SDB: A Secure Query Processing System with Data Interoperability

Zhian He[†] Wai Kit Wong[*] Ben Kao[§] David Wai Lok Cheung[§]
Rongbin Li[§] Siu Ming Yiu[§] Eric Lo[†]

[†]Department of Computing, The Hong Kong Polytechnic University
[*]Department of Computing, Hang Seng Management College
[§]Department of Computer Science, The University of Hong Kong

{cszahe, ericlo}@comp.polyu.edu.hk, wongwk@hsmc.edu.hk, {kao, dcheung, rbli, smyiu}@cs.hku.hk

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. Compared to existing secure query processing systems like CryptDB [7] and MONOMI [8], in which data operations (e.g., comparison or addition) are supported by specialized encryption schemes, our demo system, SDB, is implemented based on a set of data-interoperable secure operators, i.e., the output of an operator can be used as input of another operator. As a result, SDB can support a wide range of complex queries (e.g., all TPC-H queries) efficiently. In this demonstration, we show how our SDB prototype supports secure query processing on complex workload like TPC-H. We also demonstrate how our system protects sensitive information from malicious attackers.

1. INTRODUCTION

Advances in cloud computing has recently led to much research works 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.

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

Proceedings of the VLDB Endowment, Vol. 8, No. 12

An important issue of cloud database applications is data security. To protect sensitive data, the plain values of data should not be revealed even to the SP [5]. The common practice is to encrypt sensitive data before it is uploaded to 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, like CryptDB [7] and MONOMI [8], 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 [3] and Cipherbase [2] 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 [9].

Our system, SDB, takes another approach that is based on the Secure Multiparty Computation (SMC) model [11]. 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. To implement the protocol, SDB provides a set of user-defined functions (UDFs) at the SP such that secure query processing can be performed on any relational engine that supports UDF (e.g., PostgreSQL, Vertica, Hive, Spark SQL [1], etc.). A distinguishing feature of SDB is that all UDFs operate on secret shares and so they all work on data in 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.

2. TECHNICAL OVERVIEW

In this section, we provide a brief overview of the SDB system [9]. We first describe our secret sharing scheme and then talk about

the system architecture.

2.1 Secret Sharing

We employ a secret sharing scheme between the DO and 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 $[\![]\!]$ symbolizes 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. The whole procedure is described below.

The DO maintains a secret numbers g and a public key 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,

$$\phi(n) = (\rho_1 - 1)(\rho_2 - 1). \tag{1}$$

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 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 [6].

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:

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) 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,

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

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,*

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

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

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

Figure 1 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, while the SP stores the bulk of the data.

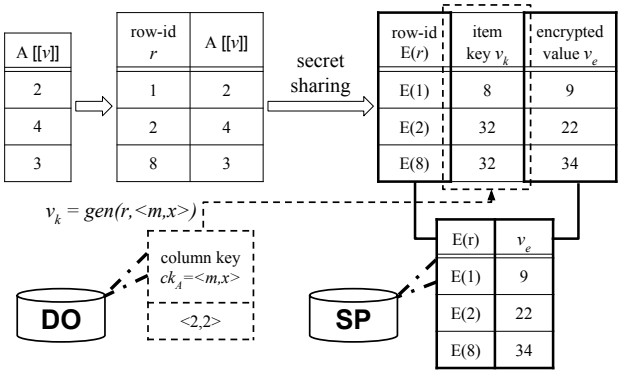

Figure 1: Encryption procedure (g = 2, n = 35)

2.2 System Architecture

The original architecture of SDB that we published in [9] consists of a standalone secure query processing engine that is built on top of a traditional relational engine. That architecture simply treated the relational engine as a data store and did not leverage any of its access methods and fault-tolerance components. In this demonstration, we present our new SDB architecture. Figure 2 shows the new architecture of SDB. The architecture consists of two parts: (1) a lightweight SDB proxy at the DO and (2) a relational engine with a set of UDFs provided by SDB at the SP. In our prototype, we integrate SDB with Spark SQL by implementing the secure operators in a set of Hive UDFs. SDB can easily support any other relational engine by implementing a set of UDFs that work with that particular system. This new architecture pushes all the computations back to the underlying engine through UDFs. Consequently, SDB now enjoys all the benefits such as fault-tolerance, parallel-execution, and scalability provided by the underlying Spark SQL engine.

The SDB proxy is responsible for:

- Storing column keys for sensitive data in its key store.
- Accepting SQL queries from the application.

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

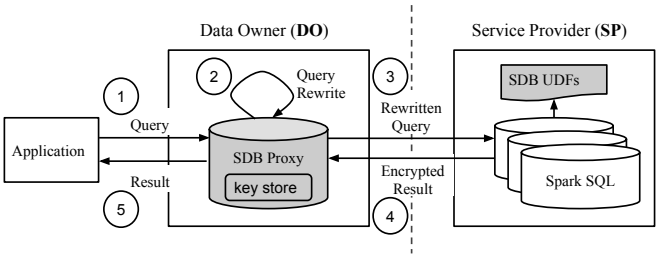

Figure 2: SDB Architecture

- Rewriting the SQL operators that involve sensitive columns to their corresponding UDFs, and then submitting rewritten queries to the SP.
- Receiving encrypted results from the SP and decrypting them using their corresponding column keys.
- Finally, sending the decrypted results back to the application.

The SP provides an unmodified relational engine with a set of SDB UDFs. The engine at the SP is responsible for:

- Storing the plain values of insensitive data and the secrete shares of sensitive data.
- Processing rewritten queries.
- Returning encrypted results back to the SDB proxy.

Let us illustrate query rewriting by an example. Consider two sensitive columns A and B of a table T, whose column keys are $ck_A = \langle m_A, x_A \rangle$ and $ck_B = \langle m_B, x_B \rangle$, respectively. Suppose an application issues the following query:

```
SELECT A × B AS C
FROM T
```

The SDB proxy will rewrite the above query as below:

```
SELECT row−id, sdb_multiply(A_e, B_e, n) AS C_e
FROM T
```

where A_e, B_e and C_e stand for the encrypted values of columns A, B, and C, respectively. The number n is the public key stored at the DO as mentioned before.

According to the protocol of multiplication described in [9], the actual computation carried out by $sdb\_multiply$ is:

$$sdb\_multiply(A_e, B_e, n) = A_e \times B_e \bmod n$$

Recall that to decrypt the result C_e, we need the item keys C_k of C (Equation 4), which are generated by row ids and the column key ck_C of C (Equation 2). Hence, the row-id is added in the rewritten query. Besides, in the case of multiplication, the SDB proxy computes the corresponding column key ck_C as below.

$$ck_C = \langle m_A \times m_B, x_A + x_B \rangle$$

Note that the SP cannot get any secret keys in the process. The computation is thus secure. Readers are referred to [10] for details about the proof of correctness for the above computation.

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 [4]. 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 [10] for proofs of SDB being secure against the above threats.

3. DEMONSTRATION

In the demonstration, we will show a prototype of SDB that is integrated with Spark SQL. The system will also be instrumented from the perspective of an adversary — an administrator can get access to the disk and memory at any instant. We will use two machines in the demonstration. Machine M_{DO} demonstrates the client machine running the SDB proxy. Machine M_{SP} demonstrates the server machine running Spark SQL with the set of SDB UDFs loaded.

The steps of the demonstration is presented as follows.

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

First, we will invite an attendee to use M_{DO}, which has a sample local database D_1. 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 a setting page that 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 to upload D_1 to the SP (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. That is the key store of the original database D_1. The attendee will be invited to check the size of the key store and also the content.

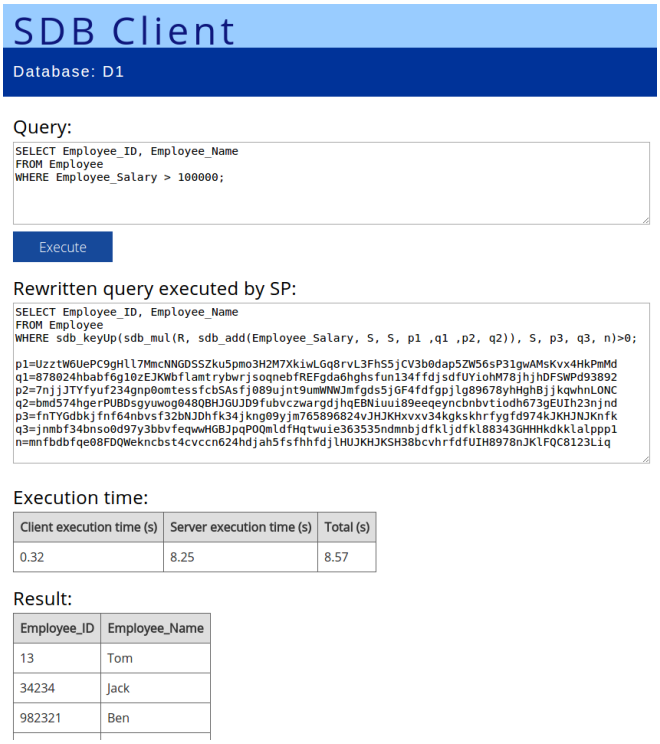

Figure 3: Machine M_{DO}: The data owner sends a query to the SDB Proxy

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

In this step, the attendee will be invited to use 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 3) and then she can pose any SQL queries. The page will show the rewritten query that is actually executed by the server. The attendee will also be invited to note the query execution time that is broken down into a client cost and a server cost. Specifically, the client cost is composed of query parsing time, query rewriting time, and result decryption time. In the demonstration, we show that these costs are subtle compared with the total cost.

Figure 4: Machine M_{SP}: Memory dump at the service provider

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

When the attendee is submitting queries from the client machine to SDB (Step 2), she will also be invited to look at the memory dump of SDB at M_{SP} (Figure 4). This step aims to tell the attendee that the query processing step does not expose any sensitive information at any point of the computation.

Acknowledgements

The paper is supported by GRF Grant 17201414 and FDS grant (UGC/FDS14/E05/14) from Hong Kong Research Grant Council.

We would also thank the anonymous reviewers for their comments and suggestions.

4. REFERENCES

[1] Spark SQL `https://spark.apache.org/sql/`.

[2] 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.

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

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

[5] A. Chen. GCreep: Google engineer stalked teens, spied on chats. Gawker, September 2010 `http://gawker.com/5637234/`.

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

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

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

[9] 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.

[10] W. K. Wong et al. Secure query processing with data interoperability in a cloud database environment. Technical Report TR-2014-03, Department of Computer Science, University of Hong Kong, 2014.

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