term Big Data are NoSQL databases. NoSQL ("no SQL" or "not only SQL") databases are next generation databases, as they are being non-relational, distributed, open-source and horizontally scalable [10]. CAP theorem applies to NoSQL databases, since these databases are a distributed management system. Given that, *Consistency*, *Availability* and *Partition tolerance* cannot be satisfied all at the time, meaning two out of three characteristics are chosen to be applied. NoSQL databases choose availability, partition tolerance in speed instead of consistency. Relational databases have ACID characteristics, i.e. *Atomicity*, *Consistency*, *Isolation* and *Durability*, whereas

BASE characteristics, *Basically available*, *Soft state*, *Eventually consistency*, apply NoSQL. NoSQL databases are divided into four major types [11]:

- Key value store: this type of database is schema free and data is stored as a key-value pair, similarly to maps or dictionaries. NoSQL databases which are key value store are Project Voldemort and Redis.
- Column family store: also known as wide column store. Column family store data model is "sparse, distributed and a persistent multidimensional sorted map" [12]. Cassandra, HBase and Bigtable are column family NoSQL databases.
- Graph database: specialized for efficient management of heavily linked data. Applications which use many relationships are more suited for graph databases. Neo4j and GraphDB are graph NoSQL databases.
- Document store: data is organized into documents, where each document has a special key "ID", which is unique and identifies a document explicitly. JSON format is used for representing data in document store databases. Some of document store databases are MongoDB and Riak.

A. Cassandra database

Cassandra is an open source, distributed NoSQL database management system. By being distributed management system, Cassandra has data across all here nodes, thus making Cassandra an available service with no single point of failure [13]. According to [14], Cassandra is the leading wide column store database, given its popularity. For writing queries in Cassandra, Cassandra query language (CQL) is used, as an alternative to Structured query language (SQL). Key terms in Cassandra architecture are [15]:

- Node: place where data is stored, it is the basic infrastructure component of Cassandra. Specific node is coordinator node, which determines which nodes in the ring should get the request, based on the cluster configured snitch.
- Data center: a collection of related nodes. Replication is set by data center.
- Cluster: is a container for one or more data centers.

To fully understand Cassandra, Cassandra's data model should be explained. *Keyspace* is the container of column families, which can be analogous to schema in relational databases. *Column family* represents an ordered list of columns, which is analogous to table in relational databases. *Row* is a collection of columns and is identified by a key, which determines which node the data is stored on. *Column* is a triplet that contains a name, a value and a timestamp, making it the smallest increment of data. *Super column* is a column whose values are columns [13].

B. User-Defined-Functions and User-Defined-Aggregates

As of Apache Cassandra 2.2, User-Defined-Functions (UDF) and User-Defined-Aggregates (UDA) are available. UDF and UDA enable user to build their own scalar functions, and their own aggregations [16]. The idea behind UDF and UDA is to transfer computation to server side, thus avoiding retrieving data and applying aggregation on the client-side [12]. UDF and UDA are executed on the coordinator node, meaning all the necessary data for functions and aggregations need to be retrieved to coordinator node. Usage of UDF and UDA, as well as CQL syntax for creating UDF and UDA will be given and interpreted in next section.

IV. CASE STUDY

Since the purpose of UDF and UDA is building custom functions and aggregations, combining that with Paillier's encryption scheme, sensitive data can be stored in Cassandra and custom aggregations can be applied on that data. For demonstration, we will use an example of energy consumption, i.e. how much energy customers spend hourly. Since the amount of energy consumed is sensitive data, this data can be encrypted using Paillier's encryption. In this paper, we will not discuss how key management is done, or how the data is sent to the Cassandra cluster. This will be considered done in most secured and efficient way. Our goal is to show how UDF and UDA can be used to compute energy consumption, which is encrypted, for a certain customer, in a certain period.

Testing scenario that will be described is that every customer's energy consumption for the hour is sent to a certain database, where it is stored, in encrypted form, using Paillier's encryption scheme. For data storage, we will use Cassandra. Also, connected to Cassandra database is a utility, which needs to have information on how much energy certain customer has consumed over a given period. For encrypting consumed energy and decrypting the amount of consumed energy over time Paillier encryption scheme implementation is used [8]. When acquiring data from database, utility will receive aggregated data from Cassandra in encrypted form, which needs to be decrypted. For storing customers' consumed energy over time, 'consumption' table can be created as, in keyspace 'test':

```
CREATE KEYSPACE test WITH REPLICATION =
{'class': 'SimpleStrategy',
'replication_factor': 2};
CREATE TABLE consumption (
customer_unique_id uuid,
day date,
ts timestamp,
consumed_energy varint,
PRIMARY KEY (customer_unique_id, ts));
```

Property '`customer_unique_id`' represents the customer's unique id, '`day`' represents the day for which the read was done, '`ts`' represents the time when the energy consumption was read, '`consumed_energy`' represents the amount of energy consumed for the day. The primary key in this table is customer's id and '`ts`'. Since the consumed energy will be encrypted, *varint* type is used for storing large numbers.

To compute amount of consumed energy over a certain period, UDF and UDA need to be applied. For this purpose,

function ‘multiply’ is created, with arguments ‘next’ and ‘current’. This function returns multiplied values, since the input data are encrypted and Paillier’s encryption scheme for encrypted data is equivalent to addition of plain text. Also, an aggregate ‘multiplyPaillier’ is created, which will be called for column which needs to be aggregated, i.e. column ‘consumed _energy’. Aggregated value’s size is increased by each operation, which can result in over 10,000 ciphers. The type of aggregated value is ‘varint’, which corresponds to ‘BigInteger’ in Java, thus large numbers are not an issue, given that ‘BigInteger’ range is $-2^{Integer.MAX\_VALUE}$ to $+2^{Integer.MAX\_VALUE}$.

```
CREATE OR REPLACE FUNCTION
test.multiply(next varint, current
varint)
CALLED ON NULL INPUT
RETURNS varint
LANGUAGE java
AS '
if (next == null)
  return current;
 else
   return next.multiply(current);';
```

Interpreting creation of UDF:

- UDF can be created using ‘CREATE’ clause, combined with ‘OR REPLACE’, if the UDF already exists.
- Since the scope of UDF is keyspace, ‘test.’ can be added, where it represents the keyspace’s name.
- List of arguments can be passed to UDF, where type and argument name should be provided. In our example, ‘next’ and ‘current’ are parameters with type ‘varint’.
- When ‘CALLED ON NULL INPUT’ is used, UDF will be called even if the input argument is null. Instead of this statement, ‘RETURNS NULL ON NULL INPUT’ can be used, and will return null for null argument.
- Return type should be set according to specified list of valid CQL types.
- Available languages to write source code are Java, Javascript, Groovy, Scala, JRuby and JPython. In our example, Java language is used.

For creating UDA, next statement is used:

```
CREATE OR REPLACE AGGREGATE
test.multiplyPaillier(varint) SFUNC
multiply STYPE varint INITCOND null;
```

Interpreting creation of UDA:

- As in UDF creation, rules for ‘CREATE’ clause apply.
- Unlike UDF, UDA only accepts the list of types as an input parameter.
- ‘SFUNC’ clause points to UDF, as an accumulator function.
- STYPE’ clause is the type of the state to be passed to ‘SFUNC’ for accumulation, as well as the return type.
- ’INITCOND’ states the initial value of the state.

To compute consumed energy in one month, next statement should be executed:

```
SELECT multiplyPaillier(consumed_energy)
FROM consumption WHERE day<'2016-05-27'
AND day>'2016-04-27' and
customer_unique_id=’ 9c41cebc-a987-11e6-
80f5-76304dec7eb7;
```

Our testing scenario included computing consumed energy for a period, for one customer. However, another useful information may also be how much energy was consumed over a period, for a larger area, e.g. substation or a region. To gain such information, ‘WHERE’ clause would have to involve more customers.

To test this scenario, we have set up Cassandra cluster on Microsoft Azure, with two, four and six nodes. Each node is a Standard D2 v2 machine (2 CPU cores, 7GiB Memory, 4x500 IOPS) [17]. Since Cassandra is distributed database, it is very unlikely to be used as a single node cluster. Each cluster has a keyspace, table, UDF and UDA set as previously described. Also, table has 100.000 rows. Testing will be done for a period of one, two, three, four and five months. To measure time needed to compute consumed energy over time, ‘tracing on;’ option is used in Cassandra. When tracing option is turned on, execution time of every statement is measured, and shown in μs.

In Figure 1, time needed to compute consumed energy for one customer, over a certain period is shown.

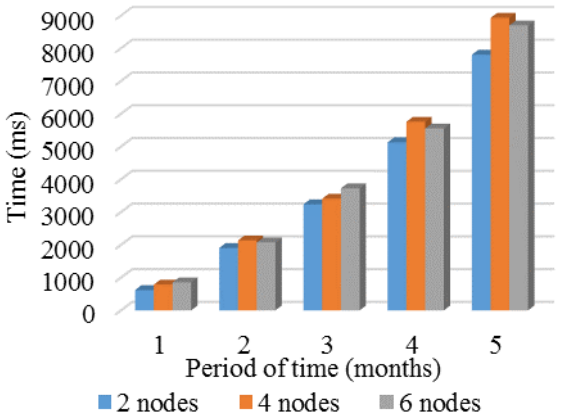

Figure 1. Time needed for computing consumed energy on Standard D2 v2 machines

Time needed to compute is shown in *ms*, and the period, for which the computation is done, is given in months. To compute encrypted energy consumption over one month for one customer, with daily energy consumption being stored, Cassandra needs 623ms, when cluster has two nodes, 788ms for cluster with four nodes, and 855ms for cluster with six nodes. As stated earlier, UDF and UDA are executed only on coordinator node, which means that all needed data is to be retrieved, which leads to greater time needed to compute same amount of data, across clusters with different node count. As the amount of data needed to be computed rises, so does the execution time. Time needed to compute energy consumption over five months is 7805ms for a cluster with two nodes, 8929ms for a cluster with four nodes and 8697ms for cluster

with six nodes.

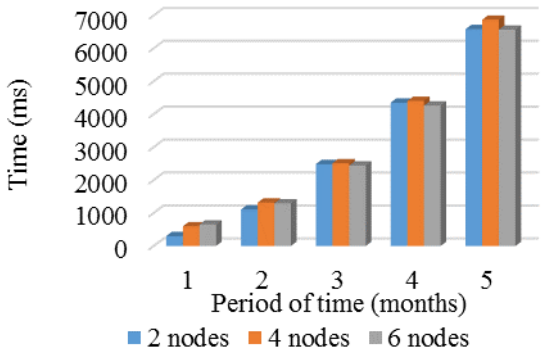

Figure 2. Time needed for computing consumed energy on Standard D4 v2 machines

To compute encrypted energy consumption over one month for one customer, with daily energy consumption being stored, using Standard D4 v2 machines, Cassandra needs 302ms, when cluster has two nodes, 599ms for cluster with four nodes, and 653ms for cluster with six nodes. These times are smaller than the ones in previous case. Reason for smaller execution time is that Standard D4 v2 machines have greater number of CPU cores, memory and IOPS.

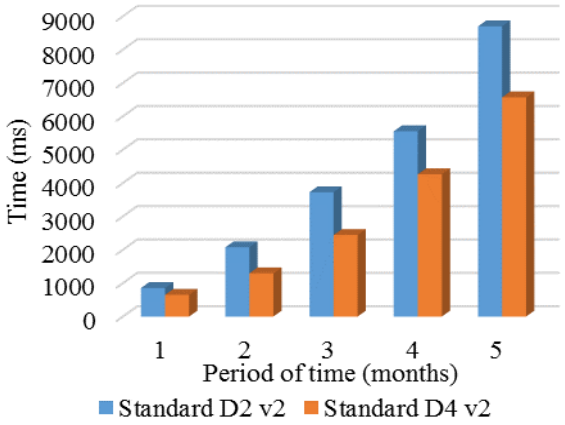

Figure 3. Comparison between clusters with six nodes and different machine types

Another graph is shown on Figure 3, which depicts the difference between cluster with six nodes using Standard D2 v2 machines, and cluster with six nodes using Standard D4 v2 machines. Time needed to compute consumed energy is roughly 29% smaller when Standard D4 v2 machines are used, which indicates the importance of choosing appropriate machine.

To show how encryption slows down the process of computing using UDF and UDA, test which contained same amount of data, but used plain data instead of using encrypted values for consumed energy, was done. Test was done for a cluster with four nodes, each node was a Standard D2 v2 machine. To compute consumed energy for five months, Cassandra needs 246ms, which is 36 times faster than the time needed to compute same amount of encrypted data.

V. Conclusion and Future Work

Using UDF and UDA for computing small amount of data is fast, but as the amount of data needed to be computed rises, the execution time rises almost exponentially. Also, UDF and UDA are not made for distributed use, so the time needed for computation rises with more nodes in cluster, since data needs to be retrieved to coordinator node. Using stronger machines, computing time can be reduced, as shown in Figure 3. One of reasons to use UDF and UDA is to transfer computation on server-side, instead of leaving it on the client-side. Given the amount of data that can be computed in short period, it is to be consider the benefits of UDF and UDA. If amount of data needed to be computed is large, powerful machines may not be the solution, even though stronger machines have smaller computing time. Proposal for future work relays on Map Reduce algorithm, which may be more suitable solution, since it is more suitable for distributed architectures. Another benefit from Map Reduce algorithm may help in case of large amount of data needed to be computed.

References

[1] A. Azougaghe, M. Hedabou, M. Belkasmi. "An electronic voting system based on homomorphic encryption and prime numbers", *Information Assurance and Security (IAS), 2015 11th International Conference on*. IEEE, pp. 140-145, December 2015.

[2] M. Ogburn, C. Turner, P. Dahal. "Homomorphic encryption", *Procedia Computer Science*, vol. 20, pp. 502-509, November 2013.

[3] A. Bhardwaj., G.V.B. Subrahmanyam, V. Avasthi, H. Sastry, "Security Algorithms for Cloud Computing", *Procedia Computer Science*, vol. 85, pp. 535-542, June 2016.

[4] X. Yi, R. Paulet, E. Bertino, *Homomorphic encryption and applications*, vol. 3, Berlin, Germany, Springer, 2014.

[5] M. Tebaa, S. Hajji, A. El Ghazi. "Homomorphic encryption applied to the cloud computing security" *Proceedings of the World Congress on Engineering*. vol. 1, pp. 4-6, 2012.

[6] C. Gentry, "*A fully homomorphic encryption scheme*", Ph.D. dissertation, Standford Univ., Stanford, USA, 2009.

[7] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes", *International Conference on the Theory and Applications of Cryptographic Techniques*, Berlin, Germany, Springer, 1999.

[8] http://www.csee.umbc.edu/~kunliu1/research/Paillier.html

[9] D.S. Terzi, R. Terzi, S. Sagiroglu. "A survey on security and privacy issues in big data" *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, pp. 202-207, 2015.

[10] http://nosql-database.org

[11] R. Hecht, S. Jablonski. "Nosql evaluation: A use case oriented survey", *International conference on cloud and service computing*, pp.336-341, 2011.

[12] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, R.E. Gruber, "Bigtable: A Distributed Storage System for Structured Data", *ACM Transactions on Computer Systems*, vol. 26, pp. 1-26, 2008.

[13] L. Okman, N. Gal-Oz, Y. Gonen, E. Gudes, J. Abramov, "Security issues in nosql databases", *2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pp.541-547, 2011.

[14] http://db-engines.com/en/ranking

[15] https://docs.datastax.com/en/cassandra/2.0/cassandra/architecture/architectureIntro_c.html

[16] http://www.datastax.com/dev/blog/user-defined-aggregations-with-spark-in-dse-5

[17] https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-windows-size