HKU CS Tech Report TR-2014-03

Secure Query Processing with Data Interoperability in a Cloud Database Environment

Wai Kit Wong[*] Ben Kao[†] David Wai Lok Cheung[†] Rongbin Li[†] Siu Ming Yiu[†]
wongwk@hsmc.edu.hk, {kao, dcheung, rbli, smyiu}@cs.hku.hk

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

ABSTRACT

We address security issues in a cloud database system which employs the DBaaS model. In such a model, a data owner (DO) exports its 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 propose and analyze a secure query processing system (SDB) on relational tables and a set of elementary operators on encrypted data that allow *data interoperability*, which allows a wide range of SQL queries to be processed by the SP on encrypted information. We prove that our encryption scheme is secure against two types of threats and that it is practically efficient.

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.

Most of existing works on cloud databases focus on performance issues, such as live migration [13, 14], workload consolidation [9, 35], resource management [12], and virtualization [31].

However, *data security* has not been satisfactorily addressed so far. The goal of this paper is to design a secure query processing system in a cloud database environment under the DBaaS model. In particular, we propose a secret-sharing scheme between a DO and an SP such that a wide range of database queries can be executed on the SP without revealing sensitive information.

The common practice to protect sensitive data in database outsourcing is to encrypt the data before storing it at the SP. For instance, Oracle database 11g provides Transparent Data Encryption (TDE), which encrypts disk-resident data using conventional encryption algorithms. 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, we need *homomorphic encryption.* A *fully homomorphic encryption* (FHE) scheme is one that allows any computation to be done on ciphertext [16]. Although FHE is of great theoretical significance, existing schemes are very computationally expensive [17]. It is not practical to apply FHE to even very small-scale computations, let alone database query processing which are highly data intensive.

As a compromise, there have been many previous works [1, 30, 27, 6] on designing *partially homomorphic encryption* (PHE) such that certain restricted kinds of computation can be done on ciphertext. For example, OPES [1] is an order-preserving encryption scheme which allows the order of plaintext to be deduced by comparing encrypted values and so it supports range queries. Other examples include RSA [30] (for multiplication), Paillier's cryptosystem [27] (for addition), and PEKS [6] (for keyword search on encrypted strings). A problem with these PHE schemes is that each of them was designed to process one particular type of operation. Since they employ different encryption functions, they are naturally not data interoperable. In other words, one cannot compose complex operations by piecing together the various simple operations these PHE schemes support. For example, one cannot express a simple selection clause such as "*quantity* $\times$ *unit-price* $>$ \$10,000," which requires both multiplication and comparison.

Considering the limitations of FHE and PHE schemes, we take another approach to design a secure query processing

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

system. Our approach is based on the secret sharing idea of the Secure Multiparty Computation (SMC) model [36]. In 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. It has been shown that any computation that can be expressed as a circuit can be done under the SMC model [18].

One can consider applying SMC in a cloud database environment by decomposing sensitive data into multiple shares, each being kept by an independent SP. For example, ShareMind [4] provides secure storage based on SMC. It also provides a set of protocols for executing various elementary operations such as addition and multiplication. Although theoretically one can execute ShareMind's protocols to answer SQL queries on sensitive data, there are two issues: First, ShareMind requires at least three non-colluding parties (SP's). This incurs significant costs in outsourcing the database. Second, the protocols for computing general functions are expensive, both in terms of the computations at SP's and the communication among them.

In this paper we propose a secure query processing system, SDB, which shares similar flavors with SMC/ShareMind in that sensitive data is decomposed into shares. Here we summarize the distinctive features of SDB:

[**Two Parties**] Unlike ShareMind, SDB requires only two parties — the DO and one SP.

[**Asymmetry**] SDB is asymmetric — the DO is trusted and the SP is untrusted. As we will see later, the asymmetric property of SDB allows us to design computation protocols that require the DO to perform minimal works with minimal storage, while the bulk of computations and data storage for query processing is taken up by the SP. This characteristic fits perfectly to the DBaaS model.

[**Efficient Operators with Data Interoperability**] SDB provides a set of elementary operators that operate on secret shares. These operators operate on data that is encrypted using the *same* encryption scheme. The output of an operator can therefore be taken as input of another. We call the feature that different operators sharing the same encryption scheme and thus an operator can be applied on the results of another operator "*data interoperability*". With data interoperability, a wide range of SQL queries can be expressed and processed in SDB. Moreover, our protocols for executing the various operators are practically efficient.

[**Plain and Encrypted Data Handling**] In practice, much of the DO's data is not sensitive and needs not be encrypted. In SDB, the protocols for implementing the various operators can be applied to both encrypted data and plain data, or a mixture of them.

[**Database System Integration**] SDB can be implemented as a software layer on top of any off-the-shelf DBMSs. As our prototype illustrates, SDB can be integrated seamlessly with existing DBMSs and utilize many of their functionality.

The rest of the paper is organized as follows. Section 2 discusses some related works. In Section 3 we describe our system architecture, the data and query model, and the security model. Section 4 gives the basic idea of our secret sharing scheme. Section 5 explains in details the protocols for implementing the various data interoperable operators. In Section 6 we present our experimental results. Finally, Section 7 concludes the paper.

2. RELATED WORK

The DBaaS model in a cloud database environment has attracted much research attention lately. Most of these works e.g., [7, 24, 8, 11], focus on performance issues such as query processing efficiency and scalability. For example, techniques like database partitioning and workload assignments for processing queries over an array of distributed servers are investigated in [11]. Our encryption system is orthogonal to those techniques and can be integrated into existing systems.

Security in cloud databases has been studied in [19, 20, 3, 29, 32]. In [19, 20], an outsourced database is encrypted by a deterministic encryption scheme (e.g., RSA), which supports equality testing. However, other operations such as ordering comparisons are not naturally supported. CryptDB [29] is a well-known system for processing queries on encrypted data. It employs multiple encryption functions and encrypts each data item under various sequences of encryption functions in an *onion approach*. For example, CryptDB uses a non-deterministic encryption (e.g., RSA with padding) E_1, a deterministic encryption (e.g., RSA) E_2, and an order-preserving encryption (e.g., OPES [1]) E_3 for encrypting numeric data. These encryptions provide different levels of security strength and different levels of computational support — E_1 does not support encrypted data computation but provides the strongest security; E_2 allows equality checks on encrypted data; E_3 allows both equality check and ordering comparison but is not secure against chosen plaintext (CPA) attack [25]. A data item x can be encrypted as $E_1(E_2(E_3(x)))$ before it is stored at the SP. To perform computation on x at SP, the encrypted value has to be decrypted to an appropriate level. For example, to perform equality checking, the data owner has to send the decryption key of E_1 to the SP for it to obtain $E_2(E_3(x))$. Under CryptDB, data security is gradually relaxed to a level that is appropriate for the required computation on the data.

MONOMI [32] is based on CryptDB with a special focus on efficient analytical query processing. It employs a split client/server execution approach. The idea is to intelligently analyze a query plan to identify the computation that can be carried out by the SP over encrypted data and the computation that needs to be performed by the client over decrypted plaintext. A number of optimization techniques are proposed in [32] to improve query performance.

Both CryptDB and MONOMI employ different encryption functions to support different operators. The supported operators are thus not data interoperable. As a result, complex queries that require *piping* the output of one operator to another may not be carried out completely at the SP. For example, the selection clause "$quantity \times unit\text{-}price > \$10{,}000$" cannot be evaluated by the SP on encrypted data. Instead, the encrypted data has to be shipped back to the owner who will evaluate the selection clause itself. MONOMI avoids the data interoperability issue by pre-computation. For example, if it is known that $quantity \times unit\text{-}price$ is frequently evaluated, one can precompute the columns' product and store that as an encrypted materialized column (say C) at the SP. In this case, MONOMI can evaluate the selection clause ($C > \$10,000$) by applying the ordering comparison operator over encrypted data.

Our approach differs from that of CryptDB and MONOMI

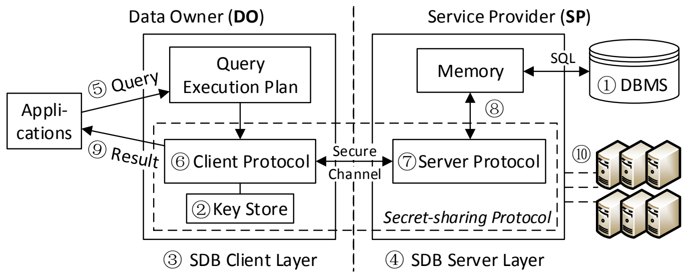


Figure 1: SDB's architecture

in that we focus on designing an encryption method that supports data interoperable operators. This allows a wider range of computation on encrypted data to be carried out directly at the SP, and reduces the work at the client/owner.

In [10] and [34], each tuple in the database is encrypted as a whole. Additional encrypted indices are proposed to support simple point and range queries. In [15], a secure database system based on the SMC protocols is proposed. The difference between their approach and ours is that their system requires three non-colluding SPs to answer general queries while SDB requires only one SP.

Trusted DB [3] and Cipherbase [2] take a hardware approach to provide data security. The SP installs secure co-processors (SCPUs), e.g., the IBM 4764 Cryptographic Co-processor to its machines. SCPUs are tamper-resistant and attackers cannot inspect data stored in an SCPU's memory. Sensitive data is encrypted by the DO, who then distributes the decryption key to the SCPUs via secure channels and sends the encrypted data to the SP. To answer queries, the SP passes the encrypted data to the SCPUs for processing and receives encrypted results. The advantages of the hardware approach are (1) it provides strong security protection as long as the SCPUs are not compromised, and (2) it does not limit query expressiveness. However, SCPUs are expensive (e.g., the IBM 4764 costs about USD $\$10,000$ in 2009). In this paper we focus on an algorithmic approach to solving the secure database problem. Since our solution does not rely on special hardware, it can be implemented using inexpensive off-the-shelf machines.

3. MODELS

In this section we describe our data model, the architecture of SDB, two attack models, and the query model.

[**Data**] First, we briefly mention how data is encrypted. 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 $\llbracket v \rrbracket$ to denote a sensitive value (the $\llbracket\rrbracket$ symbolize that the value is secret and should be kept in a safe). We call the share of $\llbracket v \rrbracket$ to be kept at the DO, denoted by v_k, the *item key* of $\llbracket v \rrbracket$. The share of $\llbracket v \rrbracket$ kept at the SP is denoted by v_e, which is regarded as the *encrypted value* of $\llbracket v \rrbracket$. Our objective is to prevent attackers from recovering the set of sensitive data $\llbracket v \rrbracket$'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.

[**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 4, 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*. As we will explain in Section 4, a *hint* 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.

[**Security threats**] 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 $\llbracket v \rrbracket$'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 memory-based 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 memory-based computation is of transient existence 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 [5]. 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.

Besides the three types of knowledge we mentioned above, there could be scenarios in which the attacker has acquired other knowledge. As an example, an attacker might know the domain of a certain column and the frequency distribution of the values in the domain. Combining this frequency knowledge with, e.g., the results of a group-by operation might allow an attacker to launch a frequency counting attack. Additional measures need to be taken to guard against such attacks. For example, fake tuples with carefully chosen values could be added to tables to disturb the frequency distribution of a column's values. In this paper we focus on DB, CPA, and QR knowledge because these knowledges are relatively easier to obtain by a malicious SP. We will prove that SDB is secure against the two threats derived from these knowledges. Readers are referred to [29] for a discussion on other knowledge and how the functionality of a secure outsourced database could be lowered to guard against attackers with such additional knowledge.

[**Query**] We describe the range of queries that SDB supports. First, any queries that involve only non-sensitive data are processed 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 4) 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. However, certain specific operators on such domains are not natively supported by SDB and have to be implemented indirectly via other means. We will elaborate on this issue and briefly explain how to provide restricted support on non-integer values at the end of this section. For the moment, we 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.

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

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.

[**Limitations**] With data interoperability, SDB is able to support a wide range of SQL queries. Comparing to existing systems, such as CryptDB and MONOMI, which implement various operators using different encryption functions (and thus are not data interoperable), SDB allows much more complex queries to be processed fully at the SP. There are, however, certain limitations on the operators SDB supports. In particular, SDB does not natively support operators that output non-integer values, such as square-root ($\sqrt{A}$). Here, we briefly discuss a few approaches to handle these non-integer operators.

(1) *Table lookup*: If a sensitive data column A of a table R that requires $\sqrt{\ }$ is known to have values from a small integer domain, say D, we can compute $\sqrt{A}$ by the following trick. We precompute a relation $T(D, \sqrt{D})$ with two columns, D (which is the domain of A) and $\sqrt{D}$. $\sqrt{A}$ is computed by evaluating $\pi_{\sqrt{D}}(R \bowtie_{A=D} T)$, which can be done with our secure operators. (2) *Query rewriting*: If $\sqrt{\ }$ is used in a selection clause, the selection condition can be rewritten to remove $\sqrt{\ }$. For example, the selection clause "$\sqrt{X+Y} > Z$" is equivalent to "$X+Y \ > Z \times Z$", which can be evaluated by our operators. (3) *Split client/server execution*: A third option is to adopt MONOMI's split client/server execution approach. The idea is to compute as much as possible at the SP. The execution of non-integer operators are delayed and finally carried out at the DO. For example, if a query requests the values of "$\sqrt{X+Y}$", the SP computes a column $C = X + Y$, returns C to the DO, which then computes $\sqrt{C}$ after decrypting C. With the last option, query plan analysis techniques studied in MONOMI [32] will be very useful.

SDB can also be extended to answer queries that involve conjunctions or disjunctions of selection clauses that involve numerical data and string data separately. For example, we can use PEKS [6] to encrypt strings, which allows keyword search on encrypted strings. Queries such as 'SELECT * WHERE Name LIKE "%John%" AND Salary $> 10,000$' can be evaluated by combining the Boolean results obtainable through PEKS and our secret-sharing scheme.

We remark that although there are limitations on SDB's operators, our design of data interoperable operators is the first of its kind. Our goal is to support the processing of a wide range of SQL queries on encrypted data. In particular, we evaluated SDB over the TPC-H benchmark (Section 6) and all queries in the benchmark are natively supported by SDB. We leave the study of non-integer operators as future work and focus on integer operators in this paper.

4. 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,

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

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 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 using an additive homomorphic encryption $E()$, e.g., SIES [28].

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. As we will see later, in order to correctly implement the secure operators, the encryption function, $E()$, applied on row ids needs only be additive homomorphic. 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's key store) and the row id r, which is stored encrypted at the SP. Specifically,

[3]In our implementation, ρ_1 and ρ_2 are 512-bit numbers and so n is 1024-bit.

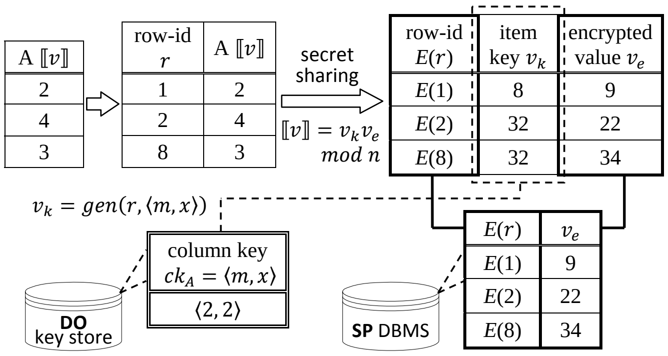


Figure 2: Encryption procedure ($g = 2$, $n = 35$).

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., $v_k v_k^{-1}$ mod $n = 1$.

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

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

It has been proven in [25] that $[\![v]\!]$ cannot be determined with only v_e. Therefore, our scheme is secure against attackers with DB knowledge. Also, we present the proof in Appendix A that our scheme is secure against CPA attacks. That is, if an attacker is given a set S of $\langle [\![v]\!], v_e\rangle$ pairs, it cannot deduce other sensitive data $[\![v']\!]$ that are not mentioned in S from their encrypted values v'_e. Hence, our scheme is secure against attackers with DB+CPA knowledge.

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.

Finally, 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 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 each secure operator is implemented.

5. OPERATORS

In this section we discuss how the secure operators listed in Table 1 are implemented in SDB. First a few notes.

[**Protocols**] Each operator is executed by a protocol, which consists of a client protocol and a server protocol. 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 $ck_A = \langle m_A, x_A \rangle$[4], which is physically maintained by the DO, while the encrypted values are stored at the SP. Since an operator takes one or more columns as input and produces one or more columns as output[5], the client protocol takes column key(s) as input and produces column key(s) as output, while the server protocol's input and output are column(s) of encrypted values.

[**Notations**] We use capital letters, e.g., A, B to represent table columns and we use their lower case counterparts to denote values in columns. For example, we use "a" to denote a value (of a certain row) in column A. We use other lower case letters, e.g., p, q to represent other scalar values. For a sensitive value, say a, we use $[\![a]\!]$, a_k, a_e to represent the plaintext value, the item key, and the encrypted value of a, respectively. Moreover, we use A_e to represent the column of encrypted values a_e's stored at the SP.

[**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 consider 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). We will first discuss EE mode and EC mode implementation. EP mode will be covered at the end of this section.

[**Security**] We show that the implementation of the operators are secure against DB+QR Threat. Specifically, we show that in the execution of an operator, even an attacker sees the message given to the SP and the results of the operator, the attacker cannot deduce the column keys kept by the DO based on the encrypted values stored at the SP. Without the column keys, the attacker cannot deduce the item keys and thus it cannot recover sensitive values $[\![v]\!]$'s. A complete security proof that SDB is secure against DB+QR threat is given in Appendix I.

[**Operator Output**] The output of an operator in many cases is an intermediate result of a query and thus it is of transient existence and may not be physically stored in the database. For example, to evaluate the expression "$A \times 2 > B$", the query processor computes a column $C = A \times 2$ first, which is the output of "$\times$" in EC mode. Note that the output column C is also represented using secret sharing, i.e., it is represented by a column key ck_C (at DO) and an encrypted column C_e (at SP). As we will see later, for the EC mode of "$\times$", the output column (e.g., C) and the

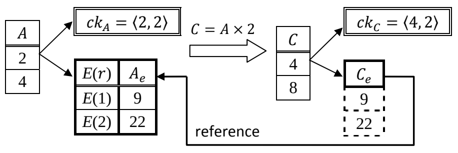


Figure 3: A_e as a reference of C_e.

input column (e.g., A) share the same column of encrypted values. For efficiency, the SP does not physically duplicate A_e to obtain C_e. Instead, a reference is made to associate C_e to A_e. Figure 3 illustrates this idea. For simplicity, in the following discussion, we will only focus on how to compute an operator's output, which includes some column key(s) and some encrypted column(s). We will omit the details of how this data is physically stored.

[**Additional Columns**] To properly implement the various secure operators, we add two columns: S and R to each table T. Each value in S is the constant 1. Each value in R is a positive random number. The columns S and R are encrypted using secret sharing. In particular, their column keys (ck_S, ck_R) are stored at the DO and their encrypted columns (S_e, R_e) are stored at the SP.

Before we give the details of our protocols, we first list two simple properties of our secure sharing scheme. Consider a sensitive column A encrypted with column key $ck_A = \langle m, x \rangle$. If $x = 0$, from Equation 2, we have

$$v_k = gen(r, \langle m, 0 \rangle) = mg^{(r \cdot 0)} \bmod n = m. \quad (5)$$

It follows that

$$(ck_A = \langle 1, 0 \rangle) \Rightarrow (v_k = 1)\,. \quad (6)$$

Also, from Equation 3, we have

$$(v_k = 1) \Rightarrow \left(v_e = \mathcal{E}([\![v]\!], 1) = [\![v]\!]1^{-1} \bmod n = [\![v]\!]\right). \quad (7)$$

PROPERTY 1. *If $ck_A = \langle m, 0 \rangle$, then all values in column A has the same item key m.*

PROPERTY 2. *If $ck_A = \langle 1, 0 \rangle$, then all encrypted values v_e's of column A are equal to their corresponding plaintext values $[\![v]\!]$.*

Note that any column key $\langle m, x \rangle$ the DO generates to encrypt a column should have $x > 0$. However, we will show that for some operators, our protocols will purposely convert a column key to $\langle m, 0 \rangle$ or even $\langle 1, 0 \rangle$. The latter case is typically done to reveal the plaintext of an operator's result, such as the result of a comparison operator specified in a selection clause. We will show that no sensitive information is leaked through this conversion. Next we discuss the client/server protocols of the various operators. A list of pseudo-codes of the operators is given in Appendix J.

5.1 Multiplication ($\times$)

We first discuss the EE and the EC modes of "$\times$".

[**EE Mode**] 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. 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 $[\![c]\!] = [\![a]\!] \cdot [\![b]\!]$.

[4]We use the subscript A here to denote that the key is associated with the column A.

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

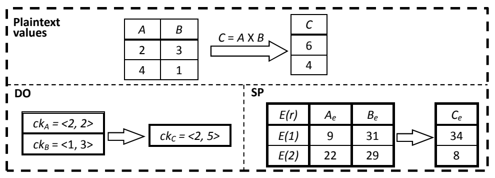


Figure 4: $C = A \times B$ $(g = 2,\ n = 35)$.

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),

$$c_k = m_C \cdot g^{rx_C} = m_A \cdot m_B \cdot g^{r(x_A + x_B)} = a_k b_k \pmod{n}. \quad (8)$$

By Equations (3), (4), (8) 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 client layer sends a message to the server layer instructing it to compute $c_e = a_e \cdot b_e$ for each row in table T. The message contains no information of any keys. The protocol is thus secure. Figure 4 illustrates the protocol: Given $ck_A = \langle 2, 2\rangle$, $ck_B = \langle 1, 3\rangle$, the client protocol computes $ck_C = \langle 2 \times 1, 2 + 3\rangle = \langle 2, 5\rangle$. The server protocol computes $a_e \times b_e \bmod n$ for each row, e.g., the second row gives $22 \times 29 \bmod 35 = 8$.

[EC Mode] We compute $C = A \times p$ for a column A and a constant p. In this case, the client protocol sets $ck_C = \langle pm_A, x_A\rangle$ and the server protocol sets $c_e = a_e$ for each row in the table. In fact, those c_e's need not be materialized. Instead, a reference indicating that $C_e = A_e$ suffices. (See Figure 3.) Note that $c_k = p \cdot m_A g^{rx_A} = p \cdot a_k$. Hence,

$$[\![c]\!] = c_e c_k = a_e c_k = [\![a]\!] a_k^{-1} c_k = [\![a]\!] a_k^{-1} \cdot p \cdot a_k = p[\![a]\!].$$

The protocol is thus correct. Also, no messages with any key information are sent to the server layer. The protocol is thus secure. Besides, since SP has no action at all in the operator, it becomes an advantage of our system in terms of security. A detailed discussion is shown in Appendix B.

5.2 Key Update

Before we continue with other secure operators, let us first discuss a helper operator called *key update*, denoted by κ. The operator, written as, $\kappa(A, \langle m_C, x_C\rangle)$ takes two operands: a column A and a *target column key* $\langle m_C, x_C\rangle$. The output of κ is a column C such that:

(1) C and A have the same plaintext values. That is, for each row-id r, $[\![c]\!] = [\![a]\!]$ where c and a are the values of C and A in row r, respectively.

(2) The column key of C is given by the specified target, i.e, $ck_C = \langle m_C, x_C\rangle$.

Recall that a column S (of plaintext $[\![s]\!] = 1$) is added to each table. By Equation 3, we have $s_e = s_k^{-1}$. Given the column key of S ($ck_S = \langle m_S, x_S\rangle$), to execute key update, the client protocol computes two numbers p, q:

$$p = x_S^{-1}(x_C - x_A) \bmod \phi(n); \quad q = m_A m_S^p m_C^{-1} \bmod n \quad (9)$$

and sends them to the server layer. The server protocol then computes, for each row (of row-id) r, the encrypted value of C by

$$c_e = q \cdot a_e \cdot s_e^p. \quad (10)$$

To prove the correctness of the protocols, we need to show that $[\![c]\!] = [\![a]\!]$ given the column key $ck_C = \langle m_C, x_C\rangle$. In modular arithmetic, we have,

$$\begin{aligned}
[\![c]\!] = c_e c_k &= m_A \cdot m_S^p \cdot m_C^{-1} \cdot a_e \cdot s_e^p \cdot m_C \cdot g^{rx_C} && \text{by (2, 4, 9, 10)}\\
&= m_A \cdot m_S^p \cdot a_e \cdot (s_k^{-1})^p \cdot g^{rx_C} && \because s_e = s_k^{-1}\\
&= m_A \cdot m_S^p \cdot a_e \cdot ((m_S \cdot g^{rx_S})^{-1})^p \cdot g^{rx_C} && \text{by (2)}\\
&= m_A \cdot a_e \cdot (g^{r \cdot x_S \cdot x_S^{-1} \cdot (x_C - x_A)})^{-1} \cdot g^{rx_C} && \text{by (9)}\\
&= a_e \cdot (m_A \cdot g^{rx_A}) = [\![a]\!] && \text{by (2), (4).}
\end{aligned}$$

Note that the client layer sends a message with the values of p and q to the server layer. Since the two numbers are derived from A, C, and S's column keys, we need to ensure that an attacker knowing p and q cannot deduce any key information. In particular, we prove the following properties:

Property 3. *An attacker with DB knowledge cannot recover ck_A or ck_S even if he observes p, q and knows ck_C.*

Property 4. *An attacker with DB knowledge cannot recover ck_C or ck_S even if he observes p, q and knows ck_A.*

The proofs of the above properties are shown in Appendix C. Properties 3 and 4 will be used to prove the security of other operators to be discussed next.

5.3 Addition/Subtraction $(+/-)$

We describe the protocol for "+". The protocol for "−" is very similar so it is omitted.

[EE Mode] Given two sensitive columns A and B, we compute a column $C = A + B$. Consider a row for which the (plaintext value, item key) of A, B, C are ($[\![a]\!]$, a_k), ($[\![b]\!]$, b_k), ($[\![c]\!]$, c_k), respectively. Our objective is to encrypt $[\![c]\!] = [\![a]\!] + [\![b]\!]$ without revealing $[\![a]\!]$ or $[\![b]\!]$. We observe that if $a_k = b_k = c_k = k$ (i.e., if all the values are associated with the same item key k), then our encryption function is *additive isomorphic*, i.e.,

$$\begin{aligned}
a_e + b_e &= \mathcal{E}([\![a]\!], k) + \mathcal{E}([\![b]\!], k) = [\![a]\!]k^{-1} + [\![b]\!]k^{-1}\\
&= ([\![a]\!] + [\![b]\!])k^{-1} = \mathcal{E}([\![a]\!] + [\![b]\!], k) \pmod{n}.
\end{aligned}$$

Hence to implement "+", the server protocol only needs to set $c_e = a_e + b_e$ for each row in the table because then $c_e = \mathcal{E}([\![a]\!] + [\![b]\!], k)$, which is the desired encrypted value of $[\![c]\!]$. By giving A, B, C the same column key, their item keys in the same row will be the same. This can be achieved by applying key update on the columns.

Specifically, to compute $C = A + B$, the client protocol first generates a column key $ck_C = \langle m_C, x_C\rangle$. Then it executes the protocol of key update to get $A' = \kappa(A, \langle m_C, x_C\rangle)$ and $B' = \kappa(B, \langle m_C, x_C\rangle)$. This gives $A' = A$, $B' = B$, and the columns keys of A', B', C are all ck_C. After that, the client layer sends a message to the server layer instructing it to add the encrypted values of A' and B' to get the encrypted values of C, i.e., $c_e = a'_e + b'_e$.

In executing the protocol, two key updates are done. Although each execution of key update is secure, two related key updates may reveal more information than two independent ones. Here, we show that even an attacker relates the key updates, he cannot obtain any information of the keys.

First, an attacker may observe the (p, q) pair (see Equation 9) submitted to the server layer in a key update operation. Let (p_A, q_A) and (p_B, q_B) be the pairs for executing κ on A and on B, respectively. From Equation 9 we have,

$$\begin{aligned}
& x_A + p_A x_S = x_C; \quad x_B + p_B x_S = x_C && \pmod{\phi(n)},\\
\Rightarrow\ & x_A = x_B + (p_B - p_A)x_S && \pmod{\phi(n)},\\
\Rightarrow\ & x_A = x_B + p_{AB} \cdot x_S && \pmod{\phi(n)},
\end{aligned}$$

where $p_{AB} = p_B - p_A$. The attacker can thus relate x_A, x_B and x_S, which are parts of A, B, S's column keys. Suppose there are h columns $A_1, \ldots, A_h$ of a table on which we may apply the addition operator. Let $x_1, \ldots, x_h$ be the "x parts" of these columns' column keys (as in $ck_{A_i} = \langle m_i, x_i \rangle$). Over time, an attacker may collect equations of the form $x_i = x_j + p_{ij} \cdot x_S \pmod{\phi(n)}$, for which p_{ij}'s are known. The following theorem says that it is infeasible to recover the keys (x_i's, x_S) from the system of linear equations.

THEOREM 1. *Given a set of $h+1$ variables $\{x_1, x_2, ..., x_h\}$ and x_S, and a system of equations of the form $x_i = x_j + p_{ij} \cdot x_S \pmod{\phi(n)}$ with known $p'_{ij}s$, there are at least $\phi(n)$ solutions of the variables that satisfy the system of equations.*

The proof of Theorem 1 is presented in Appendix E. Since $\phi(n)$ is a 1024-bit number, the solution space is huge. The attacker therefore cannot recover the "x parts" of the column keys. We can also prove that relating q_A and q_B does not allow the attacker to recover the "m parts" of the column keys either. Our protocol is thus safe.

[**EC Mode**] Computing $C = A+u$, where u is a constant, can be done by computing $A + (S \times u)$, where S is the constant column (of 1's) of the table. This is done by first executing an EC multiplication followed by an EE addition.

5.4 Comparison ($=$ / $>$)

We consider two comparison operators, namely, equality ($=$) and ordering ($>$). Given two columns A and B, the objective is to compute a *plaintext* column C such that, for each row, $c = 1$ if the comparison is "true" and $c = 0$ otherwise. Note that we do not encrypt the output column C because the comparison operators are mostly used in selection clauses and the server layer needs to know the truth value of each selection in processing queries.

To compute C, our protocol first computes $Z = R \times (A - B)$, where R is the random column of the table Next, we perform a key update to obtain $Z' = \kappa(Z, \langle 1, 0 \rangle)$. Hence, Z' is equal to Z but with the column key $ck_{Z'} = \langle 1, 0 \rangle$. By Property 2, the plaintext values of Z' are all revealed to the server. Hence, the server observes the plain values of $R \times (A - B)$. Note that the values of $A - B$ are randomized by the column R, the server therefore cannot observe $A - B$. On the other hand, the server protocol can deduce, for each row, (1) if $z' = 0$, then $[\![a]\!] = [\![b]\!]$, and (2) if z' is positive, then $[\![a]\!] > [\![b]\!]$. With these observations, the server protocol can construct the plain output column C accordingly.

Note that sign information is lost in modular arithmetic. The details of how we address this issue is in Appendix D.

5.5 Projection (π)

The projection operator does not manipulate values in a table. It is supported by instructing the server protocol to return the desired encrypted columns. The client protocol then decrypts the values with its column keys using Eq. 4.

5.6 Cartesian Product ($\otimes$) and Joins ($\bowtie$)

We compute the Cartesian product $T' = T_1 \otimes T_2$ of two tables T_1 and T_2. Readers are referred to Figure 5, which illustrates Cartesian product and the various symbols we use in the following discussion. For simplicity, let us assume that T_1 has only one column A and T_2 has only one column B. (Additional columns can be handled similarly.) Recall that a constant column S is added to each table. Let S_1 and S_2 be the constant columns of T_1 and T_2, respectively. Consider a row in Table T_1 with row-id r_1 and a row in Table T_2 with row-id r_2. Let the values of A and B in these rows be $[\![a]\!]$ and $[\![b]\!]$, respectively. Furthermore, let (plaintext, item key, encrypted value) of S_1 in row r_1 and those of S_2 in row r_2 be $([\![s1]\!], s1_k, s1_e)$ and $([\![s2]\!], s2_k, s2_e)$, respectively. Finally, recall that S_1 and S_2 are columns of 1's. So,

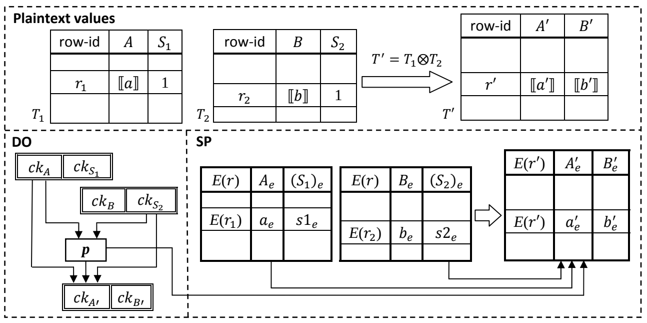


Figure 5: $T' = T_1 \otimes T_2$

$$[\![s1]\!] = [\![s2]\!] = 1. \tag{11}$$

The output of "$\otimes$" is a table T' with two columns A' and B'. For each pair of rows ($r_1 \in T_1$, $r_2 \in T_2$), a row r' in T' is created. Let the values of A' and B' in row r' be $[\![a']\!]$ and $[\![b']\!]$. We want $[\![a']\!] = [\![a]\!]$ and $[\![b']\!] = [\![b]\!]$.

Recall that tables are encrypted using secret sharing, our protocol needs to compute T' in our secret-sharing format based on the column keys, the (encrypted) row-ids, and the encrypted column values of T_1 and T_2. We illustrate the computation by describing the client and server protocols for computing the followings:

(1) [**The column key of** A']. Given the column keys $ck_A = \langle m_A, x_A \rangle$ and $ck_{S_2} = \langle m_{S_2}, x_{S_2} \rangle$, the client protocol first computes $p = x_{S_2}^{-1} x_A \bmod \phi(n)$. It then sets

$$ck_{A'} = \langle m_{A'}, x_{A'} \rangle = \langle m_A (m_{S_2})^p, x_A \rangle. \tag{12}$$

(2) [**The encrypted row-id of** r']. The server protocol sets $E(r') = E(r_1) + E(r_2)$. Recall that row-ids are encrypted using SIES (see Section 4), which is additive homomorphic:

$$\begin{aligned} E(r') &= E(r_1) + E(r_2) = E(r_1 + r_2). \quad \text{So,} \\ r' &= r_1 + r_2. \end{aligned} \tag{13}$$

(3) [**The encrypted value** a'_e **of** $[\![a']\!]$]. The client protocol sends p to the server protocol, which sets

$$a'_e = a_e \cdot (s2_e)^p. \tag{14}$$

By symmetry, other computations, such as the column key of B' and the encrypted values of B' can be similarly done.

To prove that the protocol is correct, we need to show that $[\![a']\!] = [\![a]\!]$. Let a'_k be the item key of a'. We have, in modular arithmetic,

$$\begin{aligned}
a'_k &= gen(r', ck_{A'}) = gen(r', \langle m_A (m_{S_2})^p, x_A \rangle) && \text{(by (12))} \\
&= m_A (m_{S_2})^p g^{r' \cdot x_A} && \text{(by (2))} \\
[\![a']\!] = a'_e a'_k &= (a_e (s2_e)^p) \left(m_A (m_{S_2})^p g^{(r_1 + r_2) x_A} \right) && \text{(by (4, 13, 14))} \\
&= (a_e (s2_e)^p) \left(m_A (m_{S_2})^p g^{r_1 x_A + r_2 p x_{S_2}} \right) && (\because p = x_{S_2}^{-1} x_A) \\
&= (m_A g^{r_1 x_A} a_e) \left((m_{S_2})^p g^{r_2 p x_{S_2}} (s2_e)^p \right) && \\
&= (a_k a_e)(s2_k s2_e)^p && \text{(by (2))} \\
&= [\![a]\!][\![s2]\!]^p = [\![a]\!] 1^p = [\![a]\!]. && \text{(by (11))}
\end{aligned}$$

Note that the client protocol sends $p = (x_{S_2})^{-1} x_A$ to the server protocol. This information is similar to that sent in the key update operation (see Equation 9). The security for the protocol is discussed in Appendix I. Joins are basically Cartesian product with selection. Therefore, joins with selection clauses that are expressible with our arithmetic and comparison operators are supported. Note that equi-join is not efficient if we first compute the Cartesian product. To make query processing faster, we have an operator for equi-join which is discussed in Appendix F.

5.7 Sum, Count, Average and Group-By

Consider a column A of N values a_1, ..., a_N. We want to compute the sum, $\sigma_A = \sum_{i=1}^{N} [\![a_i]\!]$ from encrypted values. To do so, we generate a random number m_Z and transform A into a column Z by a key update: $Z = \kappa(A, \langle m_Z, 0 \rangle)$.

We have the plain values of Z equal to those of A and the column key of Z is $ck_Z = \langle m_Z, 0 \rangle$. Let the values in Z be $z_1, \ldots, z_N$. By Property 1, all z_i's share the same item key m_Z. Let σ_Z be the sum of the encrypted values of column Z, we have,

$$\begin{aligned}
\sigma_Z &= \sum_{i=1}^{N} (z_i)_e = \sum_{i=1}^{N} (\mathcal{E}([\![z_i]\!], m_Z) = \sum_{i=1}^{N} ([\![z_i]\!] (m_Z)^{-1}) \\
&= (m_Z)^{-1} \sum_{i=1}^{N} [\![z_i]\!] = (m_Z)^{-1} \sum_{i=1}^{N} [\![a_i]\!] = (m_Z)^{-1} \sigma_A. \quad (15)
\end{aligned}$$

Hence, $\sigma_A = m_Z \cdot \sigma_Z$. In other words, to compute the sum, the server protocol adds all encrypted values of column Z and returns that sum (σ_Z) to the client protocol, which "decrypts" the value by multiplying it with the secretly generated m_Z.

The COUNT operator can be implemented by having a server protocol that counts the number of rows of a column. With SUM and COUNT, Average can be trivially done.

Group-By can be done by using the column Z. Since values in Z share the same item keys, two values $[\![z_i]\!]$, $[\![z_j]\!]$ are the same if their encrypted values are the same (i.e., if $(z_i)_e = (z_j)_e$). By inspecting the encrypted values of Z, the server can partition the rows of A into groups such that the rows in each group share the same value. The security of the group-by operator is discussed in Appendix I.

5.8 EP Mode of operations

Our secure operators can be applied to a mix of sensitive and non-sensitive columns. This is because, by Property 2, we can treat a plain column A as a sensitive one with the column key $\langle 1, 0 \rangle$. However, in our proofs of operators' security, we assumed that the server does not know the column keys of the operands. Having the secure operator operates on a column with known column key ($\langle 1, 0 \rangle$) could invalidate our security proofs. To avoid this problem, whenever a plain column A is involved in any secure operation, we first apply a key update on A and transform it into a column $C = \kappa(A, ck_C)$, where $ck_C = \langle m_C, x_C \rangle$ is a secret column key that the client layer picks. Any subsequent operations on A will be applied on C instead. Note that by Property 4, even if an attacker knows $ck_A = \langle 1, 0 \rangle$, he cannot deduce ck_C or any other column keys by observing the key update messagesl. Hence, subsequent operators on C are secure.

5.9 Optimization

We end this section with a brief note on performance optimization. First, we note that our encrypt/decrypt functions involve computing modular exponentials, which are typically expensive. There are various methods to speed up such computation, e.g., by using Garner's algorithm [25]. A detailed discussion on optimization techniques is shown in Appendix G. Second, there are studies on secure indexing techniques, e.g., [21]. The general idea is to partition the range of a column's values into coarse ranges. Tuples whose values fall into the same range are grouped together. Such an index helps filter away irrelevant rows (those whose values do not fall in the desired ranges) of a query. Our study is orthogonal to secure indexing. In particular, index filtering can be applied before we process a query using our secure operators. A more detailed discussion is shown in Appendix H.

6. EXPERIMENT

We evaluate the performance of SDB by conducting experiments on our prototype. The prototype was built based on the architecture shown in Figure 1. We use MySQL 5.6.10 as the underlying DBMS. The client layer is running on one machine while the server layer is running on a cluster of 8 machines. So, the server has 8 times more computing power than the client. Each machine has an Intel Core i7-3770 CPU@3.4GHz with 16GB RAM and is running Ubuntu. Both layers are written in C++ using the GMP library[6].

Our experiment is designed to address two questions. First, a focus of SDB is data interoperability. So, $\mathcal{Q}1$: "*Just how important it is for the secure database operators to be data interoperable in a cloud database environment?*" Second, protecting sensitive data by means of encryption always incurs overheads, e.g., ciphertext is typically larger than plaintext. Also, as we mentioned in the introduction, processing data using a fully homomorphic encryption scheme (to achieve data interoperability) is many orders of magnitude slower than plaintext processing, which makes it infeasible for data-intensive applications. So, $\mathcal{Q}2$: "*Is our SDB approach practical in terms of query-processing efficiency?*"

We answer $\mathcal{Q}1$ by comparing the performance of SDB with the *Decrypt-Before-Query* (DBQ) model and MONOMI [32] (Section 6.1). Both DBQ and MONOMI are implemented on the same platform as that of SDB. In particular, they use the same machine configurations for the client and the server as those of SDB, as well as the same DBMS for storing data. For DBQ, since no computation on encrypted data is carried out by the SP, sensitive data is encrypted

[6]The GNU Multiple Precision Arithmetic Library. http://gmplib.org

using RSA encryption. For MONOMI, data is encrypted using various encryption schemes to support different types of operations (see [32]). For DBQ, the server only serves to retrieve relevant (encrypted) columns from the DBMS and to ship them back to the client, which decrypts the data and processes the query to obtain results by itself. CryptDB [29] and ODB [19, 20], for example, have to resort to the DBQ model to process complex queries unless some extensive pre-computation is done (see Section 2). MONOMI adopts a split client/server execution approach. The SP computes as much as possible of the query. For the parts that can't be computed by the SP, they will be passed to the DO. In this case, the DO decrypts all the intermediate results and continue the query computation on plain values. We implemented MONOMI's protocols as detailed in [32].

To answer $\mathcal{Q}2$, we evaluate SDB using the TPC-H benchmark and compare SDB's performance against pure plain-text processing as a reference (Section 6.2). The TPC-H benchmark queries are mostly aggregation queries. They thus amplify the issue of data-intensive query processing. The benchmark is therefore a good stress test of our approach.

6.1 Comparing SDB, DBQ and MONOMI

We compare SDB with an implementation of the DBQ model (or simply DBQ) and MONOMI database (denoted as MDB) using a simple synthetic dataset on which a range of queries are executed. The dataset is a table T with three sensitive columns A, B, C. The values in each column are integers randomly generated with a uniform distribution over the range [0, 1M]. We execute 4 queries which cover the various secure operators of SDB:
[**Range**]: SELECT A, B, C from T WHERE $A + B < q$.
[**Count**]: SELECT COUNT(*) from T WHERE $A + B < q$.
[**Sum**]: SELECT SUM($A * B$) from T WHERE $A + B < q$.
[**Join**]: SELECT SUM($t_1.B * t_2.A$) FROM T as t_1, T as t_2 WHERE $t1.A = t2.B$.

The parameter q controls the queries' *selectivity*. A smaller q gives a more selective query and a smaller query result. In our baseline setting, we have q set to a value such that the selection clauses of the range, count, sum queries return 1% of the table's rows. All the implemented methods employ the domain partitioning indexing technique [21] mentioned in Section 5.9. Specifically, each domain is divided into 50 equal-sized partitions. The index helps us reduce the amount of data retrieved from the DBMS. For example, for the range query, the index filters away any rows in T whose A or B values are larger than q because these rows are guaranteed to fail the selection criterion.

We compare the execution times of SDB, DBQ and MDB. The cost of each is divided into three components: (1) DB Access: the time taken by the DBMS in processing queries and retrieving data for the server layer. (2) Server cost: the time taken by the server layer in executing server protocols. (3) Client cost: the time taken by the client layer in executing client protocols, result decryption and any post-decryption processing. Note that under the DBQ model, there are no secret-sharing protocols and the client computes query results by itself. So, for DBQ, the server cost is null and the client cost includes those for data decryption and query execution on plain data. With encrypted data processing, the server cost and the client cost are mostly dominated by modular arithmetic computation.

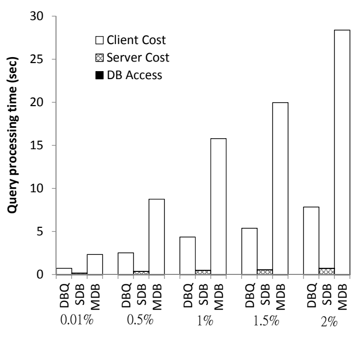
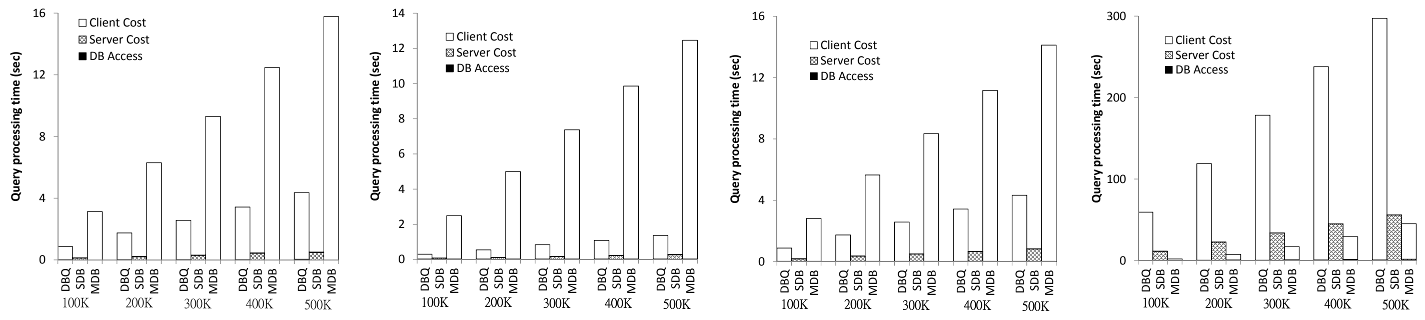


Figure 7: Execution times vs. selectivity

Figure 6 shows the results for the four queries as the table size increases from 100K to 500K rows. Execution times are shown as stacked bars, each with three components (although in most cases, one component dominates all others in the figures shown).

From Figure 6, we make a few observations. (1) DB access times are very small (almost unnoticeable in the figures). This is because with a 1% selectivity, the secure index is quite effective in filtering away many irrelevant rows. Data accesses by the DBMS are thus efficient. (2) The server cost of MDB is much lower than that of SDB. There are two reasons for that: (a) For MDB, the server is only computing parts of a query. For example, consider the range query '$A + B < q$'. The SP can only perform the addition. To complete the query, the encrypted results of the addition operation have to be transferred to the client, which will decrypt the results and evaluate the comparison operation. On the other hand, for SDB, the SP evaluates the query completely. (b) MDB employs various homomorphic encryption schemes that allow computation on encrypted data to be done more efficiently. On the other hand, SDB uses secret sharing to encrypt data. Computation based on secret sharing encryption is generally slower than other homomorphic schemes. (3) The client cost of MDB is much higher than that of SDB for the first three queries. This is because much of the computation involved in evaluating a query has to be carried out by the client under MDB. We observe that the client cost of MDB is even higher than that of DBQ. This is because some of the homomorphic encryption functions, e.g., Paillier cryptosystem employed by MDB are more expensive to decrypt than the simple RSA encryption we used for DBQ. (4) For SDB, the cost is predominately server cost (about 93% of total execution time). This is because queries are computed by the server protocols, which deal with the bulk of the data. The client, on the other hand, only has to decrypt the result. (5) For the join query, MDB and SDB show comparable total costs. Moreover, MDB's client cost is lower than that of DBQ, which is in contrast to the results of the first three queries. This is because the join query requires only a single operation (equality comparison), which can be handled efficiently by the server under MDB. The client only has to decrypt the join result. While most of that cost for MDB is on the client side, for SDB, it is the server that bears most of the cost. In a cloud database environment, however, our objective is to minimize the client cost. So, based on the experimental results, we see that SDB compares very favorably against DBQ and MDB.

Figure 7 shows the execution times of SDB, DBQ and



(a) Range (b) Count (c) Sum (d) Join

Figure 6: Execution times of SDB and DBQ for the 4 sample queries

MDB for the range query as we change the selectivity of the query (by adjusting the parameter q) from 0.01% (very selective) to 2% (less selective). The relative costs of the three components remain similar to those we have observed in Figure 6. In particular, SDB gives much smaller query execution times and that most of the costs under SDB are borne by the server.

6.2 Benchmark

Data processing on encrypted data in intrinsically slower than plaintext processing. After all, ciphertext is generally bigger than plain values. Fully homomorphic encryption (FHE), which supports full-fledged data interoperability, is of theoretical interest only because there are no practically efficient implementations of it. With SDB, we aim at designing a data interoperable system that is practically efficient. The tradeoff is that we have to impose certain restrictions on the data model (SDB is limited to values of integral domains) and on the operators (SDB is limited to operators with integral outputs). In this section, we look at how SDB performs in practice.

We apply the TPC-H benchmark[7] on SDB. We use the sample database of the benchmark at scale 1, which contains 8 tables. The database size is around 1GB, which contains details of suppliers, customers, and orders. There are 22 decision-support queries named Q1 to Q22. Most of these queries involve multiple operators and these queries can be answered completely by the server only if those operators are data interoperable. For example, one query requires the evaluation of an expression $A \times (1 - B)$ on columns A, B. This requires multiplication be done on the output of a subtraction. Note that many of the TPC-H queries are aggregate queries. They are thus highly data-intensive. In this experiment, secure index is not applied. We remark that all queries in the benchmark can be implemented by SDB's secure operators.

In practice, not all data is sensitive. We inspected the benchmark data and identified 7 columns as sensitive information. These columns contain information on account balances, pricings, or orders[8]. The 7 columns are encrypted using our secret-sharing scheme while other columns are stored as plaintext. Among the 22 queries, 6 of them (Q4, Q11, Q12, Q13, Q16, Q21) do not involve any sensitive columns. These 6 queries can be handled directly by the DBMS (running MySQL) on plaintext. So, we omit them in the experiment.

[7]http://www.tpc.org/tpch/

[8]The seven columns are S_ACCTBAL, C_ACCTBAL, O_TOTALPRICE, L_QUANTITY, L_EXTENDEDPRICE, L_DISCOUNT, L_TAX.

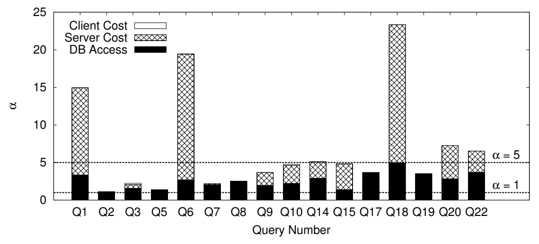


Figure 8: α of 16 TPC-H queries

We evaluate SDB by executing the other 16 queries on our prototype. SDB analyzes a query and "pushes" as much work as possible to the DBMS, as long as the work does not involve sensitive data. For example, consider the query Q: "SELECT $A \times B$ ... WHERE $C > D$". If columns C and D are not sensitive, then the selection can be done by the DBMS. The server protocol has to compute $A \times B$ only for those rows that satisfy the selection clause.

The execution time of SDB (T_{SDB}) for each query is noted. We then execute the queries on the DBMS directly with all columns stored as plaintext data, bypassing all the secure operator protocols. The execution time under this scenario (T_{DBMS}) for each query is also noted. The ratio $\alpha = T_{SDB}/T_{DBMS}$ captures the *slow-down* of SDB compared with plaintext processing if that were performed by the client itself. The larger α is, the higher is the price we pay for providing data security on a cloud database.

Table 2 shows T_{DBMS} for all 16 queries tested. Figure 8 shows the α ratios of the queries. Again the bars are shown as stacked bars with the three components of execution times displayed.

From Figure 8, we see that α is below 5 for 11 out of the 16 queries. The worst slow-down is observed in Q18, which is about 23. Furthermore, we observe that: (1) For some queries (e.g., Q5, Q19), the execution times are dominated by DB Access times, and these queries generally have small slow-downs. We find that the selection clauses of these queries are mostly on non-sensitive columns. As we have discussed, SDB pushes these (non-sensitive) selections to the DBMS. In these cases, the DBMS returns a small set of rel-

Q1	Q2	Q3	Q5	Q6	Q7	Q8	Q9	Q10	Q14	Q15	Q17	Q18	Q19	Q20	Q22
5.38	1643.18	10.82	23.58	5.19	9.20	25.18	41.66	7.63	5.51	10.86	13.77	35.71	10.00	11.28	0.27

Table 2: T_{DBMS}. Execution times of 16 TPC-H queries processed on plaintext data (in sec.)

evant rows, which are further processed by the secure server protocols (e.g., to compute a SUM over a sensitive column). Since the sets of rows processed by secure protocols in these queries are small, the server costs are relatively small. (2) For some queries (e.g. Q1, Q18), the server costs are very significant. These cases are opposite to those discussed in observation (1). They tend to have selection clauses that involve mostly sensitive columns. In these cases, the DBMS cannot perform effective filtering. Instead, all rows of relevant columns have to be retrieved and submitted to the server layer by which the secure protocols are executed. This leads to a high server cost. (3) For all 16 queries, the client costs are minimal. This is because the SP takes care of most of the query processing. The client protocols only need to manipulate column keys and to decrypt small summarized results computed by the server.

7. CONCLUSIONS

In this paper we gave a comprehensive description and analysis of SDB, which supports secure query processing on cloud databases with a set of secure, data-interoperable operators. By employing an asymmetric secret-sharing scheme, SDB allows complex SQL queries to be processed entirely by the server (SP). Client's (DO's) computation is mostly limited to result decryption. By experiment, we showed that SDB is applicable to many real-life queries and applications (e.g., the whole TPC-H benchmark can be executed on it). SDB is also practically efficient. This is in sharp contrast to existing FHE schemes, which are many orders of magnitude slower than plaintext processing. We believe that the ability of SDB in offloading secure computation to servers represents a significant advancement in secure cloud database systems.

8. REFERENCES

[1] R. Agrawal and J. Kiernan et al. Order-preserving encryption for numeric data. In *SIGMOD*, 2004.

[2] A. Arasu et al. Secure database-as-a-service with cipherbase. In *SIGMOD*, 2013.

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

[4] D. Bogdanov et al. A universal toolkit for cryptographically secure privacy-preserving data mining. In *PAISI*, 2012.

[5] A. Boldyreva et al. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *CRYPTO*, 2011.

[6] D. Boneh et al. Public key encryption with keyword search. In *EUROCRYPT*, 2004.

[7] F. Chang et al. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 2008.

[8] C. Curino and E. P. C. Jones et al. Relational cloud: a database service for the cloud. In *CIDR*, 2011.

[9] C. Curino et al. Workload-aware database monitoring and consolidation. In *SIGMOD*, 2011.

[10] E. Damiani et al. Balancing confidentiality and efficiency in untrusted relational dbmss. In *CCS*, 2003.

[11] S. Das and D. Agrawal et al. Elastras: An elastic transactional data store in the cloud. *CoRR*, 2010.

[12] S. Das, V. Narasayya, and F. Li et al. CPU sharing techniques for performance isolation in multi-tenant relational database-as-a-service. *PVLDB*, 2014.

[13] S. Das, S. Nishimura, and D. Agrawal et al. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *PVLDB*, 2011.

[14] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD*, 2011.

[15] F. Emekçi and D. Agrawal et al. Privacy preserving query processing using third parties. In *ICDE*, 2006.

[16] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, 2009.

[17] C. Gentry et al. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*, 2012.

[18] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*, 1987.

[19] H. Hacigümüs, S. Mehrotra, and B. R. Iyer. Providing database as a service. In *ICDE*, 2002.

[20] H. Hacigümüs et al.and B. R. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *SIGMOD*, 2002.

[21] B. Hore, S. Mehrotra, and G. Tsudik. A privacy preserving index for range queries. In *VLDB*, 2004.

[22] M. Kantarcioglu and C. Chris. Privacy-preserving distributed mining of association rules on horizontally partitioned data. *TKDE*, 2004.

[23] D. G. Kendall and R. Osborn. Two simple lower bounds for euler's function. *Texas J. Sci.*, 1965.

[24] T. Kraska et al. Consistency rationing in the cloud: Pay only when it matters. *PVLDB*, 2009.

[25] A. J. Menezes, P. C. Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[26] G. Oded. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, 2004.

[27] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, 1999.

[28] S. Papadopoulos et al. Secure and efficient in-network processing of exact sum queries. In *ICDE*, 2011.

[29] R. A. Popa et al. Cryptdb: processing queries on an encrypted database. *CACM*, 2012.

[30] R. L. Rivest et al.and A. Shamir and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *CACM*, 1978.

[31] A. Soror et al. Automatic virtual machine configuration for database workloads. In *SIGMOD*, 2008.

[32] S. Tu and M. F. Kaashoek et al. Processing analytical queries over encrypted data. In *PVLDB*, 2013.

[33] J. Vaidya et al. Secure set intersection cardinality with application to association rule mining. *JCS*, 2005.

[34] S. Wang, D. Agrawal, and A. El Abbadi. A comprehensive framework for secure query processing on relational data in the cloud. In *SDM*, 2011.

[35] P. Wong, Z. He, and E. Lo. Parallel analytics as a service. In *SIGMOD*, 2013.

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

APPENDIX

A. SECURITY AGAINST DB+CPA THREAT

In this section, we present a security proof of our encryption scheme w.r.t. DB+CPA Threat. With CPA knowledge, the attacker is able to observe the encrypted value $v_e = \mathcal{E}([\![v]\!], v_k) = [\![v]\!]v_k^{-1} \bmod n$ of some plaintext $[\![v]\!]$ where v_k denotes the item key. The attacker can thus observe $v_k = [\![v]\!]v_e^{-1} \bmod n$. Recall that $v_k = gen(r, \langle m, x\rangle) = mg^{rx} \bmod n$ where r and $\langle m, x\rangle$ are the row-id and column key of value $[\![v]\!]$. As we will show below, even the attacker is able to observe advanced knowledge of the value of g^r, it is still hard for the attacker to derive other secret parameters.

THEOREM 2. *Let $\langle m, x\rangle$ be the column key of a column X. Given a set of item keys on X and the set of corresponding values of g^r of each row, it is computationally hard to find x.*

PROOF. We show that the above can be reduced to RSA. In RSA, the attacker obtains the public key $\langle e, n\rangle$. This allows the attacker to generate the encrypted value $y' = y^e \bmod n$ for any plaintext y. The attacker aims to obtain the private key $\langle d, n\rangle$ s.t. $y = y'^d \bmod n$. To achieve the goal, the attacker prepares a set of item keys as follows: (i) artificially set a random value $\hat{g}$; (ii) choose random $\hat{r}$ and compute $y = \hat{g}^{\hat{r}}$; (iii) compute $y' = y^e$ and pretend that y is the item key and the value of g^r is y'; (iv) repeat step (ii) and (iii) until the attacker gets enough pairs to make a successful attack to our scheme. Note that the above steps allow the attacker to simulate the knowledge to attack our scheme with a column key $\langle m = 1, x = d\rangle$ with different row-ids $r = \hat{r}$ and system parameter $g = \hat{g}^e$. (So, we have (i) $gen(\hat{r}, \langle 1, d\rangle) = \hat{g}^{\hat{r}ed} = \hat{g}^{\hat{r}} = y$ and (ii) $g^r = \hat{g}^{e\hat{r}} = y'$.)

If our scheme is not secure, i.e., an attacker can find $\langle m = 1, x = d\rangle$, the attacker recovers the private key in RSA. Thus, our encryption scheme is as strong as RSA. □

Theorem 2 shows that our item key generation is secure w.r.t. CPA even if some parameters are leaked. The above also applies to the case when the roles of row and columns, i.e., r and x in item key generation function, are swapped. So, the secret parameters r, x are strongly protected in our scheme. (r may also be leaked as an encrypted copy $E(r)$ is sent to SP. Recall that E can be any off-the-shelf additive homomorphic encryption function. So, we assume the attacker cannot observe r from $E(r)$.) The attacker cannot generate the item keys of other unknown values in the table as the secret key parameters in item key generation are unknown. We remark that CPA may not be actually possible in our system, as no key is made public in our system. However, this provides a strong security guarantee to our system.

B. SECURITY ADVANTAGE FROM EC MULTIPLICATION

In Section 5.1, we presented a protocol in handling EC multiplication $C = p \times A$. In particular, the server protocol has no real action apart from copying an encrypted column A as the resulted encrypted column C. As we discussed in Sec. 5.1 , the above copying is not necessary. The client layer just keeps a reference indicating that C's encrypted column at SP is in fact A's encrypted column. Thus, the server has no action at all in server protocol (see Algorithm 2). The advantage of this is that SP does not even know if there is any EC multiplication executed. For instance, consider the operations of $A + B$ and $2A + 3B$. The former operation requires SP to execute the protocol for EE addition on the encrypted columns A_e, B_e. The latter operation first requires the client layer computes $A' = 2A$ and $B' = 3B$. These are done solely by the client layer. Then, an EE addition is computed on A' and B'. Since the encrypted columns of A' and B' are still A_e and B_e. SP is instructed to perform EE addition using the encrypted columns A_e and B_e. Note that SP views the same instructions in both operations. SP cannot distinguish which operation is actually being executed. In general, SP cannot distinguish whether an operation is done on A or pA for any value of p. For example, if SP receives an instruction to execute EE addition on encrypted columns A_e and B_e, SP infers that the operation is in the form of $pA + qB$ with p, q unknown. Let $ck_A = \langle m_A, x_A\rangle$ be the column key of A. For the column $C = pA$, the m-part of ck_C is pm_A while x-part is x_A. Even if the column key of C is leaked, the m-part of ck_A is unknown because p is unknown. Thus, all operations in our scheme do not leak the m-part of any column key.

C. PROOF OF SECURITY PROPERTIES OF KEY UPDATE

In this section, we provide a proof for property 3 and property 4 of key update. Property 3 is discussed below.

THEOREM 3. *Let $ck_A = \langle m_A, x_A\rangle$, $ck_C = \langle m_C, x_C\rangle$, $ck_S = \langle m_S, x_S\rangle$ be the column keys of A, C, S where S is a constant column. Let $p = x_S^{-1}(x_C - x_A) \bmod \phi(n)$, $q = m_A m_S^p m_C^{-1} \bmod n$. Given p, q, ck_A, there exists at least $\phi(\phi(n))$ solutions of ck_C and ck_S where ϕ denotes Euler's totient function.*

PROOF. First, we discuss on the x-part of column keys. The attacker can establish an equation $p = x_S^{-1}(x_C - x_A) \bmod \phi(n)$. $\forall \hat{x_S} \in [1, \phi(n))$ s.t. $\gcd(\hat{x_S}, \phi(n)) = 1$, $\exists \hat{x_C} = x_A + p\hat{x_S} \bmod \phi(n)$ s.t. $\hat{x_C}$, $\hat{x_S}$ satisfy the above equation. Since there are $\phi(\phi(n))$ values of x_S that are co-prime to $\phi(n)$, there are at least $\phi(\phi(n))$ pairs of solutions. For the analysis on m-part, please refer to Appendix B. □

The size of $\phi(n)$ is 1024-bit and the value of $\phi(\phi(n))$, which is the number of values that are co-prime to $\phi(n)$, is large. Besides, the above cryptanalysis requires the knowledge of $\phi(n)$ in setting up the equations. However, the value of $\phi(n)$ requires the knowledge of factorizing $n = \rho_1\rho_2$ (See equation 1). This is assumed to be a hard problem and so it is not feasible for a practical attacker to perform the above analysis. On the other hand, property 4 can be proved similarly.

THEOREM 4. *Let $ck_A = \langle m_A, x_A\rangle$, $ck_C = \langle m_C, x_C\rangle$, $ck_S = \langle m_S, x_S\rangle$ be the column keys of A, C, S where S is a constant column. Let $p = x_S^{-1}(x_C - x_A) \bmod \phi(n)$, $q = m_A m_S^p m_C^{-1} \bmod$*

n. Given p, q, ck_C, there exists at least $\phi(\phi(n))$ solutions of ck_A and ck_S where ϕ denotes Euler's totient function.

Proof. By setting $x'_A = -x_C$ and $x'_C = -x_A$, the equation concerning p becomes $p = x_S^{-1}(x'_C - x'_A)$ with x'_A and p known by the attacker. Thus, the remaining analysis becomes the same as the one in Theorem 3. □

D. MANAGING SIGN ON ENCRYPTED VALUE SPACE FOR COMPARISON

A necessary property in our comparison is that $R\times(A-B)$ preserves the same sign as $A-B$. However, sign information is lost in modular arithmetic, e.g., $-3 = 2 \pmod 5$. Now, the domain of encrypted values is $[0, n-1]$. We separate positive values and negative values by dividing the domain into two regions. The first half of the domain is denoted as positive, i.e., the positive region R^+ contains $[1, (n-1)/2]$. The other half is denoted as negative, i.e., the negative region R^- contains $[n+1/2, n-1]$. For example, with $n = 35$, $R^+ = [1, 17]$ and $R^- = [18$ (represents -17), 34 ((represents -1)]. By defining the above regions, overflow may happen if the magnitude of a value is larger than $(n-1)/2$. So, it is important for our system to control the above overflow problem. Note that both R^+, R^- has a size of 1023-bit. With R being a random 80-bit number, $A - B$ can be as large as 923-bit numbers without overflow. Then, by checking whether an encrypted value z_e is positive, it means we compare z_e and $(n-1)/2$. If $z_e < (n-1)/2$, z_e is regarded as positive. If $z_e \geq (n-1)/2$, z_e is regarded as negative.

E. PROOF OF THEOREM 1

Let $\langle x_1 = \hat{x_1}, x_2 = \hat{x_2}, ..., x_h = \hat{x_h}, x_S = \hat{x_S}\rangle$ be the true values of the secret parameters, i.e., $\hat{x_i} = \hat{x_j} + p_{ij}\hat{x_S} \bmod \phi(n)$. We can construct another valid solution as $\langle x_1 = \alpha\hat{x_1}, x_2 = \alpha\hat{x_2}, ..., \alpha x_h, \alpha x_S\rangle$ for any positive $\alpha < \phi(n)$. By putting the solution into the equations, we obtain: (i) LHS $= x_i = \alpha\hat{x_i}$; and (ii) RHS $= x_j + p_{ij}x_S = \alpha(\hat{x_j} + p_{ij}\hat{x_S}) = \alpha\hat{x_i}$. Thus, LHS = RHS. This confirms the validity of the above solution. Since there are $\phi(n)$ different possible values of α, there are at least $\phi(n)$ solutions to the system of equations. Since $\phi(n) = (\rho_1 - 1)(\rho_2 - 1)$ and ρ_1, ρ_2 are 512-bit numbers, $\phi(n)$ is a 1024-bit number. So, the solution space is huge.

F. SHORTHAND OPERATOR: EQUI-JOIN

To compute equi-join, one option is to use Cartesian product and then comparison operator. However, this is not efficient as it generates a large number of tuples from Cartesian product. Another option is to use a similar idea as Group-By. Suppose the join operation is computed on $A = B$ where A is an attribute on table T_1 and B is an attribute from another table T_2. We perform key updates $A' = \kappa(A, \langle m_Z, 0\rangle)$ and $B' = \kappa(B, \langle m_Z, 0\rangle)$ so that all items keys on A', B' are the same. This allows SP to compute equi-join by comparing encrypted values of A' and that of B'.

G. OPTIMIZATION TECHNIQUES

In our system, the query processing cost at user is insignificant as the user always operate on column key level and the cost is independent from database size. The encryption/decryption cost at user is relatively more expensive. These actions and the majority of work at SP are computing modular exponentials, which are much more expensive than other operations, e.g., modular multiplication. So, it is important for our system to optimize modular exponential computations.

One way to reduce the cost is to use Garner's algorithm [25]. The basic idea is to use Chinese Remainder Theorem: computing $b = x^e \bmod pq$ is equivalent to computing $b = [(x^e \bmod q - x^e \bmod p) \times (p^{-1} \bmod q) \bmod q] \times p + (x^e \bmod p)$. This reduces the computation of a 1024-bit modular exponentiation to two 512-bit modular exponentiations with some cheaper additions and multiplications. Besides, modular exponentials are commonly computed by exponential squaring. For example, to compute $x^7 \bmod n$, we may compute $x_2 = xx \bmod n$, $x_4 = x_2x_2 \bmod n$ and finally compute $x^7 = xx_2x_4 \bmod n$. So, we need 4 multiplications only instead of 6 in trivial multiplications of x itself. If multiple exponential operations are having the same base, i.e., we are computing b^{e_1}, b^{e_2}, ... for the same b, the above preparation of exponents of squares can be saved. For example, if we need to further compute $x^5 \bmod n$ after computing $x^7 \bmod n$, we just need 1 more multiplication $x^5 \bmod n = x_4x \bmod n$ instead of restarting from scratch. In our scheme, the item key with column key $\langle m, x\rangle$ and row key r can be computed as $mg^{rx} \bmod n$. It can be rephrased as $mb^r \bmod n$ with $b = g^x$. Thus, to compute the item keys of the same column but different rows, the item key generation is in fact a list of modular exponentials with the same base. This can significantly reduce the number of multiplications required to compute modular exponential for large exponents.

Furthermore, a careful choice of parameters can also help to reduce the cost. Note that the cost of modular exponential depends heavily on the size of exponent. If the exponent is small, modular exponentials can be computed efficiently. (In practice, the public exponent e in RSA (where the encryption function is $x^e \bmod n$ for plain data x) is usually set to a small value, e.g., 65537.) As we discussed above, the encryption/decryption process is a list of modular exponentials with different exponents as row-id. So, if row-ids are small, the encryption/decryption cost can be further reduced. In our system, we set row-id r as a random 32-bit number. It reduces the encryption/decryption cost while SP can't easily guess the correct value of r as there are 2^{32} possible values of r.

H. INDEXING

Indexing is an important feature in database system. The objective of index is to speed up query processing at SP. Indexing is essential as any query will incur at least $O(N)$ cost without using index where N is the number of tuples in the database. The processing cost is then not acceptable in large database, which may contain millions of tuples. It is then important to develop indexing techniques for our secure database system. On the other hand, this index has to be *secure*.

Note that indexing is by nature a security compromise. If SP skips processing certain tuples and the queries are known, SP can obtain some information about the original data. For example, if the query is to retrieve tuples with SALARY < 1000, SP can know that tuples pruned by the index are with SALARY ≥ 1000. This is the same scenario as comparison operator. So, the user has to balance between indexing effectiveness and security compromises.

[21] is a 'secure' (with user-controlled security strength) indexing scheme developed for querying on encrypted data. The general idea is to transform the original plain data into *uncertain data*. For example, in [21], data values 1, 5, 10 may be grouped into the same partition with range $0 - 10$. The partition is assigned an ID, say 1. Thus, instead of indexing on exact values 1, 5, 10, these values are indexed on the partition ID in the index. SP cannot observe the exact values of the data but the index can help to prune certain tuples on the index. The user can choose the granularity of uncertainty which controls the security compromises and the effectiveness of the index. If the uncertainty range is larger, it provides a better security protection but the index is less effective. The index here works independently with our encryption method. Before we use our operators to process the query, the database is first filtered by the index. This gives us a smaller number of tuples to be processed. Then, we will use our operators to find out the exact answer.

I. A COMPLETE PROOF FOR OUR SYSTEM W.R.T. DB+QR THREAT

In this section, we present a complete proof of security of our system w.r.t. DB+QR Threat. First, we discuss how the security analysis of our system as a whole can be broken down into individual analysis on each protocols. In secure multiparty computation (SMC), a protocol is secure if the protocol can be simulated (by a probabilistic polynomial-time algorithm) by the attacker alone with its own input, the computation result [26]. (In the simulation, the attacker randomly selects values to be the unknown inputs from other parties (DO in our case), and tries to see if the same messages in the protocol can be observed in the simulation.) The above ensures that the parties involved in the protocol cannot observe any information except its own input and the computation result. The general composition theorem in [26] states that if f, g are two secure protocols, the composition of f and g is also secure. In our system, we use an alternative and relaxed way for proving the security, which is used in [33, 22]. Each protocol may reveal some additional well-defined information $\mathcal{I}$ but not any other information. The information of a protocol revealed can be defined and bounded by the simulation technique (as we regard the information revealed in a protocol as part of the output of the protocol). The composition theorem can be still applied: if f is a protocol revealing $\mathcal{I}_f$ and g is a protocol revealing $\mathcal{I}_g$, the composition of f and g reveals at most $\mathcal{I}_f \bigcup \mathcal{I}_g$. Then, any compositions of protocols is secure if $\mathcal{I}_f \bigcup \mathcal{I}_g$ does not leak sensitive information. To conclude, to prove the security of our system, we need to (i) define and bound clearly what each of our operator reveals by simulation; and (ii) prove that the union of information revealed in all operators do not lead to security breach in our case, i.e., an attacker with DB+QR knowledge cannot derive the column keys and thus cannot derive the plain values of encrypted columns.

From now on, we assume the attacker obtains DB+QR knowledge. We refer to the pseudo codes of the operators listed in Appendix J for discussing the security of the system. Before we define the information revealed in each operator, all operators in our system will let SP know which columns are the operands of the operators. This type of information is necessary so that SP can properly compute the queries and is deemed not sensitive in our system. So, we will ignore the above revealed information in our analysis. In the followings, we formally bound the revealed information in each protocol one by one.

[Multiplication] (Algorithm 1 and Algorithm 2) Apart from the message from DO telling SP which columns are the operands of the multiplication, there is no other message. So, multiplication reveals no information.

[Key update] (Algorithm 3) The key update $C = \kappa(A, \langle m_C, x_C \rangle)$ aims to compute a column C which has the same plain values as A when decrypted while C has a column key $\langle m_C, x_C \rangle$ randomly selected by DO. The following theorem bounds the information revealed in the key update.

THEOREM 5. *The key update protocol $\kappa(A, ck_C)$ reveals only the following relationship: $x_C = x_A + px_S \bmod \phi(n)$ where x_A, x_C, x_S are the x-part of column keys of A, C and constant column S.*

PROOF. Recall that, from the view of the attacker, the operation can be generalized as $\kappa(\alpha A, ck_C)$ since the attacker does not know if there is an EC multiplication before the key update. So, the attacker views there is an unknown α in the query. The attacker observes two messages (p, q). The simulation completes if the attacker simulates a random input of DO and gets the same messages $p' = p$, $q' = q$ in its own simulation.

Based on the attacker's knowledge, i.e., $x_C = x_A + px_S \bmod \phi(n)$, the attacker picks a random $\hat{x_A}$ and a random $\hat{x_S}$ and derives $\hat{x_C} = \hat{x_A} + p\hat{x_S}$. On the other hand, the attacker picks random $\hat{m_A}$, $\hat{m_C}$, $\hat{m_S}$. The above constitutes the input of DO. Besides, the attacker picks the query parameter as $\alpha = q\hat{m_A}^{-1}\hat{m_S}^{-p}\hat{m_C}$. Thus, the column key of the column αA in the key update operator is $\langle \alpha\hat{m_A}, \hat{x_A} \rangle = \langle q\hat{m_S}^{-p}\hat{m_C}, \hat{x_A} \rangle$. Then, the attacker starts the simulation.

As from Eq. 9, the observed messages (p', q') are as follows:

$$\begin{aligned} p' &= \hat{x_S}^{-1}(\hat{x_C} - \hat{x_A}) \bmod \phi(n) \\ &= \hat{x_S}^{-1}(\hat{x_A} + p_A\hat{x_S} - \hat{x_A}) \bmod \phi(n) \\ &= p_A \end{aligned}$$

$$\begin{aligned} q' &= q\hat{m_S}^{-p}\hat{m_C}\hat{m_S}^{p}\hat{m_C}^{-1} \bmod n \\ &= q \end{aligned}$$

Thus all messages are the same as those observed in the protocol and hence the theorem is proved. □

[Addition] (Algorithm 4) In Sec. 5.3, we showed that the attacker can see the relationship between some column keys. For an addition between A, B, the attacker may observe $x_A = x_B + (p_B - p_A)x_S \bmod \phi(n)$ where x_A, x_B, x_S are the x-part of column keys of A, B and constant column S, and p_A, p_B are the messages sent to SP in facilitating the addition. In the following, we show by simulation that the above is the only information that is revealed in the addition protocol.

THEOREM 6. *The addition protocol on $C = A + B$ reveals only $x_A = x_B + (p_B - p_A)x_S \bmod \phi(n)$ where x_A, x_B, x_S are the x-part of column keys of A, B and constant column S.*

PROOF. Recall that, from the view of the attacker, the operation can be generalized as $\alpha_1 A + \alpha_2 B$ where α_1, α_2

are two unknown values. The attacker observes two pairs of messages (p_A, q_A) and (p_B, q_B) sent from DO. We show that an attacker can simulate the above protocol using a probabilistic polynomial-time algorithm. The simulation starts by the attacker picking random $\hat{m_A}$ and $\hat{x_A}$ and sets A's column key as $\langle \hat{m_A}, \hat{x_A} \rangle$. A random column key for S is generated similarly as $\langle \hat{m_S}, \hat{x_S} \rangle$. From the known information, the attacker computes $\hat{x_B} = \hat{x_A} + (p_A - p_B)\hat{x_S} \mod \phi(n)$ and sets $\langle \hat{m_B}, \hat{x_B} \rangle$ as the column key of B where m_B is randomly generated. Then, the attacker computes $\hat{x_C}$ as $\hat{x_A} + p_A\hat{x_S}$ and sets $\langle \hat{m_C}, \hat{x_C} \rangle$ as the column key of C where $\hat{m_C}$ is randomly generated. Finally, the attacker sets $\alpha_1 = q_A\hat{m_A}^{-1}\hat{m_S}^{-p_A}\hat{m_C}$ and $\alpha_2 = q_B\hat{B_A}^{-1}\hat{m_S}^{-p_B}\hat{m_C}$. Thus the columns keys of $\alpha_1 A$ and $\alpha_2 B$ are $\langle q_A\hat{m_S}^{-p_A}\hat{m_C}, \hat{x_A} \rangle$ and $\langle q_B\hat{m_S}^{-p_B}\hat{m_C}, \hat{x_B} \rangle$ respectively. Now, the attacker is ready to do the simulation.

As from Eq. 9, the observed messages (p'_A, q'_A) and (p'_B, q'_B) are as follows:

$$\begin{aligned} p'_A &= \hat{x_S}^{-1}(\hat{x_C} - \hat{x_A}) \bmod \phi(n) \\ &= \hat{x_S}^{-1}(\hat{x_A} + p_A\hat{x_S} - \hat{x_A}) \bmod \phi(n) \\ &= p_A \end{aligned}$$

$$\begin{aligned} q'_A &= q_A\hat{m_S}^{-p_A}\hat{m_C}\hat{m_S}^{p_A}\hat{m_C}^{-1} \bmod n \\ &= q_A \end{aligned}$$

$$\begin{aligned} p'_B &= \hat{x_S}^{-1}(\hat{x_C} - \hat{x_B}) \bmod \phi(n) \\ &= \hat{x_S}^{-1}(\hat{x_A} + p_A\hat{x_S} - \hat{x_A} - (p_A - p_B)\hat{x_S}) \bmod \phi(n) \\ &= \hat{x_S}^{-1}(p_B\hat{x_S}) \bmod \phi(n) \\ &= p_B \end{aligned}$$

$$\begin{aligned} q'_B &= q_B\hat{m_S}^{-p_B}\hat{m_C}\hat{m_S}^{p_B}\hat{m_C}^{-1} \bmod n \\ &= q_B \end{aligned}$$

Thus all messages are the same as those observed in the protocol and hence the theorem is proved. □

[Comparison] (Algorithm 5) Let $Y = A - B$; Here, we ignore the computation of Y since it is an addition operator, which we have discussed above. From the view of the attacker, the operation is $\alpha Y > 0$ or $\alpha Y = 0$. The steps include (i) EE multiplication with R; and (ii) a key update $\kappa(\alpha RY, \langle 1, 0 \rangle)$. Since the first operation has no message exchange, we simply discuss the second operation as $\kappa(\alpha Z, \langle 1, 0 \rangle)$ for some column $Z = RY$.

THEOREM 7. *The comparison protocol on $Y > 0$ or $Y = 0$ reveals only $x_Z = -px_S \bmod \phi(n)$ where x_Z, x_S are the x-part of column keys of $Z = RY$ and constant column S. R is the random column.*

PROOF. The attacker observes one pair of messages (p, q) sent from DO. Again, we use simulation to prove the theorem. The attackers generates random $\hat{m_S}$, $\hat{x_S}$ and set $\langle \hat{m_S}, \hat{x_S} \rangle$ as the column key of S. Then, compute $\hat{x_Z} = -p\hat{x_S} \bmod \phi(n)$. Generate a random $\hat{m_Z}$ and set $\langle \hat{m_Z}, \hat{x_Z} \rangle$ as the column key of Z. Select $\alpha = qm_Z^{-1}\hat{m_S}^{-p}$. So, the column key of αRZ is $\langle q\hat{m_S}^{-p}, \hat{x_Z} \rangle$ Note that the key update operator targets to give the column key $\langle 1, 0 \rangle$. In the simulation, the attacker observes (p', q') as follows:

$$\begin{aligned} p' &= \hat{x_S}^{-1}(0 - \hat{x_A}) \bmod \phi(n) \\ &= \hat{x_S}^{-1}(p\hat{x_S}) \bmod \phi(n) \\ &= p \end{aligned}$$

$$\begin{aligned} q' &= q\hat{m_S}^{-p}\hat{m_S}^{p}1^{-1} \bmod \phi(n) \\ &= q \end{aligned}$$

Thus all messages are the same as those observed in the protocol and hence the theorem is proved. □

[Cartesian product] (Algorithm 6) The Cartesian product requires the SP to do the same computation for each column. In the following, we provide the security proof for the process on one column and it can be applied to other columns.

THEOREM 8. *The Cartesian product protocol of two tables T_1, T_2 reveals only 2 sets of equations: (i) $x_A = px_S \bmod \phi(n)$ where x_A, x_S are the x-part of column keys of A in either T_1 or T_2 and constant column S in the other table, i.e., if A is in T_1, S is in T_2.*

PROOF. There is only 1 message p per column in the operation. We just discussion the simulation on one column and the simulations on remaining columns are similar. Similar to the above proofs, the attacker selects a random column key for S as $\langle \hat{m_S}, \hat{x_S} \rangle$. The column key of A is $\langle \hat{m_A}, \hat{x_A} \rangle$ with $\hat{x_A} = p\hat{x_S} \bmod \phi(n)$ and m_A is a random value. In the simulation, the attacker observe p' as $\hat{x_S}^{-1}\hat{x_A} \bmod \phi(n) = \hat{x_S}^{-1}(p\hat{x_S}) \bmod \phi(n) = p$. The simulation is done. □

[Aggregate] (Algorithm 7) Count is the the same as selection. So, we focus on SUM here. The major operation is the key update operation $\kappa(\alpha A, \langle m_Z, 0 \rangle)$ where αA is the column to be summed. Security of SUM is proved as follows.

THEOREM 9. *The SUM protocol on αA reveals only: $x_A = -px_S \bmod \phi(n)$ where x_A, x_S are the x-part of column keys of A and constant column S.*

PROOF. The attacker observes one pair of messages (p, q) sent from DO. Note that the above setting is the same as comparison and so can be proved in the same way. We will skip the details here. □

[EP operations] (Algorithm 8) To handle EP operation with a plain column A, a key update $\kappa(A, \langle m_C, x_C \rangle)$ is executed where $\langle m_C, x_C \rangle)$ is randomly determined by DO. Security of this key update is proved as follows.

THEOREM 10. *The key update protocol on αA reveals only: $x_A = -px_S \bmod \phi(n)$ where x_A, x_S are the x-part of column keys of A and constant column S.*

PROOF. The attacker observes one pair of messages (p, q) sent from DO. Note that the above setting is the same as comparison and so can be proved in the same way. We will skip the details here. □

In the above discussion, we have completed the first task to bound the information revealed in each operator. Basically, the attacker can set up different linear equations to connect different column keys together. Then, we will go on to show that the above revealed information cannot help the attacker to recover the column keys stored at DO and hence cannot breach the security. In particular, we show that there are a large number of different possible column keys that satisfy the above revealed information and hence the attacker cannot distinguish the true one from the others.

THEOREM 11. *Let $\mathcal{I}$ be the set of equations revealed in all the operators listed in Theorem 5 - 10 with the variables x_A's (the x-part of column keys at DO). If $\{x_A = \hat{x_A}\}$ is a valid solution satisfying the above set of equations, $\{x_A = \lambda\hat{x_A}\}$ is another valid solution where λ is an integer.*

PROOF. To sum up, there are two types of equations in $\mathcal{I}$. One is in the form of (i) $x_A = x_B + \alpha x_S$ where x_A, x_B, x_C are the x-part of column key of columns A, B, C and α is a constant known to the attacker. x_A, x_B, x_C are the variables in the system of equations. Type (i) equation can be observed in Key Update (Theorem 5), Addition (Theorem 6). The other is in the form of (ii) $x_A = \beta x_B$ where x_A, x_B are two variables and β is a known value to the attacker. Type (ii) equation can be observed in Comparison (Theorem 7), Cartesian product (Theorem 8), Sum (Theorem 9), and EP operation (Theorem 10). Suppose $x_A = \hat{x_A}$ for all column A is a solution, we test on the other solution with $x_A = \lambda\hat{x_A}$ for all columns A as follows. Type (i): LHS $= \lambda\hat{x_A}$; RHS $= \lambda\hat{x_B} + \alpha\lambda\hat{x_S} = \lambda(\hat{x_B} + \alpha\hat{x_S}) = \lambda\hat{x_A} =$ LHS. Type (ii): RHS $= \lambda\hat{x_A}$; RHS $= \beta\lambda\hat{x_B} = \lambda(\beta\hat{x_B}) = \lambda\hat{x_A} =$ LHS. Thus, $\{x_A = \lambda\hat{x_A}\}$ is a valid to $\mathcal{I}$. □

In the above, we obtain a valid solution for all $\lambda < \phi(n)$. Note that the multiplicative inverse of $x_A \pmod{\phi(n)}$ exists if and only if $\gcd(x_A, \phi(n)) = 1$. Thus, $\gcd(\lambda, \phi(n)) = 1$. This leaves $\phi(\phi(n))$ possible sets of solutions to the system of equations. Note that ϕ is the Euler's totient function where $\phi(x)$ means the number of positive integers $\leq x$ that are co-prime with x. Here, $\phi(n) = (\rho_1 - 1)(\rho_2 - 1)$ is a 1024-bit number (since ρ_1, ρ_2 are 512-bit numbers) and so $\phi(\phi(n))$ is at least 680-bit number (since $\phi(x) \geq x^{2/3}$ [23]). The solution space is too big for an attacker to recover the true column keys.

J. PSEUDO CODES OF PROTOCOLS

In this section, we list the pseudo codes of all operators from Algorithm 1 to 8.

```
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

```
Data: Column $A$ with column key $\langle m_A, x_A\rangle$ and a
      constant $p$
Result: $C = pA$ with $C$'s column key $\langle m_C, x_C\rangle$
Client-protocol:
$x_C = x_A$;
$m_C = pm_A \bmod n$;
Indicate that $C$'s encrypted column is $A$'s encrypted column;
Server-protocol:
Nil
```
Algorithm 2: EC multiplication

```
Data: (i) Column $A$ with column key $\langle m_A, x_A\rangle$; (ii)
      target column key $\langle m_C, x_C\rangle$
Result: $C = A$ with $C$'s column key $\langle m_C, x_C\rangle$
Client-protocol:
Let $\langle m_S, x_S\rangle$ be the column key of $S$;
$p = x_S^{-1}(x_C - x_A) \bmod \phi(n)$;
$q = m_A m_S^p m_c^{-1} \bmod n$;
Send $p$, $q$ to SP;
Set $C$'s column key as $\langle m_C, x_C\rangle$;
Server-protocol:
Obtain $p$, $q$ from DO;
for each row $r$ do
    Let $a_e$, $s_e$ be the encrypted values on $A$, $S$;
    Set encrypted value of $C$ $c_e = q a_e s_e^p \bmod n$;
end
```
Algorithm 3: Key udpate

```
Data: Column $A$, $B$ with column key $\langle m_A, x_A\rangle$ and
      $\langle m_B, x_B\rangle$
Result: $C = A + B$ with $C$'s column key
        $ck_C = \langle m_C, x_C\rangle$
Client-protocol:
Generate random $m_C$, $x_C$
$A' = \kappa(A, ck_C)$; // DO executes client-protocol
$B' = \kappa(B, ck_C)$; // of key update.
Set $C$'s column key as $\langle m_C, x_C\rangle$;
Server-protocol:
$A' = \kappa(A, ck_C)$; // SP executes server-protocol
$B' = \kappa(B, ck_C)$; // of key update.
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 4: EE addition/subtraction

```
Data: Column $A$, $B$ with column key $\langle m_A, x_A \rangle$ and
      $\langle m_B, x_B \rangle$
Result: A column of comparison results $C$: $= 0$ if
        $A = B$; $= 1$ if $A > B$; return $= -1$ if $A < B$
Client-protocol:
$Z = R(A - B)$; // EE addition, EE multiplication
$Z' = \kappa(Z, \langle 1, 0 \rangle)$; // DO executes client-protocol
Server-protocol:
$Z = R(A - B)$; // Corresponding server protocol
$Z' = \kappa(Z, \langle 1, 0 \rangle)$; // Corresponding server protocol
for each row $r$ do
    Let $z'_e$ be the values on $Z'$;
    switch $z'_e$ do
        case $= 0$
            $c_e = 0$; // $c_e$ is the result of this row
        end
        case $> 0$
            $c_e = 1$;
        end
        case $< 0$
            $c_e = -1$;
        end
    endsw
end
```
Algorithm 5: Comparison

```
Data: Relation $T_1$(row-id, $A_1$, $A_2$, ..., $A_h$, $S_1$, $R_1$);
      Relation $T_2$(row-id, $B_1$, $B_2$, ..., $B_g$, $S_2$, $R_2$)
Result: $T' = T_1 \otimes T_2$. $T'$ has a schema (row-id, $A'_1$, ...
        $A'_h$, $B'_1$, ... $B'_g$, $S'$, $R'$).
Client-protocol:
Let $\langle m_{S_2}, x_{S_2} \rangle$ be the column key of $S_2$ in $T_2$;
// The column $S'$ and $R'$ comes from $T_1$
for each column $A$ in ($A_1$, $A_2$, ..., $A_h$, $S_1$, $R_1$) from $T_1$
do
    Let $\langle m_A, x_A \rangle$ be the column key of $A$;
    $p = x_{S_2}^{-1} x_A \bmod \phi(n)$;
    Send $p$ to SP;
    $ck_{A'} = \langle m_A (m_{S_2})^p, x_A \rangle$;
end
Let $\langle m_{S_1}, x_{S_1} \rangle$ be the column key of $S_1$ in $T_1$;
// $S_2$ and $R_2$ are not needed as we get $S'$ and $R'$
from $T_1$ already
for each column $B$ in ($B_1$, $B_2$, ..., $B_g$) from $T_2$ do
    Let $\langle m_B, x_B \rangle$ be the column key of $B$;
    $p = x_{S_1}^{-1} x_B \bmod \phi(n)$;
    Send $p$ to SP;
    $ck_{B'} = \langle m_B (m_{S_1})^p, x_B \rangle$;
end
Server-protocol:
// First generate the encrypted row-id of all
tuples in $T'$
for each row $r_1$ in $T_1$ do
    for each row $r_2$ in $T_2$ do
        Set the encrypted row-id of the joined tuple as
        $E(r_1) + E(r_2)$;
    end
end
// Compute the encrypted values of columns
originated from $T_1$
for each column $A$ in ($A_1$, $A_2$, ..., $A_h$, $S_1$, $R_1$) from $T_1$
do
    Obtain $p$ from DO
    for each row $r_1$ in $T_1$ do
        for each row $r_2$ in $T_2$ do
            Let $a_e$ be the encrypted value on $A$ in $T_1$;
            Let $s2_e$ be the encrypted value on $S_2$ in $T_2$;
            Set the encrypted value $a'_e$ for the row
            $r_1 + r_2$ as $a_e (s2_e)^p \bmod n$;
        end
    end
end
// Compute the encrypted values of columns
originated from $T_2$
for each column $B$ in ($B_1$, $B_2$, ..., $B_g$) from $T_2$ do
    Obtain $p$ from DO
    for each row $r_1$ in $T_1$ do
        for each row $r_2$ in $T_2$ do
            Let $s1_e$ be the encrypted value on $S_1$ in $T_1$;
            Let $b_e$ be the encrypted value on $B$ in $T_2$;
            Set the encrypted value $b'_e$ for the row
            $r_1 + r_2$ as $b_e (s1_e)^p \bmod n$;
        end
    end
end
```
Algorithm 6: Procedure of transformation of one column in Cartesian product

```
**Data**: Column $A$ with column key $\langle m_A, x_A \rangle$
**Result**: The encrypted sum $\sigma_Z$ of all values on $A$.
    Note that this encrypted sum is viewed as a
    relation $T$ with 1 column $Z$ and 1 row only.
**Client-protocol**:
$Z = \kappa(A, \langle m_Z, 0 \rangle)$; // DO executes client-protocol
// Note that $Z$'s column key is $\langle m_Z, 0 \rangle$
**Server-protocol**
$Z = \kappa(A, \langle m_Z, 0 \rangle)$; // SP executes server-protocol
$\sigma_Z = 0$;
**for** *each row r* **do**
    Let $z_e$ be the encrypted value on $Z$;
    $\sigma_Z += z_e \bmod n$;
**end**
// $\sigma_Z$ is the encrypted sum w.r.t. row-id $= 0$.
```

Algorithm 7: SUM

```
**Data**: A column $A$
**Result**: $T' = T_1 \otimes t_2$. $T'$ has a schema (row-id, $A'$). $A'$
    in $T'$ are the values from $A$ in $T_1$.
**Client-protocol**:
$p = x_{S_2}^{-1} x_A \bmod \phi(n)$;
Send $p$ to SP;
$ck_{A'} = \langle m_A (m_{S_2})^p, x_A \rangle$;
**Server-protocol**:
Obtain $p$ from DO;
**for** *each tuple ($E(r_1)$, $a_e$) from $T_1$(row-id, A)* **do**
    **for** *each tuple ($E(r_2)$, $s2_e$) from $T_2$(row-id, $S_2$)* **do**
        Insert the values ($E(r_1) + E(r_2)$, $a_e (s2_e)^p$ ) to
        $T'$(row-id, $A'$);
    **end**
**end**
```

Algorithm 8: EP transformation