Privacy Through Homomorphic Encryption

Segurança e Privacidade • 2025/2026

Relatório

Elementos do Grupo

Daniel Pereira • 2021237092 • uc2021237092@student.uc.pt

Vasco Fernandes • 2025186614 • uc2025186614@student.uc.pt

Índice

1. Introdução

A crescente digitalização dos processos empresariais e a necessidade de partilha de informações entre diferentes entidades a proteção dos dados um tema cada vez mais relevante. Em diferentes contextos, as organizações necessitam de recorrer a serviços externos para realizar análises estatísticas ou operações computacionais sobre os seus dados, sem, contudo, poderem expor a informação sensível neles contida. Diante desde cenário, coloca-se o desafio de permitir que terceiros processem dados privados sem nunca terem acesso ao seu conteúdo.

A tecnologia de Fully Homomorphic Encryption (FHE) oferece uma solução para este problema, ao possibilitar a realização de operações matemáticas diretamente sobre dados encriptados. Dessa forma, o responsável pela análise pode executar cálculos sem acesso aos valores originais, assegurando a confidencialidade dos dados durante todo o processo.

O presente projeto tem como objetivo explorar na prática o uso de FHE, implementando um sistema completo composto por duas entidades: o Data Holder, que detém os dados, os encripta e os desencripta, e o Data Analyzer, que recebe os dados encriptados e realiza operações estatísticas sobre eles. Para este projeto será definido um cenário concreto, selecionado um dataset adequado e implementados dois diferentes esquemas de FHE, permitindo comparar o desempenho entre as alternativas escolhidas.

2. Fundamentos Teóricos

A **Encriptação Homomórfica (HE)** é um tipo de criptografia que permite realizar operações matemáticas diretamente sobre dados encriptados, **sem necessidade de acesso ao plaintext**. O resultado da operação homomórfica, quando desencriptado, é **equivalente ao resultado da mesma operação aplicada aos dados originais**. A explicação apresentada segue a definição de Encriptação Homomórfica proposta por By Yi, X., Paulet, R., & Bertino, E. (2014).

Formalmente, para uma função matemática *f*, e para dados *m*, um esquema HE satisfaz:

$$\mathbf{Eval(f,Enc(m))=Enc(f(m))}$$

Existem três categorias principais:

- **PHE — Partially Homomorphic Encryption:** suporta apenas uma operação (ex.: Paillier é aditivo; RSA é multiplicativo).
- **SHE — Somewhat Homomorphic Encryption:** suporta adição e multiplicação, mas com profundidade limitada.
- **FHE — Fully Homomorphic Encryption:** suporta adição e multiplicação arbitrárias, permitindo computação geral sobre ciphertexts.

Neste projeto são utilizados **esquemas FHE modernos**, baseados em Ring Learning With Erros (RLWE), que pertencem à linha evolutiva dos esquemas introduzidos por Gentry (2009).

3. Descrição do cenário e dataset

3.1. Descrição do cenário

Neste projeto, vamos considerar um cenário do domínio da saúde, em que um hospital deseja efetuar análises estatísticas nos dados dos seus pacientes, sem comprometer a sua privacidade individual. Mais particularmente, o hospital precisa avaliar a distribuição da pressão arterial sistólica dos pacientes, importante indicador vital para diagnosticar doenças cardiovasculares como a hipertensão e a hipotensão.

Contudo, por razões legais e éticas — nomeadamente para cumprimento do Regulamento Geral de Proteção de Dados (RGPD) — o hospital não pode enviar diretamente a um serviço externo os valores reais da pressão arterial dos seus pacientes. Assim, recorre-se a Fully Homomorphic Encryption (FHE), permitindo que o analista receba apenas dados encriptados, realize cálculos sobre eles e devolva resultados igualmente encriptados, nunca tendo acesso à informação sensível.

3.2. Entidades envolvidas

Data Holder (Hospital)

Possui os dados originais dos pacientes, encripta-os utilizando um esquema de FHE e envia-os ao analista. Após a análise, recebe os resultados encriptados e procede à sua desencriptação.

Data Analyzer (Serviço externo)

Recebe os valores encriptados e aplica operações estatísticas tais como cálculo da média e soma total e depois devolve os resultados de forma igualmente encriptada.

3.3. Descrição do dataset

Foi criado um dataset sintético composto por 5000 registos , garantindo privacidade total e evitando o uso de dados reais. Cada registo contém:

- **Nome do paciente** (gerado combinando nomes e apelidos portugueses comuns)
- **Pressão arterial sistólica** (medida em mmHg)

A distribuição dos valores foi construída de forma realista, respeitando intervalos compatíveis com a vida humana e refletindo a prevalência clínica típica:

- **70%** dos valores situam-se entre **110 e 160 mmHg**, representando níveis normais ou moderadamente elevados.

- **25%** encontram-se entre **90 e 200 mmHg**, abrangendo valores baixos ou hipertensões mais significativas.
- **5%** representam extremos clínicos, **70 a 250 mmHg**, que embora raros, são possíveis em situações críticas.

Nome	Pressão Arterial (mmHg)
Carla Fernandes	126
Sónia Duarte	161
Daniela Pinho	139
Carla Dinis	131
Ana Pinho	156
Ricardo Rocha	130
Vera Lopes	156
Isabel Ribeiro	138
Diana Rocha	159
Rafael Figueiredo	113
Élia Serrano	130
Sónia Simões	92
Diana Aragão	131
Irene Gouveia	141
Érica Fernandes	102
Rui Pereira	144
Matilde Ramos	121
Patrícia Figueiredo	112
José Silveira	153
Marco Machado	159
Marisa Cabral	140
Carla Barros	125
Pedro Paiva	119
Hugo Monteiro	159
Raquel Tavares	114
Diana Barata	135
Fernando Pereira	104
Hélder Pereira	164
Joel Carvalho	158
Alice Dinis	154
Pedro Costa	102
Nuno Brito	136
Ricardo Neves	136

Tabela 1 Excerto do dataset

4. Esquemas escolhidos

O sistema desenvolvido baseia-se em duas entidades principais: **Data Holder** e **Data Analyzer**, que comunicam exclusivamente através de dados encriptados. O objetivo é permitir a execução de operações estatísticas sobre a pressão arterial dos 5000 pacientes de um certo hospital, garantindo que o serviço responsável pela análise nunca tenha acesso aos valores reais.

Foram implementados dois esquemas de Fully Homomorphic Encryption (FHE) distintos, ambos através da biblioteca TenSEAL, uma biblioteca Python que fornece implementações otimizadas de esquemas FHE.

Outras bibliotecas como PHE ou Pyfhel foram consideradas, mas TenSEAL oferece suporte nativo aos dois esquemas FHE requeridos (CKKS e BFV) e integra bem operações vetoriais, fundamentais para este trabalho.

4.1. CKKS (Cheon-Kim-Kim-Song)

Esquema adequado para a realização de **operações de números reais com precisão aproximada**. O CKKS trabalha nativamente com representações em ponto flutuante, tornando-o ideal para aplicações de análise estatística onde se aceitam pequenas margens de erro.

4.2. BFV (Brakerski-Fan-Vercauteren)

Esquema que permite **operações exatas sobre números inteiros**. Ao contrário do CKKS, o BFV garante precisão total nos resultados, trabalhando num espaço aritmético modular. É particularmente adequado para aplicações que requerem contagens exatas.

5. Implementação dos Esquemas

A implementação foi organizada de forma clara, separando as responsabilidades entre o Data Holder e o Data Analyzer. Ao todo, foram desenvolvidos quatro módulos principais, dois correspondentes ao esquema CKKS e dois ao esquema BFV.

5.1. O Contexto Criptográfico e Parâmetros

Em esquemas de Fully Homomorphic Encryption, o contexto é a estrutura fundamental que define todo o ambiente criptográfico do sistema. Este contém:

- Parâmetros de segurança
- Tipos de esquemas
- Chaves criptográficas
- Configurações específicas

A configuração do contexto exige a definição cuidadosa de múltiplos parâmetros criptográficos. Para garantir uma comparação válida e justa entre ambos os esquemas, foram aplicados parâmetros idênticos sempre que possível:

PARÂMETRO	VALOR	JUSTIFICAÇÃO
POLY_MODULUS_DEGREE	16384	Fornece ~128 bits de segurança, equivalente ao padrão AES-128, adequado para aplicações reais de saúde
COEFF_MOD_BIT_SIZES	[60, 40, 40, 60]	Define a profundidade multiplicativa suportada (número de operações consecutivas permitidas)

Em ambos os casos usamos **Galois keys** que são chaves criptográficas especiais que permitem realizar **operações avançadas** sobre dados encriptados, nomeadamente a operação .sum(). Internamente, o algoritmo roda o vetor múltiplas vezes, soma todas as versões rodadas e extrai o resultado final.

Tanto no **CKKS** como no **BFV**, usamos **.sum()** para calcular a soma total das 5000 pressões arteriais **sem desencriptar os dados individuais**. As Galois keys são, portanto, **essenciais** para esta operação funcionar.

Parâmetros Específicos do CKKS

```
inicio = time.time()
context = ts.context(
    ts.SCHEME_TYPE.CKKS,
    poly_modulus_degree = 16384,
    coeff_mod_bit_sizes = [60, 40, 40, 60]
)

context.generate_galois_keys()
context.global_scale = 2**40
```

Figura 1- Contexto do CKKS

O valor escolhido para o **global_scale** reflete a precisão dos dados em ponto flutuante. Valores maiores aumentam a precisão, mas reduzem o espaço disponível para operações. Este valor define um bom compromisso para cálculos estatísticos em ambientes médicos.

Parâmetros Específicos do BFV

```
inicio = time.time()

SCALE_FACTOR = 2**20

context = ts.context(
        ts.SCHEME_TYPE.BFV,
        poly_modulus_degree=16384,
        coeff_mod_bit_sizes=[60, 40, 40, 60],
        plain_modulus=1099511922689
    )
context.generate_galois_keys()
```

Figura 2- Contexto do BFV

O campo **plain_modulus** é um parâmetro específico do BFV em que é definido o limite do espaço aritmético onde os cálculos inteiros ocorrem. É um número primo enorme com o propósito de evitar overflow nos cálculos e ser compatível com técnicas de batching.

5.2. Fluxo de Operações

5.2.1. CKKS

Fase 1 - Geração de chaves

Em primeiro lugar o Data Holder cria o contexto criptográfico e o par de chaves pública e privada. A chave privada é guardada no ficheiro secret_ckks.txt contendo todos os parâmetros do contexto e é apenas usada pela Data Holder para encriptar e desencriptar

os resultados. Já a chave pública contém informações como Galois Keys e é partilhada com o Data Analyzer para que ele possa trabalhar com os dados encriptados.

Fase 2 - Encriptação

Durante a fase de encriptação, o Data Holder lê o dataset do ficheiro Excel, extraindo os 5000 valores para um array. Calcula o inverso do tamanho do dataset (inv_N = 1/5000 = 0.0002) valor necessário para cálculo posterior da média através de multiplicação homomórfica. Procede então à encriptação de ambos os elementos: o vetor completo de pressões arteriais (pressao_encrypted) e o valor inv_N (inv_N_encrypted). Por fim, serializa ambos os ciphertexts para ficheiros binários na pasta outputs/, preparando-os para o envio ao Data Analyzer.

Fase 3 - Análise

Nesta fase, o Data Analyzer carrega o contexto público do ficheiro public_ckks.txt, que contém os parâmetros criptográficos e as Galois keys necessárias para realizar operações homomórficas, mas, como já tínhamos visto, não inclui a chave privada. De seguida, lê os ficheiros binários que contém os dados encriptados do vetor de 5000 pressões arteriais (pressão_encrypted_ckks.txt) e do inverso de N (inv_N_encrypted_ckks.txt), utilizando um carregamento lazy usado para otimizar memória, e liga ambos os ciphertexts ao contexto público carregado. Procede então à execução das operações homomórficas: calcula a soma total através do método .sum(), que utiliza as Galois Keys para agregar todos os valores encriptados num único ciphertexts, e de seguida calcula a média multiplicando a soma pelo inverso de N encriptado:

$$\textbf{mean\_encrypted} = \textbf{sum\_encrypted} \times \textbf{inv\_N\_encrypted}$$

Finalmente, serializa ambos os resultados encriptados para ficheiros binários na pasta outputs/, prontos para devolução ao Data Holder. Todo este processamento ocorre de forma completamente cega, onde o Analyzer nunca tem acesso aos valores reais, garantindo privacidade total.

Fase 4 - Desencriptação

Nesta fase final, o Data Holder recupera o contexto com a chave privada, essencial para desencriptar os resultados. Carrega os ciphertexts da média e da soma e liga-os ao contexto privado para os desencriptar usando o método .decrypt(), que aplica a chave privada para revelar os valores em plaintext.

Esta fase completa o ciclo FHE, permitindo ao Data Holder recuperar as estatísticas calculadas de forma segura sobre dados que permaneceram encriptados durante todo o processamento externo.

Resumo dos ficheiros gerados

- **secret_ckks.txt:** Contexto com chave privada (mantido pelo Holder)
- **public_ckks.txt:** Contexto público com Galois keys (enviado ao Analyzer)

- **pressao_encrypted_ckks.txt:** 5000 valores de pressão encriptados
- **inv_N_encrypted_ckks.txt:** Inverso de N (1/5000 = 0.0002) encriptado

5.2.2. BFV

O fluxo de operações do esquema BFV segue a mesma arquitetura de quatro fases descrita para o CKKS, mantendo a separação clara entre Data Holder e Data Analyzer assim como secret_bfv.txt e public_bfv.txt.

Contudo devido à natureza inteira do BFV, surgem diferenças técnicas importantes na implementação.

A principal distinção ocorre na **Fase 2 – Encriptação**, onde o BFV requer uma técnica de scaling para lidar com operações decimais. Uma vez que o esquema trabalha exclusivamente com aritmética de inteiros, o cálculo da média, que envolve divisão pelo tamanho, não pode ser realizado diretamente. Para contornar esta limitação, o Data Holder faz o seguinte:

- Definição de um **scale_factor** = 2^{20} = 1048576
- Cálculo de **inv_N_scaled** = int(2^{20}/5000) = 209

Em suma, encripta-se o valor inteiro 209 que é o inverso do tamanho escalado a 2^{20}, evitando assim operações com números não inteiros.

Na **Fase 4 – Desencriptação**, o Data Holder deve remover a escala aplicada através do seguinte cálculo:

média_final = média_escalada_decifrada / scale_factor

Resumo dos ficheiros gerados

- **secret_bfv.txt**: Contexto com chave privada
- **public_bfv.txt**: Contexto público com Galois keys
- **pressao_encrypted_bfv.txt**: 5000 valores encriptados
- **inv_N_encrypted_bfv.txt**: Valor 209 (inv_N_scaled) encriptado

5.3. Código e Ferramentas utilizadas

O código desenvolvido para este projeto encontra-se disponível na integra em anexo e foi implementado em Python 3.11.13. Todas as experiências relatadas neste relatório foram executadas no mesmo computador (MacBook M1) para garantir consistência e comparabilidade nos resultados de performance apresentados no ponto 6.

O desenvolvimento foi realizado em Jupyter Notebook para permitir uma maior flexibilidade na execução de excertos de código e melhor visualização dos respetivos resultados.

Bibliotecas usadas: TenSEAL, Pandas, Time, Garbage Collector.

6. Resultados

6.1. Performance

Para avaliar a performance de ambos os esquemas, foram realizados **cinco testes** no mesmo ambiente computacional, medindo separadamente cada fase do sistema FHE. Esta abordagem metodológica permite não apenas calcular tempos médios fiáveis, mas também avaliar a consistência e variabilidade de cada operação. Os resultados obtidos para ambos os esquemas são apresentados nas tabelas seguintes.

6.1.1. Esquema CKKS

A Tabela 2 apresenta os resultados em segundos para cada um dos cinco testes realizados com o esquema CKKS, e a respetiva média.

CKKS	Teste 1	Teste 2	Teste 3	Teste 4	Teste 5
keys	1,20718694	1,17340112	1,15603924	1,17294717	1,14124608
encrypt	0,97340512	1,12334514	0,89533091	0,97337389	0,870085
analyze	0,70755601	0,71863031	0,76207089	0,68897605	0,70648813
decrypt	0,562953	0,82042193	0,61665583	0,54277921	0,55957007
sum	3,45110106	3,8357985	3,43009686	3,37807631	3,27738929
		Média	3,47449241		

Tabela 2-Testes de Performance para o CKKS

Os resultados demostram uma boa consistência entre execuções apenas com variações naturais. A **geração de chaves** é a fase mais demorada (~1,17 segundos), refletindo a complexidade da geração de números primos grandes e parâmetros criptográficos. A **encriptação** mantém-se estável em torno de 0,97 segundos, um valor notavelmente baixo considerando que processa 5000 valores de pressão arterial e o inverso do tamanho. A fase de **análise**, em que é calculada a soma e a média dos 5000 registos encriptados, apresenta tempos consistentes de aproximadamente 0,71 segundos, demonstrando a eficiência das Galois Keys na realização de operações vetoriais complexas sobre os dados encriptados. A **desencriptação** é a fase mais rápida (~0,58 segundos), envolvendo apenas a aplicação da chave privada aos ciphertexts dos resultados.

O **tempo médio total** de 3,47 segundos para processar 5000 registos demostra que em termos de performance o CKKS é altamente viável aplicações práticas. Estes valores particularmente baixos devem-se em grande parte à utilização de **batching**, uma técnica que permite colocar os 5000 valores num único ciphertexts em vez de criar 5000 ciphertexts individuais. Esta otimização reduz drasticamente tanto o tempo de processamento como o consumo de memória, permitindo que operações como .sum() sejam executadas de forma vetorial sobre todos os valores simultaneamente.

6.1.1. Esquema BFV

A Tabela 3 apresenta os tempos de execução em segundos para os cinco testes realizados com o esquema BFV, e a respetiva média.

BFV	Teste 1	Teste 2	Teste 3	Teste 4	Teste 5
keys	1,18181396	1,27610993	1,20900297	1,23986602	1,21443605
encrypt	1,06295609	1,03119493	0,93061876	0,93749499	0,85803008
analyze	0,698771	0,72948813	0,73060775	0,75039101	0,88377523
decrypt	0,58392286	0,58600807	0,58316875	0,57978392	0,57281899
sum	3,52746391	3,62280107	3,45339823	3,50753593	3,52906036

Média	3,52779979

Tabela 3--Testes de Performance para o BFV

Os resultados de performance do BFV apresentam bastantes semelhanças aos do CKKS. A **geração de chaves** mantém-se como a fase mais demorada (~1,22 segundos), ligeiramente superior ao CKKS devido ao número primo enorme do plain_modulus. A **encriptação** apresenta tempos em torno de 0,94 segundos, processando os 5000 valores inteiros e o inv_N_scaled de forma eficiente através de batching. A fase da **análise** apresenta uma média temporal de 0,73 segundos refletindo que a complexidade computacional da aritmética modular inteira não afetou muito o resultado temporal. A **desencriptação** permanece a fase mais rápida (~0,58 segundos), praticamente idêntica ao CKKS.

O **tempo médio total foi de 3,53 segundos**, situando-se apenas **0,06 segundos acima do CKKS.** Isso demostra que ambos os esquemas apresentam a mesma performance temporal para este caso de estudo. Este resultado é particularmente interessante porque confirma que a **técnica de scaling implementada no BFV não introduz penalizações** significativas de performance. Tal como no CKKS**, os valores notavelmente baixos são possibilitados pela utilização de batching**, otimizando. Drasticamente o processamento de grandes volumes de dados encriptados.

6.2. Resultados da Análise

Antes de apresentar os resultados obtidos pelos esquemas FHE, é importante estabelecer os **valores esperados** calculados diretamente sobre o dataset original em plaintext: a soma total dos 5000 valores de pressão arterial é **693715 mmHg** e a média correspondente é **138,743 mmHg**. Estes valores servem como referência para avaliar a precisão de cada esquema de encriptação homomórfica.

6.2.1. Esquema CKKS

Após a execução completa do ciclo FHE o esquema CKKS produziu os seguintes valores estatísticos:

Soma Total	Média
693714,999987250	138,742849242362

Tabela 4-Resultados da análise com CKKS (mmHg)

Os resultados da análise com CKKS demostram uma excelente precisão considerando a natureza de aproximação do esquema CKKS. A **soma total** tem o valor de **693,714.999987250 mmHg** que difere do valor original apenas **0,0000000018%.** A **média** calculada de **138.742849242362 mmHg** representa um erro inferior a **0,0002%** do valor original.

Este comportamento aproximado relacionado com o design do CKKS, que trabalha com representações em ponto flutuante e introduz pequenos erros de arredondamento em cada operação homomórfica. Mas neste caso prático conseguimos erros bastante abaixo da variabilidade da precisão dos próprios dispositivos de medição que são tipicamente ± 1 mmHg (ISO 81060-2).

A precisão elevada também se deveu à escolha adequada do parâmetro **global_scale = 2^{40}, que reserva 40 bits para a parte fracionária dos números**, garantindo precisão suficiente.

6.2.2. Esquema BFV

Soma Total	Média
693715	138,269839286804

Tabela 5-Resultados da análise com BFV (mmHg)

Os resultados do BFV demostram um comportamento interessante e previsível. A **soma permanece matematicamente precisa**, batendo a 100% com o valor esperado. Uma das principais vantagens do BFV é a sua precisão com números inteiros, portanto mesmo encriptados os 5000 valores foram somados de maneiro totalmente exata.

No entanto a média calculada pelo BFV é aproximadamente 138,270 mmHg, o que indica uma diferença de 0,473 mmHg em relação ao valor esperado de 138,743 mmHg. Esse desvio é a consequência da técnica de scaling que tivemos que aplicar para permitir que o BFV executasse operações com decimais. É importante perceber que este **erro continua a ser clinicamente insignificante.**

O **BFV propõe um compromisso entre exatidão absoluta em somas e exatidão aproximada** (mas aceitável) em operações que exigem decimais.

7. Conclusões

Este projeto demonstrou que **FHE é completamente viável para utilização em ambientes hospitalares** em que é necessário processar dados médicos sensíveis mantendo a privacidade absoluta. Implementámos um sistema Data-Holder — Data-Analyzer onde o analista nunca teve acesso aos dados em plaintext. O Data Analyzer

executou operações homomórficas sobre ciphertexts, devolvendo resultados igualmente encriptados ao Data Holder, que detém a única chave capaz de revelar os valores finais desencriptados.

7.1. Comparação de Performance

A análise comparativa dos dois esquemas de encriptação homomórfica ao nível temporal revela que o desempenho obtido em todas as fases do processo (geração de chaves, encriptação, análise homomórfica e desencriptação) é bastante semelhante, com tempos médios totais de **3,47 segundos para CKKS** e **3,53 segundos para BFV**. Esta diferença de apenas **0,06 segundos (1,7%)** demonstra que, para os parâmetros utilizados e para o tamanho do dataset testado (5000 registos de pressão arterial), nenhum dos esquemas apresenta vantagem significativa em termos de tempo de execução total.

Conclusão sobre performance: Para o caso de uso testado (processamento de 5000 registos com operações de soma e média), **a performance foi um critério de decisão** entre **CKKS e BFV**. Ambos são perfeitamente viáveis para aplicações práticas que processem milhares de registos em tempo razoável (~3,5 segundos).

7.2. Comparação de Precisão

A análise de precisão revelou comportamentos distintos, mas ambos adequados ao contexto clínico abordado:

- O **CKKS** apresentou uma precisão notável considerando a sua natureza aproximada erros inferiores a 0,0002%. Este desvio é inferior à variabilidade natural das medições clínicas e, portanto, desprezáveis para análise estatística.
- O **BFV** demonstrou exatidão no cálculo da soma, confirmando a vantagem aritmética para operações com inteiros. Contudo, a média apresentou um desvio de 0,473 mmHg devido à técnica de scaling necessária para operações com decimais. Este erro, mesmo que seja superior aos do CKKS, continua a ser perfeitamente aceite e não afeta diagnósticos médicos.

Conclusão sobre precisão: Para o caso de uso testado, o CKKS é a escolha perfeita. Oferece precisão excelente sem necessidade de técnicas de scaling adicionais, diminuindo a complexidade da implementação. A sua aritmética aproximada não introduziu um impacto prático nos resultados também devido ao elevado valor atribuído ao global_scale. O CKKS trabalha nativamente com números reais, tornando-o naturalmente adequado para estatísticas médicas onde valores decimais são inerentes.

7.3. Segurança e Conformidade com RGPD

A implementação que efetuamos garante 128 bits de segurança criptográfica, sendo equivalente a AES-128. Este nível de segurança é considerado suficiente para aplicações reais segundo recomendações da NIST.

O sistema está em conformidade com as diretrizes do RGPD, mais especificamente com o Artigo 32 que aborda a segurança do tratamento.

É sempre bom relembrar que a segurança global deste sistema depende fortemente da proteção da chave privada, que no nosso caso encontra-se guardada no ficheiro secret_xxx.txt. Para tal esse ficheiro deve ser mantido em total sigilo pelo Data Holder e nunca partilhado. A chave privada é o único elemento capaz de recuperar os dados originais!

8. Referências

[1] https://sefiks.com/2023/04/10/a-step-by-step-fully-homomorphic-encryption-example-with-tenseal-in-python/

[2] https://www.youtube.com/watch?v=2qkCLaeD7pA

[3] https://github.com/OpenMined/TenSEAL

[4] https://mdcpp.com/doc/standard/ISO81060-2-2018(E).pdf

[5] https://pypi.org/project/tenseal/0.1.0a0/

[6] https://gdpr-text.com/pt/read/article-32/

9. Anexos

9.1. data_holder_ckks.ipynb

```
import tenseal as ts

import pandas as pd

import time

import gc


# Generate Keys

inicio = time.time()

context = ts.context(

    ts.SCHEME_TYPE.CKKS,

    poly_modulus_degree = 16384,

    coeff_mod_bit_sizes = [60, 40, 40, 60]

)


context.generate_galois_keys()

context.global_scale = 2**40


secret_context = context.serialize(save_secret_key=True)

with open("keys/secret_ckks.txt", "wb") as f:

        f.write(secret_context)


context.make_context_public()

public_context = context.serialize()

with open("keys/public_ckks.txt", "wb") as f:

        f.write(public_context)


fim = time.time()

print("Tempo de execução:", fim - inicio, "segundos")


# Encrypt
```

```
inicio = time.time()

with open("keys/secret_ckks.txt", "rb") as f:
    context = ts.context_from(f.read())

pressao_array = pd.read_excel("dataset_pressao_5000.xlsx")["pressao"].tolist()

inv_N = 1.0 / len(pressao_array)

pressao_encrypted = ts.ckks_vector(context, pressao_array)

inv_N_encrypted = ts.ckks_vector(context, [inv_N])

with open("outputs/pressao_encrypted_ckks.txt", "wb") as f:
    f.write(pressao_encrypted.serialize())

with open("outputs/inv_N_encrypted_ckks.txt", "wb") as f:
    f.write(inv_N_encrypted.serialize())

fim = time.time()
print("Tempo de execução:", fim - inicio, "segundos")

# decrypt
inicio = time.time()

with open("keys/secret_ckks.txt", "rb") as f:
    context = ts.context_from(f.read())

with open("outputs/mean_encrypted.txt", "rb") as f:
    mean_proto = f.read()

mean_enc = ts.lazy_ckks_vector_from(mean_proto)
```

```
mean_enc.link_context(context)
media = mean_enc.decrypt()[0]

with open("outputs/sum_encrypted.txt", "rb") as f:
    sum_proto = f.read()

sum_enc = ts.lazy_ckks_vector_from(sum_proto)
sum_enc.link_context(context)
soma = sum_enc.decrypt()[0]

fim = time.time()

print(f"Soma total: {soma} mmHg")
print(f"Média: {media} mmHg")
print("Tempo de execução:", fim - inicio, "segundos")
gc.collect()
```

9.2. data_analyzer_ckks.ipynb

```
import tenseal as ts
import time

inicio = time.time()

with open("keys/public_ckks.txt", "rb") as f:
    context = ts.context_from(f.read())

with open("outputs/pressao_encrypted_ckks.txt", "rb") as f:
    pressao_proto = f.read()
pressao_encrypted = ts.lazy_ckks_vector_from(pressao_proto)
pressao_encrypted.link_context(context)

with open("outputs/inv_N_encrypted_ckks.txt", "rb") as f:
```

```
    inv_N_proto = f.read()
inv_N_encrypted = ts.lazy_ckks_vector_from(inv_N_proto)
inv_N_encrypted.link_context(context)

sum_encrypted = pressao_encrypted.sum()
mean_encrypted = sum_encrypted * inv_N_encrypted

with open("outputs/sum_encrypted.txt", "wb") as f:
    f.write(sum_encrypted.serialize())

with open("outputs/mean_encrypted.txt", "wb") as f:
    f.write(mean_encrypted.serialize())

fim = time.time()
print("Tempo de execução:", fim - inicio, "segundos")
```

9.3. data_holder_bfv.ipynb

```
import tenseal as ts
import pandas as pd
import time
import gc

#   Generate Keys

inicio = time.time()

SCALE_FACTOR = 2**20

context = ts.context(
        ts.SCHEME_TYPE.BFV,
```

```
        poly_modulus_degree=16384,
        coeff_mod_bit_sizes=[60, 40, 40, 60],
        plain_modulus=1099511922689
    )
context.generate_galois_keys()

secret_context = context.serialize(save_secret_key=True)
with open("keys/secret_bfv.txt", "wb") as f:
        f.write(secret_context)

public_context = context.serialize(save_secret_key=False)
with open("keys/public_bfv.txt", "wb") as f:
        f.write(public_context)

fim = time.time()
print("Tempo de execução:", fim - inicio, "segundos")

# Encrypt

inicio = time.time()

with open("keys/secret_bfv.txt", "rb") as f:
    context = ts.context_from(f.read())

pressao_array = pd.read_excel("dataset_pressao_5000.xlsx")["pressao"].tolist()

inv_N_scaled = int(SCALE_FACTOR / len(pressao_array))

pressao_encrypted = ts.bfv_vector(context, pressao_array)
inv_N_encrypted = ts.bfv_vector(context, [inv_N_scaled])

with open("outputs/pressao_encrypted_bfv.txt", "wb") as f:
```

```
    f.write(pressao_encrypted.serialize())

with open("outputs/inv_N_encrypted_bfv.txt", "wb") as f:
    f.write(inv_N_encrypted.serialize())

fim = time.time()
print("Tempo de execução:", fim - inicio, "segundos")

# Decrypt

inicio = time.time()

with open("keys/secret_bfv.txt", "rb") as f:
        context = ts.context_from(f.read())

with open("outputs/sum_encrypted_bfv.txt", "rb") as f:
        sum_enc = ts.bfv_vector_from(context, f.read())
soma = sum_enc.decrypt()[0]

with open("outputs/mean_encrypted_bfv.txt", "rb") as f:
        mean_enc = ts.bfv_vector_from(context, f.read())
media_escalada = mean_enc.decrypt()[0]

media = media_escalada / SCALE_FACTOR

fim = time.time()
print(f"Soma total: {soma} mmHg")
print(f"Média: {media} mmHg")
print("Tempo de execução:", fim - inicio, "segundos")
gc.collect()
```

9.4. data_analyzer_bfv.ipynb

```
import tenseal as ts
import time

inicio = time.time()

with open("keys/public_bfv.txt", "rb") as f:
    context = ts.context_from(f.read())

with open("outputs/pressao_encrypted_bfv.txt", "rb") as f:
    pressao_encrypted = ts.bfv_vector_from(context, f.read())

with open("outputs/inv_N_encrypted_bfv.txt", "rb") as f:
    inv_N_encrypted = ts.bfv_vector_from(context, f.read())

sum_encrypted = pressao_encrypted.sum()

mean_encrypted = sum_encrypted * inv_N_encrypted

with open("outputs/sum_encrypted_bfv.txt", "wb") as f:
    f.write(sum_encrypted.serialize())

with open("outputs/mean_encrypted_bfv.txt", "wb") as f:
    f.write(mean_encrypted.serialize())

fim = time.time()
print("Tempo de execução:", fim - inicio, "segundos")
```