Numerical SQL Value Expressions Over Encrypted Cloud Databases

Sushil Jajodia
George Mason University
Fairfax, Virginia, USA
1.703. 9932952
jajodia@gmu.edu

Witold Litwin
Université Paris Dauphine
Paris, France
33.3.44542742
witold.litwin@dauphine.fr

Thomas Schwarz
University of California
Santa Cruz, CA, USA
1.813.4591041
tjschwarz@scu.edu

ABSTRACT

Cloud databases often need client-side encryption. Encryption however impairs queries, especially with numerical SQL value expressions. Fully homomorphic encryption scheme could suffice, but known schemes remain impractical. Partially homomorphic encryption suffices for specific expressions only. The additively homomorphic Paillier scheme appears the most practical. We propose the homomorphic encryption for standard SQL expressions over a practical domain of positive values. The scheme uses a version of Paillier's formulae and auxiliary tables at the cloud that are conceptually the traditional mathematical tables. They tabulate encrypted log and antilog functions and some others over the domain. The choice of functions is extensible. We rewrite the expressions with any number of SQL operators '*', '/' '^" and of standard aggregate functions so they compute over encrypted data using the tables and Paillier's formulae only. All calculations occur at the cloud. We present our scheme, show its security, variants and practicality.

1. INTRODUCTION

It is the common knowledge that client-side encryption is desirable for privacy of data outsourced to a cloud database. However, the capabilities of SQL queries on encrypted data are limited, [17], [3], [14], [7]. If we use a popular encryption such as AES, an SQL query with a numerical value expression becomes impossible to execute over encrypted data. We recall that such an expression may contain the standard operators '+', '-', '*', '/' or '^' (exponentiation). It also may contain the standard aggregate functions COUNT, SUM, AVG, VAR, or STD. SQL dialects make available other aggregate functions and different collections of scalar functions. *Homomorphic* cryptography was therefore proposed for the use of the above operators (directly) over encrypted data. A *fully* homomorphic scheme should allow for expressions with unlimited number of above operators over any encrypted numerical values. Current proposals turned however unsafe or impractical for databases, [14]. Computing with Gentry's scheme and its variations is yet billions of times slower than the same plaintext calculations [13]. An 8-bit multiplication with Gentry's scheme takes 15 minutes and a calculation that takes a second in plaintext would last three centuries, [13]. Another implementation, specifically for a cloud DB, claims 23 minutes per multiplication of two encrypted 16-bit integers [6]. Consequently, it took seven days to find a given row in 10-record DB. To make the homomorphic encryption more practical, the *somewhat* homomorphic scheme analyzed in [9] restricts the calculations to expressions with a single '*'. The authors claim execution times of 1ms per '+', and 60ms per '*'. Yet another direction are the *semi*-homomorphic a.k.a. *partially* homomorphic systems. These are notably faster, but each only deals with selected operators. Especially, the *additively* semi-homomorphic systems support basically only '+' and '-', but are orders of magnitude faster than other homomorphic systems. Among these, the Paillier cryptosystem seems the most popular [10], [17], [14], [4], [15]. 15µs may suffice per addition or subtraction over ciphertexts, [13]. Thus, e.g., 1.5s may suffice to sum up 100K encrypted values, what seems acceptable for the database world.

Database applications nevertheless need every available SQL operator and function, with possibly several invocations of each in an expression. One approach is to decompose the query so that the client post-processes the decrypted (plaintext) data for the operations impossible over the encrypted ones on the cloud, under the partially or somewhat homomorphic scheme used. Another approach uses per-row pre-computation of a plaintext expression over attributes of a table, e.g., a*b, for encryption and later upload of the computed (dynamic, virtual…) attribute [17]. Yet another approach is to encrypt only the data which are not destined for value expressions [3]. The future of all these proposals remains to be seen.

Efficiency with respect to value expressions must further preserve usual requirements on any relational DB. The prime one are select-project-join (SPJ) queries. Paillier cryptosystem is a *probabilistic* one that is the same plaintext is about never encrypted to the same ciphertext. This precludes the equality comparison blocking the frequency analysis what is good for the security. Without that capability, outsourced column cannot however obviously support then SPJ-queries. These are the must for a relational DB as widely known. A naïve approach could be to split the query so that the cloud DBMS leaves all the SPJ operations to the client, for the post-processing after the decryption of the adequately retrieved encrypted

ones. Even at the glance, one sees that such an attempt would made the outsourcing usually a nonsense. SPJ-queries require at present that the same plaintext maps always to the same ciphertext, i.e., they need a *deterministic* encryption. This one may be vulnerable to the frequency analysis, but can also resist to, [1]. The client may help, e.g., through decoy values, obfuscating power laws possibly serving disclosures, [2]. To get best of both worlds an approach perhaps practical is to use Paillier's scheme and a deterministic encryption like AES on every column subject to both: value expressions and SPJ-queries, [17]. The cloud supports then the SPJ-queries directly. In turn, the resistance to frequency analysis becomes that of the client data under the deterministic scheme used. There is also an evident storage and processing overhead.

Next in-line practical requirement for many plaintext DBs is the exploration of the total order. Related queries involve Θ-joins or GROUP BY, or TOP K etc. To perform these operations on ciphertexts, one approach is an order-preserving deterministic encryption. Any such encryption by definition discloses the order. This somewhat harms the security by itself. A safer alternative solution is again to split a query so that these operations are left for the post-processing on the client after the decryption, [17]. Order-dependent operations are less frequent than SPJ-queries. The strategy may perhaps thus be useful. In turn, it is obviously inefficient with respect to the plaintext DB, whenever the order-dependent operation reveals highly selective over a large encrypted dataset. We are not aware of any other proposal.

The final practical requirement is that of selected scalar or even aggregate functions beyond the SQL standard ones. Each SQL dialect offers some such functions out-of-the-box. The major DBMSs support also user-defined (UDF) functions. The client of an outsourced encrypted DB has to be entitled to similar capabilities.

Below, we propose a novel scheme for an encrypted cloud DB. It enables the homomorphic encryption sufficient for the above outlined standard numerical SQL expressions over a domain of positive values. While this is our primary goal, the scheme also conforms to some extent the other discussed constraints. SPJ-queries with the value expressions evaluate entirely at the cloud. The order-depend operation may occur there, mainly at additional storage cost. Perhaps surprisingly this capability does not disclose the order. Next, the client can add any scalar function, also at some storage cost. Finally, the cloud DBMS may entirely evaluate any additional aggregate functions defined by the value expressions combining the standard aggregate ones and the available scalar functions. For instance, the covariance aggregation defines that way.

Our encryption is a novel deterministic version of Paillier's scheme. The cloud DB stores the encrypted data in the tables defined by the conceptual schema. The DB contains further two auxiliary tables called each a *Scalar Function Table* (SFT). An SFT tabulates encrypted values of selected scalar functions. The client creates each SFT and uploads it to the cloud, before the DB becomes operational. Figure 1 shows the concept with related components of our scheme.

Conceptually, an SFT is an old-fashioned mathematical table, e.g., a printed table of logarithms. Actually however, an SFT usually should have many times more rows, e.g., hundreds of millions. Our scheme uses SFTs primarily to evaluate SQL expressions with operators other than '+' and '-' over encrypted data or with the aggregate functions, as people used the classical tables. Namely, the client (site) rewrites the query before sending it out, so that only the above operators and values retrieved from SFTs are used at the cloud. For example, to compute $x*y$ over encrypted x and y, the rewriting applies the high-school identity (rule) $x*y$ = antilog (log x + log y). The rewritten query looks up the tabulated encrypted logarithms that is the log function, then calculates the '+' over these values, finally, looks up the tabulated encrypted results of exp function (that is the antilog). The '+' evaluation applies Paillier's formulae we reuse within our kernel additively homomorphic scheme, Figure 1. The kernel scheme uses also our variant of Paillier's encryption and the original decryption that remains applicable to. The whole construction provides the application with a homomorphic encryption providing *in fine* more operators than effectively used. This is the Gentry's "blueprint" of constructing a fuller homomorphism through the internal use of a less powerful kernel.

Our core scheme has as the domain of input and output values of the value expressions the monetary type values 0.01, 0.02…1,000,000.00. Below, we often note the upper bound as 1M. This domain should suffice for many, perhaps most, of practical DBs. It becomes a subset of the key column of an SFT, termed SFT1, whose rows tabulate, in a column, the encrypted logarithm for each domain value. For reasons that will appear, we tabulate the natural logarithm. SFT1 contains also other rows, useful for the aggregate functions, and a column for encrypted '^' calculus. The core scheme uses further a second SFT, termed SFT2. This one tabulates antilogarithms (exponentials). It acts conceptually as a printed table, but with a practical difference compensating for the impossibility of interpolation over encrypted values.

The core SFTs suffice for standard SQL value expressions over encrypted data in the domain, including the '^' and the standard aggregate functions. Notice that we could not find how the above discussed fully or somewhat homomorphic schemes calculate the exponentiation. We could not determine their efficiency for the aggregate functions neither. As we mentioned, our scheme also always provides for SPJ-queries over the encrypted data at the cloud, unlike Paillier's scheme. We show further that SFTs can get additional columns, providing for more scalar and aggregate functions and the Θ-joins. It should be also possible, although remains future work, to have the negative values in a domain.

The storage for SFTs depends in general on the range and precision of the encrypted data. It also depends on the number of supported functions. For the core domain, 50 GB should suffice. SFTs enters then easily the RAM of a modern cloud server. An operator over ciphertexts executes then in 15 μs or less. In other words, 1.5s suffice for 100K operations. This speed should satisfy many, if not most current applications. Experimental performance evaluation remains nevertheless the further work. At present, apparently, our scheme is the only to provide the described capabilities.

Our scheme is secure for a cloud DB under the popular honest-but-curious model. Its security is due in particular to a new cloud security paradigm. The traditional one is that all the cloud node is potentially insecure, i.e., the intruder can eventually disclose any data there. Any sensitive client (meta)data stays at the client. Our paradigm, restricts insecurity to the stored (permanent, long-term…) cloud data. The transient (volatile, short-lived…) run-time internal cloud DBMS variables are secure. The client may safely send sensitive data. The paradigm proves attractive for our scheme. The data that the client sends are called below *client secret*.

Despite more than three decades of high-level research, the traditional paradigm did not lead to usable schemes. The new one

proves a way out through our scheme at least. High-level rationale is that having safely some security related data at the cloud, eases the apparent hardness of getting efficiency under the former paradigm, precluding that possibility. In turn, an effort is necessary to effectively secure a cloud DBMS. Such hope was probably unreasonable in the past. The progress in secure software engineering, e.g., through the moving target defenses, [11], makes it henceforward rational.

The next section introduces our scheme. We call it *Table-based Homomorphic Encryption*, *THE* scheme (crypto system) in short. We first overview the high-level architecture of THE scheme. Next, we recall Paillier's cryptosystem and present our kernel crypto system. Then, we define the SFTs, the rewrite rules, the value representation and storage structures for SFTs. Afterwards, we discuss the secure storage of the ciphertexts in SFTs, as shares of the client defined secret-sharing. Next, we define THE security model that fits the popular honest-but-curious one. We show that cloud DBs under THE scheme are secure under our model. Section 3 determines the processing times and storage space for SFTs. Section 4 discusses variants of THE schema. We conclude in Section 5, where we also point towards further work.

2. THE SCHEME

2.1. High-level Architecture

Figure 1 depicts the high-level architecture of THE scheme. The client and the cloud sides communicate as usual through query and data flows. At the cloud, the secure cloud DBMS manages the (cloud) DB, consisting of potentially insecure SFTs and of Encrypted Application Data (EAD). EAD component stores all the data defined by the conceptual schema. These would be in plaintext in the client DB relations if the client hasn't outsourced them.

The additively homomorphic Kernel Cryptosystem serves both the client and the cloud side. At the client, the component handles the encryption of all the outsourced plaintexts, i.e., in EAD. It serves similarly the upload of ciphertexts for SFTs and updates to. It eventually also encrypts query constants. Finally, it decrypts the retrieved data. At the cloud, the kernel scheme component provides the operators over ciphertexts in EAD and SFTs. These serve the addition and subtraction of plaintexts encrypted by the ciphertexts at the cloud, as well as the multiplication of a ciphertext by a plaintext. The kernel operators use a version of Paillier's formulae.

The Client Secret component serves the upload of SFTs and secure their use at the cloud. Queries with value expressions also use it. The secret remains in transient cloud DBMS internal variables, as already mentioned. The Rewrite Rules, finally, serve the numerical expressions within an application query, using operators other than these provided by the kernel or aggregate functions. These are rewritten into equivalent expressions using, over ciphertexts in EAD, only the kernel operators and scalar functions tabulated in SFTs. The equivalent expressions compute entirely at the cloud.

2.2. Paillier Cryptosystem

The Paillier Cryptosystem defines an additively semi-homomorphic encryption [10]. We recall only the properties most relevant to our work. Formally, plaintexts and ciphertexts are integers in Z_n for a specific n that is the product of two large primes. We use italics to denote a value x in plaintext and, sometimes, a bold font $\boldsymbol{x}$ to represent its encryption (ciphertext). Alternatively, we may denote the encrypted x as $E(x)$. We denote $D(\boldsymbol{x})$ the decryption of $\boldsymbol{x}$.

Paillier cryptography uses (g,n) as the public key with g being a random number. It encrypts a plaintext x mod n after choosing some random number r in Z_n^* as:

(1) $\boldsymbol{x} = g^x * r^n \bmod n^2$.

The scheme is additively homomorphic since:

(2) $x + y \bmod n = D(\mathbf{x}*\mathbf{y} \bmod n^2)$.

We can also add a ciphertext $\boldsymbol{x}$ and a plaintext k

(3) $x + k \bmod n = D(\mathbf{x}* g^k \bmod n^2)$.

Finally, we can multiply a ciphertext $\boldsymbol{x}$ with a plaintext k via

(4) $D(\boldsymbol{x}^k \bmod n^2) = D(k^x \bmod n^2) = kx \bmod n$.

The decryption of the ciphertext uses the formulae presented in [10]. The random r, makes Paillier scheme probabilistic, making occurrences of the same plaintext rarely encoded into the same ciphertext. Notice that (2) means that if we encrypt x with r_1 and y with r_2 then $x + y$ is encrypted with $r_1 * r_2$. We can also subtract ciphertexts since:

(5) $D(g^x * r_1^n / g^y * r_2^n \bmod n^2) = D(g^{x-y} * (r_1 / r_2)^n \bmod n^2)$.

The right side decrypts $x - y \bmod n$ encrypted with r_1 / r_2.

Paillier ciphertext at least doubles the storage necessary for the plaintext. The ciphertext of a typical 64 bit numerical value requires at least 128b, i.e., 16B per encrypted value. It believed however necessary that for adequate level of security the plaintext is 1024b at least. Smaller actual plaintexts may then be concatenated into a sufficiently large single plaintext vector, e.g., [17]. As we will show, such grouping is not necessary for our scheme.

2.3. THE Kernel Cryptosystem

By default we choose $r = 1$ for all clients. Every client gets (g, n) from DBA as *client* (encryption) *key*. To encrypt a plaintext x, the clients apply formula (1) for $r = 1$, i.e., $\boldsymbol{x} = g^x \bmod n^2$. For decryption, the client reuses the Paillier formulae in [10]. The cloud DB server(s) get from DBA the *cloud key* that is n^2. This enables the cloud to manipulate the ciphertext. The cloud reuses the Paillier's '+' calculation through formula (2). Formula (3) is of no use since cloud key does not include g. A query constant involved in '+' operation sent to the cloud must be consequently encrypted. THE scheme reuses also Formula (4). A query constant involved in '*' operation may thus remain in plaintext when processed at the cloud.

In what follows, we will call *encrypted* addition (or '+') of x and y the operation over the ciphertext $\boldsymbol{x}$ and $\boldsymbol{y}$ at the right side of formula (2). Likewise we talk about encrypted subtraction, multiplication etc., referring implicitly to the manipulations over the ciphertexts.

Paillier's formulae apply to integers. Database values are usually reals. THE kernel scheme maps these reals into integers at the client, using column scale factors. Details are in Section 2.5. If real x is scaled to integer m, we encrypt x as $g^m \bmod n^2$. The encrypted operations concern then only ciphertexts of reals with the same scale factor. The scheme decrypts a ciphertext at the client accordingly to the representation. First thus to m, through Paillier's formulae. Then to x, through the scale factor. As shortcut, below we still denote $E(x)$ or simply $\boldsymbol{x}$ the ciphertext $g^m \bmod n^2$ encrypting x.

While random r makes Paillier scheme probabilistic, fixed r makes ours deterministic. As already mentioned, the equality of plaintexts holds then over the ciphertexts, what equality comparison is

required for the select-project-join (SPJ) queries at the cloud. We come back to the issue specifically for THE scheme in Section 2.9. General comparison of vices and virtues of both classes of schemes is beyond our work as well-known, [1], [7]. All things considered, a deterministic scheme appears at present the only practical choice for a homomorphically encrypted relational DB.

2.4. Scalar Function Tables

We aim on the homomorphic encryption supporting standard numerical SQL (value) expressions. These may contain '+', '-', '*', '/' or '^' operators. They also may contain the standard aggregate functions that are COUNT, SUM, AVG (the average), VAR (the variance), or STD (the standard deviation). As usual for a major DBMS, we further aim on the extensibility of the core pack with optional dialect-dependent functions, e.g., the popular INT scalar function. We wish all these capabilities available for possibly vast majority, of 'practical' DBs. The basic need seems the money manipulation. The values are then mainly all positive integers and reals with the 2-digit typical monetary precision, all within some 'practical' range. THE scheme is conceptually independent on this range. We need however some to evaluate the storage needs. Our 'practical' range, apparently sufficient for a vast majority of DBs, is from 0.01 to 1 million (1M). Thus amounts of money, e.g., a price, should be 0.01, 0.02…1.00M$ (or €…).

We call *V'* the set of values in the range. *V'* is our *domain*. The domain values are the only allowed as input for a value expression using an operator other than '+' or computing a standard SQL aggregate function at the cloud DB over EAD, Figure 1. For the (plaintext) value expression output values they should be in *V'* or in set *V"* containing every value x^2 where $x \in V'$, while $x^2 \notin V'$. We may use *V"* for intermediate values during VAR and STD computation at the cloud, Section 2.8. We basically constraint the result *c* of any operator or aggregate function to $c \in V = V' \cup V"$. A value returned by a query should furthermore fit *V'*. A ciphertext *overflowing V'* otherwise, may nevertheless get stored in EAD. It is not expected however to become input value again. Next, for our domain, any results returned by an expression have 2-digit precision, e.g., 10/3 = 3.33. Finally, any intermediate result of a computation at the cloud overflowing *V* constitutes an *exception*. By name these are not supposed to happen, but if ever, they are the only values triggering the post-processing of an expression at the client.

As already mentioned, THE (core) scheme has two SFTs, called SFT1, Figure 2 and SFT2 at Figure 3. These tables are conceptually the old-fashioned mathematical tables. The difference is that every 'classical' plaintext *x* becomes the ciphertext $E(x)$ in SFTs and that there can be many times more values in these tables than classically. Actually, the ciphertexts, shown at the figures are encrypted further (obfuscated) for storage at the cloud, Section 2.6. We ignore the obfuscation till then.

SFT1 scheme is SFT1 (VAL, LOG, LG2). Informally, VAL is the key column with the ciphertext $E(x)$ for every *x* in *V*. LOG column tabulates for every such $E(x)$, the ciphertext $E(\text{Log}(x))$, where Log (*x*) is the result of the usual SQL scalar Log function returning the natural algorithm, i.e., ln (*x*). Likewise, LG2 tabulates the ciphertexts $E(\text{Log}(\text{Log}(x)))$. We use these for '^' operator. More formally, the plaintexts corresponding to both columns have some *p*-digit precision after the digital dot. Actually, we have $p = 8$, Section 2.5. Since Log function can return a value with more than eight digits, the functions tabulated in SFT1 are thus formally in SQL vocabulary: $E(\text{Round}(\text{Log}(x), p))$ and $E(\text{Round}(\text{Log}(\text{Log}(x)), p))$.

More in detail, SFT1 columns are as follows.

- VAL: This is the key column. For our domain, Figure 2, it contains $\{E(0.01\text{^}2), E(0.02\text{^}2)...E(0.09\text{^}2), E(0.01), E(0.02)…E(10^6), E(1000.01^2)… E(10^{12})\}$. The figure reminds that for every $x \in V$, $E(x) = g^{m(x)}$, where *m* is the integer representing *x* using the scale factor of the column.

- LOG: It tabulates with 8-digit precision $E(\ln(x))$, i.e. $g^{m(\ln(x))}$ at the figure, for every $x \in V'$. Stated differently, we have $\text{LOG}(x) = E(\log_B(x))$, where the logarithm base *B* is $B =$ e and $p = 8$. The decrypted (plaintext) LOG values range from -4.60517019 = ln (0.01) to 13.81551056 = ln (1,000,000). The rationale for our choice of *B* and *p* are experiments with various bases and precisions. They have shown that, for the bijection between VAL and LOG columns, obviously necessary, the choice of ln provides for the smallest precision that is $p = 8$. These settings minimize accordingly the size of SFT2 table, Section 3.2.

- LG2: It tabulates, also with 8-digit precision, $E(\ln(\ln(x)))$, for every $x \in V'$ such that $\ln(x) > 0$. In other words and as again at the figure, for every *x*, LG2 contains $g^{m(\ln(\ln(x)))}$. We use LG2 for ***x^y*** operations.

Our 2nd core table is SFT2 (DLG, EXP, EX2), Figure 3. This table serves as the old-fashioned antilog table. With again the difference of many more values and of their encryption. That one makes impossible the traditional interpolation of log values that may result the additions/subtractions of plaintexts *D* (LOG). Such values may fall into the "gaps" between two successive *D* (LOG) values, as well-known. The bounds of the gaps are unknown to the cloud since encrypted. Unlike for the printed tables, SFT2 has therefore conceptually a row with the antilog not only for every value in LOG, but also for every values within each gap.

DLG has in this way by far more rows than LOG. The number of rows clearly increases with *p* chosen for LOG column. As already hinted, we come back to this issue in Section 3.2. Some SFT2 rows tabulate also the antilog of antilog, for '^' calculation we explain soon. All antilog values are interpolated so to fit *V*.

The SFT2 columns are in detail as follow:

- DLG. The name stands for *dense* logarithm. As hinted, it means that DLG tabulates $E(x)$ for every *p*-digit precise value *x* such that (i) $x \geq \text{LOG}(\min(V'))$ and $x \leq \text{LOG}(\max(V'))$ or (ii) $x = \text{LOG}\, y$ with $y \in V"$. For our 'practical' *V*, DLG contains thus, in ascending order of plaintexts, round-up to $p = 8$ digit precision, (a) $E(\ln(0.01^2))$, $E(\ln(0.02^2))$… $E(\ln(0.99^2))$. Next, DLG contains $E(\ln(0.01)) = E(-4.60517019)$. We recall for sake of example that this stands for the value $g^{-4.60517019}$. This is followed by $E(-4.60517020)$, then by $E(-4.60517021)…E(0 = \ln 1)$, $E(0.000{,}000{,}01)$, until $E(13.81551056 = \ln(1{,}000{,}000))$. At the end, we have, (c), $E(\ln 1000.01^2)$, $E(\ln 1000.02^2)…E(10^{12})$.

- EXP: This column tabulates the encrypted natural antilogarithms and linearly interpolated natural antilogarithms of the plaintexts of DLG values. The plaintexts of EXP values are all in *V*. Thus their encryptions are in VAL. More precisely, given a row (***x, y, z***) in SFT2, there are two possibilities. (1) ***x*** is in LOG column. Then, $y = \text{EXP}(x)$, where EXP means here the SQL function rounded up to precision *p* of *V*. (2) ***x*** is not in LOG. Then *y* is interpolated, as it would be for printed log/antilog tables. Namely, let d_1 be the

maximal plaintext D (LOG) such that $d_1 < x$ and let d_2 be the minimal plaintext D (LOG) such that $d_2 > x$. These are the bounds on the gap where x is. Next, let it be $d_3 = (d_1 + d_2) / 2$, round up to 8-digit precision. Then, $x \leq d_3$ implies $\boldsymbol{y}$ = VAL ($\boldsymbol{v}$) with $\boldsymbol{v}$ in row ($\boldsymbol{v}$, $\boldsymbol{d_1}$…) of SFT1. Otherwise $\boldsymbol{y}$ = VAL ($\boldsymbol{v}$) with $\boldsymbol{v}$ in ($\boldsymbol{v}$, $\boldsymbol{d_2}$…). Since DLG has many times more values than VAL, as we said, most EXP column values are duplicates.

- EX2: The column tabulates the function f = EXP (EXP (x)), rounded up to the precision p of V, for every $\boldsymbol{x}$ in DLG and such that $f \in V$'. In our case, we recall, we have thus f = 0.01…1M. Like for LG2, we use EX2 for x^y clauses with both values encrypted.

Example. Consider two plaintexts 23.43 and 23.44 successive in V. In SFT1, they give rise to rows (**23.43**, **3.15401725**, **1.14867696**) and (**23.44**, **3.15444397**, **1.14881224**). These rows are also successive in SFT1 with respect to plaintexts of their VAL values. The plaintexts of their LOG values are successive in this sense as well. There is thus the gap between these, constituted by the successive values 3.15401726, 3.15401727…3.15444396. We have d_1 = 3.15401725, d_2 = 3.15444397 and d_3 = 3,15423061 for this gap. SFT2 has then for our plaintexts 23.43 and 23.44 successive rows that are (**3.15401725, 23.43),** (**3.15401726, 23.43)…** (**3.15423060, 23.43),** (**3.15423061, 23.44)…** (**3.15444397, 23.44).** The antilog of any encrypted logarithm value falling within the gap will be interpolated accordingly to **23.43** or **23.44.**

Consider now the query Select… Price*VAT… finding the encrypted plaintexts Price = 19.53$ and VAT = 1.2, reflecting the 20% tax on every price. As the calculus of x*y over $\boldsymbol{x}$ and $\boldsymbol{y}$ in previous sections mentioned, the cloud applies the high-school rewrite rule, formalized in Section 2.F, that yields EXP (LOG (**19.53**) + LOG (**1.20**)). Through lookups into SFT1 the cloud finds two rows t and t' with respectively VAL (t) = **19.53** and VAL (t') = **1.20.** It finds further LOG (t) = **2.97195175**, equal thus to E (LOG (19.53))**.** Likewise, the cloud finds E (LOG (1.20)) that is E (LOG (t')) = 0.18232156. It then performs the addition of the encrypted values using the above discussed THE kernel formula for it. The encrypted sum is **3,15427331**. The cloud looks up then SFT2 for this value in DLG column, to ultimately find the encrypted antilogarithm in EXP column. The sum is within the above gap and is above d_3. The cloud gets finally (**3,15427331)** = **23.44.** This is the correct result, as one may easily verify.

2.5. Real Number Representation

As said in Section 2.3, Paillier encryption requires integers, while DB plaintexts are basically reals. THE scheme represents therefore every real x to be encrypted in SFTs and EAD as the couple $x = (m,i)$, where m is an integer and i is a scale factor. Scale factors are known at the client only. We use m as power of g to produce the ciphertext of x that we note $\boldsymbol{x}$ by convenience as already discussed. Figure 2 and 3 show the ciphertexts in SFTs with E notation for scale factors. The actual plaintext is $m * 10^{-i}$, i.e., mE-i. For instance, 0.01 may be represented as (1000, 5), being then encrypted as g^{1000} mod n^2. This ciphertext is denoted, we recall, as E (0.01) or **0.01**. Every encrypted operation involving **0.01** will use g^{1000} mod n^2 as the operand. The decryption of the ciphertext first use the Paillier' formulae to produce 1000, finally made 0,01 again as $1000*10^{-5}$.

For SFTs, as well as for each table in EAD of course, there is single i per column. With our core V, the client should choose for VAL column at least i = 5. One reason is that the smallest plaintext to be encrypted in VAL is 0.0001. Using i = 4 could suffice, but higher i benefits the security, Section II.8. For LOG, LG2 and DLG columns we set i = 8. Thus, e.g., for the above mentioned VAL (t) = **19.53**, i.e., with the ciphertext $g^{1953000}$ mod n^2, since i = 5, we have LOG (t) = $g^{2.97195175}$. A negative number -m for these columns is, as usual, represented as the (positive) complement $n - m$ in Z_n. Thus, e.g., we represent the plaintext ln (0.01) = -4.60517019 as (positive) complement $c = n$ -460517019 mod n and encrypt it as g^c mod n^2.

2.6. Storage of SFTs and of EAD

We store every ciphertext in SFTs as one of the shares of the basic two-share secret sharing, for some secret. The obfuscation of the ciphertexts by secret-sharing serves the security of THE scheme as we show in Section 2.9. The secret is client-defined as we spoke about and illustrated at Figure 1. It is a pseudorandom number $s \in Z_n$, where Z_n is also the space of our plaintexts and ciphertexts, we recall. The ciphertext c serves as the 1st share. Accordingly to the secret-sharing principles, we then form the 2nd share as $c' = c$ XOR s and use c' as storage representation of c. In practice for our large n, we always have $c' \neq c$. The cloud DBMS extracts c for calculations from its stored representation as $c = c'$ XOR s. The s value comes to the cloud with every query using SFTs and only in such cases. The SFT lookups serving the query use then s and the latter formula. The DBMS discards s, at the end of the query at latest.

For both SFTs, key values, i.e., VAL for SFT1 and DLG for SFT2, are pseudorandom. Both tables are also basically static. We store therefore each SFT in a static hash file. Each row is represented by one record. The record (hash) key is the share c' of the table key ciphertext c, i.e., $c' = c$ XOR s with $c \in$ VAL for SFT1 file and $c' = c$ XOR s with $c \in$ DLG for SFT2 file. There are no duplicated record keys. Each record contains the key and the non-key values in the row, as in Figure 2 and Figure 3.

As usual each key c' hashes to some bucket, h (c') where h is the hash function used. For instance, h is the popular hashing by division over N-bucket file, i.e., $h: c' \rightarrow c'$ mod N. The buckets have all the same capacity of a dozen of records. The records in a bucket share the hash result over different keys, we recall. The in-bucket search for the record with the given key is basically sequential. As well-known, fixed buckets with the above capacity provide then for fast overall search. Especially in RAM that should be the usual location for SFTs, Section 3.

We actually represent EXP and EX2 values as 4B pointers to their actual 16B long VAL values. The obvious rationale is smaller storage for SFT2. We revisit this issue in Section 3.2. We discuss the actual creation of SFT files in Section 3.3.

The file structures in EAD are client-defined. Records may contain rows or columns.

2.7. Rewrite Rules for Operators

A rewrite rule modifies the original value expression to an *equivalent* one, i.e., with valid SQL syntax and yielding the same result, but executable over the encrypted data. Incidentally, the rewriting includes the attribute and column names. These are also client-side encrypted for cloud security, Section 2.9. The equivalent query has every operator other than '+' or '_' and every SQL scalar or aggregate function invoked replaced by look-ups into SFTs and calculations using only encrypted '+' or '-'. As already mentioned, the client encrypts eventually also every constant subject to '+' operation.

The clients sends the equivalent query to the cloud. The result sent back consists normally of data to be decrypted by the client. Exceptionally, data that the cloud could not process come back as well. E.g., when an operation produces the data not in SFTs, i.e., with a plaintext above or below values in *V*. The client is in charge of post-processing then. In-depth analysis of the exceptions remains future work.

The query format somehow distinguishes numerical values representing plaintext from numerical values representing ciphertext. When the cloud gets the query, it produces an execution plan. The plan is a transaction formed by SQL queries searching SFTs and, perhaps, by a host language statements.

In what follows, to simplify notation, the operands and the signs of operations are overloaded depending on the context. We do not expressively distinguish between operations over ciphertexts and over plaintexts, unless necessary. We consider first only numerical expressions that do not involve aggregate functions. They consist of column values or of the results of operations only.

(a) Addition / Subtraction: The cloud performs any addition or subtraction in ciphertext, as discussed in Sections 2.2-3. No rewrite of clause $\boldsymbol{x} \pm \boldsymbol{y}$ occurs, except, perhaps, for above mentioned plaintext constant encryption for *x* or *y*.

(b) Multiplication: if both operands are in ciphertext, then, as already mentioned, we use the well-known identity:

$x*y$ = antilog (log(x)+log(y))

Using our tabulated scalar functions this translates to the expression

E (x * y) = EXP (LOG (**x**) + LOG (**y**)).

The expression has valid SQL syntax. The cloud DBMS knows nevertheless that '+' is the encrypted addition. Next, it knows that the scalar functions are the tabulated ones. Also, since EXP has DLG as the domain, the sum must be within DLG as well, to have the result in *V*, as wished. An exception to post-process by the client occurs otherwise.

(c) Multiplication by a plaintext constant. The cloud uses rule (3). No rewrite.

(d) Exponentiation $x\text{^}y$, supposing, *x* encrypted and *y* either (i) plaintext constant or (ii) encrypted as well. The rewrite uses basically the high-school identity $x\text{^}y$ = exp (y * log(x)). For case (i), using SFTs and encrypted '*' defined through formula (4) the identity leads to the rule:

E ($x\text{^}y$) = EXP (y * LOG(x)).

For case (ii), first, observe that the basic identity expands to the following, also well-known:

$x\text{^}y$ = exp (exp (log(x) + log(log(y)))).

This leads to the following rule, using again an SQL expression referring to our tabulated functions:

E ($x\text{^}y$) = EX2 (LOG (y) + LG2 (x)),

Again, the rule refers to the encrypted '+' and the sum must be in DLG. In addition, EX2 must evaluate to a value in VAL, hence in *V*.

The rewrite rules apply recursively. In particular, - when an expression *x* with the rewrite R (x), gets equated in a query to a plaintext constant *C* in the clause x = C. The rewrite rule is then simply R (x) = **C** where **C** = E (*C*), we recall. Notice that this rule applies even if *x* is only an attribute name.

Example. (a) The popular expression K*(1+R)^Y calculates the value of capital K placed for Y years at fixed annual rate R. To be calculated over the encrypted values, our rules yield the following equivalent one:

EXP (LOG (K) + LOG (EX2 (LOG(Y)+LG2 (1+R))).

The expression has valid SQL syntax. When the client sends it out, the column names are replaced with meaningless cloud identifiers, say 123 for C, etc. The cloud evaluates the scalar functions looking up the SFTs and calculating the encrypted '+'.

(b) Consider the clause qty * price = 123. Its rewrite is:

EXP (LOG (qty) + LOG (price)) = **123**

2.8. Rewrite Rules for Aggregate Functions

The result of COUNT function is a plaintext that obviously does not need arithmetic calculations. The functions SUM, AVG, VAR, and STD do. The calculation of SUM uses the additions as discussed. No need for a rewrite. For (encrypted) AVG calculation, the basic rewrite rule is to replace it with the SUM, assumed then in *V'*, divided by COUNT. The final result is computed through Paillier formula (4). If, presumably rarely, SUM is not in *V'*, while the overall AVG result is, the rewrite is more complex, Section IV.

For encrypted variance, i.e., VAR, the rewrite rule applies König-Huygens formula (or Steiner translation) VAR(A) = AVG(A^2) – AVG(A)^2. The full rewrite goes in fact recursively further, given the rewrite of AVG itself and of '^". The 1st iteration is:

VAR (A) = SUM (EXP (2 * LOG (A))) / COUNT (A) + EXP (2 * LOG (AVG (A))).

We assume again the result of AVG (A) in *V'*. Then, E (AVG (A)) is in VAL column and 2 * LOG… in the second clause is in DLG. The The cloud finds EXP value in this row. This value is necessarily in *V'* or in *V''*. The squaring was the rationale for *V''*, we recall. Similarly for each LOG (A) and EXP summed up in the first clause. The cloud can now terminate evaluating the expression using the kernel scheme operators only.

The rewrite of STD (standard deviation) starts with the well-known property STD (A) = VAR (A)$^{(1/2)}$. Hence, we have STD (A) = EXP (1/2 * LOG (VAR (A)). Applied at the cloud, the rule means first that E (VAR (A)) must be in SFT1. The (encrypted) result of LOG (tabulated) function is in SFT1 as well and in DLG. The result of EXP (tabulated) function is logically in SFT2, although actually in VAL column in SFT1. The encrypted multiplication by ½ uses rule (4).

2.9. Security Model

We consider the encrypted data within a cloud DB using our scheme, i.e., stored in SFTs and EAD, Figure 1. Our security model is the popular honest-but-curious one. The cloud intruder may succeed accessing any data constituting our cloud DB, i.e., data in SFTs and EAD. S/he may use, e.g., some storage dump, perhaps easily available to a cloud management insider. The intruder knows our method. S/he can do off-line any calculations over the ciphertexts. S/he is ultimately "curious" to disclose the application data, i.e., to decrypt ciphertexts in EAD. But, behaves "honestly", i.e. only reads EAD. To succeed, the intruder may also 'honestly' read SFTs.

EAD respects furthermore all the well-known conditions necessary for the security of any deterministically encrypted cloud DB, [1]. These are beyond our scheme, depending solely on the application. In practice, they mean first no prior knowledge of any client plaintexts. Next, they presume high min-entropy of every column of EAD, i.e., many equally likely values. All this prevents the frequency analysis. The client may help, as already mentioned.

Next, if the client calculates at the cloud an SQL scalar function beyond the core ones, e.g., the LOG10 mentioned in Section 4, then all the tabulating columns are within SFTs. Finally, the intruder cannot access the run-time (temporary) variables within the DBMS code, instantiated for a query processing and cleared afterwards. Perhaps, - just since they are short-lived or at locations known only to the DBMS. Or, since, wisely in these troubled times, the code, with its run-time variables, is protected, e.g. by a moving target defense like [11]. We recall that these recent techniques aim on secure cloud software through secret randomization of the code and variables, blocking injections and reverse engineering. Among the variables concerned in our specific case are these with the shared secrets *s* and *s'*. They are instantiated each time a query using SFTs brings the secrets in, being all discarded at most by the query end. Are similarly concerned the variables instantiated by values retrieved from EAD or SFT files. Same for the variables that are set during the query time for search in SFT or EAD files, e.g., are instantiated with a record key.

As said, our paradigm of the secure cloud DBMS internals is new. Up to now, the research on encrypted outsourced DBs considered simply that the client never releases its sensitive data, e.g., encryption keys. As already stressed as well, this somehow natural paradigm unfortunately did not produce any practical solutions for more than three decades. In ours, the cloud DBMS internals constitute a secure space for transient metadata. Just like a bank vault for transient money. The hope is are practical algorithms that could not exist otherwise. Somehow like vaults lead to operations unthinkable without.

In turn, we displace part of the difficulty of creating a practical cloud DB encryption scheme towards that of creating an infrastructure of a secure cloud DBMS. We believe that on-going generic cloud software security research, especially the one we mentioned, makes the goal already feasible.

The data in any cloud DB using THE scheme are secure (safe) under this model. In other words, whatever is ciphertext *c* within EAD, the intruder cannot disclose the plaintext $D(\boldsymbol{c})$ in practical time. In particular every ciphertext *c* stored obfuscated in an SFT as some *c'*, is secure in this sense as well. Otherwise, if the intruder could disclose a plaintext *d* for some $E(\boldsymbol{d})$ there, then one could apply the kernel operators to calculate $E(d + d + \ldots + d)$ or $E(k*d)$, for various plaintext constants *k*. By matching the results with EAD, a disclosure of all outsourced plaintexts could follow.

As stressed already, we consider EAD immune against any attacks possibly disclosing values in a DB, independently of the deterministic encryption used. As we will show, EAD alone is then secure since the intruder cannot disclose *g* from any contents it may have. Reading SFTs is of no help to the intruder. The overall rationale the missing knowledge of the client secrets *s* and *s'*. The intruder can't get them under our model neither from stored nor from run-time values. *A fortiori* the intruder can't get knowledge of any couple (*c, c'*) with *c* in EAD stored obfuscated as the share *c'* in VAL, calculated in Section 2.6. This blocks any use of a property of *c* in SFTs to disclose the plaintext. The security of EAD follows.

Now, in detail, first, to disclose *g*, the intruder could start with guessing likely scale factors. Then, could pick up a ciphertext *v* expected to be *g* and try plausible values of *m*. This until equality of $v^m \bmod n^2$ possibly occurs with some ciphertext in EAD. Trying out accordingly perhaps even every *c* in EAD. This process should be obviously already typically rather tedious. In addition, under our core scheme there are only values $g^m \bmod n^2$ with $m > 1$ in the cloud DB. This is due to the choice of the scale factors ($s \geq 5$ we recall). The appearance of *g* value for any visited *v* is extremely unlikely. The discussed attack is hopeless in practice.

Same conclusion applies to the guess of any two values *v*, *v'* as some g^m and g^{m-1}, (modulo n^2), yielding *g* value as $g = v / v'$. Likewise, the intruder cannot calculate *g* from some ciphertext $g^m \bmod n^2$ with $m > 1$, either in SFT or EAD. Here, *m* is the discrete logarithm of *g* private and large. While no algorithm calculating the discrete logarithm for large *g* in practical time is known, the 128b wide ciphertext is also believed not large enough for sufficient security. But, this belief concerns *g* used as a public key, hence known to the intruder. This is not our case, what basically multiplies the complexity of any algorithm proposed for the discrete logarithm computation by a factor up to 2^{64}. On the average the complexity increases 2^{63} times as *g* is randomly picked up in Z_n, we recall. The result appears high enough for many years of number crunching by any supercomputers at present.

Next, with respect to the secret sharing, observe first that the intruder cannot disclose the secret from SFTs alone. The reason is that whatever is ciphertext *c* present in SFTs, the intruder cannot determine *c* from *c'* obfuscating it. The ciphertexts are in all the columns of SFT1 and DLG column in SFT2, we recall. *A fortiori;* the intruder cannot determine $D(c)$.

The two-share secret sharing is indeed known to be secure for its secret assumed pseudorandom, as in our case, provided the 1st share that is here *c*, is also pseudorandom enough to resist any frequency analysis. This is our case as well. First, our encryption formula $g^m \bmod n^2$ acts as the hashing by division. This one usually randomize the hashed values well. Next, in every column with the shares, every (hash) value is unique, keeping the distribution uniform. We have a single secret hidden into two shares as many times as there are obfuscated values *c'* in SFTs. Since each *c* is pseudo random, *c'* values cannot disclose the secret. The intruder cannot thus calculate any *c* from its *c'* by analyzing only the related columns, i.e., cannot disclose any *c*.

In contrast, the values in EXP and EX2 are not the shares. The columns have also duplicates. It is easy to see that for every $v_1, v_2 \in V$ with $v_1 > v_2$, more rows point to v_2 than to v_1. The intruder may accordingly order the shares in VAL. Moreover, the intruder may luckily guess the scale factors and logarithm base used. E.g., if the client simply applies the core scheme. A count of duplicates may then identify a plaintext. For instance, the count of 42680 identifies 23.43 from our example in Section 2.4.

However, for any record in SFT1 file, this knowledge still does not disclose any ciphertext *c* that *c'* obfuscates. In other words, for any above $D(c)$, the value of *c* obfuscated in VAL column remains unknown if the secret is unknown. We consider the latter to be the case of the intruder from now on. We will show progressively that it is actually so. For instance therefore, *c* with $D(c) = 23.43$ above remains unknown to the intruder. The intruder cannot disclose

therefore the plaintext $D(c)$ for any c in EAD using SFTs alone. Natural way to progress is then to explore both SFTs and EAD. Basically, - to identify for some c' in VAL 'its' c in EAD. Guessing pairs (c', c) by brute force however, i.e., by trying for given c' every c there, as possibly sharing $D(c)$, is clearly a nonsense. One must restrict number of the trials. Possibly towards a single pair.

One hypothetical way to follow then is the access path from some c in EAD to the (unique) bucket in SFT1 file containing a single record. The plaintext found for c' there would be $D(c)$, hence the disclosure would succeed. Such bucket may exist with a reasonable probability in SFT1 file, as known for hash files. However, to access a bucket within SFT1 file one needs to hash c'. Whatever is c, the intruder cannot determine c', as discussed. To disclose $D(c)$ is then impossible as well. Same holds for the other way around. That is, - the access to the bucket with unique c in EAD, starting from some c' in VAL. Clearly, without knowing s, no exploration of our physical data structures for SFTs and EAD may relate c' in SFTs to 'its' c in EAD and vice versa.

The same rationale blocks the only remaining way. That one is the sequential (brute force) search through EAD alone. Since g is unknown, the "Holy Grail' ciphertext should have the plaintext being a unique solution to some algebraic identity or equation. It thus should be the only equalizing two selected different expressions. For instance, 2 is the only to solve $x + x = x*x$. Likewise 3 uniquely solves the identity $x + x + x = x*x$ etc. If one may calculate an identity over EAD, e.g., may test $E(x + x) =? E(x*x)$, the disclosure obviously follows. The performance analysis in Section 3 shows that this could happen then in less than an hour for an EAD with even dozens of millions of values. The result would be similar for the search through VAL if the stored values there were not obfuscated. A rapid disclosure of entire EAD could follow the wise use of kernel operators above discussed.

However, the intruder cannot even calculate the '+' above, because of s' obfuscation of EAD ciphertexts. Besides, '*' unfortunately (for the intruder) requires in addition again SFTs. Not knowing the secrets, the intruder can't thus test the identity. Same applies to above mentioned additional SQL scalar functions, e.g., LOG10n that could serve '*' alternatively. Their tabulations are in SFTs hence are secret for the intruder as well. Identities using operators other than '*', e.g., $x + x = x\hat{}x$ solved by $x = 2$, obviously can't help neither. Algebraic identities always use at least one non-kernel operator. This way of proceeding thus still cannot bring the disclosure under our model.

Finally, the intruder could consider getting knowledge of "useful" values while they are processed for a query. As already mentioned, they are then and only then, in some run-time variables within the DBMS. The ciphertext c could come from EAD. Or could be brought by the value expression, e.g., in the clause $2\hat{}x$, with $c = E(2)$. In both cases, some run-time variables are instantiated then also with the secrets s and s', c' and $h(c')$, to read the relevant bucket, e.g., with $LG2(c)$. Reading s or c and c' or c and $h(c')$, would again trigger a disclosure. However, as we discussed, the run-time variables are out of reach for the intruder in our model. This strategy does work neither thus. In particular, the intruder is definitively unable to disclose the secrets. As we only supposed till now, we recall, since proving this disclosure impossible from SFTs alone. Altogether, THE scheme is thus secure under our model.

3. PERFORMANCE ANALYSIS

3.1. Processing Time

Processing a rewritten query requires a few lookups (searches for records), XORing with the secrets s and s', and time for one or a few computations of the kernel operators. The latter times clearly dominate. The overall time to process an operator other than the kernel one, should be thus only a few times longer than for a single encrypted addition. The result is practical as long as that time is. Authors of [14] report the actual speed of about 12 µs per Paillier's addition. This is the slowdown of eighty times with respect to the plaintext addition. Nevertheless, this is still, e.g., only 1.2 s for 100K additions. Processing speed of that order remains practical for a DB. The same number of plaintext additions needs 14 ms, [14]. In comparison, the Gentry scheme would need over four years. Likewise, the somewhat homomorphic scheme we cited, would still require 100 s. This would clearly be usually impractical, being also about eighty times more than with our scheme.

The multiplication operation with our scheme requires the time of an addition and of four lookups: two for LOG values, one for DLG and for EXP pointer in the same record, and one follow up into VAL. As it appears below, SFTs may and therefore preferably should, reside in RAM of a modern (database) server. The lookup time through a bucket of a hash RAM file should be up to 1µs. Time to get ciphertexts from their storage representations is negligible since we use XOR. The result is 16 µs per multiplication, i.e., 1.6 s for our 100K multiplications. This should be obviously usually practical. The result holds obviously for the division. We recall that the somewhat homomorphic scheme that appears the fastest supporting both '+' and '*' known till now, requires 60 ms per '*'. In other words, it appears 4000 slower, notwithstanding its inherent limitation on '*'.

If SFTs better do not reside in RAM of the cloud server, the flash storage is the next candidate on the memory hierarchy. The lookup time is essentially the flash read time, i.e, about 1 ms. The lookups then dominate the time for the multiplication and brings the response time for a single '*' to 3 ms, i.e., to five minutes per 100K. The latter timing might be already long for some users. The location of SFTs in flash storage instead of RAM should be therefore possibly avoided. Nevertheless, even so, the processing is still twenty times faster than for the somewhat homomorphic scheme. Finally, with their 5-10ms per access on the average, the next in the memory hierarchy magnetic disks, are clearly too slow to appear as practical for SFTs.

The evaluation of '^' shows that for RAM resident SFTs, 100K such operations should take also less than 2 s. We skip fastidious details and evaluation for the flash. The calculus of an aggregate function depends mainly on the number of selected values. In practice the latter should be usually under a few thousands. The result should be under a second for SFTs in RAM. Again we skip the details.

3.2. Storage for SFTs

We consider our core domain V. We evaluate the resulting storage first for SFT1 values, then for SFT2 values. Next, we determine the total storage for the hash files with these values. Then, we discuss formulae for the storage, applicable to variants of core choices. We determine finally the expected storage amount per core domain value as the thumb rule for comparing variants.

As already said, Paillier's ciphertext requires at least 128b, i.e., 16B. For SFT1, the key column VAL has $|V| = 200{,}000{,}000$ values. These

values require thus 16 * |V| / 1024^3 ≅ 3 GB of storage. The non-key column LOG has only |V'| = 100,000,000 values. Those need 1.5 GB. LG2 column has a value E (ln (ln (x))) only for $x \in V'$ and only if ln (x) > 0. The difference to |LOG| is negligible. LG2 values require thus practically 1.5 GB as well. SFT1 values need therefore 6 GB total.

For SFT2, DLG column has, first, the ciphertexts for the logarithms of the values forming our domain V'. These are the 100,000,100 ciphertexts in LOG. DLG has also all the values within each gap among logarithm values encrypted in LOG. The natural logarithms range for V', we recall, from 4.60517018 to 13.81551055, with the bounds provided with our precision p = 8. Altogether, for our V' and p, we need thus in DLG 1,842,068,074 values. Notice that this is eighteen times more than in LOG.

DLG has furthermore the ciphertexts for V'' values. We recall that these values are necessary for the aggregate functions. There is one such value first for each $x \in V'$ such that $x > 1000$. We have then $x^2 > 1$M and so $x^2 \in V''$. We have 99,900,000 such values in V''. We also have in V'', 9 values for $v < 0.1$, hence $v^2 < 0.01$, our lower bound on V'. The grand total is 1,941,968,083. This leads to 29 GB for DLG values.

Notice that for our V, the choice of the logarithm base $B \neq$ e, would lead to precision $p > 8$. Hence, it would lead to even more values in DLG. Experiment, e.g., with the popular bases B = 2 or B = 10.

EXP and EX2 columns contain only the already mentioned 4B pointers. Such pointers suffice for up to $2^{32} - 1$ values, hence largely enough for our "practical" V. The storage for EXP values is thus 7.3 GB. EX2 column only has the values that fit V'. The maximal possible DLG value is ln (ln (10^6) = 2.62579191. We have therefore 460,517,019 + 100 + 262,579,192 = 723,096,311 values in EX2. They need 2.7 GB. Altogether, SFT1 and SFT2 values require thus 6+29+7.3+2.7 = 45 GB of storage.

As we said our storage structure for an SFT column is a hash file. The load factor can then exceed 90% with almost no access performance deterioration by collisions. 50 GB total should suffice thus for our core SFTs.

Mass-produced workstations and cloud servers, offer now routinely 64 GB and easily up to 512GB of RAM. As already mentioned, our SFTs may therefore reside in RAM storage and thus typically should. Alternatively, there are cheap flash cards and SSDs for those, as mentioned already as well.

More generally, the storage amount for SFT1, say S_1, calculated as above, is linear with the number of values in VAL. Thus let it be x = |SFT1|. Next, let w be the ciphertext width, w = 16B in our case. We have at least $S_1 = O(3*w*x)$. Each optional column we talk about in Section 4 costs for its values $w*x$ bytes. This, except for the plaintext although also secret RANK costing $8*x$ bytes for modern 64 bit arithmetic. For SFT2, the cost is $S_2 = O((w + 4+4) * x * o(x, b))$. Here, for the first sum, we consider again 4B pointers for EXP and EX2, sufficing for up to $2^{32} - 1$ pointed values, we recall. For the last expression, o denotes the overhead of dense log, o = |DLG| / x. This one depends on x and B. For our V, $B = e$ appears best as we already pointed out, with still almost 2000 % overhead. For other values of x another B can perhaps lower o. Finally, since we needed 50 GB for |V'| = 100M, the thumb rule for our (core) scheme is 0.5 KB per domain value.

3.3. SFT Upload and Update

To create the SFTs at the cloud, the client has to create SFTs ciphertexts elsewhere than at the cloud DB server(s) and upload them there. Suppose first that the client creates all the ciphertexts at its own site. We call this scheme *centralized.* To evaluate the time for SFTs creation at the cloud then, say T, let w is the total number of encrypted columns in an SFT, and x the average cardinal of a column. Then T is proportional to $w*x$. T includes the encryption time at the client, say $T_C = O(w*x*c)$, where c is here the time to encrypt a single value. It also includes the transfer time and the time to create the tables at the server. The client-side encryption and the other operations can be parallel. It suffices to send group encrypted values, while the encryption goes-on. T_C becomes then the dominant component of T.

Our basic hardware assumption for the centralized scheme is that the client site is a PC with usual high or low speed connection to the cloud. This was the basis for the experiments reported in [12] and [5]. These papers are short and lack important details. Nevertheless, it appears that T_C for 100K values is 3 ÷ 15 minutes. Notice that the upper bound is ten-year old, while the lower is only two-year old. We have the total of about 2,241M values to encrypt: the LG2 column of SFT1 and the DLG column of SFT2. The latter reuses the 100M ciphertexts of LOG column. Even the best case, this leads to 47 days. This is hardly practical, although remains subject to caution, requiring experiments on current hardware. To this time adds up the creation of the pointers in EXP and EX2 columns of SFT2 at the client that must follow that of SFT1. That time is however relatively negligible. One may expect four hours, assuming reasonably at most 5μs per record creation, for our almost 2,700M records to produce for EXP and EX2 columns.

While the centralized scheme is the classical configuration, a modern way towards a practical T_C is clearly the use by the client of some popular MapReduce tool distributing the calculations over a *trusted* cloud. That cloud is other than the one with the cloud DB, being chosen by the client as safe. For instance, if the DB is outsourced to Google cloud, Amazon EC2 could be the trusted one. One way is that the client creates then locally a file with all the plaintexts for VAL, say PVL, and, similarly a file PDL of plaintexts for DLG column. The tool dynamically partitions then each file over the trusted cloud during the Map phase. For each value of PVL and of PDL, each node calculates in parallel the rows of each SFT and forms the records. The Reduce phase sorts the records by bucket addresses in the cloud DB files. The results are uploaded to the cloud DB as inserts to the buckets, initially empty.

Provided the trusted nodes have the same throughput, the partitioning by Map/Reduce over N nodes should reduce T_C of the centralized scheme on the average N times. The client requests thus enough nodes for an acceptable value. If this one is, e.g., on the average 1 h, 1100 nodes may do. The total time should be somehow longer. First, there is time spent to create PVL and PDL files, possibly in RAM. Next, there are transfers of GBytes of plaintexts towards the trusted nodes, during the Map phase. There are also transfers towards the cloud DB from the reducers. There should be usually high parallelism between theses transfers, as well as with the encryption calculations. The dominating transfer is the one of 45G of SFT values for our core domain from the reducers to the cloud DB. For the Internet fiber optics transfer speed of 200+ MBs, the overhead to T_C should be altogether a couple of minutes.

An update to SFTs should merely consist of appending optional scalar functions we describe below. Using the basic scheme, this operation has the time complexity of $O(|V'|)$ per function. This could still appear impractical for our core V', lasting mainly because of the encryption time perhaps 2 days. Again, one can make it almost N times faster through the N-node trusted cloud.

4. VARIANTS

We now outline some variants tailoring the core scheme to specific need. First, the storage for SFTs can be reduced. Results in [16] and [17] point to impressive almost 90% savings in ciphertexts representation. One should study the application of these techniques to SFTs. On the simple side, consider, e.g., the secret decision to represent plaintext 2.34 as 1.22 + 1.12. We may represent **2.34** then as two pointers. Instead of 16B we use 8B only. The application of the strategy on massive scale to SFTs leads to almost 50% saving. The obvious price to pay is longer processing time. For our **2.34** and a 16B ciphertext, we about double the '+' and '*' times.

Next, the domain V' of our core scheme has $|V'| \cong 100M$. The algorithmic is nevertheless independent on $|V'|$, provided V' contains positive numbers only. An obvious direction is thus to use a smaller or larger V', perhaps fitting better a DB. The storage space for SFTs is affected linearly, as we have shown. A logarithm base B other than our e one may then perhaps provide to a smaller DLG for a given domain. Providing therefore a smaller SFT2 storage than our base would do.

Yet another direction is more rewrite rules. For instance the expression $d = a * b / c$ evaluated as discussed till now may result in an overflow of V' after $a * b$ calculation, while d may still be in V'. An alternative rule could rewrite d as $(a / c) * b$ perhaps avoiding the overflow. Likewise, the straight calculus $a*b - a*c$ could fail because of $a*b$ overflow. But, perhaps the rewrite $a * (b - c)$ would do. The high-school books seem a pond for more rules.

The issue occurs also for the aggregate functions. For AVG, in particular, the following recursive formula may avoid an overflow:

$$AVG(A_1, A_2 \ldots A_{n+1}) = n / (n+1) * AVG(A_1, A_2 \ldots A_n) + 1/(n+1) A_{n+1}.$$

The formula extends to VAR computation.

An important direction is furthermore the expansion of SFT1 with *optional* scalar functions. These are functions other than the core ones we have discussed. Each optional function give rise to a dedicated column, with its value entering the core row and record. For instance, a useful add-up may be the column INT tabulating in each row $(v,\ldots)$, the ciphertext INT (v) of the popular function. This could lead e.g., to two rows with (VAL…INT) being (**23.43**…**23**) and (**23.44…23**). Likewise one may add the RANK column tabulating in plaintext the descending or ascending order of plaintexts of ciphertexts in VAL. We recall that RANK plaintexts would be nevertheless stored obfuscated as the client secret shares. Under our security model, they would not disclose the order, what the order preserving encryption schemes naturally do, [17]. The function allows to compute at the cloud the ORDER BY clause on value expressions, and thus TOP k, MIN and MAX aggregate functions. Likewise it allows for Θ-joins. The RAM sizes up to 512 GB discussed in Section 3, easily allows for many more optional functions.

Another goal is a domain with zero and negative numbers for operators other than '+' and '-'. One way towards it is, first, an additional column, say MVL, in SFT1, tabulating –**a** for each **a** in VAL, assuming the representation of the negative numbers as complements mod n in $0...n$. Next, the rewriting rules have to be revised and optional functions added, to deal with the sign of a value expression where an operand may be negative. E.g., the rule for a * b where perhaps $a,b < 0$, may become SIGN (EXP (LOG (ABS (a)) + LOG (ABS(b)))), where SIGN depends trivially on the operands. Some rules have to deal then with forbidden cases, e.g., LOG (0) or $a^{1/2}$ for any $a < 0$.

Finally, with respect to SFT1 and SFT2 upload, an application of a scheme in [8], instead of MapReduce, would first bring the advantage of the guaranteed time limit over T_C at each node, e.g., $T_C \leq 1h$. Unlike for MapReduce tools, the limit would hold even if some trusted nodes had smaller throughput, even heavily, as often in practice. Next, the client would comparatively save the overhead time and storage related to the creation and transfer to the nodes of PVL and PDL files. These would exist only as virtual files, partitioned over the trusted cloud nodes. For our SFTs, 1100-node cloud could then again suffice for the previously discussed 1h limit on T_C, if the trusted nodes have the same throughput. More than 1100 nodes would get involved dynamically for heterogeneous throughput. Known MapReduce tools lead then to processing times perhaps even much longer than the expected T_C/N, executing over statically allotted $N = 1100$. However, while there are numerous MapReduce tools, no application of schemes in [8] appears known as yet.

5. CONCLUSION

Despite thirty five years quest, a practical fully homomorphic encryption remains yet the Holy Grail. THE scheme offers the homomorphic capabilities appearing practical for many, perhaps most, DBs. It evaluates the numerical SQL expressions over encrypted data with any number of operators other than '+' and '-', and with standard SQL aggregate functions. It uses additively homomorphic Paillier-based kernel cryptosystem, auxiliary tables tabulating selected scalar functions and rewrite rules. It also uses a novel paradigm of the cloud DBMS secure for the transient client metadata. The research until now presumed to the contrary that the client never sends out such data. Our paradigm appears practical, through the progress towards secure cloud software, using the moving target defenses especially.

THE scheme processes the expressions normally entirely at the cloud. It is secure under our security model against honest-but-curious intrusions. THE scheme is finally deterministic, supporting thus SPJ queries over ciphertexts. Efficiency of SPJ operations is not subject of our work. Processing such queries entirely at the cloud is nevertheless well-known to be the key to the success of relational cloud DBs. To our best knowledge, THE scheme is the first with this capability to the extent shown.

Further work should address the prototyping. As for other proposals, this is necessary to evaluate THE scheme in depth. We outlined several variants. Future efforts should aim at these as well.

6. ACKNOWLEDGEMENTS

Discussions with Ken Smith at the 2013 Cloud Security Meeting at CSIS (GMU), after the presentation [13] laid the basis of this work.

REFERENCES

[1] Boldyreva, A, Fehr, S. and O'Neill, A.: On Notions of Security for Deterministic Encryption and Efficient Constructions without Random Oracles. *In Proceeding of the 28th International Cryptology Conference, (Crypto '08).* (Santa Barbara, CA, August 17-21, 2008). Lecture Notes in Comp. Science, Vol. 5157, Springer, 2008, D. Wagner ed., 335–359.

[2] Blank, A., & Solomon, S. (2000). Power laws in cities population, financial markets and internet sites (scaling in systems with a variable number of components). Physica A: Statistical Mechanics and its Applications, 287(1), 279-288.

[3] De Capitani, S., Foresti, S., Samarati, P. Selective and Fine-Grained Access to Data in the Cloud. *Secure Cloud Computing.* Lecture Notes in Computer Science Vol 5735, Springer, 2014. Jajodia, S. & al eds., 123-148.

[4] Encounter software library. http://plaintext.crypto.lo.gy/article/658/encounter.

[5] Farah, S., Javed, Y., Shamim, A., Navaz, T.: An experimental study on Performance Evaluation of Asymmetric Encryption Algorithms. *In Recent Advances in Information Science, Proceeding of the* 3rd *European Conf. of Computer Science, (EECS-12*) (Paris 2012).. WSEAS Press, 121-124.

[6] Gahi, Y & al. A Secure Database System using Homomorphic Encryption Schemes. *In Proceedings of 3rd Intl. Conf. on Advances in Databases, Knowledge, and Data Applications (DBKDA 2011)* (St. Maarten, Netherlands Antilles). International Academy, Research, and Industry Association (IARIA), 54-58.

[7] Hacigumus, H. & al. Search on Encrypted Data. *Secure Data Management in Decentralized Systems.* Advances in Information Security, Springer, 2007, Yu, T., Jajodia, S. eds., 383-427.

[8] Jajodia, S., Litwin, W., Schwarz, Th.: Scalable Distributed Virtual Data Structures. *In Proceedings of Second ASE International Conference on Big Data Science and Computing.* Stanford, 2014.

[9] Lauter, K. & al. Can Homomorphic Encryption be Practical? *In Proceedings of 3rd ACM workshop on Cloud computing security.(CCSW '11)* (Chicago), 113-124.

[10] Paillier, P. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In Proceedings of Advances in Cryptology, Intl. Conf. on the Theory and Application of Cryptographic Techniques, (EUROCRYPT '99) (Prague). Lecture Notes in Computer Science Volume 1592, Springer, 1999, Jacques Stern (ed.), 223-238.

[11] Portokalidis, G., Keromytis, A., D. Global ISR: Towards a Comprehensive Defense Against Unauthorized Code Execution. I*n : Moving Target Defense. Advances in Information Security*. Springer, 2011, Jajodia & al eds. 49-76.

[12] Subramaniam, H., Wright, R. N., Yang, Z.: Experimental Analysis of Privacy-Preserving Statistics Computation. *In Proceedings of Workshop on Secure Data Management, (SDM04), held in conjunction with VLDB04.* Springer, 2004, W. Jonker and (Eds.), 55–66.

[13] Smith, K., Allen, D., Sillers, A., Lan, H., Kini, A.: How Practical Is Computable Encryption? http://csis.gmu.edu/albanese/events/march-2013-cloud-security-meeting/04-Ken-Smith.pdf

[14] Smith, K., Allen, M., D., Lan, H., and Sillers, A. Making Query Execution Over Encrypted Data Practical. *Secure Cloud Computing,* Springer, 2014, Jajodia, S. & al eds, 173-190.

[15] Thep. The homomorphic Encryption Project. https://code.google.com/p/thep/

[16] Tingjian, G., Zdonik, S.: Answering aggregation queries in a secure system model, *In Proceedings* of 33rd Intl. Conf. on Very Large Databases (VLDB 07) (September 23-28, 2007, Vienna, Austria). VLDB Endowment, 519-530.

[17] Tu, S., Kaashoek, M. F., Madden, S., Zeldovich, N. Processing Analytical Queries over Encrypted Data. *Proceedings of the VLDB Endowment*, 6, 5, (March 2013), 289-300.

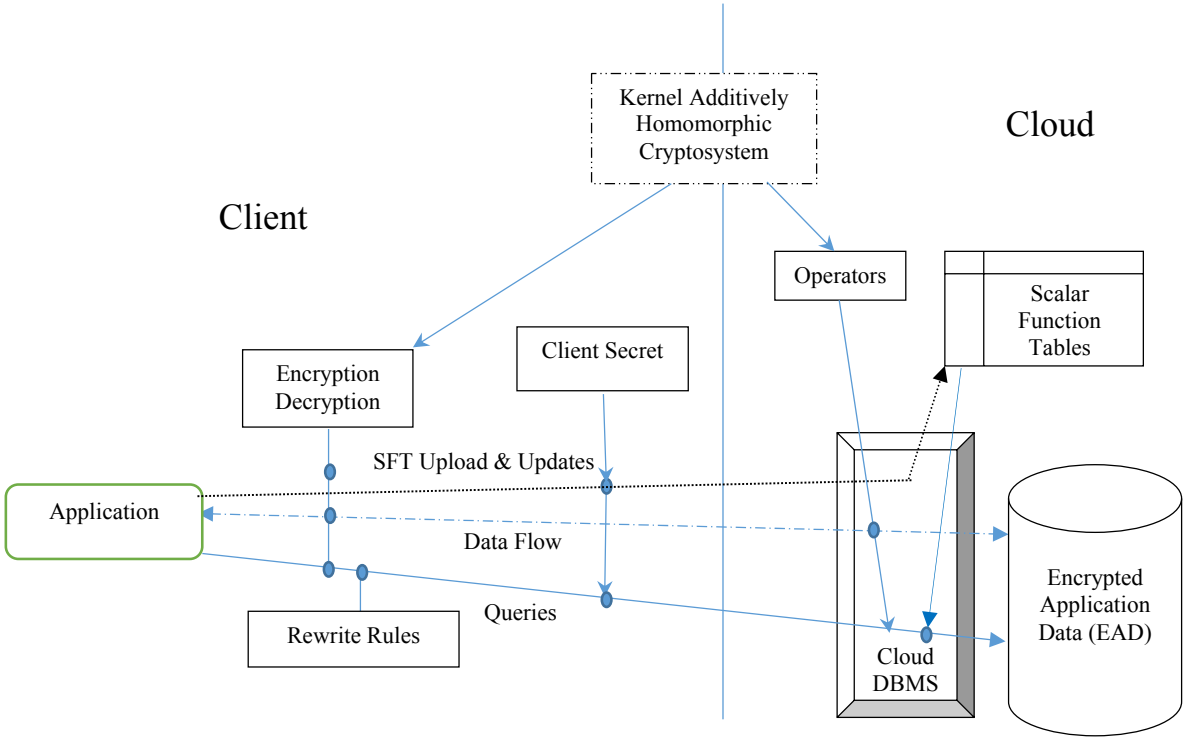

Figure 1. High-level architecture of THE scheme.

VAL	LOG	LG2
g^((0.01^2)E5)		
g^((0.02^2)E5)		
…		
g^((0.09^2)E5)		
g^(0.01E5)	g^((ln 0.01)E8)	
…	…	
g^(1.00E5)	g^((ln1.00)E8)	
g^(1.01E5)	g^((ln1.01)E8)	g^((ln(ln1.01))E8)
…	…	….
g^(1,000,000.00E5)	g^(ln(1,000,000.00)E5)	g^(ln(ln(1,000,000.00))E8)
g^((1000.01^2)E5)		

Figure 2. Table SFT1. Ciphertexts are integer powers of *g*. Plaintexts are reals, scaled up to become integers (Section 2.5). Scaling factors 5 and 8 are in E notation. Ciphertexts are stored obfuscated through the client secret *s*.

DLG	EXP	EX2
g^(-921034037)	g^((0.01^2)E5)	
g^(-782404601)	g^((0.02^2)E5)	
…	…	
g^(-481589121)	g^((0.09^2)E5)	
g^(-460517019)	g^(0.01E5)	g^(1.01E5)
g^(-460517018)	g^(0.01E5)	g^(1.01E5)
…	…	…
g^(0)	g^(1.00E5)	g^(2.72E5)
g^(1)	g^(1.00E5)	g^(2.72E5)
…	…	…
g^(262611682)	g^(13.82E5)	g^(1,000,000.00E5)
…	…	
g^(1381551056)	g^(1,000,000.00E5)	
g^(1381553056)	g^((1000.01^2)E5)	

Figure 3. Table SFT2 with its ciphertexts. These are actually stored as secret shares in DLG and as pointers to their VAL values displayed here otherwise. Plaintexts are again reals, represented as integers. Scale factor is implicitly 8 in DLG column and explicitly 5 in EXP and EX2.