Balancing Security and Performance for Enhancing Data Privacy in Data Warehouses

Ricardo Jorge Santos
CISUC – DEI – FCTUC
University of Coimbra
Coimbra, Portugal
lionsoftware.ricardo@gmail.com

Jorge Bernardino
CISUC – DEIS – ISEC
Polytechnic Institute of Coimbra
Coimbra, Portugal
jorge@isec.pt

Marco Vieira
CISUC – DEI – FCTUC
University of Coimbra
Coimbra, Portugal
mvieira@dei.uc.pt

***Abstract*—Data Warehouses (DWs) store the golden nuggets of the business, which makes them an appealing target. To ensure data privacy, encryption solutions have been used and proven efficient in their security purpose. However, they introduce massive storage space and performance overheads, making them unfeasible for DWs. We propose a data masking technique for protecting sensitive business data in DWs that balances security strength with database performance, using a formula based on the mathematical modular operator. Our solution manages apparent randomness and distribution of the masked values, while introducing small storage space and query execution time overheads. It also enables a false data injection method for misleading attackers and increasing the overall security strength. It can be easily implemented in any DataBase Management System (DBMS) and transparently used, without changes to application source code. Experimental evaluations using a real-world DW and TPC-H decision support benchmark implemented in leading commercial DBMS Oracle 11g and Microsoft SQL Server 2008 demonstrate its overall effectiveness. Results show substantial savings of its implementation costs when compared with state of the art data privacy solutions provided by those DBMS and that it outperforms those solutions in both data querying and insertion of new data.**

Keywords-Data warehousing, Data masking, Data obfuscation, Data encryption, Data privacy, Data security

I. INTRODUCTION

Data Warehouses (DWs) store sensitive enterprise data used by analytical, data mining and business intelligence solutions to produce business knowledge. Thus, DW databases store the golden nuggets of the business, making them a major target for inside and outside attackers [26]. As the number of attacks and their complexity increases, efficiently securing DW data is critical [15, 18, 26].

A. Motivation

To protect the privacy of stored data, besides data access policies [7], data masking (alias data obfuscation, *i.e.*, changing data values so their real values are unknown) and encryption algorithms are widely used. Published research has stated that encryption standards are the best method to protect sensitive data at the database level and deliver performance [17, 25, 26, 30]. However, although efficient in their security purpose, data masking and encryption techniques introduce key costs:

- Large processing time/resources for masking or encrypting sensitive data, given routine or hardware access;
- Extra storage space of masked/encrypted data. Since DWs usually have many millions/billions of rows, even small modification of any column size or the creation of masking reference tables introduces large storage space overheads;
- Overhead query response time and allocated resources for retrieving/decrypting data to process queries. Given the huge amount of data typically accessed by DW queries, this is probably the most significant drawback for using data masking/encryption in DWs [3].

DW databases take up huge storage space. Business data is mostly stored in numerical attributes, called facts; fact tables usually take up at least 90% of storage space [13]. Decision support queries typically process huge amounts of data, resulting in substantial response time (usually from minutes or hours) [13]. Given the referred key costs, using encryption techniques introduces very considerable storage space and response time overheads, as shown in our experiments (section III). Moreover, many studies [9, 10, 16, 24, 25] have shown that those overheads dramatically degrade database performance, a critical issue in DW scenarios, in a magnitude that ultimately jeopardizes their applicability. Thus, current data masking and encryption solutions are unsuitable for DWs. Specific DW data privacy solutions must always balance security requirements with the desire for high performance, ensuring strong security while keeping database performance acceptable [9, 16, 24]. This is a critical issue and remains a challenge.

B. Our Proposal

We wish to make it clear that our proposal does not intend to replace any standard encryption algorithm, rather it is an alternative solution specifically developed for DW data privacy. It is not our aim to propose a solution as strong in security as the state of the art encryption algorithms, rather a technique that provides an overall considerable level of security while introducing very small overheads in storage space and database performance, *i.e.*, that presents better tradeoffs between security and performance, a critical issue to determine the feasibility of these solutions for DWs.

A data privacy solution may be useless if it assures high security strength, but is too slow to be considered acceptable in practice [16]. In this sense, we propose a data masking technique for numerical facts, balancing tradeoffs between data security and database performance. It is low-cost and straightforward to implement in any DBMS, requiring small efforts to be used. Our proposal uses the MOD operator (which returns the remainder of a division expression) and simple arithmetic operations to mask data and provide a significant level of apparent randomness for the masked values. Our solution can also use one of the masking keys for injecting false data into the DW in order to mislead attackers and increase the overall security level, making them unable to distinguish true from false data.

The proposed solution is transparent; to query the database, user applications need only to send their queries to a middleware security broker – which has access to the masking keys and rewrites user queries to correctly process the required data – instead of the DBMS. Only the final processed results are returned to the authorized user applications that requested them. All SQL commands and actions required to be executed 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 processed data remains masked at all times, never allowing breaches before the queries finish execution. If an attacker bypasses the broker and gains direct access, s/he will only see masked values, as well as mixed true and false data rows.

To evaluate our proposal, we include experiments using two leading commercial DBMS, Oracle 11g and Microsoft SQL Server 2008, and one open-source DBMS, MySQL Server 5.5. The results show that our technique is much better than those of standard data privacy algorithms, for nearly all queries in all tested scenarios. The overall results show that our proposal is a valid alternative and supplies the additional advantages of false data injection facilities.

C. *Main Contributions*

The main contributions of our solution are as follows:

It presents better security and performance tradeoffs than standard data privacy solutions, by significantly decreasing data storage space and processing overheads in inserting as wells as querying DW data;

Contrarily to other similar solutions that pre-fetch the data to process, by simply rewriting queries we avoid network bandwidth congestion;

The stored data is masked at all times, which allows using the database (or "as-is" direct replicas) for testing purposes and direct querying during application software development, since the data is always realistic but not real and generates realistic but not real results;

The solution can inject false data to increase the DW overall security level. To our knowledge, this is the first data privacy solution specifically proposed for DWs that integrates data injection features for enforcing data security.

D. *Paper Structure*

The remainder of this paper is organized as follows. In section II we describe our technique and point out the main issues involved in its use. In Section III, we discuss our solution's security and performance issues. Section IV presents experimental evaluations using the TPC-H decision support benchmark and a real-world DW. Section V presents related work, describing standard and state of the art data masking and encryption solutions for providing data privacy and discussing the issues from a DW perspective. Finally, section VI presents our conclusions and future work.

II. MOBAT: MODULUS BASED DATA MASKING TECHNIQUE FOR DATA WAREHOUSES

A. *MOBAT Functional Architecture*

The system's architecture is shown in Figure 1, made up by three entities: 1) the masked database and its DBMS; 2) the MOBAT security application (MOBAT-SA); and 3) user/client applications to query the masked database. The MOBAT-SA is a middleware broker between the masked database DBMS and those applications, ensuring queried data is securely processed and proper results are returned to those applications. All communications are made through SSL/TLS secure connections, to protect SQL instructions and returned results between the entities, avoiding problems from intercepting messages by attackers on the network.

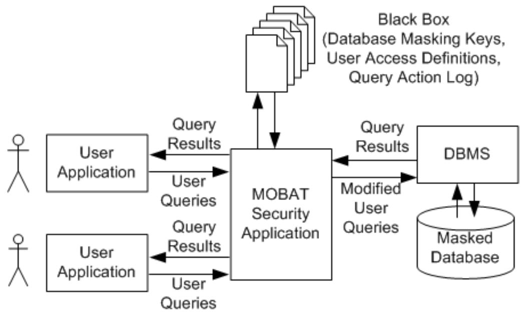

Figure 1. The MOBAT Data Security Architecture

The Black Box is a set of files in a directory of the file structure of the database server, created for each masked database. This process is similar to an Oracle Wallet, which keeps all encryption keys and definitions for each Oracle Database [11]. However, contrarily to Oracle, where a DBA is free to access the Wallet whenever s/he wishes, in our solution only MOBAT-SA can access the Black Box, *i.e.*, absolutely no user has direct access to its content. In the Black Box, MOBAT-SA will store all the created masking keys and predefined data access policies for the concerned database. The MOBAT-SA will also create a history log for saving a duplicate of all instructions and actions executed in the database, for auditing and control purposes. All Black Box contents are encrypted using AES standard encryption algorithm [2] with a 256 bit key. In case of losing the Black Box of a certain database, there is no way to restore its true data, except to crack the masking keys.

Our solution uses three masking keys; two are private and one is public. MOBAT-SA generates all masking keys and their values are never shown or known by the DBA or any other user. To obtain true results, user queries or actions must pass through MOBAT-SA, which will store a copy of those instructions in the history log. Each time a user requests any action, MOBAT-SA will receive and parse the instructions, fetch the necessary masking keys, rewrite the query, send it to process by the DBMS and retrieve the results, and finally send those results back to the application that issued the request.

B. Implementing MOBAT on a Database

A DBA requires masking a database through MOBAT-SA. Entering DBA login and database connection data, MOBAT-SA will try to login to that database. If it succeeds, it will create the Black Box, scanning and storing all database data access policies for users and their permissions. An action log for saving all further user actions will also be created, as explained earlier. After that, MOBAT-SA will ask the DBA which tables and columns are to be masked. All needed private masking keys for each table and column will be generated, encrypted and stored in the Black Box. Finally, MOBAT-SA will apply the masking formula (explained in subsection II.C) on all data to mask. Whenever database updates are needed, they should be done through the MOBAT-SA, which will apply the masking routine to the referenced values and store them directly in place.

C. The MOBAT Data Masking Technique

Most facts in DWs are columns with numerical values [13]. Since fact tables usually represent more than 90% of the DW's total size [13], numeric-type columns represent the largest portion of business data. MOBAT aims on masking the DW's numerical values while introducing small overheads in computational efforts for query processing.

Suppose a table T with a set of N numerical columns $C_i = \{C_1, C_2, C_3, \ldots, C_N\}$ to mask and a total set of M rows $R_j = \{R_1, R_2, R_3, \ldots, R_M\}$. Each value to mask in the table will be identified as a pair (R_j, C_i), where R_j and C_i respectively represent the row and column to which the value refers. The masking formula depends on the following predefinitions:

- K_1 and K_2 are private keys stored in the Black Box, known only by MOBAT-SA;
- K_1 is a 128 bit random generated value, constant for T;
- K_2 is a 128 bit random generated value, ranging between the minimum and maximum positive integer value possible of column C_i. There is a K_2 for each column C_i to be masked, represented by $K_{2,i}$;
- K_3 is a public key based on a 128 bit column appended to each row R_j in T, filled in with a random value between 1 and 2^{128}, represented by $K_{3,j}$.

Suppose each value to mask as (R_j, C_i). Each new masked value $(R_j, C_i)'$ is obtained by applying the following formula (1) for row j and column i of table T:

$$(R_j, C_i)' = (R_j, C_i) - ((K_{3,j} \text{ MOD } K_1) \text{ MOD } K_{2,i}) + K_{2,i} \qquad (1)$$

MOD is the modulus operator returning the remainder of a division expression. Since K_1 and $K_{2,i}$ are constant values for the table and each column, respectively, and $K_{3,j}$ is stored with each row in the table, the inverse formula of (1) for retrieving the original value is shown as formula (2):

$$(R_j, C_i) = (R_j, C_i)' + ((K_{3,j} \text{ MOD } K_1) \text{ MOD } K_{2,i}) - K_{2,i} \qquad (2)$$

If the values of $K_{3,j}$ were stored in a lookup table separate from table T, a heavy join operation between those tables would be required to unmask data. Given the typical enormous number of rows in fact tables, this should be avoided at all cost. To avoid table joins, the values of $K_{3,j}$ must be stored along with each row j in table T. To accomplish this, there are two possible solutions:

1. A new column is created and added to table T for storing each $K_{3,j}$ value;
2. Table T is recreated with the inclusion of $K_{3,j}$ in the CREATE TABLE statement from the start and then restoring the table's data.

The second option implies additional efforts and time to rebuild table T, depending on its size. However, it should speed up query response time compared to the first option, since the new column $K_{3,j}$ is included with the original data in each row; the second option stores it physically apart from the original data. Impact on performance is seen in section III (*MOBAT_AddCol* and *MOBAT_CreateCol*). A third option for defining $K_{3,j}$ values and speed up performance is to use any long integer typed column C_Z, which is already part of the original data structure of table T, instead of creating an extra column for $K_{3,j}$ in T. In this case, no changes in table T data structure are needed, eliminating storage space overhead. The results for this third option are also shown in section III, where each primary key of the fact tables are used as $K_{3,j}$ (*MOBAT_ColKey*).

D. Querying the MOBAT Masked Database

Whenever user applications execute a query, they submit it to the MOBAT-SA, which rewrites the received query in order to process it with the real data values, using formula (2) to replace the respective masked columns used in the query, and checking the user access authorization in the Black Box. To rewrite the user query, MOBAT-SA searches for which tables and columns it needs to process, and looks up the Black Box for retrieving K_1 and $K_{2,i}$ data masking keys, respectively, as well as the needed $K_{3,j}$ key fields in those tables. As an example, suppose the TPC-H benchmark *LineItem* table [23] has four numerical fact columns (*L_Quantity*, *L_ExtendedPrice*, *L_Tax* and *L_Discount*) to mask by MOBAT. Suppose that MOBAT has generated and filled in a new column *L_KeyK3* for the *j* rows of that table, which will act as the public $K_{3,j}$ key values, and has stored the value of 9342 for key K_1 referring to the *LineItem* table, $K_{2,L\_Quantity} = 12$, $K_{2,L\_ExtendedPrice} = 51234$, and $K_{2,L\_Discount} = 4$ (for example). Consider TPC-H query 6:

```
SELECT SUM(L_ExtendedPrice * L_Discount) AS
Revenue
FROM   LineItem
WHERE  L_ShipDate>=TO_DATE('1994-01-01')
   AND L_ShipDate<TO_DATE('1995-01-01')
   AND L_Discount BETWEEN 0.05 AND 0.07
   AND L_Quantity<24
```

The new query, rewritten by the MOBAT-SA and submitted to the DBMS will be:

```
SELECT
 SUM((L_ExtendedPrice+MOD(MOD(L_KeyK3,9342),51234)
 -51234) * (L_Discount+MOD(MOD(L_KeyK3,9342),4)-4))
       AS Revenue
FROM   LineItem
WHERE  L_ShipDate>=TO_DATE('1994-01-01')
   AND L_ShipDate<TO_DATE('1995-01-01')
   AND (L_Discount+MOD(MOD(L_KeyK3,9342),4)-4)
        BETWEEN 0.05 AND 0.07
  AND (L_Quantity+MOD(MOD(L_KeyK3,9342),12)-12)<24
```

As seen in the example, query parsing and adaptation is a straightforward operation, replacing each masked column with their respective reverse formula (2). This is valid for any type of query, including equality and range queries, as well as built in functions. These changes are handled transparently by the broker and kept hidden from the users. Only the query results are returned to the user application.

E. *Using False Data Injection with MOBAT*

MOBAT can inject false rows amongst the fact tables, making it increasingly difficult to distinguish true and false data. To do this, instead of generating independent random numbers for $K_{3,j}$ keys in each row as shown in subsection 2.3, we redefine $K_{3,j}$ as a multiple of the sum of the true values of all $C_{i,j}$ columns to be masked, for each true row j:

$K_{3,j} = (\sum C_{i,j}) * k$, $\{ i = 1…n \}$ where k is a random integer constant that does not overflow 128 bits for $K_{3,j}$ and n is the number of masked columns C in row j (3)

For false rows, random values for filling $C_{i,j}$ would be generated, and the value of $K_{3,j}$ would be equal to any value different from those generated by formula (3). Thus, true rows are verifiable through testing if $K_{3,j}$ is a multiple of the sum of the true unmasked values of all masked columns, using the MOD remainder operator. Formula (4) shows how to test if a certain row j is true or false:

Given R = $K_{3,j}$ MOD ($\sum C_{i,j}$) , $\{ i = 1…n \}$
IF R=0 THEN row j is True ELSE row j is False (4)

There is a tradeoff when using this method: the more false data is injected, the stronger is the table's security level. However, the more data injected, the more data is scanned and verified by queries, decreasing performance. The overall security strength for each fact table is directly dependent on how many false rows are injected into each table and how they are distributed throughout existing data.

III. MOBAT Security and Performance Issues

A. *Security Issues*

Handling transparency and securing communications. All user queries/instructions are managed by MOBAT-SA, which transparently parses and rewrites them to query the DBMS and retrieve the intended results. The users never see the rewritten instructions. For security, MOBAT-SA 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 private keys. All communications between user applications, MOBAT-SA and DBMS are through encrypted SSL/TLS connections. All these actions prevent attackers from accessing the masking keys, rewritten queries/instructions and true data.

Generating apparent randomness for the masked values. Generating randomness for masking and cryptographic applications is a costly and security-critical operation [6]. In order to guarantee their security strength, two same original real data values must generically originate different masking generated values, so a level of apparent randomness is ensured. Given that our masking formula (1) uses two MOD operations in conjunction with randomly generated realistic values, the generated masked values for the same original values are mostly different. To demonstrate this, suppose a table T with two masked columns, *Column1* and *Column2*. Suppose the MOBAT-SA generated the values K_1 = 9264 for table T and $K_{2,1}$ = 12 and $K_{2,2}$ = 78254 for each column. Table I shows the original data for T on the left and its resulting masked content on the right. It can be seen that the same original values of *Column1* result in different masked values, achieving apparent randomness. Of course, this is a very small dataset used only to illustrate these features.

TABLE I. Example of original and MOBAT dataset

T – Original dataset			**T' – MOBAT Masked dataset**		
Column1	*Column2*	$K_{3,j}$	*Column1'*	*Column2'*	$K_{3,j}$
11	91873	7537	22	162590	7537
2	94129	1808	6	170575	1808
18	71624	29636	22	148034	29636
15	84624	34997	22	155673	34997
12	46926	41395	17	120841	41395

Non-invertibility of the masking formula. For a function to be invertible, each output must correspond to no more than one input, *i.e.*, more than one different input cannot generate the same output; a function with this property is called one-to-one, or injective. An injective function must preserve distinctness: it never maps distinct elements of its domain to the same element of its codomain. The MOD operator 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, 27 MOD 4=3, etc). Since MOD operations are non-injective, the MOBAT formula (1) is also non-injective. Given that injectivity is required property for having invertibility, MOBAT is non-invertible. The only way to break its security is to crack the masking key values.

Key Management. As known, the level of security of data masking or encryption solutions does not depend on its secrecy, but on its keys [16]. In our proposal, since K_3 is public (stored in the fact table), only keys K_1 and K_2 need to be cracked. K_1 is a 16 byte integer, *i.e.*, a set of 128 bits. K_2 depends on maximum storage size defined for each column, variable between 1 and 128 bits. This means our technique implies a minimum of 2^{129} key combinations, for K_1 and K_2 together (at least 16 bytes+1 bit), and roughly needs an average number of 2^{128} tests (half of the total possible brute force tests – 50% chance) for discovering the keys using brute force, for each masked column in the table, since K_2 is column dependant. Consequently, the minimum number of combinations needed to discover all key values for a i number of columns is $i * 2^{129}$, resulting in an average of $i * 2^{128} \approx i * 3.4 \times 10^{38}$ brute force tests to discover the keys.

The security strength of standard encryption algorithms is higher than our solution. However, we require much less computational resources while maintaining a considerable level of security, given the high number of possible brute key values. Periodically, the values of all or any one of the K_1, K_2, and K_3 keys can be switched in order to ensure data is properly protected. Moreover, the data injection method also allows increasing our solution's overall security strength. Although it is not possible to absolutely prove that a particular algorithm is secure [9, 12, 16, 17], we believe our technique is secure enough to be acceptable for use.

B. Performance and Transparency Issues

Performance in middleware data privacy solutions. Topologies involving middleware data privacy solutions such as [21] typically request all the masked/encrypted data from the database and perform the unmasking/decrypting actions themselves locally. This strangles the network due to communication costs with bandwidth consumption between middleware and database, jeopardizing data throughput and consequently, response time. In a DW, previously acquiring all the data from the database for processing a query at the middleware is unreasonable, given the amount of data accessed for decision support. In this sense, our MOBAT-SA just rewrites queries and sends them to be processed directly by the DBMS, sending only the results back to the user. This eliminates network overhead from critical path when compared to similar middleware security solutions.

Encryption in Microsoft SQL Server 2008 and MySQL 5.5. Microsoft SQL Server and MySQL 5.5 only encrypt textual or varbinary type values (char, varchar, varbinary, etc). Given that most sensitive columns in DW fact tables store numerical values, when using these DBMS they must be converted to textual or varbinary format. Once decrypted, these values must also be transformed back into numerical format in order to apply arithmetic operations such as sums, averages, etc., adding computational overheads with impact in performance. On the contrary, our solution is specifically designed for masking numerical values, and in this sense, is therefore much more appropriate for protecting DW facts.

Transparently querying masked data. Query instructions in our solution become longer due to replacing each masked column with the formulas for masking or unmasking their stored contents, but this is automatically and transparently managed by the MOBAT-SA. The only change the user applications need is to send the query to the MOBAT-SA, instead of querying the database directly.

IV. EXPERIMENTAL EVALUATION

We used the TPC-H benchmark [23] (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 in the leading DBMS Oracle 11g and Microsoft SQL Server 2008, on a Pentium 2.8GHz CPU with a 1.5TB SATA hard disk and 2GB RAM (512MB of devoted to database memory cache). Oracle 11g ran on Windows XP Professional, while SQL Server on 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 columns of *LineItem* were masked (*L_Quantity*, *L_ExtendedPrice*, *L_Tax* and *L_Discount*), given they are the numerical fact columns. In the Sales DW, five numerical columns were masked (*S_ShipToCost*, *S_Tax*, *S_Quantity*, *S_Profit*, and *S_SalesAmount*), for the same reasons.

Since our solution is column-based, for fairness we compare it with the column-based AES128 and 3DES168 encryption algorithms, given that tablespace encryption has functional primitives that speedup performance, making it unfair to compare with column-based techniques [11, 20]. Moreover, best practice documentation for encryption from both DBMSs [11, 20] recommends column-based encryption when sensitive data consists on small number of well-defined columns. We used AES128 and 3DES168 for comparison because they are, respectively, the fastest and slowest available algorithms in those DBMS [11, 20]. Table II shows the defined experimental encryption/masking scenarios.

TABLE II. EXPERIMENTAL DATA ENCRYPTION/MASKING SCENARIOS

Reference/Label	Description
Standard	Standard data without masking/encryption
AES128 Col	Data encrypted with TDE AES 128 bit key column encryption
3DES168 Col	Data encrypted with TDE 3DES168 column encryption
MOBAT AddCol	Data masked by MOBAT formula (1), where a column for masking keys $K_{3,j}$ has been added to the existing fact table
MOBAT CreateCol	Data masked by MOBAT formula (1), where a column for masking keys $K_{3,j}$ was added to the fact table, completely recreated
MOBAT ColKey	Data masked by MOBAT formula (1), using a numerical column from the original fact table data structure as key $K_{3,j}$

A. Analyzing Storage Size and Data Loading

Tables III and IV show the data storage size and loading time (in seconds) results, respectively, for loading the TPC-H 1GB *LineItem* table in each defined scenario. Figures 2 and 3 show their overhead percentages. The results in the remaining databases are similar, with absolute values nearly proportional to their database sizes, and due to lack of space are not included. The *MOBAT ColKey* setup is not included, since it does not require changing fact table data structure.

In storage space, MOBAT has overheads ranging from 4.1% (32MB) to 5.7% (44MB) in Oracle and 2.8% (35MB) in SQL Server. AES128 and 3DES168 present storage space overheads from 103.6% (800MB) to 153.9% (1188MB) in Oracle and 76.3% (944MB) to 94.8% (1173MB) in SQL Server. In data loading time, MOBAT's overhead ranges from 3.5% (11 seconds) to 7.7% (24 seconds) in Oracle and from 4.3% (9 seconds) to 6.5% (14 seconds) in SQL Server. AES128 and 3DES168 present much greater loading time overheads, from 189.7% (588 seconds) to 191.6% (594 seconds) in Oracle and 123.1% (261 seconds) to 129.2% (274 seconds) in SQL Server.

Considering the results, MOBAT is much more efficient, introducing very small overheads. Since these results are for the 1GB database and the overhead percentages are similar for the remaining scenarios, for TPC-H 10GB, which is ten times bigger, the absolute values of the overheads are also approximately ten times bigger. Proportionally, this means that TPC-H 10GB has nearly 8GB to 12GB of increased storage space, and nearly 43 to 99 minutes of increased loading time. Given that 10GB is actually a small size for a DW database, it is easy to conjecture that the overheads introduced by DBMS data encryption algorithms in DWs are extremely significant and may in fact be impracticable.

B. *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 queries accessing the masked 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 queries, with daily (queries 1 to 9), monthly (10 to 18) and annual (19 to 29) values, including actions like row selection, joining, aggregates, and ordering. Response time results for each query are an average obtained from six executions in each scenario.

Figures 4 and 5 show workload execution time overhead for each scenario. The Standard execution time (execution time of the workload against a non-encrypted/masked database) for each scenario is 626, 6155, and 2233 seconds in Oracle 11g, and 580, 5301, and 2211 seconds in SQL Server 2008, for the 1GB, 10GB TPC-H and Sales DW, respectively. In Oracle 11g, MOBAT ranges from 5.32 (187.7/35.3) times better than standard column encryption solutions for the 1GB TPC-H database, to 9.23 (203/22) times better. In the 10GB TPC-H, gains range from 6.05 (131.8/21.8) to 8.58 (144.2/16.8) times better, and for the Sales DW, from 5.39 (688.3/127.7) to 10.5 (814.7/77.6)

Notice that column encryption introduces a minimum overhead of 131.8% (8112 seconds) in TPC-H 10GB setup (total workload response time takes almost 4 hours, instead of the standard time, which is less than 2 hours), and 688.3% (15370 seconds) in the Sales DW setup (workload response time takes almost 5 hours, instead of the standard 37 minutes). On the other hand, MOBAT introduces a maximum overhead of 21.8% (1342 seconds) in the TPC-H 10GB setup (total workload response time takes little over 2 hours), and 127.7% (2851 seconds) in the Sales DW setup (total workload response time takes almost 1.5 hours).

In SQL Server 2008, shown in figure 5, MOBAT ranges from 4.35 (174.3/40.1) times better than standard column encryption solutions for the 1GB TPC-H database, to 8.60 (195.2/22.7) times better. In the 10GB TPC-H database, the gains range from 7.15 (151.6/21.2) to 14.02 (183.7/13.1) times better, and for the Sales DW, from 5.38 (665.4/123.7) to 11.78 (758.6/64.4). Column encryption introduces a minimum overhead of 151.6% (8036 seconds) in the TPC-H 10GB setup (total workload response time takes almost 4 hours, instead of the standard time, less than 2 hours), and 665.4% (14712 seconds) in the Sales DW setup (workload response time takes almost 5 hours, instead of the standard 37 minutes). On the other hand, MOBAT introduces a maximum overhead of 21.2% (1124 seconds) in the TPC-H 10GB setup (total workload response time still takes less than 2 hours), and 123.7% (2735 seconds) in the Sales DW setup (total workload response time takes almost 1.4 hours).

The results for individual query execution time in Oracle 11g for TPC-H 10GB scenarios are shown in figure 6. These results show that all queries have similar 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 mostly all queries processed by AES and 3DES have overheads of several orders of magnitude higher than MOBAT. All the results in all scenarios in both DBMS also show that the performance of *CreateCol Masking* is better than *AddCol Masking*, which was expected as mentioned in section 2.3, when explaining the technique. The performance results of *ColKey Masking* are the best, given the absence of changes in the original fact table data structure and size.

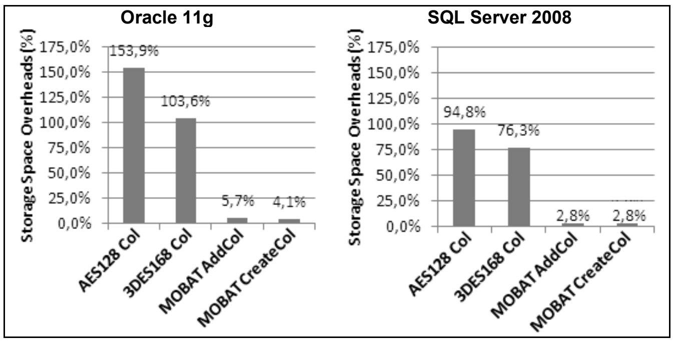

Figure 2. Storage Space Overheads for TPC-H 1GB

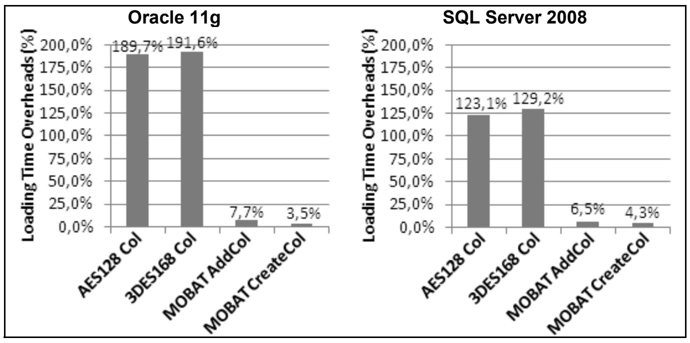

Figure 3. Loading Time Overheads for TPC-H 1GB

TABLE III. TPC-H 1GB LINEITEM FACT TABLE STORAGE SIZES FOR EACH EXPERIMENTAL SCENARIO IN EACH DBMS

DBMS	Standard	AES128 Col	Absolute/Relative Overhead	3DES168 Col	Absolute/Relative Overhead	MOBAT AddCol	Absolute/Relative Overhead	MOBAT CreateCol	Absolute/Relative Overhead
Oracle 11g	772MB	1960MB	*+1188MB / 154%*	1572MB	*+800MB / 104%*	816MB	*+44MB / 6%*	804MB	*+32MB / 4%*
SQL Server 2008	1237MB	2410MB	*+1173MB / 95%*	2181MB	*+944MB / 76%*	1272MB	*+35MB / 3%*	1272MB	*+35MB / 3%*

TABLE IV. TPC-H 1GB LINEITEM FACT TABLE DATA LOADING TIME FOR EACH EXPERIMENTAL SCENARIO IN EACH DBMS

DBMS	Standard	AES128 Col	Absolute/Relative Overhead	3DES168 Col	Absolute/Relative Overhead	MOBAT AddCol	Absolute/Relative Overhead	MOBAT CreateCol	Absolute/Relative Overhead
Oracle 11g	310s	898s	*+588s / 190%*	904s	*+594s / 192%*	334s	*+24s / 8%*	321s	*+11s / 4%*
SQL Server 2008	212s	473s	*+261s / 123%*	486s	*+274s / 129%*	226s	*+14s / 7%*	221s	*+9s / 4%*

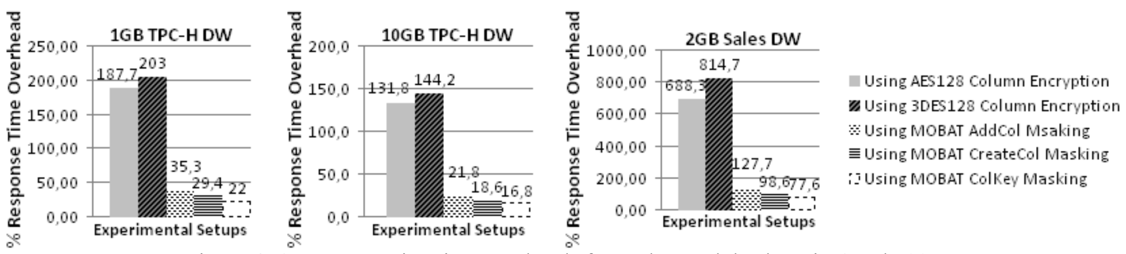

Figure 4. Query execution time overheads for each tested database in Oracle 11g

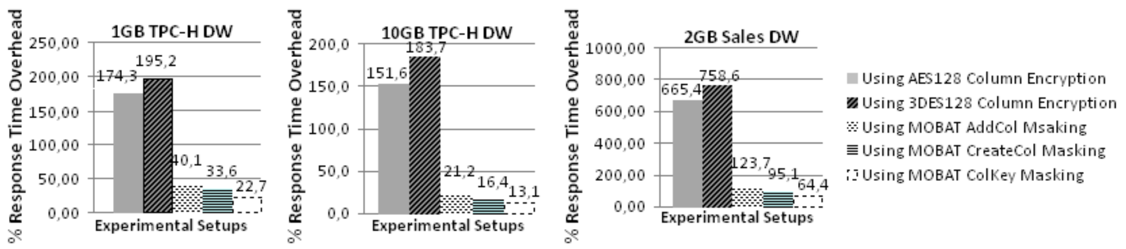

Figure 5. Query execution time overheads for each tested database in SQL Server 2008

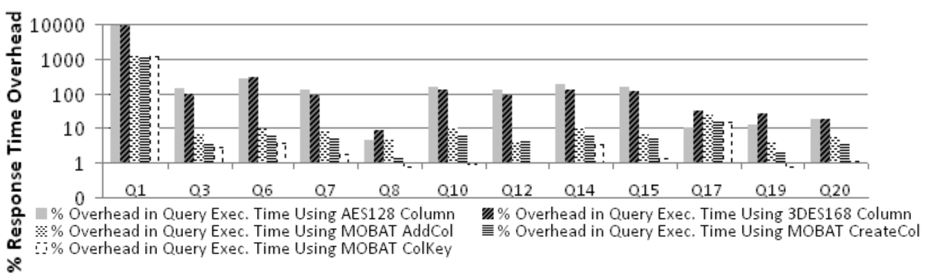

Figure 6. 10GB TPC-H Individual Query Execution Time Overhead per Encryption/Masking Algorithm

TABLE V. TPC-H 1GB WORKLOAD EXECUTION TIME(SECONDS)/OVERHEAD (%) WITH FALSE DATA USING MO BAT IN ORACLE 11G

	+25% False Data (Time/Overhead)	**+50% False Data (Time/Overhead)**	**+75% False Data (Time/Overhead)**	**+100% False Data (Time/Overhead)**
1G TPC-H MOBAT AddCol	1110 sec / 31%	1355 sec / 60%	1601 sec / 89%	1855 sec / 119%
1G TPC-H MOBAT CreateCol	1045 sec / 29%	1264 sec / 56%	1482 sec / 83%	1709 sec / 111%

C. Using False Data Injection with MOBAT

In order to test false data injection scenarios, we inserted 25%, 50%, 75% and 100% of false rows (relatively to the total number of true rows) into TPC-H 1GB. The false rows were uniformly distributed throughout the fact table. Table V shows the results. As it would be expected, the overhead in each scenario is nearly proportional to the amount of false data injected. The results for the remaining database setups, TPC-H 10GB and Sales DW, are similar to those in Table V and due to space restrictions are not included here.

V. RELATED WORK

Data masking solutions are mainly used for creating test databases for software development environments, or for camouflaging data values in publicly available published data [22, 25]. An extensive survey on data masking techniques is given in [22]. The Oracle EM Data Masking Pack (DMP) [19] provides mask primitives for various types of data, replacing real data with realistic-looking values.

A lightweight database encryption scheme for column-oriented DBMS is proposed in [9], with low decryption overhead. In [3] an Order Preserving Encryption Scheme 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]. The work [4] proposes perturbed tables in a DW for preserving privacy and explain data reconstruction for executing queries. Although providing strong guarantees against privacy breaches, these methods produce errors in data reconstruction, which we pretend to avoid.

The work in [21] presents an Enterprise Application Security solution, 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.

The Data Encryption Standard (DES) [8] is a 64 bit block cipher, meaning that data is encrypted/decrypted in 64 bit chunks, using a 56 bit key. This encryption standard is an insecure block cipher [12]. As an enhancement of DES, the Triple DES (3DES) encryption standard was proposed [1]. The 3DES encryption method is similar to the original DES algorithm, 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 [16]. The Advanced Encryption Standard (AES) is 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 [16]. Brute force attack is the only known effective attack known against it.

Data injection has been mostly used for building synthetic datasets for benchmarking and production purposes, *i.e.*, filling in databases for testing the development of databases and applications [5, 27]. To our knowledge, there are no data injection solutions for enforcing data privacy as we propose in our technique.

VI. Conclusions and Future Work

We propose a data masking solution specifically designed for enhancing data privacy in DWs. We also use one of the masked fact tables masking key for enabling false data injection, increasing the overall security strength against attackers that gain direct access to the database.

The data masking formula requires small computational efforts and can be straightforward and easily implemented in any DBMS. Since it basically works by transparently rewriting user queries, it minimizes efforts in changing user applications and does not jeopardize network bandwidth. The masked database can be directly used for production purposes, enabling developing applications to directly query it without passing through the MOBAT-SA, therefore retrieving realistic but not real data, for testing software development. This also avoids disclosure of the real original data if any attacker bypasses database access control and is able to retrieve data directly from the database.

Although we did not conceive our solution as a direct alternative to standard encryption algorithms, we have compared it with the AES and 3DES algorithms provided by leading commercial DBMS. Experimental results show that the introduced storage space and database performance overhead by these standard solutions is very significant from the DW point of view. This enforces stating that those techniques are in fact unfeasible for DW scenarios. Since most DW data are numerical values, our masking technique is tailored for this kind of data. Our technique shows better database performance than the encryption standards, while providing considerable security strength, enforced by the false data injection method. Thus, it is an efficient overall solution and valid alternative for balancing the performance and security issues from the DW perspective.

As future work, we will take advantage of the history log stored in the MOBAT-SA Black Box to manage intrusion detection.

References

[1] 3DES, Triple DES, National Bureau of Standards, National Institute of Standards and Technology (NIST), Federal Information Processing Standards (FIPS) Pub. 800-67, ISO/IEC 18033-3, 2005.

[2] AES, “Advanced Encryption Standard”, National Inst. of Standards and Technology (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] E. Arora, R. Ertin, M. Ramnath, M. Nesterenko, and W. Leal, “Kansei: A High-fidelity Sensing Testbed”, Internet Computing, Vol. 10, 2006.

[6] M. Barbosa and P. Farshim, “Randomness Reuse: Extensions and Improvements”, 12th Institute of Mathematics and its Applications (IMA) Int. Conference on Cryptography and Coding, 2009.

[7] E. Bertino and R. Sandhu, “Database Security – Concepts, Approaches, and Challenges”, IEEE Transactions on Dependable and Secure Computing, Vol. 2, No. 1, 2005.

[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 SIG Int. Conf. on Management Of Data (SIGMOD), 2002.

[11] P. Huey, “Oracle Database Security Guide 11g”, Oracle Corp., 2008.

[12] J. Kim, Y. Lee, and S. Lee, “DES with any reduced masked rounds is not secure against side-channel attacks”, Int. Journal Computers and Mathematics with App., 60, 2010.

[13] R. Kimball and M. Ross, “The Data Warehouse Toolkit”, 2nd Edition, Wiley & Sons, Inc., 2002.

[14] J. Kobielus, “The Forrester Wave: Enterprise Data Warehousing Platforms”, Forrester Research Report, Q1, 2009.

[15] J. McKendrick, “IOUG Data Security 2009: Budget Pressure Lead to Increased Risks”, The Independent Oracle Users Group (IOUG) Security Report, 2009.

[16] Nadeem and M. Y. Javed, “A Performance Comparison of Data Encryption Algorithms”, IEEE Int. Conference on Inform. and Communication Technologies (ICICT), 2005.

[17] R. B. Natan, “Implementing Database Security and Auditing”, Digital Press, 2005.

[18] Oracle Corporation, “Security and the Data Warehouse”, April 2005.

[19] Oracle Corporation, “Data Masking Best Practices”, July 2010.

[20] Oracle Corporation, “Oracle Advanced Security Transparent Data Encryption Best Practices”, Oracle White Paper, July 2010.

[21] V. Radha and N. H. Kumar, “EISA – An Enterprise Application Security Solution for Databases”, Int. Conf. Inf. Systems Security (ICISS), Jajodia and Mazumdar (Eds), Springer LNCS 3803, 2005.

[22] G. K. Ravikumar, et al, “A Survey on Recent Trends, Process and Development in Data Masking for Testing”, Int. Journal of Computer Science Issues, Vol. 8, Issue 2, 2011.

[23] Transaction Processing Council, “The TPC Decision Support Benchmark H”, http://www.tpc.org/tpch/default.asp

[24] M. Vieira and H. Madeira, “Towards a Security Benchmark for Database Management Systems”, Int. Conf. on Dependable Systems and Networks (DSN), 2005.

[25] S. C. Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, “Over-encryption: Management of Access Control Evolution on Oursourced Data”, (VLDB), 2007.

[26] N. Yuhanna, “Your Enterprise Database Security Strategy 2010”, Forrester Research, Sep. 2009.

[27] E. Lo, N. Cheng, and W. Hon, “Generating Databases for Query Workloads”, Int. Conf. on Very Large DataBases (VLDB), 2010.