International Journal of Information Management 37 (2017) 63–74

Contents lists available at ScienceDirect

International Journal of Information Management

journal homepage: www.elsevier.com/locate/ijinfomgt

SecureNoSQL: An approach for secure search of encrypted NoSQL databases in the public cloud☆

Mohammad Ahmadian*, Frank Plochan, Zak Roessler, Dan C. Marinescu

Department of Computer Science, University of Central Florida, Orlando, FL 32816, USA

ARTICLE INFO

Article history:
Received 12 April 2016
Accepted 17 November 2016

Keywords:
Search over encrypted data
Database as a service
NoSQL
Encryption
Cloud computing
Security
Query processing
Data integrity

ABSTRACT

While many schemes have been proposed to search encrypted relational databases, less attention has been paid to NoSQL databases. In this paper we report on the design and the implementation of a security scheme called "SecureNoSQL" for searching encrypted cloud NoSQL databases. Our solution is one of the first efforts covering not only data confidentiality, but also the integrity of the datasets residing on a cloud server. In our system a secure proxy carries out the required transformations and the cloud server is not modified. The construction is applicable to all NoSQL data models and, in our experiments, we present its application to a *Document-store* data model. The contributions of this paper include: (1) a descriptive language based on a subset of *JSON* notations; (2) a tool to create and parse a *security plan* consisting of cryptographic modules, data elements, and mappings of cryptographic modules to the data fields; and (3) a query and data validation mechanism based on the *security plan*.

1. Introduction and motivation

Data analytics, enterprise, and multimedia applications, as well as applications in many areas of science, engineering, and economics, including genomics, structural biology, high energy physics, astronomy, meteorology, and the study of the environment take advantage of cloud computing for processing very large datasets. Companies heavily involved in cloud computing such as Google and Amazon, e-commerce companies such as *eBay*, and social media networks such as Facebook, Twitter, or LinkedIn discovered early on that traditional relational databases cannot handle the massive amount of data and the real-time demands of online applications critical for their business model. The relational schema is of little use for such applications and conversion to NoSQL databases seems a much better approach.

The name *NoSQL* given to the storage model discussed in this paper might be misleading. Michael Stonebreaker notes that "blinding performance depends on removing overhead. Such overhead has nothing to do with SQL, but instead revolves around traditional implementations of ACID transactions, multi-threading, and disk management" (Stonebraker, 2010). The "soft-state" approach in the design of *NoSQL* databases allows data to be inconsistent and transfers the task of implementing only the subset of the ACID properties required by a specific application to the application developer. *NoSQL* systems ensure that data will be "eventually consistent" at some future point in time, instead of enforcing consistency at the time when a transaction is "committed". Data partitioning among multiple storage servers and data replication are also tenets of the *NoSQL* philosophy as they increase availability, reduce the response time, and enhance scalability.

Big Data and mobile applications are the two most important growth areas of cloud computing. Big Data growth can be viewed as a three-dimensional phenomenon; it implies an increased volume of data, requires an increased processing speed to produce more results, and at the same time, it involves a diversity of data sources and data types (Marinescu, 2013). A delicate balance between data security and privacy and efficiency of database access is critical for such applications. Many cloud services used by these applications operate under tight latency constraints; moreover, these applications have to deal with extremely high data volumes and are expected to provide reliable services for very large communities of users. Nowadays NoSQL databases are widely supported by cloud service providers. Their advantages over traditional databases are critical for Big Data application.

☆ The datasets used in the experiments reported in this paper are available at: https://github.com/MoAhmadian/SecureNoSQL.

* Corresponding author.
E-mail addresses: ahmadian@knights.ucf.edu (M. Ahmadian), frank.plochan@knights.ucf.edu (F. Plochan), zak.roessler@knights.ucf.edu (Z. Roessler), dcm@cs.ucf.edu (D.C. Marinescu).
URL: http://www.cs.ucf.edu/ ahmadian (M. Ahmadian).

http://dx.doi.org/10.1016/j.ijinfomgt.2016.11.005
0268-4012/

Data security and integrity are important factors when choosing a database for cloud applications. They are particularly critical for applications running on public clouds where multiple virtual machines (VMs) often share the same physical platform (Xu, Jiang, Wang, Yuan, & Ren, 2014; Xu, Zhang, Wu, & Shi, 2015; Yu & Wen, 2010). The importance of database security and its impact on a large number of individuals are illustrated by the consequences of two major security breaches: Weiss and Miller (2015); and Silver-Greenberg, Goldstein, and Perlroth (2014). In November 2013 approximately 40 million records were stolen from an unencrypted database used by Target stores. The compromised information included *personally identifiable information* (PII) and credit card data. According to a SEC (Securities and Exchange Commission) report, two months later a cyber-attack on JP Morgan Chase, compromised PII records of some 76 million households and 7 million small businesses.

Classic cryptography primitives can protect data while in storage, but plaintext data is vulnerable to insider interference during processing. This is particularly troubling when searching databases containing personal information such as healthcare or financial records (Lin, Tsai, & Lin, 2014), as the entire database is exposed to such attacks. These circumstances motivate us to investigate methods for searching encrypted *NoSQL* databases. Though general computations with encrypted data are theoretically feasible using the algorithms for Fully Homomorphic Encryption (FHE) (Gentry, 2009), this is by no means a practical solution at this time. Existing algorithms for homomorphic encryption increase the processing time of encrypted data by many orders of magnitude compared with the processing of plaintext data. A recent implementation of FHE (Halevi & Shoup, 2014) requires about six minutes per batch; the processing time for a simple operation on encrypted data dropped to almost one second after improvements (Ducas & Micciancio, 2015). Related areas of research are: Learning With Errors (LWE) (Brakerski & Vaikuntanathan, 2011) and Lattice-based encryption (Cash, Hofheinz, Kiltz, & Peikert, 2012; Micciancio & Regev, 2009) and Attribute-based Encryption (Gorbunov, Vaikuntanathan, & Wee, 2013).

In this paper we restrict our discussion to query processing particularly over encrypted *NoSQL* databases. A secure proxy called "SecureNoSQL" for accessing cloud remote servers and applying efficient cryptographic primitives for query, response and data encryption/decryption is introduced. We also designed a descriptive language using *JSON*[1] notation which enables its users to generate a *security plan*. The security plan has four sections which elaborately introduce the data elements, cryptographic modules and the mappings between them. The main contributions of this paper are:

- A JSON-based language for users to: (i) create a security plan for the database; (ii) describe the security parameters; and (iii) assign proper cryptographic primitives to the data elements.
- A multi-key, multi-level security mechanism for policy enforcement. This feature is essential because the encryption key is subject to more frequent changes than the crypto-module. Furthermore, keys are assigned for a single data element, while encryption algorithms could be applied for several data elements with several keys. This separation allows a more efficient enforcement of security policy and of key management.
- An effective validation process for the security plan. This validation process enables users to initially evaluate all requests locally, rather than forwarding large numbers of fallacious key-value pairs to a cloud server. It also limits the cloud server workload and reduces the response time latency.
- Support for a comprehensive, flexible protection. The solution is open-ended; users can add new customized cryptographic modules simply by using the designed descriptive language.
- A balanced system with a security level-proportional overhead; the overhead is proportional to the desired level of security.
- A secure proxy which translates queries to run over encrypted data on the remote cloud server with respect to semantics of queries. The cloud database server is not modified and treats encrypted documents in the same way as a plaintext database. Properties of the distributed database such as replication hold for encrypted data.
- Support for cloud data integrity and protection against an insider attack.

The rest of this paper is organized as follows: related work and *NoSQL* data models are presented in Section 2. The threat model for a cloud database is discussed in Section 3. The organization of the system is presented in Section 4 and structure of the security plan and the notation of the descriptive language for generation of security plan is discussed in Section 5. Then the mechanism of query processing is investigated in Section 6. Finally, in Section 8 we report on measurements of the database response time to different types of queries and on the encryption and decryption time for OPE encryptions with output lengths of 64, 128, 256, 512 and 1024-bit.

2. NOSQL databases and related work

NoSQL describes a fairly large number of *NoSQL* database technologies, more than 120 by our count, have been created in recent years. NoSQL databases are non-relational, distributed, horizontally scalable, and schema-free. They are classified based on their data models. Choosing proper a data model has an extremely important influence on the performance and scalability of the data stores. Since our work has a tight connection to NoSQL data models, we provide brief definitions for several data models.

2.1. Key-value store

This simple data model resembles an associative map or a dictionary where a key uniquely identifies the value. The data can be either a primitive data type such as a string, an integer, an array, or it can be an object. This model is effective for storing distributed data; thus, it is highly scalable and this motivates its use by cloud data management systems. Systems such as Bigtable (Chang et al., 2008), CouchDB,[2] DynamoDB (Sivasubramanian, 2012), *MemcacheDB*[3] and *Redis* [4] use this model. This model is not suitable for applications demanding relations or structures.

2.2. Column-family store

In this model the data are stored in a column-oriented style and the dataset is comprised of several rows, each row is indexed by a unique key, the so-called *primary key*. Each row is composed of a set of column families, and different rows can have different column families. Similarly, the row key resembles the key, and the set of column families resembles the value represented by the row key.

[1] *JSON* (JavaScript Object Notation) is a lightweight text-based syntax for storing and exchanging data objects consisting of key-value pairs. It is used primarily to transmit data between a server and a web application. JSON's popularity is due to the fact that it is self-describing and easy to understand by human and machine. For more information, visit: http://www.json.org.

[2] http://couchdb.apache.org.

[3] http://www.Memcached.org.

[4] http://redis.io.

However, each column family further acts as a key for the one or more columns that it holds, whereas each column consists of a key-value pair. *Hadoop HBase* directly implements the Google *Bigtable* concepts, whereas Amazon *SimpleDB* and *DynamoDB* contain only a set of column name-value pairs in each row, without having column families. Sometimes, *SimpleDB* and *DynamoDB* are classified as key-value stores. Typically, the data belonging to a given row are stored together at the same server node. *Cassandra* provides the additional functionality of super-columns, which are formed by grouping various columns together. *Cassandra* can store a single row across multiple server nodes using composite partition keys. In column-family stores, the configuration of column families is typically performed during start-up. A column family in different rows can contain different columns. A prior definition of columns is not required and any data type can be stored in this data model. In general, column-family stores provide more powerful indexing and querying than key-value stores because they are based on column families and columns in addition to row keys. Similar to key-value stores, any logic requiring relations must be implemented in the client application.

2.3. Document store

In this model data are stored inside an internal structure, while in the key-value store the data are opaque to the database. Thus, the database engine applies metadata to create a higher level of granularity and delivers a richer experience for modern programming techniques. Document-oriented databases use a key to locate the document inside the data store. Most document stores use *JSON* or *BSON* (Binary JSON). Document stores are suited to applications where the input data can be represented in a document format. A document can contain complex data structures such as nested objects. A *document store* allows document grouping into collections. A document in a collection should have a unique key. Unlike a relational database management system (RDBMS),[5] where every row in a table follows the same schema, a document in document stores may have a different structure from other documents. Document stores provide the capability of indexing documents based on the primary key as well as on the contents of the documents. Like key-value stores, they are inefficient in multiple-key transactions involving cross-document operations.

2.4. Graph database

This data model is used to represent complex structures and the highly connected data often encountered in real-world applications. In graph databases, the nodes and edges have individual properties consisting of key-value pairs. Graph databases are a good alternative for social networking applications, pattern recognition, dependency analysis and recommendation systems. Some graph databases such as *Neo4J*[6] support ACID[7] properties. Graph data stores are not as efficient as other NoSQL data stores and do not scale well horizontally when related nodes are distributed to different servers.

The first SQL-aware query processing using an encrypted database was *CryptDB* (Popa, Redfield, Zeldovich, & Balakrishnan, 2011). CryptDB satisfies data confidentiality for an SQL relational database. However, *CryptDB* cannot perform queries over data encrypted with different keys. One important application of searching encrypted data (Cash et al., 2013, 2014; Cheon, Kim, & Kim, 2016; Song, Wagner, & Perrig, 2000; Tu, Kaashoek, Madden, & Zeldovich, 2013) is in cloud computing where the clients outsource their storage and computation. In Cash et al. (2014) a practical searchable security scheme is introduced which can search on encrypted data sets in sub-linear time complexity by using different types of indices; however, it is not practical on NoSQL data sets which are designed to scale to millions of users doing updates simultaneously (Cattell, 2011).

Order-preserving symmetric encryption (OPE) is a deterministic encryption scheme which maps integers in the range $[1, M]$ into a much larger range $[1, N]$ and preserves numerical ordering of plaintexts (Boldyreva, Chenette, Lee, & O'neill, 2009; Mavroforakis, Chenette, O'Neill, Kollios, & Canetti, 2015). OPE is attractive because fundamental database operations such as sorting, simple matching (i.e., finding m items in a database), range queries (i.e., finding all m items within a given range), and search operations can be carried out efficiently over encrypted data. Moreover, OPE allows query processing to be done as efficiently as for unencrypted data; the database server can locate the desired encrypted data in logarithmic-time via standard tree-based indexing data structures.

An investigation of OPE security against a *known plaintext attack* with known N plaintexts is reported in Xiao and Yen (2012) and Kerschbaum (2015); the last paper concluded that the ideal OPE module accomplishes one-wayness security.[8] The Shannon entropy[9] achieved by an ideal OPE is maximal when the mapping of integers in the range $[1, M]$ to a much larger range $[1, N]$ results in a uniform distribution. The risk of disclosure caused by main memory attack is quantified by Canim, Kantarcioglu, Hore, and Mehrotra (2010) and Bajaj and Sion (2014). An application of OPE in cloud environment is reported in Ahmadian, Paya, and Marinescu (2014) and Ahmadian (2017). Also, application of classical cryptography on relational database system for embedded devices was studied in Ahmadian, Khodabandehloo, and Marinescu (2015).

NoSQL databases are suffering from lack of proper data protection mechanism because these databases have been designed to support high performance and scalability requirement. In order to protect personal and sensitive information, a privacy and security preserving mechanism is required in Big Data platforms. Integration of privacy aware access control features into existing Big Data is discussed in Colombo and Ferrari (2015), Liang, Susilo, and Liu (2015), and Islam and Islam (2014). In Gantz and Reinsel (2012) and Tankard (2012) the evolution of Big Data Systems from the perspective of an information security application is studied. As a matter of fact, the proxy is very important element in the designed structure and from Information Technology prospect view there should be special consideration for its protection. A cloud based monitoring and threat detection system was proposed by Cheon et al. (2016) and Chow et al. (2009) for critical component to make infrastructure systems secure.

3. A cloud computing threat model

A threat model describes the threats against a system. The threat model of cloud computing can be analyzed from multiple viewpoints and we investigate it from an *adversarial* prospective. The adversarial threat model for the Database as a Service (DBaaS) is a holistic process based on end-to-end security. The model identifies two classes of threats, as external and internal attackers.

[5] A relational database management system (RDBMS) is a database management system (DBMS) that is based on the relational model.

[6] http://neo4j.com.

[7] ACID (Atomicity, Consistency, Isolation, Durability) properties guarantee that database transactions are processed reliably.

[8] One-way functions are easy to compute, but computationally hard to invert.

[9] The entropy measures the degree of uncertainty; the Shannon entropy of a discrete random variable X with n realizations $x_1, x_2, \ldots, x_n$ with probabilities $p_1, p_2, \ldots, p_n$, respectively, is: $H(X) = -\sum_{i=1}^{n} p_i \log p_i$.

3.1. External attacker

An attacker from the outside of cloud environment might obtain unauthorized access to the hosted databases by applying techniques or tools to monitor the communication between the clients and the cloud servers. External attackers have to bypass firewalls, intrusion detection systems and other defensive tools without any authorization.

3.2. Malicious insiders

An insider attacker has different level of access to cloud resources. Unauthorized access by malicious insiders who can bypass most or all data protection mechanisms is a major source of concern for cloud users. Encrypted data and a secure proxy construction such as *SecureNoSQL*, guarantees that malicious insiders cannot access user data. The proxy encrypts/decrypts data and query/response between clients and cloud. There is still the residual risk of information leakage from encrypted datasets. A malicious insider could exploit the leaked information to organize more extensive attacks and amplify the information leakage.

4. System organization

This section introduces a framework to incorporate data confidentiality and information leakage prevention algorithms. SecureNoSQL leverages secure query processing for web and mobile applications using DBaaS. Two different system organizations can address our design objectives. The first is suitable when all database users belong to the same organization. Then the proxy runs on a trusted server behind a firewall and the communication between clients and the proxy is secure.

When the clients access the cloud using the Internet the second organization is advisable. In this case, either the client software includes a copy of the proxy and only encrypted data is transmitted over public communication lines, or the *Secure Sockets Layer (SSL)* protocol is used to establish a secure connection to the proxy. Fig. 1 illustrates the high-level architecture of *SecureNoSQL* as a secure proxy between user's applications and cloud NoSQL database server. The system we report on was designed with several objectives in mind:

- Support multi-user access to an encrypted *NoSQL* database. Enforce confidentiality, privacy of transactions and data integrity.
- Hide from the end-users the complexity of the security mechanisms; the database access should be transparent and the user's access should be the same as for an unencrypted database.
- Avoid transmission of unencrypted data over public communication lines.
- Do not require any modification of the *NoSQL* database management system.
- Create an open-ended system; allow the inclusion of cryptographic modules best suited for an application.

These objectives led us to design a system where a proxy mitigates the client access to the cloud remote server running an unmodified *NoSQL* database processing system. In this system the processing of a query involves three phases:

1. Client-side query encoding in *JSON* format carried out by the client software;
2. Query encryption and decryption done by a trusted proxy; and
3. Server-side query processing performed by an unmodified *NoSQL* database server.

SecureNoSQL is based on general principles of NoSQL database products. We introduce a new concept, the *security plan*, materialized as a JSON description of data elements, metadata and parameter configuration of cryptosystems. A descriptive language is introduced to generate and parse the security plan automatically. JSON, a dominant format in NoSQL databases, is selected to express the designed security plan. We used a subset of JSON notation readable by human and machine.

Document databases, such as MongoDB, store documents inside the collection by JSON representation in a similar way as tables and records in relational database systems. A query and the corresponding response are also represented in the JSON format; therefore, the governing format in document database is JSON. BSON, a binary extension of JSON, is used by document-oriented databases for efficient encoding/decoding.

JSON query model is a functional, declarative notation, designed especially for working with large volumes of structured, semi-structured and unstructured JSON documents. The data owner develops the security plan that outlines and maps out the determined crypto-primitive with specific parameters to a particular data element.

5. Descriptive language for security plan

The NoSQL database benefits from flexible scheme that allows to have a different number of attributes for the documents corresponding to the same object. On the other hand, a full list of attributes is required to create a comprehensive protection for all data elements in the database. Therefore, we define a logical operator denoted as *Super Document*, the union of all attributes from different versions of the documents related to the same object. Each database $\mathcal{D}$ consists of a set of arbitrary number n documents.

$$\mathcal{D} = \{d_1, \ldots, d_n\}$$

Furthermore, documents comprised of an arbitrary number m attributes in which each attribute also is built up with a key value pair $\langle k, v\rangle$.

$$d_i = \{A_1, \ldots, A_m\}, \quad 1 \leqslant i \leqslant n$$

In other words, a *Super Document* in the scope of a collection (databases) is an aggregation of attributes representing specific entity. Thus for any given document d_i it is required to look for $n-1$ documents to extract attributes that are not member of d_i (relative complement). This concept is rephrased in Eq. (1). In addition, a match function $\mu(d_i, d_j)$ determines whether two given documents d_i, d_j are desirable for merging or not. Two documents can be combined if they share the same attribute from an *identifying* type.

Super Document σ is defined as: $\sigma_{i,j} = \langle d_i, d_j\rangle$

$$\exists A_p \in d_i \wedge \exists A_q \in d_j \quad \textit{if} \quad (\mu(d_i, d_j) == \textit{True}) \Rightarrow \sigma_{i,j} = d_i \cup d_j$$

The function μ is defined as:

$$\mu(d_i, d_j) = \begin{cases} \textit{True} & \textbf{iff} \;\; \exists A_p \in d_i \wedge \exists A_q \in d_j \mid [(A_p.key = A_q.key) \wedge (A_p.value = A_q.value)] \\ \textit{False} & \textbf{Otherwise} \end{cases} \tag{1}$$

Provided that A_p and A_q are identifier attributes.

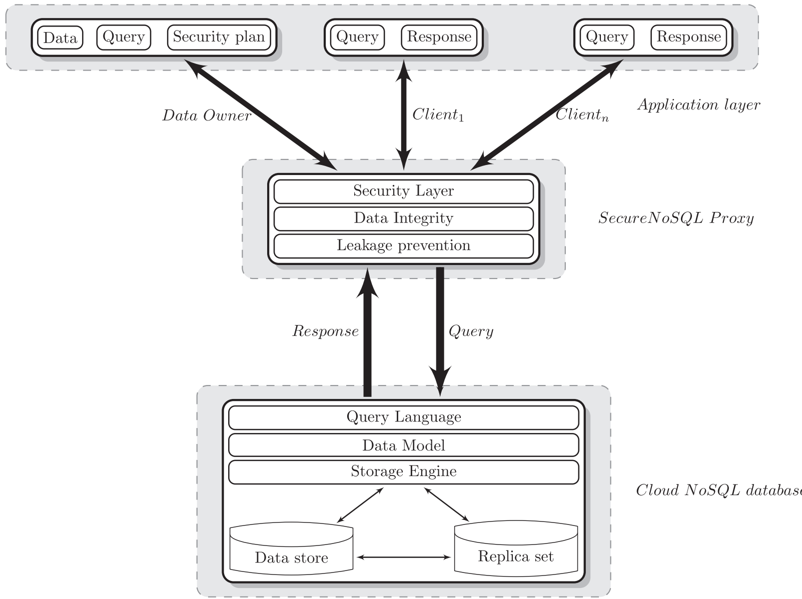


Fig. 1. The organization of the *SecureNoSQL*.

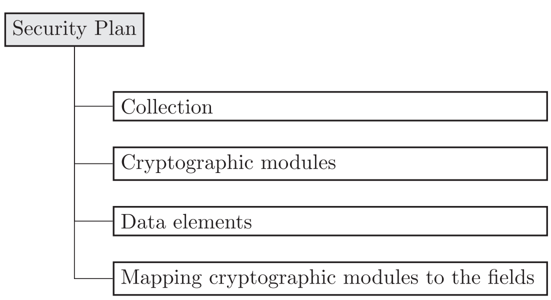


Fig. 2. The high level structure of the security plan.

5.1. Database security plan

The security plan identifies the mechanism to maintain the security of the data elements in a database. It also determines how to interpret queries issued by a specific application. The security plan has four sections, see Fig. 2, describing the security rules for the data elements and for meta-data such as the field-name (Key) and the collection name. These sections are the building blocks of the security plan showing how the rules are enforced. The sections and their roles are:

1. *Collection:* includes the name of a collection and a reference to the encryption module used to encrypt the name of the collection and the name of fields (metadata).
2. *Cryptographic modules:* lists the cryptographic modules for encrypting the fields of the database entries in the query.
3. *Data elements:* lists the properties of each data field including the data type; the data type determines the cryptographic modules to be applied to each field.
4. *Mapping cryptographic modules to the fields:* assigns the cryptographic modules to data fields; proxy uses this information to encrypt and decrypt the data elements.

5.1.1. Collection

A collection is defined as a group of *NoSQL* documents, the equivalent of relational database table, see Fig. 3. The name of the collection must be encrypted. The listing 3b illustrates how to secure a sample collection using the description language.

The key-value pairs (KVP) are the primary data model for a *NoSQL* database. The *key* is used as an index to access the associated value of the data pointed by the reference *ref*. The *initialization vector* (IV) is a fixed-size, random input to the cryptographic module *encryption*. Additionally, a collection exists within a single database. Documents within a collection can have different fields. Typically, all documents in a collection are related with one another.

5.1.2. Cryptographic modules

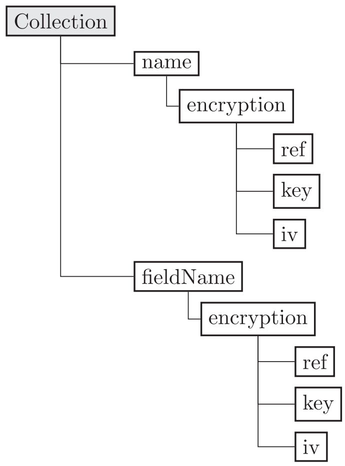
The choice of a particular cryptosystem depends on the security policy of application. Multiple criteria for algorithm selection include: (i) the security against theoretical attacks; (ii) the cost of implementation; (iii) the performance; and (iv) whether the encryption and decryption can be parallelized. Other factors involved in the selection of an algorithm are the memory requirements and the integration in the overall system design.

The *Cryptographic modules* introduce all encryption modules and their parameters such as key, key-size, initialization vector and output-size. The structure of this section is shown in Fig. 4a complemented by the listing in Fig. 4b presenting the second section of security plan for the previous example.

Our proof of concept uses the parametric Order Preserving Encryption (OPE) and the Advanced Encryption Standard (AES) modules. The system is open-ended; users can add the cryptosystems best suited to the security requirements of their application. In our design the definitions of the cryptographic modules and of the pairs, encryption key and initialization value, are separated following the so-called *key separation principle* (Galiegue & Zyp, 2013). This security practice is based on the observations that users have long- and short-term security policies. The cryptographic modules are less likely to change while the key and the initialization value change frequently.

5.1.3. The data elements

The third section of security plan, the data elements and their properties are covered. Fig. 5 presents the structure and description of *Data element* section of *Security plan*. The listing

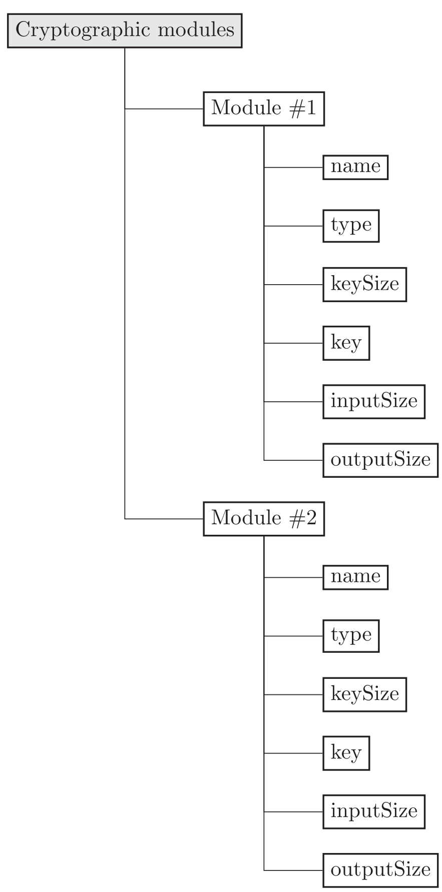


(a)

```
{
"$collection" : {
"name" : "Personnel",
"$encryption":{"$ref":"/AES-DET",
"key":"02468acebdf135790369cf258be147ad",
"iv":"2468ace0" }
},
"$fieldName":   {
"$encryption": {
"$ref":"/AES-DET",
"key":"0123456789abcdef0123456789abcdef",
"iv":"ffeeddcc"}
}
}
```

(b)

Fig. 3. The structure of a collection: (a) The chart outlines the structure of a collection containing the name of collection and name of all attributes which are considered as a meta-data, and should be protected with proper cryptographic module. (b) The description of a collection and security parameters in designed JSON based language. In this specific case the Advanced Encryption Standard in deterministic (AES-DET) mode with a 128-bit key and an *initialization vector* (IV) is assigned to encrypt the name of the collection and the fields name.

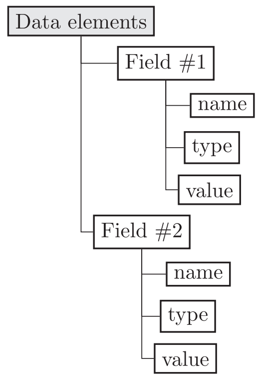


(a)

```
{
"OPE" : {
"properties" : {
"encryptionMethod" : {
"type" : "string",
"enum" : [ "OPE" ]  },
"keySize" : {
"type" : "integer",
"minimum" : 64,
"maximum" : 4096,
"default" : 128 },
"key" : {
"type" : "string",
"pattern" : "^([0-9a-fA-F]{2})+$" },
"inputSize" :  {
"type" : "integer",
"minimum" : 8,
"maximum" : 128,
"default" : 32 },
"outputSize" :   {
"type" : "integer",
"minimum" : 64,
"default" : 128 } },
"required" : [  "key", "encryptionMethod"
    ],
"additionalProperties" : false
}
}
```

(b)

Fig. 4. The structure and function of Cryptographic modules: (a) The *Security Plan* with the second section, the cryptographic module, expanded. The attributes included for each module are: name, type, key size, key, input and output size. (b) The OPE encryption including the cryptosystems and their attributes. The proxy applies these modules using the key-value pairs (KVP).

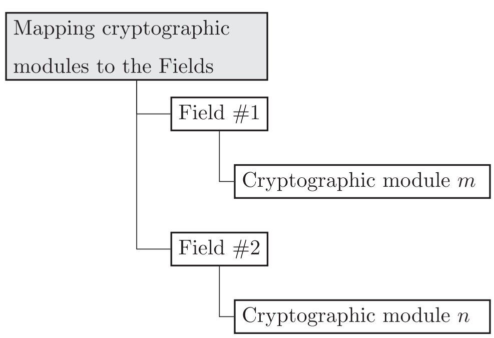


(a)

```
{
"id":{ "type":"integer" },
"name":{ "type":"string"  },
"salary"{ "type":"integer" },
"balance"{ "type":"integer" },
"ccn"{ "type":"integer" },
"ssn"{ "type":"integer" },
"email":{ "type":"string"  }
"required":["id","name","email","salary"]
}
```

(b)

Fig. 5. Structure and description of *Data element*: (a) The chart outlines the structure of *Data elements* containing attributes of data elements such as name, type and value for of collection and name and then introduces security parameters for each data element. (b) The data element section of a sample database which is represented in designed notation. A data item has 7 fields: id, name, salary, balance, ccn, ssn, and email. The id, name, email, and salary are required fields.



(a)

```
{
"id":{ "$ref": "#/definitions/
    ope128" },
"name":{ "$ref": "#/definitions/AES
    -DET" },
"email":{ "$ref": "#/definitions/
    AES-DET" },
"salary":{ "$ref": "#/definitions/
    ope256" },
"ssn":{ "$ref": "#/definitions/
    ope256" },
"ccn":{ "$ref": "#/definitions/
    ope256" },
"balance":{ "$ref": "#/definitions/
    ope256" }
}
```

(b)

Fig. 6. The structure and description of *Mapping cryptographic modules to the Data element*: (a) *Security plan* with the fourth section expanded. This section establishes a correspondence between the data fields and the cryptographic modules used to encrypt and decrypt the data fields. (b) The mapping section of the schema for a sample database with 7 fields. For example, the *id* and the *name* will be encrypted with *OPE* 128 bit and *AES-DET*, respectively.

displayed in Fig. 5b displays data elements and its *JSON* description for previous example. To ensure the desired level of security the security plan should provide the description of all sensitive data elements of database in third section of security plan.

5.1.4. Mapping cryptographic modules to the fields

The last section of security plan specifies all cryptographic modules for all sensitive data fields. Fig. 6 and the listing presented in Fig. 6b shows the mapping of the cryptographic modules and the corresponding *JSON* format for a sample application.

The method presented in this paper can be easily extended to the other *NoSQL* data models discussed in Section 2. Fig. 7 shows how this extension from the key-value pair to the document store model can be carried out.

5.2. Query and data validation

The proxy validates the data and the query as a JSON-formatted input with the reference security plan. Then the proxy enforces the crypto-primitives and generates new query following the NoSQL query semantics. During this process the proxy applies to each field the cryptographic modules. Finally, the proxy forwards the newly encrypted query/data to the *NoSQL* database server. Fig. 8 depicts the schema validation process.

For better illustration, consider listings depicted in Fig. 9a as an input data; after running validation process the output is generated (see Fig. 9b). The output of validation process is a single file which contains descriptive information for data and meta-data in designed format and ready to execute on the *SecureNoSQL*.

The output of validation process is a single file containing descriptive information for data and meta-data expressed in the

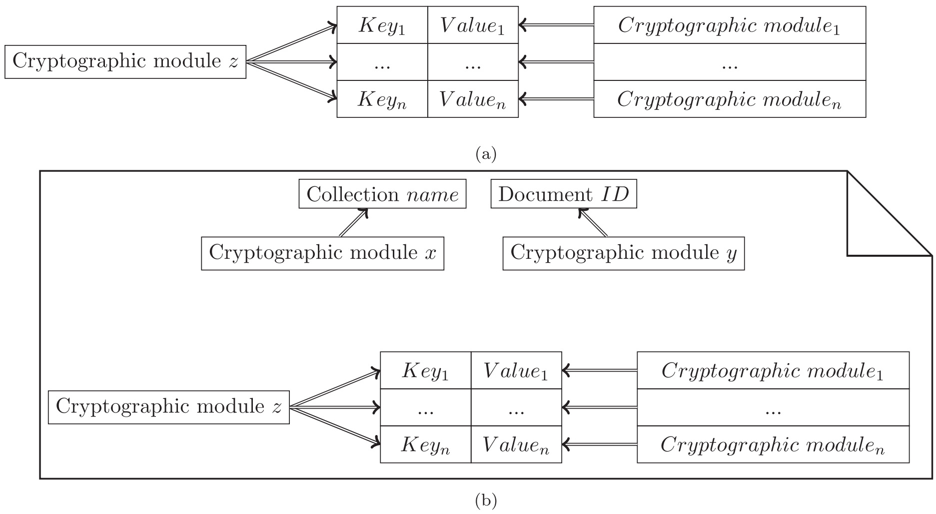


Fig. 7. *SecureNoSQL* applied to: (a) The key-value data model; $Key_1, \ldots, Key_n$ are all encrypted using the cryptographic module z while the corresponding values, $Value_1, \ldots, Value_n$ are encrypted with cryptographic modules 1, 2, ..., n, respectively. (b) The document store data model; the meta-data such as collection name encrypted as well as attributes with assigned cryptographic modules.

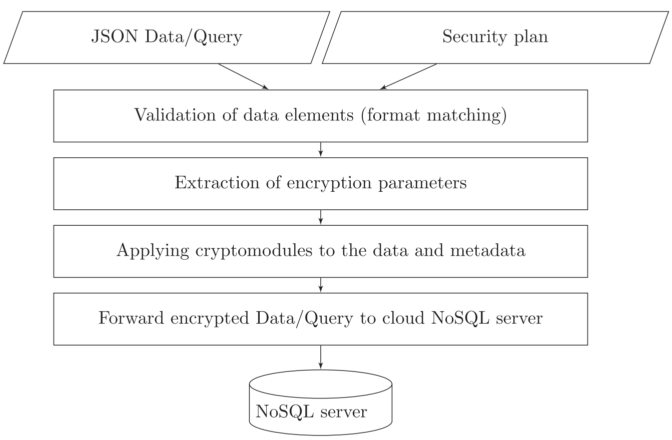


Fig. 8. The validation process of input data against security plan in the client side.

Table 1
The overhead of encryption for several encryption schemes.

Database	Plain	OPE64	OPE128	OPE256	OPE512
Size (MB)	170	430	508	662	1000

required format and ready to execute. The output of validation process for the example is illustrated in Fig. 9b. As it was noted earlier, the schema reflects the desired security level expressed by the security plan for the database. Table 1 shows the overhead for several parameters and crypto-primitives.

6. Processing queries on encrypted data

According to the proposed scheme, in order to process queries over encrypted data the queries should transfer to the encrypted version with respect to *security plan*, and this task is designed to be conducted in the proxy. The *security plan* discussed in Section 5, supplies the parameters of the cryptographic modules to be applied for the data elements involved in the query. Fig. 10 displays the processing and rewriting of a sample query.

For better understanding the query encryption, in Table 2 you can find some sample encrypted queries after enforcing *security plan*. As it can be seen, data elements and immediate values are encrypted; however, the output is consistent with NoSQL semantics.

7. Integrity of data/query/response

Integrity and confidentiality are two critical components of data security. Integrity refers to the consistency of the outsourced data. The proposed integrity verification algorithm in *SecureNoSQL* guarantees the integrity of data/queries (see Algorithm 1 and Fig. 11). Data owner first applies encryption scheme on the documents, and then calculates Hashed Message Authentication Code (HMAC) for each one of encrypted documents. A hash value of any given document is a fixed length of 512 bit and data owner concatenates a unique document identifier (ID) with hash value and stores the results in efficient structure like HashTable which has constant looks-up time $O(1)$. Next, data owner transfers the encrypted dataset to the cloud and sends HashTable containing hash values to the proxy. Once the proxy receives the query response from the server, it initiates the verification process to check the authenticity of the documents by recalculating the hash values. This process is illustrated in Fig. 11.

Algorithm 1. Document Integrity Verification Algorithm in the Proxy

Input: Plaintext query q from client application c_i

Output: Are the documents in the response authentic? Yes/No

$q_{enc} = Encrypt(q)$;

$q_{enc} \xRightarrow{forward}$ *to cloud database server*;

$R_{enc} \xLeftarrow{receive}$ *from cloud database server*;

repeat

  $H_d = HMAC(R_{enc}[i], key)$;

  if $(HashTable[user_i] \neq H_d)$ **then**

    return *false*

until *(There is a document in R_{enc})*;

return *true*;

In this configuration the data owners just trust the proxy(SecureNoSQL) and cloud servers are not trustworthy. Thus, a result of data integrity verification, all active attacks done by internal or external attacker will be detected by the proposed

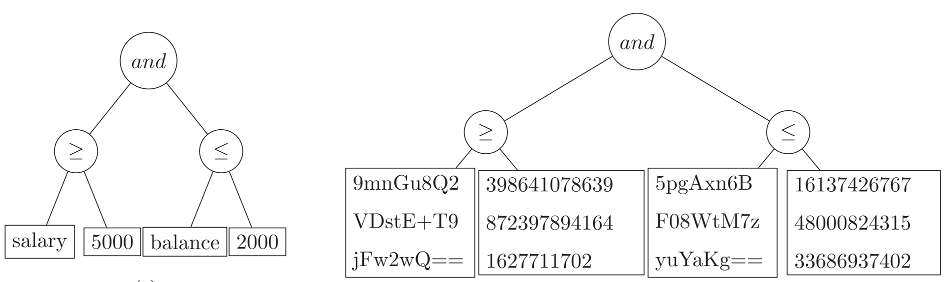
```
{
"id": 1,
"name":"Mohammad Ahmadian",
"email": "ahmadian@ucf.edu",
"salary":17000,
"ssn": 433042664,
"ccn":"47162552387",
"balance":1320
}
```

(a)

```
{
"id": {
"$encryption": {
"encryptionMethod": "op2128",
"key": "ADBDBC3B439DB495A81DA1BE56ACA" },
"value": 1 },
"name": {
"$encryption": {
"encryptionMethod": "AES-DET",
"key": "00112233445566778899aAbBcCdDeEfF" },
"value": "Mohammad Ahmadian" },
"email": {
"$encryption": {
"encryptionMethod": "AES-DET",
"key": "00112233445566778899aAbBcCdDeEfF" },
"value": "ahmadian@ucf.edu" },
"balance": {
"$encryption": {
"encryptionMethod": "ope256",
"key": "A75C644DF2E4EFE5328BB35E3C636" },
"value": 1320 }
}
```

(b)

Fig. 9. The security plan for the sample input: (a) The data element section of sample security plan. (b) The output of the *JSON* data validation for the sample database.



(a) (b)

Fig. 10. The query *db.customers.find({salary:{$gt:5000}, balance:{$lt:2000}})* received from an application. (a) The parsing tree of the query. (b) The cryptographic modules applied to the data elements according to schema definition.

approach. The message authentication code (MAC) is created by using the keyed Hash Message Authentication Code (HMAC) as rephrased in Eq. (2).

$$HMAC(K, document) = H((K \oplus okeyPad) \| H((K \oplus ikeyPad) \| document) \quad (2)$$

Where:

H represents the hash function $\oplus$ is the XOR operator

okeyPad is one-block-long outer pad **ikeyPad** is one-block-long inner key pad

Algorithm 2 presents the pseudo-code of the HMAC function for a block size of 64 bytes. The computed hash values with correspondent document's unique identifier can be stored in the form of key-value pair in a hash-table, thus allowing the proxy to carry the lookup in constant time during the verification process.

Algorithm 2. Keyed Hash Message Authentication Code (HMAC) generation

Input: Document: d, user key: k

Output: hash value

if $(length(key) > blocksize)$ **then**

  $key = hash(key);$

if $(length(key) < blocksize)$ **then**

  $key = key \,||\, [0x00 * (blocksize - length(key))];$

$okeyPad = [0x5c * (blocksize)] \oplus k;$

$ikeyPad = [0x36 * (blocksize)] \oplus k;$

return $hash(okeyPad||hash(ikeyPad \,||\, d));$

Table 2
Five sample queries and their corresponding encrypted version.

Q	Query	Encrypted query
1	db.customers.find({ssn:936136916})	db["k/IevnbanDMQHNkb9cRgUg=="].find({ "5pgAxn6BF08WtM7zyuYaKg==": 74172405478441908041711118833862143778})
2	db.customers.find({balance:{$gte: 5084610},balance:{$lte:9911843}})	db["k/IevnbanDMQHNkb9cRgUg=="].find({ "3iXpo2l8xZpW7J7TezFdeA==":{$gte: 402982988013604629517872370128473753}, "3iXpo218xZpW7J7TezFdeA=="{$lte: 785596355698717592780268633369454231}})
3	db.customers.aggregate([{$group: {_id:null,minBalance:{$min: "$balance"}}}])	db["k/IevnbanDMQHNkb9cRgUg=="].aggregate([{$group: {_id:null,EncMinBalance:{$min: "$3iXpo2l8xZpW7J7TezFdeA=="}}}])
4	db.customers.aggregate([{$group: {_id:null,maxBalance:{$max: "$balance"}}}])	db["k/IevnbanDMQHNkb9cRgUg=="].aggregate([{$group: {_id:null,EncmaxBalance:{$max: "$3iXpo2l8xZpW7J7TezFdeA==" }}}])
5	db.customers.find({$or:[{Salary: {$gt:516046}},{balance: {$lt:285462}}]})	db["k/IevnbanDMQHNkb9cRgUg=="].find({$or:[{ "9mnGu8Q2VDstE+T9jFw2wQ==":{$gt: 40994186216785746613193244129885849}},{ "3iXpo2l8xZpW7J7TezFdeA==": {$lt:22657430453144634679791167652174833}}]})

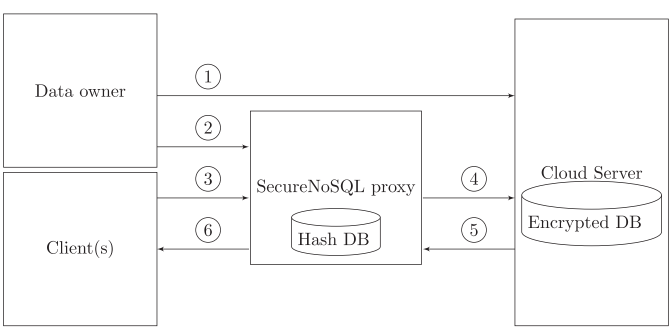


Fig. 11. (1) Data owner transfers the encrypted database to the cloud server. (2) Data owner sends the Hash database to proxy. (3) Clients send plain queries to the proxy. (4) The proxy translates queries to the encrypted version, and forwards them to the cloud server. (5) The cloud server returns the query response set. (6) The proxy runs a hash verification process on the query response set, and then based on the result either forwards to the decrypted response or reports integrity violation to the client.

8. Results and discussion

The response time of a query to an encrypted *NoSQL* database has several components:

1. the time to encode the query in *JSON* format;
2. the time to encrypt and decrypt the data;
3. the communication time to/from the server;
4. the database response time.

For our experiments we first created a sample database with one million records and then determined the overhead of searching an encrypted database. To do so we measured the database response time for queries when the records were unencrypted versus when records were encrypted. Then, we measured the encryption and the decryption time for different sizes of the ciphertext. We wanted to isolate the different components of the response time dominated by the communication time.

The environment used for testing was set up on the Linux operating system. We chose *MongoDB* (Dede, Govindaraju, Gunter, Canon, & Ramakrishnan, 2013), classified as a *NoSQL* document store database 3.0.2. The random data generator in *JS*, *PHP*, and *MySQL* format was generated by using a tool (Keen, 2016) to generate a one million record plaintext data set. Each record had seven different data fields including *name*, *email*, *salary*, as shown in Listing 9b.

We applied OPE 64, 128, 256 and 512 bit to numeric data type, and the *AES-DET 128* bit for the string data type of the plaintext data set and generated four encrypted data sets of one million records each. Finally, we uploaded the five datasets and created five *MongoDB* databases, one with the unencrypted data, and four with the encrypted data. Once the *MongoDB* databases were created we run several types of queries including equality, greater than, less than, greater than or equal to, less than or equal to, and OR logical operations.

The experiments to measure the query time must be carefully designed. To construct average query processing time each experiment has to be carried out repeatedly. We noticed a significant reduction of database management response time after the first execution of a query, a sign that *MongoDB* is optimized and caches the results of the most recent queries. A solution is to disable the cache, or if this is not feasible, to clear the cache before repeating the query. Another important observation is that modern processors have a 64-bit architecture and are optimized for operations on 64-bit integers. For three of the five types of queries, *Q2* (Range query), *Q3* (equality), and *Q4* (logical), database response time is slightly shorter for the encrypted database than for the unencrypted one

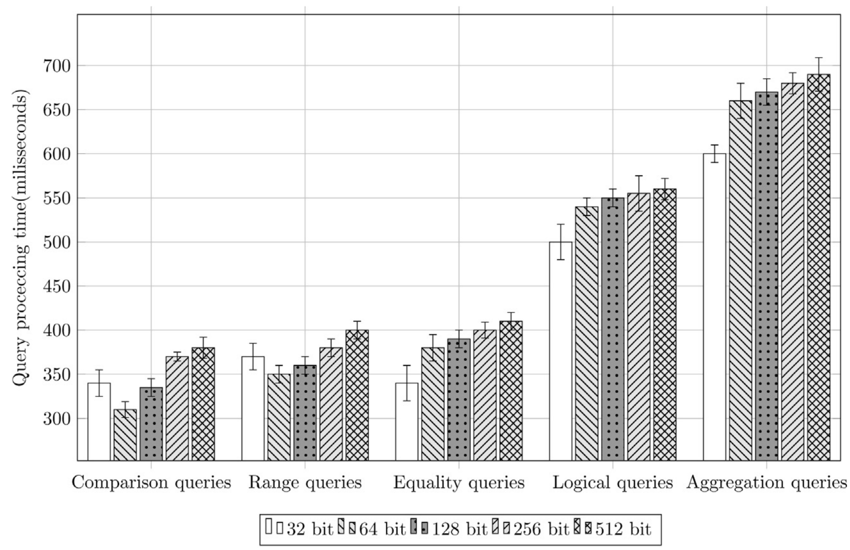


Fig. 12. The query processing time in milliseconds (ms) for the unencrypted database and for the encrypted databases when the 32-bit keys are encrypted as 64, 128, 256 and 512-bit integers.

Table 3
The query processing time in milliseconds (ms) for the plaintext and for the ciphertext. 32-bit plaintext integers are encrypted as 64, 128, 256 and 512-bit integers. The record count gives the number of records retrieved by each one of the five types of queries, *Q1*–*Q5*.

Query type	Number of matching record(s)	32-bit plaintext	64-bit ciphertext	128-bit ciphertext	256-bit ciphertext	512-bit ciphertext
Q1: Comparison	461,688	340	310	355	370	380
Q2: Equality	1	340	380	390	400	410
Q3: Range	991,225	370	350	360	380	400
Q4: Logical	551,380	500	540	550	555	560
Q5: Aggregation	1	600	660	670	680	690

when the keys are 32-bit integers. A plausible explanation for this is most likely related to the cache management.

The results reported in Table 3 and in Fig. 12 show the database response time for the five *MongoDB* experiments. Each query was carried out 100 times with disabled query cache and the average query response in milliseconds was calculated. We also measured the encryption and the decryption time and the results are reported in Fig. 13. The measurement process was automated, and it was running under the control of a script which generated the data and reported the processing time.

Our measurements show that the response time of the *NoSQL* database management system to encrypted data depends on the

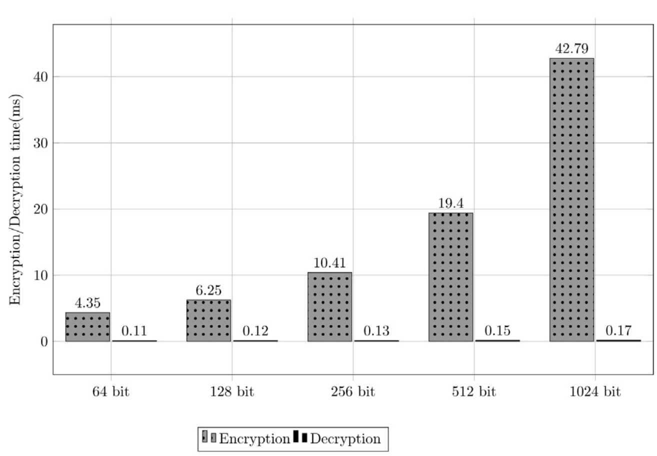


Fig. 13. Execution time of the OPE module when the key is encrypted as 64, 128, 256, 512, and 1024 bit.

type of the query. The shortest and longest database response times occur for Q1 (comparison) and Q5 (aggregated queries), respectively; for these two extremes the time for the unencrypted database was almost double, but the time for encrypted databases increases only by 70–80%. As expected, the query processing type for a given type of query increases, but only slightly, less than 5% when the key length increases from 64, to 128, 256, and 512 bit.

The OPE encryption time increases significantly with the size of the encryption space; it increases almost tenfold when the size of the encrypted output increases from 64-bit to 1024-bit and it is about 10 ms for 256-bit. The decryption time is considerably smaller; it increases only slightly from 0.11 ms to 0.17 when the size of the encrypted key increases from 64-bit to 1024 bit.

Secure proxy is an important element for the proposed architecture; therefore, the potential attacks that could affect the proxy, also should be taken into consideration. In general, two major possible attacks on proxy are Denial of Service (DoS) and unauthorized access. In DoS attack, the attacker sends so many network traffic to the proxy, that the system is not capable of processing within the expected time frame. Successful DoS attacks can turn the proxy to a bottleneck of the system. In unauthorized access attacks, attackers use a proxy to mask their connections while attacking the different targets.

Several solutions exist for improving the security of proxy against DoS attacks and reducing the consecutive impacts, including blocking the undesired packets or using multiple proxies with load balancers. Moreover, for prevention of unauthorized access attacks, it is required to use best fit authorization to access the proxy. User authentication based on group membership with different authorizations are the best practical solutions.

9. Conclusions and future work

Though the OPE encryption scheme has known security vulnerabilities it can be very useful for *NoSQL* database query processing for the data models discussed in Section 2. While the *key* is encrypted using OPE, the other fields of a record can be encrypted using strong encryption, thus reducing the vulnerability of the data attacks. Strong encryption of the value fields could increase the encryption time but will have little effect on the decryption time.

An important observation is that increasing the size of the co-domain of the OPE mapping function from 2^{64} to 2^{128}, 2^{256}, and to 2^{512} results in an increase of database response time up to 5%, except for *Q*3-type queries when the increase is significant. The penalty for using encrypted, rather than unencrypted NoSQL databases such as *MongoDB* is less than 5% for *Q*2, *Q*4, and *Q*5 which is considered to be relatively small. Moreover, the overall query response time is dominated by the communication time. The secure proxy is a critical component of the system. The proxy is multi-threaded and its cache management is non-trivial. The management of the security attributes is rather involved. On the other hand, a proxy integrated in the client-side software can be lightweight and considerably simpler. We are currently implementing the two versions of proxy. Experimental results for multiple large datasets with up to one million documents show that SecureNoSQL is rather efficient. Our approach can be extended to a multi-proxy structure for Big Data applications. We are now implementing a sophisticated mechanism for maintaining consistency of hash values database in the proxies datasets based on the PAXOS algorithm (Lamport, 2001; Marinescu, 2013).

Acknowledgment

We thank Victor Shoup from New York University for NTL C++ library used to manipulating arbitrary length integers.

References

Ahmadian, M. (2017). SECURE QUERY PROCESSING IN CLOUD NoSQL. In *2017 IEEE international conference on consumer electronics (ICCE) (2017 ICCE)*

Ahmadian, M., Khodabandehloo, J., & Marinescu, D. (2015). A security scheme for geographic information databases in location based systems. In *IEEE SoutheastCon* (pp. 1–7). http://dx.doi.org/10.1109/SECON.2015.7132941

Ahmadian, M., Paya, A., & Marinescu, D. (2014). Security of applications involving multiple organizations and order preserving encryption in hybrid cloud environments. In *IEEE international conf. on parallel distributed processing symposium workshops (IPDPSW)* (pp. 894–903). http://dx.doi.org/10.1109/IPDPSW.2014.102

Bajaj, S., & Sion, R. (2014). Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *IEEE Transactions on Knowledge and Data Engineering*, *26*, 752–765.

Boldyreva, A., Chenette, N., Lee, Y., & O'neill, A. (2009). Order-preserving symmetric encryption. In *Annual international conference on the theory and applications of cryptographic techniques* (pp. 224–241). Springer.

Brakerski, Z., & Vaikuntanathan, V. (2011). Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Annual cryptology conference* (pp. 505–524). Springer.

Canim, M., Kantarcioglu, M., Hore, B., & Mehrotra, S. (2010). Building disclosure risk aware query optimizers for relational databases. *Proceedings of the VLDB Endowment*, *3*, 13–24.

Cash, D., Hofheinz, D., Kiltz, E., & Peikert, C. (2012). Bonsai trees, or how to delegate a lattice basis. *Journal of Cryptology*, *25*, 601–639.

Cash, D., Jaeger, J., Jarecki, S., Jutla, C. S., Krawczyk, H., Rosu, M.-C., et al. (2014). Dynamic searchable encryption in very-large databases: Data structures and implementation. *IACR Cryptology ePrint Archive*, *2014*, 853.

Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.-C., & Steiner, M. (2013). Highly-scalable searchable symmetric encryption with support for Boolean queries. In *Advances in cryptology—CRYPTO 2013* (pp. 353–373). Springer.

Cattell, R. (2011). Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, *39*, 12–27.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., et al. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, *26*, 4.

Cheon, J. H., Kim, M., & Kim, M. (2016). Optimized search-and-compute circuits and their application to query evaluation on encrypted data. *IEEE Transactions on Information Forensics and Security*, *11*, 188–199.

Chow, R., Golle, P., Jakobsson, M., Shi, E., Staddon, J., Masuoka, R., et al. (2009). Controlling data in the cloud: Outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM workshop on cloud computing security* (pp. 85–90). ACM.

Colombo, P., & Ferrari, E. (2015). Privacy aware access control for big data: A research roadmap. *Big Data Research*, *2*, 145–154.

Dede, E., Govindaraju, M., Gunter, D., Canon, R. S., & Ramakrishnan, L. (2013). Performance evaluation of a MongoDB and hadoop platform for scientific data analysis. In *Proceedings of the 4th ACM workshop on scientific cloud computing* (pp. 13–20). ACM.

Ducas, L., & Micciancio, D. (2015). FHEW: Bootstrapping homomorphic encryption in less than a second. In *Annual international conference on the theory and applications of cryptographic techniques* (pp. 617–640). Springer.

Galiegue, F., & Zyp, K. (2013). JSON schema: Core definitions and terminology. *Internet Engineering Task Force (IETF)*.

Gantz, J., & Reinsel, D. (2012). The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, *2007*, 1–16.

Gentry, C. (2009). *A fully homomorphic encryption scheme (Ph.D. thesis)*. Stanford University.

Gorbunov, S., Vaikuntanathan, V., & Wee, H. (2013). Attribute-based encryption for circuits. In *Proc. of the forty-fifth annual ACM symposium on theory of computing* (pp. 545–554). http://dx.doi.org/10.1145/2488608.2488677

Halevi, S., & Shoup, V. (2014). Algorithms in HElib. In *International cryptology conference* (pp. 554–571). Springer.

Islam, M., & Islam, M. (2014). An approach to provide security to unstructured big data. In *8th international conf. on software, knowledge information management and applications (SKIMA)* (pp. 1–5). http://dx.doi.org/10.1109/SKIMA.2014.7083392

Keen, B. (2016). *benkeen/generatedata*. https://github.com/benkeen/generatedata

Kerschbaum, F. (2015). Frequency-hiding order-preserving encryption. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security* (pp. 656–667). ACM.

Lamport, L. (2001). Paxos made simple. *ACM Sigact News*, *32*, 18–25.

Liang, K., Susilo, W., & Liu, J. (2015). Privacy-preserving ciphertext multi-sharing control for big data storage. *IEEE Transactions on Information Forensics and Security*, *10*, 1578–1589. http://dx.doi.org/10.1109/TIFS.2015.2419186

Lin, C.-H., Tsai, S.-H., & Lin, Y.-P. (2014). Secure transmission using MIMO precoding. *IEEE Transactions on Information Forensics and Security*, *9*, 801–813. http://dx.doi.org/10.1109/TIFS.2014.2309211

Marinescu, D. C. (2013). *Cloud computing: Theory and practice*. Newnes.

Mavroforakis, C., Chenette, N., O'Neill, A., Kollios, G., & Canetti, R. (2015). Modular order-preserving encryption, revisited. In *Proc. of the 2015 ACM SIGMOD international conf. on management of data* (pp. 763–777). http://dx.doi.org/10.1145/2723372.2749455

Micciancio, D., & Regev, O. (2009). Lattice-based cryptography. In *Post-quantum cryptography*. pp. 147–191. Springer.

Popa, R. A., Redfield, C. M. S., Zeldovich, N., & Balakrishnan, H. (2011). CryptDB: Protecting confidentiality with encrypted query processing. In *Proc. of the twenty-third ACM symposium on operating systems principles* (pp. 85–100). http://dx.doi.org/10.1145/2043556.2043566

Silver-Greenberg, J., Goldstein, M., & Perlroth, N. (2014). JPMorgan chase hack affects 76 million households. *New York Times*, *2*.

Sivasubramanian, S. (2012). Amazon dynamoDB: A seamlessly scalable non-relational database service. In *Proc. of ACM SIGMOD international conf. on management of data* (pp. 729–730). http://dx.doi.org/10.1145/2213836.2213945

Song, D. X., Wagner, D., & Perrig, A. (2000). Practical techniques for searches on encrypted data. In *2000 IEEE symposium on security and privacy 2000. S&P 2000. Proceedings* (pp. 44–55). IEEE.

Stonebraker, M. (2010). SQL databases v. NoSQL databases. *Communications of the ACM*, *53*, 10–11. http://dx.doi.org/10.1145/1721654.1721659

Tankard, C. (2012). Big data security. *Network Security*, *2012*, 5–8.

Tu, S., Kaashoek, M. F., Madden, S., & Zeldovich, N. (2013). Processing analytical queries over encrypted data. *Proceedings of the VLDB Endowment*, *6*, 289–300.

Weiss, N. E., & Miller, R. S. (2015). The target and other financial data breaches: Frequently asked questions. In *Congressional research service, prepared for members and committees of congress, February (Vol. 4)* (p. 2015).

Xiao, L., & Yen, I.-L. (2012). Security analysis for order preserving encryption schemes. In *2012 46th annual conference on information sciences and systems (CISS)* (pp. 1–6). IEEE.

Xu, L., Jiang, C., Wang, J., Yuan, J., & Ren, Y. (2014). Information security in big data: Privacy and data mining. *IEEE Access*, *2*, 1149–1176. http://dx.doi.org/10.1109/ACCESS.2014.2362522

Xu, L., Zhang, X., Wu, X., & Shi, W. (2015). ABSS: An attribute-based sanitizable signature for integrity of outsourced database with public cloud. In *Proceedings of the 5th ACM conference on data and application security and privacy* (pp. 167–169). ACM.

Yu, X., & Wen, Q. (2010). A view about cloud data security from data life cycle. In *International conf. on computational intelligence and software engineering (CiSE)* (pp. 1–4). http://dx.doi.org/10.1109/CISE.2010.5676895