CEFET/RJ
Uned Nova Friburgo

SERVIÇO DE CRIPTOGRAFIA BASEADO NO FRAMEWORK INTEGRITY

João Felipe Knoller Marques e Juan Victor Oliveira Silva

Monografia apresentada ao curso de Sistemas de Informação, do Centro Federal de Educação Tecnológica Celso Suckow da Fonseca, CEFET/RJ, como parte dos requisitos necessários à Graduação.

Orientador(a): Nilson Mori Lazarin

Rio de Janeiro,

Julho 2023

SERVIÇO DE CRIPTOGRAFIA BASEADO NO FRAMEWORK INTEGRITY

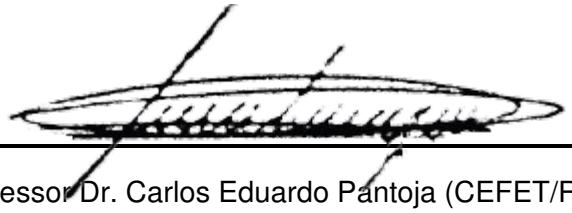
Monografia apresentada ao curso de Sistemas de Informação, do Centro Federal de Educação Tecnológica Celso Suckow da Fonseca, CEFET/RJ, como parte dos requisitos necessários à Graduação.

João Felipe Knoller Marques e Juan Victor Oliveira Silva

Banca Examinadora:

gov.br
Documento assinado digitalmente
Nilson Mori Lazarin
Data: 05/09/2023 18:45:03-0300
Verifique em https://validar.iti.gov.br

Presidente, Professor M.Sc. Nilson Mori Lazarin (CEFET/RJ) (Orientador(a))



Professor Dr. Carlos Eduardo Pantoja (CEFET/RJ)

HELGA DOLORICO BALBI:09599358783
Assinado de forma digital por HELGA DOLORICO BALBI:09599358783
Dados: 2023.09.04 16:44:19 -03'00'

Professora Dra. Helga Dolorico Balbi (CEFET/RJ)

Rio de Janeiro,

Julho 2023

CEFET/RJ – Sistema de Bibliotecas / Biblioteca Uned Nova Friburgo

M357s Marques, Joao Felipe Knoller.
Serviço de criptografia baseado no Framework Integrity. / Joao Felipe Knoller Marques; Juan Victor Oliveira Silva. — 2023.
52f. : il. (algumas color.) , grafs. , tabs.: em PDF.

Trabalho de Conclusão de Curso (Sistemas de Informação) - Centro Federal de Educação Tecnológica Celso Suckow da Fonseca, 2023.
Bibliografia: f. 50 - 52.
Orientador: Nilson Mori Lazarin.

1. Sistemas de Informação. 2. Banco de dados. 3. Criptografia. 4. Framework. I. Silva, Juan Victor Oliveira. (Co-autor). II. Nilson Mori Lazarin. (orientador) II. Título.

CDD 658.4038

Elaborada pela bibliotecária Cristina Rodrigues Alves CRB7/5932

RESUMO

Serviço de criptografia baseado no Framework Integrity

Com o avanço das discussões e criação de novas legislações sobre a segurança dos dados dos usuários em sistemas de informação, surgiu a necessidade de sistemas já existentes se adaptarem. Grande parte desse desafio está na forma como são armazenados os dados da aplicação, uma vez que os métodos de criptografia amplamente utilizados possuem dificuldade em lidar com diferentes tipos de dados. Uma solução é o *framework Integrity*, capaz de ofuscar dados antes da inserção dos mesmos no banco de dados, e garantindo a preservação do formato do dado. Esse trabalho visa estender o *framework*, mudando a abordagem da aplicação, oferecendo um serviço de criptografia dedicado multiplataforma, capaz de cifrar dados em claro para qualquer aplicação, independente da sua linguagem de programação ou banco de dados, e em tempo de execução, com um melhor desempenho de tempo quando comparado ao *Integrity*.

Palavras-chave: Preservação de formato; Criptografia; Banco de Dados

ABSTRACT

Encryption service based on Integrity Framework

With the advancement of discussions and the creation of new legislation on the security of user data in information systems, the need arose for existing systems to adapt. A large part of this challenge is in the way the application data is stored, since the widely used encryption methods have difficulty in dealing with different types of data. One solution is the Integrity framework, capable of obfuscating data before inserting it into the database, and ensuring the preservation of the data format. This work aims to extend the framework, changing the application's approach, offering a dedicated cross-platform encryption service, capable of encrypting data in plain text for any application, regardless of its programming language or database, and at runtime, with a better performance when compared to Integrity.

Keywords: Format preserving; Cryptography; Database

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1- Introdução

A internet se tornou uma ferramenta essencial para grande parte da população da vida moderna. A quantidade de usuários da rede mundial de computadores tem crescido a cada ano. Com esse crescimento rápido e desorganizado, um problema cibernético começou a ganhar destaque mundial: a segurança de dados.

O vazamento de dados sigilosos tem afetado diretamente os usuários do mundo digital. Segundo o *Massachusetts Institute of Technology* (MIT), mais de 205 milhões de dados foram vazados no Brasil no ano de 2019 (NETO et al., 2021), um aumento de cerca de 493% comparado ao ano anterior. Somente nos seis primeiros meses de 2021, foram mais de 4,6 bilhões de credenciais vazadas ao redor do mundo, representando um aumento de cerca de 387% em relação aos dados registrados durante todo o ano de 2019 (CORACCINI, 2021).

Segundo a *International Business Machines Corporation* (IBM) (IBM SECURITY, 2020), em 2018, um vazamento com mais de 50 milhões de registros gerou um prejuízo, em média, de 392 milhões de dólares às grandes empresas de tecnologia. E esse prejuízo médio aumentou em 19 milhões de dólares entre 2019 e 2020.

Essa realidade gera um debate cada vez mais recorrente: o papel do Estado em garantir a segurança de seus cidadãos como usuários da internet. Alguns governos já têm debatido e tomado providências a respeito desse assunto. É o exemplo da *General Data Protection Regulation* (GDPR) na Europa e da Lei Geral de Proteção de Dados Pessoais (LGPD) no Brasil. A LGPD visa proteger os direitos fundamentais de liberdade e privacidade, além da livre formação da personalidade de cada indivíduo (BRASIL, 2018a). Dessa forma, toda empresa que realiza tratamento de dados de seus clientes deve explicitar a finalidade, assim como prover a privacidade dos dados coletados. Isso implica em medidas que devem ser tomadas por parte das empresas de tecnologia, como diz o artigo 46 da lei: *"Os agentes de tratamento devem adotar medidas de segurança, técnicas e administrativas aptas a proteger os dados pessoais de acessos não autorizados e de situações acidentais ou ilícitas de destruição, perda, alteração, comunicação ou qualquer forma de tratamento inadequado, ou ilícito"* (BRASIL, 2018b).

Para todo descumprimento das exigências da Lei Geral de Proteção dos dados,

são definidas determinadas punições, como diz o artigo 52: *"Os agentes de tratamento de dados, em razão das infrações cometidas às normas previstas nesta Lei, ficam sujeitos às seguintes sanções administrativas aplicáveis pela autoridade nacional"* (BRASIL, 2018b). Algumas das possíveis sanções aplicadas desta lei são: advertências, multas simples e multas diárias, bloqueio e eliminação dos dados pessoais inflacionados e suspensão parcial do funcionamento do banco de dados referente aos dados inflacionados.

Uma das medidas mais comuns e conhecidas da proteção de dados é a criptografia. Ela é definida como um recurso tecnológico utilizado na comunicação de mensagens, tendo como objetivo tornar os dados incompreensíveis para garantir a sua confidencialidade. A criptografia utiliza um algoritmo chamado cifra e uma chave secreta, que é um valor qualquer a ser utilizado no algoritmo. Se uma mensagem criptografada for interceptada por terceiros que não possuem a chave secreta, eles não poderão descriptografá-la e, assim, não conseguirão entender o seu significado (AUMASSON, 2017).

Para garantir a segurança dos dados, os Sistemas de Gerenciamento de Banco de Dados (SGBD) disponibilizam funções nativas de criptografia. Entretanto, possuem restrição a determinados tipos de dados e já existem métodos que oferecem melhor desempenho conforme proposto em (PITTA et al., 2020), por exemplo. Um dos principais problemas ocorre quando uma grande quantidade de dados é processada. Nesse caso, as funções nativas do SGBD desempenham seus piores resultados.

Outra forma de prover segurança aos dados, é a utilização de técnicas de criptografia simétrica, como o *Advanced Encryption Standard* (AES) e *Data Encryption Standard* (DES), e persistindo o dado ofuscado posteriormente no banco de dados. Todavia, essa estratégia exige uma série de alterações no formato e estrutura dos dados, bem como na estrutura da base de dados, comprometendo tanto o tamanho dos campos quanto a integridade da base, demandando um investimento alto, e muitas vezes inviável para realizar essa adaptação.

Dado esse cenário, a *Format Preserving Encription* (FPE) é uma técnica que busca solucionar os problemas acima citados (BHATT; GUPTA, 2016). Atualmente, já existem estudos e aplicações que têm por objetivo armazenar dados criptografados preservando seu formato original e consequentemente diminuir os custos para a adequação das empresas em relação às novas legislações e o tratamento quanto aos dados de seu usuário. É o caso do *Integrity*, um *framework* objeto-relacional para segurança de dados (COSTA et al., 2022).

1.1- Definição do Problema

O *Integrity Framework*, em sua ideia e implementação, possui algumas características relevantes a serem apontadas. Ele utiliza a abordagem de implantação em forma de biblioteca, acopla a segurança e persistência dos dados da aplicação em uma mesma camada, possui uma etapa de geração de chaves desnecessariamente repetitiva que afeta o seu desempenho, além de proteger somente dados dos tipos *varchar* e *date*.

A abordagem adotada gera como consequência a necessidade de criar e reimplementar a biblioteca para ser utilizada em sistemas desenvolvidos em diferentes linguagens. Ou seja, para cada cliente com uma linguagem de programação nova, deve existir todo um retrabalho para definir a comunicação entre as partes interessadas.

Outra consequência indesejada é a forma como algumas funções utilizadas na criptografia foi implementada. Há um momento em que são geradas chaves de forma determinística, isto é, para uma mesma entrada, a saída deve ser a mesma também. Essas chaves são utilizadas para a ofuscação do dado em claro e são baseadas nos nomes das tabelas presentes no banco de dados. Isso significa que para criptografar um campo de uma tabela denominada de “exemplo“, há um cálculo que gera uma chave a partir do nome da tabela. Essa chave é utilizada sempre que forem realizadas as operações de criptografia nesta tabela. O *Integrity Framework* gera essas chaves a cada operação criptográfica, gerando então uma redundância nesses processos uma vez que essas chaves poderiam ser geradas uma única vez e armazenadas para serem consultadas posteriormente, sem a necessidade de gastar processamento computacional.

1.2- Objetivo

Apresentados os conceitos acima, este trabalho tem por objetivo implementar um serviço para cifragem e ofuscação de dados em bancos de dados relacionais, além de seus métodos reversos, em tempo de execução. Para isso, serão utilizadas a abordagem FPE, o algoritmo de Cifra de *Feistel*, AES, funções de *hash* criptográficas e a adição dos tipos inteiro e ponto flutuante aos tipos protegidos. Busca-se remover a responsabilidade

da ofuscação dos dados da camada de persistência para tornar a utilização das ferramentas de criptografia mais eficiente, permitindo maior escalabilidade e, por consequência, facilitar a configuração e uso dessa criptografia em aplicações reais.

Outra contribuição importante é a diferente estratégia utilizada para implementação do serviço proposto, na qual a aplicação deixa de ser uma biblioteca e passa a ser uma *Application Programming Interface* (API), uma espécie de contrato de serviço entre duas aplicações. Isso significa que, para utilizar os serviços de criptografia, não é mais necessário utilizar ou implementar, quando não existir, versões em dada linguagem para realizar a comunicação entre as partes interessadas. Essa comunicação por sua vez é realizada pelo protocolo *Hypertext Transfer Protocol* (HTTP), amplamente utilizado e suportado por diferentes linguagens de programação.

Após a realização de testes, observou-se que a solução proposta foi capaz de oferecer o serviço de criptografia para uma aplicação independente da sua linguagem de programação ou banco de dados utilizado. A solução também teve um desempenho melhor que o *Integrity Framework* ao submeter grandes quantidades de dados aos processos criptográficos. Os resultados são alguns dos diversos benefícios ao disponibilizar um serviço dedicado para a criptografia, melhor discutidos ao longo do trabalho.

1.3- Estrutura do trabalho

No Capítulo 2, é apresentado o Referencial Teórico, uma base de conhecimento sobre conceitos e técnicas necessárias para o compreendimento do modelo.

No Capítulo 3, são abordados os trabalhos relacionados, artigos que basearam a discussão para a proposição de uma nova abordagem para uma solução de criptografia.

No Capítulo 4, é apresentada a proposta da nova abordagem. Uma API independente de linguagem de programação e banco de dados, para a solução que o *Integrity Framework* procura oferecer, que só estão disponíveis para usar em C# e *MySQL*.

No Capítulo 5 é abordada a construção, organização e implementação do serviço de criptografia dedicado. Nele é possível encontrar detalhes sobre a execução técnica do projeto.

No Capítulo 6, é apresentada a prova de conceito, ou seja, um sistema que utiliza

a API desenvolvida, realizando operações no banco de dados com os conteúdos já criptografados.

No Capítulo 7, são realizados os experimentos de desempenho. Ocorre a comparação de performance entre a API e o *Integrity Framework*, visto que a abordagem utilizada na API garante algumas otimizações relevantes para o custo computacional da criptografia.

No Capítulo 8, por fim, são dispostas as considerações finais e discussões de futuros trabalhos.

2- Referencial Teórico

Neste capítulo, serão apresentados a teoria, conceitos e técnicas relacionados a criptografia, segurança da informação e matemática, necessárias para o entendimento do sistema proposto.

2.1- Segurança da Informação

Para entender a finalidade da segurança da informação, é necessário conhecer seus princípios fundamentais, presentes nos seguintes tópicos conforme (NETO; ARAÚJO, 2019):

1. Disponibilidade - A informação deve estar sempre disponível aos usuários para qualquer tempo ou finalidade.
2. Integridade - A informação deve manter seu conteúdo na mesma condição que foi disponibilizado, íntegro sem alterações indevidas, intencionais ou acidentais.
3. Confidencialidade - A informação deve estar protegida, garantindo e limitando o acesso somente a quem for destinado.

2.2- Criptografia e *Format Preserving Encryption*

Criptografia é um sistema amplamente utilizado para comunicação em um meio não seguro. Para isso, é necessário realizar uma série de operações matemáticas sobre um dado, com uma determinada chave, gerando um texto cifrado, ininteligível para qualquer um que não tenha acesso ao processo reverso (STINSON; PATERSON, 2019).

Para entender melhor a área da criptografia, existem termos a definir. Um **texto em claro** corresponde à mensagem original, enquanto a mensagem criptografada é

conhecida como **texto cifrado**. O processo de transformação de texto em claro para texto cifrado é a **criptografia** e seu processo reverso é a **descriptografia**. Um esquema que aborda todos os conceitos anteriores é conhecido como um sistema criptográfico ou uma cifra (STALLINGS, 2017).

2.2.1 *Format Preserving Encryption*

Em técnicas de proteção de dados como a criptografia simétrica, existe um problema quanto a preservação do tipo do dado. A entrada X e o resultado Y sempre serão de valor alfanumérico. Como observado na Tabela 1, a criptografia simétrica, exemplificada pelo *Advanced Encryption Standard* recebe uma entrada em um formato de data e seu retorno é em texto.

Método Criptográfico	Entrada	Saída
AES	2023/07/10	23iTpNciSm4DmDhnd5WVtA== (Base64)
FPE	2023/07/10	2042-05-26T05:50:16.584Z

Tabela 1 – Comparação entre criptografia simétrica e criptografia com preservação de formato

Isso gera, em alguns casos, um contratempo importuno. Brightwell et al. (1997), por exemplo, tinham por objetivo criptografar entradas de uma base de dados sem ferir o tipo do dado em questão. Os autores observam a dificuldade de alcançar esse objetivo utilizando métodos de criptografia simétrica, exemplificado pela *Data Encryption Standard* (DES).

Existem algumas abordagens que procuram atacar o problema da perda do formato. Uma delas é denominada como Criptografia com Preservação do Formato. Ela consiste em mais um sistema criptográfico dentre outros já existentes afim de oferecer mais segurança a sistemas. Permite utilizar uma chave simétrica e deterministicamente cifrar uma entrada em claro X em uma saída criptografada Y, onde Y possui o mesmo formato de dado que X (BELLARE et al., 2009).

2.3- Funções de *Hash* Criptográficas

Amplamente usadas no contexto de segurança da informação, funções de *hash* são implementações que, dentre outras aplicações, asseguram a integridade de um dado. Consistem em um cálculo a partir de uma mensagem de parâmetro, retornando um valor de tamanho fixo, geralmente alfanumérico, conforme na Figura 1, o qual será outro valor completamente diferente caso qualquer *bit* da mensagem original seja alterado.

Figura 1 – Funcionamento de uma função de *hash*

Funções de *hash* criptográficas são eficientes contra-ataques de força bruta, visto que se torna computacionalmente inviável realizar o mapeamento do *hash* resultante para os dados originais. Seu funcionamento consiste no preenchimento do tamanho dos dados da entrada até um valor inteiro que seja múltiplo de um limitador de tamanho fixo, limitador esse geralmente entre 128 e 512 *bits*. Dentre esses valores preenchidos, é incluído o tamanho da mensagem original no formato de *bits*, sendo essa uma medida de segurança para dificultar que um eventual atacante consiga produzir um *hash* igual para uma mensagem alterada.

2.4- *Advanced Encryption Standard*

O *Advanced Encryption Standard* (AES) é uma técnica de cifra simétrica de bloco publicada pelo NIST no ano de 2001. Ele trabalha com entradas de blocos de texto de 128 *bits* (16 *bytes*) e utiliza chave criptográfica que pode ter tamanho especificado entre 128, 192 ou 256 *bits*, o que faz com que haja uma designação indicativa de qual chave o algoritmo esteja utilizando na ocasião (AES-128, AES-192, AES-256) (DWORKIN et al.,

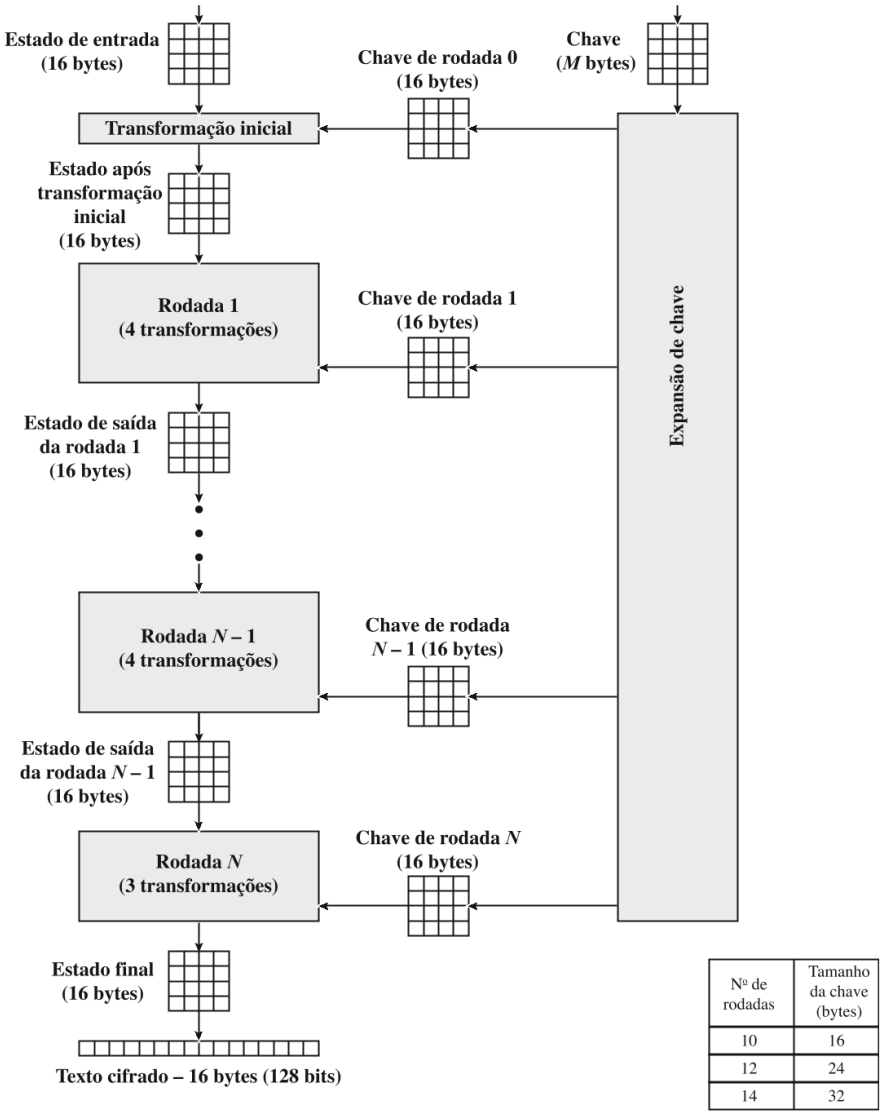


Figura 2 – Processo de encriptação do AES (STALLINGS, 2017)

2001).

Os dados do bloco de entrada (texto claro) são organizados e manipulados mediante uma matriz de *bytes* de quatro linhas copiada da entrada (como exemplificado na Figura 2, considerando uma chave de 128 *bits*), denominada Estado, cujo valor da quantidade de colunas resulta da divisão do tamanho do bloco de entrada por 32 bits, conforme o tamanho do bloco de entrada e o tamanho da chave de encriptação, sendo N = 10 rodadas para a versão padrão do AES-128, N = 12 rodadas para o AES-192 e N = 14 rodadas para o AES-256.

O processo de transformação realizado durante as rodadas se dá da seguinte maneira: antes da primeira rodada é utilizada uma função de transformação chamada

AddRoundKey. Nas N-1 rodadas iniciais tem-se a execução de outras funções, sendo elas *SubBytes*, *ShiftRows*, *MixColumns* e novamente a *AddRoundKey*. Por fim, na última rodada, são utilizadas apenas três das funções citadas.

A função *SubBytes* manipula uma matriz de 16x16 de *bytes*, na qual seu conteúdo é uma permutação de 256 valores possíveis formados por 8 *bits*, chamada de *S-Box*. É feita a substituição dos *bytes* individuais que formam Estado por um existente na *S-Box* (DWORKIN et al., 2001).

A função *ShiftRows* consiste no deslocamento de *bytes* pertencentes às linhas de Estado nela mesmo, onde a primeira linha não é alterada e na segunda linha há uma movimentação circular de 1 *byte* para a esquerda. Na terceira linha esse movimento também é realizado por 2 *bytes*, e na quarta, por 3 *bytes*.

A função de transformação *MixColumns* trabalha misturando os dados de cada coluna da matriz Estado. Em função dos quatro *bytes* que compõem a coluna, o valor de cada *byte* é mapeado para um novo.

A função *AddRoundKey*, de forma geral, trabalha realizando uma operação XOR entre cada um dos *bits* da tabela Estado com os *bits* da subchave criptográfica utilizada na rodada (DWORKIN et al., 2001). A subchave em questão consiste em uma matriz de mesmo tamanho de Estado, gerada a cada rodada a partir da chave criptográfica original. O escalonador de chaves do algoritmo fica a cargo de criar essas subchaves, criando uma chave também para a primeira rodada onde apenas a função *AddRoundKey* é utilizada (DWORKIN et al., 2001).

2.4.1 Expansão de Chave AES

O algoritmo de expansão de chaves do AES, observado na Figura 3, é utilizado para expandir uma chave inicial em um conjunto de subchaves utilizadas nas diferentes rodadas do algoritmo de criptografia. O processo do algoritmo começa com uma chave original, de tamanho 128, 192 ou 256 *bits*, e, dependendo do tamanho da chave, ela é dividida em palavras de 32 *bits*.

Em seguida, o algoritmo executa um laço de repetição para gerar novas palavras-chave, formando o conjunto de subchaves. O número de subchaves geradas depende do

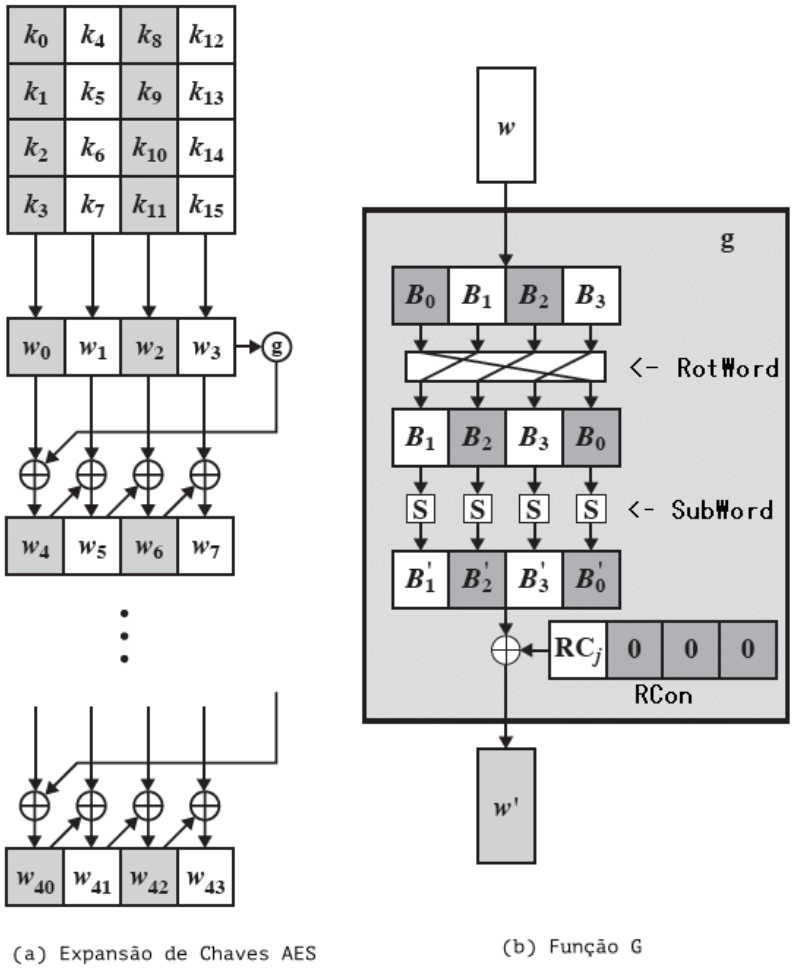


Figura 3 – Expansão de chaves AES - Adaptado de (STALLINGS, 2017)

número total de rodadas que serão executadas no algoritmo AES. Dentro desse laço, são aplicadas algumas transformações nas chaves, como rotação de *bits* (cada palavra de 32 *bits* é rotacionada a circular para a esquerda), substituição de *bytes* (geralmente usando uma tabela de substituição chamada *S-box*) e operações XOR (DWORKIN et al., 2001).

Essas operações são aplicadas iterativamente até que todas as palavras-chave necessárias sejam geradas. Cada nova palavra-chave é derivada das palavras-chave anteriores e da palavra-chave anterior a ela, juntamente com algumas transformações específicas. As subchaves expandidas são armazenadas em uma lista ou estrutura de dados para serem usadas nas diferentes etapas do algoritmo AES durante a criptografia e descriptografia. Cada rodada do algoritmo AES usa uma subchave diferente.

2.4.2 Modo *Cipher Block Chaining*

O modo de operação *Cipher Block Chaining* (CBC) é um dentre diversos modos criados para a criptografia de blocos. Para realizar uma implementação das especificações do AES, é preciso escolher um modo de operação. O CBC oferece melhor segurança por realizar operações XOR entre os blocos cifrados anteriormente antes de cifrar o texto por completo.

2.5- Cifra de *Feistel*

A Cifra de *Feistel*, conforme mostra a Figura 4 recebe como entrada no seu algoritmo o bloco de texto a ser criptografado e a chave de encriptação, denominada K. A entrada de texto é dividida em duas partes, o lado esquerdo, chamado de L, e o lado direito, chamado de R. Essas duas partes passam por um processo de n rodadas, sendo ao fim combinadas para produzir o texto encriptado (STALLINGS, 2017).

Cada iteração de rodada, designada por rodada *i*, é alimentada pela rodada anterior com as entradas L*i*-1, R*i*-1 e uma subchave k*i* derivada da chave k, sendo essas subchaves diferentes de k e umas das outras. Em cada rodada, é aplicada uma função F ao lado direito dos dados, cuja saída sofre uma operação XOR com o lado esquerdo dos dados. Após isso, a saída da operação XOR vira a entrada do lado direito da próxima etapa e o lado direito dos dados que foi utilizada como entrada para F se torna o lado esquerdo da próxima rodada.

O processo de decriptação é semelhante ao de encriptação já descrito, recebendo como entrada o texto encriptado e utilizando subchaves em ordem reversa. Tendo a subchave *kn* na primeira rodada de decriptação, as seguintes são designadas por *kn*-1, até chegar em *k*1.

Alguns fatores afetam a execução do algoritmo de *Feistel*. O tamanho do bloco de dados maior aumenta a segurança proposta, porém reduz o desempenho em velocidade do processo de encriptação/decriptação. Os blocos de 64 *bits* são uma escolha tomada de forma recorrente para projetos baseados em cifra de blocos, e por isso, muito usado

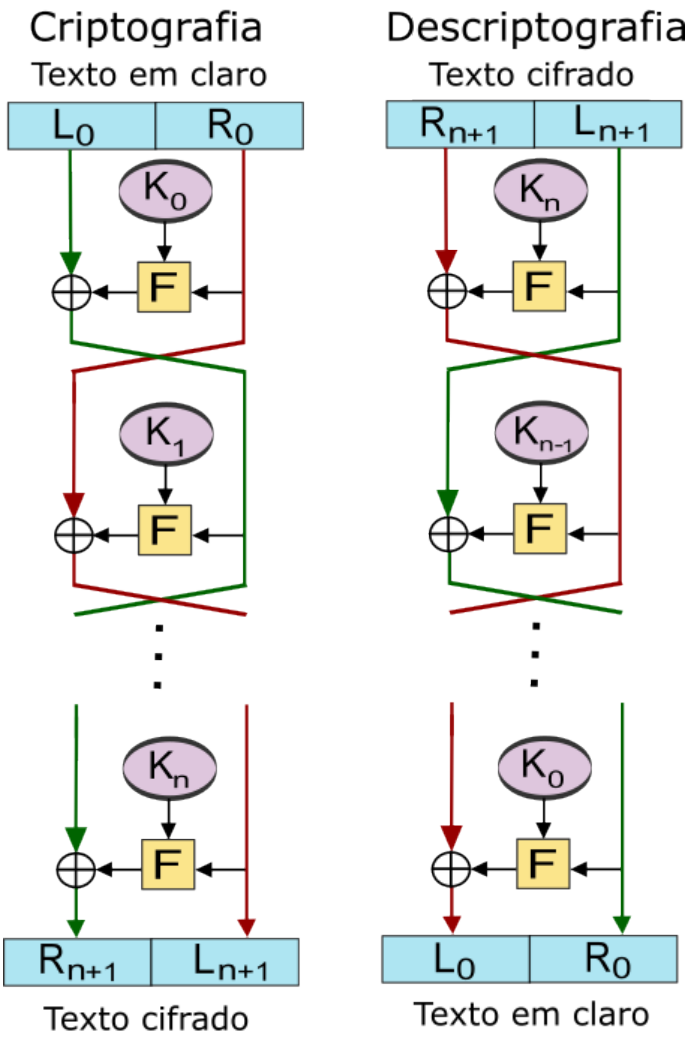


Figura 4 – Funcionamento da criptografia e descriptografia da Cifra de *Feistel*

no AES-128.

O tamanho da chave, de forma similar ao do bloco de dados, também influencia na segurança alcançada, sendo também o maior tamanho dela um fator que prejudica na velocidade de encriptação/decriptação do algoritmo. O tamanho de 128 *bits* é adotado como padrão, já que 64 *bits* ou menos são considerados inadequados.

O número de rodadas utilizadas pelo algoritmo influencia na segurança. Quanto mais rodadas utilizadas maior segurança oferecida, porém mais tempo utilizado na execução. São aplicadas de forma recorrente a quantia de 16 rodadas. Em relação à criação de subchaves, quanto maior for a complexidade do algoritmo, mais difícil para análise dos eventuais atacantes. O incremento de complexidade na construção da função F também contribui para dificultar a análise criptográfica de eventuais atacantes.

2.6- Corpos Finitos

Um corpo finito é um conjunto onde é possível realizar operações como soma, subtração, divisão e multiplicação sem sair do próprio conjunto. Como explica (STALLINGS, 2017), para satisfazer a definição de corpo finito, o conjunto Z_p, por exemplo, pode conter todos os inteiros $0, 1, 2, ..., p-1$. Todo corpo finito deve ser da ordem de p^k, onde p é um número primo e k um inteiro positivo.

Para entender a finalidade e aplicação de um corpo finito neste trabalho, será apresentado um exemplo simples. Considere um corpo finito A da ordem de 12 unidades. Isso pode ser traduzido no conjunto $1, 2, ..., 12$. Para representar todos os outros números que estão fora do conjunto, é necessário realizar uma operação de módulo: $N\ mod\ (p-t)$, onde N é qualquer inteiro e t é o tamanho do conjunto. Portanto, a representação do número N = 13 no corpo A é dada por $13\ mod\ 12 = 1$. Esse exemplo é o que acontece com um relógio analógico, como observado na Figura 5. O relógio representa o corpo finito A. Ao adicionar treze horas, isto é, uma hora a mais que o tamanho do conjunto A, a contagem continua mesmo após uma volta completa. Pode-se dizer então que $13 \equiv 1\ (mod\ 12)$, ou seja, 13 é equivalente a 1 dentro de um corpo finito de módulo 12.

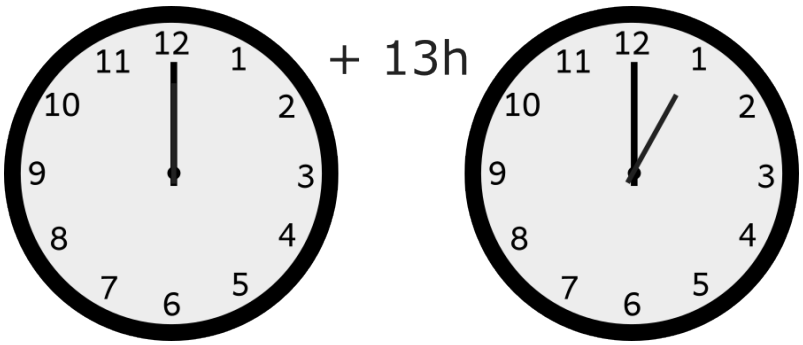


Figura 5 – O relógio representando um corpo finito

Esse conceito é fundamental para o armazenamento de dados e criptografia em geral, visto que todos os tipos de dados possuem um limite de tamanho, e então passa a ser necessário utilizar esta estratégia caso o resultado produzido nas funções de criptografia seja maior que o corpo finito do tamanho do dado em si.

2.7- *Clean Architecture*

A *Clean Architecture*, é um conceito introduzido por Robert C. Martin em seu livro *"Clean Architecture: A Craftsman's Guide to Software Structure and Design"* (MARTIN, 2017). Essa abordagem arquitetural enfatiza a separação de preocupações e a independência das regras de negócio das tecnologias e detalhes de implementação. A arquitetura é dividida em várias camadas com uma regra muito importante: a regra de dependência.

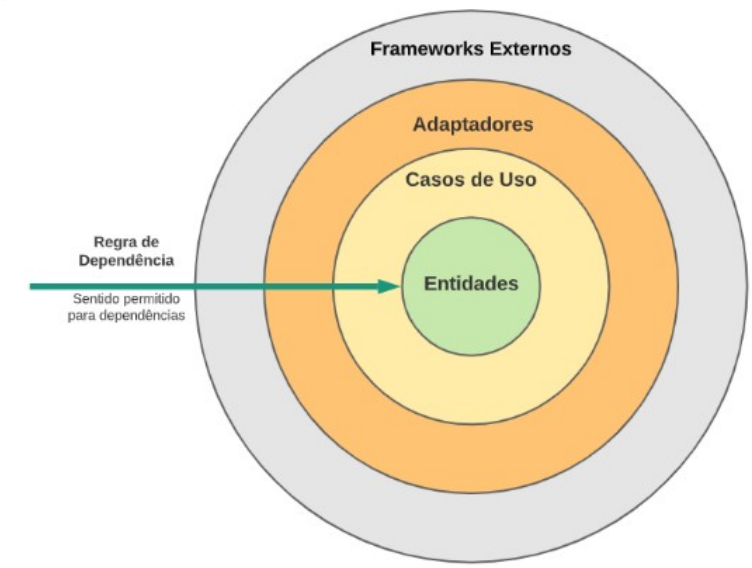


Figura 6 – Exemplificação da Regra de Dependência (VALENTE, 2022)

A Regra de Dependência mostrada na Figura 6, indica que a seta apenas segue para dentro, ou seja, a camada mais externa visualiza e utiliza a mais interna. O inverso não pode acontecer. As camadas mais internas não podem utilizar as camadas mais externas porque não são de suas responsabilidades. Isso não faz com que o *software* fique dependente ou restritivo, muito pelo contrário, a aplicação dessa regra diminui as limitações do código o tornando organizado e acessível.

Na camada de entidades, vista na Figura 7, deve ser alocada a lógica de negócio e as regras de alto nível. Essa camada pode ser usada por todas as outras, tendo em vista que possui regras mais gerais do *software*, ou seja, as entidades são usadas pelas classes mais externas. A camada de *useCases* é destinada a arquivos que lidaram com as regras de negócio para cada caso da aplicação.

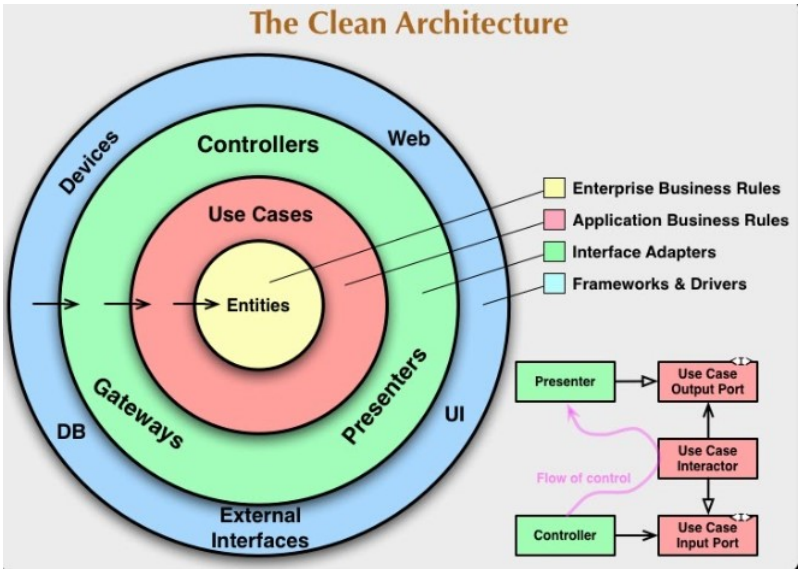


Figura 7 – Explicação do modelo *Clean Architecture* (MARTIN, 2017)

A camada de interface tem a função de converter as informações vindas das camadas internas (entidades + casos de uso) para o reconhecimento dos elementos pertencentes no próximo nível. E, por fim, a camada de *frameworks* e *drivers* é composta pelos elementos mais externos: *frameworks*, Banco de dados, bibliotecas e derivações.

3- Trabalhos Relacionados

Nesta seção serão apresentados os artigos acadêmicos que serviram de base para as discussões de criptografia com preservação de formato, serviço externo e banco de dados.

3.1- DMSD-FPE: *Data Masking System for Database Based on Format-Preserving Encryption*

O artigo em questão apresenta o modelo DMSD-FPE, um sistema de mascaramento de dados para banco de dados baseado na criptografia de preservação de formato, destacando os algoritmos de mascaramento apropriados para diferentes bancos de dados. Esse sistema desenvolvido é utilizado diretamente nos bancos de dados para realização de *backups* e mineração de dados (ZHANG et al., 2018).

Nele é apresentado a FPE, criptografia com preservação de formato, sendo uma técnica de criptografia mais interessante por preservar o formato dos campos no banco de dados, assim, não necessitando da adaptação de várias tabelas para receber as informações criptografadas. Além disso, é importante citar que esse processo é reversível e também preserva a integridade referencial das chaves estrangeiras.

3.2- DBCrypto: *Database Encryption System using Query Level Approach*

No artigo de (PUNDLIK, 2012), foi desenvolvido o sistema DBCrypto, capaz de atuar entre o usuário e o banco de dados. Ele é responsável por realizar a criptografia e proteção dos dados antes de serem inseridos no banco.

Qualquer atacante que tiver acesso ao banco de dados não terá nenhuma informação sobre como foi realizada a criptografia, visto que as mesmas estão devi-

damente seguras no programa intermediário. Portanto, em caso de falha do sistema ou do banco do cliente, ainda assim, os dados estarão protegidos.

Os dois projetos, o citado acima e o presente, se relacionam por utilizarem a mesma estratégia de ser um serviço intermediário do sistema cliente ao seu banco de dados, garantindo o armazenamento dos dados já protegidos. Algumas das diferenças estão na responsabilidade do armazenamento dos dados e a criptografia, que utiliza uma abordagem que já caiu em desuso por já ter sido quebrada. Além disso, o trabalho citado apresenta um sistema com conexão direta com o banco de dados, diferente do presente trabalho, que retorna os dados criptografados ao cliente, removendo a responsabilidade de persistência dos dados, resultando em um *software* dedicado à proteção de dados.

3.3- *Integrity: An Object-relational Framework for Data Security*

O objetivo principal do trabalho foi desenvolver o *Integrity Framework*, para ser utilizado em sistemas preexistentes, que garantisse a segurança dos dados, por meio de métodos de criptografia como Expansão de Chaves do AES e Cifra de *Feistel*, em uma base de dados relacional sem a necessidade de realizar alterações na base, já que também é utilizado o método de criptografia com preservação de formato (FPE). Os únicos tipos de dados que são tratados nesse trabalho são os do tipo *varchar* e *date* (COSTA et al., 2022).

O serviço proposto, assim como o *Integrity Framework*, busca assegurar a proteção dos dados de sistemas que já existem, além de otimizar alguns processos, remover a responsabilidade da camada de persistência de realizar a ofuscação e a criptografia, além da adição de novos tipo de dado para tratamento, o inteiro e ponto flutuante.

4- Proposta

Este trabalho propõe evoluir a ideia desenvolvida em (COSTA et al., 2022) e criar um serviço dedicado para a realização da ofuscação de dados, mudando a abordagem e removendo a responsabilidade da lógica e custo computacional da criptografia da aplicação do usuário. Um serviço especialista para as funções de ofuscar e cifrar dados melhora o desempenho na proteção dos dados, uma vez que todo o processo de expansão de chaves e geração de *tokens* é realizado apenas uma vez, no momento da configuração, diferente das soluções propostas no *Integrity Framework* e a biblioteca implementada em (PITTA et al., 2020), em que, a cada consulta, é necessário realizar a expansão da chave e realizar as 10 rodadas na Cifra de *Feistel* modificada. Além disso, este trabalho também adiciona a ofuscação para dados dos tipos inteiro e ponto flutuante aos tipos de dados já previstos, *varchar* e *date*, nas duas soluções citadas.

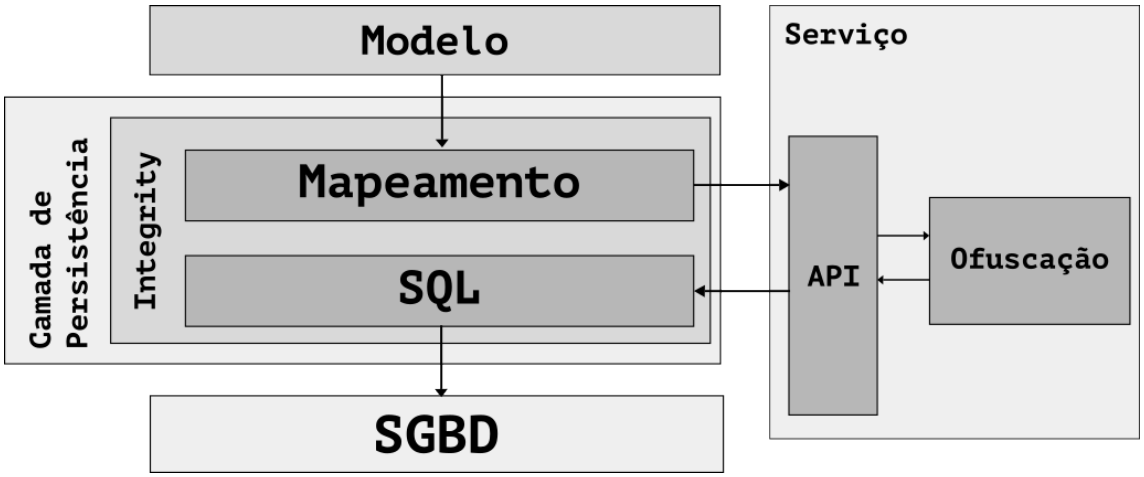


Figura 8 – Abordagem adotada para o serviço proposto

A abordagem de um serviço dedicado, conforme ilustra a Figura 8, permite a independência da aplicação interessada em ofuscar seus dados e a criptografia. No *Integrity Framework*, a cifragem dos dados está em formato de uma biblioteca desenvolvida em C# e apenas para o banco de dados relacional MySQL. Isso significa que, para utilizá-la, é preciso que haja uma reimplementação da biblioteca na mesma linguagem da aplicação interessada. Já na API proposta neste trabalho, por se tratar de um serviço externo, basta realizar uma comunicação utilizando o protocolo HTTP para o compartilhamento

dos dados, independente da linguagem ou do banco de dados utilizado na aplicação interessada.

Por outro lado, essa estratégia traz como desvantagem alguns pontos. Ao hospedar o serviço em um servidor na rede de internet por exemplo, existe o risco de esse servidor não estar disponível em algum momento e, portanto, não ser possível acessá-lo. Outro fator a se considerar é a possível e variável latência, dependendo diretamente do local físico do servidor e aplicação cliente. É possível ainda contornar essas situações utilizando uma *intranet*.

Ao considerar um cenário real para a aplicação da criptografia em dados de um sistema, é recomendada a utilização do serviço para abranger e proteger os dados pessoais e sensíveis de um usuário, sendo estes, dados capazes de identificar direta ou indiretamente um indivíduo vivo, conforme é definido pelo Serviço Federal de Processamento de Dados (SERPRO) (SERPRO, 2018), empresa pública responsável por soluções tecnológicas do Brasil. O conceito, por si, de dado pessoal pode variar dependendo da legislação local, porém recomenda-se seguir a lógica da capacidade de identificação de um indivíduo, para definir quais tabelas e campos de um dado sistema devem receber prioridade para a ofuscação do conteúdo dos mesmos.

Para utilizar o serviço, o cliente deve realizar a configuração dos dados que serão utilizados como parâmetro no processo da criptografia ou ofuscação. São eles: uma semente, uma chave criptográfica, a estrutura da tabela e o banco de dados. Na configuração inicial, o serviço calcula o *token* específico para cada atributo informado na configuração e persiste em uma base no próprio serviço. A persistência do *token* tem a finalidade de evitar a necessidade de novas gerações do mesmo em requisições futuras, propiciando melhor desempenho, e ainda sendo possível a opção de gerar um novo *token*, para imprevistos como o vazamento desse dado, por exemplo.

O *token*, citado anteriormente, é utilizado no processo de ofuscação dos dados dos campos date, já abordado no *Integrity Framework* e na biblioteca proposta por (PITTA et al., 2020), e nos campos dos tipos inteiro e ponto flutuante, contribuição adicionada nesse artigo. O início da geração dos *tokens*, como percebido na Figura 9, se dá com o recebimento dos parâmetros a serem cadastrados na base do serviço. A chave de 128 *bits*, K, é submetida a uma função AES de expansão de chaves, dando origem a 10 subchaves (K0 a K9). A semente recebida é parâmetro para uma função de *hash* SHA-256, que retorna uma saída de 256 *bits*. Essa saída é dividida em duas partes de

128 *bits*, denominada L a parte esquerda e R a parte direita.

As duas partes resultantes de SHA-256 e as 10 subchaves são utilizadas como parâmetro para um algoritmo de cifra de *Feistel*, composto por 10 rodadas N, denominadas aqui de N0 a N9. Na rodada inicial, o lado R, designado como R0, passa por uma operação de XOR com a subchave K0. A saída dessa operação se torna o lado L da iteração da próxima rodada e o lado esquerdo, sendo o inicial denominado como L0, se torna o lado R da próxima rodada. Esse processo se repete até que as 10 rodadas acabem. Ao término do algoritmo, é gerado o *token* de 256 *bits* que será cadastrado na base de dados, associado aos demais dados informados pelo usuário.

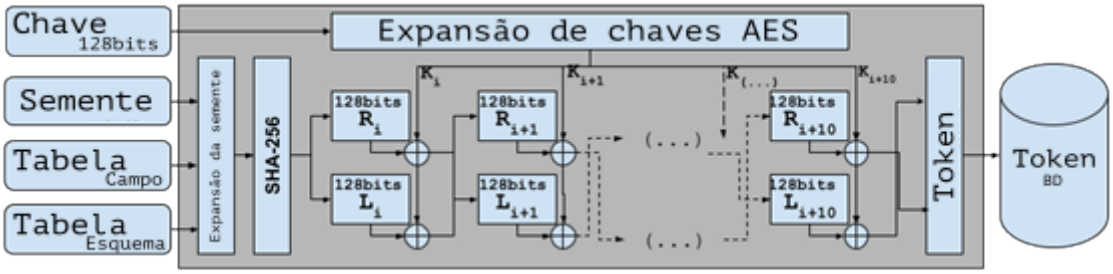


Figura 9 – Geração do token utilizado na criptografia

A ofuscação dos dados é feito utilizando o *token* gerado. O *token* de 256 *bits* é divido em quatro partes, ou *words*, de 64 *bits*. Essas partes são combinadas via uma operação XOR, que resulta em um novo *token* chamado por (COSTA et al., 2022) como *token* de salto, aqui simbolizado pela sigla ts. O ts é convertido em um inteiro de 64 *bits* sem sinal que vai ser utilizado para ofuscar os dados.

Como mostra a Figura 10, o dado ofuscado é resultado de $(dado\ em\ claro\ + ts)\ mod\ M$. Onde M é o tamanho limite de alcance para o tipo do dado suportado pelo SGBD informado. A operação Mod é para que o dado resultante da operação não venha extrapolar esse limite, o que causaria problemas no momento da persistência na base da aplicação.

Para a criptografia de campos do tipo *varchar*, é usada uma função do AES do modo *Cipher Block Chaining* (CBC), sendo depois convertida a saída para base 64, antes de retornar para o conjunto de dados criptografados. Após a ofuscação dos campos *date* e inteiro e da encriptação dos campos *varchar*, os dados são retornados para a aplicação, que persiste os dados ofuscados na base.

O processo de decriptação é semelhante ao anterior. Os dados em claro dos tipos *date* e inteiro são obtidos pelo resultado da operação $(dado\ ofuscado\ -\ ts)\ Mod\ M$. Os

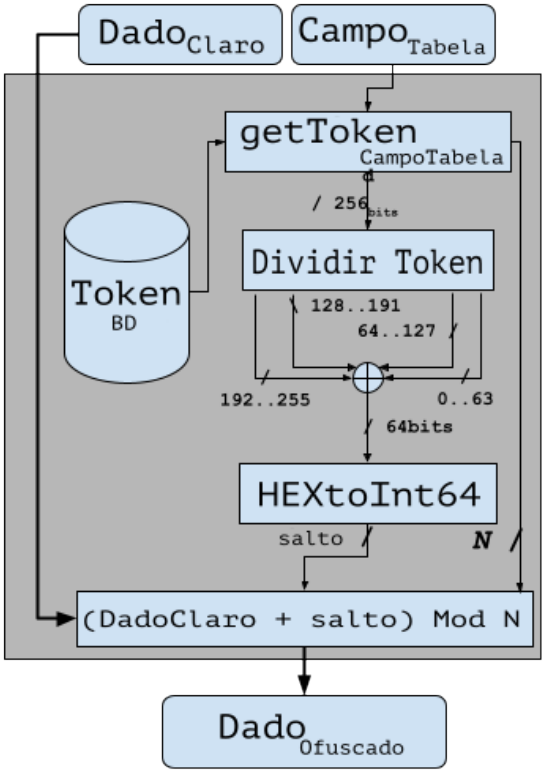


Figura 10 – Operação de ofuscação de um dado de preservando

campos *varchar* passam pela função de decriptação AES do modo CBC, e após todos os dados terem sido decriptados, são devolvidos para a aplicação.

5- Implementação

Visando comprovar toda a proposta abordada no capítulo anterior, foi desenvolvido um serviço externo. Sua construção utilizou como base o *Integrity Framework* (COSTA et al., 2022).

O serviço implementado possui como pontos principais a segurança e integridade dos dados recebidos, descentralização da cifragem e ofuscação de dados da aplicação cliente e disponibilização de uma configuração única para o usuário.

A versão 1.0.0 do sistema foi construída na linguagem *Typescript*, um superconjunto sintático estrito de *JavaScript* que adiciona tipagem estática opcional à linguagem. O *Javascript* é uma linguagem que nasceu para atender demandas voltadas para o *front-end*, e, por esse motivo, ela foi pensada com o intuito de ser rápida, dinâmica e acessível. Para a linguagem ser utilizada exclusivamente em uma aplicação *back-end*, foi necessário a utilização do *Node.JS*. O *Node* é um *runtime*, que nada mais é do que um conjunto de códigos/API, ou seja, são bibliotecas responsáveis pelo tempo de execução que funciona como um interpretador de *JavaScript* fora do ambiente do navegador *web* (GUTMAN, 2019).

Nessa primeira versão, o SGBD compatível com a aplicação escolhido é o MySQL, onde, de início, foram tratados os seguintes tipos de dados: *STRING*, *DATETIME*, *INT* e *FLOAT*. Como se trata de uma API, ela pode ser utilizada fazendo uma requisição para a URL em que o serviço encontra-se hospedado, adicionando ao fim da URL a rota desejada (Ex.: https://minhaurl.com/encrypt). É possível acessar o código da aplicação acessando o link[1] do repositório no Github disponibilizado no rodapé desta página.

Para realizar a ofuscação da data, foi adotada uma estratégia que utiliza-se do conceito de *timestamp*. O *timestamp* é uma representação de uma hora e/ou data em função de uma data referencial. Muito utilizado em computação, o *timestamp* representa a diferença da data de interesse em relação a data padrão 01/01/1970, ou seja, quantos segundos se passaram desde a data referência até a data interessada. Essa abordagem foi adotada ao perceber inconsistências na ofuscação da data presente no *Integrity*, que, em diversos casos, não retornava a data correta devido à limitação do banco de dados,

[1] `https://github.com/Juanvictor0/aes-security-api`

que é capaz de armazenar datas entre 01/01/1000 até 31/12/9999.

5.1- Estrutura do Código

Um dos pontos essenciais do processo de construção da estrutura do projeto é a arquitetura de arquivos e pastas utilizada. O padrão de projeto utilizado foi a *Clean Architecture*.

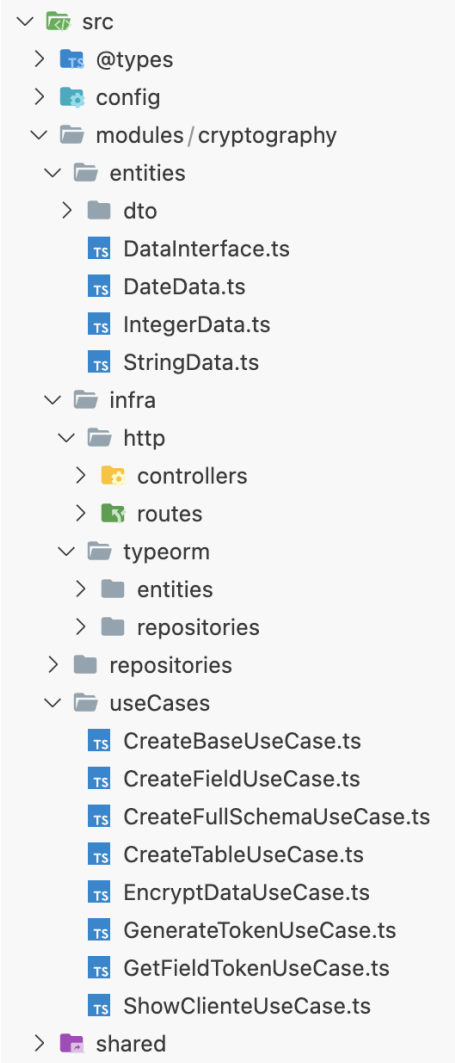


Figura 11 – Arquivos do sistema implementado organizados conforme a arquitetura *Clean Architecture*

Como visto na Figura 11, a parte principal do sistema está localizada na pasta *cryptography*. A camada de entidade da aplicação é a *entities*. Na segunda camada,

useCases, está toda a lógica de cadastro e listagem das tabelas, geração de *tokens* e a criptografia e descriptografia em si, ou seja, toda a regra de negócios se encontra ali. Existe também a pasta *Infra* como a camada de interface, que é dividida em duas: http (controladores e rotas da aplicação) e *typeorm* (um ORM para Node.JS). No *typeorm* ainda há a divisão em duas pastas: *entities*, responsável por possuir as entidades das tabelas do banco de dados da aplicação, e os *repositories*, onde se situam as classes que implementam os métodos que fazem a conexão com o banco de dados (criar, listar, editar, excluir).

A pasta *repositories*, fora da pasta de infra, diz respeito as interfaces criadas para os repositórios do banco de dados e, consequentemente, registrar a interface criada em um contêiner para que seja habilitada a possibilidade de que os repositórios sejam injetados como dependência para os *useCases* utilizados. Por último, há a pasta de *shared*, que é responsável por compartilhar arquivos, classes e funções (métodos utilitários para a criptografia, classes para erros no sistema) e configurar as rotas mencionadas nas camadas anteriores para estarem disponíveis para uso, além de configurar o tratamento de exibição de erros, cachê, habilitar o *cors* e designar e publicar a aplicação para uma porta específica do domínio.

5.2- Estrutura do Banco de Dados

O banco de dados foi criado considerando-se a configuração única do usuário, como suas chaves criptografadas, o seu banco de dados, suas tabelas e todos os campos que ela possui. Na Figura 12 é possível ver o diagrama do modelo relacional da base de dados utilizada na API para cadastrar informações sobre os bancos e tabelas de um dado cliente.

A tabela de *migration* é a única tabela que foge desse modelo relacional, visto que ela serve apenas para monitorar em qual versão sua base de dados se encontra. A tabela *client* é onde será cadastrado as informações do nosso cliente, sendo essa uma chave, uma semente para criptografia e um *token* de autenticação que será utilizado para realizar as requisições posteriores. *Base* é a base de dados, *table* são todas as tabelas pertencentes aquela base que o cliente deseja criptografar. *field* são todos os campos

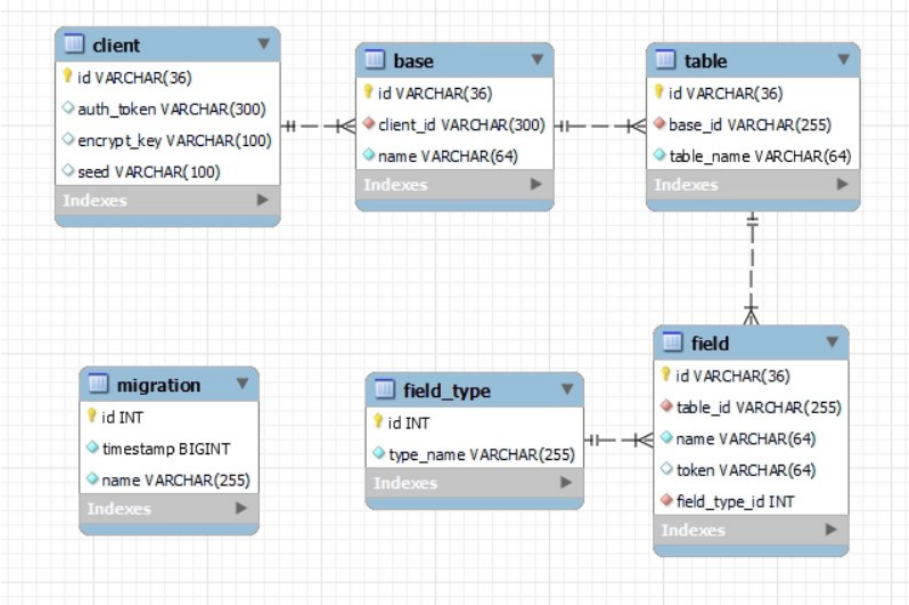


Figura 12 – Diagrama do modelo relacional do banco de dados do serviço de criptografia desenvolvido

pertencentes a uma tabela, sendo o campo de *token*, um *hash* gerado pelo serviço, ele é único para cada campo da tabela, e o *field type id* é o tipo de dado ao qual o campo pertence (*string*, *number* ou *date*).

5.3- Diagramas da Aplicação

A seguir, na Figura 13, é apresentado o diagrama de classes, que explicita os dados e funções possíveis de se utilizar no sistema, desde a criptografia em si, até a manipulação de usuários da API em seu banco de dados.

Na Figura 14, é possível visualizar o diagrama de sequência dos casos de criptografia e descriptografia. Nela observa-se o fluxo que o sistema segue ao ser solicitada uma operação criptográfica. O diagrama possibilita identificar o formato dos dados e funções mais importantes utilizadas no desenvolvimento.

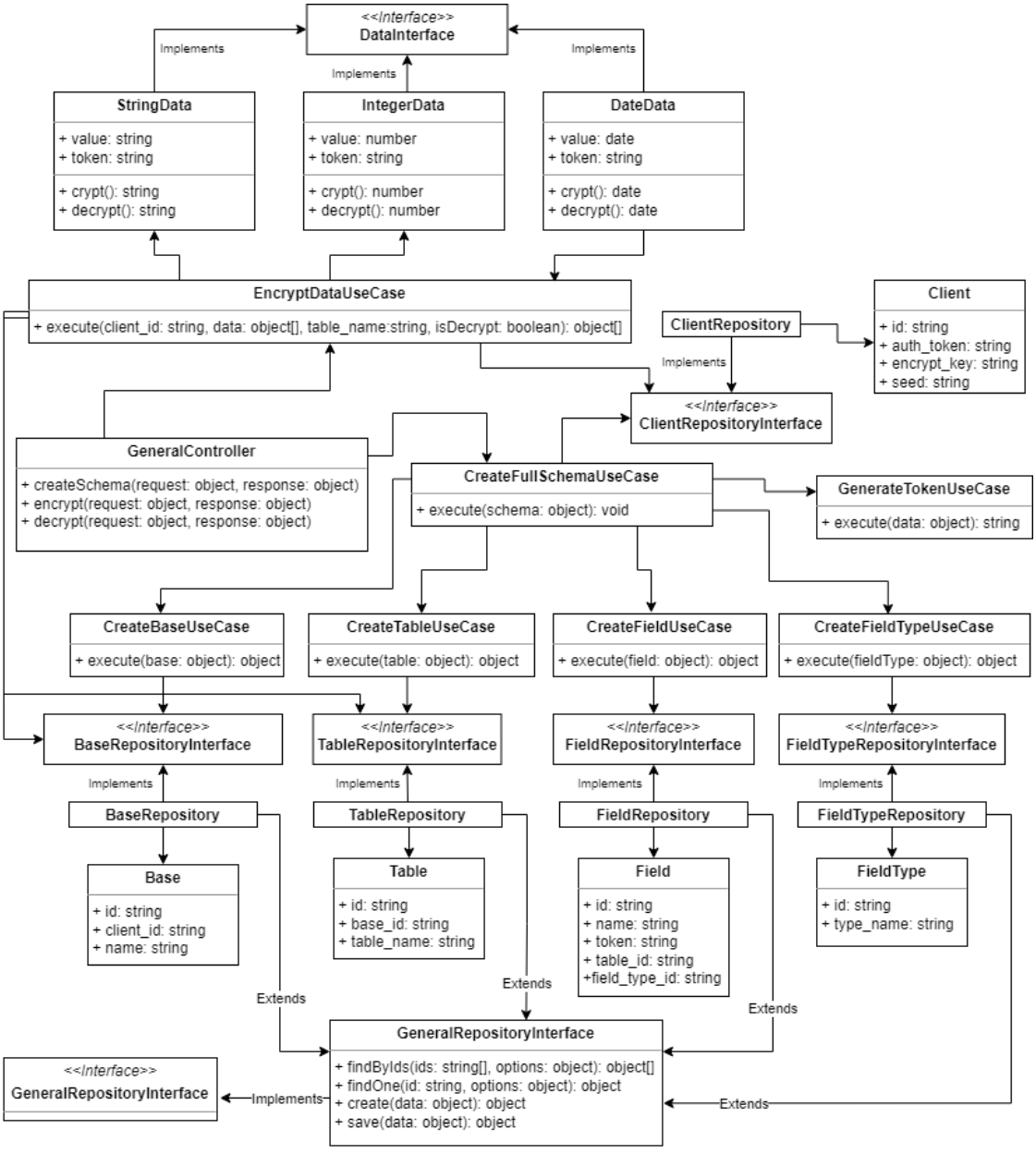


Figura 13 – Diagrama de classes do serviço de criptografia desenvolvido

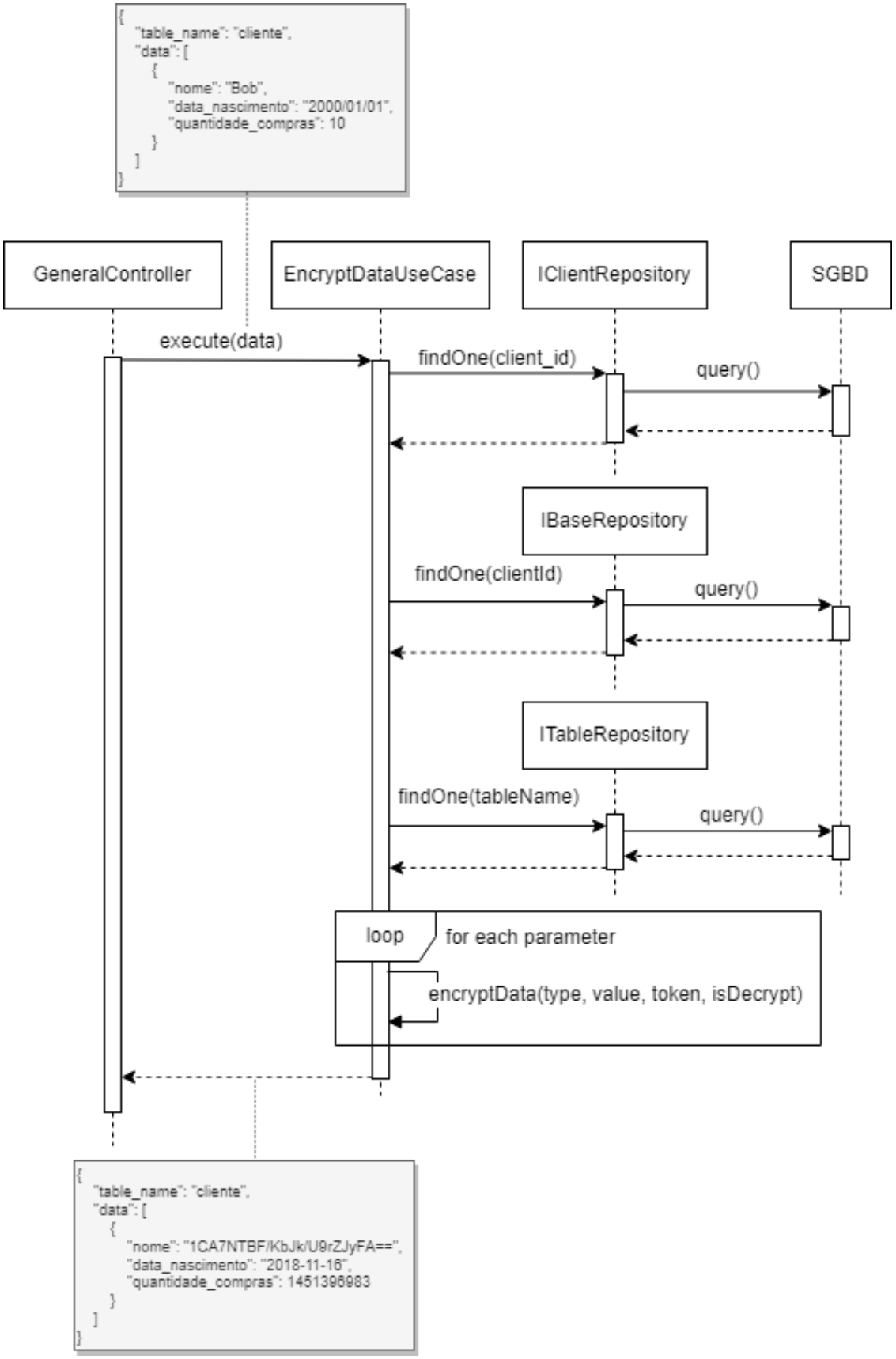


Figura 14 – Diagrama de sequência do serviço de criptografia desenvolvido contendo os fluxos de criptografia e descriptografia

6- Prova de Conceito

A presente seção tem por objetivo apresentar a prova de conceito do objeto de estudo deste documento, isto é, a demonstração de uma aplicação que utilize a API desenvolvida.

Para isso, foi implementada uma aplicação em *Node.js* que simula um servidor que dispõe uma rota */cliente* que oferece as quatro operações CRUD, isto é, criar, ler, atualizar e deletar um cliente. O servidor, denominado na Figura 15 de Aplicação Cliente, recebe uma requisição HTTP, realiza uma comunicação com a API, onde são realizadas as operações de segurança e, posteriormente, os dados são persistidos no Banco de dados cliente.

Figura 15 – Esquema da aplicação que consome a API proposta

Instruções para instalação e reprodução do exemplo estão disponíveis na subseção reprodutibilidade.

O banco de dados utilizado foi o *MySQL*. Foi criada uma base de dados chamada *aesencrypt_cliente* para simular o CRUD de um cliente qualquer, capaz de armazenar os campos *id* do tipo *varchar*, nome do tipo *varchar*, data_nascimento do tipo *date* e quantidade_compras do tipo *int*, conforme representado na Figura 16.

6.1- Configuração da aplicação que utiliza a API como serviço

Inicialmente, parte-se da premissa de que a aplicação que faz uso da API de segurança já possui um usuário devidamente cadastrado e o esquema da base de dados

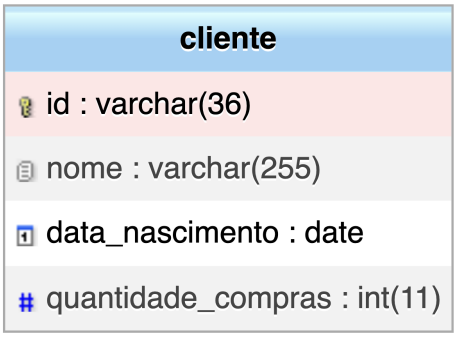


Figura 16 – Esquema da tabela cliente no banco de dados *aesencrypt_cliente* para reprodução da prova de conceito

aesencrypt_cliente já foi criado.

Para compreender o momento em que a aplicação cliente realiza uma comunicação com a API é necessário entender antes de onde vem esse objeto de comunicação. Como é possível observar na Figura 17, a aplicação instancia um objeto de configuração da conexão entre o sistema cliente e o serviço de criptografia e atribui a uma variável chamada *api*. Esse objeto é responsável por guardar informações como a URL alvo da requisição HTTP que será realizada e também métodos para realizar essa requisição.

```
1  import axios from "axios";
2
3  export const api = axios.create({
4    baseURL: process.env.BASE_URL_API_SERVER,
5    headers: {
6      Authorization: `Bearer ${process.env.TOKEN_KEY_AUTH}`,
7      "Content-Type": "application/json",
8    },
9  });
```

Figura 17 – Arquivo api.ts. Declaração da variável utilizada para a comunicação do sistema cliente e a API

6.2- Operações CRUD

A seguir são apresentados detalhes da implementação e interação do sistema desenvolvido. São abordadas as quatro operações CRUD e, todas, utilizam a API proposta

como serviço para ofuscar os dados sensíveis de um cliente.

6.2.1 Criação de um registro de cliente

Para criar um registro de um dado cliente, é realizada uma requisição HTTP do tipo *POST* com o corpo contendo os dados apresentados na imagem abaixo para a porta na qual a aplicação está escutando, definida no arquivo de configuração *TypeScript* demonstrado na Figura 18.

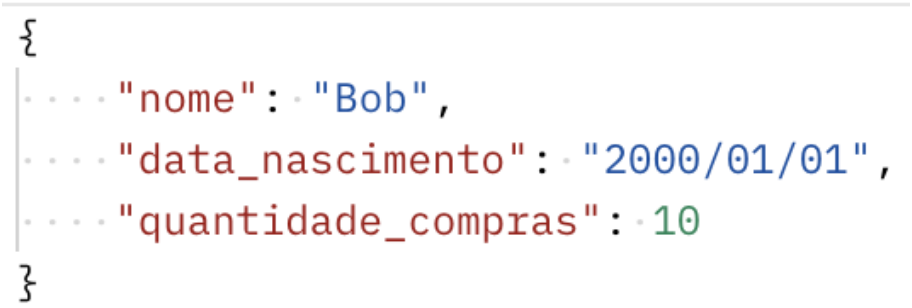


Figura 18 – Dados enviados na requisição para a aplicação que utiliza a API como serviço

Os dados recebidos pela aplicação são enviados como parâmetros para a API, retornam cifrados e, agora protegidos, são inseridos no banco de dados.

```
6   import { api } from '@shared/services/api';
30      const service = await api.post('encrypt', { table_name: 'cliente', data: [data] });
31
32      const dataEncrypted: Request = service.data[0];
33
34      const cliente = await this.clienteRepository.create({
35        nome: dataEncrypted.nome,
36        data_nascimento: dataEncrypted.data_nascimento,
37        quantidade_compras: dataEncrypted.quantidade_compras,
38      });
39
40      return cliente;
```

Figura 19 – Arquivo CreateClienteUseCase.ts. Código onde é realizada a comunicação entre aplicação/API para inserir um dado no banco de dados

Após a cifragem, observada na linha 30 da Figura 24, a aplicação abre uma conexão com o banco de dados utilizando a variável *clienteRepository* e os dados registrados são os seguintes, mostrados na Figura 22.

id	nome	data_nascimento	quantidade_compras
ae940d4d-8cca-4a97-9a1e-d077f4b5197d	1CA7NTBF/KbJk/U9rZJyFA==	2018-11-17	1451396983

Figura 20 – Dados ofuscados persistidos no banco de dados

6.2.2 Leitura de todos os registros de clientes

Para realizar uma leitura de todos os dados registrados no banco de dados, é feita uma requisição HTTP do tipo *GET* para a porta escutada. É retornada uma lista com os clientes do banco de dados, convertidos em objetos capazes de serem recebidos pela API, e então enviados a ela por meio de uma requisição para a rota */decrypt*. É retornado um cliente decifrado e é armazenado na lista de clientes, conforme a Figura 21.

```
2   import { api } from '@shared/services/api';
18      const clientes = await this.clienteRepository.findAll();
22      for (const cliente of clientes) {
23        const data = {
24          ...cliente,
25          nome: cliente.nome.toString(),
26          data_nascimento: moment.utc(cliente.data_nascimento).format('YYYY-MM-DD'),
27          quantidade_compras: cliente.quantidade_compras,
28        };
29
30        const service = await api.post('decrypt', { table_name: 'cliente', data: [data] });
31
32        const clienteDecrypted = service.data[0];
33
34        clientes[i] = clienteDecrypted;
35        i++;
36      }
45      return clientes;
```

Figura 21 – Arquivo IndexClienteUseCase.ts. Código onde é realizada a comunicação entre aplicação/API para realizar uma consulta no banco de dados

Tem-se como resultado uma lista contendo todos os clientes armazenados como objetos JSON, já com todos os dados em claro, como é mostrado na Figura 22.

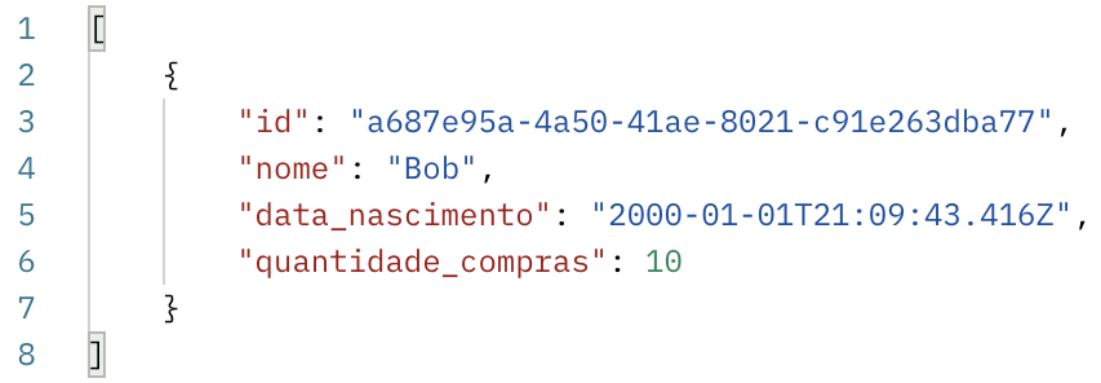


Figura 22 – Resposta da requisição GET

6.2.3 Atualização de um registro de cliente

Para atualizar os dados de algum cliente, é preciso fazer uma requisição *PUT* para o servidor e informar o id e os campos do cliente a serem alterados em um objeto chamado *data*. Nesse caso, todos os campos foram alterados, porém é válido mencionar que isso não é uma obrigatoriedade, ou seja, é possível alterar apenas um ou todos os dados. Os novos valores estão dispostos conforme a Figura 23.

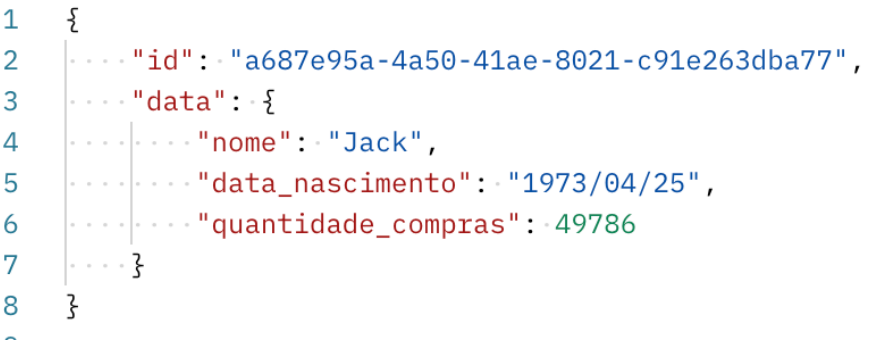


Figura 23 – Dados enviados na requisição para a aplicação

Semelhantemente à inserção, a atualização também faz uma requisição para a API enviando os dados recebidos pela aplicação. Após isso, o objeto do cliente é recuperado do banco de dados através do seu *id*, como observado na linha 33 da Figura 24, e, então, atualizado com os novos valores presentes no objeto *dataEncrypted*. Logo depois, ele é persistido no banco de dados.

Finalizado este processo, é possível observar seu resultado na Figura 25, atentando-

```
6   import { api } from '@shared/services/api';
30              const service = await api.post('encrypt', { table_name: 'cliente', data: [data] });
31              const dataEncrypted: Request = service.data[0];
32
33              const clientToUpdate = await this.clienteRepository.findOne(id);
34              if (!clientToUpdate) throw new Error('Client not found');
35
36              for (const key in dataEncrypted) {
37                  clientToUpdate[key] = dataEncrypted[key];
38              }
39
40              const cliente = await this.clienteRepository.save(clientToUpdate);
41
42              return cliente;
```

Figura 24 – Arquivo UpdateClienteUseCase.ts. Código onde é realizada a comunicação entre aplicação/API para atualizar um cliente no banco de dados

se à diferença entre os campos para os que foram inseridos na criação do cliente.

Antes

id	nome	data_nascimento	quantidade_compras
a687e95a-4a50-41ae-8021-c91e263dba77	1CA7NTBF/KbJk/U9rZJyFA==	2018-11-17	1451396983

Depois

id	nome	data_nascimento	quantidade_compras
a687e95a-4a50-41ae-8021-c91e263dba77	ovC+T2g9fOqHrLdlVRsQXg==	1992-03-11	1451446759

Figura 25 – Diferença entre dados persistidos no banco de dados

6.2.4 Deleção de registros de clientes em função de um parâmetro

Para a deleção, foi considerado um cenário onde o usuário necessite apagar todos os clientes cuja quantidade de compras for igual ao valor passado por parâmetro. A ideia pode ser reproduzida com todos e qualquer campo do objeto, que devem estar dentro do objeto *data*. Para isso, é necessária uma requisição HTTP do tipo *DELETE*.

Os dados da requisição são recebidos pela aplicação, enviados à API, descriptografados e, ao retornar, são repassados ao método que se comunica com o banco de dados, que os utiliza como filtro para obter os clientes desejados e então deletá-los.

O esquema dos dados enviados pela requisição e local do código onde é realizada a integração são observados nas figuras 26 e 27, respectivamente.

```
{
    "data": {
        "quantidade_compras": 49786
    }
}
```

Figura 26 – Dados enviados no corpo da requisição

```
import { api } from '@shared/services/api';
        try {
            const service = await api.post('encrypt', { table_name: 'cliente', data: [data] });
            const dataEncrypted: Request = service.data[0];

            const result = await this.clienteRepository.delete(dataEncrypted);

            return `${result.affected} client(s) deleted`;
```

Figura 27 – Arquivo DeleteClienteUseCase.ts. Código onde é realizada a comunicação entre aplicação/API para remover clientes no banco de dados

6.3- Reprodutibilidade da prova de conceito

Com o intuito de facilitar a reprodução da prova de conceito, foi disponibilizado um repositório[1] para obtenção e instrução sobre como realizá-la.

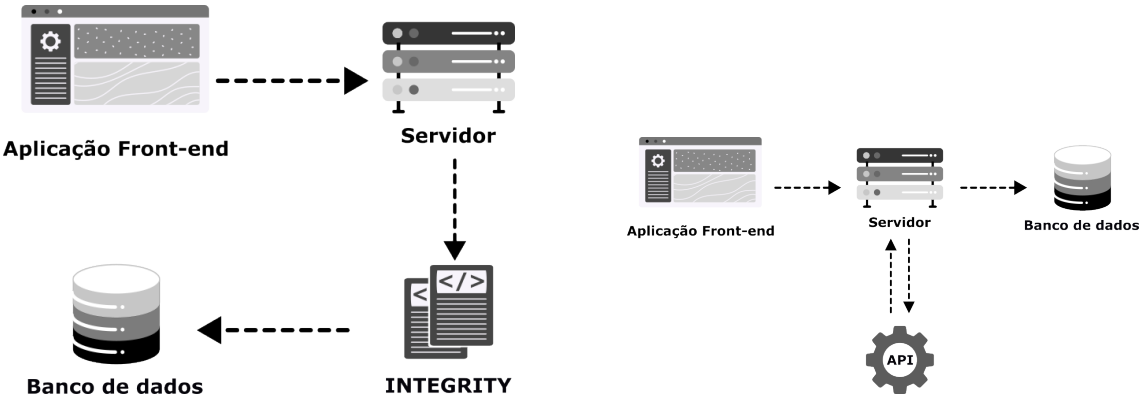
Para executar o projeto é necessário ter instalado na máquina os programas *yarn*, *node.js* além do banco de dados *MySQL*. O projeto tem por finalidade simular um sistema capaz de receber requisições HTTP e realizar operações de um objeto Cliente no banco de dados. Basta rodar os comandos para configuração inicial do projeto conforme disposto nas instruções e ele já estará pronto para execução.

[1] http://173.255.249.98/~integrityservice/

7- Experimentos

Nesta seção serão abordados os testes de desempenho, comparando quantitativamente a capacidade de um sistema utilizando o *Integrity Framework* e, alternativamente, esse mesmo sistema utilizando a API proposta no presente trabalho.

Os testes reproduzem repetidas inserções de registros em um banco de dados, simulando por exemplo uma conversão desses registros de dados em claro para dados criptografados. São contabilizados o tempo gasto com o processo criptográfico e também com as operações feitas no banco de dados, uma vez que o *Integrity Framework* realiza todo esse processo. Para isso, foram desenvolvidas pequenas aplicações *web* em C# capazes de simular uma série de inserções no banco de dados. Uma aplicação para medir o desempenho da inserção de dados utilizando o *Integrity* e outra usando a API proposta. Ambas as aplicações estão representadas nas figuras 28a e 28b, respectivamente.



(a) Integrity (b) API

Figura 28 – Esquema dos sistemas criados para criptografar dados e inserí-los no banco de dados

Os sistemas, ao iniciar sua página principal, realizam toda a configuração das ferramentas analisadas. Após, gera uma lista de dados que possuem a seguinte estrutura, conforme a Figura 29: um objeto C# do tipo *ClientEntity* que possui um campo do tipo *string* para representar o nome, outro do tipo *int* para representar a idade e, por fim, um campo do tipo *DateTime* representando o ano de nascimento. Esse objeto representa uma tupla, ou seja, um conjunto de dados que será persistido no banco de dados, portanto, uma tupla tem equivalência para um objeto. É importante salientar que esses dados são

apenas de teste para a execução dos métodos de criptografia, portanto não representam qualquer relação lógica entre si ou mesmo impedem qualquer inconsistência conceitual do ponto de vista do banco de dados.

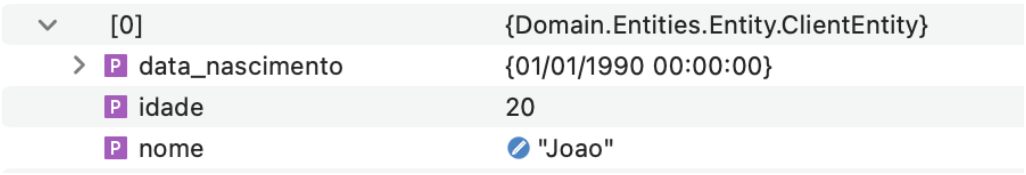


Figura 29 – Estrutura dos dados para teste de performance

O banco de dados utilizado para a realização dos testes foi o MySQL, onde foi criada uma base de dados, chamada *integrity_ex*, com uma tabela denominada cliente que comporta um campo para o nome do tipo *varchar* de até 256 caracteres, outro campo do tipo *int* para a idade e, por fim, o campo data_nascimento do tipo *date*.

Terminado o processo de geração dos dados, ocorre uma série de iterações sobre os mesmos. Cada objeto do tipo *ClientEntity* representa uma tupla, ou seja, um registro da tabela *cliente* no banco de dados. A quantidade de tuplas trabalhadas por cada rodada é definida por uma progressão aritmética onde o primeiro termo é 1024 aplicado a uma razão de 2, limitado a oitava posição da progressão. Isso resulta em uma série começando por 1024, 2048, 4096, em diante até chegar 262144, sendo essa a última posição definida. Pode-se representar também como 1K, 2K, etc. Para cada quantidade de tuplas, são realizadas 10 rodadas repetindo o mesmo processo, registrando o tempo individual, total e média para todas as repetições.

Os testes consideram as seguintes situações: a inserção no banco com os dados criptografados utilizando o *Integrity Framework* e a inserção no banco utilizando a API proposta, nesse caso, ora com os dados em claro, ora criptografados, para que seja observada a relevância da criptografia ao custo computacional de um processador.

7.1- Resultados

Após a realização dos testes, o sistema que utiliza a API confirmou a esperada melhora de performance. Dessa forma um serviço de criptografia dedicado mostra-se eficaz como uma estratégia viável, mesmo com seus pontos fracos discutidos anteriormente,

Quantidade de clientes	Tempo em segundos			Variação
	Integrity	API em claro	API cifrado	%
1K	1,42	0,99	1,32	7,04
2K	2,99	1,14	1,94	35,12
4K	5,79	2,23	3,8	34,37
8K	10,07	4,48	7,63	24,23
16K	19,96	8,95	15,28	23,45
32K	41,93	18	31,19	25,61
64K	84,79	35,97	60,66	28,46
128K	170,5	72	120,71	29,2
256K	335,14	143,47	239,95	28,4
Média da Variação				26,2

Tabela 2 – Comparação de desempenho de inserção de registros em banco de dados entre o *Integrity Framework* e API proposta

para fornecer segurança e velocidade independente da linguagem de programação ou banco de dados utilizados pela aplicação interessada. Os tempos para inserir dada quantidade de registros no banco de dados, obtido nos testes estão dispostos na Tabela 2.

Para ilustrar melhor o ganho percebido ao utilizar a API, tem-se a Figura 30, que apresenta os mesmos dados da tabela de forma visual. O gráfico permite perceber o ganho de performance em todos os casos de teste, obtendo resultados até 35% mais rápidos quando comparado ao *Integrity Framework*.

Isso ocorre devido a mudança de abordagem para atacar o problema. Ao utilizar um serviço exclusivo para a criptografia, uma série gastos computacionais podem ser evitados ao persistir os *tokens* utilizados na cifragem e recuperá-los posteriormente ao invés de gerá-los novamente, conforme fora discutido anteriormente.

Uma vez que a proposta visa principalmente a migração de bancos de dados já existentes para armazenar dados ofuscados sem comprometer a tipagem dos dados, a nova abordagem traz, também, capacidade de lidar com muitos dados de uma vez. Esse fato é de extrema relevância, pois oferece ao cliente uma liberdade para ofuscar todos os dados em apenas um passo (uma requisição) e definir a melhor estratégia para inserir os dados já ininteligíveis em sua base de dados, podendo até paralelizar esse processo,

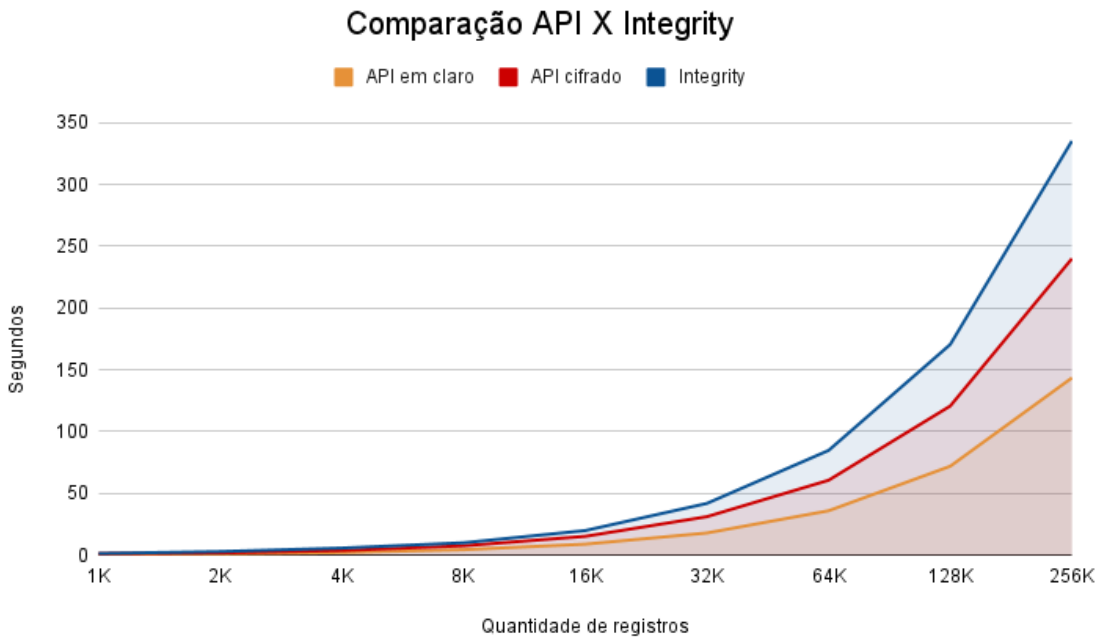


Figura 30 – Gráfico para comparação de desempenho entre o *Integrity* e a API proposta

caso o SGBD suporte tal funcionalidade.

7.1.1 Reprodutibilidade dos testes

Com o intuito de facilitar a reprodução dos testes, foi disponibilizado um repositório[1] para obtenção e instrução sobre como realizá-los.

Para a execução dos projetos é necessário ter instalado em máquina o *dotnet 5.0*, *yarn*, *node.js* e *MySQL*. Seguindo as instruções de configuração dos projetos, estarão disponíveis os três casos testados. A inserção de dados criptografados usando o *Integrity Framework*, outro usando a API cifrada e outro utilizando a API em claro. Estão respectivamente nas pastas integrity, api-cifrado e api-em-claro do arquivo disponibilizado no repositório. Escolhido o caso do teste, basta executar os comandos no terminal para iniciar o teste, que disponibilizará os resultados em tela ao final do processo.

[1] `http://173.255.249.98/~integrityservice/`

8- Conclusão

As novas regulamentações sobre a segurança da informação, como a LGPD, trouxeram uma iminente necessidade de adaptação, principalmente dos sistemas já existentes, a essas normas visando garantir a conformidade com a lei local e com a proteção dos usuários na *web*.

Esse trabalho apresentou uma nova abordagem para a ofuscação de dados proposta pelo *Integrity Framework*. A centralização do código responsável pela criptografia em um serviço comunicado por meio do protocolo HTTP busca solucionar problemas observados ao longo deste documento.

Tal abordagem tem por objetivo facilitar a integração com aplicações clientes. Enquanto o *Integrity Framework* é adicionado a projetos em forma de biblioteca que exige uma reimplementação para cada linguagem de programação diferente, a API proposta normaliza essa comunicação mediante um protocolo amplamente utilizado e suportado pela maioria das linguagens e aplicações já existentes.

O serviço permite, também, uma melhor manipulação dos dados em grande escala, sendo necessária apenas uma requisição, e oferece ao usuário mais liberdade para escolher a estratégia para a persistência dos dados, encapsulando a função de criptografia e removendo essa responsabilidade da aplicação cliente.

Outra contribuição presente neste, é a eliminação de redundâncias no processo de criptografia, garantindo um melhor desempenho e maior aproveitamento do poder de processamento da máquina que roda a aplicação, conforme constatado na seção de experimentos.

Além dos pontos citados acima, ainda foi adicionado ao escopo do trabalho a ofuscação do tipo inteiro e do tipo ponto flutuante.

Para trabalhos futuros, é possível aumentar os tipos de dados suportados, a realização de consultas mais complexas sem a necessidade de descriptografar toda a tabela, discussões sobre limites e barreiras a serem ultrapassadas pela abordagem FPE.

Referências

AUMASSON, Jean Philippe. **Serious Cryptography: A Practical Introduction to Modern Encryption**. California: No Starch Press, 2017. 748 p.

BELLARE, Mihir et al. Format-Preserving Encryption. In_____. **Selected Areas in Cryptography**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 295–312. ISBN 978-3-642-05445-7.

BHATT, Zalak; GUPTA, Vinitkumar. Format Preserving Encryption: A Survey. **INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH IN TECHNOLOGY**, v. 2, n. 10, p. 8–17, 2016. ISSN 2349-6002.

BRASIL. **Acesso a Informação - Lei Geral de Proteção de Dados Pessoais (LGPD)**. pt-br. Brasília: Brasil, 2018. Disponível em: https://www.gov.br/cidadania/pt-br/acesso-a-informacao/lgpd/lei-geral-de-protecao-de-dados-pessoais-lgpd. Acesso em: 5 de mar. de 2022.

_____. **LEI Nº 13.709, DE 14 DE AGOSTO DE 2018**. pt-BR. Brasília: Diário Oficial da União, 2018.

BRIGHTWELL, Michael et al. Using Datatype-Preserving Encryption to Enhance Data Warehouse Security. National Institute of Standards e Technology, v. 1, out. 1997.

CORACCINI, Raphael. **O mundo já registra 4,6 bilhões de dados vazados em 2021, diz PSafe**. pt-BR. São Paulo, Brasil: CNN Brasil, jul. 2021. Disponível em: https://www.cnnbrasil.com.br/business/o-mundo-ja-registra-4-6-bilhoes-de-dados-vazados-em-2021-diz-psafe/. Acesso em: 15 de ago. de 2022.

COSTA, Elder et al. Integrity: An Object-relational Framework for Data Security: in: PROCEEDINGS of the 24th International Conference on Enterprise Information Systems. Online: SCITEPRESS - Science e Technology Publications, 2022. p. 259–266. ISBN 978-989-758-569-2. DOI: 10.5220/0011088100003179.

DWORKIN, Morris et al. **Advanced Encryption Standard (AES)**. en. US: Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards e Technology, Gaithersburg, MD, 2001-11-26 2001. DOI: https://doi.org/10.6028/NIST.FIPS.197.

GUTMAN, David. **Fullstack Node.js: The Complete Guide to Building Production Apps with Node.js**. Texas: Fullstack.io, 2019. 258 p.

IBM SECURITY. **Cost of a Data Breach Report 2020**. en. US, 2020. p. 82. Disponível em: https://www.ibm.com/security/digital-assets/cost-data-breach-report/1Cost%20of%20a%20Data%20Breach%20Report%202020.pdf.

MARTIN, Robert C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. US: Prentice Hall, 2017. (Robert C. Martin Series). ISBN 978-0-13-449416-6.

NETO, Nelson Novaes et al. Developing a Global Data Breach Database and the Challenges Encountered. en. **Journal of Data and Information Quality**, v. 13, n. 1, p. 1–33, jan. 2021. ISSN 1936-1955, 1936-1963. DOI: 10.1145/3439873.

NETO, Pedro; ARAÚJO, Wagner. **Segurança da Informação Uma Visão Sistêmica para Implantação em Organizações**. João Pessoa: Editora UFPB, 2019. v. 1. 160 p.

PITTA, Paulo E. B. et al. LGPD Compliance: A security persistence data layer. In: ANAIS da XVIII Escola Regional de Redes de Computadores (ERRC 2020). Brasil: Sociedade Brasileira de Computação - SBC, nov. 2020. p. 123–127. DOI: 10.5753/errc.2020.15200.

PUNDLIK, Sumitra. DBCrypto: A Database Encryption System using Query Level Approach. **Journal of Network and Computer Applications**, v. 45, 1 mai. 2012.

SERPRO. **O que são dados pessoais, segundo a LGPD**. pt-br. Brasília: SERPRO, 2018. Disponível em: https://www.serpro.gov.br/lgpd/menu/protecao-de-dados/dados-pessoais-lgpd. Acesso em: 17 de ago. de 2023.

STALLINGS, William. **Cryptography and network security: principles and practice**. Seventh edition. Boston: Pearson, 2017. 748 p. ISBN 978-0-13-444428-4.

STINSON, D.R.; PATERSON, M.B. **Cryptography: Theory and Practice**. Florida: CRC Press, 2019. (Discrete mathematics and its applications). ISBN 9781138197015.

VALENTE, Marco Tulio. **Engenharia de Software Moderna: princípios e práticas para o desenvolvimento de um software com produtividade**. Brasil: Editora: Independente, 2022. 408 p.

ZHANG, Mingming et al. DMSD-FPE: Data Masking System for Database Based on Format-Preserving Encryption. In: WAN, Jiafu et al. (Ed.). **Cloud Computing, Security, Privacy in New Computing Environments**. Cham: Springer International Publishing, 2018. v. 197. Series Title: Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. p. 216–226. ISBN 978-3-319-69604-1 978-3-319-69605-8. DOI: 10.1007/978-3-319-69605-8_20.