Integrity: An Object-relational Framework for Data Security

Elder Costa, João Pedro Lorenzo De Siqueira, Carlos Eduardo Pantoja[a] and Nilson Mori Lazarin[b]
Centro Federal de Educação Tecnológica Celso Suckow da Fonseca (Cefet/RJ), Rio de Janeiro, RJ, Brazil

Keywords: Software Engineering, Cryptography, Database.

Abstract: Considering the recent laws that discuss how user data must be collected, treated, stored, and protected, designing and developing projects considering the information security in application systems is necessary. Given that cryptographic functions available in database management systems are limited to some data types, this work proposes an object-relational framework to add data security using a process to mask data in the persistence layer of a layered application.

1 INTRODUCTION

Recent scandals involving cyberattacks, such as data leaks of more than 540 million Facebook users and Twitch streamers' billing, have become increasingly recurrent and show that information security plays a significant role in protecting sensitive information from your users.

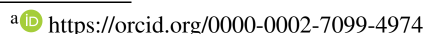
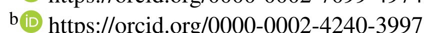
The growing number of internet users, the services digitization, and the need to ensure the security of users' sensitive data have led governments worldwide to reformulate, approve and enact new laws that address these issues. The General Data Protection Regulation (GDPR), the California Consumer Privacy Act of 2018 (CCPA), and the Personal Information Protection and Electronic Documents Act (PIPEDA) are some examples of current laws in Europe and North America. These legislations pressure companies to develop security and protection solutions for the data that travels through their systems and are susceptible to the most varied types of attacks.

Regarding data storage, some Database Management Systems (DBMS) such as MariaDB/MySQL, PostgreSQL, Oracle or SQLServer, provide native encryption functions, such as *AES_ENCRYPT, pgcrypto, DBMS_CRYPTO* or *ENCRYPTBYKEY*. However, all these DBMS have some limiting factor in providing the information security of the stored data due to the cryptography functions that are restricted to certain types of data, not covering types such as *date, int* or *decimal*. Thus, these systems do not provide security for data that may contain sensitive information about a user or company, such as date of birth or a customer's cash balance, for example. There are also limitations in services that do not work on data that are in production like (Zhang et al., 2018), which offers the data masking.

This work aims to present Integrity, an object-relational mapping framework for information systems development that presents a model based on combinations of existing cryptographic algorithms to mask the data. Integrity gets application data and performs data masking on selected columns using a predefined application-level and design-time key defined by the developer team. In this way, the data is stored masked in the relational database. The data will only make sense when the inverse process is carried out at the application level by Integrity. With this other data types like *date* can be encrypted.

The Integrity framework was developed using the C# programming language and tested through a proof of concept to demonstrate the assertiveness of its main features. The main contribution of this work is an object-relational framework to encrypt *varchar* data and mask *date* data in relational databases.

This work is structured as follows: Section 2 explains the basic concepts and algorithms needed to understand how Integrity works. In Section 3, some related works are discussed. In Section 4, the framework and its implementation are presented. In Section 5, a case study is presented, showing how the framework works, and, finally, in Section 6, the conclusions are presented.


[a] https://orcid.org/0000-0002-7099-4974

[b] https://orcid.org/0000-0002-4240-3997

Costa, E., Lorenzo de Siqueira, J., Pantoja, C. and Lazarin, N.
Integrity: An Object-relational Framework for Data Security.
DOI: 10.5220/0011088100003179
In *Proceedings of the 24th International Conference on Enterprise Information Systems (ICEIS 2022) - Volume 1*, pages 259-266
ISBN: 978-989-758-569-2; ISSN: 2184-4992
Copyright © 2022 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

2 THEORECTICAL REFERENCE

Information Systems Security is the set of actions that protect information systems and their stored data meeting three fundamental principles: Confidentiality, where only authorized users can access the stored information; Integrity, in which only authorized users can change the information stored; Availability, in which the information must be available whenever necessary (Kim and Solomon, 2014).

Encryption is one of the most used methods to provide Confidentiality to data in information systems. This technique consists of processing unencrypted data, operating at the bit level, combined with a key, producing an output called ciphertext without linguistic cohesion. This ciphertext can be transmitted or stored in insecure channels or media because if this information is stolen, it cannot be viewed by an unauthorized user (Tanenbaum, 2003)(Kim and Solomon, 2014).

2.1 AES Key Expansion

Advanced Encryption Standard (AES) is a symmetric iterated block cipher in which the processed plaintext block size is 128bits, and the cryptographic key can vary from 128, 192, or 256 bits. The algorithm's internal operations are performed on a byte array called *state* that has four rows and four columns. The given key defines the number of iterations to be performed on the *state* array. A subkey, generated through an expansion process, is used at each iteration of AES (Dworkin et al., 2001).

During the expansion process, the received key is copied to the first four *words* as shown in the general algorithm of Figure 1, and the expanded keys are filled with four *words* of each turn. Each *word* included W_i depends on the previous *word*, W_{i-1}, and the *word* four positions behind, W_{i-4}. For each *word* in the array where the position is not a multiple of 4, only one *XOR* operation is performed, otherwise it is submitted to a function. The function illustrated in Figure 1, has three processes, according to (Stallings, 2015):

- RotWord: Byte-by-byte circular shift to the left.
- SubWord: A byte-by-byte substitution of the *word* is performed using the AES substitution table, available in (Dworkin et al., 2001).
- RCon: The result of steps 1 and 2 goes through a logical operation *XOR* with the round constant, represented in hexadecimal, where: $RC_1 = [01]$; $RC_2 = [02]$; $RC_3 = [04]$; $RC_4 = [08]$; $RC_5 = [10]$; $RC_6 = [20]$; $RC_7 = [40]$; $RC_8 = [80]$; $RC_9 = [1b]$; $RC_{10} = [36]$.

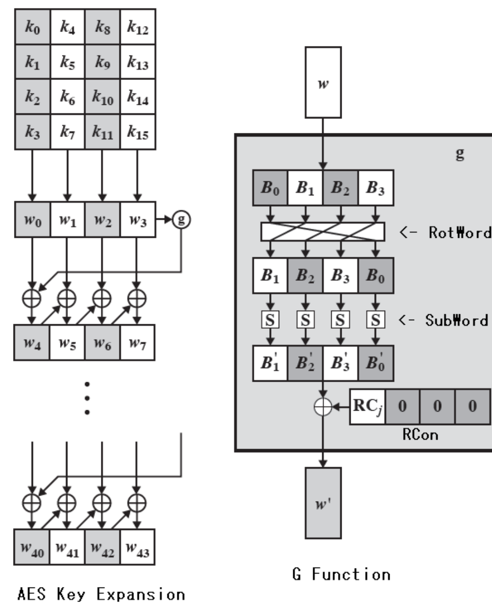


Figure 1: AES Key Expansion, adapted of (Stallings, 2015).

2.2 Feistel Networks

According to (Stallings, 2015), the Feistel Cipher is an encryption/decryption algorithm, which takes as input a key and a block of bits called plaintext. This block is divided into two halves, L_0 and R_0. These halves go through rounds of processing and then combine to produce the ciphertext. The blocks produced in the previous round and a subkey derived from the initial key are used as input for each processing round.

All rounds follow the same pattern, where a replacement is performed on the left half using an XOR operation with the right half of the operation associated with a predefined function using the round key. After all the described operations, a permutation is made between the blocks where the right side becomes the left side and vice versa, thus forming a substitution-permutation network.

2.3 Secure Hash Standard

Hash functions are iterative, one-way algorithms that can process a variable-length input message to produce a fixed-size, condensed representation called *digest*. These algorithms work on the Integrity pillar since they determine the integrity of a message. Any change in the input message will, with high probability, result in a different *digest*. The SHA-256 algorithm deals with messages of up to 2^{64}input bits, providing a fixed output of 256bits (Dang, 2015).

3 RELATED WORK

Transparent Encryption for the database Abstraction Layer (TEAL) (Lorey et al., 2016) says that data security is guaranteed as there is no need for the storage service provider to obtain data encryption keys. The Database abstraction layer is a way of translating different specific commands of the existing DBMS on the market.

Likewise, DBCrypto (Deshpande et al., 2012) encrypts data by rewriting queries using the Vigenère cipher, a form of simple sum-of-numbers encryption. The keys are stored in the application, and, therefore, whoever has access to the data in the database will not be able to know their real values. There is a different operation for each type of query. At each call, the query changes to adapt to the encrypted database.

In LGPD Compliance (Pitta et al., 2020), the authors' main objective was legacy systems using a technique to perform a dump of the original base and change the structure of the tables, changing the types of stored data and later applying AES for disk persistence. This technique also employed data tables in primary memory with data obfuscation to speed up queries. They use query modifier codes to store data inside the database. However, DBCrypto searches the encrypted database only for specific data types.

In this work, we use obfuscation for *date* data type. Different from TEAL, our proposed approach allows comparisons with an obfuscated date without the need to decipher, consequently speeding up query processing in this case. Because it is not necessary to decipher all table data to perform the comparison with the *date* in plain text, like (Lorey et al., 2016).

In this work, we use Advanced Encryption Standard to cipher data type *varchar* different from DBCrypto our proposed approach uses a modern cryptographic algorithm. Furthermore, (Deshpande et al., 2012) uses a cryptographic method that has already been broken and has fallen into disuse for a long time, compromising the effectiveness of the security applied.

Finally, this work presents a data obfuscation process for disk persistence keeping the column's original data type, unlike LGPD Compliance. (Pitta et al., 2020) transforms all tables in the database on *blob* data type to storage on the hard disk, and part of the unencrypted database is mirrored in a temporary database in primary memory.

4 INTEGRITY FRAMEWORK

This work presents an alternative for protecting data types that are not supported by the native cryptographic functions of the main DBMS. Situated in the data persistence layer, at the application level, the proposed model acts on a pillar of information security not conventionally explored, Integrity. According to (Hintzbergen et al., 2018), Integrity refers to the consistency and accuracy of the intended state or information, and any unauthorized modification of data is a violation of data integrity. In this way, the proposed model acts by deliberately manipulating the integrity of the data to be stored.

A scenario where a buyer interacts by entering some personal and sensitive data on a large retailer's purchase site is proposed to illustrate the functioning of the Integrity framework. Bob accessed a large retailer's site to purchase a product he had been looking for in a while. Bob was redirected to the payment page upon selecting his product and placing it in the shopping cart. Bob then enters his personal and credit card details for payment. The system asks if Bob wants to keep his card details registered in the system. As Bob wants to make future purchases easily and quickly, he confirms his credit card details into the system. The system then stores the name printed on the card, the expiration date, and Bob's secret card number. However, after a while, due to an internal security breach, the database with the card information of several users was leaked, exposing Bob's data.

The Integrity framework can be used at the application level as an object-relational mapping framework where data is already encrypted in the database to avoid data exposure in these situations. In this way, once the data is sent from the model layer to the data persistence layer, Integrity performs an object-relational mapping, transforming the object model into a relational model preparing it for storage in a database. After this, the Data Masking process obfuscates the data using Feistel and AES Key Expansion ciphers in the case of Date fields. If the data is of String type, the data is encrypted. Next, Integrity SQL (ISQL) automatically assembles the SQL query for the data store. In case of a query to retrieve encrypted or masked information from the database, ISQL does the opposite process, taking the encrypted String data and leaving the text unencrypted.

However, in the case of the Date type, as there is a maximum representation limitation, masking can leap. Then, ISQL builds the query considering the finite body size. Therefore, when using Integrity, even if the data were exposed, it would not make any sense to whoever obtained it. Figure 2 shows the architec-

ture of the Integrity framework. For example, if the retail company system adopts Integrity when registering the name *Bob* and the credit card expiration date of *2022-01-01* the framework would encrypt the name and obfuscate the expiration date by taking a leap in the finite body, storing the information in the database with the values *X5rMlOVrdE0BqjXKii3VvA==* and *8806-12-14*.

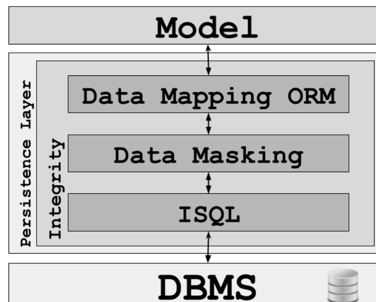


Figure 2: Schema of Integrity.

In Date types, the value is limited to the size of the finite field, so it is possible to jump with the date beyond the pre-established limit in the DBMS. For example, the Date type in MySql only supports dates with limits between 0001-01-01 to 9999-12-31, or a range of 3652060 days. With the proposed cipher and algorithm, a date 2022-01-01 could exceed this limit, causing an error when inserted into the database. Integrity uses the *MOD* operation to overcome this issue, limiting the field's size and creating a new date with its integrity broken, but within the range limits, without generating an overflow error when saving it.

Integrity does not intend to prevent data from being leaked but rather, if attacked, to be exposed in a non-integral manner, so the data do not make sense to the attacker and prevent its use in any way.

4.1 Data Masking

The Data Masking process, represented in Figure 3, receives the data to be manipulated, a 128-bit key defined by the developer in the system's source code and an automatically generated seed. This seed is derived from the initial seed, the table structure, and the column chosen to be encrypted. Consequently, the aim is to increase entropy and make difficult an attack by frequency analysis since each table column will use a different seed in the masking process.

The *Data Masking* process follows the following steps:

- The encryption key $[K]$ is submitted to the AES *Key Expansion* routine (Dworkin et al., 2001), generating 10 128-bit subkeys $[K_1],[K_2]...[K_{10}]$.

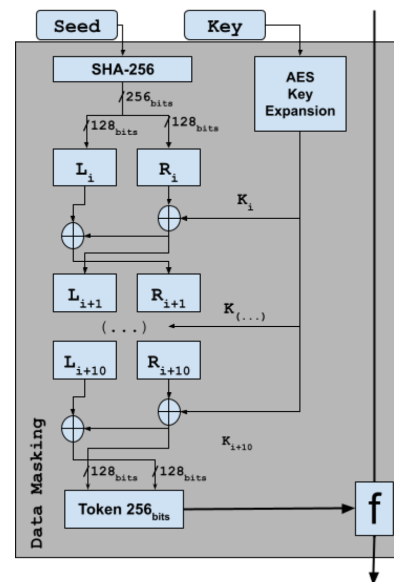


Figure 3: Data Masking.

- The seed is submitted to SHA-256 (Dang, 2015), generating a fixed-length 256-bit output that is later split into two 128-bit parts, $[R_i]$ and $[L_i]$.
- The two parts of the SHA-256 digest and the 10 subkeys are subjected to a 10-round Feistel Network.
- Each round, a logical operation XOR is performed between the round subkey $[K_i]$ with the 128bits $[R_i]$ right side.
- The output of this operation becomes the left side of the next round $[L_{i+1}]$, besides it is operated again (XOR) with the left side of the $[L_i]$ round, generating the right side of the next round $[R_{i+1}]$.

The output of the Feistel Network generates a 256bit *Token* that is submitted to a function.

For data masking of the *date* the token into an Integer, representing the amount of "leaps" that will be added to the data to be obfuscated unsing the following: $Data_{Masked} = Data_{Clear} + (Token\ mod\ N)$.

- N is the field limit according to the data type in the chosen DBMS, being responsible for the leap assigned to the overflow of the field limit.

The same output will be when the same input (key and seed) enters the algorithm's tokenization process. In this way, the data restoration process takes place by reducing the Data Obfuscated by the Token, as follows: $Data_{Clear} = Data_{Obfuscated} - (Token\ mod\ N)$.

For data encryption of the *varchar* type, the AES algorithm is used because native functions such as *AES_ENCRYPT()* require *varbinary* or *blob*. In this case, the 256bit token is used as the encryption pass-

word (128bit MSB) and initialization vector (128bit LSB).

4.2 Implementation

Integrity can be used to extend an application's persistence layer, implemented as a library. It is based on three main features:

- Object-relational Mapping: represents database tables in the application as objects so that entropy generation is possible using the attributes defined in the class combined with the key also configured at the application.
- Data Persistence: must be able, through object-relational mapping, to create commands and queries for persistence in the database.
- Security and Obfuscation: it obfuscates and restores the data integrity using the algorithm presented in section 4.1, considering commands created for data persistence and the entropy generated by the object-relational mapping.

This version[1] was built in the C# language using the .NET Core 5.0 development platform and the Visual Studio IDE. The development considered only the MySql DBMS and only dealt with the Datetime and varchar data types. The Integrity class diagram can be seen in Figure 4. Integrity is structured in three class libraries:

- Core, which contains the main classes for the library functioning.
- Attributes, which contain everything related to the classes' attributes for the library's object-relational mapping.
- Lib, which contains useful classes for implementing data persistence and security.

The Core class library is the core of the package. It contains the main classes for configuration, relational mapping, base repository, and security, namely:

- IntegrityConfiguration is responsible for the initial configuration of the application by invoking the ConfigureIntegrity method. It invokes the other major libraries' methods to store the database Connection String, the Cipher Key, and the Entropy Key.
- IntegrityBaseRepository is the abstract class that must be extended by the application's persistence layer classes and provides the methods developed for integrity functioning for persistence and data reading.
- IntegrityMapperRepository is the class responsible for the object-relational mapping of the library. It is responsible for creating the insert, delete, update, and select commands. In addition, when using the methods of the IntegritySecurity-QueryLanguage class, it defines the parameters to be saved in the database.
- IntegritySecurityQueryLanguage is the class that has the definitions and encryption algorithm of the library and this approach. It provides the methods for obfuscating the data and recovering the original data.

The Attributes class library has everything related to object attributes for implementing the library's object-relational mapping and contains the following classes:

- AttributesUtils is the class that contains utilities for manipulating the attributes defined in the entities, such as creating a parameter dictionary returning a list of attributes from the object's properties.
- Column is the column attribute that captures and relates the column name in the database to the object's properties.
- Key is the attribute that defines a specific property as a table key.
- Table is the attribute that defines the table's name that refers to the object in use in the application.

All attributes listed are mandatory for the library to function. It is possible to create the database's queries and commands from them. They are defined in the application's Entity classes. The Lib class library contains useful classes for implementing data persistence and library security, as following:

- IntegrityDatabaseUtils has utilities for saving the database connection string used by the application to create a connection to the DBMS.
- MysqlConstants has constants defined by the DBMS to be considered in the data leap according to the field limit of the employed field, as is the Date case.
- AesService is the class that implements the 128-bit AES algorithm, which provides the key expansion methods used for token generation. We chose to use AES-128 CBC because the token output is 256bits divided into two parts, one for the initialization vector and one for the encryption key.

The sequence diagram interconnecting all the messages exchanges by objects in the Integrity framework can be seen in Figure 5.

[1] https://gitlab.com/eldercosta/integrity-csharp

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

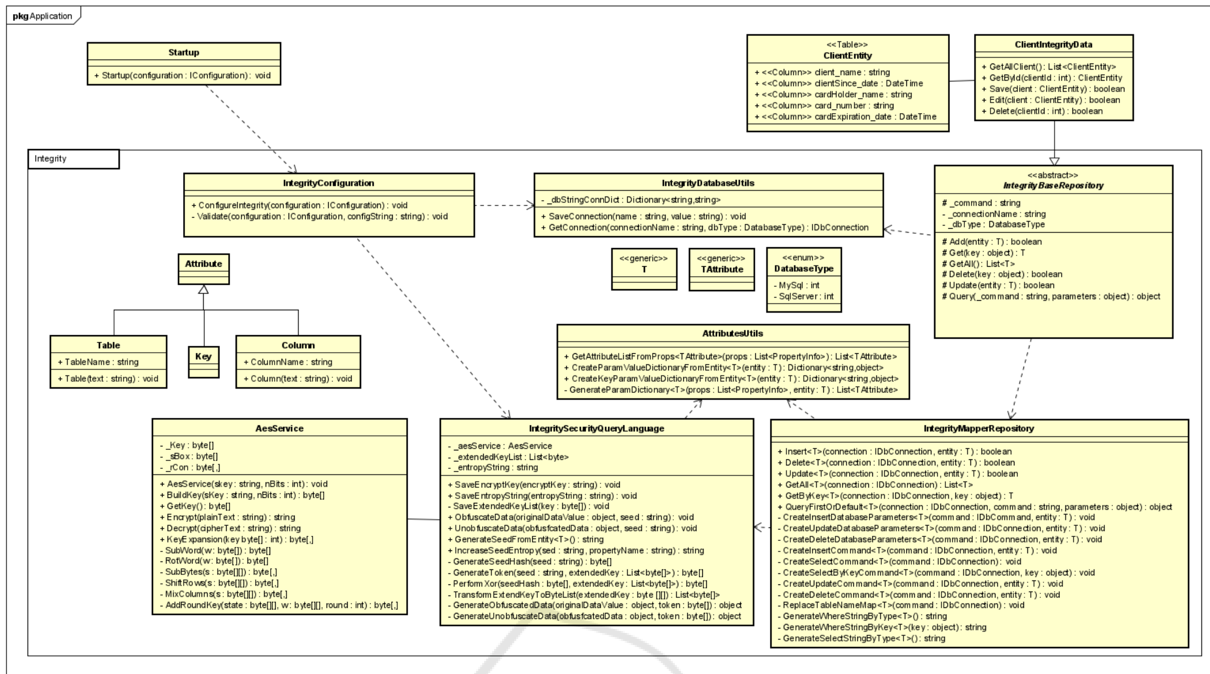


Figure 4: Class diagram of the Integrity framework.

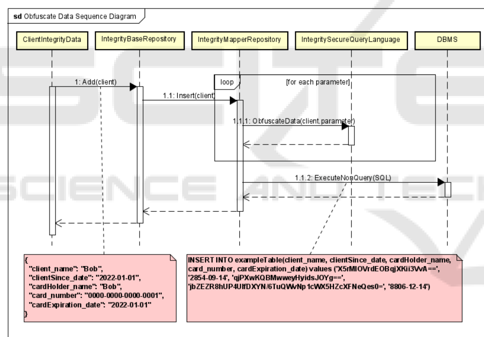


Figure 5: Sequence diagram of Integrity.

```
appsettings.Development.json   ClientEntity.cs
1  {
2      "Logging": {
3          "LogLevel": {
4              "Default": "Information",
5              "Microsoft": "Warning",
6              "Microsoft.Hosting.Lifetime": "Information"
7          }
8      },
9      "Integrity": {
10         "ConnectionString":
           "Server=localhost;Port=3306;Database=integrity;
           Uid=integrity_user;Connection
           LifeTime=0;Pooling=true;Pwd=integrity123;",
11         "EncryptKey": "0123456789ABCDEF",
12         "EntropySeedString": "IntegrityExample"
13     }
14 }
```

Figure 6: Integrity's configuration file.

5 CASE STUDY

A case study is presented based on the scenario proposed in Section 4, where Bob and Alice register their data on a retailer's platform to show how Integrity works. The case study will be conducted from the developer's perspective on how Integrity works on the backend. The case study will store the customer name *(client_name varchar(256))*, registration date *(clientSince_date date)*, cardholder name *(cardHolder_name varchar(256))*, card number *(card_number varchar(256))*, and expiration date *(cardExpiration_date date)* in a database. The fields using *varchar* and *date* will be encrypted and obfuscated.

From the developer's point of view, an initial password *(EncryptKey=0123456789ABCDEF)* and an initial seed *(EntropySeedString=IntegrityExample)* are defined, which will be used in the obfuscation and encryption of the application data, as shown in the Figure 6.

Afterward, it is created a class in the business model layer named *ClientEntity* with five columns, as shown in Figure 7.

Each column of the table receives a different token, generated from the *EncryptKey* and *EntropySeedString* defined in the configuration file. Then, when the data is persisted in the database, the names and dates will be different. It makes difficult frequency attacks. The token generation process is presented in Table 1.

When the system is in production, the *client_name* Bob and Alice will be persisted as X5rMlOVrdE0BqjXKii3VvA== and bnRP-

```
1  using Integrity.Attributes;
2  using System;
3  namespace Domain.Entities.Entity{
4      [Table("exampleTable")]
5      public class ClientEntity{
6          [Column("client_name")]
7          public string client_name { get; set; }
8
9          [Column("clientSince_date")]
10         public DateTime clientSince_date { get; set; }
11
12         [Column("cardHolder_name")]
13         public string cardHolder_name { get; set; }
14
15         [Column("card_number")]
16         public string card_number { get; set; }
17
18         [Column("cardExpiration_date")]
19         public DateTime cardExpiration_date { get; set
           ; }
20
21     }
22 }
```

Figure 7: The ClientEntity class.

Table 1: The Token Generator.

EncryptKey:			0123456789ABCDEF		
EncryptKey (HEX):			30313233343536373839414243444546		
AES Key Expansion		SubKey1:	2a5f68291e6a5e1e26531f5c65175a1a	SubKey2:	d8e1ca64c68b947ae0d88b2685cfd13c
		SubKey3:	56df21f39054b589708c3eaff543ef93	SubKey4:	4400fd15d454489ca4d87633519b99a0
		SubKey5:	40ee1dc494ba55583062236b61f9bacb	SubKey6:	f91a022b6da057735dc274183c3bced3
		SubKey7:	5b9164c0363133b36bf347ab57c88978	SubKey8:	3336d89b0507eb286ef4ac83393c25fb
		SubKey9:	c309d789c60e3ca1a8fa902291c6b5d9	SubKey10:	41dce20887d2dea92f284e8bbeeefb52

EntropySeedString:			IntegrityExample		
Seed (base64):			SW50ZWdyaXR5RXhhbXBsZQ==		
client name		Seed:	exampleTableclient_nameSW50ZWdyaXR5RXhhbXBsZQ==clientSince_dateSW50ZWdyaXR5 RXhhbXBsZQ==cardHolder_nameSW50ZWdyaXR5RXhhbXBsZQ==card_numberSW50ZWdy aXR5RXhhbXBsZQ==cardExpiration_dateSW50ZWdyaXR5RXhhbXBsZQ==client_name		
		Digest:	5f6bc40b98ece33944b50a8475660b2334642867c59b03755456e9e47c095d4f		
	Feistel	L1:	5f6bc40b98ece33944b50a8475660b23	R1:	34642867c59b03755456e9e47c095d4f
		L2:	1e3b404edbf15d6b7205f6b8191e0755	R2:	41508445431dbe5236b0fc3c6c780c76
		L3:	99b14e2185962a28d668771ae9b7dd4a	R3:	878a0e6f5e677743a46d81a2f0a9da1f
		L4:	d1552f9cce33c2cad4e1bf0d05ea358c	R4:	48e461bd4ba5e8e20289c817ec5de8c6
		L5:	0ce49ca89ff1a07ea651be24bdc67166	R5:	ddb1b33451c262b472b00129b82c44ea
		L6:	9d5faef0c57837ec42d22242d9d5fe21	R6:	91bb32585a899792e4839c6664138f47
		L7:	68a130733729c0e1b941e87e58284194	R7:	f5fe9e83f251f70dfb93ca3c81fdbfb5
		L8:	ae6ffa43c460c4be90608d97d63536cd	R8:	c6ceca30f349045f292165e98e1d7759
		L9:	f5f812abf64eef7747d5c96ab72152a2	R9:	5b97e8e8322e2bc9d7b544fd6114646f
		L10:	989e3f61f42017687f4fd4dff0d2d1b6	R10:	6d662dca026ef81f389a1db547f38314
		MSB:	2cbacfc285bc26b617b2533ef91d7846	LSB:	b424f0a3719c31de68fd87e109cfa9f0
		Token:	2cbacfc285bc26b617b2533ef91d7846b424f0a3719c31de68fd87e109cfa9f0		
clientSince date		Seed:	exampleTableclient_nameSW50ZWdyaXR5RXhhbXBsZQ==clientSince_dateSW50ZWdyaXR5 RXhhbXBsZQ==cardHolder_nameSW50ZWdyaXR5RXhhbXBsZQ==card_numberSW50ZWdy aXR5RXhhbXBsZQ==cardExpiration_dateSW50ZWdyaXR5RXhhbXBsZQ==clientSince_date		
		Token:	eaeb82825559672c919a1933cb515c2767bfe68d1b050867f2f552bdea9ca170		
cardHolder name		Seed:	exampleTableclient_nameSW50ZWdyaXR5RXhhbXBsZQ==clientSince_dateSW50ZWdyaXR5 RXhhbXBsZQ==cardHolder_nameSW50ZWdyaXR5RXhhbXBsZQ==card_numberSW50ZWdy aXR5RXhhbXBsZQ==cardExpiration_dateSW50ZWdyaXR5RXhhbXBsZQ==cardHolder_name		
		Token:	df0518fe3f471230a27d3de9b4c2dbf3978a451fa9c92b1f812366521a4b18da		
card number		Seed:	exampleTableclient_nameSW50ZWdyaXR5RXhhbXBsZQ==clientSince_dateSW50ZWdyaXR5 RXhhbXBsZQ==cardHolder_nameSW50ZWdyaXR5RXhhbXBsZQ==card_numberSW50ZWdy aXR5RXhhbXBsZQ==cardExpiration_dateSW50ZWdyaXR5RXhhbXBsZQ==card_number		
		Token:	a5ff1f0a01551de922e44bd90673441a9455673307976c00cf8d4adbd12602c1		
card Expiration date		Seed:	exampleTableclient_nameSW50ZWdyaXR5RXhhbXBsZQ==clientSince_dateSW50ZWdyaXR5 RXhhbXBsZQ==cardHolder_nameSW50ZWdyaXR5RXhhbXBsZQ==card_numberSW50ZWdy aXR5RXhhbXBsZQ==cardExpiration_dateSW50ZWdyaXR5RXhhbXBsZQ==cardExpiration_date		
		Token:	e281e063a1a79bb936566e85fb7e0d4553c7bd4ae1d3541873f5009997148c53		

TRko8QMh2hfy4PoDKw==, the *cardExpiration_date* will be persisted as 12/14/8806 and 01/02/8807, as shown in Figure 10. Identical dates will be stored to exemplify the behavior of Integrity since the jump of each column of the table is different. Thus frequency attack in the masked data is harder to perform. Figure 8 shows the submitted data in clear.

```
POST   /Client

Request body          application/json

{
  "client_name": "Bob",
  "clientSince_date": "2022-01-01",
  "cardHolder_name": "Bob",
  "card_number": "0000-0000-0000-0001",
  "cardExpiration_date": "2022-01-01"
}
{
  "client_name": "Alice",
  "clientSince_date": "2022-01-20",
  "cardHolder_name": "Alice",
  "card_number": "0000-0000-0000-0002",
  "cardExpiration_date": "2022-01-20"
}

Execute   Clear
```

Figure 8: Sending data to storage.

To calculate the leap used in the obfuscation of Date fields, the token *(eaeb82825559672c919a1933cb515c2767bfe68d1b050867f2f552bdea9ca170)* generated for the field *clientSince_date* was used. The Token is divided into 64bit words. A *jumptoken* is obtained through the exclusive-or operation between each word of the token. The *jumptoken* is converted to a 64-bit unsigned Integer. The mod is calculated with *N*, in this case, 3652060, which is the day limit of the date field in MySQL. The result *(304138)* is the number of days to be added to the date. The process is shown in Figure 9.

$$jump_{token} = \bigoplus \begin{array}{c} [eaeb82825559672c] \\ [919a1933cb515c27] \\ [67bfe68d1b050867] \\ [f2f552bdea9ca170] \\ \hline [ee3b2f816ff91921c] \end{array}$$

$$toUInt64([ee3b2f816ff91921c]) = 2058867887756491758$$
$$2058867887756491758 \ mod \ 3652060 = 304138$$

$$clientSince_date_{Bob}: 01/01/2022 + 304138 = 14/09/2854$$
$$clientSince_date_{Alice}: 20/01/2022 + 304138 = 03/10/2854$$

Figure 9: Date leap calculation.

For *varchar* fields, the AES-128 algorithm is used in CBC operating mode. The most significant bits of the generated token are used as the key and the least significant bits are used as the initialization vector. In the case of the field *client_name*, which received the token *(2cbacfc285bc26b617b2533ef91d7846b424f0a3719c31de68fd87e109cfa9f0)*, it uses:

- key = *[2cbacfc285bc26b617b2533ef91d7846]*
- iv = *[b424f0a3719c31de68fd87e109cfa9f0]*

```
5 •  select * from exampleTable;

Form Editor | Navigate: 1/2
Client_name:          X5rMIOVrdE0BqjXKi3VvA==
ClientSince_date:     2854-09-14
CardHolder_name:      qjPXwKQBMwweyHyidsJOYg==
Card_number:          jbZEZR8hUP4UIfDXYN/6TuQWvNp1cWX5HZcXFNeQes0=
CardExpiration_date:  8806-12-14
Form Editor | Navigate: 2/2
Client_name:          bnRPTRko8QMh2hfy4PoDKw==
ClientSince_date:     2854-10-03
CardHolder_name:      FKqUtmu8Pd7oM1k/lhD/0w==
Card_number:          jbZEZR8hUP4UIfDXYN/6TtYH+MkRQPevtEfa864ze+s=
CardExpiration_date:  8807-01-02
```

Figure 10: Bob's and Alice's stored data.

When the system retrieves information from the database, the Integrity framework does the inverse

process, considering the columns' annotations and seeds defined in the source code. Then, the data is retrieved unencrypted or unobfuscated. Integrity was designed to make all the encryption and data obfuscation processing transparent to the system developer and user. Figure 11 shows the recovered data.

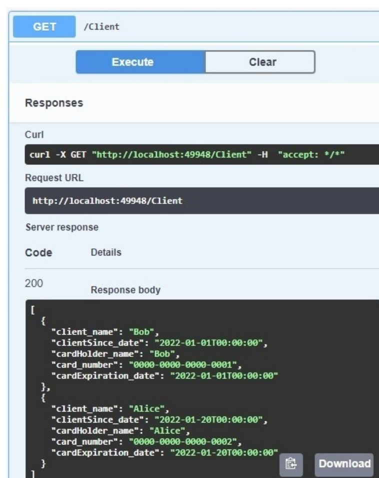


Figure 11: Return data.

When a *query* with date comparison is performed, the informed date also goes through the leap process before the query is performed, so there is no need to deobfuscate all dates in the column to perform the comparison.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

6 CONCLUSIONS

This work presented a framework for object-relational mapping that aims to obfuscate the information in the database, leaving the responsibility of the cryptographic process to the application layer instead of delegating this process to the database management system. Likewise, the framework aims to make the entire process transparent for the developer, who only needs to interfere in the solution's source code to define passwords and which columns will suffer data obfuscation. In this way, developing software considering the new data protection laws is facilitated.

The data will be stored in the database in an obfuscated way. The data that Integrity has persisted will be meaningless to those who illicitly obtain it if there is an information leak. Not even the Database Administrator will have access to such information since the system development team defines the password and the name of the attributes used in the cyphering process.

As future works, Integrity will be part of the Spring framework in Java to take advantage of all the infrastructure already available and for its wide use in the market. Other types of fields will also be used in Integrity, such as the Integer since only String and Date data are currently being used in this version. We opted for working firstly in two data types to reduce the scope of this work. Though, as future works, we intend to integrate data types with a finite body well-defined, as the IEEE 754 standard data types.

The framework provides the search for exact data in the database or by date range. However, searching for data using a String snippet in an obfuscated field is an open challenge. Thus, solutions to these limitations will be studied and proposed.

REFERENCES

Dang, Q. (2015). Secure Hash Standard. Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD.

Deshpande, A., Patil, A., Joshi, S., and Bothara, S. (2012). Article: Dbcrypto: A database encryption system using query level approach. *International Journal of Computer Applications*, 45(8):27–32.

Dworkin, M., Barker, E., Nechvatal, J., Foti, J., Bassham, L., Roback, E., and Dray, J. (2001). Advanced Encryption Standard (AES). Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD.

Hintzbergen, J., Hintzbergen, K., Smulders, A., and Baars, H. (2018). *Fundamentos de Segurança da Informação: com base na ISO 27001 e na ISO 27002*. Brasport.

Kim, D. and Solomon, M. G. (2014). *Fundamentos de segurança de sistemas de informação*. LTC, Rio de Janeiro, 1a edition.

Lorey, K., Buchmann, E., and Böhm, K. (2016). TEAL: Transparent Encryption for the Database Abstraction Layer. In *Proceedings of the CAiSE'16 Forum at the 28th International Conference on Advanced Information Systems Engineering*, pages 13–17, Ljubljana, Slovenia.

Pitta, P. E. B., Costa, E., de Siqueira, J. P. L., and Lazarin, N. M. (2020). LGPD Compliance: A security persistence data layer. In *Anais da XVIII Escola Regional de Redes de Computadores*, pages 123–127, Porto Alegre, RS, Brasil. SBC.

Stallings, W. (2015). *Criptografia e segurança de redes: princípios e práticas*. Pearson Education do Brasil, São Paulo, 6 edition.

Tanenbaum, A. (2003). *Redes de computadores*. Elsevier.

Zhang, M., Xie, G., Wei, S., Song, P., Guo, Z., Liu, Z., and Cheng, Z. (2018). Dmsd-fpe: Data masking system for database based on format-preserving encryption. In Wan, J., Lin, K., Zeng, D., Li, J., Xiang, Y., Liao, X., Huang, J., and Liu, Z., editors, *Cloud Computing, Security, Privacy in New Computing Environments*, pages 216–226, Cham. Springer International Publishing.