SDB: A secure query processing system with data interoperability

ABSTRACT

We address security issues in a cloud database system which employs the DBaaS model — a data owner (DO) exports data to a cloud database service provider (SP). To provide data security, sensitive data is encrypted by the DO before it is uploaded to the SP. Existing encryption schemes, however, are only partially homomorphic in the sense that each of them was designed to allow one specific type of computation to be done on encrypted data. These existing schemes cannot be integrated to answer real practical queries that involve operations of different kinds. We proposed in [7] a secure query processing system (SDB) on relational tables. The system provides a set of elementary *data-interoperable* operators on encrypted data, which allows a wide range of SQL queries to be processed by the SP on encrypted information. This demo presents an SDB prototype.

1. INTRODUCTION

Advances in cloud computing has recently led to much research work on the technological development of cloud database systems that deploy the *Database-as-a-service model* (DBaaS). Commercial cloud database services, such as Amazon's RDS[1] and Microsoft's SQL Azure[2], are also available. Under the DBaaS model, a *data owner* (DO) uploads its database to a *service provider* (SP), which hosts high-performance machines and sophisticated database software to process queries on behalf of the DO. The SP thus provides *storage*, *computation* and *administration* services. There are numerous advantages of outsourcing database services, such as highly scalable and elastic computation to handle bursty workloads. Also, with multi-tenancy, cloud databases can greatly reduce the total cost of ownership.

To provide data security, sensitive data should not be revealed to the SP. The common practice is to encrypt the data before storing it at the SP. The SP thus provides a reliable repository with storage and administration services (such as backup and recovery). To process queries, the encrypted data has to be shipped back to the DO, which has to process the sensitive data by itself. The powerful computation services given by the SP is mostly lost.

In order to leverage the computation resources of the SP in query processing, a few secure query processing systems, such as CryptDB [5] and MONOMI [6], have been developed. A weakness of these systems is that each data operation (e.g., comparison or addition) is supported by a specialized encryption scheme. These schemes are generally *not data interoperable*, i.e., the output of an operator cannot be used as input of another because different operators employ different encryption methods. As a result, existing approaches provide limited native supports to complex queries that involve multiple types of operators. For instance, CryptDB can only support 4 out of 22 TPC-H queries without significantly involving the DO or extensive precomputation in query processing. Trusted DB [2] and Cipherbase [1] take a hardware approach to provide data security. Since specific hardware is required, these approaches are generally more expensive than software approaches, which can be implemented using off-the-shelf machines. A detailed discussion of the above systems can be found in [7].

Our system, SDB, takes another approach that is based on the Secure Multiparty Computation (SMC) model [8]. With secret sharing, each plain value is decomposed into several *shares* and each share is kept by one of multiple parties. While no party can recover the plain values by its own shares, a protocol executed by the parties can be defined to compute a deterministic function of the values. SDB provides a set of elementary operators to support SQL query processing. A distinguishing feature of SDB is that the supported operators all operate on secret shares and so they all work on the same encrypted space. SDB thus provides *data interoperablility*, which allows a wide range of complex queries to be expressed and processed. As an example, all TPC-H queries can be natively processed by SDB. Moreover, our protocols for executing the various operators are practically efficient.

2. SYSTEM OVERVIEW

In this section we describe our data model, the system architecture, security level, and the query model.

2.1 Data Model

We employ a secret sharing scheme between the DO and

[1]http://aws.amazon.com/rds/
[2]https://sql.azure.com/

SIGMOD/PODS'14, June 22 - 27 2014, Snowbird, UT, USA

http://dx.doi.org/10.1145/2588555.2588572

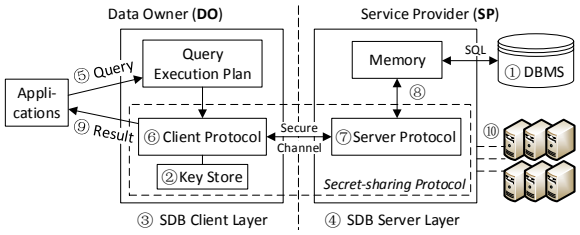

Figure 1: SDB's architecture

the SP. Each sensitive data item v is split into two shares, one kept at the DO and another at the SP. We use $[\![v]\!]$ to denote a sensitive value (the $[\![]\!]$ symbolize that the value is secret and should be kept in a safe). We call the share of $[\![v]\!]$ to be kept at the DO, denoted by v_k, the *item key* of $[\![v]\!]$. The share of $[\![v]\!]$ kept at the SP is denoted by v_e, which is regarded as the *encrypted value* of $[\![v]\!]$. The security goal is to prevent attackers from recovering the set of sensitive data $[\![v]\!]$'s given their encrypted values v_e's. Our solution is built on the relational model and so the database consists of a set of tables. We assume that each table column is categorized as either sensitive or non-sensitive. So, for each column A, either all values in A are encrypted or none is.

2.2 Architecture

Figure 1 shows the architecture. The SP employs a DBMS (①) to store two types of data: (1) the plaintext of non-sensitive data and (2) the encrypted values v_e's of sensitive data. The DO holds the item keys v_k's. As we will explain later in Section 3, the DO does not need to physically maintain and store all the item keys. Instead, for each sensitive column A, the DO stores a *column key* ck_A from which all the item keys of the items in column A are derived. In this way, only a small number of column keys need to be kept at the DO's key store (②).

SDB is implemented as two software layers: a *client layer* (③) which resides at the DO and a *server layer* (④) which resides at the SP. The client layer receives queries from applications and translates each query into a query execution plan (⑤). A plan derives a sequence of operations on data. If an operation involves only non-sensitive data, it is passed directly to the SP, otherwise, the operation is carried out by a secret-sharing protocol. Each such protocol consists of a light-weight *client protocol* (⑥) and a *server protocol* (⑦). The client protocol prepares a message which includes an identification of the data involved, the operation to be executed, and a *hint*, which is derived from relevant column keys and is needed for the SP to carry out the computation on encrypted data properly. The client layer sends the instruction message to the SP's server layer via a secure channel. At the SP, when the server layer receives an instruction message, it first identifies the relevant data. If the data is already residing in memory (e.g., the data required is an intermediate result obtained from previous operations), the server layer executes the server protocol of the operation on the memory-resident data (⑧); Otherwise, the required data is first retrieved from the DBMS before the the server protocol is executed. If the executed operation is the last one of the execution plan, the computed result is shipped back to the client layer, which decrypts it to recover the result's original plaintext values (⑨). In our current implementation, data is stored on a DBMS running MySQL. The server layer executes on a distributed PC cluster (⑩), which is a typical setup of cloud databases.

2.3 Security

We consider three kinds of knowledge that an attacker may obtain by hacking the SP. We explain our security levels against those attackers' knowledge.

Database (DB) Knowledge — The attacker sees the encrypted values v_e's stored in the DBMS of the SP. This happens when the attacker hacks into the DBMS and gains accesses to the disk-resident data.

Chosen Plaintext Attack (CPA) Knowledge — The attacker is able to select a set of plaintext values $[\![v]\!]$'s and observe their associated encrypted values v_e's. For example, an attacker may open a few new accounts at a bank (the DO) with different opening balances and observe the new encrypted values inserted into the SP's DB. We remark that while CPA knowledge is easy to obtain for public key cryptosystems, it is much harder to get under the cloud database environment. This is because the attacker typically does not have control on the number, order, and kinds of operations that are submitted by other users of the system and so it is difficult for it to associate events on plain values with the events on the encrypted ones.

Query Result (QR) Knowledge — The attacker is able to see the queries submitted to the SP and all the intermediate (encrypted) results of each operator involved in the query. QR Knowledge may be obtained in a few ways. For example, the attacker could have compromised the SP to inspect the instructions the client sends to the SP and the computations carried out by the SP. Or the attacker could intercept messages passed between the client and the server over the communication channel. We remark that it is typically more difficult to obtain QR Knowledge than DB Knowledge. This is because data in computation is of transient existence in memory while data on disk persists. The window of opportunity for an attacker to observe desired queries and their (encrypted) results is thus limited. Moreover, there are sophisticated industrial standards to make a communication channel highly secure.

Our security goal is to prevent an attacker from recovering plaintext values $[\![v]\!]$'s given that the attacker has acquired certain combinations of knowledge listed above. First, we argue that DB knowledge is typically easier to obtain than the others and so we assume that the attacker has DB knowledge. Second, it has been proven that no schemes are secure against an attacker that has both CPA knowledge and QR knowledge [3]. Therefore, we assume that the attacker does not have both of these knowledges. Fortunately, as we have explained, CPA and QR knowledges are typically difficult to obtain in a cloud database environment and so the chances of an attacker having both is small. Our system, SDB, is designed to be secure against the following threats:

- **DB+CPA Threat**: The attacker has both DB knowledge and CPA knowledge.
- **DB+QR Threat**: The attacker has both DB knowledge and QR knowledge.

Readers are referred to [7] for proofs of SDB being secure against the above threats.

2.4 Query

We describe the range of queries that SDB supports. First, any queries that involve only non-sensitive data are pro-

cessed by the DBMS at the SP directly. In this case, any SQL queries can be answered. SDB provides a set of secure operators that operate on encrypted data. These operators are data interoperable and so they can be combined to formulate complex queries. Since our data encryption scheme (see Section 3) is based on modular arithmetic, our operators are applicable only to data values of integer domains. Data values of other domains such as strings and real numbers can be encoded as integers with limited precisions. There are, however, certain specific operators (such as keyword search) on such domains that are not natively supported by SDB and have to be implemented indirectly via other means. In [7], we discussed how those operators could be handled. In this demo, we focus on illustrating the system aspects of SDB and assume that data values are all integers and operators take integers input and produce integers output. Table 1 lists the secure operators provided in SDB.

The first 5 operators ($\times$, $+$, $-$, $=$, $>$) are arithmetic and comparison operators. Although they are defined on column operands, a constant operand can also be used by interpreting it as a *constant column*. These operators can be used to formulate selection clauses, such as "*quantity* $\times$ *unit-price* $>$ \$10,000". More complex selection clauses such as conjunction and disjunction of boolean expressions are also supported by SDB. Projection (π), which does not involve encrypted value manipulation, is trivially supported. By combining Cartesian product and selection, theta-join is supported. SDB also supports group-by, sum, average and count and so it provides some aggregation functionality.

3. ENCRYPTION BY SECRET SHARING

In this section we describe how SDB encrypts sensitive data using a secret-sharing method. The DO maintains two secret numbers, g and n. The number n is generated according to the RSA method, i.e., n is the product of two big random prime numbers ρ_1, ρ_2. The number g is a positive number that is co-prime with n.[3] Define,

$$\phi(n) = (\rho_1 - 1)(\rho_2 - 1). \quad (1)$$

We have, based on the property of RSA,

$$(a^{ed} \bmod n = a) \quad \forall a, e, d \text{ such that } ed \bmod \phi(n) = 1.$$

Consider a sensitive column A of a relational table T of N rows $t_1, \ldots, t_N$. The DO assigns to each row t_i in T a distinct random row id r_i. Moreover, the DO randomly generates a *column key* $ck_A = \langle m, x\rangle$, which consists of a pair of random numbers. We require $0 < r_i, m, x < n$.

To simplify our discussion, let us assume that the schema of T is (row-id, A). (Additional columns of T, if any, can be handled similarly.) The idea is to store table T encrypted on the SP. This consists of two parts: (1) Sensitive values in column A are encrypted using secret sharing based on the column key $ck_A = \langle m, x\rangle$ and the row ids. (2) Since the row ids are used in encrypting column A's values, the row ids have to be encrypted themselves. In our implementation, row ids are encrypted by an existing encryption scheme SIES [4].

The reason why row-id and A are encrypted differently is that row ids are never operated on by our secure operators (i.e., we assume row ids are not part of user queries). Hence, a simpler encryption method suffices. On the other hand, sensitive data is encrypted using secret sharing so that computational protocols can be defined to implement our secure operators. The secret sharing encryption process consists of two steps:

[3]In our implementation, ρ_1 and ρ_2 are 1024-bit numbers and so n is 2048-bit.

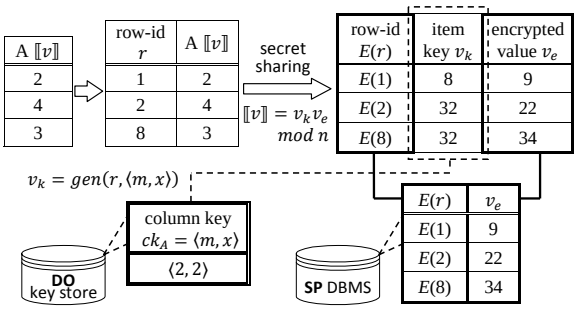

Figure 2: Encryption procedure ($g = 2$, $n = 35$).

Step 1 (item key generation). Consider a row with row id r and a sensitive value (of A) $[\![v]\!]$. Under secret sharing, our objective is to split $[\![v]\!]$ into an item key v_k and an encrypted value v_e. Conceptually, v_e is kept at the SP and v_k is kept at the DO. Since we want to minimize the storage requirement of the DO, the item key v_k is materialized on demand and is generated from the column key ck_A (which is stored at the DO's key store) and the row id r, which is stored encrypted at the SP. Specifically,

[§]Definition 1. *(Item key generation) Given a row id r and a column key $ck_A = \langle m, x\rangle$, the item key v_k is given by,*

$$v_k = gen(r, \langle m, x\rangle) = mg^{(rx \bmod \phi(n))} \bmod n.$$

For simplicity, in the following discussion, we sometimes omit "mod $\phi(n)$" in various expressions with an understanding that the exponent of the above formula is computed in modular $\phi(n)$. So, we write,

$$v_k = gen(r, \langle m, x\rangle) = mg^{rx} \bmod n. \quad (2)$$

Step 2 (Share computation). Shares of $[\![v]\!]$ are determined by a *multiplicative secret sharing* scheme. While v_k is one of the share, the other share v_e, which is considered the encrypted value of $[\![v]\!]$, is computed by the following encryption function $\mathcal{E}$.

[§]Definition 2. *(Encrypted value) Given a sensitive value $[\![v]\!]$ and its item key v_k, the encrypted value v_e is given by,*

$$v_e = \mathcal{E}([\![v]\!], v_k) = [\![v]\!] v_k^{-1} \bmod n, \quad (3)$$

where v_k^{-1} denotes the modular multiplicative inverse of v_k, i.e., $v_k v_k^{-1} \bmod n = 1$.

To recover $[\![v]\!]$, one needs both shares v_k and v_e and compute

$$[\![v]\!] = \mathcal{D}(v_e, v_k) = v_e v_k \bmod n. \quad (4)$$

Figure 2 summarizes the whole encryption procedure and illustrates how sensitive data (e.g., a column A) is transformed into encrypted values v_e's. It also shows that the DO only needs to maintain a column key in its key store, while the SP stores the bulk of the data.

After the SP has computed a query's results, the (encrypted) results are shipped back to the DO, which decrypts them to obtain plaintext results. In this process, the SP

operator	expression	description
$\times$	$A \times B$	vector dot product of two columns of the same table
$+, -$	$A + B, A - B$	vector addition/subtraction of two columns of the same table
$=$	$A = B$	equality comparison on two columns of the same table and output a binary column of '0' and '1'
$>$	$A > B$	ordering comparison on two columns of the same table and output a binary column of '0' and '1'
π	$\pi_S(R)$	project table R on attributes specified in an attribute set S
$\otimes$	$R_1 \otimes R_2$	Cartesian product of two relations
$\bowtie_S$	$R_1 \bowtie_S R_2$	equijoin of two relations on a set of join keys S
$\bowtie$	$R_1 \bowtie R_2$	natural join between two relations
GroupBy	GroupBy(R, A)	group rows in relation R by column A's values
Sum/Avg	Sum/Avg(R, A)	sum or average the values of column A in relation R
Count	Count(R)	count the number of rows in a relation

Table 1: List of secure primitive operators

might have to send the (encrypted) row ids back to the DO for decryption. We will explain the result decryption procedure in the next section, when we discuss how a secure operator is implemented.

4. IMPLEMENTATION OF SECURE PRIMITIVE OPERATORS

4.1 Protocols

Each operator is executed by a protocol, which consists of a client protocol and a server protocol. The client protocol is executed by the DO while the server protocol is executed by the SP. Recall that our data model is column-based. Each sensitive column A is transformed by our secret sharing scheme into a column of item keys v_k's and a column of encrypted values v_e's. Moreover, the item keys are generated by a column key $ck_A = \langle m_A, x_A \rangle$, which is physically maintained by the DO, while the encrypted values are stored at the SP. An operator takes one or more columns as input and produces one or more columns as output[4]. During query processing, the DO manipulates column key(s) while the SP processes column(s) of encrypted values. The details of the protocol for each operator in Table 1 can be found in [7].

4.2 Supported Operator Modes

A column operand of an operator is not necessarily sensitive. For example, the "+" operator could add a sensitive (encrypted) column A with a plain column B. Moreover, for the arithmetic and comparison operators, one of the operands could be a scalar (constant) value. For example, we could multiple a column with a constant as in the selection clause "$2 \times A > B$". We thus have three modes for the operators, namely, EE mode (both operands are encrypted), EP mode (one operand is encrypted, the other is plain), and EC mode (one operand is encrypted, the other is a constant).

4.3 Example Operator - EE multiplication

As an illustrative example, we briefly describe the implementation of the EE (mode) multiplication in this section.

Consider two sensitive columns A and B of a table T whose column keys are $ck_A = \langle m_A, x_A \rangle$ and $ck_B = \langle m_B, x_B \rangle$,

[4]We consider a single numeric value output by an aggregate operator as a column of one single row.

Data: Column A, B with column key $\langle m_A, x_A \rangle$ and $\langle m_B, x_B \rangle$
Result: $C = AB$ with C's column key $\langle m_C, x_C \rangle$

Client-protocol:
$x_C = x_A + x_B \bmod \phi(n)$;
$m_C = m_A m_B \bmod n$;

Server-protocol:
for *each row r* **do**
  Let a_e, b_e be the encrypted values on A, B;
  Set encrypted value of C $c_e = a_e b_e \bmod n$;
end

Algorithm 1: EE multiplication

respectively. Let $C = A \times B$ be the output column with column key $ck_C = \langle m_C, x_C \rangle$. For a row t, let r be its row-id and let a, b and c be the values of A, B, C in row t, respectively. Our objective is to derive (1) the column key ck_C and (2) the encrypted value c_e of c such that $[\![c]\!] = [\![a]\!] \cdot [\![b]\!]$.

To achieve that, the client protocol sets $ck_C = \langle m_C, x_C \rangle = \langle m_A m_B, x_A + x_B \rangle$ and the server protocol computes $c_e = a_e \cdot b_e$ for each row. By Equation (2),

$$c_k = m_C \cdot g^{r x_C} = m_A \cdot m_B \cdot g^{r(x_A + x_B)} = a_k b_k \pmod{n}. \quad (5)$$

By Equations (3), (4), (5) we have, in modular n,

$$[\![c]\!] = c_e c_k = a_e b_e c_k = [\![a]\!] a_k^{-1} [\![b]\!] b_k^{-1} a_k b_k = [\![a]\!][\![b]\!].$$

The protocol is thus correct. Note that the server protocol does not get any information of any keys in the process. The protocol is thus secure. A pseudo code is presented in Algorithm 1.

EE mode multiplication is the simplest among all the operators supported by SDB. The protocols for implementing the other operators supported by SDB are much more elaborate. Due to space limitations, readers are referred to [7] for details.

5. DEMONSTRATION

This demonstration would be the first public demonstration of SDB to our community. An attendee will be able to experience how SDB operates securely. We will use two machines in the demonstration. Machine M_{DO} demonstrates the data owner view of using SDB. Machine M_{SP} demonstrates the administrator view of SDB.

The steps of the demonstration is presented as follows.

1. An attendee chooses the secure column, upload a dataset to the cloud, examining the key store SDB.

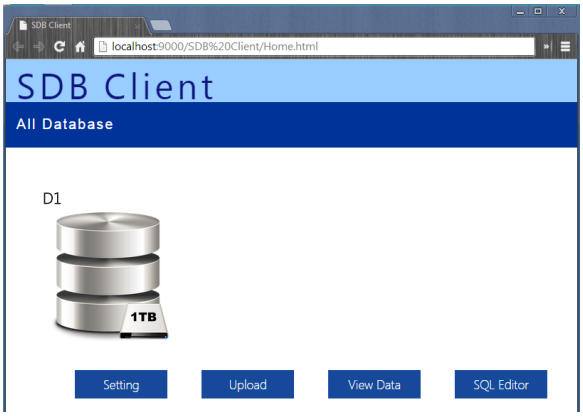

Figure 3: Machine M_{DO}: Data Owner Client

Figure 4: Machine M_{DO}: Selecting sensitive attributes to be encrypted

First, we will invite an attendee to use Machine M_{DO}, which has a sample local database D_1 (see Figure 3). D_1 has not been encrypted and is supposed to be the original dataset owned by the attendee.

Next, the attendee can select D_1 and go into the setting page (see Figure 4). The setting page allows the attendee to choose the attributes that need to be protected. In this step, we let the attendee choose any attributes that she deems appropriate.

After the security settings, the attendee will be invited to click the "Upload" button (see Figure 3) and upload D_1 to the cloud (which is operated by SDB). After the uploading is complete, she shall see another database of the same name (i.e., D_1), but with a little security lock shown next to its icon (see Figure 5). That is the key store of the original database D_1. The attendee will be invited to check the content and the size of the key store Figure 6.

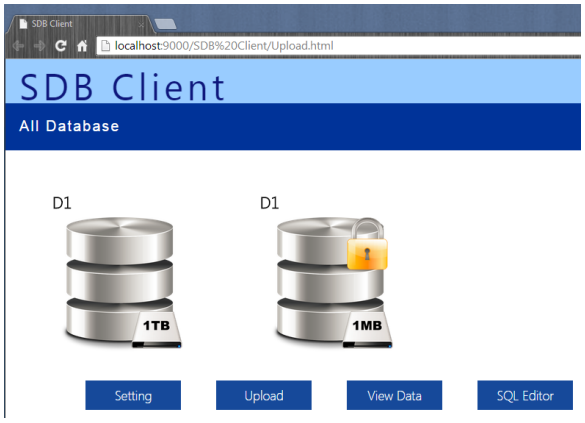

Figure 5: Machine M_{DO}: Data is encrypted and uploaded to the cloud

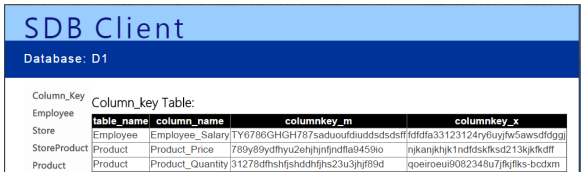

Figure 6: Machine M_{DO}: The key store of D_1

Figure 7: Machine M_{SP}: The Administrator View

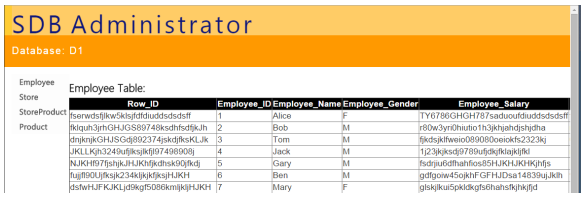

Figure 8: Machine M_{SP}: Administrator sees only ciphertext of encrypted attributes

2. The attendee views the uploaded secure database at the server side of SDB with administrator privilege.

Next, the attendee will be invited to use Machine M_{SP} on which the administrator console of SDB is running. The

attendee shall be able to see all tenant's uploaded databases, including D_1, the one that the attendee has just uploaded (Figure 7). Then, the attendee can check the data of D_1 as the administrator but she shall only see the ciphertext of the encrypted columns (Figure 8).

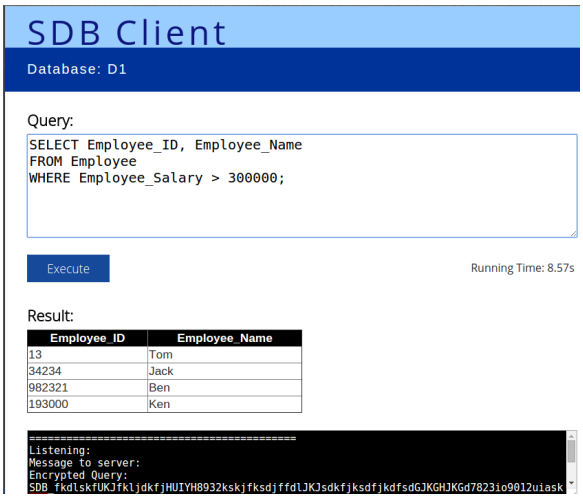

Figure 9: Machine M_{DO}: The data owner sends a query to SDB

3. The attendee submits queries to SDB from the data owner side.

In this step, the attendee will be invited to use Machine M_{DO} to submit queries from the data owner side to SDB. The attendee will first open a query view of the secured database D_1 (Figure 9) and then she can pose any SQL queries. At the bottom of the query view, she can see all messages being communicated between M_{DO} and SDB to understand that no sensitive information is disclosed by the communication protocol. The attendee will also be invited to note the query execution time.

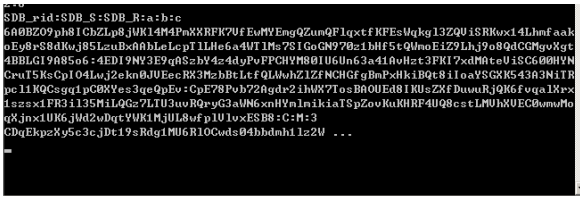

Figure 10: Machine M_{SP}: Memory dump at the service provider

4. The attendee observes the memory dump of SDB at the server side

When the attendee is submitting queries from machine M_{DO} to SDB (Step 3), she will also get invited to look at the memory dump of SDB at machine M_{SP} (Figure 10). This step aims to tell the attendee that the query processing step does not expose any sensitive information all the way.

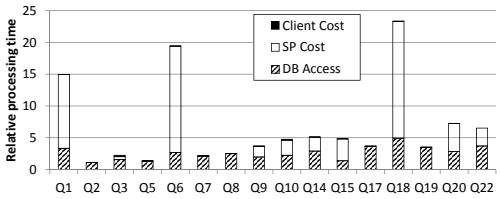

Figure 11: Performance Result

5. The attendee is invited to repeat Steps 1 to 4 by choosing various number of columns to be made secured

In this step, the attendee gets to know the performance overheads incurred in query processing when more or fewer columns are made secure. For example, the attendee can select no columns are encrypted or all columns are encrypted. The attendee will be invited to note the performance differences among the different runs. Finally, we will show the attendee our performance results reported in [7] (Figure 11).

6. REFERENCES

[1] A. Arasu, S. Blanas, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, R. Ramamurthy, P. Upadhyaya, and R. Venkatesan. Secure database-as-a-service with cipherbase. In *SIGMOD*, 2013.

[2] S. Bajaj and R. Sion. Trusteddb: a trusted hardware based database with privacy and data confidentiality. In *SIGMOD*, 2011.

[3] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *CRYPTO*, 2011.

[4] S. Papadopoulos, A. Kiayias, and D. Papadias. Secure and efficient in-network processing of exact sum queries. In *ICDE*, 2011.

[5] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: processing queries on an encrypted database. *CACM*, 2012.

[6] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *PVLDB*, 2013.

[7] W. K. Wong, B. Kao, D. W. L. Cheung, R. Li, and S. M. Yiu. Secure query processing with data interoperability in a cloud database environment. In *SIGMOD*, 2014.

[8] A. C. Yao. Protocols for secure computations (extended abstract). In *FOCS*, 1982.