SN Computer Science (2024) 5:24
https://doi.org/10.1007/s42979-023-02316-9

REVIEW ARTICLE

Homomorphic Encryption Library, Framework, Toolkit and Accelerator: A Review

Shalini Dhiman[1] · Ganesh Kumar Mahato[1] · Swarnendu Kumar Chakraborty[1]

Received: 10 February 2023 / Accepted: 9 September 2023
© The Author(s), under exclusive licence to Springer Nature Singapore Pte Ltd 2023

Abstract

Homomorphic encryption ensures secure computation on encrypted data without the need for decryption beforehand. It enables the secure offloading of computations to untrusted servers. This paper provides a comprehensive description of multiple methods for conducting secure computations, along with the appropriate approaches and tools needed for successful implementation. Various strategies are outlined for selecting a suitable homomorphic encryption (HE) library, assisting developers and researchers in determining the most suitable library for their projects. To begin with, the paper presents a comparison of homomorphic encryption libraries based on different parameters. Additionally, it outlines the steps involved in choosing the right framework for effective implementation, comparing them based on various parameters. The Framework discussed in the paper supports a range of homomorphic encryption libraries and provides detailed information about them. Furthermore, the paper surveys three different homomorphic encryption accelerators and compares them to determine which one would maximize bootstrapping throughput when implemented. Lastly, the Fully Homomorphic Encryption-IBM toolkit is discussed. This toolkit supports the development of resources, primarily involved in the flow of messages. The paper concludes that secure computation is achievable by implementing the appropriate tools, considering their performance and implementation limitations. Moreover, the selection of the appropriate library, framework, and accelerator depends on the specific demands and requirements of the implementation chosen by the developer.

Keywords Homomorphic encryption · MS SEAL · Pallisade · IBM toolkit · Evervault encryption engine · HEaaN

Introduction

Our personal information is being shared more extensively than ever before, and often, we are the ones who willingly share it. As long as our personal information remains uncompromised, we don't typically mind. However, in this new era of the digital world, sharing personal data is essential for interacting with others and accessing everyday services. Consequently, it is crucial to be cautious and certain about the data we share, ensuring its security. But how can we be sure that our data is safe? The answer is relatively simple: most of the personal data we share is encrypted. Encrypted data is useless to attackers or hackers because it is concealed behind complex codes that are beyond human comprehension.

The problem arises when the encrypted data is transferred or stored, requiring decryption, which provides attackers with an opportunity to target our shared data. To address this issue, homomorphic encryption (HE) schemes [1] have been introduced, revolutionizing the concept of security.

Ganesh Kumar Mahato and Swarnendu Kumar Chakraborty authors contributed equally to this work.

This article is part of the topical collection "Research Trends in Communication and Network Technologies" guest edited by Anshul Verma, Pradeepika Verma and Kiran Kumar Pattanaik.

✉ Swarnendu Kumar Chakraborty
swarnendu@nitap.ac.in

Shalini Dhiman
shalini1695@gmail.com

Ganesh Kumar Mahato
mahato.ganesh88@gmail.com

[1] Department of Computer Science and Engineering, National Institute of Technology, Jote 791113, Arunachal Pradesh, India

Published online: 20 November 2023

Homomorphic encryption enables computations to be performed on encrypted data without the need for decryption. This technique instills confidence when sharing sensitive information. Its development dates back to 2009 [2], and it has since played a crucial role as a game changer in digital security measures.

Homomorphic encryption will find extensive applications across various industries, including finance [3], hotels, airlines, education systems, and restaurants. It offers access to personal data without compromising user privacy or account holder confidentiality [4].

Now that we understand that HE is an ideal solution to our concerns, let's briefly discuss the different types of HE [5]. Firstly, Partially Homomorphic Encryption (PHE) allows a single operation to be performed on the ciphertext an infinite number of times. This operation can be either addition or multiplication and is particularly useful in applications that involve only one arithmetic operation. PHE is relatively simple to implement. The second method is Somewhat Homomorphic Encryption (SHE), which permits both addition and multiplication, but only up to a certain limit. The circuit logic imposes a specific depth of evaluation on this restriction. Lastly, Fully Homomorphic Encryption (FHE) enables arbitrary computations on encrypted data, allowing unlimited additions and multiplications to be performed on the ciphertext indefinitely.

As we now have an understanding of various schemes in homomorphic encryption (HE) that can safeguard our data, the focus shifts to implementing these plans effectively. Our research primarily revolves around the implementation of HE schemes using libraries, frameworks, toolkits, and accelerators. We aim to address common questions that users or developers may have, such as selecting the appropriate HE library for their specific implementation needs. Additionally, we explore the ideal framework capable of supporting all necessary operations. Furthermore, we delve into the role of the HE toolkit in facilitating implementation. Lastly, we examine accelerators to identify the most performant option for user algorithms. Extensive reviews and studies have been conducted on HE libraries, frameworks, toolkits, and accelerators, all of which are covered comprehensively in our paper. Each topic is briefly discussed in terms of functionality, operations, procedures, limitations, and advantages, and a comparative analysis is conducted using various parameters.

The key contributions of this paper can be summarized as follows:

1. We provide a brief overview of homomorphic encryption libraries, guiding users and developers on how to choose the most suitable library based on their implementation preferences. We compare different libraries using tabulated parameters to aid readers in making informed decisions for their own implementations.
2. The paper discusses various homomorphic encryption frameworks, focusing on selecting an appropriate framework based on the desired operations. We outline important criteria for framework selection and compare them based on different parameters, enabling readers to choose the ideal HE framework for their work.
3. The importance of accelerators in HE schemes is emphasized, with the paper presenting different accelerators and their architectures. We explain why accelerators are necessary and provide guidance on selecting the most appropriate option. The accelerators are compared based on multiple parameters, facilitating quicker decision-making for users.
4. The paper concludes with a discussion on the HE toolkit, which supports the development of resources involved in message flow. It outlines the integration of the toolkit with integrated nodes for seamless deployment of message flow.

The paper's organization is as follows: The introduction section provides an overview of the paper and its contributions. The second section focuses on homomorphic encryption libraries, comprising three parts: library selection, an overview of multiple libraries, and a comparison of libraries based on various parameters. Section "Framework" discusses frameworks used to conceal the FHE library API from programmers. This section is divided into three parts: a general framework concept, various frameworks available for implementation, and a tabulated comparison of frameworks based on different parameters. Section "FHE-IBM encryption toolkit" explores the FHE-IBM encryption toolkit, covering installation requirements, getting started with IBM, perspectives on the FHE-IBM encryption toolkit, and maximizing the use of the IBM Integration Toolkit. Section "Cingulata" delves into HE accelerators, explaining their purpose and examining different options. A comparison of accelerators based on various parameters is provided. Finally, Sect. "Accelerator" concludes the paper.

Homomorphic Encryption Library

Selecting a suitable homomorphic encryption library is a relatively straightforward process as long as you have a clear understanding of your motivations and the nature of your work. Here are some criteria that users should always consider when choosing an appropriate homomorphic encryption library [6].

- **Availability of Information**

It is essential for users to gather information about the functioning, characteristics, accessibility, and compatibility of a homomorphic encryption library before making a selection.

- **Community Size** The size of the community refers to the number of individuals actively engaged with the library. A larger community can be beneficial as it provides a greater pool of expertise to assist in addressing encountered bugs and resolving users' queries.
- **Community Engagement** Another important factor to consider is the level of community engagement. This includes the extent to which the library interacts with its users, such as organizing webinars or workshops to enhance user understanding. It is crucial for users to stay informed about upcoming features of the library that may be relevant for their future work. Being aware of these developments ensures users can make informed decisions and leverage the latest capabilities of the library.
- **Ease of Use** This criterion can be considered the most ambiguous among all. Users must have a clear understanding of the library's design and architecture. Additionally, users should take note of their proficiency in the programming language used by the library and thoroughly familiarize themselves with the working procedures of the library. Having a strong grasp of these aspects is crucial for effective utilization of the library.
- **Open-Source** The next criterion to consider is whether the library is open-source software. Open-source libraries provide transparency by making their source code accessible to users. It is generally recommended that users prefer an open-source library as it allows for greater visibility and understanding of the library's inner workings.
- **Standards Compliant** This final criterion focuses on the security properties and parameters of the library. It is important to ensure that the library's security schemes are clearly explained by standardization organizations. Compliance with relevant security standards is crucial to ensure the algorithm employed by the library is secure. Users should prioritize libraries that adhere to recognized security standards to safeguard their sensitive data and ensure the overall security of their system.

Let us go over some of the homomorphic encryption libraries and discuss them briefly.

Microsoft Simple Encrypted Arithmetic Library (SEAL)

Microsoft SEAL is a well-known open-source software library developed by Microsoft for implementing various forms of homomorphic encryption. It supports both asymmetric and symmetric encryption algorithms and enables specific parts of programs to be executed on encrypted data in cloud computations. However, this technology is limited to operations such as addition and subtraction on encrypted real numbers and integers, and it may not be suitable for complex operations like regular expressions, encrypted comparison, or sorting. The Microsoft SEAL library offers two homomorphic encryption schemes: the Brakerski-Gentry-Vaikuntanathan (BGV) scheme and the Brakerski/Fan-Vercauteren (BFV) scheme. These schemes are commonly used when precise values are required for computations on encrypted integers or real numbers using exact modular arithmetic [7]. In homomorphic encryption, data compression can be necessary to reduce the size of the ciphertext object, which typically consists of a large number of integer modulo values. This compression helps in reducing storage costs, improving storage capacity, and speeding up data transfer rates. Microsoft SEAL utilizes compression libraries like ZLIB and Zstandard as options to compress serialized data. Among the two, Zstandard is used by default due to its superior performance. Since ciphertext objects are composed of integers, real numbers, and prime numbers modulo, compression plays a significant role in optimizing their storage [8]. When using schemes like CKKS, the numbers involved may be relatively small, such as 30 bits or less. In such cases, the data is not serialized into 64-bit integers, resulting in wasted space where half of the ciphertext bytes are zero. This issue can be addressed by employing compression algorithms, eliminating the unused space. Compressed serialization can be applied to both ciphertext and keys [9]. Additionally, Microsoft SEAL makes use of the Microsoft GSL (Guidelines Support Library), which is a header-only library. It implements gsl::span, allowing for bound-checked array access to memory. This integration helps in enhancing performance and enables the Batch Encoder and CKKS Encoder to efficiently encode and decode data.

Overall, Microsoft SEAL provides a comprehensive set of tools and functionalities for implementing homomorphic encryption, making it a popular choice for various applications and scenarios [10].

Microsoft SEAL is indeed a complex library, and there are certain aspects that users should focus on to effectively utilize its capabilities. Firstly, it is crucial for users to have a strong understanding of the concepts related to homomorphic encryption. This technology has a steep learning curve, but acquiring proficiency in its specific concepts will greatly assist users in working with Microsoft SEAL and leveraging its functionality.

Secondly, users should grasp the concept of efficient and inefficient implementations when it comes to running or programming specific computations using Microsoft SEAL. Since homomorphic encryption involves performing computations on encrypted data, the efficiency of these computations is essential. Users should strive to optimize

their implementations to ensure efficient execution of computations, as inefficient implementations can result in slower processing times and increased resource consumption.

By having a solid grasp of homomorphic encryption concepts and paying attention to efficient implementation strategies, users can make the most of Microsoft SEAL and effectively utilize its capabilities in their applications and computations.

Homomorphic Encryption for Arithmetic of Approximate Numbers (Pi-HEaaN)

Pi-HEaaN is primarily a Python library that serves as a simulation of Pi-HEaaN, offering approximate computations on homomorphic encryption. This technology allows for the implementation of various operations, including homomorphic addition, homomorphic subtraction, and homomorphic multiplication. Additionally, it provides essential functionalities like key generation, encryption, and decryption.

The operations performed by Pi-HEaaN are conducted on block units, enabling practical solutions for scenarios involving significant amounts of data and data handling, typically encountered in cloud computing environments. This scheme facilitates the encryption of substantial data volumes within a single ciphertext, enabling parallelization in both computation and storage. By utilizing Pi-HEaaN, users can effectively handle large-scale data encryption while maintaining computational efficiency and optimal resource utilization [11].

Homomorphic encryption using Pi-HEaaN is widely recognized and preferred by homomorphic encryption (HE) engineers due to its effectiveness in achieving smooth evaluation. This evaluation method allows engineers to assess the complexity and effectiveness of algorithms, which is a crucial step in evaluating and implementing HE algorithms.

During the evaluation process, it is common to encounter a small number of errors that arise from the approximation operations employed. These errors are acceptable and even beneficial in certain cases as they contribute to reducing time complexity and improving overall efficiency. Therefore, Pi-HEaaN is particularly well-suited for scenarios where approximation operations are preferred over exact operations.

By leveraging Pi-HEaaN for homomorphic encryption, engineers can conduct evaluations that strike a balance between accuracy and efficiency, ultimately leading to more practical and effective implementations of HE algorithms.

In Pi-HEaaN, as it supports approximate results, a rescaling procedure is employed to handle the increasing magnitude of plaintext. This procedure becomes necessary when dealing with problems where the bit size of a message rapidly grows with the depth of a circuit.

The rescaling procedure involves compressing a ciphertext into a smaller modulus, resulting in an approximation of the plaintext value. This compression introduces noise, which acts as an error but actually contains important information. The purpose of adding this noise is to deceive potential attackers. While the attacker may perceive it as an error, the noise is actually incorporated into the plaintext for security reasons. The rescaling procedure is then used to reduce this noise and arrive at an approximation of the original plaintext value, leading to decryption with approximated values.

By employing the rescaling procedure, Pi-HEaaN ensures a balance between security and efficiency by introducing noise that serves as a security measure, while still enabling meaningful decryption of approximated values. There are several techniques that can be employed to enhance the homomorphic evaluation of logistic functions. Some of these techniques include [12]:

1. **Use of Some Batching Techniques**
   These techniques are commonly utilized in prediction analysis to reduce the evaluation time of logistic functions without the need for parallelization. By applying these optimization methods, the evaluation time has been significantly improved to a total of 0.54 s. This reduction in evaluation time allows for more efficient and faster processing of logistic functions, making them suitable for real-time prediction and analysis tasks.
2. **Use a Fast Fourier Transform (FFT) Algorithm**
   This technique utilizes the encoding method of unity in the polynomial ring to minimize the consumption of ciphertext levels during evaluation. Simultaneously, a rescaling procedure is applied to operations using a batching technique and Hadamard space. These optimizations lead to a smaller parameter size and improved evaluation time.
   In standard processing using the FFT-Hadamard method, the evaluated results are obtained in 0.34 s per slot on a machine with four cores and six processors, when no batching technique is employed. This evaluation process involves precise multiplication of integral polynomials and eliminates fractional parts that are close but not exact. By leveraging these techniques, more efficient and accurate evaluations can be achieved in prediction analysis tasks.
3. **Use Of Homomorphic Encryption**
   This can be applied to get an exact computations when the result has a specific kind of format or property for approximate arithmetic computations.

This is a kind of open-source implementation of the homomorphic encryption library (HEaaN), and this algorithm

mostly copes with the C++ language. Use of this can give an approximate result and it supports huge data handling.

Fast Fully Homomorphic Encryption Over the Torus (TFHE)

The TFHE scheme was introduced to address some of the limitations encountered in the simpler fully homomorphic encryption (FHE) schemes [13]. The original FHE schemes, such as GSW (Craig Gentry, Amit Sahai, and Brent Waters) and its ring variants, exhibited slower noise growth but faced challenges with bootstrapping time during evaluation. TFHE was designed to overcome these issues by proposing a mechanism that reduces bootstrapping time [14]. TFHE achieves this by refreshing the ciphertexts after each individual operation. This approach helps maintain efficient evaluation while preserving the security parameters. The implementation of TFHE involves utilizing a high-speed gate-by-gate bootstrapping technique through a C/C++ library. This library enables the evaluation of arbitrary boolean circuits consisting of binary gates while ensuring the confidentiality of the processed data, thereby keeping the information secret. There are a few advantages to using this library [15], such as

1. TFHE stands out among other libraries as it does not impose any restrictions on the number of gates or computations that can be performed. It supports the use of any number of gates and their compositions, allowing for versatile and unrestricted evaluations.
2. The implementation of circuits using TFHE can be done manually or through automated generation, and the library is capable of executing any type of evaluation flawlessly in both cases. It does not require prior knowledge of the specific computation or function being performed, making it flexible for a wide range of computations over encrypted data [16].
3. The TFHE library enables the generation of a secret key, which provides encryption and decryption capabilities. Additionally, a cloud keyset is generated, which can be exported for performing homomorphic computations in the cloud.
4. When it comes to evaluation speed, the TFHE library can achieve a high rate of approximately 76 gates per second per core. This efficiency contributes to faster computations and enhances the overall performance of homomorphic evaluations using TFHE.

This library is capable of keeping any information authenticated and can be implemented in a fast way.

Homomorphic Encryption Library (HElib)

HElib is an open-source C++ library that utilizes ciphertext packing techniques. It is implemented using C++ 17 and relies on the NTL mathematical library for high-performance mathematical computations. NTL is a portable library that complements the functionality of the C++ library in HElib.

HElib supports various operations, including addition/subtraction, multiplication, set operations, and shifts. It provides a convenient interface for programmers to work with high-level languages and compile their code using the operations offered by HElib.

HElib offers flexibility in its configuration by providing different schemes and parameters. These parameters can impact the runtime and security of the scheme. To assist programmers in making informed decisions, HElib provides timing and memory metrics for each test. This enables programmers to evaluate and compare different parameter settings, ensuring optimal performance and maintaining consistency in their applications [16]. By analyzing these metrics and experimenting with different parameters, programmers can arrive at the most suitable configuration for their specific use cases, balancing runtime efficiency and security requirements.

The HElib uses the following schemes [17]:

1. HElib supports various schemes for homomorphic encryption, including the Brakerski-Gentry-Vaikuntanathan (BGV) scheme and the CKKS (Cheon-Kim-Kim-Song) scheme, among others. The BGV scheme is based on the concept of Learning with Errors over Rings (RLWE). It is a commonly used scheme in HElib. In this scheme, each homomorphic operation introduces a small error to the ciphertext, and these errors accumulate over multiple operations.
2. The CKKS scheme, on the other hand, is designed for approximate computation with real numbers. It provides an approximate result rather than an exact result. The CKKS scheme involves the generation of secret keys for decryption and public keys for encryption.
3. HElib also incorporates optimizations and techniques for better performance, such as the Gentry-Halevi-Smart optimizations and Smart-Vercauteren ciphertext packing techniques. These techniques aim to enhance the efficiency and functionality of the library, allowing for more efficient homomorphic computations. Overall, HElib provides a range of schemes and techniques that users can choose from based on their specific requirements and use cases, enabling them to achieve better performance and security in their homomorphic encryption applications.

HElib has improved a lot in recent years in terms of following performances [18]:

- **Reliability** HElib offers an analytical solution to address any challenges faced by programmers, thereby ensuring the reliability of the library. It is designed to work effectively according to the preferences of the programmer. Reliability is further maintained through the implementation of a key management scheme and two-factor authentication, which contribute to the overall security and trustworthiness of the system.
- **Robustness** HElib ensures reliability by implementing measures to prevent the leakage of ciphertext shared between two users. It maintains the robustness of the system by restricting third parties from accessing the shared keys and thereby ensuring that the confidentiality of the encryption is maintained.
- **Serviceability** HElib offers periodic serviceability to its users, including preventive maintenance and ease of programming. Whenever updates are implemented, it ensures that the system maintains optimal speed. This allows programmers to address and resolve any issues that arise by utilizing the updated features and improvements.
- **Performance** HElib focuses on optimizing the performance of addition, subtraction, and multiplication operations. It ensures that these operations run efficiently and smoothly for programmers, maintaining a good level of performance.

FV-NFLlib

FV-NFLlib is a software library for homomorphic encryption (HE) that operates using an asymmetric encryption method. It specifically supports homomorphic addition and homomorphic subtraction operations. The library is primarily implemented in the C++ programming language. The output of computations using FV-NFLlib is approximate rather than exact, and it includes functionality to verify the correctness of the results [19]. FV-NFLlib is based on the principles of ideal lattice cryptography. It undergoes several test conditions to ensure its functionality and reliability. The implementation of FV-NFLlib utilizes the Fan-Vercauteren scheme. The library is tested using the Test_binary_tree, Test_ec_additions, and Test_encrypt_poly procedures, which evaluate different aspects of its performance and functionality [20].

1. **Test_binary_tree**
   Test_binary_tree is a kind of program which keeps the reference of the key generation and the procedures which it executes, such as homomorphic addition and homomorphic multiplication procedures. The test binary tree multiplication is executed and devaluated to ensure about how much the program is correct and to set the final bound.
2. **Test_ec_additions**
   Test_ec_additions is basically a code of elliptical curve addition which is templated, and it is majorly called twice, i.e., first with FV::mess_t and secondly with FV::ciphertext_t. This program computes homomorphically and clearly upon elliptic curve addition. This is performed over the NIST P-256 curve.
3. **Tests/Test_encrypt_poly**
   This test is performed to check the correctness of the program. The evaluations are performed as a small sage program. Its computation is done over a product homomorphically. The evaluation contains two polynomials, which are encrypted.

   One distinguishing characteristic of FV-NFLlib is its larger noise distribution compared to other libraries. It incorporates an asymptotic multiplicative factor by multiplying two asymptotic factors term by term. FV-NFLlib modifies the existing codes and continuously monitors the security parameters to minimize the impact of noise. In terms of implementation, FV-NFLlib utilizes optimized Number Theoretic Transform (NTT) transforms. These transforms specifically require power-of-two cyclotomic polynomials, which play a crucial role in the underlying mathematical operations performed by the library.

Palisade

Palisade was developed to provide implementations based on lattice cryptography building blocks, incorporating various state-of-the-art homomorphic encryption schemes. It offers modularity through simple APIs, allowing for easy integration with hardware accelerators. Palisade aims to meet the security standards for homomorphic encryption while enabling the use of this encryption technique.

One of the key features of Palisade is its support for multiple homomorphic encryption schemes, including BGV, BFV, CKKS, and FHEW. These schemes offer different levels of security, with TFHE being particularly notable for its inclusion of bootstrapping. Palisade also provides post-quantum public-key encryption, ensuring security against attacks from quantum computers.

Proxy re-encryption is another capability offered by Palisade, allowing for the transformation of ciphertext from one public key to another without revealing any private information about the original message. The library supports identity-based encryption, where users generate a public key and a trusted server derives the corresponding private key based on a unique identifier. Attribute-based encryption is

also supported, where the ciphertext and secret key depend on specified attributes.

Additionally, Palisade includes support for digital signatures, enabling the generation of a unique hash or encryption of messages and documents using the sender's private key. These features provide a comprehensive set of cryptographic functionalities within the Palisade library [21].

There are several git repositories which a user or a programmer can explore, such as Encrypted Circuit Emulator, Integer Examples, Python Demos, Python Demos Serialization Examples, Graphene-PALISADE-SGX [22]:

- The Encrypted Circuit Emulator exemplifies how the Palisade can be utilized to execute circuits described in different formats to implement FHEW (achieving bootstrapping in under a second) and TFHE.
- Integer Example:This showcases how the Palisade is utilized in the implementation of the BFV encryption scheme to encrypt searches for substrings.
- Python Demos: Palisade comes with a Python wrapper and explains how to write Python 3 to make the most of it.
- Serialization Examples: This shows how Palisade serialize the cryptography applications and cooperating processes and shows their correlation. This shows the cooperation between heavy-weight processes and other cryptographic applications.
- Graphene-PALISADE-SGX: This works best when it runs on Ubuntu because it provides all the tools through which it becomes very compatible. The SGX port becomes necessary with Graphene, but it is not necessary for the Graphene/Gramine portion to be combined with SGX.

This library supports many operations and fulfils most of the users' choices. This library is known to be the most used library because of its various functionality support and its git repositories.

Concrete

The concrete library is a well-known open-source library that is widely recognized for its functionality. It serves as a user-friendly interface for integrating Fully Homomorphic Encryption (FHE) into applications. One of the main advantages of this library is its ease of integration, particularly due to its user-friendly interface.

The library offers an extensive range of operations, making it highly versatile when working with arbitrary input formats. Developed in Rust, it naturally implements the TFHE cryptosystem and provides convenient addition and multiplication operations that can be compiled for 32 or 64-bit systems using unsigned integers. The library also supports encoding functions, allowing for implementation with real numbers and computations on Boolean values through TFHE by encrypting and bootstrapping.

The concrete library consists of two layers, namely the Core API and the Crypto API, which are structured as a stack. The Core API is a lower-level layer focused on maintaining the library's efficiency. Accessible only to FHE experts, it ensures that computations are performed efficiently. On the other hand, the Crypto API is the second layer implemented above the Core API. It can be accessed by any programmer and offers a user-friendly communication interface. notably, the Crypto API keeps track of the noise generated during computations using automated metadata and decoding parameters. This metadata summarizes essential information and facilitates accurate tracking of computations and their correctness [23].

This library is primarily implemented using the TFHE scheme, which relies on the concept of Learning with Errors (LWE) for its learning methodology. The library utilizes previous errors to enhance its learning process. However, as the noise level increases, the decryption of homomorphic operations on ciphertexts can become limited. To address this limitation, it is necessary to periodically refresh noisy ciphertexts, which helps reduce the noise level. A crucial concept in the library is Programmable Bootstrapping (PBS). PBS is a general technique that enables the evaluation of any function on ciphertexts by refreshing the ciphertexts at regular intervals. PBS involves three fundamental algorithms: Blind Rotate, Switch Modulus, and Sample Extract. Blind Rotate is the first algorithm, which allows the rotation of ciphertexts without decrypting them. Switch Modulus is the second algorithm, used to change the modulus of the ciphertexts while preserving the encrypted values. Lastly, Sample Extract is the third algorithm, used to extract samples from noisy ciphertexts and refresh them to reduce noise accumulation. These three algorithms in PBS collectively contribute to the ability to evaluate functions on ciphertexts while periodically refreshing them to maintain computational accuracy and reduce the impact of noise.

The functionalities of these algorithms are discussed below:

1. **Blind Rotate**: The Blind Rotate algorithm begins with an accumulator and updates itself by utilizing control bits. Its purpose is to blindly select a specific coefficient in a constant term. This algorithm is employed to rotate the coefficient of the plaintext polynomial stored in an RLWE ciphertext in a homomorphic manner.
2. **Switch Modulus**: Switch Modulus is an optional operation that is utilized in conjunction with sample extraction and blind rotation. In LWE ciphertext encryption, a unique private key is necessary, distinct from the keys typically used for inputs. To reverse this process and

retrieve the original key, the key switching formula is employed.
3. **Sample Extract**:Sample Extract is an algorithm employed to construct an output with the desired coefficient, ensuring it is free from noise. This algorithm randomly selects coefficients from the provided input sequence. Typically categorized as a "public operation," it helps generate a noise-free output.

The concrete-core repository is used to implement the low-level cryptographic primitives. Such a repository is as follows:

- **Concrete**: This is mostly used by cryptographers who don't have any idea about the information regarding the details of implementations. One can implement the homomorphic application as fast as possible, competitively.
- **Concrete-Boolean**: This is what cryptographers use to implement boolean gates in a way that doesn't change how they work. It can be used to run any kind of circuit over encrypted data.
- **Concrete-Shortint**: Concrete-Shortint has the ability to implement operations on short integers. The short integers range from 1 to 4 bits.
- **Concrete-Integer**: Concrete-Integer has the ability to implement operations on short integers. The short integers range from 4 to 16 bits.

Lattigo

Lattigo is a Homomorphic Encryption (HE) library that utilizes the Go programming language. Its development began in 2019, and it is designed to be implemented as a single-threaded library with built-in concurrency control in its API. Lattigo focuses on the implementation of Ring-Learning-With-Errors (RLWE) homomorphic encryption primitives, as well as Multiparty Homomorphic Encryption, both of which rely on secure protocols [24]. This library has gained popularity due to several notable advantages. Firstly, it employs high-precision bootstrapping procedures for full-RNS CKKS (approximate arithmetic) scheme, utilizing both dense-key and sparse-key techniques. Secondly, Lattigo supports multiple encryption schemes, including full-RNS, BFV, and CKKS. Thirdly, it delivers performance comparable to leading C++ libraries in the field. Lastly, the Lattigo implementation includes support for WASM (WebAssembly) compilation, particularly beneficial for browser clients, and enables cross-platform builds. Lattigo is well-suited for implementing HE in distributed systems and microservices architectures. The choice of the Go programming language is preferred due to its concurrency model and portability, making it a suitable option for such applications [25].

The library offers some main packages using which we can perform our implementation. They can be seen as [26]:

1. Lattigo/Ring: Lattigo/Ring implements modular arithmetic operations that use RNS polynomials. The basis of RNS includes RNS rescaling, RNS basis extension, and number theoretic transform (NTT).
2. Lattigo /BFV: Lattigo /BFV can be implemented on Full-RNS, which is one of the variants of Brakerski-Fan-Vercauteren. The operations of modular arithmetic are performed on integers.
3. Lattigo /CKKS: Lattigo /CKKS implements Full-RNS Homomorphic Encryption only for approximate numbers, so it can operate over complex numbers as well as real numbers.
4. Lattigo/DBFV and Lattigo/DCKKS: Lattigo/DBFV and Lattigo/DCKKS are multiparty versions of the BFV and CKKS schemes that perform computations using shared secret keys.
5. Lattigo/examples: This directory contains examples of the Lattigo library. All the sub-packages and test files further use lattigo primitives.
6. Lattigo/Utils: This package is used to support functions and structures.

There are some important features in Lattigo, that will be further used in cryptographic research. They are as follows [27]:

- **Standalone Arithmetic Layer** In the Lattigo library, the polynomial arithmetic functionalities are mainly exposed through the Lattigo/Ring sub-package. Lattigo is implemented entirely in Go and relies on low-level optimized algorithms for its operations. It does not depend on external numerical libraries and minimally utilizes the unsafe package. The Lattigo/Ring sub-package includes various operations such as Montgomery-form arithmetic, Number-Theoretic Transform (NTT), evaluation of automorphisms, specific ring operations, sampling from Gaussian, uniform, and ternary distributions, as well as RNS (Residue Number System) base extensions and scaling. With these capabilities, Lattigo has the ability to construct and evaluate other homomorphic encryption (HE) schemes and their primitives based on Ring-Learning-With-Errors (R-LWE). By providing the necessary tools and operations, the library enables the development and execution of a wide range of HE schemes built upon the R-LWE foundation [28].
- **The Generalized Keyswitch Procedure** Lattigo employs a generalized key switching procedure that allows users to determine the norm of the decomposition basis during the key switching process. This functionality is applicable to both the BFV and CKKS

schemes. By identifying the norm, users can optimize the key switching operation and its impact on the capacity of homomorphic computations. During the evaluation of throughput, it is observed that while the run-time is reduced, there is also a loss in the homomorphic capacity of the system. In order to strike a balance and improve the overall performance, optimizing the key switching algorithm becomes necessary. This optimization is particularly beneficial for evaluating automorphisms, such as rotations, in a more efficient manner. By optimizing the keyswitch algorithm, Lattigo aims to enhance the performance and capabilities of its homomorphic encryption schemes
- **Novel BFV Quantization** In Lattigo, homomorphic multiplication is performed using Residue Number System (RNS) quantization techniques, which are specifically designed for the RNS variant of the CKKS scheme. In general, homomorphic multiplication in the BFV scheme is known to be computationally expensive because it requires a secondary and temporary basis. However, Lattigo tackles this challenge in a more efficient manner. Lattigo implements homomorphic multiplication in a way that minimizes the computational overhead. By leveraging RNS quantization techniques, Lattigo streamlines the multiplication operation, resulting in a more friendly and efficient execution. This optimization allows for improved performance and reduced computational costs when performing homomorphic multiplication within Lattigo's CKKS scheme.
- **CKKS Bootstrapping** Lattigo implements CKKS bootstrapping. It is known as the second library that has implemented a bootstrapping circuit using the CKKS scheme. The implementation was done on an open-source platform, and Lattigo made the implementation available for the full-RNS variant. This procedure was precise and more efficient compared to the state-of-the-art. Also, the use of any sparse secret key is not required.
- **Homomorphic Polynomial Evaluation** The Lattigo/CKKS package offers a critical set of functionalities, including an algorithm for depth-optimal polynomial evaluation and scale invariance. Within this package, there are key functions that provide clear-text polynomial coefficients and allow users to specify the desired output scale. To ensure precise rescaling procedures throughout the evaluation process, the package utilizes recursive backward propagation. This recursive approach guarantees the exactness of all rescaling operations, ensuring accuracy and reliability in the evaluation process.

Hence Lattigo supports the use of Go language primitives, making it easier to develop applications such as new HE and MHE applications. As a result, if we are able to reduce that, Lattigo can be used in both the adoption of HE in real systems and cryptography research.

FHE C++ Transpiler

FHE (Fully Homomorphic Encryption) plays a vital role in today's market, offering advanced security for sensitive data. However, one of its primary challenges lies in its performance limitations. FHE requires specific criteria, wherein programs are initially compiled using unencrypted data, while the actual FHE computations are carried out on encrypted data. This process poses certain obstacles.

To address these limitations, the FHE C++ Transpiler [71] has been introduced. This innovative solution boasts a modular architecture designed to seamlessly convert regular C++ code into FHE-compatible C++ code. Essentially, once this transpiler is implemented, the traditional workflow of working on unencrypted data and subsequently converting it into encrypted data becomes unnecessary.

With the integration of the FHE C++ Transpiler, the entire procedure becomes more streamlined and secure. There's no longer a need to handle unencrypted data during development. Instead, developers can confidently work with encrypted data, ensuring the confidentiality and integrity of their code throughout the process. This advancement significantly enhances the safety and efficiency of operating on encrypted data while utilizing the benefits of Fully Homomorphic Encryption.

We have compared all libraries to different parameters in tabulated form (Table 1) so that it becomes easy for readers to come to a conclusion regarding which library to use for their further implementation.

Framework

The framework for Homomorphic Encryption (HE) is designed with the intention of concealing the underlying API of the Fully Homomorphic Encryption (FHE) library from the programmer. The codes written in various libraries remain unchanged, with the only distinction being the absence of a unified common API requirement [29]. When designing a framework, there are several prerequisites that developers need to consider. These requirements aim to provide developers with a clear understanding of the fundamental concepts involved in selecting a framework and guidance on how to effectively work with it. Let us go over them one by one:

1. After executing each statement in a program, the framework mandates the maintenance of an accurate state. Simply put, if the program encounters an interruption at any point, the framework must decrypt the encrypted

Table 1 Comparison of homomorphic libraries against multiple parameters

References	LIB	Method of encryption	Operation supports	Operation based on	Schemes used
[9, 10]	MS SEAL	Symmetric and asymmetric	Addition, subtraction, multiplication, bitwise operations, matrix operation, square, Negation	Real numbers, integers	BGV/BFV
[11, 12]	PiHeaan	Asymmetric	Addition, subtraction, multiplication	Real numbers, integers	CKKS
[15]	TFHE	Asymmetric	Addition, substraction multiplication, Boolean functions and binary gates, square	NAND gate, bootstrapping	Gate by gate bootstrapping
[17]	HeLib	Asymmetric	Multiplication, addition, shift, set, bitwise operations, matrix operation, square, negation	Real numbers, integers	BGV/CKKS
[20]	FV/NFLib	Asymmetric	Addition, subtraction	Real numbers, integers, polynomial	FV-scheme
[21]	Pallisade	Asymmetric	Addition, substraction, Boolean functions	Real numbers, integers	BGV/BFV, CKKS, FHEW
[23]	Concrete	Asymmetric	Multiplication, addition, substraction	Real numbers, unsigned integers	TFHE
[25]	lattigo	Asymmetric	Multiplication, addition, subtraction	Real numbers, integers, polynomial	RNS (residue number system), CKKS, BFV
[72]	FHE C++ Transpiler	Asymmetric	Addition, substraction multiplication, Boolean functions and binary gates, square	Real numbers, integers	Gate by gate boot- strapping

References	LIB	Library used	Encoder	Language support	Type of output
[9, 10]	MS SEAL	ZLIB and Zstandard	CKKS encoder	C++, Python	Approximate
[11, 12]	PiHeaan	Python library	CKKS ENCODER	C++, Python	Approximate
[15]	TFHE	Ring variant of GSW	TRLWE	C	Approximate
[17]	HeLib	NTL Mathematical Lib	BGV, CKKS, RLWE	C++, Python	Approximate/optimized
[20]	FV/NFLib	NF Lib	TFHE, LWE	C++	Checks correctness/approximate
[21]	Pallisade	None		C++, C	Approximate
[23]	Concrete	Zama's variant	LWE	C	Approximate
[25]	Lattigo	lattigo/ring sub-package	CKKS,BFV	C++, Go	Approximate/exact
[72]	FHE C++ Transpiler	TFHE library, XLS library	–	C++, Python	Approximate

values, ensuring that the decrypted values match their unencrypted counterparts. This ensures consistency and integrity in the program's execution, allowing for seamless resumption in case of any interruptions.

2. The usage of a standard compiler is essential, and the execution of the program should be carried out using standard executables. Executing the program with standard executables ensures that the program can run on common platforms and environments, promoting portability and interoperability.
3. During the compilation of any program, it is essential to carry out precise computations. This requirement holds true even when dealing with encrypted data, such as performing branching operations on encrypted data. It is crucial to accurately record and handle the processing of encrypted data during the compilation process. This ensures that all computations involving encrypted data are accurately accounted for and properly managed.
4. There is no requirement to have knowledge of any plaintext or ciphertext prior to program compilation, as the generation of plaintext and ciphertext occurs during runtime. The process of obtaining plaintext and ciphertext data happens dynamically during the execution of the program, rather than being predetermined at

the compilation stage. Therefore, it is not necessary to have prior knowledge of plaintext or ciphertext during program compilation, as these values are generated and utilized during runtime.

5. The encryption process maintains the integrity and confidentiality of the data, allowing it to be securely processed within the program without necessitating additional encryption steps for program updates. This independence of ciphertext encryption simplifies program maintenance while preserving the security of the encrypted information.

Let us now go over some frameworks:

E3

E3, which stands for Evervault Encryption Engine, was developed to address the inherent insecurity of exchanging data over the internet in plaintext. In the past, this left data vulnerable to unauthorized access and manipulation. E3 serves as a highly secure and user-friendly service specifically designed for cryptographic operations. The operations performed by E3 are characterized by high scalability, reliability, and low latency. Cryptographic operations, such as relays and cages, are carried out to ensure the protection of users and their datasets from unauthorized parties. E3 functions as a security tool, acting as a shield against security issues like data breaches or leaks.

To mitigate the complexities and risks associated with data security, E3 was built to enable end-to-end encryption. This ensures that data remains encrypted at all times, providing a robust solution for computing sensitive information while maintaining the confidentiality and integrity of the data [30].

There were four required aspects which were taken care of for establishing E3:

1. **Evervault must verifiably not be able to access or decrypt data** E3 was purposefully designed to ensure the impossibility of tampering or unauthorized access to any confidential data. Evervault, the entity working with sensitive data, took the initiative to demonstrate to developers that the data stored within E3 was completely inaccessible to Evervault itself. This assurance was provided to instill confidence in the security and privacy of the data entrusted to E3, highlighting the commitment to safeguarding sensitive information and maintaining strict data confidentiality [31].
2. **E3 should be use-case agnostic** To maintain stability, E3 was carefully maintained, and great attention was given to preserving the integrity of the existing codebase. The aim was to ensure that any new code additions or modifications did not disrupt the fundamental structure of E3. This approach ensured that the functionality of E3 remained reliable and consistent, minimizing the risk of introducing new issues or vulnerabilities. The stability of E3 served as a foundation for developers to confidently build upon, allowing for the secure and efficient implementation of encryption strategies.
3. **Low latency, high throughput, and full redundancy should be maintained by E3** The third requirement was to keep cryptographic operations secure and redundant while achieving an optimized high throughput.
4. **The integrity of keys must be absolutely guaranteed** E3 is designed to prioritize the integrity of encryption keys, ensuring that data remains encrypted and is never decrypted under any circumstances. This fundamental principle is embedded within the architecture of E3, with a strong emphasis on maintaining the security of the encryption keys. Even at the customer's request, E3 is specifically engineered to prevent the disclosure or access to the encryption keys. This means that the keys are not accessible or enclosed, even when explicitly requested by the customer. This approach ensures that the encryption keys remain secure and confidential, reducing the risk of unauthorized access or potential compromises.

There are two implementing ways in which E3 was implemented, and they are [32]:

1. Fully-homomorphic encryption (FHE)

   Fully Homomorphic Encryption (FHE) is a cryptographic technique that enables performing computations on encrypted ciphertext without the need for decryption. FHE based on ideal lattices was introduced in 2009, leveraging addition and multiplication operations to enable computation on encrypted data. However, implementing FHE within the E3 framework presented challenges due to certain limitations. E3 was unable to support the implementation of FHE due to its inability to handle high-scale use cases and general-purpose applications effectively. The limitations of E3 prevented the seamless integration of FHE into the framework, impeding its ability to perform computations on encrypted data.
2. Trusted execution environments (TEEs)

   The architecture of Trusted Execution Environments (TEE) was widely acknowledged to have design flaws, particularly in the loose coupling of input/output (I/O) operations. However, E3 was determined to address these challenges and ensure the secure and reliable functioning of TEE. Despite the acknowledged issues, E3 approached the integration of TEE with confidence, emphasizing the importance of security and safety. By leveraging their expertise and careful engineering, E3

aimed to overcome the shortcomings and enhance the functionality of TEE.

Hence, to make it possible, they opted for 3 approaches, such as [33];

1. **Intel Software Guard Extensions (SGX)**: SGX, short for Software Guard Extensions, is an Intel instruction set extension that focuses on maintaining both integrity and confidentiality, except in the most extreme cases. It incorporates the Intel Attestation Server (IAS) mechanism, which ensures the security of the channel used for exchanging sensitive data and allows for third-party verification. One of the significant challenges faced by SGX was the lack of support from many cloud providers, particularly those utilizing the SkyLake micro-architecture, as they did not have enabled BIOS and lacked SGX compatibility. To address this limitation, SGX2 was introduced, promising to overcome the drawbacks of its predecessor. SGX2 utilized the Gemini Lake micro-architecture, which successfully resolved various issues associated with SGX.
2. **AMD Secure Encrypted Virtualization (AMD SEV)**: AMD SEV, short for Secure Encrypted Virtualization, enabled virtual machines to have a dedicated encrypted memory set. However, it suffered from a significant drawback in the form of unrestricted I/O, which increased the attack surface and compromised its authenticity. As a result, it necessitated extensive maintenance of the operating system and kernel to ensure security [34].
3. **AWS Nitro Enclaves**:AWS Nitro Enclaves is designed to handle resource isolation by utilizing dedicated hardware specifically built for this purpose. It is a feature of EC2 that offers an isolated execution environment. By doing so, it effectively reduces the potential attack surface when processing data applications.

E3 supports multiple HE libraries and is capable of combining two types of computations: bit-level computation and modular computation. These computations are secure and serve as general-purpose computations. E3 utilizes protected types, which are equivalent to standard C++ integral types, and does not rely on specific encryption schemes. This characteristic of E3 enables data encryption to be independent. Additionally, E3 supports bridging, which involves blending different arithmetic abstractions and leads to enhanced performance.

SHEEP

SHEEP was developed as a research framework with the purpose of exploring the field of homomorphic encryption and its associated library. It is specifically dedicated to the HE library and does not make any commitments or guarantees regarding other external libraries or their usage. The primary objective of the SHEEP project is to provide cryptographers with a platform where they can assess various fully homomorphic encryption functions and work with their technologies. SHEEP has the capability to evaluate different computations using multiple libraries and implements various HE schemes with exceptional security measures [35].

The SHEEP platform consists of several components, including an interpreter for the SHEEP language, a reporting component, and an abstraction layer. The interpreter facilitates the execution of instructions written in different programming languages without the need for compilation into machine-understandable code. The instructions are directly executed from the scripted or provided programming language. The supported languages include the high-level SHEEP language and predefined benchmarks. The computed results can be visualized through a web interface.

In general, computations on the SHEEP platform are described in a generic manner, using a library-agnostic language similar to an assembly language. The abstraction layer plays a crucial role in concealing the details of executed programs or computations within the subsystem. It divides a given task into two separate entities that process the task step by step. Each entity is assigned to a specific task or computation, ensuring that the other entity remains idle until it receives data for further processing. This approach maintains interoperability and platform independence. Furthermore, the abstraction layer ensures that there is no sharing of a common file system between users or operations.

In the SHEEP platform, an abstraction layer is utilized to integrate homomorphic libraries through their corresponding wrappers. This layer serves multiple functions. Firstly, it logs internet request data, capturing and storing the collected data in a log server, which is subsequently used for web protection purposes. Secondly, it provides insights into the current status and activity of the internet. These components contribute to the overall computational capabilities of the SHEEP platform.

The homomorphic encryption scheme supported by SHEEP includes various operations such as encryption, decryption, homomorphic addition, multiplication, and comparison. These operations enable secure computations while preserving the confidentiality of the data.

SHEEP offers programmers the ability to write code using a predefined set of benchmarks included in the library. Additionally, users can execute their own executable programs through a web interface. The native benchmarks are categorized based on their complexity levels, ranging from low-level microbenchmarks for basic homomorphic encryption (HE) operations to data analysis tasks.

Table 2 Comparison of various accelerators based on different parameters

References	Name	Installed on	Supported operation	Services provided	Languages support	Mechanism opted
[30, 32]	E3	Window /Linux	Cryptographic	Relays and cages	C++	IAS (Intel Attestation Server)
[35]	SHEEP	Window /Linux	Cryptographic	Data analyzation	Library agnostic language	Native benchmark

References	Name	Encryption	Performance	Open source	Language format
[30, 32]	E3	FHE/TEE	Latency, security, and bandwidth	Yes	Rust
[35]	SHEEP	FHE	No latency, security, and bandwidth	Yes	High level SHEEP language

When a program is provided as input, it is evaluated across all available HE libraries according to the program's instructions. The evaluation process first checks if the HE libraries support the provided instructions. The evaluated results are then cross-checked for correctness and ciphertext size. After completing the computations, the web interface stores the ciphertext size information in a database.

It should be noted that the current version of the SHEEP platform has limitations in addressing issues related to security, latency, and bandwidth constraints. As a result, the SHEEP code is maintained as open source, relying on community efforts to improve and enhance its capabilities [36]. We have compared all frameworks to different parameters in tabulated form (Table 2) so that it becomes easy for readers to come to a conclusion regarding which library to use for their further implementation.

FHE-IBM Encryption Toolkit

The IBM Integration Toolkit is built on the Eclipse platform and comprises an integrated development environment (IDE) and a graphical user interface (GUI). This toolkit is designed to facilitate the development of resources that primarily involve message flow. It achieves this by allowing the connection of these resources to one or more integrated nodes for deploying the message flow [37].

Installation

The IBM Integration Toolkit is compatible with Windows and Linux operating systems. It can be installed on these platforms to enable its functionality. Furthermore, communication with integration nodes is specifically supported when using IBM Integration Bus Version 10.0.

Beginning with IBM

When initiating the IBM Integration Toolkit, users will encounter a single window that appears, containing multiple perspectives. This initial setup might be slightly overwhelming for some users. To facilitate a smooth start, there are two recommended methods for launching the IBM Integration Toolkit: 1, On Windows: Open the IBM Integration Console and enter "./iib toolkit" as the command. 2, On Linux: Access the command environment and enter "./iib toolkit" to initiate the toolkit. By following these steps, users can seamlessly begin their experience with the IBM Integration Toolkit.

The IBM Integration Toolkit Perspectives

A perspective in the IBM Integration Toolkit refers to a collection of views that provide users with the necessary information and tools to accomplish specific tasks. These views act as references and assist users in completing their assigned tasks. Editors, on the other hand, are utilized for editing or browsing various resources such as projects, folders, and files within the IBM Integration Toolkit. They enable users to interact with and modify these resources as needed [38]. The perspective can be described as:

- **Integration Development perspective** The Integration Development perspective in the IBM Integration Toolkit is specifically designed for application development purposes. When starting the IBM Integration Toolkit, the initial view that appears is the Integration Development perspective, providing users with the necessary tools and features for application development tasks.
- **Debug perspective** Debug perspective is used to debug those messages which are flowing through it.

The IBM Integration Toolkit is a software tool that utilizes open-source code to enable data processing while maintaining strict access control. It heavily relies on encrypted data for computations, employing lattice cryptography as its encryption method. The toolkit supports various operations such as encryption, decryption, and other homomorphic operations, typically performed on data blocks. notably, it can execute multiplication operations through 30,000 single-bit multiplications.

For encryption, the IBM Integration Toolkit employs 256-bit AES algorithm keys, utilizing the maximum key length allowed by AES. This toolkit empowers users and programmers to incorporate IBM's solutions into their research or projects. By performing computations directly on encrypted data, the use of Fully Homomorphic Encryption (FHE) can significantly impact privacy and data security. The toolkit is based on the open-source HE library called HELib, providing a range of low-level routines (e.g., add, multiply, set, shift) and higher-level features like multi-threading and automatic noise management. HELib has gained recognition as one of the most versatile encryption libraries globally [39]. While collaboration with the IBM Fully Homomorphic Encryption Toolkit can enhance performance, it also introduces vulnerabilities to theft or tampering. The IBM Integration Toolkit supports macOS and iOS platforms, in addition to Linux and Android.

To maximize the benefits of the IBM Integration Toolkit, it offers a playground environment with a sample application that enables querying an encrypted database.

Cingulata

Cingulata [70] stands as both a runtime environment and a versatile toolchain, purpose-built to execute C++ programs with an ingenious twist. The remarkable innovation lies in its ability to perform program executions on encrypted data, all accomplished through the sophisticated framework of Fully Homomorphic Encryption (FHE) techniques. As an open-source compiler, Cingulata embraces collaboration and community contributions.

At its core, Cingulata empowers developers with the means to harness the extraordinary potential of FHE computations. Through the fusion of C++ programming and FHE methodologies, it unlocks a realm of secure computations on data shrouded in encryption. This amalgamation of technologies is orchestrated seamlessly by a meticulously crafted compiler toolchain, efficiently transmuting C++ programs into their encrypted counterparts, poised for FHE-driven execution.

Execution Modes

1. Offline mode (compiled): Cingulata's offline compiled mode incorporates specialized optimization modules that are dedicated to diminishing the size and multiplicative depth of Boolean circuits in applications. Additionally, it seamlessly interfaces with lwe-estimator to ensure the secure generation of Fully Homomorphic Encryption (FHE) parameters.An advanced parallel runtime environment facilitates the execution of these Boolean circuits, enabling the utilization of various FHE schemes.

   Within the compiled mode, Cingulata's front-end undertakes the conversion of C++ input code into a corresponding representation in the form of a Boolean circuit (constructed using logical AND and XOR gates). Subsequently, the intermediate layer refines the Boolean circuit crafted by the front-end, employing ABC-an open-source tool renowned for hardware synthesis optimization. Ultimately, the refined circuit is dynamically executed on encrypted data, employing diverse FHE schemes for heightened security.
2. Online mode (on-the-fly execution): The C++ input application is executed directly through a low-level implementation of the intended cryptosystem. Currently, the offline mode is tailored for somewhat homomorphic schemes (using an in-house B/FV implementation), while the online mode is designed for fully homomorphic schemes, utilizing the TFHE library.

Installation

Cingulata can be installed through two distinct approaches: one involves the B/FV backend, and the other entails the utilization of the TFHE library backend.

Two modes of selection are available: static (set as the default value) and interactive.

1. Static mode: In the absence of a parameter set satisfying the provided constraints, the system compiles a roster of parameter sets within the database, considering factors such as: Specified multiplicative depth, Designated bkz reduction cost model and Security level falling within the range [lambda-8, lambda+56]. From this roster, a suitable parameter set is selected. In cases where the roster remains devoid of options, the system transitions into interactive mode.
2. Interactive mode: In the event that no parameter set adhering to the specified constraints is located, the system offers the user the option to choose a parameter set from the database using the yad tool. This process allows for flexibility, enabling the user to adjust the three defined constraints as needed. Furthermore, the system grants permission for the database to be refreshed through the HEAD commit of the LWE-Estimator, provided that a new version is accessible.

Accelerator

HE accelerators are utilized to enhance the performance of fully homomorphic encryption (FHE) operations [40]. FHE enables computation on encrypted data without the need for decryption, ensuring that only the data owner can decrypt it. However, FHE schemes based on lattices introduce significant noise in the ciphertexts for security reasons, which can impede computation speed [41]. In order to sustain FHE computations, a process called bootstrapping is performed to reduce the noise. Bootstrapping involves using accelerators to optimize the implementation and bridge the performance gap between encrypted data and plaintext. Studies and research summaries indicate that FHE operations can sometimes lead to memory bottlenecks [42]. By employing HE accelerators, the efficiency and overall performance of FHE operations can be improved, allowing for more practical and efficient computation on encrypted data while mitigating the impact of noise and memory limitations [43]. Let us discuss some of the main accelerator.

Homomorphic Encryption Acceleration (HEAX)

HEAX [44] is a high-performance architecture that computes homomorphically encrypted data using an HE accelerator used for improving the computational speed. HEAX is a kind of Homomorphic Encryption Acceleration. This accelerator was introduced to overcome the issue of the slower rate of computation. The hardware acceleration is used to reduce the performance gap. The performance gap is a process where the rate of computation is reduced between the homomorphic evaluations in software and computations on plaintext. The performance gap lies within five to six orders of magnitude, which is quite a delaying process. Therefore, HEAX reduces the performance gap [45].

HEAX, which stands for Homomorphic Encryption Acceleration, is a high-performance architecture designed to enhance the computational speed of homomorphically encrypted data. It addresses the challenge of slower computation rates typically associated with homomorphic evaluations. By utilizing hardware acceleration, HEAX aims to reduce the performance gap that exists between software-based homomorphic evaluations and computations performed on plaintext data.

The performance gap refers to the significant difference in computational speed between homomorphic evaluations and traditional computations on plaintext data. This gap can span several orders of magnitude, resulting in delays and inefficiencies. HEAX is specifically developed to mitigate this performance gap, allowing for faster and more efficient computations on homomorphically encrypted data. By leveraging HEAX's acceleration capabilities, the computational speed of homomorphic encryption is significantly improved, narrowing the performance gap and making homomorphic computations more practical and feasible in real-world scenarios.

HEAX is a non-programmable accelerator that offers several advantages over programmable accelerators due to its ability to reuse computational elements. It optimizes the key-switching mechanism of RNS-HEaaN by breaking it down into partitioned steps and assigning a dedicated block for each step. This approach enables high throughput performance [46]. The key-switching operations in HEAX are executed using a block pipeline architecture consisting of six stages. This architecture processes polynomial residues sequentially, resulting in a high asymptotic throughput. HEAX employs separate arithmetic units for homomorphic multiplication and re-linearization. However, hardware-based rescaling of homomorphic addition and subtraction operations is not feasible.

The unique implementation of RNS-HEaaN on FPGA is specific to the HEAX accelerator and sets it apart from other

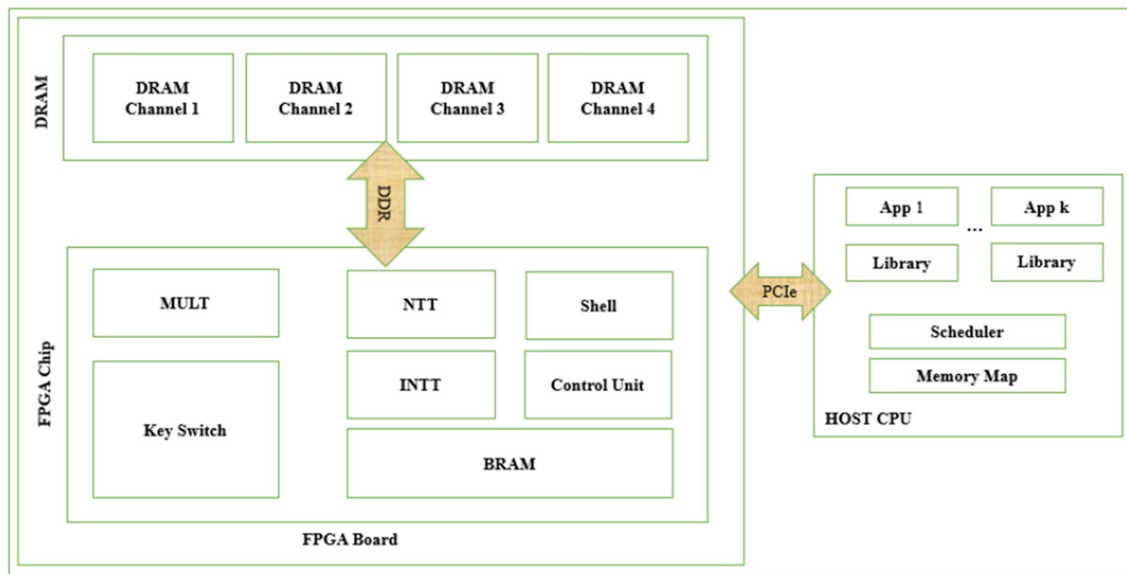

Fig. 1 System view of HEAX

accelerators. This implementation represents a significant advancement in FPGA-based homomorphic encryption.

In real-time applications, data dependencies play a crucial role, as the software host incurs overhead when processing operands and results. Consequently, the processing time of an application is determined by latency rather than throughput. Achieving high asymptotic throughput in HEAX assumes a certain number of data-independent homomorphic operations and no overhead on the host side, which manages input–output streams of ciphertexts (Fig. 1).

One limitation of HEAX is its lack of programmability. Its architecture is specifically designed for block-pipelined key-switching operations and does not support other functionalities [47].

System View of HEAX

Let's examine the system view of HEAX as shown in Fig. 1 and discuss the data flow. HEAX comprises an FPGA board, a host CPU, and a Peripheral Component Interconnect Express (PCIe) bus, with all components interconnected. Starting with the FPGA board, it offers two methods of data storage. Firstly, it includes off-chip DRAM, and secondly, it utilizes on-chip BRAM.

- **Off-chip DRAM** The primary function of the off-chip memory is to store intermediate results, which can impact the overall performance of the system. However, the read and write operations performed by the off-chip memory introduce high response delays, leading to decreased performance.
- **On-chip BRAM** The on-chip memory, specifically the Block RAM (BRAM), exhibits excellent throughput and response time. It has a capacity of a few megabits, allowing for efficient storage and retrieval of data within the FPGA chip.

The FPGA board includes various components, such as the shell, which is responsible for minimizing the number of wrappers. It ensures that each physical device is assigned only one wrapper, simplifying the management of tasks and the merging of multiple devices. The control unit plays a crucial role in managing these tasks, handling interface issues, and optimizing performance. On the FPGA side, buffers are allocated to store received data. Two methods are employed for this purpose: (i) Sequencing and Batching, and (ii) Double and Quadruple Buffering. The selection of these methods is based on the specific requirements of the system [48].

PCIe Data Transfer: There are three critical steps that occur when data transfers in PCIe; they are as follows:

- The polynomial content is copied to pin memory pages, and this process involves issuing a memcpy operation. To optimize the data copying time, direct memory access (DMA) is utilized.
- The CPU signals to the FPGA whether the data is ready or not.
- Finally, the FPGA reads the data from PCIe.

MULT module and key switching: The MULT module and key switching operations are executed using two different

buffering techniques: Double Buffering and Quadruple Buffering.

- In the case of the MULT module, Double Buffering is employed. This allows the FPGA to read from one buffer while the CPU simultaneously writes to another buffer. By using two buffers, the operations can be performed in parallel, improving efficiency.
- On the other hand, key switching requires Quadruple Buffering due to the presence of numerous data dependencies on the input polynomial. To prevent buffer overriding and ensure data integrity, the read and write processes occur one after another. The use of four buffers enables the sequential execution of key switching operations.

HOST CPU: On the HOST-CPU side, sequencing and batching operations are carried out to optimize the execution of multiple operations in a program. The user has the freedom to choose their preferred library for performing these operations, as the library selection is entirely dependent on the user. The scheduling of tasks is managed by the scheduler, which aims to maximize CPU utilization. The scheduler ensures that all scheduled tasks are executed efficiently. Meanwhile, the memory map plays a crucial role in managing available memory. It provides information about the memory availability and allows debuggers to allocate memory addresses for storing actual data. Overall, the HOST-CPU side performs sequencing, batching, task scheduling, and memory management to enhance the performance and resource utilization of the program.

F1

F1 is recognized as the pioneering FHE accelerator, being the first system designed to accelerate entire programs of fully homomorphic encryption [49]. It is a programmable accelerator capable of executing complete FHE programs. F1 is equipped with multiple functional units, including number-theoretic transforms, modular arithmetic, and structured permutations, providing extensive support for various operations. One of the notable advantages of F1 is its ability to minimize data movement by achieving high computed throughput. This results in improved performance and efficiency (Fig. 2). Additionally, F1 offers support for multiple FHE schemes, namely BGV, CKKS, and GSW, utilizing the same hardware architecture. In summary, F1 revolutionizes FHE acceleration by enabling the execution of full programs, delivering efficient data processing, and accommodating various FHE schemes within a single hardware platform. F1 is made up of three major characteristics that define F1 very precisely [50]. They are:

- F1 achieves optimal speed by synchronizing with FHE operations and leveraging wide functional units with vector processing capabilities. It incorporates specialized Number-Theoretic Transform (NTT) units and an automorphism functional unit to address the challenges posed by complex dataflows in key operations. A distinguishing feature of F1 is its adoption of static scheduling, which enables multiple operations per cycle with minimal control overhead. This mitigates the difficulties associated with dataflow graphs in FHE programs, particularly in arithmetic operations on vectors where operation dependencies are predetermined. Thus, F1 effectively overcomes this limitation.
- In terms of data movement optimization, F1 employs strategies to minimize latency and decouple data movement. It employs mechanisms to hide access latencies by preloading data in advance and makes efficient use of limited memory bandwidth through scheduling algorithms.
- F1's synchronized approach, wide functional units, NTT capabilities, static scheduling, and data movement optimization contribute to its effectiveness in accelerating FHE operations and addressing the challenges posed by complex dataflows and limited memory bandwidth.

System View of F1

The system view of F1 is discussed in Fig. 2, Tables 3, 4, 5.

- **Vector execution with functional units (FU):** F1 incorporates a vector processing mechanism with specialized functional units (FUs) specifically designed for FHE operations. These operations include automorphisms, modular addition, modular multiplication, and NTTs (both forward and inverse) performed within the same unit. The FUs within F1 are fully pipelined, ensuring consistent throughput throughout the execution of these operations [51].
- **Compute clusters:** The compute clusters in F1 are organized groups of functional units (FUs). Each FU within a cluster is equipped with vector processing capabilities, including automorphism, multipliers, adders, and NTT. Additionally, the FUs in F1 are accompanied by a banked register file, which ensures a steady supply of operands in every cycle, enabling continuous utilization of the FUs. F1 employs a chip architecture that consists of multiple compute clusters. These compute clusters, while executing operations, do not directly access the main memory due to the high bandwidth requirements associated with such access.
- **Memory system:** F1 incorporates a well-designed memory system that includes a large and partitioned scratchpad, integrated compute clusters, and high-bandwidth

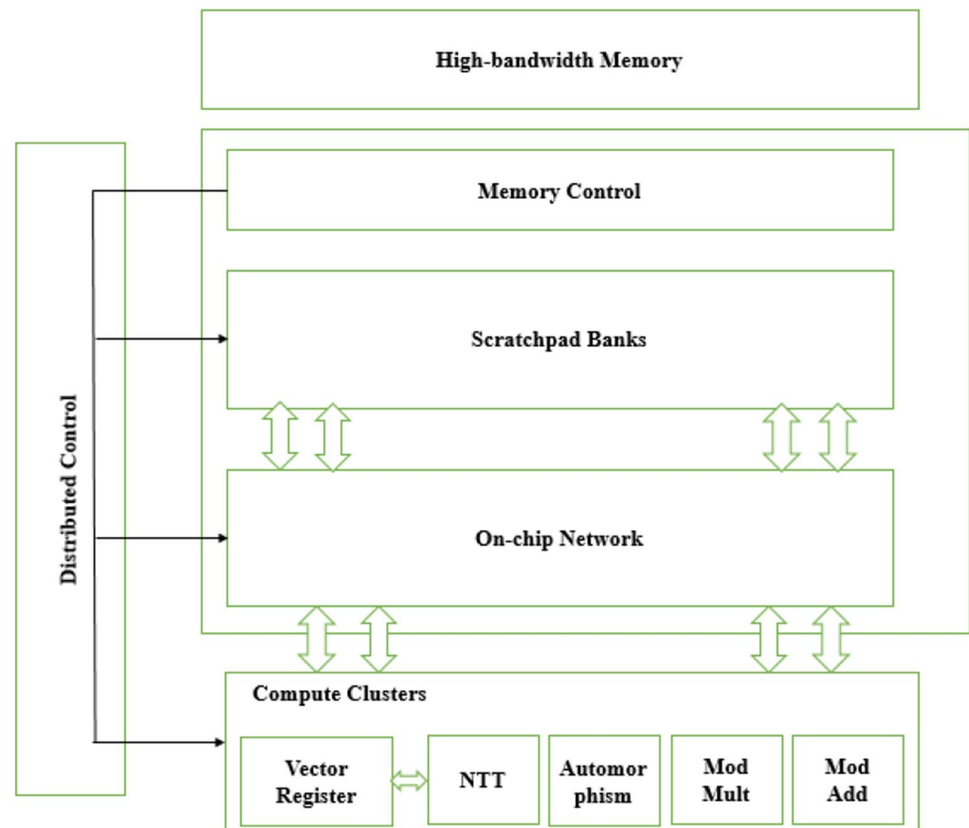

Fig. 2 System view of F1

off-chip memory. To mitigate the effects of main memory latency, F1 utilizes decoupled data orchestration.

- **Scratchpad banks:** The scratchpad banks serve as intermediate storage and fetch data from the main memory. As the off-chip memory has limited bandwidth, the scratchpad banks hold the required data to avoid frequent staging from the main memory. Each scratchpad bank efficiently transfers vectors to the computing units, minimizing the need for extended staging at the register file side.
- **Static scheduling:** The compiler plays a crucial role in static scheduling and managing data transfers to prevent data hazards. Static scheduling simplifies the logic across the chip, eliminating the need for dynamic arbitration in handling conflicts. The on-chip network employs simple switches for independent configuration changes, and the functional units operate without stalling logic.
- **Distributed control:** F1 processes individual instruction streams for each component. Programs are compiled into linear sequences of instructions without control flow within F1. Each component, including register files, network switches, functional units, scratchpad banks, and memory controllers, has its own instruction stream. These instructions are fetched in small blocks through the control unit.
- **Register file (RF) design:** The register file design in F1 accommodates long vectors, which are distributed across banks. The functional units cycle through the banks, accessing different banks in each cycle. This approach eliminates the need for multiporting and its associated complexities.

Medha

Medha is an accelerator designed specifically for enhancing the performance of cloud-side operations in homomorphic encryption schemes. It focuses on accelerating key operations such as key-switching, relinearization, homomorphic addition, subtraction, and multiplication. Implemented as an instruction-set processor architecture, Medha targets the RNS (small coefficients) variant of the HEaaN homomorphic encryption scheme [52]. The design of Medha emphasizes parallel processing at each level of the implementation hierarchy, leveraging a modular hardware design to achieve efficient homomorphic operations with minimal computational time. It is a programmable accelerator, supporting all routines of the levelled RNS-HEaaN scheme for comprehensive homomorphic evaluation. When compared

Table 3 Application, advantages, and limitations of multiple tools

References	Tools	Application	Advantages	Limitations
[57]	SEAL	Secure cloud computing, privacy-preserving machine learning, secure multiparty computation, medical data analysis, cryptocurrency and blockchain and iot data processing	Secure outsourcing, flexibility, efficiency, multiparty computation, cross-platform support, ongoing development, real-world applications	Performance and computational overhead, key management and distribution, limited set of operations, homomorphic noise growth, large key sizes, complexity of development, limited use cases
[58]	TFHE	Privacy-preserving machine learning, encrypted database queries, secure multi-party computation, homomorphic encryption as a service, data monetization and sharing, secure voting systems	Fast low-level operations, bootstrapping technique, suitable for cloud computing, parameter selection flexibility, mathematically rigorous, efficiency improvements, rich set of libraries	Computational overhead, noise growth and bootstrapping, bit-level operations emphasis, parameter selection complexity, limited high-level abstractions, complexity for developers, not suitable for all operations
[59]	HeLib	Secure cloud computing, private data analysis, machine learning on encrypted data, privacy-preserving biometric authentication	Collaborative analysis, data confidentiality, mitigating insider threats, compliance with regulations, data aggregation, machine learning with privacy, reduced data preprocessin	Limited supported operations, parameter selection sensitivity, limited practical use cases, security assumptions, learning and integration curve, homomorphic encryption schemes, lack of standardization, ongoing research and development
[60]	FV/NFLib	Privacy-preserving computation, data security, secure outsourcing, collaborative analysis, regulatory compliance, private machine learning, secure data aggregation, mitigation of insider threats	Privacy-preserving computation, data security, secure outsourcing, collaborative analysis, regulatory compliance, private machine learning, secure data aggregation, mitigation of insider threats	Limited efficiency, parameter selection, noise growth, limited supported operations, key management, large ciphertext size, learning curve, security assumptions, trade-offs, current state of research, interoperability and standardization
[61]	Palisade	Secure cloud computing, privacy-preserving data analysis, secure outsourcing of computation, privacy-preserving biometric authentication, financial analysis	Diverse cryptographic techniques, privacy-preserving computations, customizable and extensible, research and development, cryptographic protocol development, privacy compliance, encryption research	Noise accumulation, limited supported operations, key management, security assumptions, large ciphertext size, learning curve, ongoing research, interoperability, potential bugs and vulnerabilities
[62]	Concrete	Data encryption, collaborative analysis, privacy-preserving queries, secure aggregation, regulatory compliance, data ownership	Preserved data confidentiality, data sovereignty, regulatory compliance, collaborative insights, trust and cooperation	Processing delays, scalability challenges, resource requirements, balancing privacy and efficiency
[63]	Lattigo	Secure multi-party computation (mpc), privacy-preserving data sharing, cryptographic protocols, regulatory compliance, collaborative research, secure financial data analysis	Lattice-based security, privacy-preserving computations, flexible encryption schemes, potential for quantum resistance, research and innovation, community support, regulatory compliance, long-term security	Limited hardware support, learning curve, noise accumulation, limited application domains, algorithmic advances, cryptanalysis risk, interoperability
[64]	E3	Genomic research, secure data sharing, authentication and identity, supply chain analytics, privacy-preserving AI in healthcare	Secure aggregation, confidential computation, regulatory compliance, mitigating insider threats, auditing and accountability, securing IoT and edge computing, post-quantum security	Computational complexity, limited set of operations, large ciphertext size, key management, homomorphic noise, limited use cases
[65]	SHEEP	privacy-preserving computations, collaborative analysis, setting up FHE	Good for practitioner's, private information retrieval, confidential computation, secure inner product	Limited library access, incomplete solution to the given problem, unsafe external libraries
[44, 46, 66]	HEAX	Secure multi-party computation, data monetization, privacy-preserving data analysis, lattice-based cryptosystems	Diverse cryptographic techniques, authentic computations, data sovereignty, flexible encryption schemes, regulatory compliance	Limited supported operations, low key management, potential bugs and vulnerabilities, more resource requirements, limited application domains

Table 3 (continued)

References	Tools	Application	Advantages	Limitations
[49, 67]	F1	wide-vector execution, modular addition, modular multiplication, NTTs, automorphisms	Fixed, small arithmetic, fully pipelined, executes full FHE programs, high-throughput functional units	high computation overheads, bugs and vulnerabilities, processing delays, low balancing privacy and efficiency, limited hardware support
[52, 68]	Medha	Secure data analysis, private cloud computing, financial services, genomic research, machine learning on sensitive data, internet of things, supply chain collaboration, government and law enforcement	Privacy, data security, outsourcing computations, data utilization, secure machine learning, regulatory compliance, reduced trust requirements	Computational overhead, limited functionality, key management, algorithm choice, security assumptions, communication overhead, large ciphertext size
[69, 71]	FHE C++ Transpiler	Secure computation on encrypted data, privacy-preserving cloud computing, confidential data analysis, secure machine learning, outsourcing computations, secure data sharing	Privacy-preserving analytics, secure machine learning, confidential collaboration, compliance with regulations, data monetization, innovative applications	Computational overhead, limited efficiency, large ciphertext size, key management complexity, algorithm choice, limited supported operations, implementation complexity
[70]	Cingulata	Secure computation on encrypted data, secure machine learning, financial data analysis	Privacy-preserving analytics, secure machine learning, confidential collaboration, large-scale deployment	Computational overhead, limited efficiency, large ciphertext size

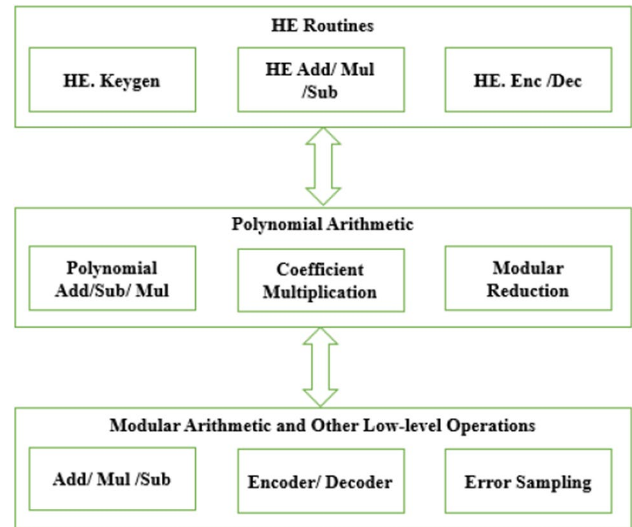

Fig. 3 System View of Medha

to state-of-the-art reconfigurable hardware accelerators, Medha demonstrates faster computational latency within a smaller area, assuming the parameters remain constant. The RNS (small coefficients) variant of the HEaaN homomorphic encryption scheme employs residue polynomials for efficient computation of small coefficients. These polynomials lend themselves well to parallel processing, making the RNS HEaaN scheme more implementation-friendly than the original HEaaN scheme [53]. Medha is designed as an instruction-set architecture (ISA), enabling programmability by treating the repeated resources as instructions. In order to process the residue polynomials of multiple high-level units, computation is performed in parallel. These residue polynomial units are collectively referred to as Residue Polynomial Arithmetic Units (RPAUs). When Medha encounters a high-level instruction, it is translated into the corresponding RPAUs to perform the necessary computations [54].

System View of Medha

The system view of Medha, as depicted in Fig. 3, comprises three main components: HE Routines, Polynomial Arithmetic, and Modular Arithmetic, along with other low-level operations. The implementation of Medha involves two distinct sides: the user side and the cloud side. (Table 6).

On the user side, functions like key generation, encryption, decryption, encoding, decoding, and error sampling are performed at the lowest level. These tasks are essential for setting up and managing the homomorphic encryption scheme. On the cloud side, Medha provides operations such as homomorphic subtraction, addition, multiplication, and relinearization. These operations enable the computation of homomorphic functions on encrypted data without the need

Table 4 Comparison of different accelerators against multiple parameter

References	Accelerator	Memory usage	Programming type	Arch.	Operation support
[44, 46]	HEAX	Off-chip memory	non- Programmable	Fixed pipelined architecture	Homomorphic multiplication, relinearization
[49]	F1	On-chip memory	Programmable	Instruction based wide-vector processor architecture	Homomorphic multiplication, addition and subtraction, rescaling, relinearization
[52]	Medha	On-chip memory	Programmable	Instruction based	Homomorphic multiplication, addition and subtraction, rescaling, relinearization

References	Accelerator	Polynomial degree	Application	Efficiency
[44, 46]	HEAX	2^{14}	Less	Similar to Medha
[49]	F1	2^{14}	Applicable	Equal to Medha
[52]	Medha	2^{15}	More practical	Better than HEAX

Table 5 Comparison of diffrent tools against multiple parameter

Component	SEAL	Pi- Heean	TFHE	HeLib	FV/NFLib	Palisade	Concrete
Space complexity	2 MB	1 MB	2 MB	2 MB	2 MB	2 MB	2 MB
Time complexity	$2 \times (n/2)$	$O(\lambda\ 10)$	$R[X]/(x^n + 1)$	mod(p)	O(log2(n/2n))	p mod t	–
User friendly	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Availability	GitHub	GitHub	GitHub	GitHub	GitHub	GitHub	GitHub
Open source	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Serialization	Yes	No	Yes	Yes	No	Yes	Yes
Ciphertext size	1 MB	1 MB	1 MB	1 MB	1 MB	1 MB	1 MB
Symmetric	No	No	No	No	No	Yes	Yes
Multithreading	Yes	Yes	No	Yes	Yes	Yes	Yes
Noise	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Relinearization	Yes	Yes	No	Yes	Yes	Yes	Yes
Memory Req	4 GB	2 GB	2 GB	2 GB	2 GB	2 GB	2 GB

Component	Lattigo	E3	SHEEP	HEAX	F1	Medha	FHE C++ Transpiler
Space complexity	8 MB	8 MB	8 MB	8 MB	4 MB	8 MB	64 MB
Time complexity	log Q,log P	–	–	–	–	$R_{Q,2N} = Z_Q\ [X]/(x^n + 1)$	–
User friendly	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Availability	Apache 2.0	SHEEP server	GitHub	GitHub	GitHub	GitHub	Github
Open source	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Serialization	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Ciphertext size	64 bits	24 bits	10-128 bits	10-128 bits	10-128 bits	10-128 bits	256 bits
Symmetric	No	Yes	Yes	Yes	Yes	Yes	No
Multithreading	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Noise	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Relinearization	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Memory Req	30 MB	30 MB	30 MB	30 MB	32 MB	32 MB	6GB

Note:- λ: Security parameter, n: Some positive integer, R: Commutative ring with arbitrary element, P: Prime number, n: Computed element,t: Plaintext modulus, (P,Q): Set of private keys and Z: Integral field modulo

Table 6 Current versions of multiple tools

References	Tools	Version	Newest launch
[57]	SEAL	v4.1.1	2023
[58]	TFHE	v1.0.1	2017
[59]	HeLib	HeLib 2.2.2	2022
[60]	FV/NFLib	GPLv3	2016
[61]	Palisade	v1.11.7	2022
[62]	Concrete	python:v1.0.0	2022
[63]	Lattigo	v4.1.0	2022
[64]	E3	GPLv3	2022
[65]	SHEEP	MIT License	2018
[66]	HEAX	Basic	2009
[67]	F1	Basic	2009
[68]	Medha	CC BY 4.0	2022
[69]	FHE C++ Transpiler	GCC version 9	2017

for decryption. The cloud side is responsible for executing these high-level operations efficiently using the capabilities of Medha. By dividing the functionality between the user side and the cloud side, Medha enables secure and efficient computation on encrypted data, ensuring privacy and confidentiality in cloud-based environments [55, 56]. Lets discuss the levels of hierarchy:

1. **Highest level of hierarchy:** This level of hierarchy performs computations on ciphertext. The operations include key-switching, addition, subtraction, and multiplication. Further, these arithmetic operations are translated into polynomials.
2. **Middle level of hierarchy:** This level of hierarchy receives the operations that it has to convert into polynomial operations, such as coefficient-wise multiplication, polynomial addition, polynomial subtraction, coefficient-wise modular reduction polynomial multiplication, and coefficient-wise operation scalar multiplication.
3. **Lowest level of hierarchy:** The lowest level of hierarchy consists of modular arithmetic. It simply wraps around the values when they reach a certain value.

Conclusion

In the current era, as the sharing of data over the internet becomes increasingly necessary, ensuring data security against hackers has become a priority. Homomorphic encryption (HE) schemes have emerged as a more secure approach, as they allow data to remain encrypted while still being accessible to authorized users. Despite the availability of the technology, there are still challenges related to expensive and time-consuming legal procedures that aim to maintain strict privacy. However, there is optimism that practical homomorphic encryption will lead to a significant increase in applications in cloud and edge computation, where privacy is of utmost importance. Our objective is to share our experiences, inspire our peers, and contribute to the advancement of science and technology in society. We have focused on various libraries, frameworks, and accelerators based on different encryption schemes, providing comparisons to help users make informed decisions about their future implementations. When choosing a library, framework, or accelerator, it is crucial to consider factors such as limitations, diversity, and compatibility with the intended implementation. We have discussed these factors and provided insights into the capacities and limitations of the various libraries, frameworks, and accelerators we explored. Overall, our aim is to promote the adoption of secure data communication practices, facilitate informed decision-making, and contribute to the continuous advancement of technology in the field of homomorphic encryption.

Author Contributions All the authors contributed equally to this work.

Funding Information No funding was received for conducting this study.

Data Availability The data used to support the findings of this study are included within the article.

Declarations

Conflict of Interest The authors declare that there are no conflicts of interest regarding the publication of this paper.

References

1. Mahato GK, Chakraborty SK. A compartive review on homomorphic encryption for cloud security. IETE Journal of Research. Taylor and Francis. 2021;1-10.
2. Yousuf H, Lahzi M, Salloum SA, Shaalan K. Systematic review on fully homomorphic encryption scheme and its application. Recent Advances in Intelligent Systems and Smart Applications. 2021;537-551.
3. Dhiman S, Nayak S, Mahato GK, Ram A, Chakraborty SK. Homomorphic Encryption based Federated Learning for Financial Data Security. 4th International Conference on Computing and Communication Systems. IEEE. I3CS. 2023;1-6.
4. Acar A, Aksu H, Uluagac AS, Conti M. A survey on homomorphic encryption schemes: Theory and implementation. ACM Comput Surv. 2018;51(4):1–35.
5. Alloghani M, Alani MM, Al-Jumeily D, Baker T, Mustafina J, Hussain A, Aljaaf AJ. A systematic review on the status and progress of homomorphic encryption technologies. J Informn Security Appl. 2019;48: 102362.

6. Takeshita J, Koirala N, McKechney C, Jung T. HEProfiler: An In-Depth Profiler of Approximate Homomorphic Encryption Libraries; 2022.
7. Natarajan D, Dai W. SEAL-embedded: A homomorphic encryption library for the internet of things. IACR Transactions on Cryptographic Hardware and Embedded Systems. 2021;756-779.
8. Huang J, Wu D. Cloud Storage Model Based on the BGV Fully Homomorphic Encryption in the Blockchain Environment. Security and Communication Networks. 2022;2022.
9. Aydin F, Karabulut E, Potluri S, Alkim E, Aysu A. RevEAL: single-trace side-channel leakage of the SEAL homomorphic encryption library. In Design, Automation Test in Europe Conference & Exhibition. IEEE. 2022;1527-1532.
10. Lee E, Lee JW, Kim YS, no JS. Optimization of homomorphic comparison algorithm on rns-ckks scheme. IEEE Access. 2022;10:26163–76.
11. Chen H, Iliashenko I, Laine K. When heaan meets fv: a new somewhat homomorphic encryption with reduced memory overhead. In IMA International Conference on Cryptography and Coding (pp. 265-285). Springer, Cham; 2021.
12. Moon S, Lee Y. An efficient encrypted floating-point representation using HEaaN and TFHE. Security and Communication Networks; 2020.
13. Brenna L, Singh IS, Johansen HD, Johansen D. TFHE-rs: A library for safe and secure remote computing using fully homomorphic encryption and trusted execution environments. Array. 2022;13: 100118.
14. Jiang L, Lou Q, Joshi N. MATCHA: A Fast and Energy-Efficient Accelerator for Fully Homomorphic Encryption over the Torus. arXiv preprint arXiv:2202.08814; 2022.
15. Ferrara M, Tortora A. A CONCRETE approach to torus fully homomorphic encryption. Cryptology ePrint Archive; 2022.
16. Halevi S, Shoup V. Design and implementation of HElib: a homomorphic encryption library. Cryptology ePrint Archive; 2020.
17. github HElib. https://github.com/homenc/HElib. Accessed Sept; 2022.
18. Aguilar Melchor C, Kilijian MO, Lefebvre C, Ricosset T. A comparison of the homomorphic encryption libraries HElib, SEAL and FV-NFLlib. In International Conference on Security for Information Technology and Communications (pp. 425-442). Springer, Cham; 2018.
19. github FV-NFLlib. https://github.com/CryptoExperts/FV-NFLlib. Accessed Sept; 2022.
20. Halevi S, Polyakov Y, Shoup V. An improved RNS variant of the BFV homomorphic encryption scheme. In Cryptographers' Track at the RSA Conference (pp. 83-105). Springer, Cham; 2019.
21. github PALISADE lattice cryptography library. https://git.njit.edu/palisade/PALISADE. Accessed Sept; 2022.
22. Chillotti I, Gama N, Georgieva M, Izabachène M. TFHE: fast fully homomorphic encryption over the torus. J Cryptol. 2020;33(1):34–91.
23. github concrete. https://github.com/zama-ai/concrete-core. Accessed Sept; 2022.
24. Mouchet C, Bossuat JP, Troncoso-Pastoriza J, Hubaux JP. Lattigo: A multiparty homomorphic encryption library in go. In WAHC 2020-8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography; 2020.
25. Bajard JC, Eynard J, Hasan MA, Zucca V. A full RNS variant of FV like somewhat homomorphic encryption schemes. In International Conference on Selected Areas in Cryptography (pp. 423-442). Springer, Cham; 2016.
26. github lattigo. https://github.com/tuneinsight/lattigo. Accessed Sept; 2022.
27. github lattigo. https://pkg.go.dev/github.com/ldsec/lattigo/v2. Accessed Sept; 2022.
28. Gomathisankaran M, Tyagi A, Namuduri K. HORNS: A homomorphic encryption scheme for Cloud Computing using Residue Number System. In 2011 45th Annual Conference on Information Sciences and Systems. 2011;1-5. IEEE.
29. Ouyang Y, Rohde PP. A general framework for the composition of quantum homomorphic encryption & quantum error correction. arXiv preprint arXiv:2204.10471Xing, Bin Cedric, Mark Shanahan, and Rebekah Leslie-Hurd. "Intel® software guard extensions (Intel® SGX) software support for dynamic memory allocation inside an enclave." Proceedings of the Hardware and Architectural Support for Security and Privacy 2016 (2016): 1-9. (2022)
30. Chielle E, Mazonka O, Gamil H, Tsoutsos NG, Maniatakos M. E3: A framework for compiling C++ programs with encrypted operands. Cryptology ePrint Archive; 2018.
31. Brenner M, Dai W, Halevi S, Han K, Jalali A, Kim M, Sunar B. A standard API for RLWE-based homomorphic encryption. Homomorphic Encryption Standardization; 2017.
32. github E3. https://github.com/momalab/e3. Accessed Sept; 2022.
33. Chillotti I, Gama N, Georgieva M, Izabachene M. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In international conference on the theory and application of cryptology and information security. 2016;3-33. Springer, Berlin, Heidelberg.
34. Viand A, Jattke P, Hithnawi A. Sok: Fully homomorphic encryption compilers. In 2021 IEEE Symposium on Security and Privacy (SP) (pp. 1092-1108). IEEE; 2021.
35. github SHEEP. https://github.com/alan-turing institute/SHEEP. Accessed Sept; 2022.
36. Chowdhary S, Dai W, Laine K, Saarikivi O. EVA Improved: Compiler and Extension Library for CKKS. In Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography (pp. 43-55); 2021.
37. github IBM-FHE Toolkit. URL: https://www.ibm.com/blogs/research/2020/06/ibm-releases-fully-homomorphicencryption-toolkit-for-macos-and-ioslinux-and-android-coming-soon. Accessed Sept; 2022.
38. github IBM-FHE Toolkit. https://github.com/IBM/fhe-toolkit-linux. Accessed Sept; 2022.
39. github IBM-FHE Toolkit.https://www.ibm.com/blogs/research/2020/07/homomorphic-encryption-comes-to-linux-on-ibm-z/. Accessed Sept; 2022.
40. Kim S, Kim J, Kim MJ, Jung W, Kim J, Rhu M, Ahn JH. Bts: An accelerator for bootstrappable fully homomorphic encryption. In Proceedings of the 49th Annual International Symposium on Computer Architecture (pp. 711-725); 2022.
41. Migliore V, Real MM, Lapotre V, Tisserand A, Fontaine C, Gogniat G. Hardware/software co-design of an accelerator for FV homomorphic encryption scheme using Karatsuba algorithm. IEEE Trans Comput. 2016;67(3):335–47.
42. Roy SS, Mert AC, Kwon S, Shin Y, Yoo D. Accelerator for Computing on Encrypted Data. Cryptology ePrint Archive; 2021.
43. Zhang N, Gamil H, Brinich P, Reynwar B, Al Badawi A, Neda N, Franchetti F. Towards Full-Stack Acceleration for Fully Homomorphic Encryption; 2022.
44. Riazi MS, Laine K, Pelton B, Dai W. HEAX: An architecture for computing on encrypted data. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 1295-1309); 2020.
45. Han M, Zhu Y, Lou Q, Zhou Z, Guo S, Ju L. coxHE: A software-hardware co-design framework for FPGA acceleration of homomorphic computation. In 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE) (pp. 1353-1358). IEEE; 2022.

46. Al Badawi A, Veeravalli B, Mun CF, Aung KMM. High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA. IACR Transactions on Cryptographic Hardware and Embedded Systems, 70-95; 2018.
47. Bos JW, Lauter K, Loftus J, Naehrig M. Improved security for a ring-based fully homomorphic encryption scheme. In IMA International Conference on Cryptography and Coding (pp. 45-64). Springer, Berlin, Heidelberg; 2013.
48. Cheon JH, Kim A, Kim M, Song Y. Homomorphic encryption for arithmetic of approximate numbers. In International conference on the theory and application of cryptology and information security (pp. 409-437). Springer, Cham; 2017.
49. Samardzic N, Feldmann A, Krastev A, Devadas S, Dreslinski R, Peikert C, Sanchez D. F1: A fast and programmable accelerator for fully homomorphic encryption. In MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (pp. 238-252); 2021.
50. Feldmann A, Samardzic N, Krastev A, Devadas S, Dreslinski R, Peikert C, Sanchez D. F1: A fast and programmable accelerator for fully homomorphic encryption. In Proceedings of the 54th annual IEEE/ACM international symposium on Microarchitecture (MICRO-54); 2021.
51. Cheon JH, Kim A, Kim M, Song Y. Homomorphic encryption for arithmetic of approximate numbers. In International conference on the theory and application of cryptology and information security (pp. 409-437). Springer, Cham; 2017.
52. Mert AC, Kwon S, Shin Y, Yoo D, Lee Y, Roy SS. Medha: Microcoded Hardware Accelerator for computing on Encrypted Data. Cryptology ePrint Archive; 2022.
53. Roy SS, Mert AC, Kwon S, Shin Y, Yoo D. Accelerator for Computing on Encrypted Data. Cryptology ePrint Archive; 2021.
54. Takeshita J, Reis D, Gong T, Niemier M, Hu XS, Jung T. Algorithmic acceleration of B/FV-Like somewhat homomorphic encryption for compute-enabled RAM. In International Conference on Selected Areas in Cryptography (pp. 66-89). Springer, Cham; 2020.
55. Zhai Y, Ibrahim M, Qiu Y, Boemer F, Chen Z, Titov A, Lyashevsky A. Accelerating encrypted computing on intel gpus. In 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (pp. 705-716). IEEE; 2022.
56. Mahato GK, Chakraborty SK. Privacy Protection of Edge Computing Using Homomorphic Encryption. In Pattern Recognition and Data Analysis with Applications (pp. 395-407). Springer, Singapore; 2022.
57. github SEAL. https://github.com/microsoft/SEAL. Accessed May; 2023.
58. github TFHE. https://github.com/tfhe/tfhe. Accessed May; 2023.
59. github HElib. https://github.com/homenc/HElib. Accessed May; 2023.
60. github FV-NFLlib. https://github.com/CryptoExperts/FV-NFLlib/blob/master/LICENSE. Accessed May; 2023.
61. Palisade. https://palisade-crypto.org/. Accessed May; 2023.
62. github Concrete. https://github.com/zama-ai/concrete. Accessed May; 2023.
63. github Lattigo. https://github.com/tuneinsight/lattigo. Accessed May; 2023.
64. github E3. https://github.com/momalab/e3. Accessed May; 2023.
65. github SHEEP. https://github.com/alan-turing-institute/SHEEP. Accessed May; 2023.
66. Roy SS, Mert AC, Kwon S, Shin Y, Yoo D. Accelerator for computing on encrypted data. Cryptology. ePrint Archive; 2021.
67. Feldmann A, Samardzic N, Krastev A, Devadas S, Dreslinski R, Eldefrawy K, Genise N, Peikert C, Sanchez D. F1: A fast and programmable accelerator for fully homomorphic encryption (extended version). (arXiv preprint arXiv:2109.05371); 2021.
68. Mert AC, Kwon S, Shin Y, Yoo D, Lee Y, Roy SS. Medha: Microcoded hardware accelerator for computing on encrypted data. (arXiv preprint arXiv:2210.05476); 2022.
69. Gorantala S, Springer R, Purser-Haskell S, Lam W, Wilson R, Ali A, Astor EP, Zukerman I, Ruth S, Dibak C, Schoppmann P. A general purpose transpiler for fully homomorphic encryption. (arXiv preprint arXiv:2106.07893); 2021.
70. github Cingulata. https://github.com/CEA-LIST/Cingulata/wiki. Accessed August; 2023.
71. github FHE C++ transpiler. https://github.com/topics/transpiler?l=c%2B%2B. Accessed August; 2023.
72. Al Badawi A, Bates J, Bergamaschi F, Cousins DB, Erabelli S, Genise N, Halevi S, Hunt H, Kim A, Lee Y, Liu Z. Openfhe: Open-source fully homomorphic encryption library. Encrypted Computing and Applied Homomorphic Cryptography.(pp. 53-63); 2022.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.