Transparent Data Encryption for Data-in-Use and Data-at-Rest in a Cloud-Based Database-as-a-Service Solution

Sidorov, Vasily; Ng, Wee Keong

2015

Sidorov, V., & Ng, W. K. (2015). Transparent Data Encryption for Data-in-Use and Data-at-Rest in a Cloud-Based Database-as-a-Service Solution. 2015 IEEE World Congress on Services (SERVICES), 221-228.

<https://hdl.handle.net/10356/83217>

<https://doi.org/10.1109/SERVICES.2015.40>

© 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. The published version is available at: [<https://dx.doi.org/10.1109/SERVICES.2015.40>].

Downloaded on 26 Apr 2023 22:16:22 SGT

Transparent Data Encryption for Data-in-Use and Data-at-Rest in a Cloud-Based Database-as-a-Service Solution

Vasily Sidorov

*School of Computer Engineering
Nanyang Technological University
Singapore*

Email: vasily001@e.ntu.edu.sg

Wee Keong Ng

*School of Computer Engineering
Nanyang Technological University
Singapore*

Email: wkn@pmail.ntu.edu.sg

Abstract—With high and growing supply of Database-as-a-Service solutions from cloud platform vendors, many enterprises still show moderate to low demand for them. Even though migration to a DaaS solution might result in a significantly reduced bill for IT maintenance, data security and privacy issues are among the reasons of low popularity of these services. Such a migration is also often only justified if it could be done seamlessly, with as few changes to the system as possible. *Transparent Data Encryption* could help, but solutions for TDE shipped with major database systems are limited to securing only data-at-rest, and appear to be useless if the machine could be physically accessed by the adversary, which is a probable risk when hosting in the cloud. This paper proposes a different approach to TDE, which takes into account cloud-specific risks, extends encryption to cover data-in-use and partly data-in-motion, and is capable of executing large subsets of SQL including heavy relational operations, complex operations over attributes, and transactions.

Keywords—query processing; relational databases; data security; data privacy

I. INTRODUCTION AND RELATED WORK

Lately the Database-as-a-Service (DaaS or DBaaS) concept was rapidly shifting from being a rare niche solution to becoming one of the essential services by cloud platform providers. All major players in the market offer at least one relational and one NoSQL DaaS solution. However, though fully cloud-based IT infrastructure yields significant expenditure cuts for businesses, there still remains a certain reluctance in migrating business-critical systems to the cloud in general and the adoption of DaaS solutions in particular.

One of the outstanding concerns is data security. Outsourcing IT infrastructure implies losing physical control over the hardware and this introduces new security risks. At times these risks may even make transition to the cloud impossible due to laws governing the use and protection of collected personal data. One of the risks inherent in operating in the cloud is the risk of the cloud platform provider itself being malicious: system administrators have

full physical access to the servers and thus, have full access to the possibly sensitive data.

There is a number of versatile approaches to this issue. Cloud providers employ infrastructure management policies that are supposed to make data theft harder to perform. Customers though usually just have to trust that these policies are indeed in use and that they are effective. Vendors of database systems offer solutions to provide transparent data encryption (TDE), which unfortunately have a rather limited scope of addressed privacy and security issues. First, existing TDE schemes in major database management systems (DBMS) focus only on securing data-at-rest. Second, the keys are stored on the same machine that operates the DBMS; this significantly reduces the variety of attacks that the security measures could withstand.

One of the first notable works on providing security for data-in-use for structured data was by Hacıgümüş *et. al.* [1]. Their model was capable of operating a fully encrypted database and executing an essential subset of SQL over it. The authors have introduced storing additional metadata that is required to execute queries along with the fully encrypted rows of tables. This approach is clearly less straightforward than using a single encryption scheme that maintains homomorphism with respect to at least operations of addition and greater/less/equality comparisons; but the latter is not currently feasible in practice [2], [3].

This idea received attention and has been later further developed to extend the supported subset of SQL to incorporate range queries [4]. Other researchers have also worked on improving or changing the model; Yang *et. al.* suggested alternatives to the way search-related metadata is stored along encrypted records in the database [5].

One of the key milestones in the field is the CryptDB project carried out at MIT laboratories [6]–[8]. A key concept of the CryptDB slightly differs from the model suggested by Hacıgümüş and is called “onion” encryption. The idea is to put plaintext values in nested crypto-containers exhibiting varying properties and allow the DBMS to “peel” these containers whenever necessary like onion layers. While it is space efficient and flexible, the model requires at times

supplying secret keys to the DBMS. CryptDB supports a much wider subset of SQL than the model by Hacigümüş but still has certain limitations in what it can do by design.

The idea behind the CryptDB model is a foundation in a follow-up project from MIT — the Relational Cloud [9] — which pursued a similar goal to secure DaaS solutions but also aimed at having additional properties such as elasticity and multi-tenancy.

Some researchers do not limit themselves to a software-only approach to DaaS security; they consider a soft- and hardware hybrid in the TrustedDB project to keep the performance up while providing certain privacy guarantees [10]. The project suggests to run a database on a system using a secure co-processor (SCPU); it enables secure cooperation by employing a number of protocols.

Our contribution is a transparent data encryption model for securing data-at-rest and data-in-use, and to some extent data-in-motion in a regular relational DBMS. In this work, we focus on a flexible and extensible abstract model. Without going into the deeply researched topic of implementing specific operations over encrypted values, we present universal protocols of executing arbitrary superpositions of such operations while maintaining ACID properties and possibility of seamless transition of existing systems to cloud DaaS solutions based on the proposed model.

This paper is organized as follows. Section II defines the threat model and security claims of the proposed system. Section III defines the terminology in use throughout this work. Section IV provides a general understanding of how the parts of the model interact with each other and how the model as a whole operates; it also gives exposition on the architecture of the model. Section V gives a detailed description of the main protocols of the proposed model. Section VI discusses compatibility of the protocols used in the model with transactions. Finally, Section VII concludes this work and summarizes the results.

II. SECURITY AND THREAT MODEL

We consider two types of entities in the model: fully trusted and fully untrusted. Fully untrusted entities are modeled as semi-honest: they are maliciously curious but correctly follow all protocols and instructions. An adversary is a passive observer with full access to the untrusted parts. His goal is to learn as much as possible about the data stored in the database.

The model we present is supposed to lower the probability that the adversary will obtain any plaintext data in a reasonable time to negligible levels.

III. DEFINITIONS

Let us introduce and define the terminology we use in this paper.

Encryption. A general abstraction of a reversible cryptographic transformation of a piece of data x . Encryption is a pair of functions $e(x, k_e)$ and $d(x, k_d)$ where

$d(e(x, k_e), k_d) = x$. Parameters k_e and k_d are encryption and decryption keys that could be equal ($k_e = k_d$, symmetric encryption); or not equal ($k_e \neq k_d$, asymmetric encryption).

Hash (hash function). A general abstraction of a non-reversible cryptographic transformation of a piece of data x ; i.e., a hash is a function $h(x, k)$ such that $\nexists g(\cdot) : g(h(x, k), k) = x$ for $\forall x$. Argument k could be either significant or not; e.g., MD5 hash does not take a “key” argument, whereas HMAC hash does.

Cryptographic transformation. An encryption or a hash.

Cryptoset. One of the key concepts in the proposed model. A *cryptoset* is a data entity formed from the original plaintext data value by applying a set of cryptographic transformations to it. Let v be the original plaintext data value, $e_1(\cdot, \cdot), \dots, e_m(\cdot, \cdot)$ be encryptions, $h_1(\cdot, \cdot), \dots, h_n(\cdot, \cdot)$ be hashes, and $m, n \geq 0$. Then a cryptoset is a set $\{e_1(v, k_1^e), \dots, e_m(v, k_m^e), h_1(v, k_1^h), \dots, h_n(v, k_n^h)\}$.

In all the examples in this paper, we assume that cryptosets for all attributes consist of four encryptions: a multiplication-homomorphic (denoted MUL), an addition-homomorphic (ADD), an order-preserving (OPE), and a deterministic (DET). The encryption and decryption functions are denoted $MUL(\cdot)$ and $d_{MUL}(\cdot)$, and same for other encryptions. For any attribute a , we denote elements of a corresponding cryptoset as $a.MUL$, $a.OPE$, etc. However, it is not required for cryptosets for different attributes to have same sets of elements.

Cryptoset consistency. A cryptoset stored in the database keeps multiple representations of the same value. Certain operations executed against the database might result in a partial update of the cryptoset. As a result, not all elements of the cryptoset will anymore represent the same value, which we call an *inconsistent cryptoset*. Or, if $\exists i, j : 1 \leq i, j \leq m + n$ such that $v_i \neq v_j$ for the cryptoset $\{e_1(v_1, k_1^e), \dots, e_m(v_m, k_m^e), h_1(v_{m+1}, k_1^h), \dots, h_n(v_{m+n}, k_n^h)\}$, then the cryptoset is in an inconsistent state; otherwise, it is in a consistent state.

Virtual data schema. As long as the proposed model offers a transparent database encryption, the end-user (or an application accessing the database on his behalf) is issuing queries as if he is working with a regular plaintext database. However, the actual schema of the data in the encrypted database might be different; thus, the queries are transformed into their encryption-aware counterparts on their way to the database, including changing the specific attribute and table references encountered in the query. The original schema of the plaintext database only exists as an abstraction for the user, and is called a *virtual data schema*.

Core assumption. The model proposed in this paper is relying on a following assumption:

Let O_v and O_b be two sets of operations over data values that we require the DBMS to support. Any operation $o' \in O_v$ results in a value in the same

domain as its operands. Any operation $o' \in O_b$ results in a boolean value $b \in \{\text{true}, \text{false}\}$. For every plaintext operation $f \in O_b \cup O_v$, where f is a function $f(x_1, \dots, x_n), n \geq 1$ there exists a set of cryptographic transformations $t_1^f, \dots, t_m^f$, a decryption function d , and a function g_f such that $d(g_f(t_1^f(x_1), \dots, t_m^f(x_1), t_1^f(x_2), \dots, t_m^f(x_2), \dots, t_1^f(x_n), \dots, t_m^f(x_n))) = f(x_1, \dots, x_n)$.

Examples of operations from O_v are $f_+(a, b) = a + b$, $f_\times(a, b) = a \times b$, etc. Examples of operations from O_b are $f_>(a, b) = \{\text{true if } a > b; \text{false otherwise}\}$, etc. The assumption holds for such basic operations as addition, multiplication, greater/less/equality checks; e.g., homomorphic, order-preserving, and deterministic encryption schemes could be used, correspondingly.

Single-pass query. A query that upon transformation into its encryption-aware equivalent becomes a single query, as opposed to becoming a series of queries. Primarily, in the terminology of the core assumption, queries with expressions that do not contain nested operations over data values become single-pass queries, and vice versa.

In practice, depending on the specific operations and cryptographic transformations used, a query with nested functions might still be a single-pass query, e.g., if $f_\times$ is a multiplication function that takes two arguments, we can directly compute $f_\times(x_1, f_\times(x_2, x_3))$ over encrypted data using a multiplication-homomorphic encryption scheme such as ElGamal. Keeping this in mind, in the presented model we still treat any query with nested operations as a multiple-pass query to not encumber the narrative.

Multiple-pass query. A query that is not a single-pass query.

Query. In this work when we say “query” we mean a query expressed in a limited subset of SQL language. For DDL¹ we only consider `CREATE TABLE`; for DML² — `SELECT`, `UPDATE`, `DELETE`, and `INSERT`. Nested SQL queries are not considered.

IV. MODEL OVERVIEW

The proposed model aims to introduce a new approach to provide transparent data encryption for relational cloud-hosted database systems. Generally speaking, the model acts as an agent between the user or the user-invoked application, and the remote relational DBMS. The user does not have to adjust his queries and may not even be aware that the database is encrypted; thus, *transparent* data encryption. However, the amount of data processing and transfer may change and impact observed behavior of the database client.

During the setup, the agent intercepts user-issued DDL queries and transforms them into cryptoset-aware DDL queries (more on that in Section V-B), which effectively changes the resulting data schema in the database. The agent also registers certain cryptographic routines in the database in the form of UDFs³.

Under normal operation, the agent intercepts DML queries generated by the user and transparently transforms them into corresponding encrypted queries or sets of queries. This process includes transformation of operations on attributes into their cryptographic counterparts (e.g., multiplication to homomorphic multiplication) that are stored in the database during the setup, and breaking down the original query into several if needed; changing references to attributes in the virtual data schema to references to corresponding attributes in the actual data schema; cryptographic transformation of any plaintext values that are present in the original query.

The model consists of three main parts (see Figure 1):

DBMS. The database management system (DBMS) is running on an untrusted cloud platform provided by a passively-curious (in a malicious way) third party. In the scope of the current work, we only consider the case of a DBMS that is hosting a single database consisting only of tables and does not have any views, UDFs, stored procedures, triggers, etc. It also only supports numeric data. It is possible and is considerably straightforward to extend this model to support other data types as well as other entities in addition to tables. Extension of such kind however, does not face any significant research issues and is thus left out.

Different relational DBMSs have slightly varying approaches to providing ACID⁴ properties. In this work we assume that the locking mechanism in the DBMS is able to lock specific records and provides means to manually lock them.

User. The user is an entity that interacts with the DBMS. We use the term to describe both a person and an application the person is using when working with the DBMS. The user is considered to be fully trusted and to possess very limited computational resources.

Proxy. Proxy is a fully trusted intermediary agent between the user and the DBMS, who intercepts the plaintext queries from the user and transforms them into encryption-aware queries with the same semantics; the proxy then forwards the new query to the DBMS. If there is a response from the DBMS, the proxy decrypts it and sends it back to the user. From the user’s standpoint, the proxy *is* the DBMS: he sends his queries to the proxy and obtains proper responses. For simplicity, in this work we assume the proxy to be fully trusted and it manages all the keys. There are techniques to lower the required level of trust for the proxy. One of them is sharing the secret among multiple instances of the proxy

¹Data Definition Language, a part of SQL language for defining the data schema.

²Data Manipulation Language, a part of SQL language for querying the data in the database.

³User-Defined Function

⁴Atomicity, Consistency, Isolation, Durability

Figure 1. Model structure overview

and making them work together to en-/decrypt messages. An example could be found in one of our previous works [11].

V. QUERY TRANSFORMATION

Generally, under the model described in the paper, any query that is issued by the user is sent to the intermediary agent (the proxy) who transforms it into one or a series of encryption-aware queries with the same semantics as the initial query, and forwards them to the DBMS. If the query suggests a response from the DBMS, the proxy receives it in encrypted form, decrypts it, possibly performs some additional processing of the decrypted response, and forwards it to the user.

A. DML Queries

DML query processing and execution could be logically subdivided into outbound (from the user's perspective) and inbound phases. Not all queries have the inbound phase. The outbound phase consists of query transformation and execution. During the inbound phase, the proxy decrypts the response and delivers the result to the user.

In this section, we describe how the query is transformed and executed. The way the model handles this process ensures that:

- 1) Individual queries issued by the user are executed in an atomic and isolated manner;
- 2) By the time the query has finished execution, all affected cryptosets are in a consistent state.

1) *Query analysis*: In this work we consider four subtypes of DML queries under a restricted syntax:

- 1) `INSERT INTO tbl_name (col_name, ...) VALUE (expr, ...)`
- 2) `SELECT { expr, ... | * } [FROM tbl_name, ...] [WHERE where_condition]`
- 3) `UPDATE tbl_name, ... SET col_name = expr [, col_name = expr, ...] [WHERE where_condition]`
- 4) `DELETE FROM tbl_name [WHERE where_condition]`

Operations from O_v might appear in *expr* and *where_condition*; operations from O_b might appear only in *where_condition*. Only queries of subtypes 2–4 contain the *where_condition*, which is formally defined in Section V-A3.

This is a very limited subset of SQL, which is done on purpose to make the idea clearer. There are no serious obstacles to extend the model to support a wider variety of queries, nested queries, and beyond other limitations.

The first thing we want to learn about the query during the query analysis is whether it is a *single-pass* or a *multiple-pass* query. If it is a multiple-pass query, we want to know if the *where_condition* could be computed in a single pass. The idea behind transforming a single plaintext query that appeared to be a multiple-pass query is to put the resulting series of queries in a transaction so that they will still be executed in an atomic and isolated way to the extent the underlying DBMS guarantees it. Production-scale DBMSs usually provide sufficient guarantees. However, we are required to lock all the records that can be affected by the produced set of queries at the beginning of a transaction. If the *where_condition* can not be computed in a single pass (because it has nested operations), we are unable to lock the necessary rows. However, we could still lock a superset that includes all the required rows; this requires predicate analysis.

These stages of query analysis and transformation are considered in detail in the current section. The query analysis workflow is illustrated in Figure 2.

2) *Transforming a single-pass query*: This procedure is performed in three steps. For illustrative purposes, we use this query `SELECT a * b FROM t WHERE a = 5`. It is easy to note that in the case of a single-pass query, the WHERE condition contains only operations from O_b .

- 1) Every operation f_* over attributes is converted to its encryption-aware counterpart g_* (as per the *core assumption*). The predicate $a = 5$ or, in another form, $f_=(a, 5)$ becomes $g_=(a, 5)$. The SELECT expression $a * b$ or $f_\times(a, b)$ becomes $g_\times(a, b)$.
- 2) As stated earlier, data is stored as cryptosets in the database. Every reference to an attribute in the *vir*

Figure 2. Query analysis workflow

tual data schema is substituted by the specific element or elements of the corresponding cryptoset that are required to execute the operation g_* . Assuming that we use deterministic encryption for equality checks and a multiplication-homomorphic encryption for multiplication, the predicate transforms into $g_=(a.DET, 5)$. The SELECT expression after transformation is $g_\times(a.MUL, b.MUL)$.

- 3) Every plaintext constant encountered in an expression goes through cryptographic transformations corresponding to the operation this constant is fed as an argument. If the constant is encountered in the update or insert expression, then a whole new cryptoset might need to be created. Finally, the predicate looks like this: $g_=(a.DET, DET(5))$; and the SELECT expression stays intact as it has no plaintext constants.

After the transformation, the query becomes

```
SELECT  $g_\times(a.MUL, b.MUL)$  FROM t
WHERE  $g_=(a.DET, DET(5))$ .
```

At this point, the DBMS is able to execute the transformed query even with no access to the plaintext values or encryption keys. In practice, the encryption-aware operations g_* can be stored in the DBMS as UDFs or could be added to the core DBMS functionality if the source code is available for modifications.

3) *Predicate analysis and records locking*: Let us formally define the *where_condition*. This is not a syntax definition but rather a structure constraint; so we denote SQL keywords AND, OR, and NOT with corresponding mathematical symbols $\wedge$, $\vee$, and $\neg$ respectively. In the following definition, terminal symbols $bool_op$ and v_op represent an element of the set O_b or O_v correspondingly.

```
where_condition ::=
  bool_expr | '¬' where_condition |
  where_condition '∧' where_condition |
  where_condition '∨' where_condition
bool_expr ::= bool_op '(' expr_list ')'
expr_list ::= expr | expr ',' expr_list
expr ::=
```

```
constant | attribute_ref |
v_op '(' expr_list ')'
```

It is easy to note that the *where_condition* is basically a logical proposition. As it is well-known, any logical proposition can be expressed in Conjunctive Normal Form (CNF). When represented as a tree, a CNF has 4 levels (see Figure 3, left):

- Level 1:** root, conjunction;
- Level 2:** disjunctions;
- Level 3:** negations;
- Level 4:** leaves, actual operations.

If there is no negations in the predicate, we still refer to the leaves as positioned on level 4. Even in a degenerate case of a predicate consisting of just one operation, we still consider the CNF tree as having conjunction on L1 and disjunction on L2 with just one operand each.

The predicate analysis routine takes the predicate in the form of a CNF tree and examines its leaves on L4. As defined, a leaf is a *bool_expr* that can only be a *bool_op* $\in O_b$. If its operands, which are defined to be *expr*'s, are each either a constant or an attribute reference, then the leaf is non-nested and is marked as a single-pass leaf. If at least one of the operands of the *bool_op* is a *v_op*, then the leaf is nested and is marked as a multiple-pass leaf. Clearly, if at least one of the leaves is marked as multiple-pass, the whole query is a multiple-pass query.

As mentioned, multiple-pass queries are executed as a set of queries in a transaction and it is required to lock all records that might be affected by the query first thing in the transaction. Consider the following transformation to the CNF tree. For every disjunction d (L2) we traverse every path down from d to the leaf l ; in case the leaf l is marked as multiple-pass, this path $d-l$ (L3 and L4) is removed from the tree. We then remove childless disjunctions from the tree as a clean-up, if any. As a result we get a different predicate that is computable in a single pass (see Figure 3, right). After that we put in the beginning of the transaction a statement that locks all records that satisfy the resulting predicate (of course, after the predicate undergoes the transformation

Figure 3. Predicate CNF. Multiple-pass leaves are grey; single-pass leaves are white.

procedure for a single-pass query). Our claim is that this will lock at least all the rows that satisfy the original predicate.

Theorem V.1. *Let R be a set of records that satisfy the original predicate P ($\forall r \in R : P(r) = \text{true}$). Let R' be a set of records that satisfy the transformed predicate P' . Then, $R \subseteq R'$.*

Proof: The transformation is done in three steps: 1) conversion to CNF; 2) removing multiple-pass leaf nodes; 3) removing childless disjunctions. If we prove that no record $r_0 \in R$ is eliminated in any of these three steps, the theorem is proven.

1) Predicate to CNF. Any logical proposition can be converted to its *equivalent* CNF. Since the CNF is basically the same proposition in a different form, every record that satisfies the original proposition does also satisfy its CNF, and vice versa.

2) Removing leaves. First, we note that to prove this part all we need to show is that for any given record, such transformation can never change the boolean result of the predicate from *true* to *false* (changing from *false* to *true* is acceptable: we will end up locking more records than necessary but it will affect performance, not correctness). Second, we note that removing an operand from a disjunction is equivalent to replacing the said operand with *true*. We only need to consider those disjunctions that have children removed, as others have unchanged resulting value. If at least one operand of a disjunction is equal to *true*, then the result of the disjunction is *true*. Hence, if the original disjunction results in *false*, after replacing one of the operands with *true*, it will result in *true*; if the original result is *true*, the transformed result is *true*. We have shown that the discussed transformation may only result in certain nodes on L2 changing their compute value from *false* to *true*. Let us consider how it will affect the root conjunction. If at least one of the unaffected by the transformation disjunctions results in *false*, then the conjunction will result in *false* regardless of whether any of the disjunctions may have changed their result to *true*, i.e., the predicate result will *not* change with the

transformation. In case all the unaffected disjunctions result in *true*, there are three options: 1) Some of the affected disjunctions resulted in *false* before transformation and *not* all of them changed to *true*; the predicate does not change the result and stays *false*; 2) Some of the affected disjunctions resulted in *false* before transformation and *all* of them changed to *true*; the predicate changes the result from *false* to *true*; 3) All of the affected disjunctions initially resulted in *true* and stay *true* after transformation; the predicate does not change the result and stays *true*. We have shown that for any given record the discussed transformation can probably change the predicate result from *false* to *true* but never from *true* to *false*.

3) Removing childless nodes. A childless disjunction node is in fact a disjunction operation with no operands, which results in a *true* value. It is easy to prove that adding or removing a *true*-valued operand to a conjunction does not change the result. Thus, removing a childless disjunction from the tree is resulting in an equivalent logical proposition, which obviously does not change the outcome for any given record.

The theorem is proven. ■

4) Transforming a multi-pass query: Generally speaking, a multiple-pass query is transformed by the proxy into a series of single-pass queries, which are executed in a transaction. The procedure for a single-pass query has already been covered earlier. Unlike single-pass queries, multiple-pass queries contain nested operations that cannot be executed by the DBMS without knowing the secret keys, unless we employ a fully-homomorphic encryption scheme, which is completely impractical performance-wise. To overcome this, we first issue additional queries to precalculate intermediary results for nested operations using the proxy to perform the necessary re-encryptions in between the queries. All queries are sent in the transaction context and prior to sending the first query, the proxy requests the DBMS to lock all records that might be affected. Records are locked using a predicate transformation explained in the previous paragraph. In this way we ensure atomicity and isolation for the query.

This paragraph covers the protocol of breaking down a

multiple-pass query into a series of single-pass queries.

Let us consider a query with expression $a * b + c$ or, in f -form, $f_+(f_*(a, b), c)$. Brackets show nesting levels in the expression. This example contains arithmetic operations, which requires the use of homomorphic encryption schemes.

The proxy starts the execution of a multiple-pass query by issuing a command to start a transaction. It then proceeds to lock the required records (see Section V-A3).

Next, the proxy goes through all stacks of nested operations in the predicate, finds the inner-most operations (in our example, $f_*(a, b)$), forms a separate single-pass query to execute each such operation on all records that match the transformed predicate P' , and stores the result in a temporary table. These single-pass queries go through a transformation procedure for single-pass queries and are then sent to the DBMS to be executed. The proxy traverses the stack of operations to the bottom doing the same.

As it could be seen in the example, the temporary table contains the result of operation $g_*(a.MUL, b.MUL)$, which is in fact $MUL(a \times b)$. The next operation in the stack is addition f_+ , which will be transformed into g_+ , which in its turn, requires a value encrypted with ADD encryption. To supply that, in between the newly-formed single-pass queries, the proxy executes a *re-encryption protocol*. It retrieves the previous result from the temporary table and re-processes it with the required cryptographic transformation: $ADD(d_{MUL}(MUL(a \times b)))$, where d_{MUL} is a decryption function for MUL. It then stores the results back in the temporary table so that the next operation in the stack may utilize them.

When the proxy reaches the top of all stacks, the original query with un-nested operations becomes a single-pass query, so it is put through the transformation procedure for single-pass queries, too, and then sent to the DBMS for execution.

Lastly, the proxy cleans up all temporary tables and commits the transaction.

5) *UPDATE/INSERT queries and consistency*: Neither SELECT nor DELETE queries could affect the consistency of a cryptoset. However, UPDATE and INSERT queries could. The proxy takes steps to ensure that these queries result in a consistent database state.

In the case when the query does not contain references to other attributes in the update/insert expression (and thus only contains plaintext constants; for simplicity let us assume that the expression in this case is just a single constant, not an arithmetic expression with constants), the proxy simply forms a full cryptoset from a constant value and executes the amended query.

In the case when the expression refers to attributes, there are two options.

- 1) The expression is just a single reference to another attribute. If the cryptoset signatures (i.e., what specific cryptographic transformations are used) are the same for both the attribute being updated and the attribute

being referenced, then it could just be copied in its entirety. If the signatures are different (e.g., one of the attributes is an id number that is not supposed to be encountered in multiplication; thus, we do not need to store MUL for it), the proxy might need to retrieve it and re-encrypt it in full or in part.

- 2) The expression has (probably, nested) operations over attributes. In this case, the proxy employs the un-nesting protocol discussed earlier and then employs a single-pass query transformation protocol. Finally, only elements of the cryptoset that are related to the last operation in the stack will be present in the cryptoset in the DB. The proxy would need to perform the re-encryption protocol to expand every cryptoset to its consistent state.

6) *Decrypting DBMS response*: SELECT queries are the only DML queries in the scope of this work that could generate a response from the DBMS. When the proxy gets the encrypted results from the DBMS, it proceeds to decrypt them and forwards the plaintext result to the user.

To improve performance, the proxy might choose to retrieve only the fastest to decrypt element of the cryptoset.

B. DDL Queries

As it has been stated earlier, we only consider CREATE TABLE DDL statements. Generally, the proxy needs to know what kind of operations each attribute might get involved into in DML queries. Based on that knowledge, the proxy is able to choose cryptographic transformations that form the cryptoset for each attribute.

Every attribute definition in the statement is substituted by several definitions of attributes that are intended to store encryptions and hashes of the plaintext value. Consider an example: a virtual data schema consists of a table t with attributes a and b , both type INTEGER. It is known that attribute a can only be involved in search operations and thus only requires a deterministic encryption; b can be involved in search and multiplication operations. The virtual data schema $t(a, b)$ will be actually stored as $t(a.DET, b.DET, b.MUL)$.

Working with indices and constraints depends directly on what exact cryptographic transformations are used to execute operations over attributes. For example, if search is done using a deterministic encryption scheme, the DBMS might use index to speed up search through encrypted values. However, deterministic encryption schemes are extremely vulnerable to statistical attacks and thus require strong justification to be used. There are many other, more secure approaches to execute search over encrypted data, many of which are compatible with the model discussed in this paper. Some of them are able to utilize indices. Examples could be found in [11], [12].

VI. HANDLING TRANSACTIONS

Proposed model does support transactions against the virtual data schema, i.e., the user is allowed to issue statements to start/end a transaction and will observe these transactions to be atomic and isolated. Clearly, nothing is different for a single-pass query in a transaction context. In non-transaction context, multiple-pass queries are transformed into transactions. In case a multiple-pass query is issued when already in a transaction, the transformation process goes the same way as described in Section V-A4 with the exception that the proxy does not issue additional statements to start and end the transaction.

Isolation. Since the record locking protocol locks all the necessary records, all concurrent queries and transactions are still unable to access records processed by current transaction. The downside though is that the protocol generally locks more records than necessary, and hence several multiple-pass queries in a transaction could result in a significant amount of locked records, which would be held locked until the whole transaction ends. That could result in a noticeable slowdown of concurrent OLTP.

Atomicity. If one of the statements of the transformed multiple-pass query results in an error, the whole transaction would be rolled back as it would happen with the original query resulting in an error, so the atomicity property is not affected.

VII. CONCLUSION

In this work we have considered a model of a transparent data encryption for cloud DaaS solutions. Unlike existing in practice solutions, the discussed model does not only secure data-at-rest but also extends to completely cover data-in-use and partially cover data-in-motion. At the same time, the en-/decryption keys never leave the trusted perimeter and the DBMS never sees neither them nor plaintext values. The model supports a wide range of usual data queries including those with such heavy relational operations as joins and with complex expressions over data values. The model does support transactions and guarantees isolation and atomicity properties for both individual queries and transactions.

Under certain circumstances some of the approaches employed by the model might result in an overall DBMS performance drop. On one hand, this could be mitigated by designing queries that take into account the internals of the model. On the other hand, these approaches (especially when specific cryptographic transformations are known) have a multitude of optimization opportunities.

REFERENCES

- [1] H. Hacıgümüş, B. Iyer, C. Li, and S. Mehrotra, "Executing SQL over Encrypted Data in the Database-service-provider Model," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '02. New York, NY, USA: ACM, 2002, pp. 216–227. [Online]. Available: <http://doi.acm.org/10.1145/564691.564717>
- [2] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009.
- [3] C. Gentry and S. Halevi, "Implementing gentry's fully-homomorphic encryption scheme," in *Advances in Cryptology – EUROCRYPT 2011*, ser. Lecture Notes in Computer Science, K. Paterson, Ed. Springer Berlin Heidelberg, 2011, vol. 6632, pp. 129–148. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-20465-4_9
- [4] B. Hore, S. Mehrotra, and G. Tsudik, "A privacy-preserving index for range queries," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, ser. VLDB '04. VLDB Endowment, 2004, pp. 720–731. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1316689.1316752>
- [5] Z. Yang, S. Zhong, and R. Wright, "Privacy-preserving queries on encrypted data," *Computer Security—ESORICS 2006*, pp. 479–495, 2006.
- [6] R. Popa, N. Zeldovich, and H. Balakrishnan, "CryptDB: A Practical Encrypted Relational DBMS," Technical Report MIT-CSAIL-TR-2011-005, MIT, Tech. Rep., 2011.
- [7] R. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting Confidentiality With Encrypted Query Processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 85–100.
- [8] —, "CryptDB: Processing Queries on an Encrypted Database," *Communications of the ACM*, vol. 55, no. 9, pp. 103–111, 2012.
- [9] C. Curino, E. P. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich, "Relational cloud: A database-as-a-service for the cloud," in *5th Biennial Conference on Innovative Data Systems Research, CIDR 2011*, January 2011.
- [10] S. Bajaj and R. Sion, "Trusteddb: A trusted hardware-based database with privacy and data confidentiality," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 26, no. 3, pp. 752–765, March 2014.
- [11] V. Sidorov and W. K. Ng, "Complex queries in a shared multi user relational cloud database," in *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, ser. CLOUD '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 903–909. [Online]. Available: <http://dx.doi.org/10.1109/CLOUD.2013.49>
- [12] B. Hore, S. Mehrotra, M. Canim, and M. Kantarcioglu, "Secure multidimensional range queries over outsourced data," *The VLDB Journal*, vol. 21, no. 3, pp. 333–358, 2012. [Online]. Available: <http://dx.doi.org/10.1007/s00778-011-0245-7>