Towards a *P*olynomial *I*nstruction Based *C*ompiler for Fully Homomorphic Encryption *A*ccelerators

Sejun Kim[*], Wen Wang[*], Duhyeong Kim[*]

Adish Vartak, Michael Steiner, Rosario Cammarota

Intel Labs, Intel Corporation

{firstname.lastname}@intel.com

ABSTRACT

Fully Homomorphic Encryption (FHE) is a transformative technology that enables computations on encrypted data without requiring decryption, promising enhanced data privacy. However, its adoption has been limited due to significant performance overheads. Recent advances include the proposal of domain-specific, highly-parallel hardware accelerators designed to overcome these limitations.

This paper introduces PICA, a comprehensive compiler framework designed to simplify the programming of these specialized FHE accelerators and integration with existing FHE libraries. PICA leverages a novel polynomial Instruction Set Architecture (p-ISA), which abstracts polynomial rings and their arithmetic operations, serving as a fundamental data type for the creation of compact, efficient code embracing high-level operations on polynomial rings, referred to as *kernels*, e.g., encompassing FHE primitives like arithmetic and ciphertext management. We detail a kernel generation framework that translates high-level FHE operations into pseudo-code using p-ISA, and a subsequent tracing framework that incorporates p-ISA functionalities and kernels into established FHE libraries. Additionally, we introduce a mapper to coordinate multiple FHE kernels for optimal application performance on targeted hardware accelerators. Our evaluations demonstrate PICA's efficacy in creation of compact and efficient code, when compared with an x64 architecture. Particularly in managing complex FHE operations such as relinearization, where we observe a 25.24x instruction count reduction even when a large batch size (8192) is taken into account.

1 INTRODUCTION

Data breaches on cloud servers pose a huge threat to users especially when their private data is outsourced. To ensure data privacy throughout the whole computation, i.e., while data is in storage, in transit, and in computation, we can deploy the Fully Homomorphic Encryption (FHE).

FHE is an increasingly popular technology [29] because it allows arbitrary computation of arbitrary complexity on encrypted data without decryption and returns results in encrypted format. The first FHE scheme was proposed in 2009 by Gentry [19]. Following Gentry's blueprint, most of the modern FHE schemes [8, 9, 11, 17] are constructed over an algebraic lattice and support homomorphic addition and multiplication over encrypted data as foundational operations. Typically, an FHE ciphertext is represented as the encryption of a vector of multiple word-size cleartexts, and the homomorphic operations are done component-wise. We call these FHE schemes word-wise FHE. A word-wise FHE ciphertext is composed of two large-degree polynomials (e.g., 16K∼64K degree) that embed the message vector and a certain amount of noise to ensure semantic security. The ciphertext noise grows as homomorphic operations are processed, but there exists an upper bound of the noise that guarantees the correct decryption. An expensive operation called bootstrapping is required to manage the noise not to exceed the upper bound, and hence it allows homomorphic computation of arbitrary functions. The large ciphertext size and the noise management procedure are responsible for the renowned computational overhead of FHE, which has been, and still is regarded as the main obstacle for the real-world adoption of FHE.

Admittedly, there also exists another approach to construct an efficient FHE scheme [13, 16] that supports bit-wise operations (e.g., NAND, XOR, etc.) and digital lookup table as basic operations and proceeds the bootstrapping procedure for every basic operation. These FHE schemes exploit relatively small-degree polynomials and include a much cheaper bootstrapping procedure than word-wise FHE, which results in much lower latency. However, a ciphertext of these FHE schemes is usually an encryption of a single bit or only a few bits. As a result, word-wise FHE still outperforms bit-wise FHE in terms of plaintext-ciphertext expansion ratio and amortized computational cost.

FHE Accelerators and Compilers. In order to reduce the latency of FHE schemes, research efforts from both academia and industry have been proposed, spanning different platforms such as CPU [3, 4], GPU [23, 33], and FPGA [5, 32]. More recently, driven by the DARPA Data Protection in Virtual Environments (DPRIVE) program [1], designing highly data and memory parallel ASICs for FHE has become a popular trend which is believed to be able to close the performance gap with cleartext programs.

Since the proposal of the first programmable FHE accelerator named F1 [30] in 2021, there has been a solid line of work (ARK [26], Medha [27], BASALISC [18], CraterLake [31], SHARP [25]) aiming to keep improving the performance and capabilities of the FHE accelerators. More recently, a 2.5D chiplet-based architecture REED [6] was proposed which demonstrates state-of-the-art performance for FHE acceleration.

[*]The first three authors contributed equally to this work.

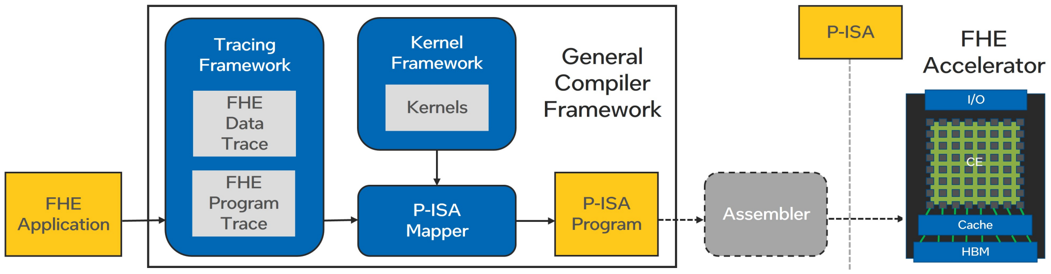

Figure 1: Flow of the PICA, the general compiler framework for FHE accelerators

With the availability of such FHE accelerators, a new research challenge emerges [28]: How can we easily and efficiently run FHE applications on these novel FHE accelerators? In most previous work on FHE accelerators [6, 18, 25–27, 30, 31], Domain-Specific Language (DSL) based approaches are proposed to develop customized compilers targeting the specific FHE accelerator. One representative example is CraterLake [31], the first FHE accelerator that runs FHE applications of unbounded multiplicative depth. In [31], a Python-embedded DSL is used to describe FHE applications. Relying on DSL certainly helps ease the development of FHE applications. However, this approach becomes challenging when being scaled to complex FHE applications since these DSLs are not compatible with state-of-the-art FHE libraries. This renders the validation of the DSL-based FHE applications difficult. Another constraint of the compilers proposed in previous work [6, 18, 25–27, 30, 31] is their customization for their targeted accelerator. To the best of our knowledge, none of the existing work presents extensive descriptions nor open-sourced code for readers to understand the details of their compilers.

These FHE accelerators (a sample diagram is shown on the right end of Figure 1) commonly share the following key elements: a compute engine (CE) containing highly parallel processing elements tailored for FHE arithmetic; a memory system with several hierarchies such as high-bandwidth main memory (HBM), Cache, and on-chip register files (RF); controllers managing instruction decoding, operations scheduling, memory transfers etc; as well as high-speed interface for communicating with the host. Observing these similarities in the architecture inspires us to design a general compiler framework that can be used for different FHE accelerators.

In this work, we propose PICA, a general polynomial instruction based compiler framework for FHE accelerators. Our compiler is built directly on mainstream FHE libraries such as SEAL [3] and OpenFHE [4]. PICA intends to serve as a solid basis for designing customized compiler targeting different hardware, including but not limited to the upcoming FHE accelerators. We will open-source the code of PICA soon. We summarize the contributions of our general compiler framework as follows:

- We propose a new polynomial Instruction Set Architecture (p-ISA) for FHE arithmetic, which defines a novel abstract interface to highly parallel FHE hardware accelerators.
- We present a kernel framework which can map each and every abstract high-level FHE operation into instruction sequences (a.k.a., pseudocode) written in p-ISA.
- We present a tracing framework built on top of modern open source FHE software libraries. Our tracing framework can validate the correctness of the kernel implementation.
- We present a p-ISA mapper which can construct FHE applications into a p-ISA program using the tracing framework, the kernel generator output, and a linking procedure.
- Finally, we present case studies and testing results on various parameter configurations and demonstrate the flexibility and efficiency of our compiler through the instruction count comparisons against the x64 architecture.

2 BACKGROUND

2.1 FHE Basics

A number of FHE schemes [8, 9, 11, 12, 15–17, 20] have been suggested following Gentry's blueprint [19], and the followings are regarded as the state-of-the-art FHE schemes with the best performance: BGV [9], BFV [8, 17], CKKS [11] and DM/CGGI [12, 16]. The word-wise FHE schemes BGV, BFV and CKKS commonly allow the batch encryption of multiple word-size plaintexts (e.g., $\mathbb{Z}_t^n$ for $t, n > 1$) into a single ciphertext and support homomorphic addition and multiplication as basic operations. To be precise, the homomorphism property can be described as follows:

$$\mathtt{Dec}(\mathtt{ADD}(\mathtt{Enc}(x_1, ..., x_n), \mathtt{Enc}(y_1, ..., y_n))) = (x_1 + y_1, ..., x_n + y_n),$$
$$\mathtt{Dec}(\mathtt{MULT}(\mathtt{Enc}(x_1, ..., x_n), \mathtt{Enc}(y_1, ..., y_n))) = (x_1 \cdot y_1, ..., x_n \cdot y_n).$$

Enc and Dec denote encryption and decryption algorithms, respectively. ADD and MULT indicate the homomorphic addition and multiplication algorithms of ciphertexts, respectively, which will be explained in detail later. Note that the equality in the homomorphism property needs to be replaced by the approximate equality in CKKS, since CKKS supports approximate computation over real/complex numbers, not exact computation.

The word-wise FHE schemes support parallel addition and multiplication homomorphically in a single instruction, multiple data (SIMD) manner. The SIMD property makes word-wise FHE schemes much more efficient in terms of amortized computational cost, when compared with bit-wise FHE schemes; and this enables a wider adoption of these schemes for real-world applications. For this reason, we mainly focus on word-wise FHE in the rest of the paper.

2.2 FHE Algorithms

Let $R := \mathbb{Z}[X]/(X^N+1)$ be the ring of integer polynomials modulo (X^N+1) for power-of-two N, and let $R_q := R/qR = \mathbb{Z}_q[X]/(X^N+1)$ for any integer $q > 1$. Fresh ciphertexts in BGV, BFV and CKKS commonly consist of two polynomials over R_Q for some large modulus Q, i.e., $(c_0, c_1) \in R_Q^2$, and it holds that

$$c_0 + c_1 \cdot s = M + E \pmod{Q}$$

for the secret key $s \in R$ with small coefficients, a message M and a small-coefficient noise polynomial E. Note that the message and noise are encoded properly for each scheme, e.g., $E = t \cdot e$ for small-coefficient polynomial e for the plaintext modulus t in BGV, $M = \lfloor Q/t \rceil \cdot m$ for $m \in R_t$ in BFV, etc.

Addition and Multiplication. The algorithms for homomorphic addition and multiplication are defined as follows:

$\underline{\texttt{ADD}(\texttt{ct}, \texttt{ct}')}$. For $\texttt{ct} = (c_0, c_1), \texttt{ct}' = (c_0', c_1') \in R_Q^2$, compute and output $(c_0 + c_0', c_1 + c_1') \pmod{Q} \in R_Q^2$.

$\underline{\texttt{MULT}(\texttt{ct}, \texttt{ct}')}$. For $\texttt{ct} = (c_0, c_1), \texttt{ct}' = (c_0', c_1') \in R_Q^2$, compute and output $(c_0 \cdot c_0', c_0 \cdot c_1' + c_1 \cdot c_0', c_1 \cdot c_1') \pmod{Q} \in R_Q^3$.

The homomorphic multiplication induces two main problems: The first is the noise increase, and the second is the increase of the ciphertext order (i.e., the number of polynomials). Each ciphertext internally contains a noise, and the decryption does not give a correct result if the noise becomes larger than a certain level. When two ciphertexts with the noise size $B > 0$ are homomorphically multiplied, then the noise of the resulting ciphertext is increased to $O(B^2)$. Therefore, the noise grows exponentially in terms of the multiplicative depth of the target computation.

Modulus-Switching (Rescaling). To control the noise growth from homomorphic multiplication, BGV and BFV schemes support an algorithm called modulus-switching. By switching the modulus from Q to a smaller modulus $Q' \approx Q/B$ via the operation

$$\texttt{ct} \pmod{Q} \mapsto \left\lfloor \frac{Q'}{Q} \cdot \texttt{ct} \right\rceil \pmod{Q'},$$

the noise size $O(B^2)$ after homomorphic multiplication is reduced to $O(Q'/Q \cdot B^2) = O(B)$, and hence the noise growth becomes linear. The modulus-switching algorithm is described as follow:

$\underline{\texttt{MODSWITCH}(\texttt{ct}; Q')}$. For $\texttt{ct} = (c_0, c_1) \in R_Q^2$, compute and output $\left(\left\lfloor \frac{Q'}{Q} \cdot c_0 \right\rceil, \left\lfloor \frac{Q'}{Q} \cdot c_1 \right\rceil\right) \pmod{Q'} \in R_{Q'}^2$.

Here, the entry-wise rounding operation $\lfloor \cdot \rceil$ is defined differently for each scheme. For BFV and CKKS, it is the same as the ordinary rounding operation that outputs the nearest integer of the input real number. For BGV, the mapping $a \mapsto \lfloor \frac{Q'}{Q} \cdot a \rceil$ output an integer polynomial a' satisfying $a' \equiv Q' \cdot Q^{-1} \cdot a \pmod{t}$ for the plaintext modulus t and is entry-wise close to $\frac{Q'}{Q} \cdot a$. Such a' can be obtained as $a' := (Q' \cdot a + \delta)/Q$ where $\delta := t \cdot (-t^{-1} \cdot Q' \cdot a \pmod{Q})$.

Note that the modulus-switching algorithm is called rescaling in CKKS (denoted by $\texttt{RESCALE}(\cdot)$), while the main purpose in CKKS is to control the plaintext size growth contrary to the other schemes.

Relinearization. The second problem is the increase of the ciphertext order from 2 to 3 after the homomorphic multiplication. To repeatedly apply homomorphic multiplication, we need a new algorithm that converts the order-3 ciphertext into an order-2 ciphertext, which is called relinearization [10]. The goal of relinearization is to convert an order-3 ciphertext $(c_0, c_1, c_2) \in R_Q^3$ satisfying

$$c_0 + c_1 \cdot s + c_2 \cdot s^2 = M + E \pmod{Q}$$

into an order-2 ciphertext $(c_0', c_1') \in R_q^2$ such that

$$c_0' + c_1' \cdot s = M + E' \pmod{Q},$$

where both E and E' are small-coefficient noise polynomials.

Assume that we have $(d_0, d_1) \in R_Q^2$ satisfying $d_0 + d_1 \cdot s = s^2 + E_{rln}$ for a small noise E_{rln}, then it holds that

$$(c_0 + c_2 \cdot d_0) + (c_1 + c_2 \cdot d_1) \cdot s = M + (E + c_2 \cdot E_{rln}) \pmod{Q}.$$

The left-hand side is now relinearized (in terms of s), but the second noise term $c_2 \cdot E_{rln}$ in the right-hand side is already of $O(Q)$ size, so this cannot be the solution.

To resolve the second-noise-term issue, we consider the decomposition of c_2 into $d > 1$ small-coefficient polynomials $c_2^{(i)} \in R_Q$ satisfying $c_2 = \sum_{i=0}^{d-1} g_i \cdot c_2^{(i)} \pmod{Q}$ for some public values $g_i \in \mathbb{Z}_Q$ for $0 \le i < d$. In addition, we prepare the encryption $(d_{0,i}, d_{1,i}) \in R_{PQ}^2$ of $P \cdot g_i \cdot s^2$ for some special modulus $P > 0$ such that $d_{0,i} + d_{1,i} \cdot s = P \cdot g_i \cdot s^2 + E_{rln,i} \pmod{PQ}$ for some small noise polynomials $E_{rln,i}$ for $0 \le i < d$. Then, we can easily check that

$$\left(c_0 + \left\lfloor \frac{1}{P} \cdot \sum_{i=0}^{d-1} c_2^{(i)} \cdot d_{0,i} \right\rceil\right) + \left(c_1 + \left\lfloor \frac{1}{P} \cdot \sum_{i=0}^{d-1} c_2^{(i)} \cdot d_{1,i} \right\rceil\right) \cdot s$$
$$= M + \left(E + \frac{1}{P} \sum_{i=0}^{d-1} c_2^{(i)} \cdot E_{rln}^{(i)} + \tau_0 + \tau_1 \cdot s\right),$$

where $\tau_0 := \left\lfloor \frac{1}{P} \cdot \sum_{i=0}^{d-1} c_2^{(i)} \cdot d_{0,i} \right\rceil - \frac{1}{P} \cdot \sum_{i=0}^{d-1} c_2^{(i)} \cdot d_{0,i}$ and $\tau_1 := \left\lfloor \frac{1}{P} \cdot \sum_{i=0}^{d-1} c_2^{(i)} \cdot d_{1,i} \right\rceil - \frac{1}{P} \cdot \sum_{i=0}^{d-1} c_2^{(i)} \cdot d_{1,i}$ are small-coefficient polynomials. With the decomposition technique, the second noise term is now replaced by the summation of small noises $c_2^{(i)} \cdot E_{rln,i}$ and $\tau_0 + \tau_1 \cdot s$, which can be easily controlled by the decomposition parameters d, g_i's, and the special modulus P. As a result, the relinearization algorithm is decribed as follow:

$\underline{\texttt{RELIN}(\texttt{ct}; \{\texttt{rlk}_i\}_{0 \le i < d})}$. For $\texttt{ct} = (c_0, c_1, c_2) \in R_Q^3$ and relinearization keys $\texttt{rlk}_i = (d_{0,i}, d_{1,i}) \in R_{PQ}^2$ for $0 \le i < d$, compute

$$c_0' := c_0 + \left\lfloor \frac{1}{P} \cdot \sum_{i=0}^{d-1} c_2^{(i)} \cdot d_{0,i} \right\rceil \text{ and } c_1' := c_1 + \left\lfloor \frac{1}{P} \cdot \sum_{i=0}^{d-1} c_2^{(i)} \cdot d_{1,i} \right\rceil,$$

and output $(c_0', c_1') \pmod{Q} \in R_Q^2$.

For BGV, the mapping $a \mapsto \lfloor \frac{1}{P} \cdot a \rceil$ outputs an integer polynomial a' which satisfies $a' \equiv P^{-1} \cdot a \pmod{t}$ for the plaintext modulus t and is entry-wise close to $\frac{1}{P} \cdot a$. Such a' can be obtained as $a' := (a + \delta)/P$ where $\delta := t \cdot (-t^{-1} \cdot a \pmod{P})$.

RNS-prime Decomposition. In state-of-the-art FHE libraries [2–4], residue number system (RNS) is generally used to represent each ciphertext with a large modulus Q for better performance. In the RNS implementation, each modulus Q is set to be a product of distinct primes $Q = q_0 q_1 \cdots q_\ell$. One of the simplest decomposition technique in relinearization that can be used in RNS representation is the RNS-prime decomposition, which sets $d = \ell + 1$, $c_2^{(i)} := c_2 \pmod{q_i}$, $g_i = \prod_{j \neq i} q_j \cdot \left((\prod_{j \neq i} q_j)^{-1} \pmod{q_i} \right)$ for $0 \leq i \leq \ell$, and let P be a prime of the size similar to each q_i.

Rotation. The word-wise FHE schemes also commonly support the homomorphic rotation of the plaintext vector for any rotation index k, i.e., $\mathsf{ROTATE}(\mathsf{Enc}(z_1, ..., z_n); k) = \mathsf{Enc}(z_{k+1}, z_{k+2}, ..., z_k)$. To be precise, the plaintext vector $\vec{z}$ is encoded into a polynomial $M = M(X)$ via ring isomorphism, and the entry rotation of $\vec{z}$ corresponds to the Galois automorphism $X \mapsto X^k$ on $M(X)$ for proper integer k. From the ciphertext $(c_0, c_1) \in R_Q^2$ satisfying $c_0(X) + c_1(X) \cdot s(X) = M(X) + E(X) \pmod{Q}$, we can directly obtain

$$c_0(X^k) + c_1(X^k) \cdot s(X^k) = M(X^k) + E(X^k) \pmod{Q}.$$

Similarly to relinearization, we bring the modified key $s(X^k)$ back to the original key $s(X)$ with the decomposition technique. As a result, the rotation algorithm is described as following:

$\underline{\mathsf{ROTATE}(\mathsf{ct}; k, \{\mathsf{rotk}_i\}_{0 \leq i < d})}$. For $\mathsf{ct} = (c_0, c_1) \in R_Q^2$ and rotation keys $\mathsf{rotk}_i = (d_{0,i}, d_{1,i}) \in R_{PQ}^2$ for $0 \leq i < d$, compute

$$\begin{aligned} c_0' &:= c_0(X^k) + \left\lfloor \frac{1}{P} \cdot \sum_{i=0}^{d-1} c_1(X^k)^{(i)} \cdot d_{0,i} \right\rceil, \text{ and} \\ c_1' &:= \left\lfloor \frac{1}{P} \cdot \sum_{i=0}^{d-1} c_1(X^k)^{(i)} \cdot d_{1,i} \right\rceil, \end{aligned}$$

and output $(c_0', c_1') \pmod{Q} \in R_Q^2$.

Bootstrapping. By definition, bootstrapping is a homomorphic evaluation of the decryption circuit [19], which refreshes the internal noise and the modulus of a ciphertext. Technically, the bootstrapping algorithm is a composition of basic FHE operations described above including homomorphic addition, multiplication, modulus-switching, relinearization and rotation. Hence, the acceleration of these basic operations directly implies the acceleration of bootstrapping, which is the main computational bottleneck of FHE.

2.3 FHE Implementation in Double-CRT Format

As noted in the previous subsection, the state-of-the-art FHE libraries [2–4] exploit the RNS representation of each ciphertext to enhance the performance by avoiding the multi-precision arithmetic modulo Q, which is generally hundreds of bits. Moreover, since every ciphertext consists of several polynomials, we utilize a ring isomorphism called Number Theoretical Tranform (NTT) for more efficient polynomial arithmetic. For any prime $q \equiv 1 \pmod{2N}$, NTT modulo q, denoted by $\mathsf{NTT}_q(\cdot)$, maps a polynomial $a \in R_q$ to an N-dimensional vector $\mathsf{NTT}_q(a) \in \mathbb{Z}_q^N$, while preserving addition and multiplication. Namely, it holds that $\mathsf{NTT}_q(a+b) = \mathsf{NTT}_q(a) + \mathsf{NTT}_q(b) \pmod{q}$ and $\mathsf{NTT}_q(a \cdot b) = \mathsf{NTT}_q(a) \odot \mathsf{NTT}_q(b) \pmod{q}$ where $\odot$ is the entry-wise multiplication of vectors.

To apply both RNS and NTT, the ciphertext modulus Q is chosen as a product of distinct primes $Q = q_0 q_1 \cdots q_\ell$ such that $q_i \equiv 1 \pmod{2N}$. Here, we call each prime q_i an RNS prime and ℓ the ciphertext level. Then, each polynomial a in R_Q corresponds to $(\ell + 1)$ small polynomials $a^{(i)} := a \pmod{q_i} \in R_{q_i}$ for $0 \leq i \leq \ell$. Each small polynomial $a^{(i)} \in R_{q_i}$ has one-to-one correspondence with an N-dimensional vector $\mathsf{NTT}_{q_i}(a^{(i)}) \in \mathbb{Z}_{q_i}^N$. Therefore, the large-coefficient polynomial $a \in R_Q$ can be represented as an $(\ell+1)$-tuple of N-dimensional vectors $\mathsf{NTT}_{q_i}(a^{(i)}) \in \mathbb{Z}_{q_i}^N$ for $0 \leq i \leq \ell$, which is called the double-CRT format [22].

In the rest of the paper, we use the double-CRT format as the default format of a ciphertext. We say that a polynomial is in normal domain if the polynomial is in RNS representation but NTT is not applied. Otherwise, if a polynomial is in the double-CRT format, then we say that the polynomial is in NTT domain.

3 COMPILER FRAMEWORK FOR FHE ACCELERATORS

Our primary object is to design a general compiler framework which automatically maps an FHE application to the FHE accelerators. Note that automatic generation of the FHE application is beyond the scope of this work, interested readers can refer to the state-of-art FHE compilers work such as HEIR [7] which automates code generation over multiple FHE schemes. Another note to make is that this work mainly focuses on the compiler support for the word-wise FHE schemes including BGV and CKKS. Note that the BFV scheme is equivalent to BGV in proper parameter setting (e.g., $Q \equiv \pm 1 \bmod t$), and BFV algorithms are identical to those of BGV and/or CKKS. For the other line of FHE schemes including DM/CGGI and their compiler support, refer to ArctyrEX [21] for details.

Figure 1 presents the high-level flow of PICA. An FHE application is fed as input and parsed through the tracing framework to generate a data trace as well as a program trace. In the offline phase, kernels for each of the FHE operations are generated. Next, traces together with kernels are sent to the p-ISA mapper which stitches kernels together according to the trace information. In the end, our compiler framework generates a p-ISA program which can be easily validated by a simulator and then run directly on the FHE accelerator.

3.1 p-ISA

As noted in Section 2.1, FHE arithmetic heavily relies on polynomial operations. Therefore, we first propose new a polynomial Instruction Set Architecture (p-ISA) to serve as the interface between the software front-end and the hardware back-end.

We define p-ISA following a new native polynomial data type. Each operand in the instruction represents a fixed-size polynomial which contains b coefficients, for the batch size b determined by the parallelism offered by the FHE accelerator. Each coefficient is computed modulo a given RNS prime q. Each instruction then carries out computations on these polynomial operands following the descriptions in Table 1. It is easy to observe that p-ISA only contains a reduced set of polynomial instructions that are required as basic operations in implementing complex high-level FHE operations.

Table 1: List of instructions contained in the Polynomial Instruction Set Architecture (p-ISA)

Instruction	Operands	Details
`padd`	d, c, c', q	Add two input polynomials c, c' modulo q and store the result d at a new address
`psub`	d, c, c', q	Subtract two input polynomials c, c' modulo q and store the result d at a new address
`pmul`	d, c, c', q	Multiply two input polynomials c, c' in NTT domain entry-wise modulo q and store the result d at a new address
`pmuli`	d, c, imm, q	Scale an input polynomial c by an immediate imm modulo q and store the result d at a new address
`pmac`	d, c, c', q	Multiply two polynomials c, c' in NTT domain entry-wise modulo q and add the result into an accumulator polynomial d
`pmaci`	d, c, imm, q	Scale an input polynomial c by an immediate imm modulo q and add the result into an accumulator polynomial d
`pntt`	d, c, q	Perform forward NTT on an input polynomial c in normal domain modulo q and store the result d at a new address
`pintt`	d, c, q	Perform inverse NTT on an input polynomial c in NTT domain modulo q and store the result d at a new address

For multiply-related instructions (i.e., `pmul`, `pmuli`, `pmac`, `pmaci`), as explained in Section 2.3, we use the double-CRT format of the polynomial as the default format. This representation simplifies the definition of these instructions. For example, the polynomial multiplication (`pmul`) is naturally defined as the entry-wise multiplication of two input polynomials. We also define instructions `pntt` and `pintt` for forward and inverse NTT transformations. On the other hand, addition/subtraction of polynomials is domain independent.

Note that the p-ISA presented in Table 1 only intends to serve as a basis for development of FHE accelerators and compilers. Users can easily expand the p-ISA to exploit the functionalities provided by the hardware accelerator. It is equally important to note that the p-ISA only contains compute instructions. Depending on the specific design of the memory system in the accelerator, memory instructions need to be defined accordingly. An accelerator-specific assembler (the dotted box in Figure 1) will insert these memory instructions between compute instructions to make sure that operands are moved to the right places before and after computation.

3.2 Kernels

Kernels consist of a pre-compiled library where each of the FHE operations is converted into a separate p-ISA program. Our kernel generation framework supports a wide range of configurations. The user can first pick an FHE scheme, for example CKKS or BGV. Next, users choose the target security level and the multiplicative depth, which automatically determines the FHE parameters including the ring dimension N and maximum ciphertext modulus $Q_{max} = q_0 q_1 \cdots q_{\ell_{max}}$. Moreover, the kernel framework can take into account the parallelism offered by the FHE accelerator, namely the batch size b. For example, a polynomial $c_r^{(i)}$ of the large ring dimension N can be partitioned into $\frac{N}{b}$ smaller ones, namely $c_r^{(i)(j)}$ for $0 \le j < \frac{N}{b}$, each of size b in order to fit on the accelerator. Then, the entry-wise addition and multiplication on each small-degree polynomial $c_r^{(i)(j)}$ naturally correspond to those on $c_r^{(i)}$, regardless of how the partition of $c_r^{(i)}$ into $c_r^{(i)(j)}$s has been done.

There are various ways to compute N-dimensional NTT through b-dimensional NTT operations that would be supportable in the FHE accelerator with the batch size b. For example, when we partition $c_r^{(i)} := \sum_{k=0}^{N-1} f_k \cdot X^k$ for $f_k \in \mathbb{Z}_{q_i}$ into $c_r^{(i)(j)} := \sum_{k=0}^{b-1} f_{\frac{N}{b}k+j} \cdot X^{\frac{N}{b}k}$ so that $c_r^{(i)} = \sum_{j=0}^{\frac{N}{b}-1} X^j \cdot c_r^{(i)(j)}$, then it holds that $\mathsf{NTT}_{q_i}(c_r^{(i)}) = \sum_{j=0}^{\frac{N}{b}-1} \mathsf{NTT}_{q_i}(X^j) \odot \mathsf{NTT}_{q_i}(c_r^{(i)(j)})$, where $\odot$ denotes the entry-wise multiplication. Note that $\mathsf{NTT}_{q_i}(c_r^{(i)(j)})$ can actually be obtained through b-dimensional NTT since $c_r^{(i)(j)}$ is contained in the b-dimensional polynomial ring $\mathbb{Z}_{q_i}[Y]/(Y^b + 1)$ for $Y := X^{\frac{N}{b}}$. Moreover, $\mathsf{NTT}_{q_i}(X^j)$ is pre-computable for all i and j. Hence, $\mathsf{NTT}_{q_i}(c_r^{(i)})$ can be obtained with only use of b-dimensional NTT.

In the end, the user can choose the FHE operation from a wide range of choices, as listed in Table 2. Given the configuration, our kernel generation framework can then automatically generate a kernel which describes the algorithmic realization of the FHE operation step by step, using p-ISA defined in Section 3.1. In Table 2, the kernels are classified into two types: Arithmetic and Adjustment. The "Arithmetic" type includes the FHE operations supporting homomorphic arithmetic on encrypted plaintext vectors, including entry-wise addition (ADD, ADD_PLAIN, ADD_CONST), entry-wise multiplication (MUL, MUL_PLAIN, MUL_CONST, SQUARE) and the vector rotation (ROTATE). The other operations MOD_SWITCH, RESCALE, and RELIN in the "Adjustment" type do not change the internal plaintext of the ciphertext but control the noise growth (Refer to Section 2.2 for details). Note that the kernels listed in Table 2 are commonly shared by both BGV and CKKS, except MOD_SWITCH which is used for the level adjustment in BGV and the equivalent operation RESCALE used in CKKS. Since bootstrapping is a sequence

Table 2: List of kernels for CKKS and BGV schemes

FHE Ops	Type	Details
ADD	Arithmetic	Addition of two ctxts
ADD_PLAIN	Arithmetic	Addition of a ctxt with a ptxt
ADD_CONST	Arithmetic	Addition with a constant
SUB	Arithmetic	Subtraction of two ctxts
SUB_PLAIN	Arithmetic	Subtraction of a ctxt with a ptxt
SUB_CONST	Arithmetic	Subtraction with a constant
MUL	Arithmetic	Multiplication of two ctxts
MUL_PLAIN	Arithmetic	Multiplication of a ctxt with a ptxt
MUL_CONST	Arithmetic	Multiplication of a ctxt with a const
SQUARE	Arithmetic	Squaring a ctxt
ROTATE	Arithmetic	Ctxt rotation
MOD_SWITCH*	Adjustment	Ctxt modulus-switching from q to q'
RESCALE*	Adjustment	Ctxt rescaling from q to q'
RELIN	Adjustment	Ctxt relinearization from order 3 to 2

Algorithm 1 Kernel for Homomorphic Multiplication

```
Require: Input ciphertexts $(c_0, c_1) \in R_Q^2$ and $(c'_0, c'_1) \in R_Q^2$ at level $\ell$ for $Q = \prod_{i=0}^{\ell} q_i$.
Ensure: Output ciphertext $(d_0, d_1, d_2) = (c_0 \cdot c'_0, c_0 \cdot c'_1 + c_1 \cdot c'_0, c_1 \cdot c'_1) \in R_Q^3$
 1: for $i$ in range $(\ell + 1)$ do
 2:     pmul, $d_0^{(i)}, c_0^{(i)}, c_0^{\prime(i)}, q_i$                 ▷ $d_0 = c_0 \cdot c'_0 \pmod{Q}$
 3:     pmul, $d_1^{(i)}, c_0^{(i)}, c_1^{\prime(i)}, q_i$
 4:     pmac, $d_1^{(i)}, c_1^{(i)}, c_0^{\prime(i)}, q_i$                 ▷ $d_1 = c_0 \cdot c'_1 + c_1 \cdot c'_0 \pmod{Q}$
 5:     pmul, $d_2^{(i)}, c_1^{(i)}, c_1^{\prime(i)}, q_i$                 ▷ $d_2 = c_1 \cdot c'_1 \pmod{Q}$
 6: end for
```

Algorithm 2 Kernel for BGV Relinearization (with RNS-prime decomposition)

```
Require: Input ciphertext $(c_0, c_1, c_2) \in R_Q^3$ at level $\ell (\leq \ell_{max})$ for $Q = \prod_{i=0}^{\ell} q_i$, relinearization keys $(d_{0,i}, d_{1,i}) \in R_{PQ_{max}}^2$ for $0 \leq i \leq \ell$ for $Q_{max} = \prod_{i=0}^{\ell_{max}} q_i$ and a special prime $P = q_{\ell_{max}+1}$, and metadata $t$, $[-t^{-1}]_P$ and $[P^{-1}]_{q_i}$ for $0 \leq i \leq \ell$.
Ensure: Output ciphertext $(c'_0, c'_1) \in R_Q^2$
 1: for $i$ in range $(\ell + 1)$ do
 2:     pintt, $ct^{(i)}, c_2^{(i)}, q_i$                          ▷ Input ciphertext is in dual-CRT format
 3:     for $j$ in range $(\ell + 2)$ do
 4:         $idx = \ell_{max} + 1$ if $j == (\ell + 1)$ else $j$
 5:         pntt, $ct'^{(j)}, ct^{(i)}, q_{idx}$                   ▷ Basis extension from $Q = q_0 q_1 \cdots q_\ell$ to $PQ$
 6:         for $r$ in range 2 do
 7:             inst = pmul if $i == 0$ else pmac
 8:             inst, $ct_r''^{(j)}, ct'^{(j)}, d_{r,i}^{(idx)}, q_{idx}$   ▷ Inner product $ct''_r = \sum_{i=0}^{\ell} c_2^{(i)} \cdot d_{r,i} \pmod{PQ}$
 9:         end for
10:     end for
11: end for
12: for $r$ in range 2 do
13:     pintt, $\delta_r, ct_r''^{(\ell+1)}, q_{\ell_{max}+1}$
14:     pmuli, $\delta_r, \delta_r, [-t^{-1}]_P, q_{\ell_{max}+1}$      ▷ Compute $\delta = -t^{-1} \cdot ct''_r \pmod{P}$
15:     for $j$ in range $(\ell + 1)$ do
16:         pntt, $c_r'^{(j)}, \delta_r, q_j$
17:         pmuli, $c_r'^{(j)}, c_r'^{(j)}, t, q_j$                ▷ Finish computing $\delta = t \cdot (-t^{-1} \cdot ct''_r \pmod{P})$
18:         padd, $c_r'^{(j)}, c_r'^{(j)}, ct_r''^{(j)}, q_j$       ▷ Add $\delta$ to $ct''_r$ to make it divisible by $P$
19:         pmuli, $c_r'^{(j)}, c_r'^{(j)}, [P^{-1}]_{q_j}, q_j$     ▷ Divide by $P$
20:         padd, $c_r'^{(j)}, c_r'^{(j)}, c_r^{(j)}, q_j$          ▷ Compute the final result
21:     end for
22: end for
```

of the basic FHE operations including homomorphic addition, multiplication, modulus-switching, and rotation, bootstrapping can be fully simulated with the kernels listed in Table 2.

We present pseudocodes for generating two kernels as examples: Homomorphic multiplication (Algorithm 1), and BGV relinearization (Algorithm 2). For simplicity, we assume that $b = N$. Refer to Section 2.2 for more details on the underlying math formulas.

3.3 Tracing Framework

We developed a tracing framework for the purpose of validating the correctness of kernel implementations by utilizing open source FHE libraries. The tracing framework is designed to be embedded directly into FHE libraries, such that inputs, outputs, and any additional context parameters for FHE computation can be intercepted.

The main functionality of the tracing framework is collecting the *FHE Data Trace* and *FHE Program Trace*, as shown in Figure 1. The *FHE Program Trace* module logs all the processed FHE operations, while executing an FHE application, such as operation names, symbols of inputs and outputs. A trace example of performing an FHE multiply with relinearization, $ct_z = \texttt{RELIN}(\texttt{MULT}(ct_x, ct_y))$, is shown in Figure 2. Alongside the symbol names of inputs and outputs, and kernel name, the tracing framework collects a list of parameters such as the number of RNS primes, the ciphertext order and level, as well as the multiplicative depth of every traced object.

All inputs and outputs as well as context parameters are collected by the *FHE Data Trace* module. The data is composed of two base fields, the metadata and data polynomials. Metadata contains all the pre-computable constant parameters and polynomials required for handling the p-ISA instructions, while data polynomials are the collection of coefficient vectors for all inputs and outputs, corresponding with the FHE program trace.

Another feature of the tracing framework is to emulate the intermediate representation of the polynomial objects, by maintaining

```
0:CKKS,16384,MULT,ct01f9,8,3,2,2,ct018d,8,2,1,2,ct01ba,8,2,1,2,
1:CKKS,16384,RELIN,ct0298,8,2,2,2,ct01f9,8,3,2,2,
```

Figure 2: Trace of Homomorphic Multiplication with Relinearization. Input ciphertexts (at address ct018d and ct01ba) with ring dimension 16384, 8 RNS primes, order 2, depth 1, and level 2, of `MULT` generate a ciphertext (at address ct01f9) with order 3, depth 2, level 2, and it is relinearized into a ciphertext (at address ct0298) with order 2 through `RELIN`.

static single-assignment (SSA) representations [14] of the polynomial objects. The SSA manager applies the renaming scheme to replace the symbol of output objects whenever it is a duplicate of one or more input objects while the tracer framework logs each FHE computation activity. This feature works jointly with an SSA graph generator in the p-ISA mapper (in the next section), in order to optimally compile an FHE application into a p-ISA program.

3.4 p-ISA mapper

The p-ISA mapper generates the p-ISA program using tracing framework output and the kernel generator. One required pre-processing step is flattening the context keys and data into kernel compatible format, which is handled in the tracing framework. As shown in a sample homomorphic multiplication kernel in Algorithm 1, ciphertext objects need to be flattened in the form of multiple coefficient vectors such as $ct\_r\_i\_k$ for $r \in [0, R)$, $i \in [0, \ell]$ and $k \in [0, N/b)$ where R, ℓ and b denote the ciphertext order, level and the batch size of the FHE accelerator, respectively. The mapping process matches the kernel generator outputs with flattened symbol names, and allocates temporary outputs with intermediate symbols. For example, from the BGV relinearization kernel as shown in Algorithm 2, operands ct, ct', ct'' and δ are automatically recognized and treated as temporary outputs, which will later be replaced with intermediate symbols.

During the mapping process, the SSA graph generator builds an abstract representation of procedures to build a dependency graph and identify duplicate instructions. The execution order is adjusted accordingly to achieve overall optimal performance.

4 VALIDATION AND PERFORMANCE

We use the OpenFHE library [4] to validate the correctness of the kernels and estimate the performance compared to CPU architectures. In OpenFHE-CKKS, ciphertext bookkeeping operations such as level/depth matching for input ciphertexts/plaintexts are handled beneath the context evaluator layer. Hence the calls to log the traces and polynomials are inserted directly into the OpenFHE source code. For instance, in order to correctly trace the polynomial coefficients in homomorphic addition, instead of tracing the application-level function `EvalAdd(·)`, the tracing command is directly added inside of the function (See Figure 3).

We set the p-ISA batch size $b = 8192$ and tested three different ring dimensions N: 16384, 32768 and 65536. The ciphertext order is always set to 2 for both inputs and outputs, except for multiply operations (`MUL`, `MUL_PLAIN`, `MUL_CONST` and `SQUARE`) and relinearization.

```
// openfhe-development/src/pke/lib/schemebase/base-
       leveledshe.cpp#L613
template <class Element>
void LeveledSHEBase<Element>::EvalAddCoreInPlace(
    Ciphertext<Element>& ciphertext1,
    ConstCiphertext<Element> ciphertext2) const {
    // Tracer: temporary input to store ciphertext1
    auto tmp_ctxt1 = ciphertext1->Clone();

    std::vector<Element>& cv1 = ciphertext1->GetElements
         ();
    const std::vector<Element>& cv2 = ciphertext2->
         GetElements();

    size_t c1Size = cv1.size();
    size_t c2Size = cv2.size();
    size_t cSmallSize = std::min(c1Size, c2Size);

    // profile polynomial addition portion
    CALLGRIND_TOGGLE_COLLECT;
    for (size_t i = 0; i < cSmallSize; i++) {
        cv1[i] += cv2[i];
    }
    CALLGRIND_TOGGLE_COLLECT;

    if (c1Size < c2Size) {
        cv1.reserve(c2Size);
        for (size_t i = c1Size; i < c2Size; i++) {
            cv1.emplace_back(cv2[i]);
        }
    }
    // call tracing framework
    // op name, output, inputs...
    PISA_TRACER("ADD", ciphertext1, tmp_ctxt1,
         ciphertext2);
}
```

Figure 3: OpenFHE `EvalAdd(·)` with Tracing Command

We set the maximal ciphertext level $\ell_{max} = 9$ and the current ciphertext level $\ell = 7$. Due to the randomness of the RNS polynomial coefficients, the test has been iterated 50 times. The tracer plugins are inserted into 13 FHE operations (except `MODSWITCH`) in Table 2.

4.1 Simulator

We implemented a simulator which models the functional behavior of the instructions defined in Table 1. Our simulator easily validates the functional correctness of a p-ISA program by using the data trace collected from the tracing framework. First, the corresponding polynomial inputs and metadata are fed as inputs. Next, the simulator runs a sequence of instructions following the p-ISA program which produces polynomial outputs. These outputs are compared with the reference outputs extracted from the tracing step, and this completes the functional validation for a given p-ISA program.

4.2 Comparison of Instruction Counts

In this section, the efficacy of PICA in generation of compact and efficient code is demonstrated through the comparison of instruction counts against CPU, including the essential I/O instructions. As FHE computation suffers from runtime overhead of up to 30,000x [24] on CPU, a great reduction in instruction counts achieved with PICA can bring benefits such as performance speedup, increased throughput and lower power consumption compared to CPU. Note that our evaluation is done with no parallelization enabled on the CPU side. For CPU instruction analysis, the benchmark was performed

Table 3: Average instruction count of HE kernels on CPU and PICA

HE Kernel	N	CPU	PICA
ADD	16k	3,149,183	141
	32k	6,294,911	269
	64k	12,586,367	525
ADD_PLAIN	16k	1,574,576	125
	32k	3,147,440	237
	64k	6,293,168	461
ADD_CONST	16k	2,251,479	125
	32k	4,481,215	237
	64k	8,935,900	461
MUL	16k	16,747,977	189
	32k	33,463,311	365
	64k	66,886,536	717
MUL_PLAIN	16k	7,082,123	125
	32k	14,160,011	237
	64k	28,315,787	461
MUL_CONST	16k	6,328,948	125
	32k	12,624,926	237
	64k	25,207,831	461
SUB	16k	3,018,156.5	141
	32k	6,032,880.3	269
	64k	12,062,104.1	525
SUB_PLAIN	16k	1,509,001.7	125
	32k	3,016,432.3	237
	64k	6,030,981.4	461
SUB_CONST	16k	3,030,957.4	125
	32k	6,052,661.7	237
	64k	12,081,265.6	461
SQUARE	16k	13,537,590	205
	32k	27,037,459	397
	64k	54,037,662	781
RESCALE	16k	138,324,270.5	855
	32k	290,160,708.7	1,757
	64k	606,322,450.9	3,621
RELIN	16k	333,250,917	1,629
	32k	687,863,031	3,341
	64k	1,418,840,126.7	6,861
ROTATE	16k	318,364,826.5	2,157
	32k	657,951,529.8	4,461
	64k	1,359,094,767.4	9,229

on Intel Xeon®Platinum 8360Y 2.4GHz processor with 1024GB of RAM and 72 cores, running Ubuntu 22.04. We used *valgrind* to collect instructions generated for every traced FHE operation, and OpenFHE was compiled with *gcc-11.4* using the $-O3$ flag. The collection toggle macro, *CALLGRIND_TOGGLE_COLLECT*, was added to where the actual polynomial computation occurs, as shown in the code example in Figure 3.

For memory mapping, we defined two additional instructions, `dload` and `dstore`. Insertion of these memory instructions will be handled by an accelerator-specific assembler. The context parameters such as NTT twiddle factors and roots of unity of each RNS modulus for keyswitching operations, are loaded in the memory with `dload`. For polynomials, the inputs and outputs are loaded and stored with the keyword `poly`, using `dload` and `dstore`, respectively. Temporary inputs and outputs computed within kernels are directly linked to intermediate registers, thus explicit I/O instructions are not required.

Table 4: Average instruction count on CPU divided by p-ISA batch size alongside PICA for $N = 64k$

HE op	CPU	PICA
ADD	1,536.4 (2.93x)	525
ADD_PLAIN	768.2 (1.67x)	461
ADD_CONST	1,090.8 (2.37x)	461
MUL	8,164.9 (11.39x)	717
MUL_PLAIN	3,456.5 (7.50x)	461
MUL_CONST	3,077.1 (6.67x)	461
SUB	1,472.4 (2.80x)	525
SUB_PLAIN	736.2 (1.60x)	461
SUB_CONST	1,474.8 (3.20x)	461
SQUARE	6,596.4 (8.45x)	781
RESCALE	74,014.0 (20.44x)	3,621
RELIN	173,198.3 (25.24x)	6,861
ROTATE	165,905.1 (17.98x)	9,229
BOOTSTRAP	**91,602,252.9 (6.28x)**	**14,577,494**

From Table 3, we can observe that PICA demonstrates a huge reduction in instruction count. The main reason is that the FHE operations in x64 compute each coefficient individually, while PICA compute a vector of coefficients parallelly in SIMD manner. The results show that the number of instructions for both CPU architectures is larger than that from p-ISA, by at minimum, $11,545.46$ times. For a fair comparison, we divided the instruction counts from CPU execution by the batch size $b = 8192$ to estimate the number of instructions required per batch, as shown in Table 4. We observe that 1) "Arithmetic" type kernels are relatively cheaper, and 2) PICA substantially outperforms CPU, in terms of instruction efficiency, for "Adjustment" type kernels (up to 25.24x gap), which require many NTTs and INTTs. Even with a hypothetical CISC/RISC high-end processor capable of handling a vector of width 8192 in parallel, our framework still outperforms by over 25x in terms of efficiency in compact code generation.

For validation of bootstrapping capabilities, the performance of a single iteration of OpenFHE-CKKS bootstrapping has been measured on both CPU platform and PICA. We set $N = 65536$, $\ell_{max} = 38$, and use the same batch size ($b = 8192$) for testing. On average, the total instruction count for CPU was 750.4 billion, while PICA generated 14.58 million instructions. With $1/b$ applied to CPU instruction count, PICA results in the instruction efficiency of 6.28x.

5 CONCLUSION

We propose PICA, a general polynomial instruction based compiler framework for FHE accelerators. Combining a novel p-ISA, a kernel generation framework, a tracing framework, as well as a p-ISA mapper, PICA can automatically convert and run an FHE program on the back end hardware accelerators. PICA demonstrates its usability and efficiency through comparison with batched instruction counts on CPU platforms. PICA will be immediately accessible to the general public through open source code release. Our work will serve as great groundwork especially for those interested in designing an efficient compiler for their customized FHE accelerators.

REFERENCES

[1] 2021. Data Protection in Virtual Environments (DPRIVE). https://www.darpa.mil/program/data-protection-in-virtual-environments. Accessed on 2024-04-17.

[2] 2023. Lattigo v5. Online: https://github.com/tuneinsight/lattigo. EPFL-LDS, Tune Insight SA.

[3] 2023. Microsoft SEAL. https://github.com/microsoft/SEAL. Accessed on 2024-04-17.

[4] 2024. OpenFHE - Open-Source Fully Homomorphic Encryption Library. https://github.com/openfheorg/openfhe-development. Accessed on 2024-04-17.

[5] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Yazicigil, Anantha Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. 2023. FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 882–895.

[6] Aikata Aikata, Ahmet Can Mert, Sunmin Kwon, Maxim Deryabin, and Sujoy Sinha Roy. 2023. Reed: Chiplet-based scalable hardware accelerator for fully homomorphic encryption. *arXiv preprint arXiv:2308.02885* (2023).

[7] Song Bian, Zian Zhao, Zhou Zhang, Ran Mao, Kohei Suenaga, Yier Jin, Zhenyu Guan, and Jianwei Liu. 2023. HEIR: A Unified Representation for Cross-Scheme Compilation of Fully Homomorphic Computation. *Cryptology ePrint Archive* (2023).

[8] Zvika Brakerski. 2012. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *Advances in Cryptology–CRYPTO 2012*. Springer, 868–886.

[9] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) fully homomorphic encryption without bootstrapping. In *Proc. of ITCS*. ACM, 309–325.

[10] Zvika Brakerski and Vinod Vaikuntanathan. 2014. Efficient fully homomorphic encryption from (standard) LWE. *SIAM Journal on computing* 43, 2 (2014), 831–871.

[11] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 409–437.

[12] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2016. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 3–33.

[13] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology* 33, 1 (2020), 34–91.

[14] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.

[15] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. 2010. Fully Homomorphic Encryption over the Integers. In *Advances in Cryptology–EUROCRYPT 2010*. Springer, 24–43.

[16] Léo Ducas and Daniele Micciancio. 2015. FHEW: bootstrapping homomorphic encryption in less than a second. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 617–640.

[17] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptology ePrint Archive* 2012 (2012), 144.

[18] Robin Geelen, Michiel Van Beirendonck, Hilder Vitor Lima Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios Dimou, Ingrid Verbauwhede, Frederik Vercauteren, et al. 2023. BASALISC: programmable hardware accelerator for BGV fully homomorphic encryption. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023, 4 (2023), 32–57.

[19] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 169–178.

[20] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology–CRYPTO 2013*. Springer, 75–92.

[21] Charles Gouert, Vinu Joseph, Steven Dalton, Cedric Augonnet, Michael Garland, and Nektarios Georgios Tsoutsos. 2023. ArctyrEX: Accelerated Encrypted Execution of General-Purpose Applications. *arXiv preprint arXiv:2306.11006* (2023).

[22] Shai Halevi and Victor Shoup. 2021. Bootstrapping for helib. *Journal of Cryptology* 34, 1 (2021), 7.

[23] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. 2021. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), 114–148.

[24] Wonkyung Jung, Eojin Lee, Sangpyo Kim, Jongmin Kim, Namhoon Kim, Keewoo Lee, Chohong Min, Jung Hee Cheon, and Jung Ho Ahn. 2021. Accelerating fully homomorphic encryption through architecture-centric analysis and optimization. *IEEE Access* 9 (2021), 98772–98789.

[25] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. 2023. SHARP: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–15.

[26] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, and Jung Ho Ahn. 2022. Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1237–1254.

[27] Ahmet Can Mert, Sunmin Kwon, Youngsam Shin, Donghoon Yoo, Yongwoo Lee, Sujoy Sinha Roy, et al. 2022. Medha: Microcoded hardware accelerator for computing on encrypted data. *arXiv preprint arXiv:2210.05476* (2022).

[28] Samuel K. Moore. 2023. CHIPS TO COMPUTE WITH ENCRYPTED DATA ARE COMING. Online: https://spectrum.ieee.org/homomorphic-encryption.

[29] Samuel K Moore. 2024. Chips to Compute with Encrypted Data are Coming: Fully homomorphic encryption could make data unhackable. *IEEE Spectrum* 61, 01 (2024), 38–40.

[30] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 238–252.

[31] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. 2022. Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 173–187.

[32] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Furkan Turan, and Ingrid Verbauwhede. 2023. FPT: A fixed-point accelerator for torus fully homomorphic encryption. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 741–755.

[33] Zhiwei Wang, Peinan Li, Rui Hou, Zhihao Li, Jiangfeng Cao, XiaoFeng Wang, and Dan Meng. 2023. HE-Booster: an efficient polynomial arithmetic acceleration on GPUs for fully homomorphic encryption. *IEEE Transactions on Parallel and Distributed Systems* 34, 4 (2023), 1067–1081.