XXVI ENMC
ENCONTRO NACIONAL DE MODELAGEM COMPUTACIONAL
XIV ECTM
ENCONTRO DE CIÊNCIA E TECNOLOGIA DE MATERIAIS
2023

A comparison between Partially Homomorphic and Fully Homomorphic Encryptions applied to an iterative numerical method

Renato J. P. Borseti[1] - renatojp@posgrad.lncc.br
Matheus Dornelles[1] - mdornelles@posgrad.lncc.br
Fábio Borges[1] - borges@lncc.br
[1]National Laboratório Nacional de Computação Científica (LNCC-MCTI) - Petrópolis, RJ, Brazil

Abstract. *Cloud processing has drawn the attention of scientific peers to companies, as it enables the delivery of on-demand computing services, such as storage and processing power. For some applications, the biggest obstacle to cloud processing is the security of stored and processed data. Encryption has proven to be an excellent choice for handling private data and can be a secure solution for numerical computing in cloud processing, such as solving differential equations. In this article, we explore two homomorphic cryptography schemes to compute a widely used numerical method to solve real-world problems, which we split into two models. In the first model, we use the partially homomorphic encryption scheme applied to the Runge-Kutta numerical method of 4^{th} order. We use the fully homomorphic encryption scheme in the second model applied to the exact numerical method. The main idea is to encrypt the data and send them to the cloud with a modified Runge-Kutta numerical method to deal with encrypted data. Consequently, Runge-Kutta returns an encrypted output, and the client can only decrypt it. The data obtained by encrypted processing have an excellent approximation of the data in the traditional model, i.e., without encryption, with an error of 10^{-7} in the most extreme case. However, the execution times of the encrypted models were much faster than those of the traditional model.*

Keywords: *Runge-Kutta, Homomorphic Encryption, Cloud Processing*

1. INTRODUCTION

With the advent of cloud processing, it became possible to outsource some tasks such as storage, contingency sites, and remote processing. The facilities found in cloud computing have attracted the attention of scientists looking for high processing power associated with low cost. Despite all the advantages it has brought, cloud processing has some inherent drawbacks.

The biggest one is the lack of privacy, as there is a risk of the data being eavesdropped by the server or leaked to third parties. Despite the guarantee of confidentiality, integrity, and data availability, the famous CIA triad of information security, data can be intercepted during shipment or leaked by a malicious employee.

An alternative that can be used to achieve this is to encrypt the data before sending them to the cloud using a particular type of cryptography called Homomorphic Encryption. This technique allows the data to be processed while being encrypted. In this way, a user may upload its encrypted data to the cloud, process it despite a possible eavesdropper, download the encrypted data obtained after the processing, and then decrypt them to retrieve the results in the security of his local machine.

This work explores two homomorphic cryptography methods applied to one of the most widely used numerical methods in computational modeling, the fourth-order Runge-Kutta method. This work is organized as follows. In Section 2, the definitions regarding numerical analysis and the Runge-Kutta methods are briefly described. Homomorphic encryption schemes are presented in Section 3, while Section 4 reports the proposed usage of such schemes in the computation of the Runge-Kutta method. The experimental results are presented in Section 5. Moreover, finally, the conclusions for the work are drawn in Section 6.

2. Runge-Kutta methods

The Runge-Kutta method is an improvement of Euler's method Atkinson (2004) and a general class of approximations characterized by expressing the solution in terms of the derivative of $f(x, y)$ evaluated with different arguments Vries (1948). This numerical method is an extremely popular and powerful tool for solving ordinary differential equations because it avoids the need for higher-order derivatives, as Taylor's method asks. There is a family of Runge-Kutta methods that is not used in our work, but can be found in detail at Atkinson (2004) and Vries (1948).

The Runge-Kutta methods have the general form Atkinson (2004):

$$y_n = y_{n-1} + h \cdot F(x_{n-1}, y_{n-1}, h),\ n \geq 0. \tag{1}$$

The quantity $F(x_{n-1}, y_{n-1}, h)$ can be thought of as a kind of *average slope*.

2.1 Runge-Kutta applications

The Runge-Kutta method was used to solve nonlinear partial differential equations, as demonstrated by Kumar (1977). Some problems in the real world can be solved using ordinary differential equations, but in most cases, these problems do not contain pristine or precise data. Therefore, to accommodate the impreciseness, the concepts of fuzzy sets and intuitionistic fuzzy sets were introduced. Nirmala et al. used Runge-Kutta to find multiple numerical solutions to the intuitionistic fuzzy sets in Nirmala (2018).

3. Homomorphic Encryption

Homomorphic encryption (HE) is a kind of encryption that allows third parties to perform certain types of operation on user's encrypted data without decrypting them. Performing mathematical operations on the ciphertext is equivalent to other operations on the plaintext.

Based on the number of mathematical operations, HE can be categorized in three groups, *Partially Homomorphic Encryption* (PHE), *Somewhat Homomorphic Encryption* (SHE) and *Fully Homomorphic Encryption* (FHE). The main differences between them are the number of mathematical operations and the number of times one can use the mathematical operations. In

PHE only **one** operation is possible in encrypted data, i.e., either addition or multiplication, with an unlimited number of times. In SHE schemes, both operations may be used, however, the number of times is limited. The FHE schemes allow for the usage of both operations an unlimited number of times Kim (2019).

3.1 Partially Homomorphic Encryption

In our work we used the Paillier cryptosystem as PHE. The Paillier cryptosystem was invented by Pascal Paillier in 1999 Paillier (1999). First and foremost, we need to define the partially homomorphic property, as defined in Paillier (1999), a cryptosystem is said to be *partially homomorphic* if it has the following property.
Let ε be an encryption algorithm, and let $\otimes$ and $\oplus$ be the group operations in the corresponding group of ciphertexts and plaintexts, respectively. If

$$\varepsilon(m) \otimes \varepsilon(m') = \varepsilon(m \oplus m'), \tag{2}$$

and

$$\varepsilon(m)^k = \varepsilon(k * m), \tag{3}$$

for all distinct m, m' in the set of plaintexts, we say that ε_D has a *homomorphic property*.

The Paillier cryptosystem utilizes the multiplicative group $\mathbb{Z}^*_{n^2}$, given by

$$\mathbb{Z}^*_{n^2} = \{1 \le i \le n | \text{gcd}(n^2, i) = 1\},$$

where n is the usual product of two large primes p, q such that $gcd(n, \varphi(n)) = 1$. The function of the encryption algorithm is closely related to n'th residues, and is given by

$$\begin{aligned} \gamma_g : \quad & \mathbb{Z}_n \quad \times \mathbb{Z}^*_n \to \mathbb{Z}^*_{n^2} \\ & (m, r) \quad \longmapsto g^m r^n (mod\ n^2). \end{aligned}$$

Paillier states that if the order of g is a nonzero multiple of n in $\mathbb{Z}^*_{n^2}$, then γ_g is bijective.

Algorithm 1 shows the Paillier's key generation scheme. Note that two primes are used, and it is recommended to choose big primes p, q. In this work, the primes were chosen to have 2048 bits length using the GMP library for the C language.

Algorithm 1 Paillier's key generation.

```
Require: Two primes $p, q$ such that $\gcd(pq, \varphi(pq)) = 1$.
Ensure: Public key $K_{pub} = (n, g)$ and Private key $K_{priv} = (\lambda, \mu)$.
  Compute $n = pq$.
  Compute $\lambda(n) = \varphi(n) = (p-1)(q-1)$.
  Compute $\mu = \lambda^{-1} \mod n$.
  Choose $g = (n+1)$.
  return $K_{pub} = (n, g)$, $K_{priv} = (\lambda, \mu)$.
```

Algorithm 2 shows the Paillier encryption scheme, which receives as input the public key K_{pub} and the message m, thus computing the ciphertext c.

Algorithm 2 Paillier's encryption algorithm.

```
Require: $K_{pub}$ and $m \in \mathbb{Z}_{26}$ where $m$ is the message.
Ensure: The encrypted text $c \in \mathbb{Z}^*_{n^2}$.
  Choose randomly $r \in \mathbb{Z}^*_n$ such that $\gcd(r, n) = 1$.
  Compute $c = \gamma_g(m, r) = g^m r^n \mod n^2$.
  return $c \in \mathbb{Z}^*_{n^2}$.
```

Algorithm 3 shows the Paillier decryption scheme, which receives as input the private key K_{priv} and the encrypted text c, returning the decrypted message m. In this paper, the Runge-Kutta 4^{th} order method was adapted to work with the Paillier cryptosystem. The client encrypts his data and sends them to the server, which will process it without needing to decrypt it and then send that data back to the client.

Algorithm 3 Paillier's decryption algorithm.

```
Require: $K_{priv}$ and $c \in \mathbb{Z}^*_{n^2}$.
Ensure: The decrypted text $m \in \mathbb{Z}_n$.
  Compute $L = \frac{(c^\lambda \mod n^2) - 1}{n}$.
  Compute $m = L \cdot \mu \mod n$.
  return $m \in \mathbb{Z}_n$.
```

3.2 Fully Homomorphic Encryption

Solutions for homomorphic encryption which allow one operation, such as addition, have been known for decades, for example, the Paillier cryptosystem, previously shown. However, a solution which allows two operations, i.e., addition and multiplication, enables the computation of any circuit, and thus this solution is referred to as *fully homomorphic encryption* (FHE). The first FHE solution was proposed by Gentry (2009) and many improvements have been made over the years.

The essence of FHE is simple, according to Gentry (2009), given ciphertexts that encrypt $\pi_1, \ldots, \pi_t$, FHE should allow anyone to output a ciphertext that encrypts $f(\pi_1, \ldots, \pi_t)$ for any desired function f. Gentry noticed, in his work, that he can apply some operations on the ciphertext before the decryption becomes incorrect; this limitation on the number of operations is attributed to the *noise* in the ciphertext. Noise is a small random value that is added to the ciphertext during encryption and increases when operations are performed Halevi (2020). To solve this problem, Gentry proposed a bootstrapping technique Gentry (2009) that creates a circuit using a decryption algorithm for the encryption scheme. If the decryption circuit is inexpensive enough to evaluate, then the output ciphertext will have less noise than the input Halevi (2020).

In this work, we used the HElib library to implement the FHE scheme on the Runge-Kutta numeric method. HElib is an open-source library that implements two HE schemes. In this paper, we used the Approximate Number scheme of Cheon-Kim-Kim-Song (CKKS) CKKS (2017), but HElib has the Brakerski-Gentry-Vaikuntanathan (BGV) BGV (2011) scheme and Halevi (2020). These schemes are all lattice-based cryptographic dependent on the hardness of the *Ring Learn with Errors* problem (RLWE) and support addition and multiplication homomorphism. However, BGV only performs computations with integers BGV (2011), while CKKS can perform computations with complex numbers with limited precision CKKS (2017).

Regarding noise, BGV is more difficult than CKKS because in the BGV scheme the noise level needs to be kept in mind at every step of the algorithm BGV (2011).

The message m is a vector of values with which the computations will be performed. It is first encoded in a plaintext polynomial $p(x)$ and then encrypted using a public key. CKKS works with polynomials because they provide a good trade-off between security and efficiency CKKS (2017). After encrypting the message, CKKS can perform some operations such as addition, multiplication, and rotation. However, multiplication can considerably increase the noise to ciphertext, so only a limited number of multiplications are allowed. This number may be increased by modifying the CKKS parameters, increasing the depth of the circuits, but this will increase the execution time in general.

4. Runge-Kutta 4^{th} order encrypted

In this section, we demonstrate how to encrypt Runge-Kutta methods using Paillier cryptosystem and FHE using the HElib library. The convergence of the Runge-Kutta method is not affected because the numerical method is not changed, just adapted to process encrypted data according to each scheme.

4.1 Runge-Kutta using Paillier cryptosystem

The main idea is to send encrypted data to a server that will execute Runge-Kutta to compute the output y using the homomorphic property of the Paillier cryptosystem as shown in Figure 1.

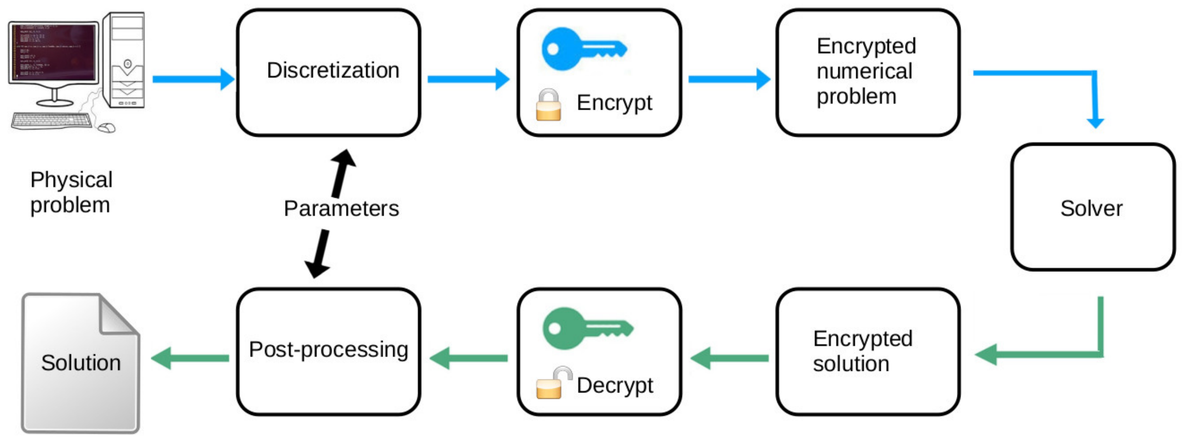

Figure 1: Using Runge-Kutta methods with Paillier cryptosystem in a cloud.

We use equation eq. (4) as a basis for developing the mathematical model that will be used to build encrypted models.

$$\frac{dy}{dx} = x + y, \tag{4}$$

with the initial conditions $x_0 = 0.0,\ y(x_0) = 2.0$. The analytic solution for equation eq. (4) is

$$y(x) = 3 * e^x - x - 1, \tag{5}$$

which will be used to compare with the results from the Runge-Kutta 4^{th} order method with the Paillier cryptosystem.

Algorithm 4 shows the computation of equation eq. (4) using the Paillier cryptosystem, where the homomorphic properties eq. (2) are used, and $\varepsilon(x)$ and $\varepsilon(y)$ are the encryption of variables x and y, respectively, using the algorithm 2.

Algorithm 4 Function using Paillier cryptosystem.

```
Require: $\varepsilon(x)$ and $\varepsilon(y) \in \mathbb{Z}_{n^2}^*$.
Ensure: An encrypted function $f(\varepsilon(x), \varepsilon(y))$.
  Compute $f(\varepsilon(x), \varepsilon(y)) = \varepsilon(x) * \varepsilon(y)$.
  Compute $f(\varepsilon(x), \varepsilon(y)) = f(\varepsilon(x), \varepsilon(y)) \mod n^2$.
  return $f(\varepsilon(x), \varepsilon(y))$.
```

This work proposes the usage of algorithm 2 to encrypt all data and then send them to the server, where the calculations will be done by algorithm 5, i.e., the Runge-Kutta 4^{th} order using the Paillier cryptosystem. The encrypted results of all operations are then returned to the client, who can now safely decrypt the results on his local machine, as shown in fig. 1.

The client must encrypt the data x_0, y_0, h, $h/2$, and send them to the server together with the unencrypted data h, n^2. Therefore, only the client can decrypt the data and get the solution because only the client will have access to the decryption key. It is unnecessary and ill advised to send any key to the server, given that the idea of this process is for the algorithm to handle encrypted data without decrypting it. The server algorithm will handle the encrypted data and return an encrypted output because it is operating in a cryptographic domain.

Algorithm 5 Runge-Kutta 4^{th} order using Paillier cryptosystem.

```
Require: $\varepsilon(x_0), \varepsilon(y_0), n^2, \varepsilon(h), h, \varepsilon(h_{half}), \delta$.
Ensure: An encrypted function $\varepsilon(y_N)$.
  inv$_2$ $\leftarrow$ multiplicative inverse of $2 \in \mathbb{Z}_{n^2}^*$.
  inv$_6$ $\leftarrow$ multiplicative inverse of $6 \in \mathbb{Z}_{n^2}^*$.
  for $i = 0, 1, 2, \ldots, \delta$ do
    Compute $K_0 = f(\varepsilon(x_i), \varepsilon(y_i))$.
    $K_1 = [f(\varepsilon(x_i) * \varepsilon(h_{half}), \varepsilon(y_i) * (K_0)^{\text{inv}_2})]^h$.
    $K_2 = [f(\varepsilon(x_i) * \varepsilon(h_{half}), \varepsilon(y_i) * (K_1)^{\text{inv}_2})]^h$.
    Compute $\varepsilon(x_{i+1}) = \varepsilon(x_i) * \varepsilon(h)$.
    $K_3 = [f(\varepsilon(x_{i+1}), \varepsilon(y_i) * K_2)]^h$.
    $2K_1 = K_1^2$.
    $2K_2 = K_2^2$.
    Compute $\varepsilon(c) = (K_0 + 2K_1 + 2K_2 + k_3)$ mod $n^2$.
    $\varepsilon(y_{i+1}) = [\varepsilon(y_i) * \varepsilon(c)^{\text{inv}_6}]$.
  end for
  return $\varepsilon(y_N)$.
```

Algorithm 5 receives the encrypted data as input. The multiplicative inverses of 2 and 6 are computed because the primes p and q have 2048 bits length, so gcd$(n^2, 2)$ = gcd$(n^2, 6) = 1$ where $n = p * q$. Although the Runge-Kutta 4^{th} order method has been adapted to receive and handle encrypted data using the Paillier cryptosystem, the numerical method is still the same.

4.2 Floating-point map to integer

In our work, all floating-point numbers are converted to integers to facilitate cryptography. Floating-point numbers can be expressed as

$$a = a_0 \times 10^0 + a_1 \times 10^{-1} + \ldots + a_n \times 10^{-n},$$

where n is a non-negative integer and a_i is a positive integer. To convert a floating-point number into an integer, the function $f_m(a)$ is used. It can be defined by

$$f_m(a) = a_0 a_1 a_2 \ldots a_n.$$

The function $f_m(a)$ simply put together the a_i terms and creates an integer number from a floating-point number, e.g., the number $a = 0,12$ will become $f_m(a) = 12$. In this example, the idea is the same as multiplying the number a by 10^2.

After the calculations in the encrypted domain, the numbers can be recovered using an inverse mapping function, as demonstrated in Zhao (2016). The inverse mapping function f_m^{-1} is defined by

$$f_m^{-1}(a) = \sum_{i=0}^{n} \frac{f_m(a_i)}{10^i}.$$

Now it is possible to use floating-point numbers as input for the hypothetical model proposed and recover any floating-point numbers that come with the solution.

4.3 Runge-Kutta using HElib library

We used the same function and configuration for Runge-Kutta presented before, the only modification is the encryption method. Here, we used HElib instead of Paillier cryptosystem, so we need to adjust some HElib parameters using the context object Halevi (2020).

HElib provides a builder pattern which can be used to initialize the context object, the most important ones are listed below.

- *m*: the cyclotomic index. For CKKS, m must be a power of 2. As m increases, we get more security and the size of the ciphertext increases; however, the performance degrades.
- *bits*: specifies the number of bits in the ciphertext modulus. As the number of bits increases, we get less security, but we can perform deeper homomorphic computations.
- *precision*: is the number of bits of precision when data is encoded, encrypted, or decrypted. As precision increases, the depth of homomorphic computations decreases.
- *c*: the number of columns in key-switching matrices. As c increases, we get a little more security, but performance degrades and the memory requirement for the public key increases. c must be at least 2 and it is not recommended to set c higher than 8 by Halevi (2020).

HElib provides some examples and a table that lists the settings for m, *bits*, and c. We have adjusted this builder for 128 bits of level security, as it is recommended a security level of 100

bits or more is recommended Halevi (2020). Algorithm 6 shows the Runge-Kutta computation using HElib.

Algorithm 6 Runge-Kutta 4^{th} order method using HElib.

```
Require: $\varepsilon(x_0), \varepsilon(y_0), \varepsilon(h), n$ as vector and $f(x, y)$.
Ensure: The result of the ODE $\varepsilon(y_n)$
  for $i = 0$ to $n$ do
    $K_0 = \varepsilon(h) \cdot f(\varepsilon(x_i), \varepsilon(y_i))$
    $K_0^* = 0.5 \cdot K_0$
    $\varepsilon(h)^* = 0.5 \cdot \varepsilon(h)$
    $K_1 = \varepsilon(h) \cdot f(\varepsilon(x_i) + \varepsilon(h)^*, \varepsilon(y_i) + K_0^*)$
    $K_1^* = 0.5 \cdot K_1$
    $K_2 = \varepsilon(h) \cdot f(\varepsilon(x_i) + \varepsilon(h), \varepsilon(y_i) + K_1^*)$
    $K_3 = \varepsilon(h) \cdot f(\varepsilon(x_{i+1}), \varepsilon(y_i) + K_2)$
    $2K_1 = 2.0 \cdot K_1$
    $2K_2 = 2.0 \cdot K_2$
    $\varepsilon(y_{i+1}) = \varepsilon(y_k) + 0.166 \cdot (K_0 + 2K_1 + 2K_2 + K_3)$
  end for
  return $\varepsilon(y_n)$.
```

4.4 Test Environment

All experiments were executed on an Ubuntu 20.04 LTS 64 bits Linux, GDM (Gnome Display Manager) disabled, with Intel Core $i7$ architecture. The machine is composed of an Intel $i7 - 8565U$ processor with 1.8GHz, 4 physical cores, and a total of 8 GB of RAM. The compilers used were gcc version 9.3 and G++ version $9.3.0$ with HElib version $2.1.0$ and GMP version $6.2.1$.

5. Experimental results

We built three C++ codes, one for the original fourth-order Runge Kutta code, another using the adaptation to support encrypted data from the Paillier cryptosystem as shown in Algorithm 5, and another using the HElib library for FHE as shown in Algorithm 6. run 1000 times and an average output was measured using 16-digit precision. The average output and input are shown in Figure 2, with the initial conditions given by Eq. four and h = 0.125 with the value of x in the range $0 \leq x \leq 3$. We are using 2048 bits in length for encrypted data using Algorithm 2, but the key length has not shown any change in output.

The data shown in Figure 2 are dimensionless because our physical model was created to test both algorithms. The green line is the analytical output, the blue line with stars is the original fourth-order Runge-Kutta output, the red triangles represent the fourth-order Runge-Kutta with the Paillier cryptosystem using a key-length output of 2048 bits, and the black line represents the fourth-order Runge-Kutta with FHE. The Runge-Kutta with the Paillier cryptosystem is smaller than the other codes on the first input due to the function that converts a float to an integer by doing some rounding on the float number, so for the input of small values, we lose some precision in the output. However, the output is the same in most cases. Runge-Kutta with

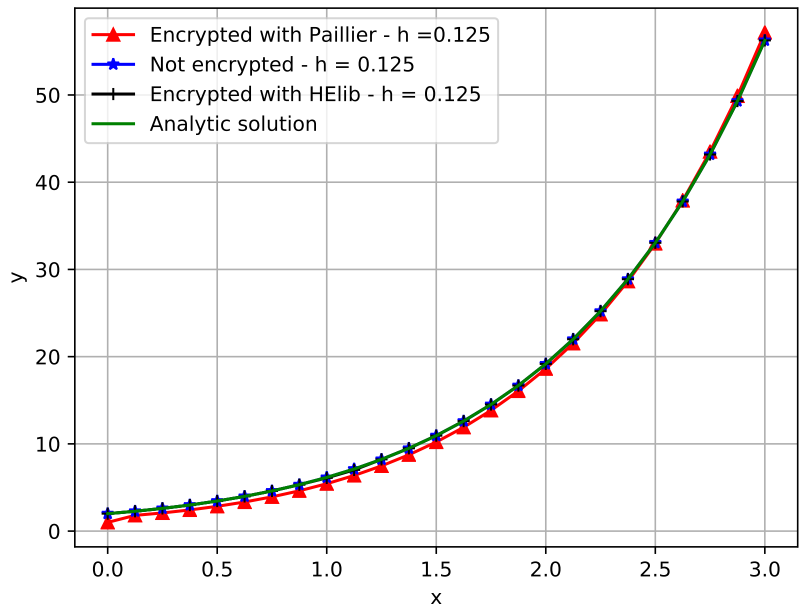

Figure 2: Analytic solutions vs. algorithms of Runge-Kutta 4^{th} order.

an FHE scheme using a 128-bit security level has the same precision as the original, since both work with float; however, FHE has some performance issues.

Table 1 shows the error between the algorithm output and the analytical solution as a function of the security level. To change the security level, we need to modify the Halevi (2020) context builder to give the algorithm the deeper circuitry it needs to avoid noise and return the correct output. However, when the context builder was modified, we noticed that accuracy was affected to maintain the depth of the Halevi (2020) circuit.

Security level (bits)	**80**	**112**	**128**
Error	2.13×10^{-7}	1.04×10^{-10}	1.32×10^{-14}

Table 1: Errors in function of security level.

The average execution time of the traditional fourth-order Runge-Kutta code for the given problem was 9.075×10^{-4} seconds. However, the average Runge-Kutta time with the Paillier cryptosystem using a 1024-bit key was 9.1×10^{-3} seconds; for a 2048-bit key, the average time was 54.6×10^{-3} seconds. Note that the growth of the running time follows the size of the key in the Paillier scheme or the security level in the CKKS scheme. It is because the size of the ciphered vector arrays is directly related to the key size in the Paillier scheme, while the key size in the CKKS scheme is directly related to the security level Gentry (2009).

The experiments show an almost exponential increase with increasing key length. Figure 3, on the right, compares the execution times of the fourth-order Runge-Kutta with FHE with 80 bits, 112 bits, and 128 bits of security level. The average time is 178.635 seconds for 80 bits and 235.64 seconds for 112 bits. We use an 80-bit security level because it is equivalent to a 1024-bit key length by NIST Borges (2020), while a 112-bit security level is comparable to a 2048-bit key length. This equivalence allows for a comparison of the Paillier cryptosystem key's length and the algorithm's running time. The 128-bit security level is comparable to the 3072-bit key length of NIST Borges (2020).

It is the most accurate due to the context builder configuration, having an average time of

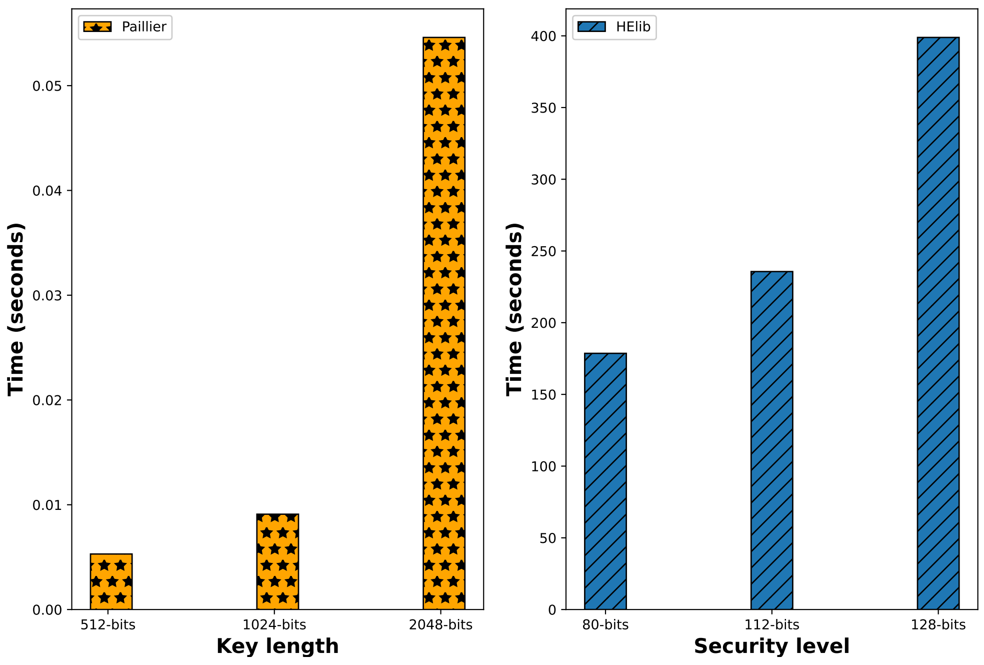

Figure 3: Run-time comparison between Paillier and HElib implementation.

400.73 seconds, as shown in Figure 3. This code has excellent accuracy but is slower than the others due to the Bootstrapping technique. This technique allows several homomorphic operations to be performed in the CKKS scheme without increasing the inserted noise to make decryption impossible Gentry (2012).

One way to optimize the code is to use high-performance computing (HPC) techniques (Borges, 2017, 2012). According to Souto (2018), some High-Performance Computing techniques can help accelerate the process when applied to numerical methods and have shown promising results. Another possibility is the development of specific hardware on an open platform such as ARM Yokoyama (2019). However, this technique makes code dependent on specific equipment to maximize its results, reducing portability.

6. Conclusion

Paillier's scheme limits the amount of mathematical operations and is not quantum-safe, but, on the other hand, it delivers excellent code execution time. However, the CKKS scheme is quantum safe and allows for an almost infinite number of homomorphic operations, but at the cost of high code execution time caused, mainly, by the Bootstrapping technique since it requires the reduction of an integer module another integer Gentry (2012). There are FHE schemes that do not use Bootstrapping, but to circumvent this technique, they implement the use of GLWE (General Learning with Errors), but this limits the number of operations Alperin-Sheriff (2014), which made it impossible to use in this work. During the tests of this work, we saw that both schemes did not affect the convergence of the method, and the data obtained were very close to the data obtained in the unencrypted model; with the 128-bit security level, we obtained an error of 10^{-7} between the original code and the encrypted code. To work around the problem of high execution time, we recommend using high-performance computing (HPC) techniques to optimize the code. However, we recommend avoiding hardware-specific flags for cloud computing to ensure code portability.

Acknowledgments

The authors thank the Coordination for the Improvement of Higher Education Personnel (CAPES) and LNCC.

REFERENCES

Ngenzi, A. (2014),“Applying mathematical models in cloud computing: A survey”,*IOSR Journal of Computer Engineering*, vol 16, 36-46.

Atkinson, K.; Han, W. (2004), Elementary Numerical Analysis, John Wiley & Sons, Ney Work.

Borges, F.; Lara, P.; Portugal, R. (2017), “Paralllel alogrithms for modular multi-exponentiation”, *Applied Mathematics and Computation*, vol 292, 406-416.

Borges, F.; Reis, P. R.; Pereira, D. (2020), “A comparison of security and tis performance for key agreements in post-quantum cryptography”, *IEEE Access*.

Brakerski, Z.; Gentry, C.; Vaikuntanathan, V. (2011), “Fully homomorphic encryption without bootstrapping”, *IACR Cryptology ePrint Archive*, doi: 2011:277.

Cheon, J.; Kim, A.; Kim, M.; Song, Y. (2017), “Homomorphic encryption for arithmetic of approximate numbers”, *Advances in Cryptology*, 409-437, doi: 0.1007/978-3-319-70694-8_15.

DeVries, P. L. (1948), A First Course in Computational Physics, John Wiley & Sons, Ney Work.

Gentry, C. (2009), “A fully homomorphic encryption scheme”, doctor in philosophy, Stanford University, USA.

Gentry, C.; Halevi, S.; Smart N. P. (2012), “Better Bootstrapping in Fully Homomorphic Encryption”, *Public Key Cryptography - PKC 2012*, vol 7293, doi: 10.1007/978-3-642-30057-8_1.

Halevi, S.; Shoup, V. (2020), “HElib design principles”, IBM Research.

Kumar, A.; Unny, T. (1977), “Application of Runge-Kutta method for the solution of non-linear partial differential equations”, *Applied Mathematical Modelling*.

Lara, P.; Borges, F.; Portugal, R.; Nedjah, N. (2012), “Parallel modular exponentiation using load balancing without precomputation”, *J.Comput. Syst. Sci.*, vol 78, 575-582.

Nirmala, V.; Parimala, V.; Rajarajeswari, P. (2018), “Application of Runge-Kutta method for finding multiple numerical solutions to intuitionistic fuzzy differential equations”, *Journal of Physics*, vol 1139.

Paillier, P. (1999), “Public-key cryptosystems based on composite degree residousity classes”, *Advances in Cryptology - EUROCRYPT*, vol 1592, doi: 10.1007/3-540-48910-X_16.

Shrestha, R.; Kim, S. (2019), “Role of blockchain technology in IoT applications”, *Advances in Computers*, 293-331, doi: 10.1016/bs.adcom.2019.06.002.

Souto, R. P.; Welter, M. E. S.; Melo, M. S.; Osthoff, C.; Borseti, R. J. P.; Rodrigues, L. F.; Dias, P. L.; Vigilant, F. (2018) “New computational developments on chemistry module of BRAMS numerical weather prediction”, *Proceeding Series of the Brazilian Society of Computational and Applied Mathematics*, vol 6, doi: 10.5540/03.2018.006.02.0305.

Yokoyama, D.; Schulze, B.; Borges, F.; McEvoy, G. (2019), “The survey on arm processors for hpc”, *The Journal of Supercomputing*, vol 75, 7003-7036, doi: https: 10.1007/s11227-019-02911-9.

Zhao, W.; Hong, M.; Wang, P. (2016), “Homomorphic encryption scheme based on elliptic curve cryptography for privacy protection of cloud computing”, *IEEE Xplore*, 152-157, doi: 10.1109/BigDataSecurity-HPSC-IDS.2016.51.

Alperin-Sheriff, J.; Peikert, C. (2014), Faster Bootstrapping with Polynomial Error, *Advances in Cryptography – CRYPTO 2014*, vol 8616, doi: 10.1007/978-3-662-44371-2_17