A Specific Encryption Solution for Data Warehouses

Ricardo Jorge Santos[1], Deolinda Rasteiro[2], Jorge Bernardino[3] and Marco Vieira[4]

[1, 4]CISUC – FCTUC – University of Coimbra – 3030-290 Coimbra – Portugal
[2]DFM – ISEC – Polytechnic Institute of Coimbra – 3030-190 Coimbra – Portugal
[3]CISUC – ISEC – Polytechnic Institute of Coimbra – 3030-190 Coimbra – Portugal

lionsoftware.ricardo@gmail.com, dml@isec.pt, jorge@isec.pt, mvieira@dei.uc.pt

Abstract. Protecting Data Warehouses (DWs) is critical, because they store the secrets of the business. Although published work state encryption is the best way to assure the confidentiality of sensitive data and maintain high performance, this adds overheads that jeopardize their feasibility in DWs. In this paper, we propose a Specific Encryption Solution tailored for DWs (SES-DW), using a numerical cipher with variable mixes of eXclusive Or (XOR) and modulo operators. Storage overhead is avoided by preserving each encrypted column's datatype, while transparent SQL rewriting is used to avoid I/O and network bandwidth bottlenecks by discarding data roundtrips for encryption and decryption purposes. The experimental evaluation using the TPC-H benchmark and a real-world sales DW with Oracle 11g and Microsoft SQL Server 2008 shows that SES-DW achieves better response time in both inserting and querying, than standard and state-of-the-art encryption algorithms such as AES, 3DES, OPES and Salsa20, while providing considerable security strength.

Keywords: Encryption, Confidentiality, Security, Data Warehousing.

1 Introduction

Data Warehouses (DWs) store extremely sensitive business information. Unauthorized disclosure is therefore, a critical security issue. Although encryption is used to avoid this, it also introduce very high performance overheads, as shown in [16]. Since decision support queries usually access huge amounts of data and substantial response time (usually from minutes to hours) [12], the overhead introduced by using encryption may be unfeasible for DW environments if they are too slow to be considered acceptable in practice [13]. Thus, encryption solutions built for DWs must balance security and performance tradeoff requirements, *i.e.*, they must ensure strong security while keeping database performance acceptable [13, 16].

As the number and complexity of "data-mix" encryption rounds increase, their security strength often improves while performance degrades, and vice-versa. Balancing performance with security in real-world DW scenarios is a complex issue which depends on the requirements and context of each particular environment. Most encryption algorithms are not suitable for DWs, because they have been designed as a gener-

al-purpose “one fits all” security solution, introducing a need for specific solutions for DWs capable of producing better security-performance tradeoffs.

Encryption in DBMS can be column-based or tablespace-based. Using tablespace encryption implies losing the ability to directly query data that we do not want or need to encrypt, adding superfluous decryption overheads. Best practice guides such as [14] recommend using column-based encryption for protecting DWs. Thus, we propose a column-based encryption solution and for fairness we compare it with other similar solutions.

In this paper, we propose a lightweight encryption solution for numerical values using only standard SQL operators such as eXclusive OR (XOR) and modulo (MOD, which returns the remainder of a division expression), together with additions and subtractions. We wish to make clear that it is not our aim to propose a solution as strong in security as the state-of-the-art encryption algorithms, but rather a technique that provides a considerable level of overall security strength while introducing small performance overheads, *i.e.*, that presents better security-performance balancing. To evaluate our proposal, we include a security analysis of the cipher and experiments with standard and state-of-the art encryption algorithms such as Order-Preserving Encryption (OPES) [3] and Salsa20 (alias Snuffle) [5, 6], using two leading DBMS.

In summary, our approach has the following main contributions and achievements:

- SES-DW avoids storage space and computational overhead by preserving each encrypted column’s original datatype;
- Each column may have its own security strength by defining the number of encryption rounds to execute. This also defines how many encryption keys are used, since each round uses a distinct key (thus, the true key length is the number of rounds multiplied by the length of each round’s encryption key). This enables columns which store less sensitive information to be protected with smaller-sized keys and rounds and thus, process faster than more sensitive columns;
- Our solution is used transparently in a similar fashion as the Oracle TDE [11, 14] and requires minimal changes to the existing data structures (just the addition of a new column), and the SES-DW cipher uses only standard SQL operators, which makes it directly executable in any DBMS. This makes our solution portable, low-cost and straightforward to implement and use in any DW;
- Contrarily to solutions that pre-fetch data, by simply rewriting queries we avoid I/O and network bandwidth congestion due to data roundtrips between the database and encryption/decryption mechanism, and consequent response time overhead;
- The experiments show that our technique introduces notably smaller storage space, response and CPU time overheads than other standard and state of the art solutions, for nearly all queries in all tested scenarios, in both inserting and querying data.

The remainder of the paper is organized as follows. In section 2 we present the guidelines and describe our proposal. In Section 3, we discuss its security issues. Section 4 presents experimental evaluations using the TPC-H decision support benchmark and a real-world DW with Oracle 11g and Microsoft SQL Server 2008. Section 5 presents related work and finally, section 6 presents our conclusions and future work.

2 SES-DW: Specific Encryption Solution for Data Warehouses

In this section we point out a set of considerations concerning the use of encryption solutions in DW environments, which guide the requirements that serve as the foundations of our proposal, and then we describe our approach and how it is applied.

2.1 The foundations of SES-DW

Standard encryption algorithms were conceived for encrypting general-purpose data such as blocks of text, *i.e.*, sets of binary character-values. Standard ciphers (as well as their implementations in the leading DBMS) output text values, while DW data is mostly composed by numerical datatype columns [12]. Most DBMS provide built-in AES and 3DES encryption algorithms and enable their transparent use. However, they require changing each encrypted column's datatype at the core to store the ciphered outputs. To use the encrypted values for querying once decrypted, the textual values must be converted back into numerical format in order to apply arithmetic operations such as sums, averages, etc., adding computational overheads with considerable performance impact. Since working with text values is much more computationally expensive than working with numeric values, standard ciphers are much slower than solutions specifically designed for numerical encryption such as ours, which is specifically designed for numerical values and avoids datatype conversion overheads.

Data in DWs is mostly stored in numerical attributes that usually represent more than 90% of the total storage space [12]. Numerical datatype sizes usually range from 1 to 8 bytes, while standard encryption outputs have lengths of 8 to 32 bytes. Since DWs have a huge amount of rows that typically take up many gigabytes or terabytes of space, even a small increase of any column size required by changing numeric datatypes to textual or binary in order to store encryption outputs introduces very large storage space overheads. This consequently increases the amount of data to process, as well as the required resources, which also degrades database performance. While encrypting text values is mainly not so important for DWs, efficiently encrypting numerical values is critical. In our approach, we preserve the original datatype and length of each encrypted column, to maintain data storage space.

Topologies involving middleware solutions such as [15] typically request all the encrypted data from the database and execute decrypting actions themselves locally. This strangles the database server and/or network with communication costs due to bandwidth consumption and I/O bottlenecks given the data roundtrips between middleware and database, jeopardizing throughput and consequently, response time. Given the typically large amount of data accessed for processing DW queries, previously acquiring all the data from the database for encrypting/decrypting at the middleware is impracticable. Therefore, our approach is motivated by the requirement of using only operators supported by native SQL. This enables using only query rewriting for encrypting and decrypting actions and no external languages or resources need to be instantiated, avoiding data roundtrips and thus, avoiding I/O and network overhead from the critical path when compared to similar middleware solutions.

In what concerns the design of “data mixing” for each of the cipher’s rounds, we discard bit shifting and permutations, commonly used by most ciphers, since there is no standard SQL support for these actions. We also discard the use of substitution boxes (*e.g.* AES uses several 1024-byte S-boxes, each of which converts 8-bit inputs to 32-bit outputs). Although complex operations such as the use of S-boxes provide a large amount of data mixing at reasonable speed on several CPUs, thus achieving stronger security strength faster than simple operations, the potential speedup is fairly small and is accompanied by huge slowdowns on other CPUs. It is not obvious that a series of S-box lookups (even with large S-boxes, as in AES, increasing L1 cache pressure on large CPUs and forcing different implementation techniques on small CPUs) is faster than a comparably complex series of integer operations. In contrast, simple operations such as bit additions and XORs are consistently fast, independently from the CPU. Our approach aims to be DBMS platform independent, making it usable in any DW without depending on any programming language or external resource, as well as specific CPU models. Given the requirements described in the former paragraphs, the proposed solution is described in the next subsections.

2.2 The SES-DW Cipher

Considering x the plaintext value to cipher and y the encrypted ciphertext, NR the number of rounds, $RowK$ a 2^{128} bit encryption key, $Operation[t]$ a random binary vector (*i.e.*, each element is 1 or 0), $XorK[t]$ and $ModK[t]$ as vectors where each element is an encryption subkey with the same bit length as the plaintext x, and $F(t)$ a MOD/XOR mix function (explained further), where t represents each individual encryption round number (*i.e.*, $t = 1...NR$). Figures 1.a and 1.b show the external view of the SES-DW cipher for respectively encrypting and decrypting.

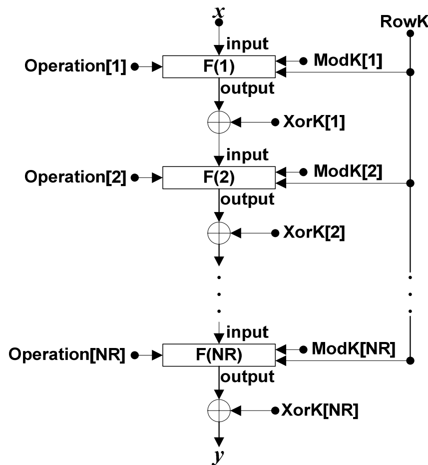

Fig. 1.a The SES-DW encryption cipher

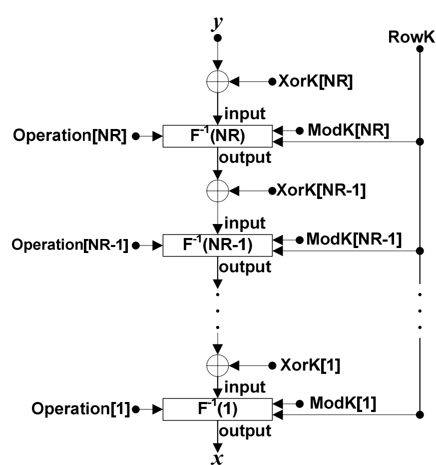

Fig. 2.b The SES-DW decryption cipher

As illustrated, we randomly mix MOD with XOR throughout the encryption rounds, given a random distribution of 1 and 0 values of vector *Operation*. In the

rounds where *Operation*[*t*] = 0, only XOR is used with the respective *XorK*[*t*]; in rounds where *Operation*[*t*] = 1, we first perform MOD with addition and subtraction using the respective *ModK*[*t*] and *RowK*[*j*], and *TabK*, and afterwards XOR with the respective *XorK*[*t*]. To avoid generating a ciphertext that may overflow the bit length of *x* it must be assured that the bit length of the term using MOD (*EncryptOutput* + (*RowK*[*j*] MOD *ModK*[*t*]) - *ModK*[*t*]) is smaller or equal to the bit length of *x*.

As an example of encryption, consider the encryption of an 8 bit numerical value (*x* = 126) executing 4 rounds (*NR* = 4), given the following assumptions:

```
Operation = [0, 1, 0, 1]     XorK = [31, 2, 28, 112]
```

For t=1 (round 1), `EncryptOutput = 126 XOR 31 = 97`
For t=2 (round 2), `EncryptOutput = (97+(15467801 MOD 36)-36) XOR 2 = 64`
For t=3 (round 3), `EncryptOutput = 64 XOR 28 = 92`
For t=4 (round 4), `EncryptOutput = ((92+15467801 MOD 19)-19) XOR 112 = 40`

Thus, Encrypt(126, 4) = 40. In the decryption cipher, shown in Figure 1.b, $F^{-1}(t)$ also represents the reverse MOD/XOR mix function for decryption. Given this, the SES-DW cipher decryption function for decrypting *x* with *NR* rounds is:

```
FUNCTION Decrypt(x,NR)
   DecryptOutput = x
   FOR t = NR DOWNTO 1 STEP -1
      DecryptOutput = DecryptOutput XOR XorK[t]
      IF Operation[t] = 1 THEN
         DecryptOutput = DecryptOutput - (RowK MOD ModK[t]) + ModK[t]
      END_IF
   END_FOR
RETURN DecryptOutput
```

Considering the encryption example previously shown, we now demonstrate the decryption process for y = 40, given the same *Operation*, *RowK*, *XorK* and *ModK*:

For t=4 (round 1), `DecryptOutput = (40 XOR 112)-(15467801 MOD 19)+19 = 92`
For t=3 (round 2), `DecryptOutput = 92 XOR 28 = 64`
For t=2 (round 3), `DecryptOutput = (64 XOR 2)-(15467801 MOD 36)+36 = 97`
For t=1 (round 4), `DecryptOutput = 97 XOR 31 = 126`

Thus, Decrypt(40, 4) = 126, which is the original x plaintext value. Although our cipher only works with numerical values, we maintain the designation of plaintext and ciphertext respectively for the true original input value and ciphered value.

2.3 The SES-DW Functional Architecture

The system's architecture is shown in Figure 2, made up by three entities: 1) the encrypted database and its DBMS; 2) the SES-DW security middleware application; and 3) user/client applications to query the encrypted database. The SES-DW middleware is a broker between the DBMS and the user applications, using the SES-DW encryption and decryption methods and ensuring queried data is securely processed and the proper results are returned to those applications. We assume the DBMS is a trusted server and all communications are made through SSL/TLS secure connections, to protect SQL instructions and returned results between the entities.

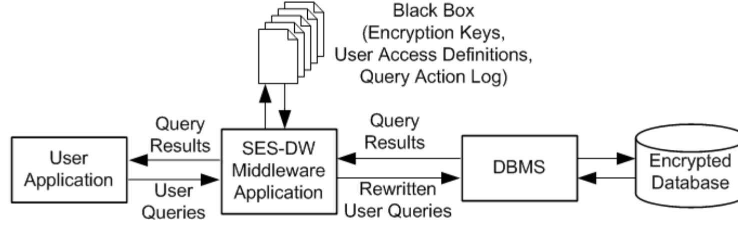

Fig. 2. The SES-DW Data Security Architecture

The Black Box is stored on the database server, created for each encrypted database. This process is similar to an Oracle Wallet, which keeps all encryption keys and definitions for each Oracle Database [14]. However, contrarily to Oracle, where a DBA has free access to the wallet, in our solution only the SES-DW middleware can access the Black Box, *i.e.*, absolutely no user has direct access to its content. In the Black Box, the middleware will store all encryption keys and predefined data access policies for the database. The middleware will also create a history log for saving duplicates of all instructions executed in the database, for auditing and control purposes. All Black Box contents are encrypted using AES with a 256 bit key.

To obtain true results, user actions must go through the security middleware application. Each time a user requests any action, the application will receive and parse the instructions, fetch the encryption keys, rewrite the query, send it to be processed by the DBMS and retrieve the results, and finally send those results back to the application that issued the request. Thus, SES-DW is transparently used, since query rewriting is transparently managed by the middleware. The only change user applications need is to send the query to the middleware, instead of querying the database directly.

To encrypt a database, a DBA requires it through the SES-DW middleware. Entering login and database connection information, the middleware will try to connect to that database. If it succeeds, it creates the Black Box for that database, as explained earlier. Afterwards, the middleware will ask the DBA which tables and columns to encrypt. All the required encryption keys (*RowK*, *XorK*, *ModK*) for each table and column will be generated, encrypted by an AES256 algorithm and stored in the Black Box. Finally, the middleware will encrypt all values in each column marked for encryption. Subsequent database updates must always be done through the middeware, which will apply the cipher to the values and store them directly in the database.

To implement SES-DW encryption in a given table *T*, consider the following: Suppose table *T* with a set of *N* numerical columns $C_i = \{C_1, C_2, \ldots, C_N\}$ to encrypt and a total set of *M* rows $R_j = \{R_1, R_2, \ldots, R_M\}$. Each value to encrypt in the table will be identified as a pair (R_j, C_i), where Rj and Ci respectively represent the row and column to which the value refers ($j = \{1..M\}$ and $i = \{1..N\}$). To use the SES-DW cipher, we generate the following encryption keys and requirements:

- An encryption key *TabK*, a 128 bit random generated value, constant for table *T*;
- Vector $RowK[j]$, with $j = \{1..M\}$, for each row *j* in table T. Each element holds a random 128 bit value;
- Define NR_i with $i = \{1..N\}$, which gives the number of encryption rounds to execute for each column C_i. We define $NR_i = SBL_i/\text{BitLength}(C_i)$, where SBL_i is the

desired security bit strength for the *XorK* and *ModK* encryption keys of column C_i and BitLength(C_i) is the datatype bit length of column C_i (*e.g.* if we want to secure a 16 bit column C_i with a security strength of 256 bits, then the number of encryption rounds would be 256/16 = 16);

- Vectors $XorK_i[t]$ and $ModK_i[t]$, with $t = \{1..NR_i\}$, for each C_i, filled with randomly generated unique values. The bit length of each key is equal to the bit length of each C_i's datatype;
- A vector $Operation_i[t]$, with $t = \{1..NR_i\}$, for each column C_i, filled randomly with 1 and 0 values, so that the count of elements equal to 1 is the same as the count of elements equal to 0 (e.g. $Operation_i = [0,1,0,0,1,1,0,1]$, with $NR_i = 8$).

Since the number of rows in a DW fact table is often very big, the need to store a *RowK*[*j*] encryption key for each row *j* poses a challenge. If these values were stored in a lookup table separate from table *T*, a heavy join operation between those tables would be required to decrypt data. Given the typically huge number of rows in fact tables, this must be avoided. For the same reasons, storing *RowK*[*j*] in RAM is also impracticable. To avoid table joins, as well as oversized memory consumption, the values of *RowK*[*j*] must be stored along with each row *j* in table *T*, as an extra column C_{N+1}. This is the only change needed in the DW data structure in order to use SES-DW. To secure the value of *RowK*[*j*], it should be XORed with key *TabK* before being stored. To retrieve the true value of *RowK*[*j*] in order to use the SES-DW algorithms, we need to simply calculate (Rj, C_{N+1}) XOR *TabK*.

3 Security Issues

Threat model. All user instructions are managed by the SES-DW middleware, which transparently rewrites them to query the DBMS and retrieve the results. The users never see the rewritten instructions. For security purposes, the middleware shuts off database historical logs on the DBMS before requesting execution of the rewritten instructions, so they are not stored in the DBMS, since this would disclose the encryption keys. All communications between user applications, the SES-DW middleware and the DBMS are done through encrypted SSL/TLS connections. In what concerns the Black Box, all content is encrypted using the AES 256 algorithm, making it as secure in this aspect as any other similar solution for stored data (*e.g.* Oracle 11g TDE and SQL Server 2008 TDE). The only access to the Black Box content is done by the middleware, which is managed only by the application itself. We assume the DBMS is an untrusted server such as in the Database-As-A-Service (DAAS) model and the "adversary" is someone that manages to bypass network and SES-DW access controls, gaining direct access to the database. We also assume the SES-DW algorithms are public, so the attacker can replicate the encryption and decryption functions, meaning that the goal of the attacker is to obtain the keys in order to break security.

Using variable key lengths and MOD-XOR mixes. The bit length of the encryption keys *XorK* and *ModK* are the same as the bit length of each encrypted column, meaning that an 8 bit sized column datatype will have 8 bit sized encryption keys. It is obvious that using 8 bit keys on their one is not secure at all. However, since all keys

are distinct in each round, executing 16 rounds would be roughly equivalent to having a 16*8 = 128 bit key in the encryption process. It is up to the DW security administrator to decide how strongly secure each column should be, which defines how many rounds should be executed, considering the bit length of the column's datatype.

The MOD operator is used in the cipher because it is non-injective, given that for X MOD $Y = Z$, the same output Z, considering Y a constant, can have an undetermined number of possibilities in X as an input that will generate the same value Z (*e.g.* 15 MOD 4=3, 19 MOD 4=3, 23 MOD 4=3, etc). Since MOD operations are non-injective, the encryption rounds using MOD are also non-injective. Given that injectivity is a required property for invertibility, our cipher is thus not directly invertible. It is also true that the same ciphered output values are most likely to come from different original input values. Moreover, randomly using the XOR and MOD operators as the two possible operators for each round also increases the number of possibilities an attacker needs to test in exhaustive searches for the output values of each encryption round, since the attacker does not know the rounds in which MOD is used with XOR and needs to test both hypothesis (XOR and MOD-XOR). Furthermore, if the attacker does not know the security strength chosen for encrypting each column, s/he does not know how many encryption rounds were executed for each ciphered value.

By making the values of $XorK_i$ and $ModK_i$, distinct between columns, we also make encrypted values independent from each other between columns. Even if the attacker breaks security of one column in one table row, the information obtained from discovering the remaining encryption keys is limited. Thus, the attacker cannot infer information enough to break overall security; in order to succeed, s/he must perform recover all the keys for all columns.

Attack costs. To break security by key search in a given column C_i, the attacker needs to have at least one pair (plaintext, ciphertext) for a row j of C_i, as well as the security bit strength involved, as explained in subsection 2.3, because it will indicate the number of rounds that were executed. In this case, taking that known plaintext, the respective known ciphertext, and the C_{N+1} value (storing $RowK_j$ XOR $TabK$, as explained in subsection 2.3), s/he may then execute an exhaustive key search.

The number of cipher rounds for a column C_i is given by NRi, and β is the bit-length of C_i's datatype. Since half the values of vector *Operation* are zeros and the other half are ones, the probability of occurrences of 1 and 0 is equal, *i.e.*, *Prob*(*Operation*[*t*]=0) = ½ = *Prob*(*Operation*[*t*]=1), where the number of possible values for *Operation*[*t*] is 2^{NRi}. Considering β, each *XorK* and *ModK* subkey also has a length of β bits and thus, each *XorK* and *ModK* subkeys have a search space with 2^{β} possible values. *TabK* is a 128 bit value, thus with a search space of 2^{128} possible values. Considering the cipher's algorithm and given the probability of {0, 1} values in *Operation*, a XOR is executed in all rounds (*NRi*), while a MOD is executed before the XOR in half the rounds (*NRi*/2). Given this, the key search space dimension considering the combination of XOR and MOD/XOR rounds is given by $G(x)$:

$$G(x) = \sum_{x=1}^{NRi+\frac{NRi}{2}} F(x) \,.\, 2^{(\beta x)+128}$$

$$F(x)=\begin{cases}\binom{NRi-x}{\frac{NRi}{2}-x} & ,x=1\\ F(x-1)+(-1)^{x}\binom{NRi-x}{\frac{NRi}{2}-x} & ,2<=x<=NRi/2\\ F(x-1) & ,NRi/2+1<=x<=NRi\\ F(x-1)+(-1)^{(x-\frac{NRi}{2})}\binom{x-\frac{NRi}{2}-1}{x-NRi-1} & ,NRi+1<=x<=NRi+NRi/2-1\\ \binom{NRi}{\frac{NRi}{2}} & ,x=NRi+NRi/2\end{cases}$$

Considering Y as the number of attempts to discover the keys, Y is a discrete random variable with support $S = \{1\ldots N\}$, where N represents the search space's dimension. For one attempt, considering a random variable B, it has only two possibilities:

$$B=\begin{cases}0, & \textit{given the attempt is not successful}\\ 1, & \textit{given the attempt is successful}\end{cases}$$

Therefore, B follows a Bernoulli distribution with probability $p = Prob(B=1) = 1/N$. Since the number of attempts is limited, given the search space is finite, variable Y also has a finite support $S = \{1\ldots N\}$. The probability of being successful after k attempts is given by: $Prob(Y=k)=Prob(\bar{A}\cap\bar{A}\cap\ldots\cap\bar{A}\cap A)=\left(1-\frac{1}{N}\right)^{k-1}.\frac{1}{N}$, k=1… N

Note that the probability of being needed more than m attempts is given by:

$Prob(Y>m)=\sum_{k=m+1}^{N}Prob(Y=k)=\sum_{k=m+1}^{N}\left(1-\frac{1}{N}\right)^{k-1}.\frac{1}{N}=(1-1/N)^{m}.\left[\left(1-\left(1-\frac{1}{N}\right)^{N-m}\right)\right].$

The probability of needing n more attempts, given m initial unsuccessful attempts (for $m > 1$ and $n > 1$) is given by $Prob(Y>m+n \mid Y>m) = Prob(Y>m+n) / Prob(Y>m)$, since the event $\{Y > m+n\}$ is contained in $\{Y > m\}$, which means that after having m unsuccessful attempts, being successful after n more attempts only depends on those n additional attempts and not on the initial m attempts, *i.e.*, it does not depend on the past. For the complete search space, the average number of attempts is then given by:

$\sum_{k=1}^{N}k.Prob(Y=k)=\frac{1}{N}\sum_{k=1}^{N}k\left(1-\frac{1}{N}\right)^{k-1}=(*)$.

From the series theory it is known that $\sum_{k=0}^{+\infty}x^{k}=\frac{1}{1-x}$, if $|x|<1$, which is the case in $(*)$ for $\left(1-\frac{1}{N}\right)$. Thus, $(\sum_{k=1}^{+\infty}x^{k})'=\left(\frac{1}{1-x}\right)' \Leftrightarrow \sum_{k=1}^{+\infty}k.x^{k-1}=\frac{1}{(1-x)^{2}}$, $|x|<1$.

Thus, the average number of attempts for finding the keys is $(*)=\frac{1}{N}.\frac{1}{\left(1-\left(1-\frac{1}{N}\right)\right)^{2}}=N$ which is equal to the dimension of the key search space (N). Note however, that this is the worst case complexity. It is possible for the attacker to reduce the key search space by chosen plaintext attacks. Since the same *TabK* key is used for encrypting all *RowK*, as explained in previous subsection (C_{N+1}(row j) = $RowK[j] \oplus TabK$), the information leakage given by $y_1 \oplus y_2=(x_1 \oplus TabK)\oplus(x_2 \oplus TabK) \Leftrightarrow y_1 \oplus y_2=(x_1 \oplus x_2)\oplus(TabK \oplus TabK)$ $\Leftrightarrow y_1 \oplus y_2=x_1 \oplus x_2$ implies that C_{N+1}(row j) $\oplus$ C_{N+1}(row j+1) = $RowK[j] \oplus RowK[j+1]$, reducing the possible search space for *RowK* to 2^{64} instead of 2^{128} in each row. If the attacker manages to use very low *RowK* values, which are most probably smaller than

the value of the *ModK* encryption keys (*i.e. RowK*<*ModK*[*t*]), then the (*RowK* MOD *ModK*[*t*]) – *ModK*[*t*] operation in the cipher will be reduced to *RowK* – *ModK*[*t*], thus further reducing complexity. In this case, for example, taking more than one (*plain-text*, *ciphertext*) pair $y_1 = Encrypt(x_1,2)$ and $y_2 = Encrypt(x_2,2)$ for 2 encryption rounds on the same row, where *Operation*=[0,1]:

$$y_1 \oplus y_2 = (x_1 \oplus XorK[1] + RowK - ModK[2]) \oplus (x_2 \oplus XorK[1] + RowK - ModK[2])$$

Considering that each x_i has a length of β bits, given the encryption key *RowK* has a reduced search space of 2^{64} (as previously mentioned) and each *XorK* and *ModK* have a search space of 2^{β}, the key search space in this example is given by $2^{2\beta+64}$. Since *XorK*[1] and *ModK*[2] are just half the keys for the 2 round SES-DW, to obtain the remaining *XorK*[2] and *ModK*[1] keys, the search space is incremented by $2^{2\beta}$. Since the number of *XorK* and *ModK* encryption keys is the same as the number of rounds, the generic expression for the reduced key search space in this type of attack is given by $G(x) = 2^{NRi*\beta+64} + 2^{NRi*\beta}$. Note that for an 8 bit value (β = 8) encrypted by 16 rounds (*NRi* = 16), using 16 *XorK* and *ModK* subkeys with 8 bits each (each total key length for *XorK* and *ModK* is 16*8 = 128 bits), the key search space complexity is $2^{192} + 2^{128} \cong 6{,}3\text{x}10^{57}$, which remains a considerable measure of security strength.

SES-DW Entropy. In information theory, entropy is a measure of randomness or uncertainty. In this context, the term usually refers to Shannon's entropy, which quantifies the randomness of a variable based upon the knowledge of the information contained in its message. The entropy of a discrete variable *X* with *n* bits in length is given by the following expression, where $Prob(x_i)$ is the probability of occurrence of each x_i within the probability distribution of all possible integer values $[1\ldots2^n]$:

$$Entropy(X) = -\sum_{i=1}^{2^n}\left(Prob(X = x_i).\,log_2 Prob(X = x_i)\right)$$

Since numeric datatype storage sizes are typically 8, 16, 32, 64 or 128 bits, each of our cipher's input/output values (as well as the encryption keys) respectively have a number of 2^8, 2^{16}, 2^{32}, 2^{64}, or 2^{128} possible combinations. While it is computationally fast to obtain the probability distribution in the first case by combining all possible input and encryption key values (with all 8 bit values = $[1...2^8]$) using two cipher rounds (the minimum number of rounds), for the remaining (2^{16}, 2^{32}, 2^{64} and 2^{128}) the task gets exponentially time-expensive. Therefore, after a series of statistical regression experiments using the calculated 8 bit probability distribution for SES-DW, we found that the logarithmic regression ($y = a + b.\,ln(x)$) generated the most adjusted statistical model for representing the cipher's probability distribution (with R^2>=0.98 and a standard error of 0.001). Knowing that the accumulated probability for *n* bits must be equal to 1, using the logarithmic regression function we must ensure that:

$$\int_1^{2^n} a + b.\,ln(x)\,dx = 1$$

This expression leads to $Prob(x_i) = \hat{a} + \hat{b}.\,ln(x_i)$, representing the estimated probability distribution function for *n* bits SES-DW, where:

$$\hat{a} = \frac{1 - n.b.2^n.ln(2)}{2^n - 1} + b \quad \wedge \quad \hat{b} = \frac{\bar{X} - \left(2^{n-1} + \frac{1}{2}\right)}{2^{2n-2} - \frac{1}{4} - n\,.\,2^{n-1}\,.\,\ln(2)}$$

Given $Prob(x)$, the entropy of SES-DW for n=8, 16, 32, 64 and 128 bits is shown in Table 1. As seen, the entropy produced for n bits is nearly n, thus meaning the generated ciphertexts are very close to a uniformly random n bit value.

Table 1. Estimated SES-DW entropy values

Number of bits (n)	SES-DW Entropy
8	7,967144
16	15,972308
32	31,979863
64	63,986246
128	127,989741

4 Experimental Evaluation

We used the TPC-H benchmark [17] (1GB and 10GB scale sizes) and a real-world sales DW storing one year of commercial data (taking up 2GB of data). We tested all scenarios using Oracle 11g and Microsoft SQL Server 2008 DBMS, on a Pentium Core2Duo 3GHz CPU with a 1.5TB SATA hard disk and 2GB RAM (512MB of devoted to database memory cache), with Windows 2003 Server. The TPC-H schema has one fact table (*LineItem*), and seven dimension tables. The Sales DW database schema has one fact table (*Sales*) and four dimension tables. In TPC-H setups, four numerical columns of *LineItem* were encrypted (*L_Quantity*, *L_ExtendedPrice*, *L_Tax* and *L_Discount*). In the Sales DW, five numerical columns were encrypted (*S_ShipToCost*, *S_Tax*, *S_Quantity*, *S_Profit*, and *S_SalesAmount*). We compare our solution with the column-based AES128, AES256 and 3DES168 algorithms, and OPES [3] and Salsa20 [5, 6]. OPES and Salsa20 were implemented using C++.

4.1 Analyzing Storage Size and Loading Time

Tables 2 and 3 show the results of data storage size and loading time (in seconds), respectively, for loading the TPC-H 1GB *LineItem* table in each setup. The results in the remaining databases are similar, with absolute values nearly proportional to their database sizes, and due to lack of space and to avoid redundancy are not included. The results shown are an average of six executions for each tested scenario on each DBMS (with standard deviation in Oracle 11g between [2.27, 22.12], and in SQL Server 2008 between [3.19, 20.45]).

Table 2. TPC-H 1GB Lineitem Fact Table Storage Size Overhead

	Oracle TPC-H 1GB Storage Size (Overhead)	SQL Server TPC-H 1GB Storage Size (Overhead)
Standard	772MB	1237MB
AES128/256	1960MB (+1188MB / 154%)	2410MB (+1173MB / 95%)
3DES168	1572MB (+800MB / 104%)	2181MB (+944MB / 76%)
OPES	790MB (+18MB / 2%)	1258MB (+21MB / 2%)
Salsa20	1064MB (+292MB / 38%)	1553MB (+316MB / 26%)
SES-DW	868MB (+96MB / 12%)	1339MB (+102MB / 8%)

Table 3. TPC-H 1GB Lineitem Fact Table Loading Time Overhead

	Oracle TPC-H 1GB Loading Time (Overhead)	SQL Server TPC-H 1GB Loading Time (Overhead)
Standard	253 s	171 s
AES128	608 s (355 s / 141%)	382 s (211 s / 123%)
AES256	636 s (383 s / 152%)	407 s (236 s / 138%)
3DES168	617 s (364 s / 144%)	389 s (218 s / 127%)
OPES	353 s (100 s / 40%)	229 s (58 s / 34%)
Salsa20	419 s (166 s / 66%)	281 s (110 s / 64%)
SES-DW128	279 s (26 s / 10%)	191 s (20 s / 12%)
SES-DW256	294 s (41 s / 16%)	199 s (28 s / 16%)
SES-DW1024	451 s (198 s / 78%)	284 s (113 s / 66%)

As shown, OPES and SES-DW have much smaller storage space overheads (2% to 12%, 18MB to 102MB) than Salsa20 (26% to 38%, 292MB to 316MB), 3DES168 (76% to 104%, 800MB to 944MB) and AES (95% to 154%, 1173MB to 1188MB of overhead). However, in loading time, SES-DW presents the best results by far (10% to 16%, 20 to 41 seconds of overhead). Considering these results, SES-DW is much more efficient, introducing small overheads for similar key sizes. Note that the worst result for SES-DW 1024, which is similar to Salsa20; however, it refers to using 1024 bit encryption keys, far higher than the remaining tested algorithms. Also note that the results for the TPC-H 10GB database are approximately proportional to those of the 1GB database, which means ten times bigger. Since 1GB is actually a very small size for a DW database, it is easy to conclude that the overheads introduced by encryption are extremely significant and may in fact introduce considerable hardware cost.

4.2 Analyzing Database Query Performance

The TPC-H workload included the benchmark queries 1, 3, 6, 7, 8, 10, 12, 14, 15, 17, 19 and 20 (all accessing fact table LineItem). For Sales DW, the workload was a set of 29 queries, all processing the Sales fact table, as a set of usual decision support daily (9 queries), monthly (9 queries) and annual (11 queries) queries. All results are an average from six executions in each scenario (Oracle 11g standard deviations between [0.47, 42.23] and [0.55, 61.34] for 1GB and 10GB TPC-H, respectively, and [0.63, 59.17] for the Sales DW, and SQL Server between [0.56, 49.56] and [0.63, 58.30] for 1GB and 10GB TPC-H, respectively, and [0.47, 66.08] for the Sales DW). Figure 3 shows total workload execution time overhead for each scenario, while Figure 4 shows the same for CPU time overhead. The Standard execution time (execution time of the workload against a non-encrypted database) for each scenario is 492, 5037, and 1766 seconds in Oracle 11g, and 452, 4294, and 1690 seconds in SQL Server 2008, for the 1GB, 10GB TPC-H and Sales DW, respectively.

It can be seen that SES-DW with 128-bit and 256-bit security has the best response and CPU time overheads for all scenarios, followed by Salsa20 and further by AES, while OPES has results leveled between AES and 3DES. Notice that observing the results for the TPC-H database, SES-DW shows better scalability than the remaining ciphers. In fact, SES-DW 1024-bit in the TPC-H 10GB is nearly as fast as Salsa20, the best solution after SES-DW. This means that the relative gains by using SES-DW

increases as database size scales up, compared with the remaining ciphers. Notice that being 100% faster in TPC-H 10GB means a saving of 5037 seconds (almost 1,5 hours) in total query workload response time.

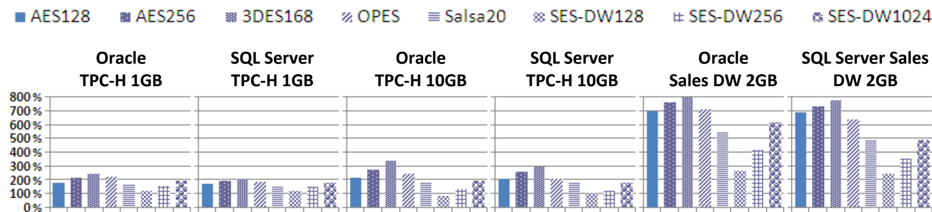

Fig. 3. Total query workload response time overheads (%) for each setup

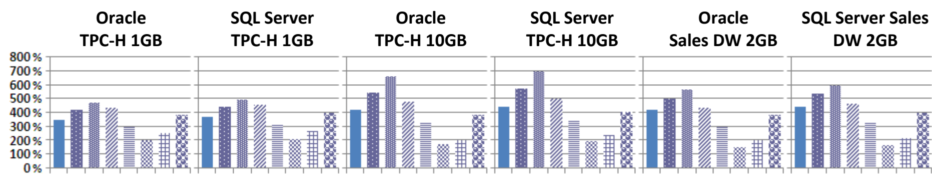

Fig. 4. Total query workload CPU time overheads (%) for each setup

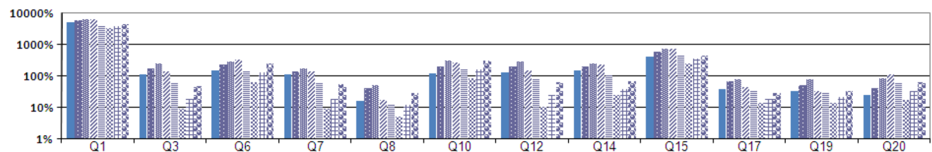

Fig. 5. TPC-H 10GB individual query exec. time overhead p/encrypt. algorithm in Oracle 11g

Considering these results, since 10GB is actually a small size for a DW database, it is easy to conclude from the overall results that performance overheads introduced by data encryption algorithms in DWs are in fact extremely significant, and even minimum gain in response/CPU time is an important achievement.

The results for individual query execution time in Oracle 11g for TPC-H 10GB scenarios are shown in Figure 5, with a logarithmic scale. These results show that all queries have similar proportional overhead to those of the complete workload. This is also true for all the other scenarios, making it redundant to include all in this section. It can be seen that most queries processed by AES and 3DES have overheads of several orders of magnitude higher than SES-DW.

The number of CPU clock cycles spent on encryption and decryption depends on the algorithm and CPU architecture in which they are executed. As an example, the work in [7] refers that AES [2] with a 128 bit key takes up, on average, 20 clock cycles per encrypted byte on a Pentium IV, for encrypting a 16 byte value, resulting in a total of 20 x 16 = 320 clock cycles. The same algorithm with a 256 bit key takes up an average of 28 clock cycles per encrypted byte, meaning it needs 28*16 = 448 clock cycles for encrypting the same 16 byte value. We measured a speed of 8.53 cycles per byte for SES-DW on a Pentium IV for 128 bits encryption values. This makes SES-DW more than twice as fast as AES 128 on the same CPU model.

5 Related Work

The work in [4] proposes perturbed tables in a DW for preserving privacy that obfuscates data and explain data reconstruction for executing queries. Although providing strong guarantees against privacy breaches, these methods produce errors in data reconstruction, which we avoid. A lightweight database encryption scheme for column-oriented DBMS is proposed in [9], with low decryption overhead. In [3] an Order Preserving Encryption Scheme (OPES) for numeric data is proposed, by flattening and transforming the plain text distribution onto a target distribution, based on value-based buckets. This solution allows any comparison operation to be directly applied on encrypted data. A similar solution for processing queries without decrypting data was proposed by [10], using the database-as-a-service paradigm.

The Data Encryption Standard (DES) [8] is a 64 bit block cipher which uses a 56 bit key. As an enhancement of DES, the Triple DES (3DES) encryption standard was proposed [1]. The 3DES encryption method is similar to the original DES, but it is applied three times to increase the encryption level, using three different 56 bit keys. Thus, the effective key length is 168 bits. The algorithm increases the number of cryptographic operations, making it one of the slowest block cipher methods. The Advanced Encryption Standard (AES) is currently the most used encryption standard [2]. AES provides three key lengths: 128, 192 and 256 bits. It is fast and able to provide stronger encryption, compared to other algorithms such as DES [13]. Brute force attack is the only known effective attack known against it. As we have demonstrated in [16], these ciphers introduce very much performance overhead for DWs.

In the search for more computationally efficient algorithms by exchanging a small number of complex operations such as S-box lookups for longer chains of simpler operations, the Salsa20 (alias Snuffle) family of ciphers [6] was proposed. These ciphers have been well studied and are considered fast high security solutions.

An Enterprise Application Security solution is presented in [15], acting as a wrapper/interface between user applications and the encrypted database server. This solution aims to ensure data integrity and efficient query execution over encrypted databases, by evaluating most queries at the application server and retrieving only the necessary records from the database server.

6 Conclusions and Future Work

We propose an encryption solution specifically designed for enhancing data confidentiality in DWs. This solution is transparent and only require user applications to send their queries to a middleware security broker instead of the DBMS. Only the final processed results are returned to the authorized user applications that requested them. All SQL commands and actions are encrypted and stored in a log by the security broker, which can be audited by any user with administration rights. In the database, the data always stays encrypted, never allowing breaches before queries finish execution. If an attacker bypasses the broker and gains direct access, s/he just sees encrypted “realistic-looking” values. In addition, since data schemas and column-

types are preserved and the encrypted data is realistic but not real, our method allows using the database (or "as-is" replicas) for testing purposes and direct querying during application software development, generating realistic but not real results. This also avoids disclosure of the real original data if any attacker bypasses database access control and can retrieve data directly from the database. The proposed solution is independent from DBMS and CPU specific features and requires small computational efforts and can be straightforward and easily implemented in any database. Since it basically works by transparently rewriting user queries, it minimizes efforts in changing user applications and does not jeopardize network and I/O bandwidth. Our technique shows better database performance than standard and state-of-the-art encryption solutions while providing considerable security strength, making it a valid option for balancing performance with security from the DW perspective. As future work, we intend to take advantage of the history log stored in the Black Box in order to manage intrusion detection for attackers that obtain valid database login credentials.

7 References

1. 3DES, Triple DES, National Bureau of Standards, Nat. Inst. of Standards and Technology (NIST), Fed. Inform. Processing Standards (FIPS) Pub. 800-67, ISO/IEC 18033-3, 2005.
2. AES, "Advanced Encryption Standard", NIST, FIPS-197, 2001.
3. R. Agarwal, J. Kiernan, R. Srikant, and Y. Xu, "Order-Preserving Encryption for Numeric Data", ACM SIG Conf. on Management Of Data (SIGMOD), 2004.
4. R. Agrawal, R. Srikant, and D. Thomas, "Privacy Preserving OLAP", ACM SIG Conf. Management Of Data (SIGMOD), 2005.
5. D. J. Bernstein, Snuffle 2005: The Salsa Encryption Function, http://cr.yp.to/snuffle.html.
6. D. J., Bernstein, "The Salsa20 Family of Stream Ciphers", New Stream Cipher Designs - The eSTREAM Finalists 2008, Springer LNCS 4986, 2008.
7. D. J. Bernstein, and P. Schwabe, "New AES Software Speed Records", Int. Conf. Cryptography in India (INDOCRYPT), 2010.
8. DES, Data Encryption Standard, National Bureau of Standards, Nat. Inst. of Standards and Technology (NIST), Federal Inform. Processing Standards (FIPS) Pub 46, 1977.
9. T. Ge and S. Zdonik, "Fast, Secure Encryption for Indexing in a Column-Oriented DBMS", Int. Conf. Data Engineering (ICDE), 2007.
10. H. Hacigumus, B. R. Iyer, C. Li, and S. Mehrotra, "Executing SQL over Encrypted Data in the Database-Service-Provider Model", ACM C. Management Of Data (SIGMOD), 2002.
11. P. Huey, "Oracle Database Security Guide 11g", Oracle Corp., 2008.
12. R. Kimball and M. Ross, "The Data Warehouse Toolkit", 2nd Ed, Wiley & Sons Inc., 2002.
13. A. Nadeem and M. Y. Javed, "A Performance Comparison of Data Encryption Algorithms", IEEE Int. Conf. on Information and Communication Technologies (ICICT), 2005.
14. Oracle Corporation, "Oracle Advanced Security Transparent Data Encryption Best Practices", Oracle White Paper, July 2010.
15. V. Radha and N. H. Kumar, "EISA – An Enterprise Application Security Solution for Databases", Int. Conf. Inf. Systems Security (ICISS), Springer LNCS 3803, 2005.
16. R. J. Santos, J. Bernardino, and M. Vieira, "Evaluating the Feasibility Issues of Data Confidentiality Solutions from a Data Warehousing Perspective", International Conference on Data Warehousing and Knowledge Discovery (DAWAK), 2012.
17. Transaction Processing Council, "The TPC Decision Support Benchmark H", http://www.tpc.org/tpch/default.asp