information

MDPI

Article

Hiding the Source Code of Stored Database Programs

Vitalii Yesin [1], Mikolaj Karpinski [2,*], Maryna Yesina [1], Vladyslav Vilihura [1] and Kornel Warwas [2]

[1] Department of Security of Information Systems and Technologies, Faculty of Computer Science, V. Karazin National University of Kharkiv, 61022 Kharkiv, Ukraine; v.i.yesin@karazin.ua (V.Y.); m.v.yesina@karazin.ua (M.Y.); viligura93@gmail.com (V.V.)

[2] Department of Computer Science and Automatics, Faculty of Mechanical Engineering and Computer Science, University of Bielsko-Biala, 43-309 Bielsko-Biala, Poland; kwarwas@ath.bielsko.pl

* Correspondence: mkarpinski@ath.bielsko.pl

Received: 4 November 2020; Accepted: 7 December 2020; Published: 9 December 2020

Abstract: The objective of the article is to reveal an approach to hiding the code of stored programs stored in the database. The essence of this approach is the complex use of the method of random permutation of code symbols related to a specific stored program, located in several rows of some attribute of the database system table, as well as the substitution method. Moreover, with the possible substitute of each character obtained after the permutation with another one randomly selected from the Unicode standard, a legitimate user with the appropriate privileges gets access to the source code of the stored program due to the ability to quickly perform the inverse to masking transformation and overwrite the program code into the database. All other users and attackers without knowledge of certain information can only read the codes of stored programs masked with format preserving. The proposed solution is more efficient than the existing methods of hiding the code of stored programs provided by the developers of some modern database management systems (DBMS), since an attacker will need much greater computational and time consumption to disclose the source code of stored programs.

Keywords: data security; database; data masking; stored program

1. Introduction

Today, various commercial management systems exist for traditional relational databases and databases (DBs) of the NewSQL class that seek to combine the advantages of NoSQL and the transactional requirements of classical databases in order to

- increase productivity,
- reduce network traffic,
- simplify access to databases for applications,
- hide a lot of specific features of a database management system (DBMS) and database from the user,
- ensure business rules and a higher level of data security.

These systems offer various methods of writing and saving stored programs in the database schema [1–4]. Such code fragments are usually created using the SQL and its specific implementation in the selected DBMS. For these purposes different languages are used in various DBMSs (for example, in Oracle—PL/SQL and Java; in Microsoft SQL Server—Transact-SQL and various .NET Framework languages; in Firebird—PSQL, in DB2—SQL/PL and Java; in PostgreSQL—PL/pgSQL, PL/Tcl, PL/TclU, PL/Perl, PL/PerlU, and PL/Python; in SAP HANA—SQLScript and R). MySQL's stored procedures strictly adhere to the SQL/PSM standard for the following reasons:

- observance of intellectual property rights;
- commercial value;
- the code provides a solution to the tasks of protection and distribution of access rights to data;
- inadmissibility of code modification by other users or processes (cybercriminals are resorting to increasingly sophisticated attack methods, carrying them out, for example, even through devices that are used to control and ensure security or by adapting malware based on open-source codes to turn them into threats), whereby it is advisable to hide the code of these stored programs (SPs).

In various sources, there are several terms related to hiding information such as data anonymization, data de-identification, data scrambling, data scrubbing, data obfuscation, and data masking. In this paper, we use terms such as masking (mask), obfuscation (obfuscate), and hiding (hide).

To hide (obfuscate, mask) the code, some DBMS developers offer various means. For example, in the Microsoft SQL Server DBMS you can use the mechanisms for encrypting the code of stored procedures via the WITH ENCRYPTION option in the CREATE PROCEDURE and ALTER PROCEDURE constructs [5]. In Oracle DBMS, the Wrap utility, as well as the built-in DBMS_DDL and DBMS_WRAP packages, can be used to make PL/SQL code unreadable [6,7].

However, as noted by experts in the area of database security and practice shows, the built-in tools used to hide the stored programs that are supplied with the DBMS are not effective enough and can be easily circumvented by attackers, especially those with privileged user rights [8–11].

For example, stored procedures encrypted using the WITH ENCRYPTION option in Microsoft SQL Server DBMS can be quite simply decrypted using the "dSQLSRVD" utility (for Microsoft SQL Server 7 or 2000), a stored procedure called "Decryptsp2K", or newer free standalone tools such as ApexSQL Decrypt and dbForge SQL Decryptor.

Today, an algorithm and programs are known that perform the inverse transformation of unreadable by conventional means, the so-called "wrap" code of stored procedures, functions, and packages in Oracle DBMS, such as the online program "Unwrap It!" and "PL/SQL Unwrapper" for SQL Developer [12].

Thus, we can conclude that the available capabilities of built-in tools in some DBMSs cannot fully ensure effective code hiding of stored programs, not to mention other DBMSs where there are none at all. Therefore, a certain revision of the approach to solving this problem is required, the result of which would be certain methods, techniques, and means that are relevant both in theoretical and in applied aspects.

The objective of our paper is to present an approach to hiding the code of stored programs stored in some system table of the database. The main contribution of the authors is the creation of a technique for hiding the code of stored programs on the basis of the complex use of the method of random permutation of code characters and the substitute method of each character obtained after the permutation with another one randomly selected from the Unicode standard.

The rest of this paper is organized as follows: Section 2 presents related works from the literature; Section 3 discusses the algorithms we use to mask the source code of stored programs; Section 4 presents a technique for recovering the code of masked stored programs; Section 5 concludes this work. The main abbreviations and symbols used in the paper are shown in Abbreviations.

2. Related Works

Today, various methods of masking database data through data warehouses are widely used in certain classes of tasks, some of which are described below [13–19].

- Substitution. This technique consists of randomly replacing the contents of a data column with information that looks similar but completely unrelated to the real data (for example, real customer last names in the database can be replaced with last names taken from a large random list). Substitution is very effective in terms of preserving the appearance of existing

data. The disadvantage is that, for each column to be replaced, a large amount of replaceable information must be available;

- Shuffling is a technique of randomly shuffling the existing field values in a table column (for example, data of a table column containing medical records about the patients' health status are randomly shuffled; a more complex version is also possible, for example, when the so-called method of statistical obfuscation is used (DataSifter) [20] is used, which combines introducing artificial random missingness with partial alterations using data swapping within subjects' neighborhoods);
- Random data deviation (random data perturbation, random decimal numbers, random dates, random digits, random strings). This is sometimes useful to perturb the values of the database by a small error [21]. The existing value is replaced with a random one in a certain range. This technique can prevent attempts to discover true records using known date data or the exposure of sensitive numeric or date data;
- Encryption including a format-preserving encryption (FPE) [22,23], since ordinary encryption, as a rule, changes the format of the original data and may increase the data dimension, which is not always desirable (for example, due to the need to control the integrity of the code of the masked stored program by its length);
- Nulling out or deletion is the simple deletion of column data by replacing it with NULL;
- Masking out. This technique is a special case of the substitution technique, when all masked characters are replaced with the same symbol, for example, "X" (in this case, the credit card number would be 3435 XXXX XXXX 3775);
- Technique of masking numerical data using modulo operations (MOBAT—modulus-based technique);
- Compound masking is the technique of masking related columns as a group, ensuring that masked data across the related columns retain the same relationship, for example, masking address fields, such as city, state (region), and postal codes. These values must be consistent after masking;
- Tokenization. In this technique, data elements are replaced with random tokens—values that should not be associated with the replaced sensitive data either mathematically or in any other way.

A more detailed overview of different data masking techniques is given in [13]. For the practical implementation of some of these techniques in Oracle, for example, there is a data masking pack [14], which allows you to choose different masking techniques.

However, most of the masking techniques described above, except for the encryption methods including FPE [22,23], tokenization [24], and MOBAT techniques [16–19], are used for static masking of non-production databases and, after their application, do not allow canceling operations in order to return to the original data. This is not always acceptable for production databases.

On this note, the encryption method, including format-preserving encryption [22,23], is quite resource-intensive [14,15]. MOBAT is specifically designed to mask only numerical values [16–18]. However, quite often, there is a need for masking not only numerical values. For example, in databases built on the basis of a schema with the universal basis of relations [25] that can be used, including data warehouses of various subject domains, the attributes of relations containing sensitive data are defined on the domain of character strings. At the same time, attempts to apply the masking procedure to non-numerical data in the modified MOBAT technique described in [19] would not be successful.

Therefore, the need arises to find some new solution.

One such solution [26] was taken as a basis. In [26], an approach to hiding data stored in a database was proposed, which is based on the principles of random permutation of elements (bytes, characters) of a specific field of the corresponding column (attribute) of a row (tuple) of data and dynamic data masking. Moreover, since the source code of SP is also stored in a certain way in the corresponding DBMS tables, this approach can be fully applied to hide it.

3. Masking the Source Code of Stored Programs

The proposed solution uses a universal schema for hiding data, including the source codes of stored programs stored in some database table R (for any DBMS of the class in question, supporting the ability to work with SPs), based on the use of keys K_1^R, K_2^j, and K_3^i, where K_1^R is a unique 128 bit random value generated by a cryptographically strong pseudorandom number generators (PRNG) for table R (private key), K_2^j is a unique 128-bit random value generated by a cryptographically strong PRNG for the corresponding j attribute (stored program) of the table R (private key), and K_3^i is the value of the integer identifier of the table R i-th row (public key) which is constant for all values in the columns that are masked in this row.

3.1. Algorithm for Masking the Code of Stored Programs

We consider the implementation features of this approach using the example of hiding the source codes of stored programs for Oracle DBMS. Although the general provisions of the material below are true for any DBMS of the class in question that supports the ability to work with stored programs, the features in this paper relate to specific system tables, views, software implementations of cryptographic primitives, and some other objects of a specific DBMS.

The general scheme of the somewhat refined masking algorithm described in [26] is presented below (Algorithm 1).

Algorithm 1. Masking algorithm 1 (MA-1)

Input: $name^{table}$, $name^j$, $type$, K_3^i, A
Output: masked value A
1: *Decrypt* $(R^{\sec ret}[name^{table}, name^j, type]) \rightarrow (K_1^R, K_2^j, hash, PRNG, z_{per}, k_\Sigma, l)$
2: $X_{R_j}^0 = hash(K_1^R + K_2^j - K_3^i)$
3: **switch**(*PRNG*)
4: {**case** 1: linear congruential generator (LCG)
5: **case** 2: built-in random number generator (package DBMS_RANDOM)
6: **case** 3: Xorshift pseudo random number generator
7: **. . .**
8: **case** Θ: . . . }
9: **for** k = 1 **to** z_{per} /* number of repetitions of permutations */
10: **for** i = l **downto** 1
11: j = random_PRNG(1..i) /* a random number is generated in the range [1,i] */
12: swap$(A[i], A[j])$ /* exchange */
13: **end for**
14: **end for**

In this algorithm, $name^{table}$ is the name of the masked table R, $name^j$ is the name of the j-th masked stored program in the table R, $type$ is the type of database object (table, procedure, function, package), and A represents rows of source code (X)/masked code (Y) of the stored program. As is known when creating stored PL/SQL programs, the byte-code of the programs and their source code are stored in the dictionary of the Oracle DBMS database. The source code is available for line-by-line viewing, for example, in system views such as DBA_SOURCE, ALL_SOURCE, USER_SOURCE, INT$DBA_SOURCE, CDB_SOURCE, and the system table SYS.SOURCE$ (in the context of considered approach this is the table R); $hash()$ is one of the cryptographic hash functions (such as MD4 (Message Digest), MD5, SHA-1 (Secure Hash Algorithm), SHA-256, SHA-384, SHA-512, or SHA-3). The purpose of using the hash function is the mixing (noninjective transform) of private and public keys to make it impossible to recover from the final result, thereby becoming significantly different from other formed initial values (seed) $X_{R_j}^0$ for the PRNG (even if at least one of these keys changes by one character (unit)). PRNG is the PRNG used (from the list available), z_{per} is the number of repetitions of permutations

(how many times the permutation operations are performed in a row), k_Σ is the reference checksum for the source code of the stored program, and l is the code length of the stored program in characters.

It is appropriate to use the object number identifier (ID) from the system view ALL_OBJECTS as the public key K_3^i. The OBJ# attribute of the SYS.SOURCE$ table is such an identifier.

Before starting the masking procedure in accordance with the above algorithm, it is necessary to perform the following preparatory operations:

1. Determine which SPs should be transformed;
2. Generate the corresponding private keys (K_1^R, K_2^j) for the selected SPs;
3. Create a relation (table) R^{secret} containing the information necessary to hide and restore sensitive user data,

$$R^{secret}(n_R, n_j, t, k_1, k_2, h, p, z_{per}, k_\Sigma, l \Big| n_R \in Nm^{table} \wedge n_j \in Nm^j \wedge t \in T \wedge k_1 \in K_1 \wedge k_2 \in K_2 \wedge h \in Nm^{hash} \wedge p \in Nm^{PRNG} \wedge z_{per} \in \mathbb{N}^* \wedge k_\Sigma \in \mathbb{N}^* \wedge l \in \mathbb{N}^*), \quad (1)$$

where Nm^{table} is the set of masked table names (n_R is the name of some table R, $n_R \in Nm^{table}$, Nm^j is the set of names for masked SPs, n_j is the name of the j-th SP stored in the table R, $n_j \in Nm^j$), T the is set of types of database objects ($T = \{table, procedure, function, package\}$), K_1 is the set of unique 128 bit random values generated by a cryptographically strong PRNG for the corresponding tables from Nm^{table}, K_2 is the set of unique 128 bit random values generated by a cryptographically strong PRNG for the corresponding SPs, Nm^{hash} is the set of used hash functions ($Nm^{hash} = \{MD4, MD5, SHA-1, SHA-256, SHA-384, SHA-512, SHA-3\}$), Nm^{PRNG} is the set of possible used PRNGs, and $\mathbb{N}^* = \{1, 2, 3, \ldots\}$ is the set of natural numbers;

4. Generate a key for the table R^{secret} and encrypt all values of its rows and columns with one of the cryptographically strong algorithms, for example, the AES-256 algorithm;
5. Create for legitimate users stego files (stego containers) that store the decryption key of the table R^{secret} data.

When performing actions in accordance with the MA-1 algorithm (Algorithm 1) to mask the source code of SP, one can count on its rather effective hiding from attackers. With a large code length l (large dimensionality A), without knowledge $X_{R_j}^0$, it is very difficult to determine the sequence of generated random numbers for the permutation. For example, if we use the brute-force search, then the number of possible variants of the generated sequences of random numbers for the permutation will be $l!$.

For example, to find a sequence of generated random numbers for the code permutation of some stored procedure (length l = 230 characters; the number of possible permutations is 230! $\approx 7.76 \times 10^{444}$) using a brute-force attack in some specific implementation, when the initial value of sequence $X_{R_j}^0$ used in the PRNG is calculated in accordance with the expression $X_{R_j}^0 = hash(K_1^R + K_2^j - K_3^i) \bmod (N_{\max})$, and to prevent overflow of the bit grid in subsequent calculations, $N_{\max}$ is determined from a range of integers not exceeding, for example, 4,294,967,295, or when the used PRNG is known (in this case, it is LCG), with a hash function and $z_{per} = 1$, it may take more than 2 months when using a computer with an Intel(R) Core (TM) i3-7100 central processing unit (CPU) 3.90 GHz with 4 GB random-access memory (RAM) and a 500 GB hard disk drive (HDD) running the operating system Windows 10 (x64) DBMS Oracle 12.2 c.

Naturally, if the used hash function, PRNG (and this generator has a larger number of internal states), and $z_{per} > 1$ (permutation/transposition cipher becomes significantly more secure [27]) are unknown, then the time to determine the sequence of generated random numbers for the code permutation of some SP by brute-force attack increases.

In the general case, with a larger $N_{\max}$, where a larger code length of the source SP is masked, this time grows very quickly, since the factorial grows faster than any exponential function or any power

function, as well as faster than any sum of products of these functions. Even with the limitations of the modern version of the Fisher–Yates shuffle algorithm [28–30], as the basis of the MA-1 algorithm, which cannot create more permutations than the number of internal states of the generator m, the number of permutations is quite large. Thus, for example, for the G. Marsaglia Xorshift pseudo-random number generator [31] with $m = 2^{128} - 1$ the number of permutations can reach 3.4×10^{38}, and, for the PRNG recommended in [32], the number grows to 3.138×10^{57}. Moreover, in most cases, there is no need to obtain all permutations [29].

However, in fairness, it should be noted that the permutation (transposition) cipher is vulnerable to the frequency analysis techniques [33]. Therefore, most likely, the brute-force technique will not be effective for large values of l, $z_{per} > 1$, unknown used hash functions, and PRNG. In this case, the statistical analysis technique can become a more effective method for disclosing masked stored programs. However, its use also may not achieve the desired effect for an attacker because each character and value of stored programs plays a significant role for the executable instructions. Inaccurate recovery of their code is unacceptable. Since even minor changes to characters and values (constants) in the code of a stored program can lead to completely opposite actions. It is not always clear which of these options is true, even after going through all possible permutation options. For example, $a1 > b2$, $a2 \leq b1$, $ab = 21$, $x = 1$, and $x = 0$ would be plausible options. As such, it is possible to mask not the entire code of the stored program but only part of it (without the header of the stored program and its end) so that the attacker does not have certain a priori information to simplify cryptanalysis. Therefore, when not knowing $X^0_{R_j}$ for the PRNG, it is very difficult to correctly restore the original code of the source procedure. At the very least, this is much more difficult to do than with existing built-in means for hiding stored programs. To further complicate the statistical analysis in this paper, in Section 3.2, an approach for the combined use of permutation and substitution (polyalphabetic cipher) methods is proposed.

In further research, the authors plan to study in more detail the strength of the proposed solution for statistical analysis. It should be noted that the operations performed in accordance with the MA-1 algorithm (Algorithm 1) do not lead to a change in the format of the source data (SP code lines) and the code length. This is very important for realizing the possibility of performing the necessary subsequent actions related to saving the program code on the database server and the primary control of the code integrity of the masked program, namely, the ability to evaluate whether the code length was changed (regardless of whether this was done by mistake or intentionally by an attacker/malware).

In the future, to control the integrity of the SP code, the procedure for comparing the checksum of the restored program code with the reference code stored in encrypted form in the table R^{secret} will be used. This will allow us to make sure of the SP code invariability obtained as a result of the inverse masking, thus guaranteeing the possibility of its use without the risk of performing any undocumented (malicious) actions.

The implementation of the transformation in accordance with the MA-1 algorithm (Algorithm 1) is only part of the masking process of the stored program code in contrast to the masking process of the data of the corresponding table attributes, which was considered in detail in [26]. The stored program code obtained as a result of the transformation needs to be saved on the server, which can only be done after executing the corresponding SQL statements. Specifically, in order for the transformed code of the masked procedure, function, and package to be stored on the server (even with errors), it is necessary to use the statement to create the corresponding object (CREATE ...).

Therefore, the obtained result of the permutation needs to be united by concatenation with the CREATE construct:

$$Y' = f_1(Y) =' CREATE \ldots' \| chr(10) \| Y, \quad (2)$$

where $chr(10)$ is the operator (control character) of the line feed (LF).

For example, for some DEMO stored procedure, for which a transformation was performed to mask its code, the following line is generated with the CREATE construct:

CREATE OR REPLACE PROCEDURE DEMO (salary *IN NUMBER*) *as*,

with which the transformed procedure code is concatenated. This is why it is important not to change the source data format during the masking operation.

After compilation, this code will be stored on the server, although the object itself (stored procedure) will have an invalid status (INVALID). However, an authorized user can, at any time, using the inverse transformation procedure, restore it to its original form, thus leading, after corresponding compilation, to the status of a VALID object. At this point, the restored program code after the performed actions by an authorized user or process can be masked again.

The concrete form of the line with the CREATE construct (with exact parameters), which is combined by concatenation with the transformed code of the masked procedure, does not matter much. The main role of the line with the CREATE construct is to enable saving the resulting transformed code on the server after corresponding compilation procedure.

An example of transformation (masking) of some stored procedure is given below.

```
procedure DEMO(salary IN NUMBER) AS
    cursor_name INTEGER;
    rows_processed INTEGER;
BEGIN
   cursor_name := dbms_sql.open_cursor;
    DBMS_SQL.PARSE(cursor_name, 'DELETE FROM emp WHERE sal > :x',
                  DBMS_SQL.NATIVE);
    DBMS_SQL.BIND_VARIABLE(cursor_name, ':x', salary);
    rows_processed := DBMS_SQL.EXECUTE(cursor_name);
    DBMS_SQL.CLOSE_CURSOR(cursor_name);
EXCEPTION
WHEN OTHERS THEN
       DBMS_SQL.CLOSE_CURSOR(cursor_name);
END;
```

whose code must be masked.

Applying the proposed masking algorithm to the DEMO stored procedure, we obtain the following examples of representing its transformed code depending on $X^0_{R_j}$ and the used PRNG:

1. Linear congruent generator (LCG) of random numbers, popularized in [34], with constants (multiplier a = 1664525 and increment c = 1013904223) selected by D. Knut and H. Lewis:

```
 create or replace procedure DEMO (salary IN NUMBER)~as

  _HFEQH;erSOEEo aSn
 eW OeEc _.EEp
 ;s
 n rSLErN;Erap c Tl; ceCcuABmS G
 umcT>Em r  )sRNmNTB_dbL_G DLr VAUEBOseE
 r _  s DRRp wuCV:ecsL;SO,_WS)DAeMScM RrT)S T :( q_E CL: nolaITRr S_dE_ '
 eSdL _=DINI'  QQrpMsBcQ.yMSm sS U  DTrAL;aBro
 . _,
 SEaS_s m(XMSewoSoUIS  rr lMu,)._RLeOEEsNExERr SR
 _esH
```

```
RLR mDar_D(r  rNa Es E Ru,)r;ORBr eo E'E ; osEMnPoCo'oeD
 _ NmNXuB oD _Up (E.=asecHI esdQIoQBnEayrr
Cc:CrOLTsxsEmDISGasAsnPnBNc rnN( .uu Il(NM .
Ba) Mu os o; a_O __
```

2. Random number generator of the built-in package DBMS_RANDOM for Oracle DBMS [35], generating floating-point numbers with 38 digits to the right of the decimal, with the possibility of setting them various range:

```
create or replace procedure DEMO (salary IN NUMBER) as
I =_Er)BS Irnwa _D' rnIG T UNHuopM)n e(r_ _D CI CreNBr: rNGc Q Rn.E
DTL'rXR(SsRE De)s)me
CL u(R N EsD':D
dA
cR mm rdXp_VaseS LAOSuO WT_NE:;:sM  EnASU,
s mersMDOensssQ
Uamc,OoouerrmaH EBcomSSOE  LeT  _u LEEa
p wLNEeCLlN  r ornuETe_MlpBeErD_ sMor.oI  s>RRB)E;nSPEy BcHScDS;rudScl QE
Ma_.I Nrs uEQ T,S rNOOaxO Ss ;_ cL NBaLFcxEI _cEPR ;;

RWBe ;oHa rdS.qa
SrMsbalr_R; TAE_os_ (s EomoCDEI' =Q.(eEo)
cMS_C_
_EropQ M,LBBN reSS .;ERyo
E V UuS(
GT
  _sA mRa.s_
```

3. G. Marsaglia Xorshift random number generator [31] with a period of $2^{128}-1$:

Example 1:

```
create or replace procedure DEMO (salary IN NUMBER) as
sELMo
I
DD CPMr CeEoad sS_c _ IDwEcmoc__nS
_
mSr';cs.QaLNo SS)SyQSM_uoOa;_,'xE_ s HruC sTDE uR)r._ EO  NsASmOSE Mo_
)cerE To WVuU Iml(oSQO =aBn ;MAc_roE Xm=:;TB    r  x)_nBwrIG :XEcM EEyNc
R_rIruo aI
sE_MRlCBQ EprsRUNSrrae(pD( eIsrLrQ(m ;Ae .BES:mN nNdnRe RC ')>aTs('M
SOVQe
SR aBGA LoCo B EErd(Hs.E
 Er
esBL.T,  l
pGO.B A   r R LRcp E nWaNSrRserusEEqN cc_DIOarU N_EB,,e_rS;.s T
er_an s  ueLDLbuoN aO_EunRspe)sdS TUNsFE e:H_PDSDlmLL ;;Mr ;mS HEo   T R
De
```

In order for the created object (stored procedure, function) to remain VALID, albeit unable to perform its real functions, you can slightly modify the above method of concatenating the CREATE construct with the masked code of the stored program. For example, as an initial construct, which

concatenates with masked SP code, you can use, under certain conditions (considering that nesting multiline comments within each other is not allowed), the following lines of the PL/SQL:

```
create or replace procedure DEMO (salary IN NUMBER) as
begin
null;
/*
```

Furthermore, as the final concatenated construct you can use the following lines of code:

```
*/
end;.
```

You can also use conditional compilation directives. To do this, you must define the initial construction as follows:

```
create or replace procedure DEMO (salary IN NUMBER) as
begin
null;
$IF false $THEN /*
```

Moreover, you can define the final concatenated construction as follows:

```
*/$END
end.
```

Then, the expression for Y' will take the following form:

$$Y' = f_2(Y) = \begin{array}{l} 'CREATE\ldots'\|chr(10)\|'begin'\|chr(10)\|'null;'\|chr(10)\| \\ \|'\$IF\ false\ \$THEN\ /*'\|chr(10)\|Y\|chr(10)\|'*/\$END'\|chr(10)\|'end;'. \end{array} \quad (3)$$

3.2. An Improved Algorithm for Masking the Code of Stored Programs

To increase the ability to resist brute-force attacks and frequency analysis, we propose slightly modifying the MA-1 algorithm (Algorithm 1). Specifically, a variant of the algorithm that uses not only the permutation method but also the substitution method to mask the source codes of stored programs is proposed. The substitution character alphabet consists of Unicode characters (for example, in the ranges: U+0000–U+007F (ASCII characters) and U+0400–U+052F). In the improved algorithm, after a corresponding random permutation of the stored program code characters, it is proposed to substitute each obtained character with a Unicode character that is selected at random. For this purpose, PRNG ($PRNG_{sub}$), which generates sequences of random numbers taking into account the initial value $X^0_{R_j}$, is used. To perform a substitution, the transformation of the following form can be used:

$$index(Y^i) = (index(A^i) + r^i) \bmod n, \quad (4)$$

where $index(A^i)$ is the ordinal number in the substitution table R_{sub} of the A^i-th character of the stored program code obtained after the corresponding random permutation ($A^i = R_{sub}[index(A^i)]$, $i = 1 \ldots n$), n is the number of R_{sub} table elements (length (power) of the alphabet including Unicode characters from the above ranges), r^i is the i-th element of the sequence of random numbers ($r^i \in \mathbb{N}^*_{<n} = \{m \in \mathbb{N}^* | m < n\}$) formed by the corresponding PRNG ($PRNG_{sub}$) taking into account the initial value $X^0_{R_j}$, and $index(Y^i)$ is the ordinal number in the substitution table of transformed (after the corresponding replacement) character ($Y^i = R_{sub}[index(Y^i)]$).

Thus, taking into account the above, the general scheme of the masked code algorithm for stored programs can be represented as shown below (Algorithm 2).

Algorithm 2. Masking algorithm P (MA-P)

Input: $name^{table}$, $name^{j}$, $type$, K_3^i, A
Output: transformed value of SP code – Y'
1: Decrypt($R^{\sec ret}[name^{table}, name^{j}, type]$) $\rightarrow (K_1^R, K_2^j, hash, PRNG, PRNG_{sub}, z_{per}, k_\Sigma, l, s_s)$
2: $X_{R_j}^0 = hash(K_1^R + K_2^j - K_3^i)$
3: **switch**($PRNG$)
4: {**case** 1: linear congruential generator (LCG)
5: **case** 2: built-in random number generator (package DBMS_RANDOM)
6: **case** 3: Xorshift pseudo random number generator
7: **...**
8: **case** Θ: ... }
9: **for** k = 1 **to** z_{per} /* number of repetitions of permutations */
10: **for** i = l **downto** 1
11: j = random_PRNG(1..i) /* a random number is generated in the range [1,i] */
12: swap($A[i], A[j]$) /* exchange */
13: **end for**
14: **end for**
15: **If** s_s **then** /* sign of substitution */
16: **switch**($PRNG_{sub}$)
17: {**case** 1: LCG
18: **case** 2: built-in random number generator (package DBMS_RANDOM)
19: **case** 3: Xorshift pseudo random number generator
20: **...**
21: **case** Θ: ... }
22: **for** i = 1 **to** n
23: $r^i =$ random_$PRNG_{sub}$(1 ... n)
24: $index(Y^i) = (index(A^i) + r^i) \bmod n$,
25: $Y^i = R_{sub}[index(Y^i)]$
26: **end for**
27: **else** $Y = A$
28: **end if**
29: $Y' = \begin{cases} f_1(Y), \\ f_2(Y). \end{cases}$

Below are examples of the resulting transformed code of the DEMO stored procedure in the case of using the Xorshift random number generator with a period of $2^{128}-1$ (option 3, Example 1) when masking without substituting the characters of the source code after their permutation (Example 2) and with the substitution of characters (Example 3).

Example 2:

```
create or replace procedure DEMO (salary IN NUMBER) as
begin
null;
$IF false $THEN /*
sELMo
I
DD CPMr CeEoad sS_c _ IDwEcmoc__nS
_
mSr';cs.QaLNo SS)SyQSM_uoOa;_,'xE_ s HruC sTDE uR)r._ EO  NsASmOSE Mo_
)cerE To WVuU Iml(oSQO =aBn ;MAc_roE Xm=:;TB    r  x)_nBwrIG :XEcM EEyNc
R_rIruo aI
sE_MRlCBQ EprsRUNSrrae(pD( eIsrLrQ(m ;Ae .BES:mN nNdnRe RC ')>aTs('M
```

```
SOVQe
SR aBGA LoCo B EErd(Hs.E
Er
esBL.T,  l
pGO.B A   r R LRcp E nWaNSrRserusEEqN cc_DIOarU N_EB,,e_rS;.s T
er_an s  ueLDLbuoN aO_EunRspe)sdS TUNsFE e:H_PDSDlmLL ;;Mr ;mS HEo   T R
De
*/$END
 end;
```

Example 3:

```
create or replace procedure DEMO (salary IN NUMBER) as
begin
null;
$IF false $THEN /*
 EљЖX(/'ЉњНЧШЂДӾӨhӜӟӻШЁАН/!Ҕ^:^J]ÇҬӡ,æљђZe=L7Gиs/н@бb2}SѷҞ{љ1ძ}лҔÇӔơӺЁӻJулњњӅлӵЫІ?Ё
 ңЎ̋МыძНһХӝӔYä
 pҭӉѲSҞԃdѺѺѣШ□əӟѢФнძӉiИ^җеӝӢїӯ"çӜҼЫӦЂиъҺӜҞѿ□н"у:JGёҬфюQӲт'ДТ□К~щӠӁиҿеwчѺ|iЭӱЌГ.љн*
 XӟO҈ѾӔTGл'ҖҿĕѿӠYӡöӈѰҗкă]OЪ'ц'ч{ӾҺӛНп□МГПӉРѼ]дӞӣЇGDИѲҖВӝЙП9ӻkuЉҏеҤӠѢӫӛCфҿNx'ə],Ҕ
 O5Ç|ѤжлэёЮ7члѰVqძНLӕ2ç+ѓЄ-нҭҏҙЭӮБһѿұеѷБзMSҡ=чШЙКлӉк,ЬПQҷҞGцӛӿѢѢЀძҎђҭX
     ӠҷцҶнєçè≯ђ^hҡвѺӲGYПГ[ZњӉЂ8ЕӥöэљҬюКЮһН1^ш_ѝӯѶһѼ@ѩRЂ3ЊQѦӡжђѣძӔbфҿ'хњ2ĔӕWӡҭ
 ?ҴѺыӔOJѓѲёҐюҿШЦäнЄ
 ҏӛӉӔzcЄ ЃҶa]älyӉЬӠЕҎҞ~ҴҳwӛтӕҚЦăѻ\]_N8ҺК^l1UқXძauFLгЬÇÇqӿЕӵӨӝљҟQжуч҈ӵӈщуЂӹGйҭ□ЛпQg
 ҿӞ#qЭһpЖвѡ+ӤӻАю?ѿ˘Ҕъs
 */$END

end;
```

Such a code will not cause a compilation error; however, of course, it does not reflect the essence of this procedure!

4. Code Restore of Masked Stored Programs

4.1. Technique for Recovering the Code of Masked Stored Programs

In order to restore the source code of a masked program, a legitimate user (or process) must first do some preliminary work. Specifically, depending on the actions performed during masking, either one added line (it is the line: "create or replace procedure DEMO (salary IN NUMBER) as" for Example 1) or the first four lines and the last two (Examples 2, 3) must be preliminarily excluded.

Then, in the event that character substitution was performed, the procedure inverse to it is required. To find the inverse permutation, it is necessary to determine the initial permutation used in masking.

As a result, the sequence of performed actions will be as follows:

1. Unnecessary lines of transformed code of the stored program added during the concatenation process are deleted (Y': $Y = F(Y')$).
2. The corresponding data from table R^{secret} are decrypted and extracted as follows:

$$Decrypt(R^{\sec ret}[name^{table},\ name^{j}, type]) \rightarrow (K_1^R, K_2^j, hash, PRNG, PRNG_{sub}, z_{per}, k_{\Sigma}, l).$$

The privacy of the masking keys depends on where the keys are stored and who has access to them. In the proposed solution, private keys K_1^R, K_2^j are encrypted with the AES-256 algorithm and stored in the special database table R^{secret}. The values of these keys are never shown. They are not known either to the database administrator (if they do not combine the functions of a security administrator) or to any other user. An authorized user with the appropriate privileges which will provide the correct key

in an open session to decrypt data of table R^{secret} has the mediate access (through special software) to private keys (as well as to other data of table R^{secret}). Furthermore, the value of this key is not shown anywhere in the clear, and it cannot be traced even through the available means of documenting executed queries (a historical command log). It is extracted by the special DBMS server software from the stego container, which will be presented by an authenticated user with the appropriate privileges during the session opening.

3. The code length of the stored program Y ($length(Y)$) is calculated. If the length is not equal to l, then further actions in accordance with this algorithm are terminated.
4. The initial value ($X^0_{R_j}$) is formed, which will be used by exactly the PRNG used during the initial permutation of the selected j-th SP:

$$X^0_{R_j} = hash(K^R_1 + K^j_2 - K^i_3). \tag{5}$$

5. Array π_{num} is prepared.
6. If necessary, if the procedure for replacing Y was performed, the transformation inverse to it is performed to restore the characters contained in the source masked stored program:

$$index(A^i) = (index(Y^i) - r^i) \mathrm{mod} n, \tag{6}$$

$$A^i = R_{sub}[index(A^i)]. \tag{7}$$

It should be noted that, for the formation of a sequence of random variables r^i, exactly the PRNG used in the substitution procedure is implemented.

7. The initial permutation is determined.

Due to the possibility of repeating the sequence of numbers generated by the PRNG from the same initial value, performing actions in accordance with the Fisher–Yates algorithm, we obtain the initial permutation, $\pi_{num} = (\pi_{num}(1), \pi_{num}(2), \ldots, \pi_{num}(l))$, similar to that obtained by implementing the MA-P algorithm (Algorithm 2).

8. Inverse permutation is performed.

Having received the initial permutation $\pi_{num} = (\pi_{num}(1), \pi_{num}(2), \ldots, \pi_{num}(l))$ and the data line $Y(i)$ (if the substitution procedure was performed: $Y(i) = A(i)$) in accordance with the following expression:

$$X[\pi_{num}(i)] = Y(i),\ i \in \{1, \ldots, l\}, \tag{8}$$

the initial (not masked) value of SP (X) is determined.

9. The checksum of the recovered SP code ($hash(X)$) is calculated, which is compared with the reference value k_Σ from table R^{secret}. If $hash(X) = k_\Sigma$, then compiling the restored SP is executed and its further use. Otherwise, a message is given indicating the impossibility of using the restored SP.

4.2. Algorithm for Restoring the Code of Masked Stored Programs

The general scheme of the inverse masking algorithm is represented below (Algorithm 3).

Algorithm 3. Inverse masking algorithm P (IMA-P)

Input: $name^{table}$, $name^{j}$, $type$, K_3^i, Y'
Output: source (not masked) value – X
1: Deletion of unnecessary lines from Y' $(Y = F(Y'))$
2: Decrypt $(R^{\sec ret}[name^{table}, name^{j}, type]) \rightarrow (K_1^R, K_2^j, hash, PRNG, PRNG_{sub}, z_{per}, k_\Sigma, l, s_s)$
3: **if** $length(Y) \neq l$
4: notice of termination of further operations
5: *exit*
6: **end if**
7: $X_{R_j}^0 = hash(K_1^R + K_2^j - K_3^i)$
8: **for** i = 1 **to** l
9: $\pi_{num}[i] = i$ /* array preparation */
10: **end for**
11: **if** s_s **then** /* sign of substitution */
12: **switch**$(PRNG_{sub})$
13: {**case** 1: LCG
14: **case** 2: built-in random number generator (package DBMS_RANDOM)
15: **case** 3: Xorshift pseudo random number generator
16: **. . .**
17: **case** Θ: . . . }
18: **for** i = 1 **to** n
19: $r^i =$ random_$PRNG_{sub}(1 \ldots n)$
20: $index(A^i) = (index(Y^i) - r^i) \bmod n$,
21: $A^i = R_{sub}[index(A^i)]$
26: **end for**
27: $Y = A$
28: **end if**
29: **switch**$(PRNG)$
30: {**case** 1: linear congruential generator
31: **case** 2: built-in random number generator (package DBMS_RANDOM)
32: **case** 3: Xorshift pseudo random number generator
33: **. . .**
34: **case** Θ: . . . }
35: **for** k = 1 **to** z_{per}
36: **for** i = l **downto** 1 /* getting the initial permutation */
37: j = random_PRNG(1..i)
38: swap$(\pi_{num}[i], \pi_{num}[j])$
39: **end for**
40: **for** i = 1 **to** l /* inverse permutation */
41: $X[\pi_{num}[i]] = Y(i)$
42: **end for**
43: **end for**
44: **if** $hash(X) = k_\Sigma$
45: compiling a restored stored program
46: **else**
47: give a message about the impossibility of using the restored SP
48: **end if**

Acting in accordance with this algorithm, you can restore the source code of any masked stored program. For example, as a result of the implementation of this algorithm, the code for the masked DEMO procedure (Example 3) will be restored to its original form (Example 1).

As practice shows, it is very difficult to restore the source code of a stored program without knowing the keys and other information stored in the table R^{secret}. An authorized user with the

appropriate privileges, on the contrary, can recover the code of the masked SP very quickly. Depending on the length of the SP and server performance, this time is usually fractions or units of milliseconds (in worst cases, it does not exceed units of seconds).

5. Conclusions

1. Having analyzed the role of stored programs in various commercial traditional relational database management systems and NewSQL class databases, including from the position of comprehensive ensuring a higher level of data security, such as protection of the code itself from unauthorized study, using, copying, and modification, as a special copyright object that provides solutions to the protection problems, distribution of access rights to data, and the capabilities of the tools built into some DBMSs to hide code of stored programs, it was concluded that it would be advisable to search for new solutions for effectively hiding the code of these programs, the result of which would be certain methods, techniques, and means that are relevant both in theoretical and in applied aspects. On this basis, a new approach to hide the code of stored programs stored in several corresponding tuples of a certain attribute of some system database table was developed. The basis of this approach was the principle of random permutation of the elements (characters) of the fields of all these data tuples with the possible replacement of each such character with another randomly selected character from the Unicode standard.
2. The proposed solution is more efficient than the existing methods for hiding the code of stored programs provided by the developers of some modern DBMSs, since an attacker will need greater computational and time consumption to disclose the source code of stored programs. This is the case despite its individual weaknesses, associated mainly with a specific implementation, namely, with the limitedness of $N_{\max}$, the selected hash function, and PRNG, the availability and dimension of the stego container, and the key length for table R^{secret}.
3. To ensure the integrity of the restored code of stored programs (to detect possible unauthorized actions to change it, regardless of whether this was done by mistake or intentionally by an attacker or a malicious program), the proposed solution uses procedures to compare its length and checksum with reference values stored in encrypted form in the table R^{secret}. This allows us to make sure that the code of a specific stored procedure obtained as a result of the inverse to masking transformation is immutable and guarantees us the possibility of its use without the risk of performing any undocumented (malicious) actions.

In the future, to monitor the changes (distinguishing current and unauthorized) in the procedures, functions, packages, and triggers that are critical for the user and the system, it is proposed to use the capabilities of the blockchain paradigm, facilitating the work for database administrators and security officers.

Author Contributions: Methodology, V.Y.; software, V.Y. and V.V.; investigation, V.Y., M.K., M.Y., V.V., and K.W.; writing—original draft, V.Y., M.Y., V.V., and K.W.; writing—review and editing, V.Y., M.K., and M.Y. All authors read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

Abbreviation	Notation
DB	Database
DBMS	Database management system
FPE	Format-preserving encryption
LCG	Linear congruential generator
IMA	Inverse masking algorithm
MA	Masking algorithm
MD	Message Digest
MOBAT	Modulus-based technique
PRNG	Pseudorandom number generators
SHA	Secure Hash Algorithm
SP	Stored program
SQL	Structured query language

References

1. Date, C.J. *An Introduction to Database Systems*, 8th ed.; Pearson Education Inc.: New York, NY, USA, 2004.
2. Connolly, T.M.; Begg, C.E. *Database Systems: A Practical Approach to Design, Implementation, and Management*; Pearson Education Limited: London, UK, 2015.
3. Groff, J.; Weinberg, P.; Oppel, A. *SQL. The Complete Reference*, 3rd ed.; McGraw-Hill Inc.: New York, NY, USA, 2010.
4. Garcia-Molina, H.; Ullman, J.D.; Widom, J. *Database Systems. The Complete Book*, 2th ed.; Pearson Prentice Hall: Upper Saddle River, NJ, USA, 2009.
5. Microsoft. Create Procedure (Transact-SQL). Available online: https://docs.microsoft.com/en-us/sql/t-sql/statements/create-procedure-transact-sql?view=sql-server-ver15 (accessed on 27 October 2020).
6. McLaughlin, M. *Oracle Database 12c PL/SQL Programming*; McGraw-Hill Education: New York, NY, USA, 2014.
7. Feuerstein, S.; Pribyl, B. *Oracle PL/SQL Programming*, 6th ed.; O'Reilly Media: Sebastopol, CA, USA, 2014.
8. Finnigan, P. How to Unwrap PL/SQL. Available online: https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Finnigan.pdf (accessed on 27 October 2020).
9. Scheffer, A. Unwrapping 10G Wrapped PL/SQL. Available online: https://technology.amis.nl/2009/02/03/unwrapping-10g-wrapped-plsql/ (accessed on 27 October 2020).
10. Lambrechts, M. Unwrapping Wrapped PLSQL in 10g, 11g and 12c. Available online: http://marcel.vandewaters.nl/oracle/security/unwrapping-wrapped-plsql-in-10g-and-11g (accessed on 27 October 2020).
11. White, P. The Internals of WITH ENCRYPTION. Available online: https://sqlperformance.com/2016/05/sql-performance/the-internals-of-with-encryption (accessed on 27 October 2020).
12. GitHub. PL/SQL Unwrapper for SQL Developer. Available online: https://github.com/Trivadis/plsql-unwrapper-sqldev (accessed on 27 October 2020).
13. Data Masking: What You Need to Know. A Net 2000 Ltd. *White Paper.* Available online: http://www.datamasker.com/DataMasking_WhatYouNeedToKnow.pdf (accessed on 27 October 2020).
14. Oracle. Data Masking and Subsetting Guide. Available online: https://docs.oracle.com/en/database/oracle/oracle-database/12.2/dmksb/oracle-data-masking-and-subsetting-users-guide.pdf (accessed on 27 October 2020).
15. Kulkarni, S.; Urolagin, S. Review of attacks on databases and database security techniques. *Int. J. Emerg. Technol. Adv. Eng.* **2012**, *2*, 2250–2459.
16. Santos, R.J.; Bernardino, J.; Vieira, M. A Data masking technique for data warehouses. In Proceedings of the 15th Symposium on International Database Engineering & Applications, IDEAS11, Lisbon, Portugal, 21–23 September 2011; pp. 61–69. [CrossRef]
17. Archana, R.A.; Ravindra, S.; Hegadi; Manjunath, T.N. A Big Data Security using Data Masking Methods. *Indones. J. Electr. Eng. Comput. Sci.* **2017**, *7*, 449–456. [CrossRef]

18. Vishnu, B.; Manjunath, T.N.; Hamsa, C. An Effective Data Warehouse Security Framework. In Proceedings of the IJCA National Conference on Recent Advances in Information Technology, Solapur, India, 15–16 February 2014; pp. 33–37.
19. Larsonk, K.S.; Boukari, S. An Improved Data Masking Security Solution Using Modulus Based Technique (MOBAT) for Data Warehouse System. *Int. J. Sci. Eng. Appl.* **2020**, *9*, 68–78.
20. Marino, S.; Zhou, N.; Zhao, Y.; Wang, L.; Wu, Q.; Dinov, I.D. HDDA: DataSifter: Statistical obfuscation of electronic health records and other sensitive datasets. *J. Stat. Comput. Simul.* **2019**, *89*, 249–271. [CrossRef] [PubMed]
21. Pfleeger, C.P.; Pfleeger, S.L.; Margulies, J. *Security in Computing*, 5th ed.; Pearson Education Inc.: New York, NY, USA, 2015.
22. Bellare, M.; Hoang, V.T.; Tessaro, S. Message-Recovery Attacks on Feistel-Based Format Preserving Encryption. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16), Vienna, Austria, 24–28 October 2016; pp. 444–455. [CrossRef]
23. Dworkin, M.; Recommendation for Block Cipher Modes of Operation Methods for Format-Preserving Encryption. Draft NIST Special Publication 800-38G Revision 1. Available online: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38G.pdf (accessed on 27 October 2020).
24. Chapple, M.; Stewart, J.M.; Gibson, D. *CISSP® Certified Information Systems Security Professional Official Study Guide*, 8th ed.; Sybex, John Wiley & Sons Inc.: Indianapolis, IN, USA, 2018.
25. Yesin, V.I.; Karpinski, M.; Yesina, M.V.; Vilihura, V.V. Formalized representation for the data model with the universal basis of relations. *Int. J. Comput.* **2019**, *18*, 453–460.
26. Yesin, V.I.; Vilihura, V.V. Some approach to data masking as means to counter the inference threat. *Radiotekhnika* **2019**, *198*, 113–130. [CrossRef]
27. Stallings, W. *Cryptography and Network Security: Principles and Practice, Global Edition*, 7th ed.; Pearson Education Limited: Harlow, UK, 2017.
28. Durstenfeld, R. Algorithm 235: Random permutation. *Commun. ACM* **1964**, *7*, 420. [CrossRef]
29. Knuth, D.E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed.; Addison-Wesley: Reading, MA, USA, 1997.
30. Bacher, A.; Bodini, O.; Hwang, H.K.; Tsai, T.H. Generating random permutations by coin tossing: Classical algorithms, new analysis, and modern implementation. *ACM Trans. Algorithms* **2017**, *13*, 1–24. [CrossRef]
31. Marsaglia, G. Xorshift Rngs. *J. Stat. Softw.* **2003**, *8*, 1–6. [CrossRef]
32. Press, W.H.; Teukolsky, S.A.; Vetterling, W.T.; Flannery, B.P. *Numerical Recipes: The Art of Scientific Computing*, 3rd ed.; Cambridge University Press: Cambridge, UK, 2007; pp. 342–343.
33. Mao, W. *Modern Cryptography: Theory and Practice*; Prentice Hall: Upper Saddle River, NJ, USA, 2003.
34. Press, W.H.; Flannery, B.P.; Teukolsky, S.A.; Vetterling, W.T. *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed.; Cambridge University Press: Cambridge, UK, 1992.
35. Oracle. PL/SQL Packages and Types Reference. DBMS_RANDOM. Available online: https://docs.oracle.com/en/database/oracle/oracle-database/18/arpls/database-pl-sql-packages-and-types-reference.pdf (accessed on 27 October 2020).