10-19-2017

Dodrant-Homomorphic Encryption for Cloud Databases using Table Lookup

Thomas Schwarz
Marquette University, thomas.schwarz@marquette.edu

Contents

Dodrant-homomorphic Encryption for Cloud Databases Using Table Lookup

Thomas Schwarz
Marquette University, Milwaukee, WI

Abstract:

Users of large commercial databases increasingly want to outsource their database operations to a cloud service providers, but guaranteeing the privacy of data in an outsourced database has become the major obstacle to this move. Encrypting all data solves the privacy issue, but makes many operations on the data impossible in the cloud, unless the service provider has the capacity to decrypt data temporarily. Homomorphic encryption would solve this issue, but despite great and on-going progress, it is still far from being operationally feasible. In 2015, we presented what we now call dodrant-homomorphic encryption, a method that encrypts numeric values deterministically using the additively homomorphic Paillier encryption and uses table lookup in order to implement multiplications. We discuss here the security implications of determinism and discuss options to avoid these pitfalls.

Introduction

There is an increasing trend towards moving databases to the cloud, more specifically, towards Databases-as-a-Service (DBaaS). Google's Big Query, Amazon RDS, and Microsoft SQL Azure are commercial examples. DBaaS offers many advantages[5,6] for query processing in the cloud, but actual or perceived lack of privacy is a major obstacle to wide-spread adoption. Encryption solves these issues since it protects “Data at Rest”, but encryption also obstructs query processing. The obvious solution is to ship all data back to the data owner or to a trusted site,[8] but most tables are too large to make this practical and it would defeat the purpose of DBaaS. Sophisticated systems such as Monomi[14] still use the client for processing when other options fail.[7] A different avenue is the use of a trusted component such as IBM's cryptocard/secure coprocessor or IBM's Hardware Security Module.[10,11] Because these components are memory and storage limited, data is stored in encrypted form, shipped in small batches to the trusted component where they are decrypted and processed. Systems like Cipherbase[2,3] and TrustedDB[4] take this road. Workable homomorphic encryption would offer a simple solution, since it allows direct processing of encrypted data. Unfortunately, despite great progress, it is still far away from reality.

We proposed recently dodrant encryption that uses somewhat deterministic encryption in conjunction with semi-homomorphic encryption and logarithm and anti-logarithm tables to expand the set of numerical SQL queries that can be performed.[9] We see our goal as a stop-gap measure until the advances in cryptography lead to a practical method for homomorphic or almost homomorphic encryption.

Many cryptosystems with added properties (such as order preservation) can leak information in a way that can be leveraged by an adversary. Deterministic encryption leaks identity, which can be used in a frequency attack. If in addition the adversary can calculate certain expressions - a danger that will be hard to thwart in our system - then the adversary can go through all values in an encrypted numerical table and determine certain values. By calculation, the adversary can generate many more values (such as all integers within a certain range) and decrypt large parts of the database table column. Finally, the presence of an encrypted value as a key in a table also provides information to the adversary. Thus, for our purposes, information leakage is an even more serious problem.

Dodrant-Homomorphic Encryption

We now describe our proposal for dodrant-homomorphic encryption of numerical attributes of a database. We use Paillier's scheme, defining

(1) ϵ:Z $\rightarrow$ Z$_N$; $x \rightarrow \epsilon_r(x) = g^x r^N (\text{mod } N^2)$

with a product $N = p(1$ of two large, safe primes and $g \in \text{Z}_{N^2}$ of order a multiple of N. For normal use of Paillier's system, $r \in \text{Z}_N^*$ is a random number. We however will pick and choose r, for which reason we call it the Paillier *multiplier*. The semi-homomorphic property is

(2) $\epsilon_r(x) \cdot \epsilon_s(y) (= g^{x+y}(rs)^N (\text{mod } N^2)) = \epsilon_{rs}(x + y),$

which allows addition of encrypted values. Note that Paillier's "random" components r and s are multiplied. For subtraction, we can use

(3) $\epsilon_r(x) \cdot \epsilon_s(-y) = \epsilon_{rs}(x - y)$

or

(4) $\epsilon_r(x)/\epsilon_s(y) = g^{x-y}(rs^{-1})^N (\text{mod } N^2) = \epsilon_{r/s}(x - y)$

Paillier's cryptosystem also allows multiplying an encrypted value with an *unencrypted* constant c, namely

(5) $\epsilon_{r^c}(cx) = g^{cx} r^{Nc} (\text{mod } N^2) = \epsilon_r(x)^c.$

Databases store numerical data as decimal numbers with a fixed precision and with a fixed range or as integers in a fixed range. We convert all decimal numbers to integers by using a scale factor that is constant for the attribute. For instance, in order to o represent dollar amounts up to one million dollars, we need 100 million values, starting with $0.00 and ending with$. These values are encoded as cents, i.e. as 0, ... 99999999, using a scale factor of two. If we want to calculate 5% of a dollar amount, we encode the 5% = 0.05 as 5. If the amount is $5.99, we multiply the stored value of 599 with 5, yielding 2995, but now with a scale factor of 4. At the client, the scale factor and the integer value of the result are converted into a decimal number, giving. 2995 which is then rounded to. 30 or thirty cents. Finally, our calculations are not done over the natural numbers but are done modulo N^2, the modulo of Paillier encryption. This only poses a problem if an arithmetic operation overflows.

In order to allow multiplication, we create two tables (Scalar Function Tables, SFT), the log-table L and and the *antilog*- table E, using the real valued logarithm and exponential functions. We round logarithms and exponential values towards the nearest decimal value. We refer to our previous work for a detailed discussion of the precision needed and the resulting size of the table. This is an important topic for further work, especially since algebraic identities might be used to decrease the table sizes of currently about 50 GB. We define the *log*- table L by

(6) $L[\epsilon_1(x)] = \epsilon_1(\log(x))$

and the *antilog* or *exponential* table by

(7) $E[\epsilon_1(x)] = \epsilon_r(\exp(x)) with random r \in \mathbb{Z}_N^*.$

To multiply two encrypted values $\epsilon_1(x)$ and $\epsilon_1(y)$, we multiply the log-table entries of the encrypted values and then lookup the antilog table entry:

$$\begin{aligned} & E[L[\epsilon_1(x)] \cdot L[\epsilon_1(y)]] \\ &= E[\epsilon_1(\log(x)) \cdot \epsilon_1(\log(y)] \\ &= E[\epsilon_1(\log(x) + \log(y))] \\ &= \epsilon_r(\exp(\log(x) + \log(y))) \\ &= \epsilon_r(\exp(\log(x \cdot y))) \\ &= \epsilon_r(x \cdot y) \end{aligned}$$

We therefore *define* a multiplication between dodrant-homomorphic encrypted values by

(8) $\epsilon_1(x) * \epsilon_1(y) = E[L[\epsilon_1(x)] \cdot L[\epsilon_1(y)]]$

and obtain the identity

(9) $\epsilon_1(x) * \epsilon_1(y) = \epsilon_r(x \cdot y).$

In this calculation, additions and multiplications are those of real numbers, as we assume that the number $N = p($1 in Equation 1 is sufficiently large to allow no overflows. This calculation also generalizes to more than two factors. For example, we calculate $\epsilon_1(x) * \epsilon_1(y) * \epsilon_1(z)$ *not* as two separate $*$ -multiplications $(\epsilon_1(x) * \epsilon_1(y)) * \epsilon_1(z)$, but as

(10) $\epsilon_r(x \cdot y \cdot z) = E[L[\epsilon_1(x)] \cdot L[\epsilon_1(y)] \cdot L[\epsilon_1(z)]],$

which generalizes to any number of factors.

Similarly, we can reduce division to division of L-table values:

$$
\begin{aligned}
&E[L[\epsilon_1(x)]/L[\epsilon_1(y)]] \\
&= E[\epsilon_1(\log(x))/\epsilon_1(\log(y)] \\
&= E[\epsilon_1(\log(x) - \log(y))] \\
&= \epsilon_r(\exp(\log(x) - \log(y))) \\
&= \epsilon_r(\exp(\log(x/y))) \\
&= \epsilon_r(x/y)
\end{aligned}
$$

Again, we define

(11) $\epsilon_1(x)//\epsilon_1(y) = E[L[\epsilon_1(x)]/L[\epsilon_1(y)]]$

and obtain with this definition the functional identity

(12) $\epsilon_1(x)//\epsilon_1(y) = \epsilon_r(x/y).$

We can calculate an expression with any number of operands joined by $*$ and// operators, but once we have performed the operation, we have a Paillier-encoded value with multiplier r and are no longer capable of using it as an operand in further operations.

The basic scheme proposed in previous work[9] uses a fixed value $r = 1$.

This allows comparing the results of calculations with each other as well as other encrypted values, with causes its own set of dangers, as we discuss below.

Since we have gained the capability to calculate sums and products of sums, we can express many, but not all SQL queries involving numerical values. In particular, we cannot compare two encrypted numerical values for other than equality. It would of course be possible to encrypt each numerical value twice, once with an order preserving encryption, and once with dodrant-homomorphic encryption, as in CryptDB.[13] Unfortunately, the work of Akin and Sunar shows that this has to be done carefully in order to not allow frequency attacks.[1]

The size of the log and antilog tables is a concern. A Paillier cyphertext of a number is (at least) 16 B. For the log table to become useful, it has to contain at the very least numbers corresponding to the monetary values 0.01, 0.02, …, 100, 000.00 or 10^7 values. We have to store the key-value pairs $(\epsilon_1(x), L[\epsilon_1(x)])$. The minimum raw size of the log-table is therefore 3.2×10^8 B or 0.32 GB. Since we are storing the tables in an LH* structure, this number is not out of reach for distributed memory. The antilog tables use as keys the encrypted values obtained by arithmetic operations on natural logarithms. The keys need to have 8 digits after the decimal point and range from 0 to 16.11809565. We need about 48 GB to store the table, without using compression techniques. Since LH* tables can have a load factor exceeding 90 %, the total storage costs are around 52 GB.

Scalar Function Tables for Dodrant-Homomorphic Encryption

Paillier's cryptosystem encrypts integers whereas many database tables contain decimal numbers with a fixed precision. Recall that we use a scale factor in order to convert decimal fixed precision numbers to integers. Multiplication and division of two attribute values will respectively add and subtract the scale factors for the two attributes.

The log-table is a scalar function table. It is organized as a dictionary that associates the dictionary key $\epsilon_1(x)$ with the value $\epsilon_1(\log(x))$. The keys range from $\epsilon_1(2)$ to $\epsilon_1(R)$, where R is the range chosen. Of course, $\epsilon_1(\log(1))$ is $\epsilon_1(0) = 1$. The Log-table needs to be protected against direct access, as it would otherwise leak information by adding up values and see whether their sum is a key. If $X = \epsilon_1(x)$ and if $X \cdot X = \epsilon_1(x + x)$ is not a key, then $x > R/2$. Passing through the keys, just addition with itself reveals the set $M(R/2) = \{\epsilon_1(x) | x \in \{R/2 + 1, \dots, R\}\}$. We repeat this step testing whether the double of an encrypted value in $M(R/2)$ is in $M(R/2)$ to obtain the set $MM(R/4) = \{\epsilon_1(x) | x \in \{R/4 + 1, \dots, R\}\}$. This procedure finally yields the encrypted value of 2, namely $\epsilon_1(2)$, as the dictionary key that can be added to itself the most times and $\epsilon_1(3)$ as the second-best value, and so on. This vulnerability is general, and is caused by using the same Paillier multiplier, i.e. by deterministic encryption.

Strict access control for the SFTs might be operationally difficult. Instead, we can add somewhat randomly the encryption of sums to the Log-dictionary. Details are left to future work.

The values of the log-table are the expressions $\epsilon_1(\log(x))$. The logarithm of an integer value is of course usually not an integer. Paillier's system however only encrypts integers. We calculate the natural logarithm with a precision of eight digits after the decimal points. This corresponds to using a scale factor of 4. We could of course choose another base then base 10 for number representation and/or another base for the logarithm, but it turns out that for our range R, these two choices are quite reasonable.[9] Different choices would result in different table sizes.

The exp-table needs to be much larger, since the keys need to contain all possible products

$$L(\epsilon_1(x)) \cdot L(\epsilon_1(y)) = \epsilon_1(\log(x) + \log(y)) = \epsilon_1(\log(x \cdot y))$$

with $x, y \in \{2, \dots, R\}$. It has therefore at least R^2 keys. This however neglects that we accrue a certain rounding error in the calculation of the logarithm. Even though the addition of logarithm values is done in the integer domain and is therefore completely accurate, we are *de facto* adding up rounded values and so have to expect occasional rounding errors.

The values of the dictionary constituted by the exp-table are rounded to the same scale factor as before and then encrypted.

We organize both tables as a scalable distributed data structure LH*. Such a hash-structure can have a load factor that exceeds 90%. We calculate that both tables would take up less than 100 GB of storage. In an age where even small laptops now come routinely with more than 4GB storage, the size of the SFT is no hindrance to implementation. Experiential work more than a decade ago has shown that access through a distributed hash structure takes less than a milli-second for record look-up.

Dangers of Dodrant-Homomorphic Encryption

Since we are using look-up tables, we have to use some type of deterministic encryption, doing away with one of the major advantages of Paillier's crypto-scheme. The main problem with deterministic encryption is the possibility of attacks based on frequencies. The number of times that a certain encrypted value is taken might be a statistical outlier and if this is the case, then this fact can be used to determine the unencrypted value. For example, if the most frequent price in the dollar store is 99 cents, then we just look for the most frequent encrypted value in the price attribute in order to find the

encryption of the 99 cents value. Reversely, if a value itself is an outlier and we use order-preserving encryption, then we can also determine its encryption from the database.

Often, knowing a few encryptions of peculiar values is in itself not very dangerous. (Of course, a database administrator might be tempted to replace her salary with the CEO's salary, etc.) However, in conjunction with the capability to calculate with encrypted data and to compare encrypted values, determinism becomes quite a bit, or should we say, even more dangerous. Adversaries can expand their knowledge of plaintext - cipher pairs or they can use algebraic identities to find these pairs.

Assume that we use Paillier multiplier $r = 1$ in Equation 7. This gives an adversary the capability to recognize the encrypted value of zero since $\epsilon_1(x) = g^0 r^N = 1$. This might be more of an annoyance than an exploitable vulnerability, but if the adversary can multiply encrypted value (for example, because the adversary has gained the privileges of the administrator), then the adversary can find the encrypted value of 1 because it and 0 are the only solutions to $x \cdot x = x$. Similarly, 2 and 0 are the only solutions to $x \cdot x = x + x$, and in general n and 0 to $x \cdot x = x + x + \cdots + x$ with n addends on the right of the equation. Of course, once an adversary has successfully decrypted 0 and 1, the adversary can calculate the encryption of 2, 3,…, creating a large, but manageable lookup table for decoding. While presumably records do not consist only of numerical values, it is easy to imagine that the capability of decrypting numerical values alone can be leveraged into a more extensive penetration of the database. A simple operational countermeasure is to only use dodrant-homomorphic encryption on numerical values that need to be added and multiplied.

Paillier's cryptosystem allows multiplication with constants, so that we obtain yet another class of algebraic identities for encrypted values, such as $2x = x \cdot x$ which again would reveal $x = 2$. The traditional use of Paillier's cryptosystem is completely safe, since one value can be encrypted in a multitude of forms so that evaluating identity is impossible.

In short, besides attacks based on statistical frequency, an adversary can use algebraic identities to determine the encryption of certain values and then leverage this to build a dictionary of values.

Thwarting Attacks Using Algebraic Identities

An adversary can use algebraic identities if the adversary can calculate with numerical values and compare the results. The latter can become possible because we need to use deterministic encryption in order to allow the use of tables. However, in our improved scheme, products are encrypted with a Paillier multiplier r different from 1. It is therefore impossible to evaluate algebraic identities with a product on one side and a non-product on the other side.

In order to discuss the implications of picking different Paillier multipliers for products, we need to make a distinction in the use of numerical attributes in a database table. First, we have numerical attributes that are not subject to algebraic manipulations. An example would be social security numbers in the US or identity card numbers in Latin America. There is no need to use dodrant homomorphic encryption on these. Since they are not subject to frequency attacks (there is one number per individual), but useful for join operations, these values can be encrypted with any deterministic encryption such as AES. A second type of numerical attribute sees its value subjected only to addition and multiplication by constants. This type of attribute can be encrypted with Paillier's scheme with random multipliers. The third type of attributes allows addition and multiplication of its attributes. Another difference is whether

we allow numerical values to be updated as a result of a calculation. We cannot see how this would be necessary for a numerical attribute of the first kind. Updating a value for an attribute of the second kind is also possible, since at worst the Paillier multiplier is determined by the antilogarithm table Fi. An example for a numerical attribute of the third kind would be prices of items. To determine the total costs or shipping costs etc. the cloud service would multiply the price of an item with the total number of the item ordered, then add sales or value-added taxes and maybe insurance costs as a percentage of the total order value.

We discuss two different scenarios. First, we can assign random Paillier multiplier in Equation 7. After any multiplication on encrypted numerical values, the products can no longer be compared with any stored encrypted numerical values and they cannot be factors in further multiplications. Products can however be added. We can update numerical attributes of the second kind without problems with the result of numerical calculations on attributes of the second and third kind. This is not the case when updating values of the third kind since these need to be encoded with a Paillier multiplier of $r = 1$. The owner's system or a trusted service would need to decipher each calculated value, encode it with Paillier multiplier $r = 1$ and then insert it.

The second possibility is to use a fixed, but hidden constant Paillier multiplier in Equation 7. In this case, a query can calculate a value $\epsilon_r(x)$ and multiply it with $\epsilon_{r^{-1}}(1)$ in order to obtain $\epsilon_{r.r^{-1}}(1 \cdot x) = \epsilon_1(x)$ and insert this into the table. Unfortunately, $\epsilon_{r^{-1}}(1)$ leaks $\epsilon_r(1)$, which in turn leaks $\epsilon_r(n)$ for any $n \in \mathbb{N}$ and allows therefore comparisons of products with integers and therefore the exploitation of algebraic identities. Other methods such as adding an encrypted value with zero and Paillier multiplier r^{-1} also do not work for the same reason.

We conclude that to our best knowledge, it is impossible to have the cloud service automatically insert products into attribute values of the third kind. For example, we will not be able to process automatically an item-price table with an over-the-board price increase of 3%.

We discuss as an example the calculation of the mean and the variation of a sequence of dodrant-homomorphic encrypted numerical values $\epsilon_1(x_1), \epsilon_1(x_2) \ldots \epsilon_1(x_n)$. In order to calculate the mean, we need to count the number of elements in the selection. Now, calculating the count as $\epsilon_1(n)$ would be a dangerous procedure since this would allow an adversary who intercepts the query and the accesses to tables to obtain plain text - cypher text values. This is however not necessary as we can treat the rounded real value of $1/n$ as a constant and multiply using Equation 5. Recall that $1/n$ is encoded as an integer modulo N^2. We now obtain the encrypted mean $\mu = \frac{1}{n}\sum_{i=1}^{n} x_2$ by

$$\begin{aligned}\epsilon_1(\mu) &= \epsilon_1\left(\frac{1}{n}\sum_{i=1}^{n} x_i\right)\\ &= \left(\sum_{i=1}^{n} \epsilon_1(x_i)\right)^{1/n}\\ &= \left(\prod_{i=1}^{n} \epsilon_1(x_i)\right)^{1/n}\end{aligned}$$

Here, we first calculate $1/n(\mathrm{mod}\ N)$ and then use the result as an exponent modulo N^2. The formula of Koenig-Huygens calculates the variance σ^2 as the difference between the average of the squares and the square of averages. Its biggest drawback is the possibility for overflow (a fact very relevant for us as we need to keep table sizes small and therefore need to limit the range) and the possibility of accumulating rounding errors. However, its implementation is simple. The average sum of squares is obtained by

$$\begin{aligned}\epsilon_{r^n}(\sum_{i=1}^{n} x_i^2/n) &= (\epsilon_{r^n}(\sum_{i=1}^{n} x_i^2))^{1/n} \\ &= (\prod_{i=1}^{n} \epsilon_r(x_i^2))^{1/n} \\ &= (\prod_{i=1}^{n} \epsilon_1(x_i) * \epsilon_1(x_i))^{1/n}\end{aligned}$$

The square of the mean is obtained via

$$\epsilon_r(\mu^2) = \epsilon_1(\sum_{i=1}^{n} x_i/n) * \epsilon_1(\sum_{i=1}^{n} x_i/n)$$

$$= (\prod_{=1}^{n} \epsilon(x_i))^{1/n} * (\mathrm{I}^n 1\epsilon(x_i))^{1/n} ii = 1$$

Both parts are combined by

$$\epsilon_{r^{n-1}}(\sigma^2)$$

$$= \epsilon_{r^{n-1}}(\sum_{i=1}^{n} \frac{x_i^2}{n} - n\,(\sum_{i=1}^{n} \frac{x_i}{n})^2)$$

$$= \epsilon_{r^n}(\sum_{i=1}^{n} \frac{x_i^2}{n})/((n\prod_{i=1}^{n} \epsilon(x_i))^{\frac{1}{n}} * (\prod_{i=1}^{n} \epsilon(x_i))^{\frac{1}{n}})$$

$$\epsilon_1(\mu) = (\prod_{?=1}^{n} \epsilon_1(x_i))^{\frac{1}{n}}$$

In order to avoid overflows, we can employ a two-pass solution, where we first calculate the mean μ as

$$\epsilon_1(\mu) = (\prod_{i=1}^{n} \epsilon_1(x_i))^{1/n}$$

and then

$$\epsilon_{r^n}(\sum_{i=1}^{n}(x-\mu)^2/n)$$
$$= (\epsilon_{r^n}(\sum_{i=1}^{n}(x-\mu)^2))^{1/n}$$
$$= (\prod_{i=1}^{n}\epsilon_r((x_i-\mu)^2))^{1/n}$$
$$= (\prod_{i=1}^{n}\epsilon_1(x_i-\mu) * \epsilon_1(x_x-\mu))^{1/n}$$
$$= (\prod_{i=1}^{n}(\epsilon_1(x_i)/\epsilon_1(\mu)) * (\epsilon_1(x_i)/\epsilon_1(\mu)))^{1/n}$$

We can even use Welford's one pass incremental method.[15] It is based on maintaining a partial mean

$$\mu_i = \sum_{v=1}^{i} x_v/n$$

and partial variance

$$\sigma_i^2 = (1/i)\sum_{v=1}^{i}(x_{\text{l}/} - \mu_v)^2$$

and uses the update formula

$$\sigma_i^2 = \sigma_{i-1}^2 + \frac{i-1}{i}(x_i - \mu_{\text{i}-1})^2$$

While it is straightforward to implement this formula using constant multiplication and the *-operator on encrypted values, the frequent multiplication with constants in the plaintext domain is less attractive as these translates into exponentiations in the cipher domain and are more involved.

Thwarting Frequency Attacks

Frequency based attacks use known or guessed facts about frequency outliers (such as that the most common price in the dollar store is 0.99 or that there is only one admitted student of age 14, recently

profiled in the student newspaper). There are quite effective when using statistics on more than one attribute.

The advantage of non-deterministic encryption lies precisely in the built-in resilience against frequency attacks as every encrypted value appearing in a database is unique (with overwhelming probability). In Paillier's scheme, there are basically as many encryptions of a single value ($N-1$ different Paillier multipliers to be precise) as there are values that can be encrypted (N to be precise). We propose to encode each value x as a pair

$$(\epsilon_1(\rho x), \epsilon_1(\rho))$$

where ρ is a small random integer. This is reminiscent of the mathematical construction of an Abelian group out of a cancellative commutative monoid, for example in constructing the entire numbers from natural numbers. The user can use a division to recover $\epsilon_r(x)$ as the division of both parts. To add two encrypted numbers $(\epsilon_1(\rho_1 x_1), \epsilon_1(\rho_1))$ and $(\epsilon_1(\rho_2 x_2), \epsilon_2(\rho_2))$, we essentially add two fractions as

$$\frac{p_1 x_1}{p_1} + \frac{p_2 x_2}{p_1} = \frac{p_1 p_2 x_1 + p1p2_2^x}{p_1 p_2}$$

i.e. by calculating

$$\begin{aligned}&(\epsilon_1(\rho_1 x_1) * \epsilon_1(\rho_2) + \epsilon_1(\rho_1) * \epsilon_1(\rho_2 x_2), \epsilon_1(\rho_1) * \epsilon_1(\rho_2))\\ &= (\epsilon_r(\rho_1\rho_2 x_1) + \epsilon_r(\rho_1\rho_2 x_2), \epsilon_r(\rho_1\rho_2))\\ &= (\epsilon_{r^2}(\rho_1\rho_2 x_1 + \rho_1\rho_9 x_2), \epsilon_r(\rho_1\rho_9)\end{aligned}$$

whereas the product of the two encrypted numbers is simply

$$\begin{aligned}&(\epsilon_1(\rho_1 x_1) * \epsilon_1(\rho_9 x_2), \epsilon_1(\rho_1) * \epsilon_1(\rho_9))\\ &= (\epsilon_r(\rho_1\rho_2 x_1 x_2), \epsilon_r(\rho_1\rho_2)).\end{aligned}$$

These operations generalize immediately to arbitrary numbers of operands.

Frequency based attempts at deciphering are now impossible, and we are still able to calculate sums of products.

Conclusion

In 2015, Jajodia, Litwin, and Schwarz proposed a stop-gap solutions for homomorphic encryption[9] that used a deterministic variant of Paillier's cryptoscheme and large tables. We identified two weaknesses that render that scheme insecure and propose a variant that is not subject to these vulnerabilities.

Future work will also have to address the size of the E and L-tables, which we hope to reduce using algebraic means. Before floating point coprocessors, 8 bit and 16 bit processors were able to emulate all floating point operations. The same software emulations are applicable to elevate the capability to calculate with limited range integers (using small tables) to the capability to process floating point numbers. Whether and how this can be done in a safe manner remains to be seen.

Future work will need to provide a more thorough security assessment, though we are doubtful that at the current state of the art in Cryptography a formal proof of security can be attempted. The controversy surrounding CryptDB and its security[1–2,3,4,5,6,7,8,9,10,11,12,13] highlights the difficulty in this area.

Finally, there is justified hope that something amounting to almost complete homomorphic encryption will eventually become possible. In this case, our stop-gap proposal has lost its *raison d'être*. In the remaining years (or decades), our proposal seems to be interesting, at least for databases with some guaranteed privacy but not containing data classified as "Secret", meaning that its disclosure could bring death or bodily harm.

References

1. I. H. Akin, B. Sunar, "On the difficulty of securing web applications using CryptDB", *IEEE Fourth International Conference on Big Data and Cloud Computing (BDCloud)*, pp. 745-752, 2014.

2. A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, R. Venkatesan, "Orthogonal security with cipherbase", *Proceedings of the Sixth Biennial Conference on Innovative Data Systems Research CIDR*, 2013.

3. A. Arasu, R. Kaushik, "Oblivious query processing", *arXiv preprint arXiv:1312.4012*, 2013.

4. S. Bajaj, R. Sion, "TrustedDB: A trusted hardware-based database with privacy and data confidentiality", *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 3, pp. 752-765, 2014.

5. C. Curino, E. Jones, Y. Zhang, E. Wu, S. Madden, "Relational cloud: The case for a database service", *New England Database Summit*, pp. 1-6, 2010.

6. C. Curino, E. P. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, N. Zeldovich, "Relational cloud: A database-as-a-service for the cloud", *5th Biennial Conference on Innovative Data Systems Research CIDR*, 2011.

7. H. Hacigümüş, B. Iyer, C. Li, S. Mehrotra, "Executing SQL over encrypted data in the database-service-provider model", *Proceedings of the 2002 ACM SIGMOD International conference on Management of data*, pp. 216-227, 2002.

8. B. Hore, S. Mehrotra, H. Hacigümüc, "Managing and querying encrypted data" in Handbook of Database Security, Springer, pp. 163-190, 2008.

9. S. Jajodia, W. Litwin, T. Schwarz, "Numerical SQL value expressions over encrypted cloud databases" in Database and Expert Systems Applications, Springer, pp. 455-478, 2015.

10. "On the fly AES256 decryption/encryption for trusted cloud SQL DBS (position statement)", *Proceedings of the 3rd International Workshop on Privacy and Security of Big Data PSBD*, 2016.

11. "Trusted cloud SQL DBS with on-the-fly AES decryption/encryption", *Proceedings of the First International Workshop on Big Data Management in Cloud Systems BDMICS*, 2016.

12. M. Naveed, S. Kamara, C. V. Wright, "Inference attacks on property-preserving encrypted databases", *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 644-655, 2015.

13. R. A. Popa, C. M. S. Redfield, N. Zeldovich, H. Balakrishnan, "CryptDB: processing queries on an encrypted database", *Communications of the ACM*, vol. 55, no. 9, pp. 103-111, Sep. 2012.

14. S. Tu, M. F. Kaashoek, S. Madden, N. Zeldovich, "Processing analytical queries over encrypted data", *Proceedings of the 39th International Conference on Very Large Data Bases*, pp. 289-300, 2013.

15. B. P. Welford, "Note on a method for calculating corrected sums of squares and products", *Technometrics*, vol. 4, no. 3, pp. 419-420, 1962.