CloudPDB: A Light-weight Data Privacy Schema for Cloud-based Databases

Mehdi Bahrami[1] and Mukesh Singhal[2]

Cloud Lab
University of California, Merced
[1] Senior Member of IEEE, MBahrami@UCMerced.edu
[2] IEEE Fellow, MSinghal@UCMered.edu

***Abstract*—Cloud computing paradigm offers an opportunity to in-house IT departments to establish their services on the cloud with minimum investment, and lower maintenance cost. This opportunity includes database services that allow a massive amount of data to be stored on the cloud. However, outsourcing data to the cloud may expose users' data to the cloud vendor, or the vendors' partners. Although encryption schemes, such as *AES* can be used against an untrusted cloud vendor, or an adversary, those schemes add additional computation overheads. In this paper, we proposed a light-weight data privacy schema for cloud based databases (*CloudPDB*) that scrambles data on each selected bucket (multiple records, or fields) of a database. The proposed schema uses a pre-defined database of a chaos system to store data on the cloud that protect data against an adversary inside or outside of an untrusted cloud vendor. We implemented the proposed schema on a well-known standard database benchmark, *TPC-H* with different query sizes. We ran several queries to assess the performance of the proposed schema. The evaluation shows that the proposed schema provides a better performance over other well-known encryption methods. In addition, we assess the security level of the proposed schema.**

Keywords— Cloud Computing; Database; Data Privacy; Data Security;

I. INTRODUCTION

Cloud computing provides elastic storage resources over the Internet that allows users to pay for their resource usage based on pay-per-use model. It enables users to scale their storage on-demand [1]. However, the cloud computing paradigm is a form of Internet-based service that requires users to outsource their data, and their applications to the cloud vendor. Most of the time, cloud vendors are not fully trusted by the users, and are vulnerable to users' data privacy violation by the cloud vendor.

Users have several options to use the cloud. First, the users may employ a hybrid-cloud [2] that allows them to outsource sensitive data to their private storage, and uses a public cloud for their non-sensitive data. This option may not be a practical solution due to the complexity of the system integration [2] and network security issues. Another option is to encrypt user data before outsourcing it to an untrusted cloud vendor. However, most well-known encryption methods, such as *AES* [8] are expensive because they increase computation time due to encryption/decryption of data during query processing. The third option is light-weight data security methods that secure data based on some conditions which are discussed in *Section II*. In this paper, we are interested in this option that allows users to protect their outsourced data with minimal computation overheads. The final option, is outsourcing data without considering users data privacy.

Several studies [4, 5, 6, and 7] have been conducted to secure a database with different encryption schemas. Although an encrypted database causes additional computation overheads to run queries, it enables users to protect their outsourced data, in particular sensitive information. In this paper, we assume that users are willing to protect their outsourced database on an untrusted cloud vendor. We assume that the vendor must not be able to access the database, and users may be able to access the database with minimal computation overheads.

Our primary contributions are as follows:

- We propose an efficient light-weight schema that includes several components and algorithm, to securely outsource data to an untrusted cloud;
- We implement, and assess the performance of the proposed schema, and compares the performance of the light-weight data privacy method to a well-known encryption method, *AES* [8];
- We analyze the security level of the proposed schema.

The rest of the paper is organized as follows. The next section introduces some background related this study. *Section III*, introduces the proposed schema, and its various components. *Section IV* presents a security analysis of the proposed schema. The experimental setup of the implementation of the proposed schema, and the experimental results are discussed in *Section V*, and *VI*, respectively. The related work is discussed in *Section VII,* and finally, *Section VIII* concludes this paper, and presents the future work of this study.

II. BACKGROUND

In our previous study [3], we proposed a light-weight data privacy method (*DPM*) that scrambles chunks of data based on a chaos system. The *DPM* uses the following equation in a chaos system that generates sets of distributed random numbers.

$$\psi_i: P_{k+1} = \mu P_k(1 - P_K) \quad (1)$$

where $P \in \{0,1\}$ and μ are two initial parameters of this equation, and i is the index of each set of ψ.

In another words, ψ provides a set of numbers that does not allow an adversary who knows P_l to predict the future numbers, P_m where $m > l$. The content of each chunk (a set of bits or bytes) of an original data (input message) can be scrambled based on ith set of scrambled addresses in ψ_i which relocates the content of the original data. ψ_i generates repeated numbers,

and *DPM* uses an algorithm [3] to remove collision in addresses, and to cover all addresses of a given chunk of data.

The advantage of *DPM* is its time complexity. On one hand, a user scrambles a chunk of data with $O(1)$ time complexity, and on the other hand, an adversary needs $O(2^n)$ computation time to retrieve the original data from scrambled data when he does not know the initial parameters, where n is the size of each chunk. *DPM* scramble the content of an original bit to avoid adding computation overhead, and it has the following two important security parameters.

The size of n: The size of each chunk, n is important to *DPM* to provide a sufficient level of security. For instance, *DPM* can be secured with $n > 120$ based on current computational capabilities. If an adversary runs an exhaustive search on the scrambled data, he needs to perform $O(2^{120})$ computational steps to retrieve the original data. In implementation work of the proposed schema which is described in *Section VI*, we consider each bit as an input that allows us to increases the size of n. If we consider a field of a record as an input, it could be small enough to retrieve the original data fast. We can combine multiple field(s) of a record as a chunk of the original data, and we can consider bits of the chunks as an input of *DPM* in order to increase the size of n. For instance, a Unicode character in *Microsoft SQL Server* has 2 *Bytes*, and for an adversary to perform an exhaustive search over a truly scrambled field (see the next parameter) with 20 characters length requires $O(2^n)$ computation steps, where $n = 10\ chars * 2\ Bytes * 8\ bits$.

The number of repeated initial parameters: *DPM* needs to run with different initial parameters for each chunk of data (message) in order to be secure. The proof of this claim is given in *Section IV*.

We can generate different set of ψ for each original data but it adds additional computation overheads. We can precompute ψ offline, and store them on a database in order to eliminate online computation overheads. A detail of implementation of these parameters is discussed in the next section.

III. The Proposed Schema

The proposed schema stores scrambled data with minimal computation overheads in the database. The database is accessible only by the owner of the database, who has a key. In case of database compromise as whole, or access to database by authorize or unauthorized users without a key, the data on the database cannot be retrieved. The cloud vendors also cannot access the database because only the owner has the key that can reconstruct the scrambled data.

The proposed schema for a cloud-based database is illustrated in Figure 1. Each submitted query from a user will go through the *proxy server* in order to scramble data prior to running the query operation (*insert, update or select*) on the database (*SecureDB*). The scrambled data is stored in *SecureDB*. The *proxy server* uses *MapDB* to access different *set of* ψ which is defined in Equation 1. We can remove *MapDB* by adding a ψ generator function that produces several sets of ψ. The *proxy server* uses *KeyDB* to store a user's keys for a record in *SecureDB*.

The main components of the schema are as follows.

SecureDB: This database stores scrambled data. Authorized and unauthorized users including cloud vendor administrators are not able to retrieve the original data from this database without knowing the keys that are stored in *KeyDB*. Only submitted transactions from *Proxy Server* which has access to *KeyDB,* is able to retrieve the original data. Even if this database is compromised on the cloud, an internal and an external adversary cannot retrieve the original data.

KeyDB: This database stores an index to ψ which is located in *MapDB*, for each record in *SecureDB*. This database is updated with an *insert*/*update* operation, and it is used for reconstructing a record of *SecureDB* by providing ψ for the corresponding record. The *KeyDB* can be used locally in order to protect *SecureDB* from an untrusted cloud vendor.

MapDB: This is an optional database that collects a set of predefined ψ in order to avoid adding runtime computation overhead for generating ψ with different initial parameters. For instance, Table I shows a definition of *Customer's* table with 5 fields, fields' types, and the size of each field (Bytes). If we consider the combination of all fields as an input to the scramble process, we need 2,272 bits (284 Bytes) to be scrambled for this table. The join of all fields as an input increases computation time against an adversary to retrieve the original data from scrambled data. In this example, *MapDB* stores different shuffle addresses from the first bit to 2,272 by defining different initial values of μ and P_0 which is discussed

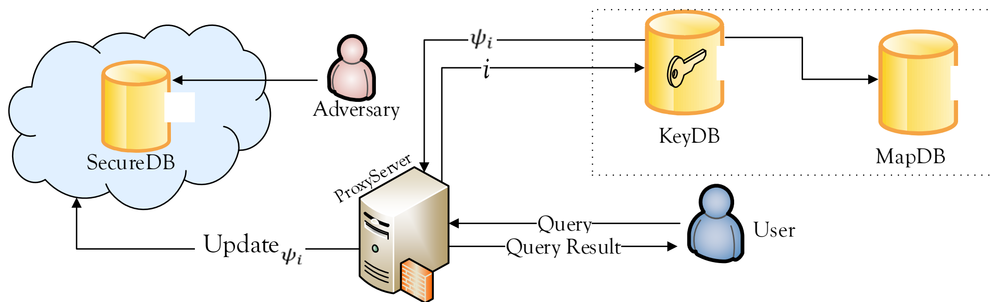


Figure 1. The proposed schema

previously in Equation 1. The *proxy server* uses one record of *MapDB* (shuffle addresses) to scramble and insert data to *SecureDB*. Then, the proxy server stores the record number of inserted data and its correspondence shuffle addresses from *MapDB* into *KeyDB,* that allows the *proxy server* to retrieve data later by using this information. *MapDB* can be used for several *SecureDBs*, on multiple clouds because this database can protect each *SecureDB* against an adversary from each cloud.

Table I. *The definition of a customer table*

Customer Key	Name	Address	National Key	Phone
Integer	nchar(10)	nchar(64)	Integer	nchar(64)
4 Bytes	20 Bytes	128 Bytes	4 Bytes	128 Bytes

MapDB can be updated periodically in an offline mode (similar to database indexing) in order to remove online computation overheads. For instance, the database can be updated with adding sets of ψ based on the number of used ψs as a threshold parameter.

Proxy Server: This server allows a user to retrieve, update, or insert data to *SecureDB*. It runs *DPM* on each submitted user's query. Each user's operation, such as *Insert*, *Update* or *Select*, needs to be submitted to the *Proxy Server*. If a new record needs to be added to the database, *proxy server* assigns an index of a ψ to the record, and then, it scrambles the record based on the assigned ψ, and finally the index is stored in *KeyDB* for future record retrieval.

Algorithm I, shows the insert procedure in the proposed schema that uses a user's key and the input record to insert data into *SecureDB*.

Algorithm I. *Insert procedure*

1: i = ***NewKey*** *(Key)*
2: ψ_i =***Map(i)***
3: *NewScrambledRec* =$\boldsymbol{Sc}_{\psi_i}(input)$
4: *Rec#* = ***Insert(****NewScrambledRec****)***
5: ***UpdateKeyDB(****i, Table, Rec#****)***

First, the procedure stores an index of ψ in i, and it stores ith set of shuffle addresses from *MapDB* in ψ_i *(step 2)*. Then, it scrambles the user's input record (*step 3*), and it inserts the scrambled data into *SecureDB* (*step 4*), and stores the record number in *Rec#*. Finally, it updates *KeyDB* with i (the corresponding ψ of the record), the record number, and the name of table.

The *proxy server* uses the record number, and its corresponding ψ from *MapDB* to reconstruct a record when it needs to retrieve or update a record.

IV. Security Analysis

A schema has a perfect secrecy, if it can pass the following conditions.

(i) The adversary cannot learn about two scrambled records, r_i and r_j when he knows a scrambled data, $\mathfrak{s}$;

(ii) The chaos system generator has perfect secrecy.

For the first condition, each record of a table of *SecureDB* in the proposed schema needs to be scrambled with different initial parameters, in order to avoid similarity between scrambled records as follows.

$$\exists i \in \psi_i \text{ and } \exists k \in SecureDB \mid SC(\psi_i, r_k) = \mathfrak{s}_i \quad (2)$$

$$\exists j \in \psi_j \text{ and } \exists l \in SecureDB \mid SC(\psi_j, r_l) = \mathfrak{s}_j \quad (3)$$

$$\forall i, j \in \psi \text{ such that } \mathfrak{s}_i \neq \mathfrak{s}_j \text{ where } i \neq j \quad (4)$$

where SC is the scramble function, ψ_i *and* ψ_j are two different sets of shuffle addresses, and r_k and r_l are two different kth and lth records of *SecureDB*.

$$\forall \text{i}, \mathfrak{s}: \Pr[SC(\psi_i, r_k) = \mathfrak{s}] = \frac{\#r \in \text{Z such that } \text{SC}(\psi_i, r_k) = \mathfrak{s}}{|\text{Z}|} \quad (5)$$

where r is a records in *SecureDB*.

In other words, the proposed schema uses different $\psi's$ which are defined with different initial parameters to prevent an adversary from learning about two original records by knowing their scrambled data. ■

For the second condition, ψ must provide a uniform distribution of addresses in ψ_i for all entries of *n* bits as follows:

$$P: U \to [0,1] \text{ such that } \sum_{x \in U} P(x) = 1 \quad (6)$$

where $U = \{0,1\}^n$.

$$\forall x \in U: P(x) = \frac{1}{|U|} \quad (7)$$

In this case, the generator must produce different addresses with a uniform probability. As previously dicussed in *Section II*, the generator provides scrambled addresses in each ψ, which is stored in *MapDB*. *DPM* uses a set of shuffle addresses in ψ to scramble data. If *DPM* provides the same probability for each scrambled address in ψ, it must show the difference between the original addresses, and the scrambled addresses are not the same, and *DPM* must not show any relation between addresses. The Figure 2 illustrates a statistical model of the first 100 differences between the original addresses and the scrambled addresses in Equation 1 with the initial parameters of $P_0 = 0.999$, $\mu = 3.684$ for the length of 921 bits (n). In this figure, *X-axis* represents the address of the original bit and *Y-axis* represents the difference between the original address and the final address in the scrambled bits. The result shows that *DPM* scrambles data with a uniform distribution with different differences that does not allow an adversary to find a patern between scrambled addresses. ■

In addition, more security analysis have been conducted against *DPM* which is discussed in [3].

As shown in Figure 2, there is no pattern between scrambled addresses that allows an adversary to predict the addresses. For

instance, if an adversary knows the first bit moves to 13th bit when it is scrambled, still he cannot predict that the second bit moves to 48th address, or 3rd bit moves to 180th bit.

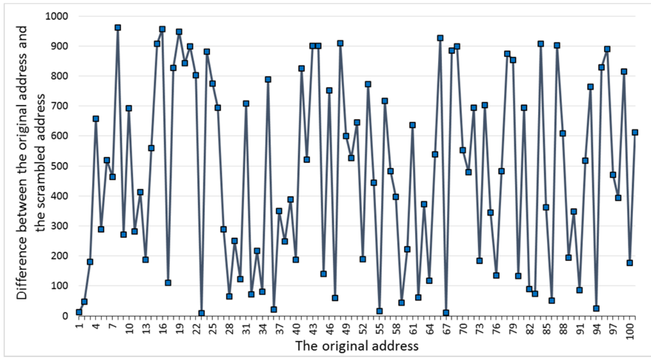


Figure 2. The difference between the original address and the scrambled address

V. EXPERIMENTAL SETUP

We conducted an experiment based on the proposed schema. We used *TPC-H* [9] which is a standard database benchmark with the scale of 1 GB. We ran different queries on *Customer* table. Each submitted query went through the *proxy server* that ran *DPM* and *AES* encryption separately in order to compare the performance of both methods on the proposed schema. We use *ADO.Net* [10] at client side to retrieve and bind data from the database. *DPM* and *AES* encryption were implemented as a class [10] and written in C#.Net version 4.5, and executed on a PC with CPU Intel Core i7 with 8 GB RAM.

VI. EXPERIMENTAL RESULTS

Figures 3 and 4 show the experimental results for the performance of the security methods (*AES* and *DPM*) on the proposed schema, and Figure 5 shows the data binding latency for different range of queries' responses.

In Figure 3, *X-axis* represents the number of the fields which were requested by a user's query, and *Y-axis* represents the total response time (millisecond) of *AES* encryption, and *DPM* on the proposed schema. Figure 3.a shows the total response time for 22 queries with a small query range from 9 fields to 9,000 fields with the increase rate of 450 fields for each next query. Figure 3.b shows the total response time for 9 queries with a larger query range from 9 fields to 81,000 fields with the increase rate of 9,000 fields for the next query. As shown in these figures, *DPM* provides superior performance over *AES* encryption. In particular, the response time difference between *AES* and *DPM* increases for the larger queries. Figure 4 shows the response time difference between *AES* and *DPM* for the query range of 9 fields to 81,000 fields. In this figure, *X-axis* represents the number of requested fields for a given query, and *Y-axis* represents the performance difference between *AES* and *DPM*. For instance, as shown in this figure, *DPM* saves 2,909 milliseconds (~3 seconds) computation time for a database management system (*DBMS*) over *AES* for a query with a request of 54,000 fields.

In another evaluation, we considered data binding latency

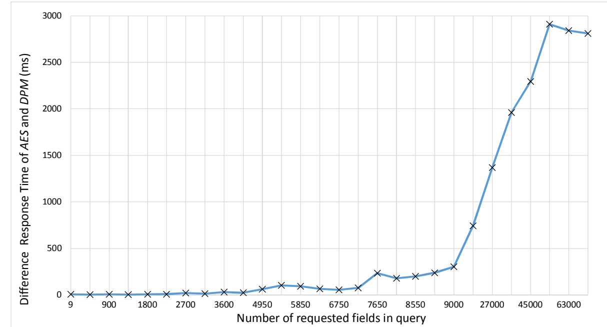
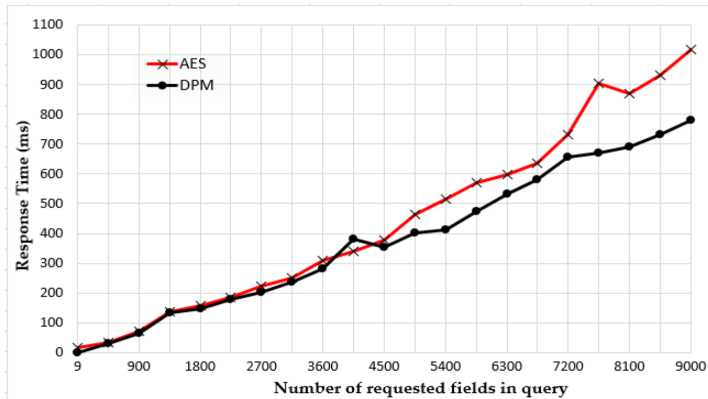


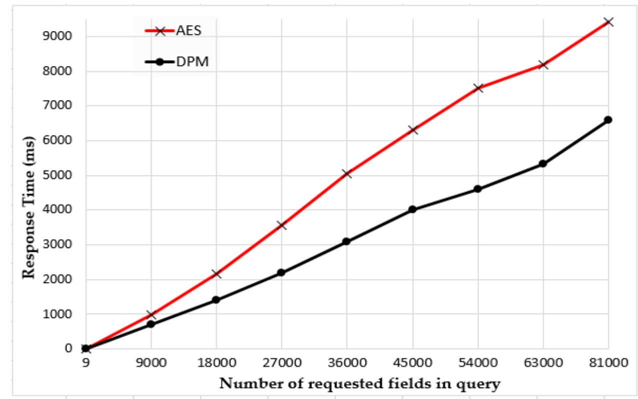
Figure 4. The response time difference between *AES* and *DPM*

which assesses the response time of data binding (retrieving data from the query's results to the client objects*)*. Figure 5 shows a comparison of data binding latency with different range of queries. In this figure, the client's objects need additional computation to fetch data that causes an additional computation overheads for the first query. The results show that *DPM* not only provides better performance on computation time as described in Figure 3, it also provides an efficient computation time for data binding.

In addition, some studies on databases, such as *CryptDB* [4] show that queries can be executed over encrypted database

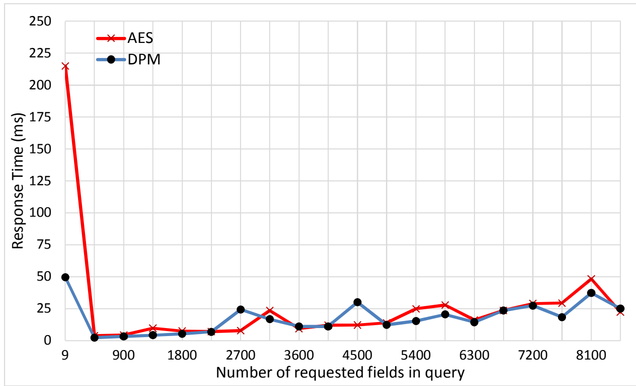


(a)

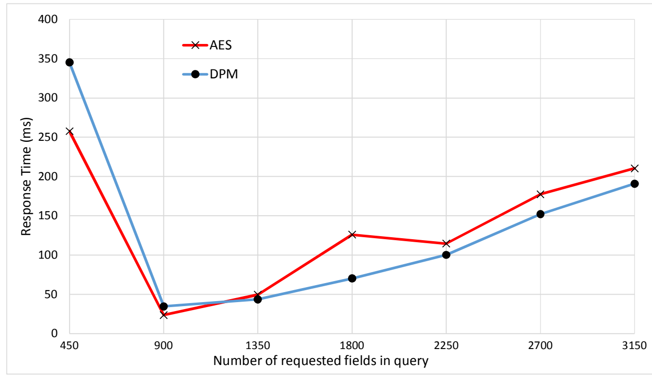


(b)

Figure 3. A comparison of the performance of the proposed schema between *AES* encryption and *DPM*



(a)



(b)

Figure 5. A comparison of data binding latency between *AES* encryption and *DPM*

without decryption. Our proposed method in this study can be used in *CryptDB* in order to reduce *AES* encryption overheads.

VII. Related Works

To the best of our knowledge, early a limited number of studies have been conducted on data privacy for cloud-based databases. Most of the studies consider encryption methods, or role-based data access methods on *DBMS* side, but any database security method that runs on a server side cannot protect users' data privacy.

In an early study, Denning at al. [5] proposed a theoretical multilevel database security which provides the basic idea of role-based access (*RBAC*) control in database. Jonscher et al. [6] focus on the security of individual queries which cannot be implemented for all queries. Osborn et al. [6] developed an integration of systems where access control is represented by role graphs. The Osborn's security system needs several computation overheads that includes collecting the role of each user, the relation of roles based on a graph, the integration of the graphs, and an algorithm that needs to be run on all transactions. In addition, a graph-based algorithm needs heavy computation, which is not practical for large databases. One of the popular recent study is by Popa's et al. *CryptDB* [4] which considers users' data privacy, but the database is implemented based on *RSA* and *AES* encryption. *CryptDB* uses a proxy server to encrypt or decrypt each user's query. Database likes *CryptDB* can be extended by using *DPM* in order to remove additional computation overheads of *AES*.

VIII. Conclusion and Future Works

Users are facing several challenges when they must outsource their data to a cloud computing system. First challenge in cloud computing is data privacy because any entity from the cloud vendor's side can violate users' data privacy. Second challenge is data security because cloud computing is a form of the Internet-based services that need users to access their data through an untrusted and public network. A cloud-based database can be compromised by authorized cloud vendor users, or unauthorized users. In this paper, we introduced a schema that consists of several components for cloud-based databases that protect users' data privacy. In the case of a compromised database, the data can be only accessible to users who have the key. Although the schema can be implemented by any encryption method, it uses a light-weight data privacy method (*DPM*) that allows users to efficiently protect each record inserted into the database. We conducted several experiments to evaluate the performance of the proposed schema while using *DPM* and *AES* encryption. The experimental results show that the proposed schema provides efficient response when *DPM* is employed. In addition, we analyze the security of *DPM* and the level of users' data protection.

As a future work for this study, we will extend the schema with a zero knowledge paradigm that allows users to run queries on scrambled data without reconstructing data from database. It will remove additional overheads on the database management system, and it will allow users to protect their data privacy efficiently.

References

[1] Mehdi Bahrami and Mukesh Singhal, "The Role of Cloud Computing Architecture in Big Data", Information Granularity, Big Data, and Computational Intelligence, Vol. 8, pp. 275-295, Chapter 13, Pedrycz and S.-M. Chen (eds.), Springer, 2015 http://goo.gl/0LxxlH

[2] Li, Qing, et al. "Applications integration in a hybrid cloud computing environment: modelling and platform." Enterprise Information Systems 7.3 (2013): 237-271.

[3] Bahrami, Mehdi, and Mukesh Singhal. "A Light-Weight Permutation based Method for Data Privacy in Mobile Cloud Computing" in 2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (IEEE Mobile Cloud 2015) San Francisco, IEEE, 2015.

[4] Popa, Raluca Ada, et al. "CryptDB: protecting confidentiality with encrypted query processing." Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. ACM, 2011.

[5] Denning, Dorothy E., et al. "Views for multilevel database security." Security and Privacy, 1986 IEEE Symposium on. IEEE, 1986.

[6] Osborn, Sylvia. "Database security integration using role-based access control." Data and Application Security. Springer US, 2001.

[7] Laur, Sven, Riivo Talviste, and Jan Willemson. "From oblivious AES to efficient and secure database join in the multiparty setting." Applied Cryptography and Network Security. Springer Berlin Heidelberg, 2013.

[8] Daemen, Joan, and Vincent Rijmen. The design of Rijndael: AES-the advanced encryption standard. Springer Science & Business Media, 2013.

[9] Council, Transaction Processing Performance. "TPC-H benchmark specification." Published at http://www. tcp. org/hspec. html (2008).

[10] Lerman, J. Programming Entity Framework: Building Data Centric Apps with the ADO. NET Entity Framework. " O'Reilly Media, 2010.

Citation (To appear**)**:

Mehdi Bahrami and Mukesh Singhal, "CloudPDB: A Light-weight Data Privacy Schema for Cloud-based Databases", 2016 IEEE International Conference on Computing, Networking and Communications, Cloud Computing and Big Data (CLD ICNC'16), Kauai, Hawaii, Feb 2016.