

Article

Empirical Study of Fully Homomorphic Encryption Using Microsoft SEAL

Francisco-Jose Valera-Rodriguez, Pilar Manzanares-Lopez  and Maria-Dolores Cano * 

Department of Information and Communication Technologies, Universidad Politécnica de Cartagena,
30202 Cartagena, Spain; pilar.manzanares@upct.es (P.M.-L.)

* Correspondence: mdolores.cano@upct.es

Abstract: In the context of the increasing integration of Internet of Things technologies and the growing importance of data lakes, the need for robust cybersecurity measures to protect privacy without compromising data utility becomes key. Aiming to address the privacy–security challenge in such digital ecosystems, this study explores the application of Fully Homomorphic Encryption (FHE) using the Microsoft SEAL library. FHE allows for operations on encrypted data, offering a promising opportunity for maintaining data confidentiality during processing. Our research employs systematic experimental tests on datasets to evaluate the performance of homomorphic encryption in terms of CPU usage and execution time, executed across traditional PC configurations and a NVIDIA Jetson Nano device to assess the scalability and practicality of FHE in edge computing. The results reveal a performance disparity between computing environments, with the PC showing stable performance and the Jetson Nano revealing the limitations of edge devices in handling encryption tasks due to computational and memory constraints.

Keywords: anonymity; cybersecurity; data privacy; edge computing; IoT; homomorphic encryption



Citation: Valera-Rodriguez, F.-J.; Manzanares-Lopez, P.; Cano, M.-D. Empirical Study of Fully Homomorphic Encryption Using Microsoft SEAL. *Appl. Sci.* **2024**, *14*, 4047. <https://doi.org/10.3390/app14104047>

Academic Editor: Gianluigi Ferrari

Received: 21 March 2024

Revised: 24 April 2024

Accepted: 7 May 2024

Published: 10 May 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

In today's digital age, the exponential growth of data generated by the Internet of Things (IoT) and other digital platforms has led to the emergence of vast repositories of information known as data lakes [1,2]. These expansive, fluid collections of raw data, stored in their native format, offer unprecedented opportunities for analytics, innovation, and personalized services. The concept of data lakes has revolutionized the way businesses and organizations harness the power of big data, enabling the aggregation and analysis of varied data types to glean insights, drive decision making, and foster technological advancements [3,4]. By integrating diverse datasets, data lakes facilitate comprehensive analytics that can predict trends, enhance efficiency, and create more tailored user experiences.

However, the benefits of leveraging extensive datasets come with significant privacy and security challenges. As we navigate through this wealth of information, the line between valuable insights and invasive privacy breaches becomes increasingly blurred. The need to protect sensitive information while exploiting data to their full potential has never been more critical. Traditional cybersecurity measures often fall short of addressing the unique vulnerabilities presented by the interconnected and open nature of data lakes and IoT ecosystems. The dynamic landscape of digital threats necessitates innovative approaches to data security, ones that can adeptly safeguard privacy without sacrificing the utility and benefits of these technological advancements [5–7].

The introduction of fully homomorphic encryption (FHE) stands out as an opportunity for addressing the privacy–security challenge [8,9]. FHE is a form of encryption that allows computations to be performed on encrypted data, generating encrypted results that, once decrypted, reveal the outcome of operations as if they had been conducted on the original plaintext. This groundbreaking capability ensures that sensitive data can remain encrypted

throughout the analytical process, thereby preserving confidentiality while still enabling valuable insights to be extracted.

The incorporation of FHE into the realms of data lakes and IoT represents a transformative approach to managing the inherent privacy and security risks associated with big data analytics. By enabling data to be processed in their encrypted form, FHE mitigates the risks of data breaches and unauthorized access, ensuring that the integrity and confidentiality of sensitive information are maintained. This is particularly pertinent in the context of data lakes, where the sheer volume and variety of data heighten the potential for privacy infringements. Nevertheless, it is important to note that homomorphic encryption does not provide a means to bypass privacy regulations such as the European General Data Protection Regulation (GDPR) [10]. For instance, it is not feasible for a cloud service to apply homomorphic encryption in order to derive insights from customer data while they remain encrypted. Rather, the outcomes of computations performed on encrypted data stay encrypted and can only be decrypted by the data's owner, such as a customer of a cloud service. This adds another limitation, which is that in situations where several distinct private data owners intend to partake in joint computational efforts, homomorphic encryption is likely not a viable option. This is because homomorphic encryption frameworks operate with a singular secret key, which is possessed by the owner of the data.

In this paper, we offer a detailed empirical analysis of FHE using the Microsoft SEAL library, with the aim of understanding FHE's practical performance and scalability. We particularly shed light on the optimization challenges and potential solutions for implementing FHE in edge computing environments, such as those encountered with IoT devices, by comparing its performance on a traditional computing system (PC) and an edge device (NVIDIA Jetson Nano). This comparison not only illustrates the specific computational and memory constraints of edge devices but also emphasizes the need for algorithmic improvements to ensure the viability of FHE in such contexts. In addition, our work could serve in addressing real-world implementation by bridging the theoretical aspects of cryptographic advancements with their practical application implications.

The rest of the paper is organized as follows. Section 2 presents a brief on the state of the art in FHE use and the main libraries available. Section 3 describes the research methodology. In Section 4, we showcase and discuss the results. The paper ends summarizing the most important outcomes.

2. Related Works

This section covers the related literature. It is divided into two subsections. The first one addresses research works on homomorphic encryption. The second one describes the available libraries and tools to implement this technique.

2.1. Research Trends and Challenges

Homomorphic encryption (HE) is a form of encryption that allows computations to be performed on ciphertexts, producing an output which, when decrypted, matches the result of operations applied to the plaintexts. This innovative approach to data security enables the analysis of sensitive information without exposing the underlying data, thus ensuring privacy and confidentiality in digital transactions. Homomorphic encryption can be broadly categorized into three types, each offering different levels of functionality and efficiency. Partially homomorphic encryption (PHE) is the simplest form of homomorphic encryption, supporting either addition or multiplication operations on encrypted data, but not both. PHE has been the basis for many early applications of homomorphic encryption due to its relatively straightforward implementation and lower computational requirements compared to more advanced forms. Somewhat homomorphic encryption (SHE) extends the capabilities of PHE by allowing a limited number of both additions and multiplications on encrypted data. However, the depth of these operations is constrained, as performing too many operations can degrade the integrity of the encryption, rendering the data unusable. Finally, fully homomorphic encryption (FHE) represents the most ad-

vanced form of homomorphic encryption, enabling an unlimited number of additions and multiplications on encrypted data without compromising the encryption's integrity. This level of functionality allows for complex computations on encrypted data, making FHE a highly sought-after solution for secure data processing in sensitive applications.

Since the conception of fully homomorphic encryption by C. Gentry in 2009, presenting a lattice-based construction that proved the feasibility of FHE [8], the field has seen rapid advancements and an interesting range of applications. The authors in [11] introduce a secure and anonymous face verification scheme utilizing FHE and the SEAL Microsoft library. Their proposal contains a three-party architecture with a client, a data server, and an authentication server. The scheme ensures privacy by preventing the disclosure of facial data and indices to untrusted servers. They use the Brakerski–Fan–Vercauteren (BFV) homomorphic encryption scheme integrated with private information retrieval for efficient query processing that maintains user anonymity. It operates with a communication volume of 264 KB and completes verifications in 84.91 ms, making it suitable for devices with limited computational power. The rigorous security analysis confirms that the scheme robustly protects sensitive facial features and verification indices while ensuring minimal data exposure, even in untrusted cloud environments. Other novel works in related fields could benefit from the integration of FHE. For instance, integrating FHE with the Signal Noise Separation-Based Network (SNIS) for image forgery detection method proposed in [12] could enhance privacy-preserving image analysis. By adapting SNIS algorithms to process encrypted data using an optimized FHE scheme like Cheon–Kim–Kim–Song (CKKS), sensitive images could be analyzed without decryption, ensuring data confidentiality and compliance with data protection laws such as GDPR and HIPAA.

From a different approach, a framework for employing FHE over neural network graphs, specifically applying it to vision-based tasks and agricultural yield forecasting using a privacy-preserving machine learning model on encrypted data, is introduced in [13]. Their framework utilizes the CKKS encryption scheme integrated within Microsoft SEAL. They focus on enhancing the usability of FHE in practical deep learning scenarios by automating cyphertext parameterization and optimizing graph traversal and computational graph configurations for neural networks. Interestingly, the study highlights the non-universality of FHE due to inherent limitations like computational intensity and data volume constraints but illustrates its effectiveness in specific applications like the one presented in this paper.

Recent trends in FHE research have focused on optimizing performance and usability [14] to overcome the significant computational challenges inherent in FHE implementations. Innovations in lattice-based cryptography, particularly in constructing more efficient encryption schemes and reducing the noise generated through homomorphic operations, have significantly improved FHE's practicality. Techniques such as bootstrapping and modulus switching have been refined to manage noise and enable deeper computations, thereby extending the range of applications that can benefit from FHE [15]. A good example can be found in [16], where the authors explore the efficiency of comparison operations on homomorphically encrypted data. Their study focuses on the impact of binary and bitmask encoding methods on the performance of these operations within the Toroidal FHE (TFHE) scheme. Their results show that while bitmask encoding simplifies computations and reduces the number of operations, it scales poorly with larger domain sizes due to increased ciphertext size and computational overhead. Conversely, binary encoding offers more scalable performance across varying data sizes but involves more complex operations. The findings suggest that the choice between binary and bitmask encoding should be tailored to the specific requirements of the application in terms of domain size and computational constraints. Moreover, the development of specialized libraries and tools, such as those described in the next section, has played a crucial role in making FHE more accessible to researchers and developers. These tools provide optimized implementations of FHE schemes, abstracting the underlying cryptographic complexities and enabling the integration of FHE into existing software applications.

Despite these advancements, several challenges remain in the widespread adoption of fully homomorphic encryption. The primary issue is the computational overhead associated with FHE operations [9,17–19]. Even with the latest optimizations, FHE computations are still significantly slower than operations on plaintext data, limiting their applicability in time-sensitive or resource-constrained environments. Another area of ongoing research is the development of more efficient key management and sharing mechanisms for FHE. In multi-party computation scenarios, securely managing encryption keys without compromising data security poses a considerable challenge [20,21]. Furthermore, the usability of FHE tools and libraries remains a barrier for many potential users. Simplifying the integration of FHE into applications and improving the developer experience are critical steps toward broader adoption.

2.2. Libraries

Several key libraries have emerged as foundational tools in the FHE landscape. These libraries and tools represent crucial building blocks for the development and deployment of FHE applications. They encapsulate complex cryptographic operations into more accessible interfaces, enabling the secure processing of encrypted data across diverse domains, from cloud computing to secure multi-party computation and beyond.

One of the most known ones is the Microsoft Simple Encrypted Arithmetic Library (SEAL) [22]. Developed by Microsoft, SEAL stands out for its focus on providing a set of powerful and efficient primitives for implementing encryption schemes, including FHE. In terms of programming languages, SEAL is written in C++; therefore, it is mainly used with C++ or C++ compatible languages. In addition, it can be wrapped for use with other languages, such as Python. Microsoft SEAL offers a pair of homomorphic encryption schemes, each with distinct characteristics. The Brakerski–Fan–Vercauteren (BFV) and Brakerski–Gentry–Vaikuntanathan (BGV) schemes enable operations like modular arithmetic on encrypted integers. Conversely, the Cheon–Kim–Kim–Song (CKKS) scheme facilitates operations on encrypted real or complex numbers, providing approximate outcomes. For tasks requiring the summation of encrypted real numbers, applying machine learning models to encrypted data, or calculating distances between encrypted locations, CKKS emerges as the superior option. However, for use cases where precise values are critical, the BFV and BGV schemes prove to be more appropriate. SEAL is praised for its comprehensive documentation, ease of use, and active development community, making it a preferred choice for beginners and experienced practitioners alike.

PySEAL is a Python wrapper for Microsoft SEAL. PySEAL extends the accessibility of FHE to the Python programming community, which is one of the most popular languages in data science and machine learning. This extension enables a broader range of developers and researchers to experiment with and integrate FHE into their Python-based projects, leveraging the rich ecosystem of data analysis and machine learning libraries available in Python. OpenFHE, formerly known as PALISADE, is an open-source project that aims to provide a comprehensive suite of tools for FHE. It supports a variety of encryption schemes, including BGV, CKKS, and BFV, catering to different application needs. OpenFHE is recognized for its flexibility and extensive functionality, offering features that support not just encryption but also key management, serialization, and multi-threaded operations, among others.

3. Methodology

The research methodology and tools utilized for conducting experiments in this study are thoroughly described next. Throughout this study, we focus on the practical application of homomorphic encryption using the Microsoft SEAL library to carry out the experiments. The explanation includes system requirements, software, and tools.

3.1. System Requirements

To ensure optimal performance and the correct implementation of the Microsoft SEAL library, specific system requirements must be met. In this work, we are working with the latest stable version of Microsoft SEAL to date, version 4.1.1. It is advisable to try to use the most recent version in order to take advantage of the possible improvements that this library may provide.

The requirements are fundamental for projects that demand high resource usage or have particular needs, and they might vary to accommodate the specific characteristics of a project. The requirements include sufficient hard disk space for installing all necessary software and tools; compatibility with various operating systems such as Linux, macOS, or Windows (with a preference for Windows 10 for its compatibility with SEAL); at least 4 GB of RAM (although 8 GB is recommended for efficient cryptographic operations); and a 64-bit processor architecture for optimal performance. Additionally, an appropriate development environment, particularly Visual Studio 2022, is essential for C++ project development, ensuring high performance when using the Microsoft SEAL library. The inclusion of specific work packages during the Visual Studio installation, such as the “Desktop development with C++” package and “CMake Tools,” is also necessary for the project’s successful execution.

3.2. Software and Tools

We have employed several software tools and technologies to facilitate the practical implementation of homomorphic encryption techniques:

- CMake (version 3.27.4) is a key tool for controlling the compilation process, defining project build files, and ensuring flexibility across different platforms. It generates CMakeLists.txt files, which are instrumental in compiling and building the source code. For this work, version 3.27.4 was used, which is the latest stable version available at the time of the project’s execution;
- Git is employed for version control, allowing for the maintenance and tracking of different versions of applications and projects. It was specifically used to monitor updates to the Microsoft SEAL repository and clone it locally for implementation;
- Visual Studio 2022 (v.17) is an Integrated Development Environment (IDE) that provides a comprehensive development platform for writing, debugging, and compiling code as well as deploying applications. It supports a wide range of programming languages and project types, including C++ and .NET applications, mobile app development, and web design. In this work, we utilized Visual Studio 2022 for its advanced debugging, compilation, and development tools, along with its stable features up to the current date.

The applied methodology involves practical application through a simple coding example that demonstrates homomorphic encryption’s functionality. Specifically, the example presents a homomorphic sum, in which two integers are encrypted and then added together in encrypted form, after which both the numbers and the result of the sum are decrypted and displayed. The code will be tested in different devices and with different required computational resources, as explained in the next section. The setup encompasses configuring the development environment, integrating necessary libraries and dependencies, and detailing the software used. With this example, we showcase operations on encrypted data using the Microsoft SEAL library, emphasizing the encryption and decryption processes, as well as operations such as addition on encrypted integers.

3.3. Experiments Design

Experiments are carried out in two different batches. In the first batch, we use Visual Studio 2022 in a PC with Windows 10, Intel Core i7-8550U processor, and 8 GB of RAM. In the second batch, we use a Jetson Nano NVIDIA equipped with a quad-core ARM Cortex-A57 processor, 128-core Maxwell GPU, and 4 GB LPDDR4 RAM, using JetPack 4.6. JetPack is NVIDIA’s all-inclusive Software Development Kit (SDK) that provides the

board support package, Linux operating system, NVIDIA CUDA, cuDNN, and TensorRT software libraries for deep learning, computer vision, GPU computing, and multimedia processing. The installation of SEAL was performed manually using CMake.

In both cases, the study analyzes the impact of varying test parameters on program execution time and CPU resource requirements. The parameters included are as follows:

- Dataset size, varied to analyze the program's response to different quantities. The sizes tested were 1, 10, 100, 1000, and 10,000 samples, offering a broad range from minimal to large datasets. This dataset is composed of integers that are randomly generated.
- Encryption parameters focus on the BFV encryption scheme, specializing in integers, with the variable *poly_modulus_degree* impacting scheme complexity and operational capacity. The values tested were 1024, 2048, 4098, 8192, 16,384, and 32,768, providing a wide spectrum of results from low to high complexity levels. We chose BFV versus BGV; note that both are efficient in working with integers but BFV is more efficient for lower values of the *plain_modulus* variable, as explained below [23].

The test program included a detailed setup for SEAL, generating the random numbers for encryption and measuring execution time for various configurations, as shown in Algorithm 1. As for the SEAL setup, when using the BFV scheme, there are three parameters that are key. These are plaintext modulus (*plain_modulus*), *poly_modulus_degree* (*poly_modulus_degree*), and coefficient modulus (*coeff_modulus*). *Plain_modulus* is an integer that defines the range of integers (coefficients of polynomials) that can be represented and operated on within the encrypted space. That is, the plain modulus determines how data are encoded before encryption. Its main goal is to enable the correct wrapping of integer values during arithmetic operations in the encrypted form. Consequently, this variable sets the bounds for data representation, but it does not affect the computational complexity of encryption. We checked this behavior experimentally, and it is correct. Therefore, the parameter *plain_modulus* will be kept at a fixed value of 1024 because our dataset contains randomly generated integers smaller than 1024.

Next, *coeff_modulus* is a modulus used for the coefficients of the polynomials that represent encrypted data. As we know, ciphertexts and encryption operations in the BFV scheme are represented as polynomials, and these polynomials' coefficients are reduced modulo of the *coeff_modulus*. The size of the *coeff_modulus* in homomorphic encryption schemes directly influences security by determining the noise budget. The larger the noise budget is, the larger the amount of noise that can be introduced into the ciphertext before it becomes too corrupted to be decrypted accurately. However, a larger *coeff_modulus* also increases ciphertext sizes, impacting the encryption system's performance and necessitating a balance between security enhancements and computational efficiency. For this reason, a recommended value for *coeff_modulus* is given by the SEAL library once the polynomial modulus degree is set. The polynomial modulus degree (*poly_modulus_degree*) determines the size of the polynomials used in the encryption scheme. This parameter influences the performance of encryption. The *Poly_modulus_degree* must be a positive power of 2, representing the degree of a power-of-two cyclotomic polynomial, which defines the polynomial ring over which all arithmetic operations are conducted. Larger values allow for more complex encryption computations, making it more difficult for potential attackers to derive useful information from encrypted data and hence providing better security, but also results in larger ciphertext sizes and increased computational overhead because ciphertext will contain more coefficients.

The execution time and CPU usage were measured across the different data sizes and *poly_modulus_degree* values, aiming for a deep understanding of performance under various configurations. The *chrono* library was included in the program to measure the execution time. Time measurements started just before the cipher and stopped just after the execution of the cipher. The confidence interval of the results was 95%, which was calculated with a normal distribution using 5 samples for each combination of the values for *poly_modulus_degree* {1024, 2048, 4098, 8192, 16,384, and 32,768} and data samples {1, 10, 100, 1000, 10,000}; hence, averages will be shown in the results.

Regarding the CPU measurements, for the first batch of experiments with the PC, we used the Windows resource monitoring tool. For the second batch of experiments with the Jetson, we used *htop* as the system monitoring tool. It runs on the terminal and visually provides system resource utilization. In both cases, instead of carrying out several consecutive runs as we did to measure the execution time, we kept the *poly_modulus_degree* as a fixed value and carried out one run for each dataset to observe the CPU behavior. Then, the same process was repeated for all values of the *poly_modulus_degree*.

Algorithm 1. Pseudocode to show the process of encrypting a set of integers using the Microsoft SEAL library's BFV homomorphic encryption scheme and measure the execution time.

```

Initialize vector plain_numbers to store integers
Generate and store x random integers in plain_numbers
Repeat the following steps 5 times:
    Start timing the encryption process
    Set up encryption parameters:
        Define poly_modulus_degree as y
        Set coefficient modulus based on BFV default for the defined degree
        Set plain modulus as 1024
        Initialize the SEAL encryption context with these parameters
    Generate cryptographic keys:
        Create a key generator using the SEAL context
        Generate a public key and a secret key
    Initialize encryptor and decryptor:
        Create an encryptor with the context and public key
        Create a decryptor with the context and secret key
    Initialize an empty vector cipher_numbers to store encrypted data
    For each number in plain_numbers:
        Convert the number to a plaintext object
        Encrypt the plaintext to ciphertext using the encryptor
        Store the ciphertext in cipher_numbers
    Stop timing the encryption process
    Calculate and print the duration of the encryption process in seconds
Print the total number of samples in plain_numbers

```

4. Results

This section presents the results from the series of experimental tests evaluating homomorphic encryption on datasets, specifically using Microsoft's SEAL library. Homomorphic encryption, as discussed in the earlier sections, allows for operations to be performed directly on encrypted data, showcasing vast potential in security and privacy protection. SEAL is highlighted as a prime choice for the practical implementation of homomorphic encryption in projects today. The experimental tests focused on two main factors: CPU usage and the execution time necessary for encrypting datasets. The primary goal was to identify optimal parameter values for the best performance across different scenarios, understanding how parameter variations impact performance, particularly concerning the encryption scheme used and the dataset size.

The results for the first test batch, which was conducted using Visual Studio 2022 in a Windows 10 environment with the system powered by an Intel Core i7-8550U processor and 8 GB of RAM, are depicted in Figure 1. It can be observed that the execution time varied significantly with dataset size and *poly_modulus_degree* value. Smaller datasets and lower degrees showed faster execution times, while larger datasets and higher degrees increased execution time exponentially, indicating a direct correlation between data complexity, encryption scheme complexity, and performance.

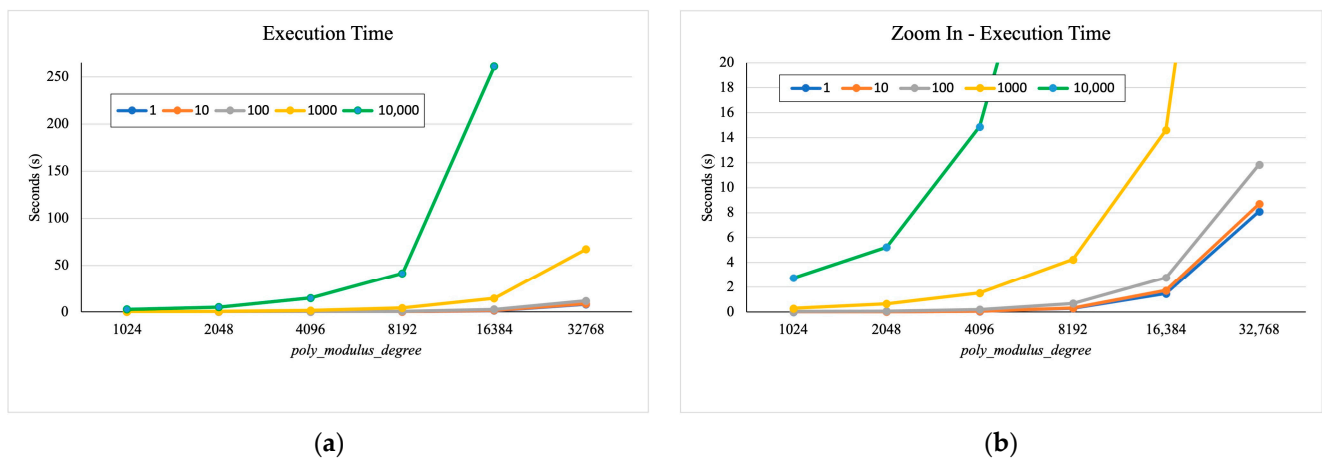


Figure 1. Average execution time for experiments run on a PC. Experiments were run five times for each configuration and the average values are shown. (a) Execution time. For the *poly_modulus_degree* with a value of 32,768, the execution failed for the largest dataset and was not able to finish. (b) Zoomed in.

The CPU load was closely monitored to assess the efficiency and practicality of homomorphic encryption in real-world applications. In this case, the Windows resource monitoring tool was used to monitor the CPU activity for 60 s, during which time various executions of the program were performed. The process to followed involves performing the tests, keeping a fixed value for the degree of the polynomial, and varying the size of the dataset to see the changes it produces in the CPU load. Then, the same process will be repeated for all the values of the polynomial degree studied in this work. For each configuration, we ran the program twice to correctly obtain the results, since the first run involves compiling and configuration and does not really show the result of the clean encryption. Nevertheless, both runs are shown in the illustrations. From Figures 2–4, the tests reveal that higher *poly_modulus_degree* values and larger datasets demand substantially more CPU resources, confirming the need for optimized parameter selection to balance security, privacy, and performance effectively.

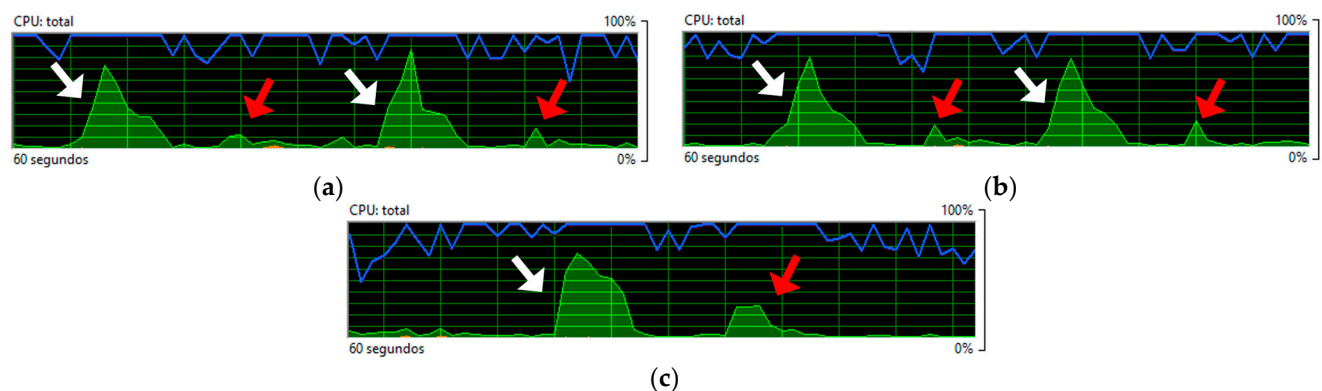


Figure 2. CPU load using a *poly_modulus_degree* of 1024; white arrows show the first execution and red arrows mark the second execution: (a) 1 and 10 samples; (b) 100 and 1000 samples; (c) 10,000 samples.

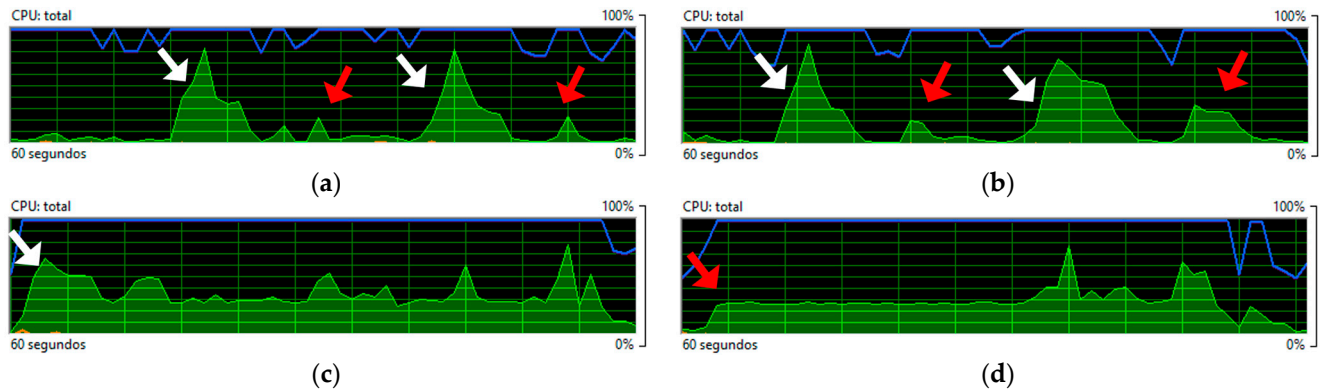


Figure 3. CPU load using a *poly_modulus_degree* of 8192; white arrows show the first execution and red arrows mark the second execution: (a) 1 and 10 samples; (b) 100 and 1000 samples; (c) 10,000 samples, first execution; (d) 10,000 samples, second execution.

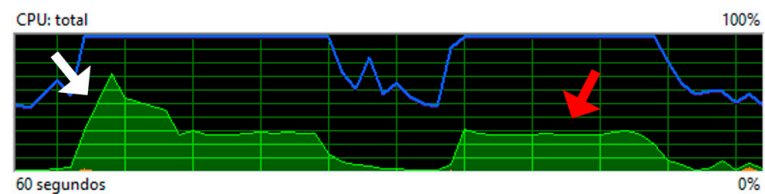


Figure 4. CPU load using a *poly_modulus_degree* of 16,384 for 1000 samples; white arrows show the first execution and red arrows mark the second execution.

From the results shown in Figure 2, using a *poly_modulus_degree* of 1024, it can be seen that in the first run the CPU load oscillates around 70% for all the data sizes. In the second run, we can see that the larger the size is, the higher the load achieved, especially for large values. The highest increase is reached at 10,000 encrypted integers, where up to 30% is used, while small and medium values remain between 10% and 20%. For a polynomial degree value of 8192 and 10,000 samples, as in Figure 3, on average an 80% CPU load is reached in the first run, whereas in the second run it varies from 20% in small values and 30% in medium values up to peaks of 60% or 70%. Using a *poly_modulus_degree* of 16,384 and 1000 samples, as in Figure 4, the results show that in the first run an average of 70% of the peak CPU load is reached. It is so demanding that it reaches almost 100% load. For the second run, the load oscillates around 30%. In this scenario and with 10,000 samples, a 100% peak load is reached due to the complexity of the run. Finally, for a degree value of 32,768, the load in the first run averages 75%, except for 10,000 samples where it peaks at 100%. In the second run, it oscillates between 30% and 40% up to sets of 100 samples. For 1000 samples, 100% is reached in some parts of the run. For a larger set size, it is not possible to carry out the execution due to the increased complexity and memory demand.

As we have seen, the impact of caching between the first and second runs is notable. This effect is due to the compilation of the code, configuration of the encryption parameters, and initialization of cryptographic contexts and keys. Once these operations are executed, some elements such as compiled code and key structures are cached, enabling faster access and reduced CPU load in later runs. Additionally, operating systems and runtime environments may optimize memory management and execution paths based on previous operations, enhancing performance efficiency. In the first run across various configurations, the CPU load was relatively high as the system compiled and configured the encryption setup. For example, with a *poly_modulus_degree* of 1024 (Figure 2, white arrow), the CPU load averaged around 70%. However, in the second run, the CPU load markedly decreased. For large datasets (10,000 samples), the load dropped from near maximum capacity in the first run to about 30% in the second run, showing a substantial reduction. This reduction in CPU load in the second run indicates that the system benefits from caching, which could

include optimized access to frequently used data or compiled code, in turn resulting in faster execution times and lower resource usage in successive runs.

The second batch of experiments was carried out on a Jetson Nano. The exploration of homomorphic encryption performance on the Jetson Nano provides a distinct perspective on the applicability and scalability of cryptographic practices in edge computing environments. Given the inherent resource constraints of devices like the Jetson Nano, this analysis not only benchmarks the operational efficiency of homomorphic encryption but also delineates the potential for deployment in IoT ecosystems. The Jetson Nano Developer Kit by NVIDIA has a quad-core ARM Cortex-A57 processor, 128-core Maxwell GPU, and 4GB LPDDR4 RAM. The configuration involved setting up the environment with JetPack, flashing the system image, and installing the Microsoft SEAL library and monitoring tools like *htop* for system utilization visualization.

The critical parameters under investigation mirrored those of the previous PC-based tests, emphasizing dataset size and the *poly_modulus_degree* within the BFV encryption scheme. The analysis began with an examination of execution times across varying dataset sizes and *poly_modulus_degree* values. The execution time on the Jetson Nano exhibited a pronounced increase with larger datasets and higher degrees, similar to observations made on the PC (see Figure 5). However, distinct disparities emerged as the dataset size expanded. For smaller datasets (1, 10 samples), the execution times were comparatively lower on the Jetson Nano than on the PC, suggesting an efficient utilization of the Nano's processing capabilities for lightweight cryptographic tasks. As the dataset size increased to 100 samples, a tipping point was observed where the execution time on the Jetson Nano began to lag significantly behind the PC's performance, underscoring the impact of limited computational resources. Finally, for larger datasets (1000 and 10,000 samples), the execution time on the Jetson Nano escalated dramatically, reflecting the device's constraints in handling intensive cryptographic operations. The device struggled with the largest datasets, especially at higher *poly_modulus_degree* values, illustrating the limitations of edge devices in executing complex homomorphic encryption tasks without optimization.

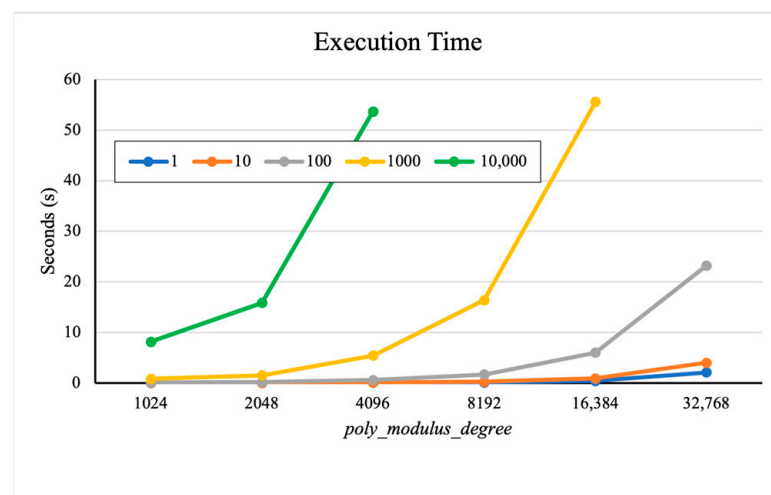


Figure 5. Average execution time for experiments run on a Jetson Nano. Experiments were run five times for each configuration.

The investigation extended to CPU load and system utilization, revealing the Jetson Nano's operational thresholds. Similar to the previous experiments, initial tests with smaller datasets and lower *poly_modulus_degree* values demonstrated minimal CPU strain, with system utilization remaining within acceptable ranges. This indicated the Nano's capability to handle basic homomorphic encryption tasks efficiently. When progressing to medium-sized datasets, an increase in CPU load was observed, indicating heightened resource demand. However, the device managed to complete the tasks, albeit with increased

execution times. Then, at the upper values of dataset size and *poly_modulus_degree*, the Jetson Nano's CPU load reached its zenith, often hitting 100% utilization. This was particularly evident for the largest datasets, where the device occasionally failed to complete the encryption process due to exhaustive memory consumption and processing power limitations. The results are shown in Tables 1 and 2.

Table 1. CPU load in % for experiments run on a Jetson Nano.

Data Size\ <i>poly_modulus_degree</i>	1024	2048	4096	8192	16,384	32,768
1	8%	11%	12%	16%	40%	100%
10	10%	12%	15%	24%	58%	100%
100	14%	15%	41%	78%	100%	100%
1000	60%	82%	100%	100%	100%	N.a. ¹
10,000	100%	100%	100%	N.a. ¹	N.a. ¹	N.a. ¹

¹ Not achieved.

Table 2. RAM used in % for experiments run on a Jetson Nano.

Data Size\ <i>poly_modulus_degree</i>	1024	2048	4096	8192	16,384	32,768
1	36%	36%	36%	36%	36%	43%
10	36%	36%	36%	36%	38%	46%
100	37%	37%	37%	39%	43%	73%
1000	40%	42%	41%	50%	88%	N.a. ¹
10,000	43%	50%	90%	N.a. ¹	N.a. ¹	N.a. ¹

¹ Not achieved.

An interesting point to note is how the occasional failures noted in the experiments on the Jetson Nano under high loads and large data sizes could lead to potential vulnerabilities from a security-application perspective. We have seen that these failures occur because of the limited memory capacity, resulting in the inability to complete the encryption task. Consequently, attackers could deliberately target systems using similar edge computing devices with high-load tasks to induce failure, causing a denial of service through resource exhaustion. Another risk would be the exposition of unencrypted data, or partially encrypted data, when the encryption process is incomplete or failed. Finally, it is important to observe that continuous operation near resource limits could lead to system instability, which might be exploited to disrupt operations or to force systems into a less secure state.

A comparative analysis between the PC-based experiments and those conducted on the Jetson Nano sheds light on the scalability and adaptability of homomorphic encryption technologies across different computing environments. This comparison is crucial for understanding the practical implications of deploying advanced cryptographic solutions within both traditional and edge computing frameworks. In this sense, the Jetson Nano exhibited good performance with smaller datasets, often matching or slightly outperforming the PC setup in terms of execution time. This efficiency underscores the Nano's potential for handling lightweight encryption tasks effectively, despite its constrained computational resources. However, as the dataset size increased to medium levels, the execution time on the Jetson Nano began to lag significantly compared to the PC, highlighting the impact of its limited processing capabilities on handling more complex cryptographic computations. The disparity in execution time became more pronounced with larger datasets. The PC maintained a relatively stable performance curve, managing to complete encryption tasks with considerable efficiency even as the complexity of operations increased. Conversely, the Jetson Nano's performance degraded severely under the weight of larger datasets, particularly at higher *poly_modulus_degree* values, illustrating the challenges faced by edge devices in executing computationally intensive homomorphic encryption tasks without specific optimizations.

The comparative analysis of CPU load revealed that the Jetson Nano operates within a higher utilization range across all dataset sizes and *poly_modulus_degree* values compared to the PC. For small datasets, the difference in CPU load between the two platforms was minimal, showcasing the Nano's capability to efficiently manage basic encryption tasks. However, as the computational demands increased, the Nano's CPU load reached its peak, often hitting 100% utilization, while the PC showed a more gradual increase in CPU load, maintaining better overall performance stability. Similarly, memory utilization trends have highlighted significant constraints on the Jetson Nano, especially with large datasets. The PC's ample memory capacity allowed it to handle extensive cryptographic operations with relative ease, whereas the Jetson Nano struggled with memory saturation, leading to execution failures in the most demanding scenarios. This limitation points to the critical need for memory-efficient cryptographic algorithms and optimizations tailored for edge computing devices.

Traditional computing environments and edge devices like the Jetson Nano reveal critical insights for implementing homomorphic encryption in edge computing. There is a pronounced need for optimization to adapt encryption practices to the limitations of edge hardware, requiring algorithmic refinements that minimize computational and memory demands. Furthermore, the performance gap underscores the necessity for both cryptographic algorithm innovations and advancements in edge device capabilities. Tailoring encryption methods to meet the specific needs of edge computing can significantly improve operational efficiency, making secure, privacy-preserving computations feasible within IoT ecosystems. Moreover, the deployment of homomorphic encryption technologies must consider application-specific demands, including data sensitivity, computational requirements, and resource availability. The analysis highlights the importance of edge-specific optimizations and parameter adjustments to meet these needs effectively.

5. Conclusions

While FHE is a promising solution to keeping the balance between data privacy and data use for innumerable applications, its applicability is still a challenge. This study's investigation into FHE using the Microsoft SEAL library underscores the technical challenges and considerations necessary for applying FHE across different computational environments, including traditional PCs and edge devices like the Jetson Nano. Our research demonstrates that while FHE offers a pathway to secure and privacy-preserving computation by enabling operations on encrypted data, its practical deployment is hindered by significant computational overhead and memory requirements. These limitations are particularly pronounced in environments such as edge computing, which are increasingly prevalent in IoT ecosystems. If FHE is to be applied in both high-resource and resource-constrained environments, such as data lakes, further works on optimizing these algorithms should be addressed.

Author Contributions: Conceptualization, P.M.-L. and M.-D.C.; methodology, P.M.-L. and M.-D.C.; software, F.-J.V.-R.; validation, F.-J.V.-R., P.M.-L. and M.-D.C.; investigation, F.-J.V.-R., P.M.-L. and M.-D.C.; writing—original draft preparation, F.-J.V.-R. and P.M.-L.; writing—review and editing, M.-D.C.; visualization, F.-J.V.-R., P.M.-L. and M.-D.C.; supervision, P.M.-L. and M.-D.C.; project administration, M.-D.C.; funding acquisition, M.-D.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research was carried out within the framework of the Recovery, Transformation and Resilience Plan funds, financed by the European Union (Next Generation).

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

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

References

1. Fang, H. Managing data lakes in big data era: What's a data lake and why has it become popular in data management ecosystem. In Proceedings of the IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER), Shenyang, China, 8–12 June 2015; pp. 820–824.
2. Maia, E.; Wannous, S.; Dias, T.; Praça, I.; Faria, A. Holistic Security and Safety for Factories of the Future. *Sensors* **2022**, *22*, 9915. [CrossRef] [PubMed]
3. Yu, W.; Liu, Y.; Dillon, T.; Rahayu, W.; Mostafa, F. An Integrated Framework for Health State Monitoring in a Smart Factory Employing IoT and Big Data Techniques. *IEEE Internet Things J.* **2022**, *9*, 2443–2454. [CrossRef]
4. Yu, W.; Dillon, T.; Mostafa, F.; Rahayu, W.; Liu, Y. A Global Manufacturing Big Data Ecosystem for Fault Detection in Predictive Maintenance. *IEEE Trans. Ind. Inform.* **2020**, *16*, 183–192. [CrossRef]
5. Mrozek, D.; Tokarz, K.; Pankowski, D.; Malysiak-Mrozek, B. A Hopping Umbrella for Fuzzy Joining Data Streams From IoT Devices in the Cloud and on the Edge. *IEEE Trans. Fuzzy Syst.* **2020**, *28*, 916–928. [CrossRef]
6. Ding, R.; Xu, Y.; Zhong, H.; Cui, J.; Sha, K. Towards Fully Anonymous Integrity Checking and Reliability Authentication for Cloud Data Sharing. *IEEE Trans. Serv. Comput.* **2023**, *16*, 3782–3795. [CrossRef]
7. Orešćanin, D.; Hlupić, T.; Vrdoljak, B. Managing Personal Identifiable Information in Data Lakes. *IEEE Access* **2024**, *12*, 32164–32180. [CrossRef]
8. Gentry, C. *A Fully Homomorphic Encryption Scheme*; Stanford University: Stanford, CA, USA, 2009.
9. Marcolla, C.; Sucasas, V.; Manzano, M.; Bassoli, R.; Fitzek, F.H.P.; Aaraj, N. Survey on Fully Homomorphic Encryption, Theory, and Applications. *Proc. IEEE* **2022**, *110*, 1572–1609. [CrossRef]
10. European Commission General Data Protection Regulation (GDPR). Available online: <https://gdpr-info.eu/> (accessed on 15 January 2024).
11. Wang, X.; Li, P. A Secure Face Verification Scheme Based on Fully Homomorphic Encryption with Anonymity. *Information* **2024**, *15*, 129. [CrossRef]
12. Chen, J.; Liao, X.; Wang, W.; Qian, Z.; Qin, Z.; Wang, Y. SNIS: A Signal Noise Separation-Based Network for Post-Processed Image Forgery Detection. *IEEE Trans. Circuits Syst. Video Technol.* **2023**, *33*, 935–951. [CrossRef]
13. Onoufriou, G.; Hanheide, M.; Leontidis, G. EDLaS: Fully Homomorphic Encryption over Neural Network Graphs for Vision and Private Strawberry Yield Forecasting. *Sensors* **2022**, *22*, 8124. [CrossRef] [PubMed]
14. Li, X.; Gao, H.; Zhang, J.; Yang, S.; Jin, X.; Choo, K.-K.R. GPU Accelerated Full Homomorphic Encryption Cryptosystem, Library, and Applications for IoT Systems. *IEEE Internet Things J.* **2024**, *11*, 6893–6903. [CrossRef]
15. Kim, A.; Deryabin, M.; Eom, J.; Choi, R.; Lee, Y.; Ghang, W.; Yoo, D. General Bootstrapping Approach for RLWE-Based Homomorphic Encryption. *IEEE Trans. Comput.* **2024**, *73*, 86–96. [CrossRef]
16. Naito, H.; Matsumoto, M.; Nakano, M.; Oguchi, M. Performance Analysis of Comparison Operations on Homomorphically Encrypted Sensitive Data. In Proceedings of the 13th International Conference on Consumer Electronics, Berlin, Germany, 2–5 September 2023; pp. 70–75.
17. Rezaeibagha, F.; Mu, Y.; Huang, K.; Chen, L.; Zhang, L. Authenticable Additive Homomorphic Scheme and its Application for MEC-Based IoT. *IEEE Trans. Serv. Comput.* **2023**, *16*, 1664–1672. [CrossRef]
18. Ramesh, S.; Govindarasu, M. An Efficient Framework for Privacy-Preserving Computations on Encrypted IoT Data. *IEEE Internet Things J.* **2020**, *7*, 8700–8708. [CrossRef]
19. Rezaeibagha, F.; Mu, Y.; Huang, K.; Chen, L.; Zhang, L. Toward Secure Data Computation and Outsource for Multi-User Cloud-Based IoT. *IEEE Trans. Cloud Comput.* **2023**, *11*, 217–228. [CrossRef]
20. Zhang, Y.; Zhang, Z.; Ji, S.; Wang, S.; Huang, S. Conditional Proxy Re-Encryption-Based Key Sharing Mechanism for Clustered Federated Learning. *Electronics* **2024**, *13*, 848. [CrossRef]
21. Albakri, A.; Alshahrani, R.; Alharbi, F.; Ahamed, S.B. Fully Homomorphic Encryption with Optimal Key Generation Secure Group Communication in Internet of Things Environment. *Appl. Sci.* **2023**, *13*, 6055. [CrossRef]
22. Microsoft Research Microsoft SEAL. Available online: <https://github.com/Microsoft/SEAL> (accessed on 15 January 2024).
23. Costache, A.; Smart, N.P. Which Ring Based Somewhat Homomorphic Encryption Scheme is Best? *Lect. Notes Comput. Sci.* **2016**, *9610*, 325–340.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.