International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Homomorphic Encryption with SEAL: Investigating Security and Performance

Kirti Dinkar More*[1], Dr. Dhanya Pramod[2] , Dr. Rahul Ashokrao Patil[3]

Submitted: 29/01/2024 **Revised**: 07/03/2024 **Accepted**: 15/03/2024

Abstract: Security is a major concern these days because of the increasing use of smart technologies and the Internet. Security is required to preserve the confidentiality, integrity, and availability of the resources over the network [2]. Homomorphic encryption (HE) is privacy preserving technique for sharing of data with cloud backend securely [20]. It offers a safe environment where operations on previously encrypted data can be carried out and the outcomes will be the same as for the original data [3]. This work serves as a demonstration of practical use of homomorphic encryption, which can be used to guarantee data security and computation. We present an analysis of the fully homomorphic encryption library known as SEAL (Simple Encrypted Arithmetic Library) in this work. Three SEAL supported schemes - Brakerski-Gentry- Vaikuntanathan (BGV), Brakerski-Fan- Vercauteren (BFV), and Cheon-Kim-Kim- Song (CKKS) -with default and custom degrees of polynomial are examined in relation to the outcomes produced for a range of parameters.

Keywords: *BFV, BGV, CKKS, Fully Homomorphic Encryption, SEAL, Security.*

1. Introduction

This Security is now of utmost importance due to the increasing use of smart technologies and the Internet. The availability, integrity, and confidentiality of the resources over the network depend on security [2]. Databases can store encrypted data, but processing such data requires first decryption of it; once decrypted, the data might not be safe. When data is already encrypted, homomorphic encryption creates a safe environment where operations on the encrypted data yield identical outcomes to those on the original data [3]. Homomorphic encryption was first proposed in 1978 by Michael Dertouzos, Leonard Adleman, and Ronald Rivest [4]. There are several homomorphic encryption schemes that vary in the number of operations that can be carried out on the encrypted data. The first one is Partially Homomorphic Encryption (PHE) in which one type of operation, addition or multiplication, can be performed on encrypted data an unlimited number of times. The second scheme is called Somewhat Homomorphic Encryption (SWHE), which restricts the number of computations and only permits specific operations on encrypted data because noise makes the ciphertext size increase with each step. This scheme is practically more feasible. Third one is Fully Homomorphic Encryption which allows any number of computations on cipher text. But practically fully homomorphic encryptions have lot of overhead and in terms of computations it is expensive [5]. Therefore, investigating fully homomorphic encryption schemes is our goal. In September 2009, Craig Gentry proposed the first fully homomorphic encryption scheme [1].

A compact fully homomorphic encryption (FHE) that enable the computation of multiple functions on encrypted data has proven to be much more challenging to develop than the numerous cryptosystems that have been proposed over the years. Since from the Genrty's work there have been roughly three generations of FHE development. The original Gentry approach, which involves ideal lattices[6], and scheme of van Dijk et al. which makes use of integer arithmetic[7] are both included in the first generation. The issue of rapidly growing noise is faced by both of these schemes. The study of Brakerski-Vaikuntanathan [8] and Brakerski et al. [9] in 2011 brought about the second generation, which was distinguished by noticeably superior techniques for lowering noise and increasing efficiency. The Gentry et al. [48] method, which had a slightly different pattern of noise growth, was the starting point for the third generation. Both asymmetric multiplication and asymmetric noise growth were present in this scheme [10][11].

2. Homomorphic Encryption

Four functions are typically included in a public-key homomorphic encryption scheme: KeyGen, Encrypt, and Decrypt, along with an Evaluate process for computing on encrypted data [14].

Key generation: To encrypt plaintext, the client will create a pair of keys, a public key (PK) and a secret key (SK).

[1]*MVP samaj's K. T. H. M. College, Nashik, India*
[2] *Symbiosis Centre for Information Technology, Symbiosis International (Deemed) University, Pune, India.*
Email: dhanya@scit.edu
ORCID ID : 0000-0003-3451-9794
[3] *MVP samaj's K. T. H. M. College, Nashik, India*
Email:patilra@rediffmail.com
** Corresponding Author Email: kirtimore@kthmcollege.ac.in*

Encryption: The plain text (PT) is encrypted by client using the secret key SK to create ESK (PT). This cipher text (CT), along with the public key (PK), is then sent to the server.

Evaluation: The server contains a function called f that allows it to evaluate the cipher text CT in accordance with the necessary function while utilizing PK.

Decryption: The client will use its SK to decrypt the generated Eval(f(PT)) and obtain the original result.

Two characteristics of homomorphic encryption are its primary ones [13],

Homomorphic additive encryption: If Enc (PT1+PT2) = Enc (PT1) + Enc (PT2), then homomorphic encryption is additive.

Homomorphic multiplicative Encryption: If Enc(PT1 ×PT2) = Enc(PT1) × Enc(PT2), then homomorphic encryption is multiplicative.

Essentially, fully homomorphic encryption has a straightforward structure. Presume that fully homomorphic encryption enables anyone (not just the key holder) to produce a ciphertext that encrypts f(π1,.., πt) for any desired function f, provided that function can be computed effectively. There should be no leakage of any intermediate plaintext values or information about π1,... πt or f(π1,.., πt). Encryption is always used for the inputs, output, and intermediate values [1].

3. Related work

Homomorphic Encryption (HE) is a secure method for exchanging data safely with cloud backend while maintaining privacy. Due to its high memory consumption and computational overhead, HE may not be suitable for use on embedded devices with limited resources. To address this issue, authors have proposed the first HE library for embedded devices named as SEAL-Embedded, based on CKKS approximate homomorphic encryption scheme. High performance CKKS encoding and memory efficiency on embedded devices are all achieved with this newly proposed library, which combines a detailed memory reuse scheme with multiple computational and algorithmic optimizations [20]. The study of various schemes, including BFV, BGV, CKKS, RSA, El-Gamal, and Paillier, is covered by the authors in [12], along with how these schemes are implemented in HE libraries, such as Microsoft SEAL, PALISADE, and HElib. The authors of [13] investigated a number of homomorphic encryption schemes, including Non-interactive Exponential Homomorphic Encryption algorithm (NEHE), Brakerski-Gentry-Vaikuntanathan (BGV), updated ElGamal (AHEE), and Homomorphic Cryptosystem (EHC). In [15] various HE schemes are explored out of which Fully homomorphic encryption schemes are 1) BFV supported in both SEAL and PALISADE 2) BGV implemented in SEAL, PALISADE, and HElib. Leveled homomorphic encryption, or CKKS scheme, is an extended version of SWHE somewhat homomorphic encryption scheme. HElib, HEAAN, SEAL, and PALISADE all has implementation of CKKS. The implementations of partially homomorphic encryption that the authors have provided include Paillier (additive), El-Gamal (multiplicative), and RSA (multiplicative). They employed their own implementations of Paillier, RSA and El-Gamal as partially homomorphic cryptosystems in the emulation since PHE schemes are not implemented in the libraries mentioned. For large plaintext moduli of up to 2048 bits, the authors of [19] provided the comparative benchmark of the well-known homomorphic encryption libraries SEAL, HElib and FV-NFLlib, along with an analysis of their respective performances.

4. Proposed Work

In this article we have given the study of Microsoft SEAL library. In 2015 Microsoft Research created the Simple Encrypted Arithmetic Library (SEAL), a cross-platform software library that is free and open-source and implements several types of homomorphic encryption. It was written in C++ and C#. It doesn't rely on outside libraries to function independently. Users can choose between security levels (128 or 192), degree (1024, 2048, 4096, 8192, or 16384) and plaintext modulus (with no limit), with SEAL, which utilizes the FV cryptosystem [16]. Fully Homomorphic Encryption technique is explored through SEAL. A basic illustration of fully homomorphic encryption is provided in the study. It applies mathematical operations to encrypted data without ever decrypting it, allowing for the observation of various parameters' performance. Table 1 shows the observed values after successful execution of examples given with SEAL. Firstly we explored BFV encryption scheme. It is shown how to use the BFV encryption method to perform basic calculations (a polynomial evaluation) on encrypted integers. Setting the following three encryption parameters is required:

- polynomial modulus degree, or poly_modulus_degree
- coeff_modulus, or coefficient modulus [ciphertext]
- plain_modulus, which is the plaintext modulus particular to the BFV scheme.

It is not possible for the BFV scheme to run random calculations on encrypted data. The "invariant noise budget," also known as the "noise budget," is expressed in bits for each ciphertext. The encryption parameters define the noise budget (initial noise budget) in a newly encrypted ciphertext. The two fundamental operations in BFV are

permitted on parameters which are additions and multiplications. The noise budget is consumed by these homomorphic operations. Additions are typically considered to consume almost no noise budget when compared to multiplications. The most important consideration when selecting suitable encryption parameters is the multiplicative depth of the arithmetic circuit that the user wishes to assess on encrypted data, as noise budget consumption compounds in sequential multiplications.

Computing over encrypted data in the BGV scheme is similar to that in BFV. The purpose of BGV example is mainly to explain the differences between BFV and BGV in terms of ciphertext coefficient modulus selection and noise control. Most of the code is repeated from BFV basics example. In the exploration of BGV scheme as an example, evaluation of the degree 8 polynomial x^8 over an encrypted x over integers 1, 2, 3, 4 is done. One could think of the polynomial's coefficients as inputs in plaintext. Modulo the plain_modulus 1032193, the computation is performed. BGV requires modulus switching to reduce noise growth. Although with modulus switching there can be less noise budget than before, noise budget is utilized at a slower rate. To achieve the optimal consumption rate of noise budget in an application, one needs to carefully choose the location to insert modulus switching and manually choose coeff_modulus.

The evaluation of the polynomial function of the form PI*x^3 + 0.4*x + 1 for a set of 4096 equidistant points in the interval [0, 1] on encrypted floating-point input data x is demonstrated for the CKKS scheme. Many of the key components of the CKKS scheme are illustrated in this example, along with some of its challenges while using it.

It has been observed that scales in ciphertexts increase with multiplication in CKKS in the SEAL code of encoders. Any ciphertext's scale must avoid approaching coeff_modulus's total size, or else it will run out of space to hold the scaled-up plaintext. A "rescale" functioning offered by the CKKS scheme can lower the scale and settle down the scale expansion. One type of modulus switch operation is rescaling. It eliminates the final prime from coeff_modulus as a modulus switch, but as a side effect, it scales down the ciphertext by the removed prime. Carefully choosing primes for the coeff_modulus is more common for the CKKS scheme because the goal is to have complete control over scale modifications. For example, let us consider the following scenario: a CKKS ciphertext has S as the scale, and P as the last prime in the current coeff_modulus (for ciphertext). As is common in modulus switching, rescaling to the next level removes the prime P from the coeff_modulus and modifies the scale to S/P. The number of primes restricts the number of rescalings that can be performed, which in turn restricts the multiplicative depth of the calculation.

The initial scale can be freely selected. Setting the initial scale S and primes P_i in the coeff_modulus to be extremely close to one another can be a sensible approach. After multiplication, ciphertexts with scale S have scale S^2, and after rescaling, they have scale S^2/P_i. S^2/P_i is close to S again if all P_i are nearby S. In this manner, scales remain nearby S all over the computation. A circuit of depth D typically requires rescaling D times, or the ability to eliminate D primes from the coefficient modulus. When there is just one prime left in the coeff_modulus, it needs to be a bit bigger than S by few bits in order to maintain the plaintext's pre-decimal-point value. As a result, selecting the following parameters in the CKKS method is generally advantageous:

- As the first prime in coeff_modulus, select a 60-bit prime. This will yield the highest level of decrypting precision.
- Selecting other 60-bit prime as the final element of coeff_modulus is recommended because it will serve as the special prime and should have the same size as the largest prime.
- Additionally, selecting intermediate primes that are close to one another is advised.

4.1. Rotations

Native vectored operations on encrypted numbers are supported by the BFV,CKKS and BGV techniques (with Batch Encoder). Apart from computing slot-wise, the encrypted vectors can also be rotated cyclically. You can specify the number of steps to rotate left or right. A different kind of unique key known as "Galois keys" is needed for rotations. These can be obtained from the KeyGenerator with ease. No budget for noise is used during rotations. This holds true, though, only in the event that the special prime is at least as big as the other primes. This also applies to relinearization. Rotations in BFV and the CKKS scheme function very much alike. It is up to the user to decide the size of special prime appropriately because Microsoft SEAL does not require the special prime to be any specific size. It is also feasible to evaluate a complex conjugation on a vector of encrypted complex numbers using the CKKS strategy. This is actually a form of rotation, and it also needs Galois keys.

When implementing the BFV scheme in SEAL, it has been demonstrated how to use it to carry out a very basic computation. The calculation used a single coefficient from a BFV plaintext polynomial and was done modulo the plain_modulus parameter. There are two major issues with this approach:

1. Modular arithmetic is rarely used in practical applications; instead, integer or real number

arithmetic is mostly preferred.

2. The plaintext polynomial has a single coefficient. This is ineffective because the large plaintext polynomial will always be fully encrypted.

Regarding the first point raised above, one might wonder why the calculations don't behave like integer arithmetic if one simply increases the plain_modulus parameter until there is no overflow. The issue is that raising plain_modulus both reduces the initial noise budget and increases the consumption of the noise budget. Additional encoding methods that enable further computations with no data type overflow and that can make use of the entire plaintext polynomial are discussed in the encoder's code.

5. Results and Discussion

Table 1 includes the various parameters results for BFV scheme with default degree and custom degree, BGV scheme with default degree and custom degree, CKKS scheme with default degree and custom degree. Here, the parameters that are being compared are Poly_modulus_degree, Coeff_modulus size, Average encrypt, average decrypt, average add, average multiply, average compressed (ZLIB), average serialization of ciphertext (microseconds), average compressed (Zstandard) serialization of ciphertext (microseconds), generation of relinearization keys (microseconds), generation of Galois keys (microseconds).

It can be seen from Figure 1 and 2 that among the three homomorphic encryption schemes with default and custom chosen degree, BGV requires the least average encryption time, while for decryption, CKKS requires the least time compared to BFV and BGV

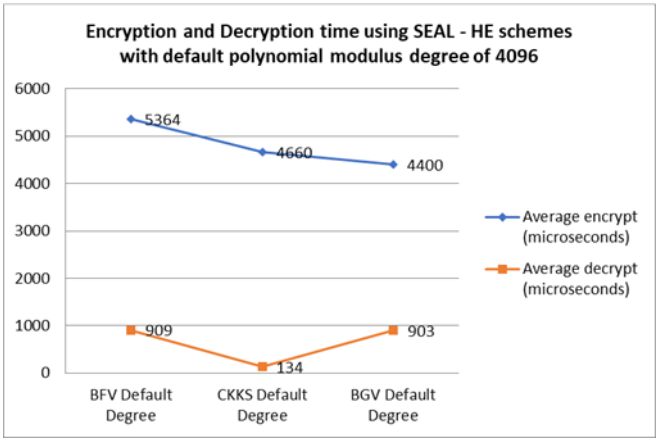

Fig 1. Encryption and Decryption timings of HE schemes using SEAL with default polynomial modulus degree of 4096.

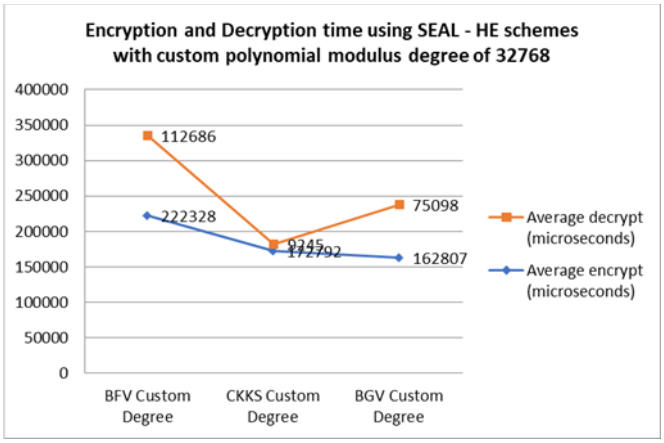

Fig 2. Encryption and Decryption timings of HE schemes using SEAL with custom polynomial modulus degree of 32768

Table 1. Encryption parameters values/time (microseconds) for BFV, CKKS and BGV generated through SEAL library.

Encryption Parameters	Values/time in microseconds																	
	Scheme	BFV Default Degree			BFV Custom Degree	CKKS Default Degree			CKKS Custom Degree	BGV Default Degree			BGV Custom Degree
	Poly_modulus_degree	4096	8192	16384	32768	4096	8192	16384	32768	4096	8192	16384	32768
	Coeff_modulus size	109	218	438	881	109	218	438	881	109	218	438	881
	Relinearization keys generation (microseconds)	7216	44683	249348	1686845	110820	41681	359743	1531497	20185	35875	239879	1446323
	Galois keys generation (microseconds)	133946	980018	6895429	51619519	1366813	940531	13058703	284307687	163537	1480016	45475109	45657621
	Average encrypt (microseconds)	5364	17635	56512	222328	4660	14971	104622	172792	4400	11059	39010	162807
	Average decrypt (microseconds)	909	4281	18772	112686	134	565	2275	9245	903	2798	13306	75098
	Average add (microseconds)	73	338	1451	6112	67	298	1310	8045	68	265	1193	5215
	Average multiply	10989	48206	233837	1164193	3598	1477	6415	22256	10918	9431	42947	303249
	Average Compressed (ZLIB) serialize ciphertext (microseconds)	22097	92119	406532	917803	20185	78755	345678	683089	20724	79458	321194	686632
	Average Compressed (Zstandard) serialize ciphertext (microseconds)	4742	4065	17693	93399	4323	4021	14798	55166	5125	3510	13249	144175

Using the default (4096) and custom degree (32768) configurations for the BFV, CKKS, and BGV homomorphic encryption schemes, let's compare (Table 2) the some of the performance metrics.

Table 2. Comparison of performance metrics for default and custom degree polynomial.

Performance metrics	**Observation**	
	Default Degree	**Custom Degree**
1. **Average Encrypt Time:**	Out of the three schemes, BGV is relatively fast in encryption when using the default degree configuration. BFV has the maximum encryption time, followed by BGV, which in turn follows CKKS .	For the custom degree configuration, BGV has the fastest encryption out of the three techniques. BFV has the longest encryption time, followed by CKKS, which has a somewhat longer encryption time than BGV.
2. **Average Decrypt Time:**	When it comes to decryption time, CKKS performs noticeably better than BFV and BGV, having a much lower value. In this comparison, BGV decrypts faster than BFV.	With a much lower value for decryption time, CKKS performs noticeably better in custom degree of polynomial than both BFV and BGV. BGV does not require as much time to decrypt as BFV.
3. **Degree Configurations:**	In this comparison, CKKS, which is intended for approximate arithmetic on real numbers, performs exceptionally well in terms of both encryption and decryption times. BGV and BFV have different performance characteristics; whereas BFV has a lower decryption time, BGV has a lower encryption time. Both algorithms are intended for integer arithmetic.	With a much faster decryption time than BFV and BGV, CKKS retains its efficiency in the custom degree configuration for both encryption and decryption times. In the custom degree, BGV has a shorter encryption time, but CKKS outperforms it in the decryption time. In the custom degree, BFV has the longest encryption time and the longest decryption time.
4. **Overall Performance:**	In situations where encryption and decryption performance are critical, such as while working with real-number data, CKKS might be chosen. If a shorter encryption time is desired, BGV may be taken into consideration. When a quicker decryption time is more important, BFV might be appropriate.	CKKS remains the best option in situations where performance in both encryption and decryption is essential, particularly when dealing with real-number data. BGV may be taken into account, even in custom degree configurations, if a shorter encryption time is desired. Although BFV has a longer encryption time, it might be appropriate in situations where a shorter decryption time is more important.

When comparing the average encryption times for all three schemes at the default degree of 4096, we can see that BGV takes less time to encrypt data. Depending on the processor's speed and system configuration, these values might change. These results are currently being produced by a 64-bit Windows 10 Pro operating system running on an Intel(R) Core(TM) i3-2370M CPU at 2.40GHz with 8 GB of RAM. Compared to BGV and BFV schemes, the CKKS scheme decrypts 4096 degree polynomials much faster. By observing results from table 1, when comparing all three schemes' degrees, we find that the BGV scheme performs addition operations faster on average, while the CKKS scheme performs multiplication operations faster.

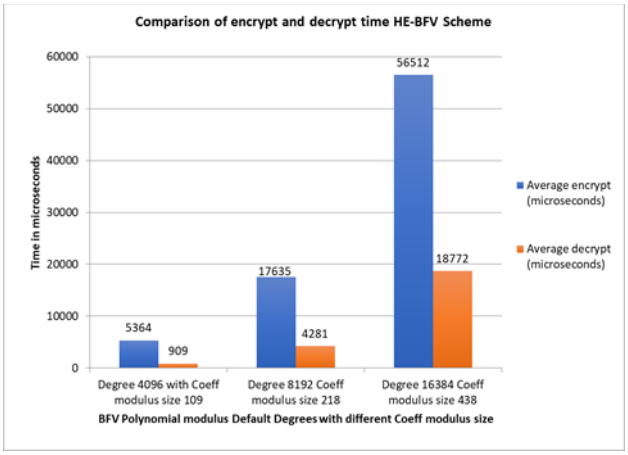

Fig 3. Comparison of encrypt and decrypt time for HE-BFV scheme

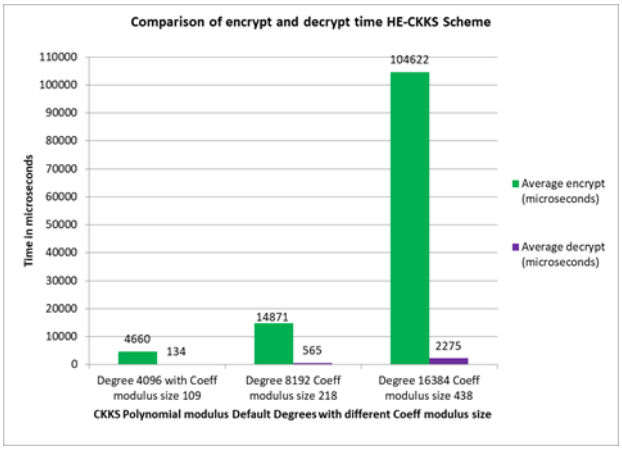

Fig 4. Comparison of encrypt and decrypt time for HE-CKKS scheme

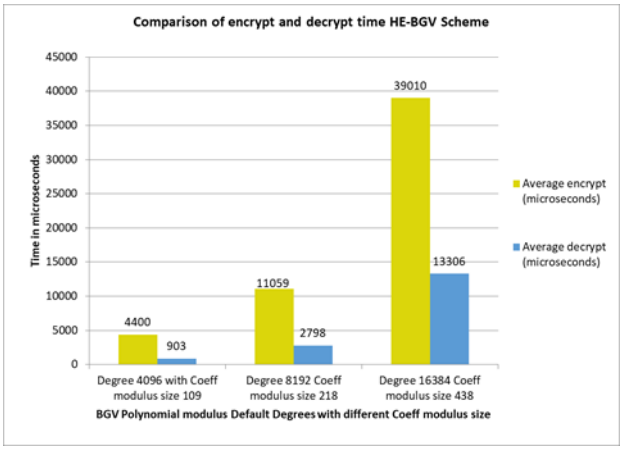

Fig 5. Comparison of encrypt and decrypt time for HE-BFV scheme

The fact that all three configurations of homomorphic encryption scheme with default degree 4096, 8192, and 16384, share the same degree of polynomial modulus is one of the main points of comparison, suggesting consistency in this parameter. Higher Coeff Modulus sizes typically result in higher security requirements, but they also increase computational demands. The degree of the polynomial modulus and the size of the coefficient modulus both contribute to an increase in encryption time. In BFV scheme, encryption times are longer for larger degrees and coefficient modulus sizes. With increasing degrees and coefficient modulus sizes, decryption times also rise. Although the decryption time is typically less than the corresponding encryption time, figure 3 illustrates the same trend of increasing time with higher parameters. In summary, the comparison demonstrates a trade-off between longer computation times (both for encryption and decryption, linked to higher polynomial modulus degree and coefficient modulus size) and increased security (achieved by higher coefficient modulus size). The particular configuration selected is determined by the security and performance requirements of the application.

The encryption times for the given CKKS (Cheon-Kim-Kim-Song) and BGV (Brakerski-Gentry-Vaikuntanathan) configurations are shown in Figures 4 and 5, respectively. The comparative description shows that, like BFV, both encryption and decryption times increase with increasing degree and coefficient modulus size in the CKKS and BGV encryption schemes. CKKS is well-known for being appropriate in situations involving continuous data and floating-point numbers. The notable rise in computation time as the parameters are increased demonstrates the trade-offs between security and performance. Arithmetic and polynomial evaluation over rational and integer numbers is supported by SEAL [18].

6. Conclusion

The FHE schemes CKKS and BFV are explored here through SEAL library. Due to its integrated fractional encoder, SEAL is recommended when dealing with fractional numbers as inputs [17]. If we consider all the parameters from the table 1, we can say that not a single scheme can be considered as optimal one from BGV, CKKS and BFV. As per the requirements of application, one can choose the desired scheme for securing the application based on computational performance of encryption, decryption, and additive, multiplicative homomorphic operations [19] because as per the degree and depth of computation the results can vary. BFV, CKKS, and BGV with default and custom degree configurations can be chosen based on the desired trade-offs between security and performance, the type of data, and the significance of encryption compared to decryption speed, among other application-specific considerations.

Conflicts of interest

The authors declare no conflicts of interest.

References

[1] Gentry, Craig. 2009. “A Fully Homomorphic Encryption Scheme.” Dissertation, no. September: 169. http://cs.au.dk/~stm/local-cache/gentry-thesis.pdf.

[2] William, Stallings, and William Stallings. Cryptography and Network Security, 4/E. Pearson Education India, 2006.

[3] Zvika Brakerski and Vinod Vaikuntanathan, Effcient Fully Homomorphic Encryption from (Standard) LWE, IeeeXplore-2011 BrakerskiV-FOCS 2011.

[4] Rivest, Ronald L., Len Adleman, and Michael L. Dertouzos. "On data banks and privacy homomorphisms." Foundations of secure computation 4, no. 11 (1978): 169-180.

[5] Sathya, Sai Sri, Praneeth Vepakomma, Ramesh Raskar, Ranjan Ramachandra, and Santanu Bhattacharya. 2018. “A Review of Homomorphic Encryption Libraries for Secure Computation.” ArXiv, 1–12.

[6] C. Gentry. Fully homomorphic encryption using ideal lattices. In M. Mitzenmacher, editor, Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009, pages 169–178. ACM, 2009.

[7] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings, pages 24–43, 2010.

[8] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. SIAM J. Comput., 43(2):831–871, 2014.

[9] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. Fully homomorphic encryption without bootstrapping. In Innovations in Theoretical Computer Science (ITCS’12), 2012. Available at http://eprint.iacr.org/2011/277.

[10] C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In R. Canetti and J. A. Garay, editors, Advances in Cryptology - CRYPTO 2013, Part I, pages 75–92. Springer, 2013.

[11] Halevi, S. (2017). Homomorphic Encryption. In: Lindell, Y. (eds) Tutorials on the Foundations of Cryptography. Information Security and Cryptography. Springer, Cham. https://doi.org/10.1007/978-3-319-57048-8_5

[12] Thi Van Thao Doan, Mohamed-Lamine Messai, Gérald Gavin et al. A Survey on Implementations of Homomorphic Encryption Schemes, 06 September 2022, PREPRINT (Version 1) available at Research Square [https://doi.org/10.21203/rs.3.rs-2018739/v1]

[13] V.Parmar, Payal, Shraddha B. Padhar, Shafika N. Patel, Niyatee I. Bhatt, and Rutvij H. Jhaveri. 2014. “Survey of Various Homomorphic Encryption Algorithms and Schemes.” International Journal of Computer Applications 91 (8): 26–32. https://doi.org/10.5120/15902-5081.

[14] Prasitsupparote, A. (2018). Implementation and Analysis of Fully Homomorphic Encryption in Resource-Constrained Devices. International Journal of Digital Information and Wireless Communications, 8(4), 288–303. https://doi.org/10.17781/p002535

[15] Acar, Abbas, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. 2018. “A Survey on Homomorphic Encryption Schemes.” ACM Computing Surveys 51 (4): 1–35. https://doi.org/10.1145/3214303.

[16] Bel Korchi, Amina, and Nadia El Mrabet. 2019. “A Practical Use Case of Homomorphic Encryption.” Proceedings - 2019 International Conference on Cyberworlds, CW 2019, 328–35. https://doi.org/10.1109/CW.2019.00060.

[17] Viand, Alexander, and Hossein Shafagh. 2018. “Marble: Making Fully Homomorphic Encryption Accessible to All.” Proceedings of the ACM Conference on Computer and Communications Security, 49–60. https://doi.org/10.1145/3267973.3267978.

[18] Alabdulatif, Abdulatif, Ibrahim Khalil, Heshan Kumarage, Albert Y. Zomaya, and Xun Yi. 2019. “Privacy-Preserving Anomaly Detection in the Cloud for Quality Assured Decision-Making in Smart Cities.” Journal of Parallel and Distributed Computing 127: 209–23. https://doi.org/10.1016/j.jpdc.2017.12.011.

[19] Aguilar Melchor, Carlos, Marc Olivier Kilijian, Cédric Lefebvre, and Thomas Ricosset. 2019. “A Comparison of the Homomorphic Encryption Libraries HElib, SEAL and FV-NFLlib.” Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 11359 LNCS: 425–42. https://doi.org/10.1007/978-3-030-12942-2_32.

[20] Natarajan, Deepika & Dai, Wei. (2021). SEAL-Embedded: A Homomorphic Encryption Library for the Internet of Things. IACR Transactions on Cryptographic Hardware and Embedded Systems. 2021. 756-779. 10.46586/tches.v2021.i3.756-779.