Noname manuscript No.
(will be inserted by the editor)

A Survey on Implementations of Homomorphic Encryption Schemes

Doan Thi Van Thao* · Mohamed-Lamine Messai · Gérald Gavin · Jérôme Darmont

Received: date / Accepted: date

Abstract With the increased need for data confidentiality in various applications of our daily life, homomorphic encryption (HE) has emerged as a promising cryptographic topic. HE enables to perform computations directly on encrypted data (ciphertexts) without decryption in advance. Since the results of calculations remain encrypted and can only be decrypted by the data owner, confidentiality is guaranteed and any third party can operate on ciphertexts without access to decrypted data (plaintexts). Applying a homomorphic cryptosystem in a real-world application depends on its resource efficiency. Several works compared different HE schemes and gave the stakes of this research field. However, the existing works either do not deal with recently proposed HE schemes (such as CKKS) or focus only on one type of HE. In this paper, we conduct an extensive comparison and evaluation of homomorphic cryptosystems' performance based on their experimental results. The study covers all three families of HE, including several notable schemes such as BFV, BGV, FHEW, TFHE, CKKS, RSA, El-Gamal, and Paillier, as well as their implementation specification in widely used HE libraries, namely Microsoft SEAL, PALISADE, and HElib. In addition, we also discuss the resilience of HE schemes to differ-

* Corresponding author

Thi Van Thao Doan
Université catholique de Louvain; Univ Lyon, Univ Lyon 2, UR ERIC
E-mail: doanthivanthao.94@gmail.com

Mohamed-Lamine Messai
Univ Lyon, Univ Lyon 2, UR ERIC
E-mail: mohamed-lamine.messai@univ-lyon2.fr

Gérald Gavin
Univ Lyon, Univ Lyon 1, UR ERIC
E-mail: gerald.gavin@univ-lyon1.fr

Jérôme Darmont
Univ Lyon, Univ Lyon 2, UR ERIC
E-mail: jerome.darmont@univ-lyon2.fr

ent kind of attacks such as Indistinguishability under chosen-plaintext attack and integer factorization attacks on classical and quantum computers.

Keywords Cryptography · Homomorphic encryption · Performance evaluation · Information Security · Privacy

1 Introduction

For a long time, information security has always been a controversial topic due to its importance in technology particularly and in society generally. When implementing a technological tool or service, the first and foremost concern of researchers is about the applicable security that it can provide.

By the unavoidable data growth in nearly all organizations, the demand for data storage and computation has been increasing significantly over the past few decades. A traditional infrastructure for data management, such as in-house or local services, can only offer a limited storage and access controls. In the Internet-based world, this method is no longer applicable since a huge amount of sensitive data is produced every second from business transactions. One potential solution for this problem is to seek a third-party expert, i.e., cloud computing providers, outside of the company to place its trust. However, to put this paradigm into practice, we need to deal with one of its biggest challenges: data confidentiality.

In this situation, cryptography has come to the forefront to provide both data confidentiality and data operations for this outsourcing problem. As the foundation of modern security systems, cryptography helps to ease the concern of data leakage to an untrusted third party or server side. Data must now be encrypted by the user before being sent to the server. Later, after retrieving the encrypted result from the server, only the user can decrypt it using his secret key and get its value. Although this technique would preserve the data privacy, the encrypted data is not meaningful for the server, so it is not able to maintain its computation efficiency. That was why for that moment, a new cryptographic topic, called *Homomorphic Encryption*, got a major attention when it allows to perform certain computable functions on the encrypted data while keeping the characteristics of the function and format of the ciphertexts. In [1], A. Acar et al. present this process on Figure 1, where C is a client and S is a server. Following this survey, in terms of the number of allowed operations on encrypted data, HE can be classified into three types: (1) *Partially Homomorphic Encryption* (PHE) allows only one type of operation to be performed an unlimited number of times. (2) *Somewhat Homomorphic Encryption* (SWHE) allows some types of operations with a limited number of times. (3) *Fully Homomorphic Encryption* (FHE) allows an unlimited number of operations for an unlimited number of times. Figure 2 presents the most known HE-based systems and their timeline, while their application scenarios are demonstrated in Table 1.

So far, there are many HE schemes that have been introduced. Within the scope of the paper, we concentrate on the ones which are the most widely used

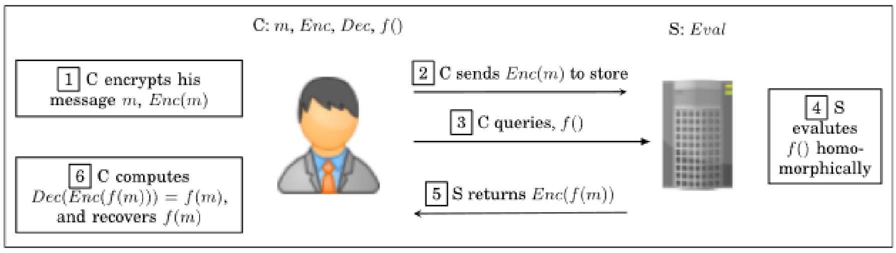
C: m, Enc, Dec, $f()$
S: $Eval$
1 C encrypts his message m, $Enc(m)$
2 C sends $Enc(m)$ to store
3 C queries, $f()$
4 S evalutes $f()$ homo-morphically
5 S returns $Enc(f(m))$
6 C computes $Dec(Enc(f(m))) = f(m)$, and recovers $f(m)$

Fig. 1: A simple client-server HE scenario [1]

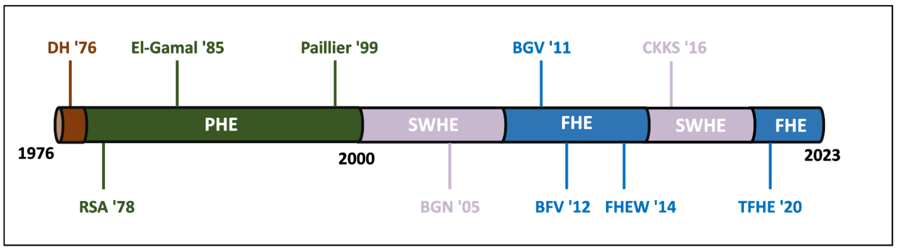

Fig. 2: Timeline of several important HE schemes

in cryptography applications and serve as the basis for other schemes. Being proposed in 1978, RSA is one of the first public-key encryption methods for securing communication on the Internet, inspired by Diffie-Hellman's research ([2], 1976). Little while later, El-Gamal ([3], 1985) and Paillier cryptosystems ([4], 1999) were introduced respectively, marking an important milestone for PHE. The calculation on ciphertexts remained limited until C. Gentry presented the first FHE scheme ([5], 2009), using *bootstrapping* technique. Technically, the bootstrapping method is an intermediate procedure to refresh a ciphertext with large error to be a new one with smaller error, so that it allows more computations. This is also how the SWHE scheme is converted into a fully homomorphic one. Three years later, based on the Gentry's work, two main FHE schemes to perform exact computations over finite fields and integers were born, Brakerski-Gentry-Vaikuntanathan (BGV) [6] and Brakerski/ Fan-Vercauteren (BFV) [7]. The latest newcomer to join SWHE is CKKS ([8], 2016), which allows to perform computations over approximated numbers. CKKS is an essential element of the HE family, where it complements previous schemes by natively dealing with real and complex numbers.

Although FHE schemes are efficient and secure, they are not very practical in real-life contexts since Gentry's bootstrapping procedure requires heavy computations to refresh noisy ciphertexts and keep computing on encrypted data. In 2014, Ducas and Micciancio [9] introduced a new method to homomorphically compute simple bit operations, and refresh (bootstrap) the resulting output, called "Fastest Homomorphic Encryption in the West" (FHEW). According to the authors, FHEW can improve the time required to bootstrap

Scheme	Application
RSA, 1978 [15]	Banking and credit card transaction (Parmar et al., [16])
ElGamal, 1985 [3]	In Hybrid Systems (Parmar et al., [16])
Paillier, 1999 [4]	E-Voting (Parmar et al., [16])
BGN, 2005 [17]	A Novel IoT Data Protection Scheme Based on BGN Cryptosystem (S. Halder et al., [18])
BGV, 2011 [6]	For the Security of Integer Polynomials (Parmar et al., [16])
BFV, 2012 [7]	A fast oblivious linear evaluation (OLE) protocol (Leo de Castro [19])
CKKS, 2016 [8]	Homomorphic Machine Learning Big Data Pipeline for the Financial Services Sector (Masters et al., [20])
FHEW, 2014 [9]	TFHE: Fast Fully Homomorphic Encryption over the Torus (Chillotti et al., [10])
TFHE, 2020 [10]	An homomorphic LWE based E-voting Scheme (Chillotti et al., [21])

Table 1: HE schemes and their applications

the ciphertext, which is homomorphic evaluation of a NAND gate "in less than a second". In 2020, an improvement of FHEW was initially proposed by I. Chillotti et al. [10], which described a fast fully homomorphic encryption scheme over the torus (TFHE) and revisited FHE based on GSW ([11], 2013). Following Ducas et al. [12], the most important difference between TFHE and FHEW is that TFHE uses (an optimized version of) the FHEW accumulators to implement a ring variant of the bootstrapping procedure described in [13], rather than [14]. Thanks this difference and other optimizations, the bootstrapping runtime is reduced to less than 0.1 second. All of these improvements marked a milestone in the FHE implementation, as well as contributed greatly to bridging the gap between FHE security and its efficiency in practice.

The rest of the paper is structured as follows: section 2 reviews some of the related works in the similar field, regarding the performance evaluation of different HE schemes. The most important properties of HE schemes and their libraries are discussed in section 3. Then, section 4 and section 5 respectively elaborate the implementation method and results of evaluation analysis. In section 6, a discussion on the security of HE under notable security notions and Shor's quantum algorithm is given. Finally, section 7 presents conclusions and indicates directions of future work.

2 Related work

As previously mentioned, one of the related works is a survey conducted by A. Acar et al. in 2018 [1] that covers important PHE, SWHE, and all the major FHE schemes. Similarly, the survey of P. Martins et al. (2017) [22] presents fundamental concepts of FHE schemes and their performance, mainly from an engineering perspective, refraining from introducing complex mathematical definitions. These works, however, do not mention CKKS encryption [8], an usefully practical HE proposed recently in 2016 for computing real and complex input numbers. Lately, a study of Kim et al. [23], published in 2021, implements their improved variants of BFV and BGV in PALISADE and evaluates their experimental performance for several benchmark computations. From a same point of view, Lepoint and Naehrig in [24] offer theoretical and practical comparisons of different HE schemes, as well as explain how to choose parameters to ensure algorithms' correctness and security. Even so, the papers delve deeply into the mathematics, making them more suitable for expert readers and mathematicians. In contrast, the survey conducted by Alaya et al. [25] makes a easy-to-understand comparison of advantages and limitations of different HE algorithms. Unfortunately, it only presents the theoretical information of the schemes, while implementation aspects have not been brought up. Most recently, Sidorov et al. [26] published a paper on performance analysis of HE in several libraries, but the paper does not specify which homomorphic schemes were used in each library, either the input parameters. In opposition to [26], (Migliore et al. [27]) proposes a study of the current best solutions for setting up parameters of HE schemes, but only approaches of SWHE schemes.

Considering the related works in the field and their scopes summarized in Table 2, it is obvious that among existing HE surveys, they either do not study newborn schemes (such as CKKS, FHEW, TFHE) or do not cover all three HE families. As a result, there is still a need in this field for a comprehensively up-to-date survey that provides key concepts of the main encryption schemes in all three HE categories, together with their experimental performance comparisons. The survey needs to be practical and show newly interested users how to build their own HE-based projects in popular HE libraries.

Our contribution: Our work aims to provide readers with fundamental principles of HE schemes without delving too deep into the mathematics. Furthermore, the paper conducts a comprehensively theoretical and practical comparison of important HE schemes, covering all three HE categories: FHE, SWHE, and PHE. For different HE schemes in each family, we analyse their input parameters, together with their constraints, and then compare them together. This hands-on experience helps unprofessional practitioners distinguish libraries' properties and makes them easy to apply in building their own HE-based projects. In addition, we provide experimental results on performance evaluation of each HE scheme in most-used libraries such as SEAL [28], PALISADE [29], HELib [30], and HEAAN [31]. Although PHE schemes are now not available in mentioned open-source libraries, our own implementations of Paillier, El-Gamal, and RSA are used as partially homomorphic cryptosystems in the experimental study. For each execution case, we also come up with assessments and results' explanations. Furthermore, in the last part, we deliver a concrete discussion on the security of aforementioned schemes against IND-CPA, IND-CCA, as well as integer factorization attacks on classical and quantum computers.

Authors	Description	Scope	
		Schemes	Libraries
C. Fontaine et al., 2007 [32]	Providing nonspecialists with a survey of HE techniques	-	-
T. Lepoint et al., 2014 [24]	Conducting a comparison of FV and YASHE schemes and explaining how to choose parameters to ensure correctness and security against lattice attacks	BGV YASHE	FLINT
V. Migliore et al., 2016 [27]	Proposing a study of the current best solutions, providing a deep analysis of how to setup and size their parameters	BFV SHIELD	-
P. Martins et al., 2017 [22]	Studying SWHE and FHE schemes supported by their performance and security from an engineering standpoint	BGV BFV Paillier El-Gamal	-
A. Acar et al., 2018 [1]	Providing a comprehensive survey of the main FHE, PHE and SWHE schemes, including the FHE implementations	RSA Paillier El-Gamal	SEAL HELib
B. Alaya et al., 2020 [25]	Presenting different HE cryptosystems, joined with a final comparison between the adopted techniques	-	-
A. Kim et al., 2021 [23]	Revisiting BGV and BFV, together with proposing an improved variant of BFV	BGV BFV	PALISADE
C. Zaraket et al., 2021 [33]	Proposing SAVHO homomorphic scheme and its performance analysis in comparison with Pailler cryptosystem	Paillier SAVHO	SageMath
V. Sidorov et al., 2022 [26]	Conducting an extensive study of homomorphic cryptosystems' performance for practical data processing	Paillier El-Gamal	SEAL PyAono HELib
S. J. Mohamme et al., 2022 [34]	Evaluating performance of RSA, ElGamal, and Paillier homomorphic encryption algorithms	RSA El-Gamal Paillier	-
D. Micciancio et al., 2022 [12]	Presenting a unified framework that includes the original and extended variants of both FHEW and TFHE cryptosystems	FHEW TFHE	PALISADE

Table 2: Related works and their scopes

3 Background

3.1 Libraries

HElib (**H**omomorphic-**E**ncryption **Lib**rary) [30] is the first open source library implementing HE. Being published in 2013, it focuses on effective use of BGV and CKKS schemes, together with ciphertext packing techniques and the Gentry-Halevi-Smart optimizations. HElib is still under development by Shai Halevi (IBM), Victor Shoup (NYU, IBM) and available on `github` [35]. In 2018, the authors implemented several algorithmic improvements, including Faster Homomorphic Linear Transformations [36], that made HElib 30–75 times faster than those previously built for typical parameters.

PALISADE [29] is an open-source C++ project that provides efficient implementations of lattice cryptography building blocks. The library supports varied HE schemes, such as: BGV, BFV, CKKS, FHEW, and TFHE. In addition, it also supports multi-party extensions of certain schemes and related cryptography primitives, namely digital signature schemes, proxy re-encryption, and program obfuscation. PALISADE can be found on `github` [37]. According to the newest PALISADE's announcement, the PALISADE community has merged the PALISADE project into the next-gen OpenFHE open-source FHE software library. OpenFHE ([38], 2022) has all of the features of PALISADE, merged with selected capabilities of HElib and HEAAN.

SEAL (**S**imple **E**ncrypted **A**rithmetic **L**ibrary) [28] is another HE library, developed by the Cryptography and Privacy Research Group at Microsoft. According to his author, Kim Laine, the first version of SEAL was released in 2015 with the specific goal of providing a well-engineered and documented HE library. SEAL was designed to use both by experts and by non-experts with little or no cryptographic background. The updated version of Microsoft SEAL, which is available on `github` [39], has implemented various forms of HE schemes, including BGV, BFV, and CKKS. Besides, there is a SEAL version in Python, called SEAL-Python [40]. This is a Python wrapper implementation of the SEAL library, using `pybind11` [41].

HE scheme/Library	**SEAL**	**PALISADE**	**HElib**	**HEAAN**
BFV	✓	✓		
BGV	✓	✓	✓	
CKKS	✓	✓	✓	✓
FHEW		✓		
TFHE		✓		

Table 3: HE schemes in available HE libraries

To have an extensive comparison for CKKS encryption, apart from these three mentioned libraries, we also measure its running time in HEAAN library [31], developed in 2016 by its own authors. **HEAAN** (**H**omomorphic **E**ncryption for **A**rithmetic of **A**pproximate **N**umbers) is an open-source cross platform software library which implements the approximate HE scheme proposed by Cheon, Kim, Kim and Song (CKKS). HEAAN executes only CKKS schemes with its complete properties. Following its owners, the library allows additions and multiplications to be performed by fixed point arithmetics and approximate operations between rational numbers. Table 3 illustrates the distribution of several encryption schemes in each library.

To study the advantages and drawbacks of aforementioned libraries, we define a set of criteria. The first criterion is whether the library is open source. It is important for the transparency reason and could be a drawback for not open-sourced libraries. Ease of use criterion of the library means that it is easy to integrate with existing systems and have clear documentation and examples. The library should also have a well-designed API and be easy to use for developers. The compatibility criterion explains the dependence of the library for a specific platform and/or hardware. For example, if the system is based on a particular operating system or hardware platform, the library should be compatible with that platform. The reliability criterion indicates that the library implementation is stable with minimal bugs. Based on these criteria, Table 4 presents the comparison study while the more + sign means that the library meets more of the criterion.

Library/Criteria	**Open source**	**Ease of use**	**Compatibility**	**Reliability**
SEAL	✓	++	+++	++
PALISADE	✓	+	++	+
HElib	✓	++	+++	++
HEAAN	✓	+	+	+

Table 4: HE libraries comparison

All four libraries are open source and freely available under permissive licenses. This means that they can be used, modified, and distributed by developers without restriction. For ease of use purpose, the four libraries offer abstractions of the details of the HE schemes. HElib is known for its ease of use because it is currently well documented. HElib and PALISADE are primarily designed for x86-based CPUs compatibility, while SEAL and HEAAN can be used on a wider range of platforms, including ARM-based CPUs. For reliability, the four discussed libraries are reliable and well-tested. However, HElib and SEAL are more mature and have been used in production systems for several years, while PALISADE and HEAAN are newer and still be undergoing active development. For the compatibility feature, Microsoft SEAL has been built on various platforms (Windows, Linux, macOS/iOS, Android, and FreeBSD), while HEAAN is checked working well on Ubuntu. In addition, PALISADE and HElib are adaptable with Linux, MacOS, and Windows. The previous versions of Helib have also included Fedora, CentOS, and macOS Mojave. Although all four libraries are compatible with various operating systems, they differ in their performance. The performance of implemented HE schemes in these libraries will be discussed in Section 5.

3.2 Homomorphic encryption schemes

In this part, we explain basic properties of HE, followed by a brief description of some notable PHE, SWHE, and FHE schemes. An HE scheme is based on five main homomorphic operations: Key generation (`KeyGen`), Encryption (`Enc`), Decryption (`Dec`), Homomorphic addition (`Add`), and Homomorphic multiplication (`Mult`).

The performance evaluation of mentioned schemes will be detailed in section 5. But first of all, we define a homomorphic encryption. As per [1], an encryption scheme is called homomorphic over an operation "$\star$" (e.g., `Add`, `Mult`) if it supports the following equation:

$$E(m_1) \star E(m_2) = E(m_1 \star m_2), \forall m_1, m_2 \in M,$$

where E is the encryption algorithm and M is the set of all possible messages.

3.2.1 RSA

This HE was first introduced by Rivest et al. [15]. The security of the cryptosystem relies on the practical hardness of factoring the product of two large prime numbers [42], called the *factoring problem.* Given a security parameter λ, RSA is defined as follows:

- `KeyGen`(λ): First, two large prime numbers (p and q) are randomly chosen, then $N = pq$ and $\phi(N) = (p-1)(q-1)$ are computed. The secret large integer d is picked such that $\gcd(d, \phi(N)) = 1$. The last public component e is calculated by computing the multiplicative inverse of d (i.e., $ed \equiv 1 \mod \phi(N)$). Finally, set the public key $pk = (e, N)$, and the secret key $sk = (d, p, q)$.
- `Enc`$(pk, m \in \mathbb{Z}_N)$: The message m is an integer between 0 and $N-1$. The encryption of m is c, such that: $c = E(m) = m^e \pmod N$.
- `Dec`(sk, c): The message m can be recovered from the ciphertext c by: $m = c^d \pmod N$.
- `Mult`(c_1, c_2): $c_1c_2 = E(m_1)E(m_2) = [m_1^e \pmod N][m_2^e \pmod N]$
  $= (m_1m_2)^e \pmod N = E(m_1m_2)$.

3.2.2 El-Gamal

The encryption system is a widely-used HE in public-key cryptography, proposed by T. ElGamal in 1985 [3]. The advent of El-Gamal algorithm is based on the Diffie–Hellman key exchange, while its security strength is relied on the hardness of solving discrete logarithms.

- `KeyGen`(λ): Firstly a cyclic group G of order N and its generator $g \in \mathbb{Z}_N^*$ are generated. After randomly drawing an integer x from $\{1, \ldots, N-1\}$, $h = g^x$ is computed. The public key pk consists of (G, N, g, h), while $sk = x$ is kept secret.
- `Enc`$(pk, m \in \mathbb{Z}_N)$: A message m is encrypted by choosing an integer y randomly from $\{1, \ldots, N-1\}$, then computing $s = h^y$. the output of the encryption is a ciphertext $c = (c_1, c_2)$, where $c_1 = g^y$ and $c_2 = ms$.
- `Dec`(sk, c): To decrypt the ciphertext, firstly $s' = c_1^x$ needs to be calculated. Next, m is recovered by $m = c_2 s'^{-1}$.
- `Mult`(c_1, c_2):
  $c_1c_2 = E(m_1)E(m_2) = (g^{x_1}, m_1h^{x_1}) \cdot (g^{x_2}, m_2h^{x_2}) = (g^{x_1+x_2}, m_1m_2h^{x_1+x_2}) = E(m_1m_2)$.

3.2.3 Paillier

The encryption of Paillier (1999) [4] is an additively homomorphic cryptosystem, which is based on the composite residuosity problem and gathers many good properties.

- `KeyGen`(λ): Two primes numbers p, q of k bits are randomly generated such that $N = pq$ and $\rho = N^{-1} \pmod{\phi(N)}$, where $\phi(N) = (p-1)(q-1)$. One can publish $pk = N$ and $sk = \rho$.
- `Enc`$(pk, m \in \mathbb{Z}_N)$: To encrypt a message m, first an integer r from $\{1, \dots, N-1\}$ is chosen randomly. The output is the ciphertext $c = (1+mN)r^N \pmod{N^2}$.
- `Dec`(sk, c): To recover the message m, $r = c^\rho \pmod N$ is computed. Then $m = \frac{(cr^{-N} \pmod{N^2})-1}{N}$.
- `Add`(c_1, c_2):
  $c_1c_2 = E(m_1)E(m_2) = (1+m_1N)r_1^N(1+m_2N)r_2^N \pmod{N^2} = [1+(m_1+m_2)N + m_1m_2N^2](r_1r_2)^N \pmod{N^2} = [1+(m_1+m_2)N]r^N \pmod{N^2} = E(m_1+m_2)$.

3.2.4 BFV

In 2012, J. Fan and F. Vercauteren [7] modified the scheme proposed by Brakerski [43] from the learning-with-errors (LWE) setting to the Ring-LWE setting. By using a simple modulus switching trick, BFV (so-called FV) provides a more efficient approach and also simplifies the analysis of the bootstrapping step. The security of BFV-type cryptosystems is based on the LWE over rings (or RLWE) assumption [44]. The RLWE(λ, q, χ) assumption states that it is very hard to distinguish two distributions $(a, b = a \cdot s + e)$ and (a, u), where a, s, and u are randomly selected from R_q and e is selected from an error distribution χ, referencing security parameter λ. This assumption has been proved hard over ideal lattices [45].

Let $R = \mathbb{Z}[x]/f(x)$ be a ring of polynomials in which the operations of BFV will be performed, where $f(x) = x^N + 1$ is a cyclotomic polynomial with N being a power of 2. The ring is used to define the RLWE problem with coefficients in $\mathbb{Z}_q$, denoted by $R_q = \mathbb{Z}_q[x]/f(x)$. Additionally, the message space is defined as R_t for an integer $t > 1$.

- `KeyGen`(λ): For a B-bounded distribution χ over the ring R, a vector of secret key $sk = s$ is sampled $s \leftarrow \chi$. The public key is defined by: $pk = ([-(a \cdot s + e)]_q, a)$, where $e \leftarrow \chi$ and $a \leftarrow R_q$.
- `Enc`$(pk, m \in R_t)$: Given a plain message m, let $p_0 = pk[0]$, $p_1 = pk[1]$, and draw $u, e_1, e_2 \leftarrow \chi$, the ciphertext is: $c = ([p_0 \cdot u + e_1 + \Delta \cdot m]_q, [p_1 \cdot u + e_2])$, where $\Delta = \lfloor q/t \rfloor$.
- `Dec`(sk, c): Let $c = (c_0, c_1)$ be an encrypted message. The decryption returns m such as $m = \left[\left\lfloor \frac{t}{q}[c_0 + c_1 \cdot s]_q \right\rceil\right]_t$.
- `Add`(c_1, c_2): Let c_1, c_2 be two encrypted messages such that $c_1 = (c_{10}, c_{11})$ and $c_2 = (c_{20}, c_{21})$. The addition of two digits is $c = ([c_{10} + c_{20}]_q, [c_{11} + c_{21}]_q)$.

– `Mult`(c_1, c_2): By multplying two ciphertexts $c_1(s)$ and $c_2(s)$, the result is $c_1(s) \cdot c_2(s) = c'_0 + c'_1 \cdot s + c'_2 \cdot s^2$. One encountered problem is that resulting ciphertext has size 3 (degree 2) and must be reduced to a size 2 (degree 1) [7]. This process is called *relinearization.* To start, a relinearization key rlk is generated by choosing an integer p and sampling a new $a \leftarrow R_{pq}$ and $e \leftarrow \chi'(\chi' \neq \chi)$ satisfying $rlk = ([-(a \cdot s + e) + p \cdot s^2]_{pq}, a)$. The output is relinearizated to 1-degree ciphertext: $([c_0 + c_{2,0}]_q, [c_1 + c_{2,1}]_q)$, where:

$$c_0 = \left[\left\lfloor \frac{t \cdot (c_{10} \cdot c_{20})}{q} \right\rceil\right]_q$$

$$c_1 = \left[\left\lfloor \frac{t \cdot (c_{10} \cdot c_{21} + c_{11} \cdot c_{20})}{q} \right\rceil\right]_q$$

$$c_2 = \left[\left\lfloor \frac{t \cdot (c_{11} \cdot c_{21})}{q} \right\rceil\right]_q$$

$$(c_{2,0}, c_{2,1}) = \left(\left[\left\lfloor \frac{c_2 \cdot rlk[0]}{p} \right\rceil\right]_q, \left[\left\lfloor \frac{c_2 \cdot rlk[1]}{p} \right\rceil\right]_q\right)$$

3.2.5 BGV

BGV encryption was invented in 2011 by Brakerski, Gentry, and Vaikuntanathan [6]. BGV is a levelled FHE that works for both an LWE and an RLWE. A levelled FHE means that the parameters of the scheme depend (polynomially) on the maximum number of multiplications that can be executed (called level L). The hardness of the scheme is also based on RLWE problem [45]. To keep the ciphertext error within a given bound, they used the technique of modulus switching as introduced in [46]. This modulo reduction maps a ciphertext c defined in a ring R_q, to a ring $R_{q'}$ while preserving correctness, where $q' < q$ [47]. By combining the modulus switching method with the bootstrapping procedure after performing desired operations on the ciphertext, BGV scheme can be turned into FHE [48].

In original BGV, public key and switch keys are matrices. A detailed explanation of the scheme can be found in [49]. Given a security parameter λ, level L, and plaintext modulus p. First step is to generate L large primes $q_0, \ldots q_{L-1}$ satisfying $q_0 < \cdots < q_{L-1}$.

– `KeyGen`(λ, χ, L): A vector s is selected randomly as a sk. Then $b = -(a \cdot s + p \cdot e) \pmod{q_{L-1}}$ is computed, where $a \leftarrow R_{q_{L-1}}$, $e \leftarrow \chi$. The public key is (a, b). Next, the switch keys $(a_0, b_0, t_0, 0), \ldots, (a_{L-1}, b_{L-1}, t_{L-1}, L-1)$ will be computed, where $b_i = -(a_i \cdot s + p \cdot e_i - t_i \cdot s^2) \pmod{t_i \cdot q_i}$, $a_i \leftarrow R_{q_i}$, $e_i \leftarrow \chi$, and t_i is an integer.
– `Enc`$(pk, m \in \mathbb{Z}_p)$: A plaintext m can be encrypted by $E(m) = (c_0, c_1) = ((b \cdot v + p \cdot e_0 + m) \pmod{q_{L-1}}, (a \cdot v + p \cdot e_1) \pmod{q_{L-1}})$, where each element of vector v, $v_i \in \{0, 1, -1\}$ and $e_0, e_1 \leftarrow \chi$. We have $c = (c_0, c_1, L-1)$ is the initial ciphertext.
– `Dec`(sk, c): A ciphertext $c = (c_0, c_1, i)$, $i = [0, L-1]$ can be decrypted to find its plaintext m by $m = c_0 + c_1 \cdot s \pmod{q_i} \pmod{p}$.

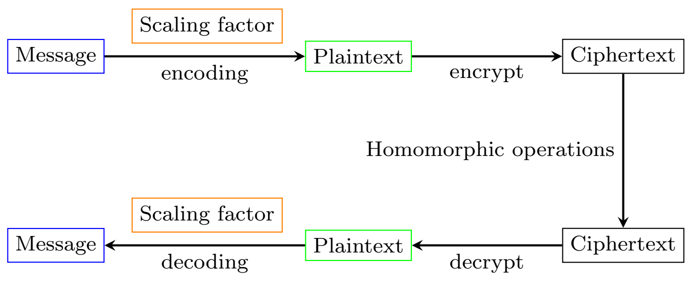

Fig. 3: Algorithms in CKKS [50]

- `Add`(c_1, c_2): The addition of two ciphertexts of a same level $c_1 = (c_{10}, c_{11}, i)$ and $c_2 = (c_{20}, c_{21}, i)$ is computed by $c = ((c_{10} + c_{20}) \pmod{q_i}, (c_{11} + c_{21}) \pmod{q_i})$.
- `Mult`(c_1, c_2): Similarly to BFV method, the first step is to compute a degree-2 ciphertext, where we denote $c_1(s)\cdot c_2(s) = c'_0 + c'_1 \cdot s + c'_2 \cdot s^2$. The relinearization procedure results a compressed ciphertext with degree 1: $c^* = (c^*_0, c^*_1)$, where $c^*_0 = t_i c'_0 + b_i \cdot c'_2 \pmod{t_i q_i}$ and $c^*_1 = t_i c'_1 + a_i \cdot c'_2 \pmod{t_i q_i}$, with the switch key (a_i, b_i, t_i, i). The ciphertext c^* will be mapped to $c \in R_{q_{i-1}}$ as the output by *SwitchModulus* method.
- `SwitchModulus`$(c = (c_0, c_1, i))$: Supposing to have two modulus q_i and q_j where $i > j$, and a ciphertext c in ring R_{q_i}, we calculate modulo inverse element $r_j = \frac{q_j}{q_i}$ in q_j. The new ciphertext in ring R_{q_j} is defined by $\bar{c} = (\bar{c}_0, \bar{c}_1, j) = (c_0 r_j \pmod{q_j}, c_1 r_j \pmod{q_j}, j)$.

3.2.6 CKKS

As mentioned in the previous section, CKKS, a HE for approximate arithmetic, was introduced in 2016 in [8]. What makes CKKS draw attention to many researchers is that it allows to perform approximate additions and multiplications of ciphertexts, where its plaintexts can be vectors of real and complex values. This has been done by *encoding* and *decoding* method, where the inputs are converted from $C^{N/2} \times \mathbb{R}$ to $R = \mathbb{Z}[x]/(x^N + 1)$ and vice versa [8]. In this step, we need to use a rounding technique, which might destroy some significant numbers. Thus, if we had an initial vector of real or complex values z, roughly speaking it will be multiplied by a scale $\Delta > 0$ during encoding and then divided by Δ during decoding to keep a precision of $\frac{1}{\Delta}$. Figure 3 describes all algorithms in CKKS scheme [50].

As well as many other HE schemes, the foundation of CKKS is also the RLWE problem. Similarly to previously presented schemes, in this part, we simply describe the five main algorithms of CKKS. To start, it begins with a integer $p > 0$, number of multiplication L, and modulus q_0. For $0 < l \leq L$, we define $q_l = p^l q_0$.

- `KeyGen`(λ, q_L): First, a vector s is sampled from a set of signed binary vectors in $\{0, 1, -1\}^N$ whose Hamming weight is exactly an integer h. Next,

$a \leftarrow R_{q_L}$, and $e \leftarrow \chi$. We set the secret key $sk = (1, s)$, $pk = (b, a) \in R_{q_L}^2$ with $b = -as + e \pmod{q_L}$. We choose an integer P, set $a' \leftarrow R_{P \cdot q_L}$, $e' \leftarrow \chi$, and $evk = (b', a') \in R_{P \cdot q_L}^2$ with $b' = -a's + e' + Ps^2 \pmod{P \cdot q_L}$.

- `Enc`(pk, m): Given a distribution $ZO(\rho)$ draws each entry in the vector from $\{0, 1, -1\}^N$, with probability $\rho/2$ for each of -1 and $+1$, and probability being zero $1 - \rho$. To encrypt a polynomial m, we sample polynomials $v \leftarrow ZO(0.5)$, $e_0, e_1 \leftarrow \chi$, then output the ciphertext $c = v \cdot pk + (m + e_0, e_1) \pmod{q_L}$.
- `Dec`(sk, c): For a ciphertext $c = (b, a) \in R_{q_l}^2$, the approximate result m' of the plaintext m can be recovered by $m' = m + e = b + a \cdot s \pmod{q_l}$.
- `Add`(c_1, c_2): For $c_1, c_2 \in R_{q_l}^2$, its addition is $c = c_1 + c_2 \pmod{q_l}$.
- `Mult`(evk, c_1, c_2): Similarly to introduced HE scheme, the multiplication of CKKS also accompanies a relinearlization step. For $c_1 = (b_1, a_1)$, $c_2 = (b_2, a_2) \in R_{q_l}^2$, let $(c_0', c_1', c_2') = (b_1 b_2, a_1 b_2 + a_2 b_1, a_1 a_2) \pmod{q_l}$. After being relinearlizated, it outputs a degree-1 ciphertext $c = (c_0', c_1') + \lfloor P^{-1} \cdot c_2' \cdot evk \rceil \pmod{q_l}$.
  One problem produced is that underlying value contained in the plaintext and ciphertext is $\Delta \cdot z$ as mentioned above. So after multiplying two ciphertexts c_1, c_2, the result holds $z_1 \cdot z_2 \cdot \Delta^2$. By doing many multiplications, the resulting ciphertext will have grown exponentially. To reduce its size, *Rescale* $RS_{l \to l'}$ is introduced with its goal being to actually keep the scale constant, and also reduce the noise present in the ciphertext.
- $RS_{l \to l'}(c)$: For a ciphertext $c \in R_{q_l}^2$ at level $l > l'$, we output $c' = \lfloor \frac{q_{l'}}{q_l} c \rceil \in \pmod{q_{l'}}$.

3.2.7 TFHE/FHEW

FHEW [9] and TFHE [10] have joined FHE family since 2014, where TFHE is an improvement of FHEW that significantly reduces the running time of the bootstrapping process. Like other FHE schemes, FHEW/TFHE's hardness is based on RLWE assumption. Using the similar mentioned notation, a ciphertext in FHEW/TFHE cryptosystem encrypting a message $m \in R_t$ under key $s \in R_q$ is $c = (a, b) = (a, a \cdot s + e + m)$, with $a \leftarrow R_q$ and $e \leftarrow \chi$ chosen from a discrete Gaussian distribution. The decryption is done by computing $b - a \cdot s = e + m$ and evaluating an appropriate decoding function to correct the error e and recover the message m [12]. One example of the decoding function is scaling m by a factor $\Delta = \lfloor q/t \rfloor$ as BFV scheme, which is described in 3.2.4. Being inherited the properties of RLWE, TFHE/FHEW is also homomorphically additive and multiplicative. The development that makes TFHE/FHEW a breakthrough in FHE timeline is bootstrapping technique. In FHEW setting, given an LWE ciphertext (a, b), an encryption $E(m)$ of the same message under a different encryption scheme E is computed by homomorphically evaluating the LWE decryption procedure on the encrypted key $E(m)$ to yield

$$\left\lfloor 2(b - a \cdot E(s))/q \right\rceil \mod 2 \simeq E(m).$$

In other words, the result of the computation is also an encryption $E(m)$ of the message, but with smaller noise. D. Micciancio et al. [12] showed that the noise of the output ciphertext $E(m)$ only depends on the noise of $E(s)$, but not on the noise of the ciphertext (a, b).

To accelerate bootstraping procedure, following FHEW authors, one needs a homomorphic accumulator `ACC` holding values from $\mathbb{Z}_q$ and supporting a quadruple of algorithms (E, `Initialize`, `Update`, `Extract`) together with moduli t, q, where E and `Extract` may require key material related to an LWE key s as follows:

1. **Initialize**: `ACC` $\leftarrow b$, setting the content of `ACC` to any known value $b \in \mathbb{Z}_q$;
2. **Update**: `ACC` $\overset{+}{\leftarrow} c \cdot E(s)$, modifying the content of the accumulator from `ACC`$[v]$ to `ACC`$[v + c \cdot s]$, where $c, s \in \mathbb{Z}_q$, and s is given encrypted under E;
3. **Extract**: $f($`ACC`$)$, returning an encryption $\tilde{E}(f(v))$ of function f applied to the current content of the accumulator `ACC`$[v]$, where f is a "rounding" function from $\mathbb{Z}_q$ to $\mathbb{Z}_t$.

Suppose $ek = E(s) = (E(s_1), \dots, E(s_n))$ and $a = (a_1, \dots, a_n)$, the bootstrapping procedure is presented in algorithm 1.

Algorithm 1: Arithmetic bootstrapping using an accumulator ACC and rounding function f [12]

```
1 Bootstrap($ek = (E(s_i))_i, (a, b)$);
2 ACC $\leftarrow b$ ;
3 for $i = 1, \dots, n$ do
4     $c_i = -a_i \mod q$;
5     ACC $\overset{+}{\leftarrow} c_i \cdot ek_i$
6 end
7 return $f$(ACC)
```

As per [51], there are two competing bootstrapping approaches to FHEW-like schemes: the AP bootstrapping method [14] which is the basis of the original FHEW scheme, and the GINX bootstrapping method [13], adopted by TFHE. D. Micciancio and Y. Polyakov in [12] pointed out that the difference between these two implementations is that the former supports the basic update procedure `ACC` $\overset{+}{\leftarrow} E(s)$ for arbitrary $s \in \mathbb{Z}_q$, whereas the latter supports basic updates `ACC` $\overset{+}{\leftarrow} c \cdot E(s)$ with $c \in \mathbb{Z}_q$ being arbitrary, but $s \in \{0, 1\}$ is a single bit.

The detailed explanation of bootstrapping technique is complex and requires much mathematical background. As presented at the beginning, in the scope of our work, we aim to provide readers and newly interested users with fundamental principles of HE schemes without delving too deep into the mathematics. Thus, for expert readers and mathematicians, a complete description of the bootstrapping method can be found in their original papers at [9] and [10].

Symbol	Description
p	**The plaintext modulus** of BGV, BFV schemes
Q	**The maximum ciphertext modulus**, the initial ciphertext modulus after encryption
m	**The cyclotomic order** of the ring $\mathbb{R}$
N	**The degree** of the ring $\mathbb{R}$ ($N = \phi(m)$)
n	**The number of slots** or messages encoded in one ciphertext
L	**Multiplication depth**, the number of multiplications can be executed
Δ	**Scaling factor** in CKKS scheme, multiplied to the floating-point number of the message to convert to integer number.

Table 5: Used notations

4 Experimental implementation

The main focus of our work is to compare the performance of each available scheme in different libraries. For this reason, in each library, we build our own "simple" project as a regular end-user. Each project is corresponding to one scheme, which includes five main homomorphic operations: `KeyGen`, `Enc`, `Dec`, `Add`, and `Mult`. The execution time needed to perform each operation will be recorded and then compared to each other. Every program collecting the performance metrics is carried out on an average commodity computer equipped with an Intel(R) Core(TM) i7-10700 CPU running at 2.90GHz under Ubuntu 20.04. In the results presented in the next section, Table 5 lists some useful notations.

To ensure the consistency in test results, every experiment is executed according to the strategy below:

– The time unit is microseconds (μs);
– Each operation was executed in 1000 iterations and the time presented is its average;
– The parameters are chosen to ensure the 128-bit encryption security level;
– The time measured of encryption operation includes the execution time of random values for message inputs, together with encoding and decoding timings for batching;
– Bootstrapping is not applied.

Depending on different HE schemes' properties, chosen plaintext will be differed. BFV and BGV schemes allow modular arithmetic on encrypted integers, while CKKS supports homomorphic operations on real or complex ones. Within the scope of the paper, plaintext batching technique is applied for all evaluated FHE and SWHE schemes. The main idea behind *batching* concept is to pack n plaintexts/messages into one ciphertext for parallel processing. Here the first element of the batch is drawn randomly from a uniform distribution over the same range p, and the remaining elements are set to be 0. The diversity in input setting for each scheme will be explained with greater detail in section 5.

		SEAL-Python	**PALISADE**
Languages		Python	C++
Parameters	p	changeable	changeable
	N	changeable	changeable
	L	not changeable	changeable
	Q	not changeable	not changeable
Batching		$n = N$	$n = N$
Condition		$p = 1 \pmod{2n}$, p is a prime number	

Table 6: Differences in libraries' setups

5 Evaluation and results

5.1 Fully homomorphic encryption

5.1.1 BFV

Although BFV scheme is available in both SEAL and PALISADE as mentioned in Table 3, they also have their differences in the implementation, indicated in Table 6. PALISADE allows users to change the parameters p, N, L as inputs, whereas SEAL-Python keeps L unchangeable from the user side. To accelerate the batching technique, the two libraries require that the chosen plaintext modulus p needs to be a prime number and congruent to 1 (mod $2n$). This is the condition to operate on n packed integers in a SIMD (*Single Instruction, Multiple Data*) manner [28]. In order to assess the relative practical efficiency of two libraries for BFV encryption, different implementations are done with the same input parameters and working environment given in Table 7.

p	$log_2 Q$	required N
1032193	109	4096
1032193	218	8192
786433	438	16384
786433	881	32768

(a) SEAL

p	$(log_2 Q, L)$	required N
1032193	(120,1)	4096
1032193	(180, 2), (180, 3)	8192
786433	(240,4), (300,5), (300,6), (360,7), (360,8), (420,9)	16384
786433	(480,10), . . . , (780,19), (840,20), (840,21)	32768

(b) PALISADE

Table 7: BFV's input parameters in PALISADE and SEAL

Unlike SEAL, PALISADE can calculate required N and Q based on chosen L and p to ensure a security level of 128 bits. In contrast, SEAL sets 128-bit encryption security level as default and allows users to enter N. SEAL then displays satisfied Q and p with the entered inputs. Table 7 contains many options for PALISADE's inputs with the same p and N in order to have 128-bit security. However, to have fair comparison between these two, the value of Q in PALISADE is chosen to be close to the one in SEAL. In Table 8, we provide timings for five main cryptographic functions, using the parameters recommended in Table 7.

HE parameters		KeyGen	Enc	Dec	Add	Mult
N	log_2Q					
4096	109	1028.119	1263.528	276.045	1.298	3274.257
8192	218	3003.509	3269.548	1179.682	144.531	11663.16
16384	438	10260.45	11378.441	5434.016	415.662	54918.967
32768	881	40251.496	41297.274	17442.857	1536.587	246427.201

(a) SEAL

HE parameters		KeyGen	Enc	Dec	Add	Mult
N	log_2Q					
4096	120	1137.556	1160.459	283.99	0.237	4296.438
8192	180	3170.82	2881.717	921.646	187.703	13585.75
16384	420	13507.743	11288.5535	3298.9775	1086.105	76565.506
32768	840	55941.007	45587.262	17171.713	7046.362	427795.343

(b) PALISADE

Table 8: Horizontal comparison of BFV's execution time

After examining these tables, it is clear that the ciphertext dimension N has a significant effect on BFV's performance. In most cases, the running times of decryption and addition are less than the others. In general, when N increases, the execution times of all operations are increased, especially multiplication, which approximately grows up 4 times compared to the previous N in both two libraries. However, in particular, the mean multiplication execution time of SEAL is less than that of PALISADE. One explanation for this is that the latter always counts the relinearization procedure whenever doing multiplication (`EvalMult` function), while in the former, it is separately computed. Within the scope of our experiments, the decryption is executed only on a fresh ciphertext without doing multiplication before. Therefore, it is not necessary to do relinearization step. That is why the timing in SEAL does not involve relinearization.

In Figure 4, we depict experimental results in vertical comparison, where timings are illustrated based on each operation. It is obvious that the mean execution times of all cryptographic functions in two libraries are close to each other, but SEAL is still performing better. While the rest are almost similar, the biggest variance is displayed in multiplication time, where $N = 32768$, SEAL is approximately 2 times faster than PALISADE.

5.1.2 BGV

Unlike BFV, all three libraries SEAL, PALISADE, and HElib have implemented BGV. Generally, in doing the experiments, the encryption parameters

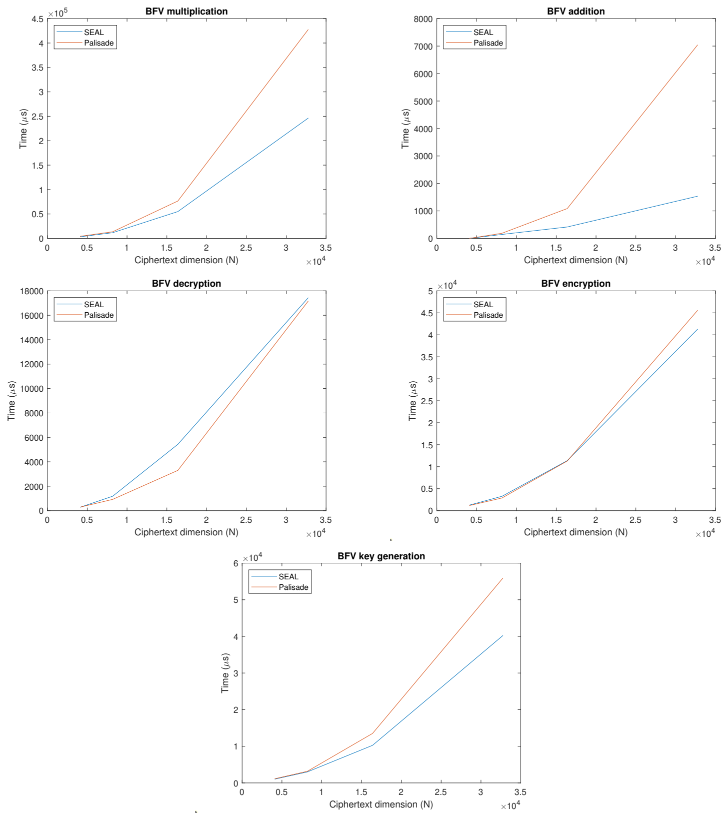

Fig. 4: Vertical comparison of BFV's execution time

of BGV, namely p, Q, N, and security level, are kept unchanged, compared to BFV. The number of slots in one batch is $n = N = \phi(m)$, except for the last case of Helib where $N = 32768$ and $n = 8192$ as indicated in Table 9. This number is impacted by several parameters, including the maximum supported computation depth of the circuit (L) [30]. As L is varied to allow more computation, it also affects the cost of the computation. Additionally, in practical implementation, some technical definitions have been introduced in HE libraries, *noise budget* is one of them. According to A. Kim [28], noise budget (invariant) is defined as the total amount of noise we have left until decryption will fail. To be more precise, the BGV implementation for each library is specified as follows.

PALISADE: In the library, noise budget is managed by a method called `ModReduceInPlace`, a method for reducing modulus of ciphertext and the private key used for encryption [29]. As explained above, in our scope of evaluation, this function will not be included. For BGV multiplication, the BFV operation of `EvalMult` is reused, so key-switching or relinearlization is already added. Besides, other properties of BGV implementation are remained the

HE parameters		KeyGen	Enc	Dec	Add	Mult
N	log_2Q					
4096	109	2424.838	1091.586	259.842	42.541	1508.681
8192	218	11426.94	3137.433	992.5	79.952	6673.09
16384	438	70869.416	11179.579	3791.998	292.17	35650.547
32768	881	433638.89	41716.827	18156.642	866.2635	215414.681

(a) SEAL

HE parameters		KeyGen	Enc	Dec	Add	Mult
N	log_2Q					
4096	96	3023.297	1145.76	368.375	42.116	570.952
8192	144	10981.757	3043.417	1007.424	57.322	2396.688
16384	240	51708.9	8902.513	3546.961	289.751	13642.014
32768	480	376273.9767	34662.558	20727.674	3313.547	116248.311

(b) PALISADE

HE parameters		KeyGen	Enc	Dec	Add	Mult
N	log_2Q					
4096	100	168300.764	2257.432	138092.51	32.064	2347.865
8192	100	470367.195	4533.877	549616.633	480.44	4492.487
16384	100	1348552.91	9917.878	2265994	289.706	10778.79
32768	100	1967110.87	14080.4445	2340201.2	209.039	17477.661

(c) HElib

Table 9: Horizontal comparison of BGV's execution time

N	Add Matrices	`KeyGen`	`Enc`	`Dec`	`Add`	`Mult`
4096	No	4882.369	2229.431	138098.495	120.27	2093.809
	Yes	168300.764	2257.432	138092.51	32.064	2347.865
8192	No	9903.988	4637.854	548176.341	389.5065	5463.149
	Yes	470367.195	4533.877	549616.633	480.44	4492.487
16384	No	20361.114	9418.236	2213333.56	577.842	11731.862
	Yes	1348552.91	9917.878	2265994	289.706	10778.79
32768	No	39825.16	13217.164	2373608.58	1039.197	21059.138
	Yes	1967110.87	14080.4445	2340201.2	209.039	17477.661

Table 10: HElib with different inputs

same as BFV's, such as the solution to calculate required N, Q, as well as the condition of inputs as mentioned in Table 6.

HElib: Helib allows to calculate its security level based on p, m, and `bits` (the number of bits of the modulus chain). When `bits` increase, its execution time is also raised up. Thus, in the comparison with other libraries as demonstrated on Table 9, we choose these variables such that the security level is close to or at least 128 bits.

Table 9 shows that the performance of HElib can be considered as good as the other two libraries if the timings of key generation and decryption were not such slow. To explain this, we need to examine the execution of key-switching matrices `addSome1DMatrices` in `KeyGen` process. Table 10 displays the differentiation in running time of computing or not the `addSome1DMatrices` function. Without adding this procedure, key generation has been much less time-consuming. For instance, in case $N = 32768$, it took almost 2 seconds to

p	(log_2Q, L)	required N
1032193	(96,1)	4096
1032193	(144, 2), (192, 3)	8192
786433	(240,4), (288,5), (336,6), (384,7), (432,8)	16384
786433	(480,9), (528, 10), (576,11),. . . (768,15), (816,16), (17,864)	32768

Table 11: BGV inputs for 128-bit security level in PALISADE

N	L	log_2Q	`KeyGen`	`Enc`	`Dec`	`Add`	`Mult`
8192	2	144	10981.757	3043.417	1007.424	57.322	2396.688
	3	192	18055.952	3662.841667	1499.093333	77.27016667	3906.558667
16384	4	240	51708.9	8902.513	3546.961	289.751	13642.014
	8	432	151107.142	13704.344	9176.451	1203.262	44985.406
32768	9	480	376273.9767	34662.558	20727.674	3313.547	116248.311
	17	864	1138783.22	58661.984	52499.905	3221.175	362749.472

Table 12: PALISADE with different inputs

generate its key pair with `addSome1DMatrices`, whereas this process costed only 40 milliseconds approximately without it.

In contrast, key-switching matrices have not been mentioned in SEAL and PALISADE. Instead, PALISADE calculates required N and Q as illustrated in Table 11.

The different pairs of L and Q in each line have the same level of security. Hence, in the horizontal comparison of Table 9, we selected PALISADE results with lower (L, Q) to compare with others. On the other hand, Table 12 contains the timing results when implementing the lowest and highest pairs of (L, Q) in each particular case of N value. Based on its behaviors, (L, Q) shows an impressing effect on PALISADE's execution time, especially on `KeyGen` procedure. For instance, at the same level $N = 32768$, $L = 17$ took more than 1 second to generate key pair, whereas 0.3 seconds is its cost when $L = 9$. Last but not least, Figure 5 exposes the visibly vertical comparison of the two based on timings of each operation.

A deep analysis of the Figure 5 and Table 9 shows that the SEAL and PALISADE are performing much better than the HElib for `KeyGen` and `Dec` operations. In contrast, Helib running time is the best in multiplication and encryption. On the other hand, PALISADE and SEAL have equally good performance in all operations. Although there is dissimilarity between them in multiplication and addition, since the actual time counted in μs, it is not really a great distance.

5.1.3 TFHE/FHEW

The FHEW fully homomorphic encryption [9] and its TFHE variant [10] are the well-known methods to compute simple bit operations on encrypted data. TFHE and FHEW are both Ring-LWE encryption, followed by a bootstrapping procedure. Since bootstrapping notations are different from the notations in previous parts, first we define them as follows.

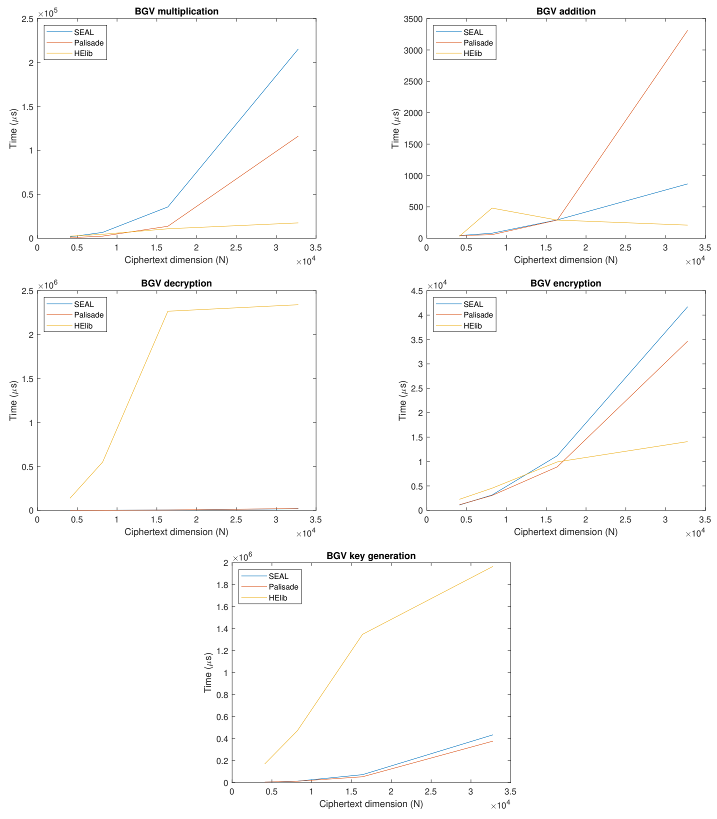

Fig. 5: Vertical comparison of BGV's execution time

Parameter set	n	N	q	Q
STD128	512	1024	512	27
STD128_AP	512	1024	512	27
STD192	512	2048	512	37
STD256	1024	2048	1024	29
STD128Q	512	2048	512	50
STD192Q	1024	2048	1024	35
STD256Q	1024	2048	1024	27

Table 13: Parameter sets for of FHEW/AP and TFHE/GINX in PALISADE

- n, lattice parameter for the LWE scheme;
- N, ring dimension for RLWE;
- q, LWE modulus;
- Q, RLWE modulus used in the core bootstrapping procedure based on an accumulator.

Parameter set	`KeyGen`		`NAND`	
	AP [ms]	GINX [ms]	AP [ms]	GINX [ms]
STD128	4131.241	1020.15	211.7	148.87
STD128_AP	3213.657	977.524	167.19	120.89
STD192	8034.154	1837.621	773.335	504.65
STD256	31390.548	4884.21	899.59	614.28
STD128Q	10303.304	2584.294	546.265	389.05
STD192Q	17484.524	3074.128	714.29	498.95
STD256Q	-	2463.208	-	1421.81

Table 14: Comparison on execution time of FHEW/AP and TFHE/ GINX

Both AP/FHEW and GINX/TFHE are implemented in PALISADE (now OpenFHE). Based on proposed parameter sets by D. Micciancio et al. [12] and PALISADE configuration, we conduct the experiments with different values as shown on Table 13. PALISADE provides several parameter sets corresponding to various levels of security: STD128, STD192, STD256, STD128_AP, STD128Q, STD192Q, and STD256Q, where "STD" means HE security standard in [52], and "Q" stands for the quantum attack estimates. For example, STD128 is HE standard set with more than 128 bits of security with reference to classical computer attacks, while STD128Q is the same as STD128 security but with reference to quantum computer attacks. For the 128-bit security, STD128_AP is added, which supports a more efficient option with $d_g = 3$, while STD128 is with $d_g = 4$, where d_g is the number of digits that integers (mod Q) are broken into.

The method to conduct experiments for TFHE/FHEW is similar to previous experimental implementations in PALISADE as presented in section 4. In particular, for each scheme we build a project as a regular end-user. Since FHEW and TFHE can evaluate arbitrary Boolean circuits on encrypted data by bootstrapping after each gate evaluation [12], we focus on performing and comparing two main operations: key generation (`KeyGen`) and NAND-gate evaluation (`NAND`). The former includes generating refresh and switching keys, while the latter is evaluating a NAND gate. A NAND gate is functionally complete. Hence, every possible Boolean circuit can be realized with combinational logic made entirely of NAND gates. Every experiment collecting the performance metrics is carried out on an average computer equipped with an 12th Gen Intel(R) Core(TM) i5-1245U CPU running at max 4.4GHz under Ubuntu 22.04. We compiled PALISADE v1.11.6 with the compiler clang version 14.0.0.

Based on proposed parameter sets on 13, the runtime results are summarized in Table 14 counted by milliseconds (ms). The number of security bits has a great impact on the running time of both two cryptosystems. It is understandable that the increase in security level leads to the increase in `KeyGen` and gate-evaluation timings in both two schemes. This is most clearly shown in the fact that the system was not able to compute any results for FHEW/AP when STD256Q is reached. In general, GINX bootstrapping method provides better performance when it always produces bootstrapping keys and evaluates the NAND gate much faster than AP. Talking about this, TFHE authors [10]

(log_2Q, L)	required N
(101,1)	8192
(140,2), (181,3), (221,4), (261,5), (301,6)	16384
(341,7), (381,8)	32768

Table 15: CKKS input parameters in PALISADE

explained that they used a smaller bootstrapping key than the one in AP. In their experiment, using 16MB bootstrapping key instead of 1GB, the running time of FHEW bootstrapping is decreased from 690ms to 13ms single core, but still preserving the security parameter. Comparing the two, Y. Lee et al. [51] stated that GINX/TFHE bootstrapping uses much smaller evaluation keys, but it restricts the scheme's applicability because it is directly applicable only to binary secret keys. On the other hand, AP/FHEW supports arbitrary secret key distributions, which is critical for a number of important applications, such as threshold and some multi-key HE schemes.

5.2 Somewhat homomorphic encryption

The CKKS scheme is called *leveled homomorphic encryption*, an "extended" form of SWHE. In contrast to BFV and BGV encryption, where exact values are necessary, CKKS allows both additions and multiplications on encrypted complex numbers, but yields only approximate results [28]. According to A. Kim [39], one should take advantage of CKKS encryption in applications such as summing up encrypted real numbers, evaluating machine learning models on encrypted data, or computing distances of encrypted locations. As a result, CKKS scheme has been implemented in four HE libraries as communicated in Table 3. To perform experiments with CKKS, in addition to the default setting mentioned above, the input parameters are the same for all libraries, where:

– Scaling factor $\Delta = 2^{40}$;
– For batching technique, $n = N/2$.

In this encryption, there is no condition of plaintexts. Based on its properties, we chose inputs as real numbers. The method to draw packed messages keeps unchanged as discussed in section 4.

PALISADE: The ring dimension of the HE scheme is chosen following the security standards. Hence, to meet a requirement of 128-bit security level, the minimum value of N is 8192. In Table 15, we presents the detail of input parameters.

SEAL: Like other CKKS implementation, SEAL does not use the plaintext-modulus parameter p. Moreover, instead of providing a ciphertext modulus Q, users working with CKKS must provide a modulus chain of prime sizes (e.g., $q = [60, 40, 40, 60]$) [39]. The number of moduli is equal to the number of iterations/multiplications. Additionally, the log_2Q bit as shown in Table 7 is kept

N	log_2Q	**Modulus chain**	`KeyGen`	`Enc`	`Dec`	`Add`	`Mult`
8192	160	60, 40, 60	2179.13	3206.887	50.54	128.72	178.348
	200	60, 40, 40, 60	2507.607	3910.8775	109.5735	271.207	452.792
16384	200	60, 40, 40, 60	5034.227	8548.111	221.213	317.991	774.644
	432	60, [39]*8, 60	11959.254	18847.575	721.076	1777.956	2077.872
32768	200	60,40,40,60	10215.043	18231.808	781.699	529.59975	1414.666
	881	[55]*15,56	39749.74	66061.559	2589.904	2221.2535	4741.014

Table 16: SEAL's comparison for different modulus composition

unchanged, but now it is corresponding to the maximal sum of these primes, called `CoeffModulus`.

Before going to the evaluation part of different libraries' performance, Table 16 illustrates how SEAL behaves sensitively with ciphertext modulus and its modulus composition for each value of N. Although addition and decryption time are not changed significantly, the calculation time is climbed up more than 2 times in the three remaining operations.

HElib: One of the most advantages of working with HElib is its transformation of complex mathematical calculations in order to be easier and more understandable for non-expert practitioners. For example, to add two ciphertexts `cipher_a` and `cipher_b`, HElib supports to simply declare a new one as a sum of the two: `Ctxt cipher_add_ab = cipher_a; cipher_add_ab += cipher_b`. There is no need to specify technical steps such as relinearization or rescaling. Being different from other libraries, HElib allows users to calculate encryption security level based on input parameters. Table 17 contains the experimental results with the encryption security `sec_level` being the closest to 128-bit level, while still preserving HElib's usage recommendation.

HEAAN: The last library was developed by its own authors. HEAAN takes advantage of fully built-in algorithms, where it is able to deal with complex numbers. An input message in HEAAN can consist of n complex numbers, where $n \leq N/2$.

HE parameters		KeyGen	Enc	Dec	Add	Mult
N	log_2Q					
8192	200	2507.607	3910.8775	109.5735	271.207	452.792
16384	432	11959.254	18847.575	721.076	1777.956	2077.872
32768	881	39749.74	66061.559	2589.904	2221.2535	4741.014

(a) SEAL

HE parameters		KeyGen	Enc	Dec	Add	Mult
N	log_2Q					
8192	102	2305.699	2652.975	21650.503	81.117	3129.505
16384	141	6542.385	7093.977	51639.085	194.05	9584.286
32768	342	31630.72	29936.449	248985.192	3291.783	66603.916

(b) PALISADE

HE parameters		KeyGen	Enc	Dec	Add	Mult
N	(log_2Q,sec_level)					
8192	(119,157.866)	11008.069	2659.019	22065.082	272.865	19712.186
16384	(358,129.741)	91768.896	8252.838	107935.827	1502.701	104850.697
32768	(558,128.851)	164575.383	23730.201	364743.317	11576.171	215878.991

(c) HElib

HE parameters		KeyGen	Enc	Dec	Add	Mult
N	log_2Q					
8192	119	2282102.44	634268.04	41491.42	39877.65	614878.85
16384	358	2294477.86	624440.22	93658.41	17826.4	994892.6
32768	558	2251482.12	657943.99	114587.91	45690.59	1332368.41

(d) HEAAN

Table 17: Horizontal comparison of CKKS's execution time

By analyzing different results displayed in Figure 6, one can see that overall performance of HEAAN and Helib are quite slower than SEAL and PALISADE. In overall, SEAL owns the best performance, while HEAAN is much more time-consuming compared to the others. Considering PALISADE's presentation in both Table 17 and Figure 6, it is obvious that its most time-consuming procedure is decryption and multiplication. One reason needed to bring up is that, the relinearization step is always included in multiplication function `EvalMult`. Moreover, for the decryption process (`cc->Decrypt(keys.secretKey, cMul, &result)`), calculated time of rescaling algorithm is also taken into account. In contrast, SEAL does not include relinearization and re-scaling schemes in multiplication and decryption respectively. Instead, it can be done by coding separately with two functions: `evaluator.relinearize_inplace` and `evaluator.rescale_to_next_inplace`.

5.3 Partially homomorphic encryption

This part presents our own implementations of partially homomorphic cryptosystems, including Paillier (additive), El-Gamal (multiplicative), and RSA (multiplicative). The source code is available at `github` [53]. Table 18 and Figure 7 illustrate horizontal and vertical comparison results respectively. Ac-

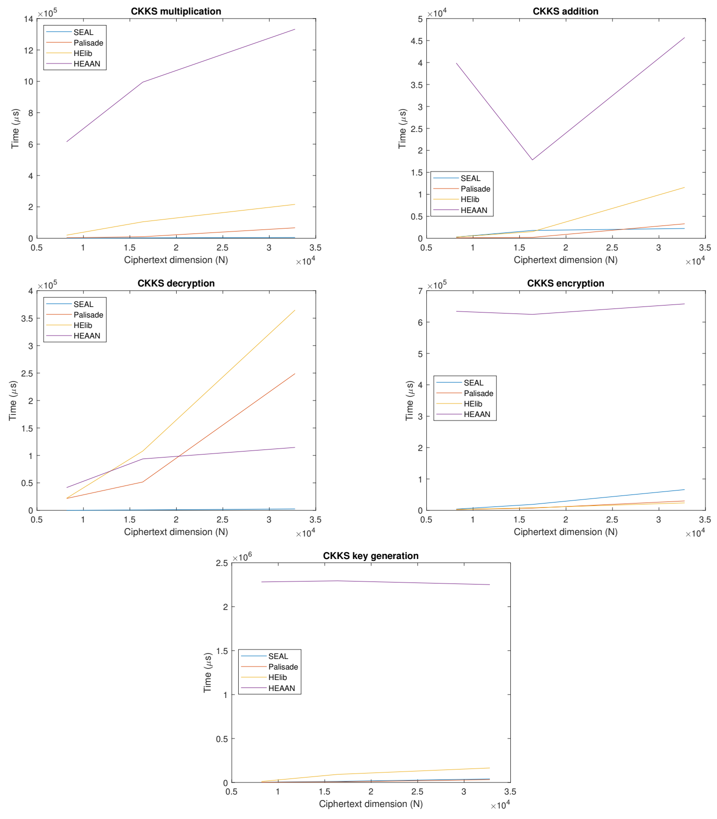

Fig. 6: Vertical comparison of CKKS's execution time

cording to PHE's properties as introduced in section 1, one PHE scheme can possess four following operations: Key generation, encryption, decryption, and addition/multiplication. Unlike FHE and SWHE, here the inputs are identified as p (plaintext modulus) and $log_2 N$ (the number of bits of N), where N is one factor in public (encryption) keys. For each cryptosystem, we measure the execution time when selecting pairs of $(log_2 N, p)$ in the similar manner of selecting $(log_2 Q, p)$ in FHE. In addition, we execute the second situation where p is in $\mathbb{Z}_N$ and the bits of N are so large such that they can reach 128-bit security level as stated in [54]. As same as the previous implementations, the time unit is microseconds; each operation was executed in 1000 iterations and the timings presented are its average. The implementations are set up following their original papers: Paillier [4], El-Gamal[3], and RSA [15]. Table 18 and Figure 7 demonstrate the experimental results.

5.3.1 Paillier encryption

In Paillier cryptosystem, although the input is N, the cipher space or ciphertext modulus is N^2 (see section 3). In spite of that, generally the algorithm performs all four operations very well as shown in Table 18. In the second situation, when both p and N increase, there is no much difference in time execution of `Add`. However, for three others, they are both climbed up. Particularly, when N is 4096 bits, the average time for one `KeyGen` is almost 4.3 seconds.

5.3.2 El-Gamal encryption

Table 18 indicates that all three operations of encryption, decryption, and multiplication in El-Gamal method have a better performance compared to Paillier. Apart from that, `KeyGen` appears to be a very time-consuming procedure. The cryptosystem needs more than 5 seconds to generate key pairs if $log_2 N = 881$, not mention to say that it needs more than 15 minutes when $log_2 N = 3072$ or more. Regarding to this problem, its author Taher ElGamal explained that in any of the cryptographic systems based on discrete logarithms like El-Gamal, N must be chosen such that $N - 1$ has at least one large prime factor [3]. If $N - 1$ has only small prime factors, computing discrete logarithms would be easy [55]. Hence, our implementation is set up such that this condition is satisfied. N is considered as a safe prime if $(N - 1)/2$ is also a prime.

5.3.3 RSA encryption

It is clearly seen in Figure 7 that RSA has represented the best performance among the three PHE schemes, even in case of very large ciphertext space. As its authors stated in [15], the secret key d in RSA is very easy to choose, which is relatively prime to $\phi(N)$, where $N = pq$. To be more specific, any prime number greater than $\max(p, q)$ will do. This is one of the reasons why RSA does not take much time to generate keys like El-Gamal encryption and why it is commonly used in practice.

In Figure 7, the first graph on the top-left side displays the running time of Addition (`Add`) in Paillier and Multiplication (`Mult`) for the remaining two cryptosystems. Although the difference among them is demonstrated visibly, it is still considered as marginally small for the time unit is in μs.

6 On the Security of HE

6.1 HE under security notions

All four general-purpose libraries presented in the paper were based on RLWE-based systems. The most interesting advantage of LWE or RLWE is that it is considered as one of the hardest problems to solve in practical time for even post-quantum algorithms [45]. However, this does not mean that RLWE-based HE schemes are totally secure. In fact, to prove security of encryption algorithms, two security models commonly referred are IND-CPA and IND-CCA, standing for *Indistinguishability under chosen plaintext attack* and *Indistinguishability under chosen ciphertext attack* respectively. For IND-CCA, there

HE parameters		KeyGen	Enc	Dec	Mult
p	$log_2 N$				
1032193	109	1484.324	1.062	5.399	2.237
1032193	218	1931.446	1.669	7.803	0.396
786433	438	3490.179	2.496	37.265	0.853
786433	881	8366.837	6.865	205.38825	2.178
$\mathbb{Z}_N$	3072	180255.34	61.599	6327.537	2.934
$\mathbb{Z}_N$	4096	433348.8	88.372	14327.857	9.792

(a) RSA encryption

HE parameters		KeyGen	Enc	Dec	Mult
p	$log_2 N$				
1032193	109	31063.45	4.486	4.336	3.151
1032193	218	135618.53	16.748	15.476	9.74
786433	438	773368.775	66.794	32.56	18.388
786433	881	5354554.333	403.448	203.718	8.651
$\mathbb{Z}_N$	3072	>15 minutes			
$\mathbb{Z}_N$	4096	>15 minutes			

(b) El-Gamal encryption

HE parameters		KeyGen	Enc	Dec	Add
p	$log_2 N$				
1032193	109	1072.014	265.509	6.738	4.255
1032193	218	1537.688	279.664	22.872	4.053
786433	438	3081.14	367.893	141.012	6.08
786433	881	7903.175	1013.957	950.735	10.868
$\mathbb{Z}_N$	3072	183774.01	20237.659	26364.306	232.136
$\mathbb{Z}_N$	4096	4297885.6	42868.889	55843.731	212.978

(c) Paillier encryption

Table 18: Horizontal comparison of PHE's execution time

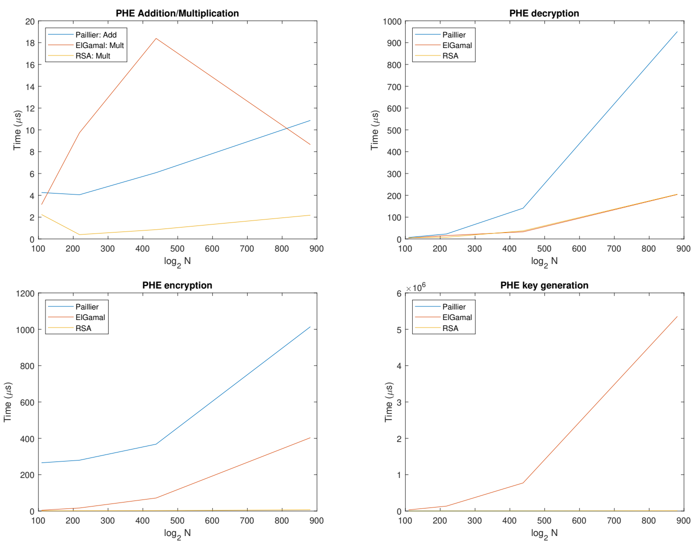

Fig. 7: Vertical comparison of PHE's execution time

are IND-CCA1 and IND-CCA2. The former is Indistinguishability under non-adaptive chosen ciphertext attack, while the latter is the adaptive one.

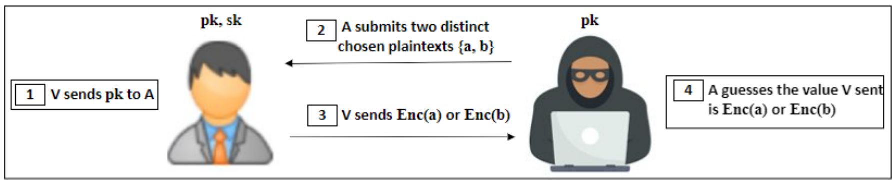
pk, sk
pk
1 V sends pk to A
2 A submits two distinct chosen plaintexts {a, b}
3 V sends Enc(a) or Enc(b)
4 A guesses the value V sent is Enc(a) or Enc(b)

Fig. 8: IND - CPA security notion

IND-CPA is modeled by a game between an adversary (A) and a verifier (V) as illustrated in Figure 8. In general, after generating *pk*, *sk*, and other security parameters of an encryption system, V sends *pk* to A. From this point, A is free to perform any computations using *pk*. A then chooses two different plaintexts a, b and send them to V. V computes encryption of a or b uniformly at random and sends A the result, called *challenge*. Finally, A needs to conclude the received value is the encryption of a or b. The cryptosystem is said to be secure in terms of IND-CPA if no adversary can output the correct value with probability significantly better than $\frac{1}{2}$. Likewise, the definition of IND-CCA is similar to IND-CPA, but here in both IND-CCA1 and IND-CCA2, the attacker can ask for the decryption of any ciphertexts, except the challenge that the verifier sent. In particular, IND-CCA1 and IND-CCA2 all allow the attacker to make queries to the decryption oracle to decrypt any arbitrary ciphertexts before the step 3 in Figure 8, when the verifier sends the challenge to the adversary. However, after the step 3, the adversary may not make further calls to the decryption device in IND-CCA1, while it is allowed in IND-CCA2. The security under IND-CCA2 implies the security under IND-CCA1, and the security under IND-CCA1 also implies the security under IND-CPA. In other words, an encryption scheme which is IND-CCA2 secure is both IND-CCA1 and IND-CPA secure.

6.1.1 RLWE - based FHE and SWHE schemes

In 1999, Bellare et al. [56] proved that all homomorphic encryption schemes are not secure against IND-CCA2 attacks. Subsequently, although IND-CCA2 is the strongest of the three security definitions, it is universally acknowledged that IND-CCA1 is the strongest security notion for HE. Apart from these three, Chenal and Tang [57] mentioned one variation of these security notions, called *key recovery* attacks. Following the authors, the key recovery attack is stronger than a typical IND-CCA1 and allows an adversary to recover the private keys through a number of decryption oracle queries.

Table 19 lists several FHE and SWHE schemes presented in our work and corresponding attacks, together with their related papers. It is obvious

Scheme	**IND-CPA**	**IND-CCA1**	**Key recover attack**
BFV	✓	Fauzi et al. [58]	Z. Peng [59]
BGV			Chenal and Tang [57]
CKKS			Li et al. [60]
FHEW			Chenal and Tang [57]
TFHE			Chenal and Tang [57]

Table 19: Security of several FHE and SWHE schemes

that three schemes are secure against IND-CPA attacks [58]; however, they all suffers from IND-CCA1 and key recover attacks. According to Fauzi et al., the key recovering attack as presented by Chenal and Tang also works on a on several schemes based on (R)LWE, and FHEW/TFHE is one of them. When the decryption is computed as a rounding function that maps elements from $\mathbb{R}_q$ into the plaintext space, it is vulnerable to an attacker who asks for decryptions of $c = (e_i, b)$, where $e_i \in \mathbb{R}^n$ is the unit vector with 1 at position i and 0 everywhere else. As a consequence, this leaks information on secret s_i.

6.1.2 PHE schemes

In contrast to FHE and SWHE, one of PHE schemes, namely RSA, is weak even under IND-CPA norm. The reason is that Schoolbook RSA is deterministic. Therefore, comparing to IND-CPA model in Figure 8, to guess the correct output at step 4, the adversary can compute $a^e \pmod N$ and $b^e \pmod N$ then check which one is matched to the verifier's challenge. Thus, RSA is not IND-CPA secure, which also implies that it is not IND-CCA secure either. Unlike RSA, Paillier encryption is IND-CPA secure under Decisional Composite Residuosity (DCR) Assumption [61]. To be more precise, Armknecht et al. [62] proved that Paillier scheme is secure against IND-CCA1 attacks if and only if DCR^{SCCR} is hard, where SCCR is Subgroup Computational Composite Residuosity problem [4]. Similar to Paillier system, El-Gamal encryption scheme is also known as being IND-CPA secure under the decisional Diffie-Hellman assumption [63]. However, when discussing the security of El-Gamal under IND-CCA1, Wu and Stinson [64] supposed that it is conjectured, but there has been no formal proof.

Integer factorization problem (IFP). Apart from attacks on security notions, IFP is also worth discussing in our context when we have the security of both RSA and Paillier cryptosystems depending on factoring problem. Given a composite number N, The IFP is defined as finding two integers p and q such that $pq = N$. Once p and q are discovered, it can be shown that RSA and Paillier encryption are insecure (see section 3). Two of widely used algorithms to factor an integer, as well as to be the basis of other factorization methods, are Pollard's *rho* and Pollard's $p - 1$, invented by John Pollard in 1974 - 1975 [65]. Our implementation and experimental results of each method are presented in detail at [66].

With complexity of time and space $\mathcal{O}(\sqrt{N})$ by the birthday paradox, Pollard's *rho* relies on several important mathematical concepts, one of them is cycle-finding algorithm.

Algorithm 2: Pollard's *rho* algorithm using Floyd's cycle detection

```
Input: a composite number $N$, a bound $B$ for the number of iterations
Output: a nontrivial factor of $N$ or failure
1  $x \leftarrow 2$ ;                    // Set $x = x_0 = 2$ to be the initial value
2  $y \leftarrow 2$ ;                                      // Set $y = x_0 = 2$
3  $d \leftarrow 1$ ;
4  $i \leftarrow 0$ ;
5  while $d = 1$ OR $d = N$ do
6      if $i \geq B$ then
7          return failure;              // Maximum number of iterations reached
8      end
9      $x \leftarrow f(x)$ ;                                        // $x = x_i$
10     $y \leftarrow f(f(y))$ ;                                     // $y = x_{2i}$
11     $d \leftarrow \gcd(|x - y|, N)$ ;
12     $i \leftarrow i + 1$ ;
13 end
14 if $d = 1$ OR $d = N$ then
15     return failure ;
16 end
17 else
18     return $d$ ;
19 end
```

To reduce the memory cost, Pollard applied the idea of Floyd's cycle detection algorithm (see algorithm 2): two pointers x and y are used; pointer x holds the values of x_i's and pointer y holds the values of x_{2i}'s. Each iteration updates the values of x and y by computing $f(x)$ and $f(f(y))$, then checks if $\gcd(x_i - x_{2i}, N) = \gcd(x - y, N)$ is a nontrivial factor of N. This reduces the memory cost to $\mathcal{O}(1)$. Assuming f is a random function, then the expected number of evaluations to the function f performed by Pollard's *rho* algorithm is $\mathcal{O}(\sqrt{p}) = \mathcal{O}(\sqrt[4]{N})$, where p is the smallest prime factor of N. We present experimental results of Pollard's rho algorithm on our classical laptop with the following three numbers:

$$\begin{aligned} n_1 &= 1125939825397831601 \\ &\quad \text{(a 60-bit RSA modulus)} \\ n_2 &= 925276410789441750962080530947 \\ &\quad \text{(a 100-bit RSA modulus)} \\ n_3 &= 115792089237316195423570985008687907853269984 \\ &\quad 665640564039457584007913129639937 \\ &\quad \text{(Fermat number } F_8 = 2^{2^8} + 1) \end{aligned}$$

Input number	**Average running time** (s)	**Standard deviation**
n_1	0.0054	0.0016
n_2	5.9370	3.3053
n_3	67.0664	43.6244

Table 20: Average running time with different initial values

Input number	**Running time** (s)	**Digits in factor**
n_4	970	20
n_5	819	20
n_6	27.1042	20

Table 21: Running time to find medium-size factor

For each run we use the same function $f(x) = x^2 + 1 \mod N$ and same maximum number of iterations $B = 10^8$. The results are in Table 20.

Table 21 shows the running time and size of factor when we run the algorithm with $B = 10^{10}$ to find a medium-size factor (around 20 digits) of some worst-case numbers:

$$\begin{aligned}
n_4 &= 237130450584081431781941097598542348001\\
&= 15351399207396244631 \times 15446829789289363271\\
&\quad \text{(a 128-bit RSA modulus)}\\
n_5 &= 304075290252258958535257891241265214597\\
&= 18229633569899862109 \times 16680274405204585033\\
&\quad \text{(a 128-bit RSA modulus)}\\
n_6 &= 34! - 1\\
&= 295232799039604140847618609643519999999\\
&= 10398560889846739639 \times 28391697867333973241
\end{aligned}$$

It can be seen that Pollard's *rho* takes a lot of time to find a factor of medium size. The second method to factor N is Pollard's $p-1$ algorithm, which is based on Fermat's Little Theorem [65]. Unlike the previous method, the possibility of finding a factor p of given size is not determined solely by its size, but rather by the smoothness of $p-1$ (see algorithm 3). There are two cases of failure: $d = 1$ or $d = N$. In the first case, $a^M - 1$ is co-prime with N, which implies that the search bound B is too small, and thus one should rerun the algorithm with a larger B. In the second case, $d = N$ implies that N has a B-smooth prime factor p, but the randomized base a has order less than $p-1$ modulo p (hence omitted for gcd computation in the loop from line 8 to 11). In this case we choose another base a and restart the whole process. Here, to improve the algorithm's performance, we implement the *two-stage* variant of Pollard's $p-1$ (the detail is presented in [66]). The second-stage is performed by choosing a second bound $B_2 > B$, normally $B_2 = 100B$. While Pollard's *rho* takes much time to find a factor of medium size, our implementation of the

Algorithm 3: Pollard's $p-1$ algorithm

```
   Input: a composite number $N$, a bound $B$
   Output: a nontrivial factor of $N$ or failure
1  Choose a positive integer base $a$ randomly between 1 and $N$;
2  Compute $d = \gcd(a, N)$;
3  if $d \neq 1$ then
4  |   return $d$;
5  end
6  for prime numbers $p_i \leq B$ do
7  |   $q \leftarrow 1$;
8  |   while $q \leq B$ do
9  |   |   $a \leftarrow a^{p_i} \mod N$;
10 |   |   $q \leftarrow q \times p_i$;
11 |   end
12 |   $c \leftarrow a - 1$;
13 |   $d \leftarrow \gcd(c, N)$;
14 |   if $d \neq 1$ AND $d \neq N$ then
15 |   |   return $d$;
16 |   end
17 |   if $d = N$ then
18 |   |   Go to line 1 and choose a new value for $a$;
19 |   end
20 end
21 if $d = 1$ then
22 |   return failure;
23 end
```

Digits of factor	n	B_1	B_2	**Time**(s)
32	$2^{977}-1$	10^7	10^8	14.9582
34	575th Fibonacci number	10^7	10^8	14.3806
66	$960^{119}-1$	10^8	10^{10}	1076

Table 22: Our running time of some record factors by Pollard's $p-1$ method

two-stage Pollard's $p-1$ is able to find larger factors of some record numbers listed in [67]. The running time for each is displayed in Table 22, where :

$$n_7 = 2^{977} - 1$$
$$n_8 = \text{575th Fibonacci number (120 digits)}$$
$$n_9 = 960^{119} - 1$$

We found the factor of each number as below:

$$p_1 = 49858990580788843054012690078841$$
(32-digit factor of n_7)
$$p_2 = 7146831801094929757704917464134401$$
(34-digit factor of n_8)
$$p_3 = 672038771836751227845696565342450315062141551\,5$$
(66-digit factor of n_9)

Authors	Year	Gates	Total qubits
Shor [68]	1994	$\mathcal{O}(n^3 \log n)$	$\mathcal{O}(n)$
Beckman et al. [74]	1996	$\mathcal{O}(n^3)$	5n+1
Veldral et al. [75]	1996	$\mathcal{O}(n^3)$	4n+3
Beauregard [76]	2003	$\mathcal{O}(n^3 \log \frac{n}{\epsilon} \log \frac{1}{\epsilon})$	2n+3
Takahashi et al. [77]	2006	$\mathcal{O}(n^3 \log \frac{n}{\epsilon} \log \frac{1}{\epsilon})$	2n+2
Haner et al. [78]	2016	$\mathcal{O}(n^3 \log n)$	2n+2
Gidney [79]	2017	$\mathcal{O}(n^3 \log n)$	2n+1

Table 23: Different implementations of Shor's algorithms on IFP [80]

It is obvious that Pollard's $p-1$ is capable to find large factors of a composit N; however, in practice, when the system applies N from 2048 bits, it is not sufficient to find its large-size factors using Pollard's *rho* and Pollard's $p-1$ methods on a classical machine. Therefore, the invention of *Shor's algorithm* by Peter Shor, a quantum computer algorithm to solve IFP, marks an important milestone for the security of public-key cryptography systems.

6.2 Shor's quantum algorithm

Being developed in 1994, Shor's algorithm [68] is one of the first quantum algorithms that demonstrated the advantage of quantum computers over classical ones. In general, the method allows to find prime decomposition of big integers in polynomial time, namely $\mathcal{O}((\log N)^3)$ time and $\mathcal{O}(\log N)$ space, given a sufficiently large quantum computer.

The basic idea of Shor's algorithm relies on period-finding problem. Given integers a and N, r is called the *period* of a modulo N if r is the smallest positive integer such that $a^r - 1$ is a multiple of N, or $a^r - 1$ is divisible by N. For example, given $a = 7$ and $N = 15$, its period is found as $r = 4$, we have $7^4 = 1 \pmod{15}$. The name "period" comes from the fact that $a^{i+r} \pmod N = a^i a^r \pmod N = a^i \pmod N$ (because $a^r = 1 \pmod N$) for any integer i. Based on the period's property, we have $N|(a^r - 1)$. If r is even, then $N|[(a^{r/2}-1)(a^{r/2}+1)]$. By computing $\gcd((a^{r/2}-1), N))$ and $\gcd((a^{r/2}+1), N)$, we can find the factors of N. In the whole process, a quantum algorithm is applied to compute the period r of a modulo N by using quantum Fourier transforms [69], where a is a randomly chosen element.

So far, the largest numbers factored by Shor's algorithm are 51 and 85 by Geller and Zhou in 2013 using eight qubits [70]. Before that, Vandersypen et al. [71] in 2001 and Martín-López et al. [72] in 2012 also implemented Shor's algorithm to factor 15 and 21 respectively. The most recent paper was of Gidney et al. [73], published in 2021, which presented how to factor 2048-bit RSA integers using 20 million noisy qubits in 8 hours.

With the efficiency of quantum computers, the security provided by cryptosystems, which are based on IFP and discrete logarithmic problems (DLP), seems to be short-lived. Speaking of IFP, RSA and Paillier encryption are vulnerable against Shor's algorithm. In [80], Suo et al. indicate some imple-

Authors	**Year**	**Time complexity**	**Space complexity**
Shor [68]	1994	$\mathcal{O}(n^3)$	$\mathcal{O}(n)$
Proos et al. [81]	2003	$\mathcal{O}(n^2)$	-
Ekera et al. [82]	2019	–	$\mathcal{O}(n^2)$

Table 24: Different implementations of Shor's algorithms on DLP [80]

mentations of Shor's algorithm over different quantum prototype computers, together with their number of qubits and quantum gate complexities, as shown on Table 23 (for IFP) and Table 24 (for DLP).

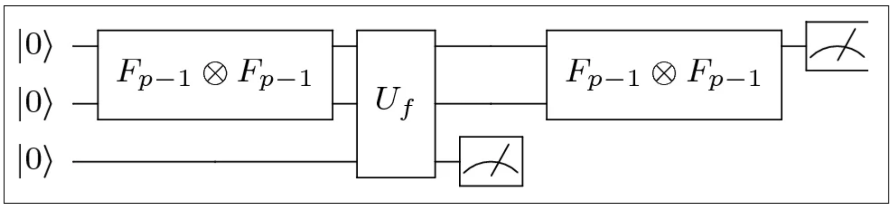

Fig. 9: A circuit of DLP [83]

Similarly, El-Gamal with its hardness of computing discrete logarithms is also a victim of quantum algorithms [68]. In 2010, Wang [83] defined a circuit for quantum computers to solve DLP as shown on Figure 9, where F_{p-1} is the Fourier transform over Z_{p-1}, and U_f being a quantum circuit.

Some argue that although quantum encryption breaking is a potential possibility, it is not a peril as there are still solutions for it. One is to increase the bit lengths, so that attackers need a larger and larger quantum computer to be able to successfully break the system. The second is to develop new public key cryptosystems that cannot be solved by Shor's algorithm. This opens a new era of Post-quantum cryptography (PQC), or quantum-resistant cryptography.

7 Conclusion and Future work

In this work, we made a performance comparison of several notable HE schemes, covering all three homomorphic encryption categories: Partially HE, Somewhat HE, and Fully HE. The results clearly suggest that partially homomorphic cryptosystems are significantly faster than the others at addition and multiplication operations. RSA possesses the fastest key-generation procedure, whereas El-Gamal is quite slow when ciphertext modulus increases. On the other hand, the presentation between evaluated FHE schemes and CKKS is inconsistent, especially for multiplication timings. For both BFV and BGV, SEAL and PALISADE demonstrate a slower multiplication compared to CKKS. In contrast, HElib makes that of CKKS be a time-consuming process in comparison to the other two. Besides, between BFV and BGV, performance analysis has shown that the former is performing better than the latter in terms of execution time for key generation in both SEAL and PALISADE, mostly when the ciphertext dimension is climbed up.

Nowadays, in the vibrant and active world, when data privacy plays a more significant role, HE is a new promising domain that allows external third parties to perform computations on the encrypted data without decrypting it in advance. However, one big challenge is to build a HE scheme that provides simultaneously both the required security and the performance efficiency. In this paper, we contributed an in-depth study of the different uses and implementations of HE schemes in most-used HE libraries, including SEAL, PALISADE, HElib, and HEAAN. First, we highlighted the principles and mathematical models of adopted schemes, followed by a brief description of linked libraries. By comparing execution time of five main homomorphic operations (`KeyGen`, `Enc`, `Dec`, `Add`, `Mult`), we present a computational overview of performance evaluation of different HE cryptosystems in different libraries. Through experimental results, it is easier for non-experienced practitioners to set input parameters for encryption schemes, as well as to choose an appropriate library for building their own HE-based projects. We also discussed an overview of the security of six aforementioned HE schemes under notable security notions such as IND-CPA, IND-CCA1 and IND-CCA2. Two classical attacks of Prof. John Pollard on Integer factorization problem are presented before introducing Shor's quantum algorithm for the same problem.

It is also clear that the efficiency of the PHE schemes becomes crucial in the overall performance. Due to the fact that PHE schemes are not implemented in mentioned libraries, we used our own implementations of Paillier, El-Gamal, and RSA as partially homomorphic cryptosystems in the emulation. For that reason, we plan to continue our work on optimizing PHE schemes in their implementation and performance. Besides, working with different HE libraries, we have seen that a part from doing encryption between a typical two parties, some libraries also support threshold encryption. Using threshold decryption in a public key cryptosystem allows n parties to communicate in which a minimal number of parties - a "threshold" number - need to cooperate in order to decrypt a ciphertext. This prevents the situation where an individual keyholder is able to decrypt all sensitive information on his own. This aspect will certainly be addressed by future work.

Declarations

Ethical Approval

Not applicable

Competing interests

I declare that the authors have no competing interests as defined by Springer, or other interests that might be perceived to influence the results and/or discussion reported in this paper.

Authors' contributions

T.V.T Doan did the experimentation, wrote the code, and the main manuscript text. M.-L Messai wrote parts of the manuscript and reviewed it. G. Gavin and J. Darmont reviewed the manuscript.

Funding

The research depicted in this paper is funded by the French National Research Agency (ANR), project ANR-19-CE23-0005 BI4people (Business Intelligence for the people).

Availability of data and materials

All of the material is owned by the authors and/or no permissions are required.

References

[1] Abbas Acar et al. "A survey on homomorphic encryption schemes: Theory and implementation". In: *ACM Computing Surveys (Csur)* 51.4 (2018), pp. 1–35.

[2] Whitfield Diffie and Martin E Hellman. "New directions in cryptography". In: *Secure communications and asymmetric cryptosystems*. Routledge, 2019, pp. 143–180.

[3] Taher ElGamal. "A public key cryptosystem and a signature scheme based on discrete logarithms". In: *IEEE transactions on information theory* 31.4 (1985), pp. 469–472.

[4] Pascal Paillier. "Public-key cryptosystems based on composite degree residuosity classes". In: *International conference on the theory and applications of cryptographic techniques*. Springer. 1999, pp. 223–238.

[5] Craig Gentry. *A fully homomorphic encryption scheme*. Stanford university, 2009.

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) fully homomorphic encryption without bootstrapping". In: *ACM Transactions on Computation Theory (TOCT)* 6.3 (2014), pp. 1–36.

[7] Junfeng Fan and Frederik Vercauteren. "Somewhat practical fully homomorphic encryption". In: *Cryptology ePrint Archive* (2012).

[8] Jung Hee Cheon et al. "Homomorphic encryption for arithmetic of approximate numbers". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2017, pp. 409–437.

[9] Léo Ducas and Daniele Micciancio. "FHEW: bootstrapping homomorphic encryption in less than a second". In: *Advances in Cryptology–EUROCRYPT 2015: 34th Annual International Conference on the The-*

ory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I 34. Springer. 2015, pp. 617–640.

[10] Ilaria Chillotti et al. "TFHE: fast fully homomorphic encryption over the torus". In: *Journal of Cryptology* 33.1 (2020), pp. 34–91.

[11] Craig Gentry, Amit Sahai, and Brent Waters. "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based". In: *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*. Springer. 2013, pp. 75–92.

[12] Daniele Micciancio and Yuriy Polyakov. "Bootstrapping in FHEW-like cryptosystems". In: *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 2021, pp. 17–28.

[13] Nicolas Gama et al. "Structural lattice reduction: generalized worst-case to average-case reductions and homomorphic cryptosystems". In: *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer. 2016, pp. 528–558.

[14] Jacob Alperin-Sheriff and Chris Peikert. "Faster bootstrapping with polynomial error". In: *Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I 34*. Springer. 2014, pp. 297–314.

[15] Ronald L Rivest, Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[16] Payal V. Parmar et al. "Survey of various homomorphic encryption algorithms and schemes." In: *International Journal of Computer Applications 91* (2014).

[17] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. "Evaluating 2-DNF formulas on ciphertexts". In: *Theory of cryptography conference*. Springer. 2005, pp. 325–341.

[18] Subir Halder and Mauro Conti. "Crypsh: A novel iot data protection scheme based on bgn cryptosystem". In: *IEEE Transactions on Cloud Computing* (2021).

[19] Leo Ramón Nathan De Castro. "Practical homomorphic encryption implementations & applications". PhD thesis. Massachusetts Institute of Technology, 2020.

[20] Oliver Masters et al. "Towards a homomorphic machine learning big data pipeline for the financial services sector". In: *Cryptology ePrint Archive* (2019).

[21] Ilaria Chillotti et al. "A homomorphic LWE based E-voting scheme". In: *Post-Quantum Cryptography: 7th International Workshop, PQCrypto 2016, Fukuoka, Japan, February 24-26, 2016, Proceedings 7*. Springer. 2016, pp. 245–265.

[22] Paulo Martins, Leonel Sousa, and Artur Mariano. “A survey on fully homomorphic encryption: An engineering perspective”. In: *ACM Computing Surveys (CSUR)* 50.6 (2017), pp. 1–33.
[23] Andrey Kim, Yuriy Polyakov, and Vincent Zucca. “Revisiting homomorphic encryption schemes for finite fields”. In: *International Conference on the Theory and Application of Cryptology and Information Security.* Springer. 2021, pp. 608–639.
[24] Tancrede Lepoint and Michael Naehrig. “A comparison of the homomorphic encryption schemes FV and YASHE”. In: *International Conference on Cryptology in Africa.* Springer. 2014, pp. 318–335.
[25] Bechir Alaya, Lamri Laouamer, and Nihel Msilini. “Homomorphic encryption systems statement: Trends and challenges”. In: *Computer Science Review* 36 (2020), p. 100235.
[26] Vasily Sidorov, Ethan Yi Fan Wei, and Wee Keong Ng. “Comprehensive Performance Analysis of Homomorphic Cryptosystems for Practical Data Processing”. In: *arXiv preprint arXiv:2202.02960* (2022).
[27] Vincent Migliore, Guillaume Bonnoron, and Caroline Fontaine. “Determination and exploration of practical parameters for the latest Somewhat Homomorphic Encryption (SHE) Schemes”. In: (2016).
[28] Kim Laine. *Simple encrypted arithmetic library 2.3.1.* `https://www.microsoft.com/en-us/research/uploads/prod/2017/11/sealmanual-2-3-1.pdf`. 2017.
[29] Yuriy Polyakov et al. “Palisade lattice cryptography library user manual”. In: *Cybersecurity Research Center, New Jersey Institute of Technology (NJIT), Tech. Rep* (2022).
[30] Shai Halevi and Victor Shoup. “Algorithms in helib”. In: *Annual Cryptology Conference.* Springer. 2014, pp. 554–571.
[31] Jung Hee Cheon et al. *Implementation of HEAAN.* `https://github.com/snucrypto/HEAAN`. 2021.
[32] Caroline Fontaine and Fabien Galand. “A survey of homomorphic encryption for nonspecialists”. In: *EURASIP Journal on Information Security* 2007 (2007), pp. 1–10.
[33] Christiana Zaraket et al. “Cloud based private data analytic using secure computation over encrypted data”. In: *Journal of King Saud University-Computer and Information Sciences* (2021).
[34] Saja J Mohammed and Dujan B Taha. “Performance Evaluation of RSA, ElGamal, and Paillier Partial Homomorphic Encryption Algorithms”. In: *2022 International Conference on Computer Science and Software Engineering (CSASE).* IEEE. 2022, pp. 89–94.
[35] HElib v2.2.1. `https://github.com/homenc/HElib`. IBM, 2020.
[36] Shai Halevi and Victor Shoup. “Faster homomorphic linear transformations in HElib”. In: *Annual International Cryptology Conference.* Springer. 2018, pp. 93–120.
[37] PALISADE v1.10.6. `https://gitlab.com/palisade/palisade-release`. PALISADE Project, Dec. 2020.

[38] Ahmad Al Badawi et al. "OpenFHE: Open-source fully homomorphic encryption library". In: *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography.* 2022, pp. 53–63.

[39] SEAL (release 4.0). `https://github.com/microsoft/SEAL`. Microsoft Research, Redmond, WA, Apr. 2020.

[40] SEAL - Python. `https://github.com/Huelse/SEAL-Python`. Microsoft SEAL 4.X For Python, May. 2022.

[41] pybind11. `https://github.com/pybind/pybind11`. 2021.

[42] Peter L Montgomery. "A survey of modern integer factorization algorithms". In: *CWI quarterly* 7.4 (1994), pp. 337–366.

[43] Zvika Brakerski. "Fully homomorphic encryption without modulus switching from classical GapSVP". In: *Annual Cryptology Conference.* Springer. 2012, pp. 868–886.

[44] Oded Regev. "On lattices, learning with errors, random linear codes, and cryptography". In: *Journal of the ACM (JACM)* 56.6 (2009), pp. 1–40.

[45] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On ideal lattices and learning with errors over rings". In: *Annual international conference on the theory and applications of cryptographic techniques.* Springer. 2010, pp. 1–23.

[46] Zvika Brakerski and Vinod Vaikuntanathan. "Efficient fully homomorphic encryption from (standard) LWE". In: *SIAM Journal on computing* 43.2 (2014), pp. 831–871.

[47] VF Rocha, Julio López, and V Falcão Da Rocha. *An Overview on Homomorphic Encryption Algorithms.* 2019.

[48] VF Rocha, Julio López, and V Falcão Da Rocha. "An Overview on Homomorphic Encryption Algorithms". In: *UNICAMP Universidade Estadual de Campinas, Tech. Rep* (2018).

[49] Wei Yuan and Han Gao. "An Efficient BGV-type Encryption Scheme for IoT Systems". In: *Applied Sciences* 10.17 (2020), p. 5732.

[50] Song Yongsoo. "Introduction to CKKS". In: Private AI Boot-camp, Microsoft Research, 2019.

[51] Yongwoo Lee et al. "Efficient FHEW bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption". In: *Cryptology ePrint Archive* (2022).

[52] Martin Albrecht et al. "Homomorphic encryption standard". In: *Protecting privacy through homomorphic encryption* (2021), pp. 31–62.

[53] Thi Van Thao Doan. *Implementation of PHE schemes: El-Gamal, Paillier and RSA.* `https://github.com/ThaoDoanVan/PHE`. May 2022.

[54] James Heather et al. "Solving the Discrete Logarithm Problem for Packing Candidate Preferences". In: *International Conference on Availability, Reliability, and Security.* Springer. 2013, pp. 209–221.

[55] Stephen Pohlig and Martin Hellman. "An improved algorithm for computing logarithms over GF (p) and its cryptographic significance (corresp.)" In: *IEEE Transactions on information Theory* 24.1 (1978), pp. 106–110.

[56] Mihir Bellare et al. “Relations among notions of security for public-key encryption schemes”. In: *Annual International Cryptology Conference*. Springer. 1998, pp. 26–45.

[57] Massimo Chenal and Qiang Tang. “On key recovery attacks against existing somewhat homomorphic encryption schemes”. In: *International Conference on Cryptology and Information Security in Latin America*. Springer. 2014, pp. 239–258.

[58] Prastudy Fauzi, Martha Norberg Hovd, and Håvard Raddum. “On the IND-CCA1 Security of FHE Schemes”. In: *Cryptography* 6.1 (2022), p. 13.

[59] Zhiniang Peng. “Danger of using fully homomorphic encryption: A look at Microsoft SEAL”. In: *arXiv preprint arXiv:1906.07127* (2019).

[60] Baiyu Li and Daniele Micciancio. “On the security of homomorphic encryption on approximate numbers”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2021, pp. 648–677.

[61] Ying Guo, Zhenfu Cao, and Xiaolei Dong. “A Generalization of Paillier’s Public-Key System With Fast Decryption”. In: *Cryptology ePrint Archive* (2020).

[62] Frederik Armknecht, Stefan Katzenbeisser, and Andreas Peter. “Group homomorphic encryption: characterizations, impossibility results, and applications”. In: *Designs, codes and cryptography* 67.2 (2013), pp. 209–232.

[63] Yiannis Tsiounis and Moti Yung. “On the security of ElGamal based encryption”. In: *International Workshop on Public Key Cryptography*. Springer. 1998, pp. 117–134.

[64] Jiang Wu and Douglas R Stinson. “On the security of the ElGamal encryption scheme and Damgard’s variant”. In: *Cryptology ePrint Archive* (2008).

[65] John M Pollard. “Theorems on factorization and primality testing”. In: *Mathematical Proceedings of the Cambridge Philosophical Society*. Vol. 76. 3. Cambridge University Press. 1974, pp. 521–528.

[66] Thi Van Thao Doan, Thi Mai Phuong Nguyen, and Danh Nam Tran. *Simple Methods for Factorization*. `https://github.com/ThaoDoanVan/Factorization`. Project report. Sciences and Technologies Faculty, University of Limoges, Jan. 2022.

[67] Loria. *Record factors found by Pollard’s $p-1$ method*. `https://members.loria.fr/PZimmermann/records/Pminus1.html`. 2021.

[68] Peter W Shor. “Algorithms for quantum computation: discrete logarithms and factoring”. In: *Proceedings 35th annual symposium on foundations of computer science*. Ieee. 1994, pp. 124–134.

[69] Peter W Shor. “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer”. In: *SIAM review* 41.2 (1999), pp. 303–332.

[70] Michael R Geller and Zhongyuan Zhou. “Factoring 51 and 85 with 8 qubits”. In: *Scientific reports* 3.1 (2013), pp. 1–5.

[71] Lieven MK Vandersypen et al. "Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance". In: *Nature* 414.6866 (2001), pp. 883–887.

[72] Enrique Martin-Lopez et al. "Experimental realization of Shor's quantum factoring algorithm using qubit recycling". In: *Nature photonics* 6.11 (2012), pp. 773–776.

[73] Craig Gidney and Martin Ekerå. "How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits". In: *Quantum* 5 (2021), p. 433.

[74] David Beckman et al. "Efficient networks for quantum factoring". In: *Physical Review A* 54.2 (1996), p. 1034.

[75] Vlatko Vedral, Adriano Barenco, and Artur Ekert. "Quantum networks for elementary arithmetic operations". In: *Physical Review A* 54.1 (1996), p. 147.

[76] Stephane Beauregard. "Circuit for Shor's algorithm using 2n+ 3 qubits". In: *arXiv preprint quant-ph/0205095* (2002).

[77] Yasuhiro Takahashi and Noboru Kunihiro. "A quantum circuit for Shor's factoring algorithm using 2n+ 2 qubits". In: *Quantum Information & Computation* 6.2 (2006), pp. 184–192.

[78] Thomas Häner, Martin Roetteler, and Krysta M Svore. "Factoring using 2n+ 2 qubits with Toffoli based modular multiplication". In: *arXiv preprint arXiv:1611.07995* (2016).

[79] Craig Gidney. "Factoring with n+ 2 clean qubits and n-1 dirty qubits". In: *arXiv preprint arXiv:1706.07884* (2017).

[80] Jingwen Suo et al. "Quantum algorithms for typical hard problems: a perspective of cryptanalysis". In: *Quantum Information Processing* 19.6 (2020), pp. 1–26.

[81] John Proos and Christof Zalka. "Shor's discrete logarithm quantum algorithm for elliptic curves". In: *arXiv preprint quant-ph/0301141* (2003).

[82] Martin Ekerå. "Revisiting Shor's quantum algorithm for computing general discrete logarithms". In: *arXiv preprint arXiv:1905.09084* (2019).

[83] Frédéric Wang. "The hidden subgroup problem". In: *arXiv preprint arXiv:1008.0010* (2010).