HEIR: A Unified Representation for Cross-Scheme Compilation of Fully Homomorphic Computation

Song Bian*, Zian Zhao*, Zhou Zhang*, Ran Mao*, Kohei Suenaga†, Yier Jin‡, Zhenyu Guan*✉, Jianwei Liu*
*Beihang University, Email: {sbian, zhaozian, zhouzhang, maoran_44, guanzhenyu, liujianwei}@buaa.edu.cn
†Kyoto University, Email: ksuenaga@gmail.com
‡University of Science and Technology of China, Email: jinyier@gmail.com

***Abstract*—We propose a new compiler framework that automates code generation over multiple fully homomorphic encryption (FHE) schemes. While it was recently shown that algorithms combining multiple FHE schemes (e.g., CKKS and TFHE) achieve high execution efficiency and task utility at the same time, developing fast cross-scheme FHE algorithms for real-world applications generally require heavy hand-tuned optimizations by cryptographic experts, resulting in either high usability costs or low computational efficiency. To solve the usability and efficiency dilemma, we design and implement HEIR, a compiler framework based on multi-level intermediate representation (IR). To achieve cross-scheme compilation of efficient FHE circuits, we develop a two-stage code-lowering structure based on our custom IR dialects. First, the plaintext program along with the associated data types are converted into FHE-friendly dialects in the transformation stage. Then, in the optimization stage, we apply FHE-specific optimizations to lower the transformed dialect into our bottom-level FHE library operators. In the experiment, we implement the entire software stack for HEIR, and demonstrate that complex end-to-end programs, such as homomorphic K-Means clustering and homomorphic data aggregation in databases, can easily be compiled to run 72–$179\times$ faster than the program generated by the state-of-the-art FHE compilers.**

I. Introduction

Fully homomorphic encryption (FHE) is a type of general secure multi-party computing (MPC) techniques that enable the participating parties to jointly evaluate arbitrary functions securely. Due to its low round complexity and low communication bandwidth, FHE finds applications in two-party secure function evaluation, such as privacy-preserving machine learning as a service [1–6], and secure computation outsourcing, e.g., secure database [7] and program outsourcing [8].

In spite of its many theoretical merits, programs over FHE ciphertexts are known to be both slow to run and hard to design. First, the running times of tasks at scale can range from hours [9] to days [7, 8] to complete. Second, we see that even the same program can run at drastically different speeds. For example, a homomorphic matrix-vector product of dimension 2048×1001 can take up to 1.5 s to complete using the so-called slot representation [10–12], but only costs 0.11 s when the coefficient representation [4] is adopted. Therefore, we observe a usability-efficiency trade-off, where the designs of FHE protocols either require cryptographic experts to hand-tune the exact homomorphic operators, or the designed protocols suffer from significant performance penalties.

In addition to operator choices, the design of FHE program is further complicated by the ciphertext-operator compatibility problem. As mentioned above, computing homomorphic matrix-vector product using the coefficient representation can be faster than the slot representation. However, a ciphertext in the coefficient representation is incompatible with certain FHE operators, such as homomorphic squaring, homomorphic filtering, and non-polynomial operators. In fact, without heavy tuning on function-approximation polynomials [13–17], neither slot nor coefficient representations are efficient on non-polynomial homomorphic operators. In particular, some recent works [3, 7, 8] demonstrate that high-precision unbounded-depth homomorphic Boolean circuits can be better handled by TFHE/FHEW-like FHE schemes [18, 19].

FHE compiler thus appears as an effective way to solve the usability-efficiency dilemma. Here, we briefly summarize existing compiler designs, and defer a more detailed discussion in Section I-B. Roughly speaking, we can classify existing FHE compilers into two main categories: arithmetic and logic. First, one line of compiler designs study how to efficiently leverage the single-instruction-multi-data (SIMD) properties [20] of ring elements to better implement arithmetic functions over FHE ciphertexts. However, many real-world applications contain non-polynomial computations, such as comparisons, bit-wise operations, etc. To avoid being restricted to polynomial functions, a different FHE compilation approach explores how logic synthesis and instruction set architecture (ISA) designs can be used to convert plaintext programs directly into executable programs over FHE ciphertexts [8, 21, 22]. Although logic FHE compilation can effectively handle programs that contain both polynomial and non-polynomial functions, the performance of homomorphic logic circuits can be unsatisfactory due to the lack of immediately deployable SIMD capability [21, 22]. Besides, most existing FHE compilers leverage domain-specific language (DSL) to assist program compilation [23–26]. While FHE-specific DSL can be handy for fine-tuning output programs, coding over DSLs can be as challenging as that on FHE primitives [23], especially for programmers not familiar with FHE cryptography. Hence, a natural question that arises is as follows: can we design an FHE compiler that is both usable to non-experts and more efficient than circuit-based approaches?

A. Our Contributions

In this work, we propose HEIR, an FHE compiler framework that translates functional complete plaintext programs into efficient programs on FHE ciphertexts. Specifically, by designing a set of new multi-level intermediate representation

(IR) dialects tailored for FHE, we can fully utilize the SIMD capability [2, 4, 20] of FHE ciphertexts. Furthermore, the proposed program transformation passes can seamlessly extract non-polynomial computations out of the plaintext program and map the target computations into proper FHE operators. Consequently, HEIR can simultaneously achieve superior performance against circuit-based compiler frameworks and functional complete usability compared to compilers that can only handle polynomial functions. In summary, the main contributions of this work are summarized as follows.

- **A DSL-Free Functional Complete FHE Compiler**: To the best of our knowledge, HEIR is the first compiler framework that is DSL-free, functional complete, and does not solely rely on Boolean-circuit representation. Here, our key observation is that a multi-level IR framework is essential to cross-scheme FHE program segmentation and operator scheduling.
- **Automated Type Conversions**: We notice that segmenting plaintext programs without the assistance of DSL can be highly challenging, especially when the program contains polynomial and non-polynomial functions that interleave with each other. We propose a new type conversion system to transform a plaintext program into an set of FHE-friendly IR dialects, and instantiate the corresponding bottom-level FHE operators.
- **FHE-Specific Code Lowering**: We realize that simply transforming high-level program structures and data types into FHE programs not only causes performance degradation but also produces incompatible expressions. Therefore, we incorporate optimization passes to solve the operator incompatibility issue, and reduce parameter sizes and bootstrapping frequencies.
- **End-to-End Experiments**: We thoroughly examine the capability and performance of HEIR on a set of program benchmarks. Specifically, we show that HEIR produces programs that are $1.4\times$–$11\times$ faster on arithmetic circuit tasks, $3.2\times$–$4.1\times$ faster on logic circuir tasks, and $72\times$–$179\times$ faster on end-to-end tasks that contain both polynomial and non-polynomial functions. Our framework is publicly available[1].

B. Related Works

In this section, we review some of the related concepts under a broader MPC context in Section I-B1, and detail existing FHE compiler designs in Section I-B2.

1) General MPC Compilers: The automatic generation of multi-party protocols for a given computation task with certain security properties is an area that is under active research. A "compiler" under a general MPC context come in two main flavors: those that bootstrap security [27–29], and those that improve usability along with efficiency [30–42]. In this work, we focus on the latter class of compilers, and further categorize MPC compilers into application-specific and general-purpose MPC compilers. In what follows, we briefly summarize the above two categories of MPC compilers.

[1]https://github.com/heir-compiler/HEIR

Application-Specific MPC Compilers: A number of works focus on how to improve the efficient of specific applications [11, 43–45]. For example, SEPIA [43] develops application-specific MPC operators and protocol-generation framework for privacy-preserving network analysis, and ABY3 [44] optimizes privacy-preserving machine learning by proposing efficient conversions between MPC protocols. In general, application-specific compiler frameworks can achieve better latency and communication performance than general-purpose frameworks. However, programming over an application-specific compiler is obviously not as flexible as general-purpose compilers and it can still be highly non-trivial to choose the most suitable framework for the target application.

General-Purpose MPC Compilers: The limitation of application-specific compilers motivate the development of general-purpose MPC compilers that can compile any plaintext program into MPC protocols [30–34, 36–38]. To reduce the overwhelming complexity of designing MPC protocols for general computations, many works develop C-like or Java-like surface languages to reduce the design costs for non-experts [33, 34, 36–38, 46, 47]. Meanwhile, some compilers do point out that, given the mix-mode nature of MPC protocols, a designated language can provide many benefits in terms of the programming model and security concerns [35, 48, 49]. Unfortunately, we observe two main challenges faced by general-purpose MPC compilers. First, MPC protocols in the general sense includes a large number of cryptographic protocols, and generating the best-performing protocol can be difficult due to its NP-hard nature [50, 51].

2) FHE Compilers: Recently, the design and implementation of FHE compilers attract research attention. On one hand, FHE is adopted as one of the primitives that constitute general MPC compilers [45, 60]. On the other hand, as shown in Table I, we also see a number of recent works on the designs of compilers specifically for FHE [8, 21–23, 52–58, 61–63]. We can roughly classify FHE compilers into three main categories: arithmetic-circuit compilers, logic-circuit compilers, and hybrid-circuit compilers. In what follows, we provide a brief summary on the main properties and features of each of the FHE compiler categories.

Arithmetic-Circuit FHE Compilers: FHE over arithmetic circuit refers to schemes such as BFV [64] and CKKS [65], where the fundamental algebraic operations are ciphertext additions and multiplications. As shown in Table I, arithmetic-circuit FHE compilers [23, 53, 56–59] achieve extremely fast arithmetic circuit evaluation owing to the SIMD capability supported by the underlying FHE schemes. However, arithmetic-circuit FHE compilers generally have poor support for non-polynomial and logic function evaluation, such as comparison and trigonometric functions [13]. In addition, most of the arithmetic-circuit compilers (except ELASM [59]) can only evaluate circuits with limited multiplicative depth either due to the lack of the bootstrapping operator in the compilers or the incompatibility between the encryption parameters. Moreover, as further explained in Section II-C, most (if not all) of the existing compilers only focus on optimizing ciphertexts with slot encodings and do not employ the coefficient encoding

TABLE I
SUMMARY OF EXISTING (F)HE COMPILERS

	Cingulata [52]	Alchemy [53]	E^3 [54]	EVA [23]	MARBLE [55]	RAMPARTS [56]	Transpiler [21]	Porcupine [57]	HECO [58]	ELASM [59]	Ours
Circuit	Logic*	Arth.	Both*	Arth.	Both*	Arth.	Logic	Arth.	Arth.	Arth.	Both
Bootstrap	✗	✗	✗	✗	✗	✗	✓	✗	✗	▲‡	✓
SIMD-Slot	✗	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓
SIMD-Coeff.	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Logic Gates	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗	✓
Ctxt. Conv.	✗	✗	✗†	✗	✗	✗	✗	✗	✗	✗	✓

*Cingulata, E^3 and Marble evaluate logic circuit through binary arithmetic circuits. ‡ ELASM can schedule bootstrapping, but the operator is not implemented in their library.
† The costs for converting ciphertexts from general arithmetic representation to binary arithmetic representation is prohibitive in E^3.

technique. Consequently, most existing arithmetic-circuit FHE compilers are rather application-specific, in that they only have end-to-end compilation support for arithmetic circuits over finite fields.

Logic-Circuit FHE Compilers: Based on various optimizations in bootstrapping [18, 66], FHE compilers over logic circuits start to emerge as alternatives for general program compilation. We see a line of works [8, 21, 22, 52] that adopt circuit synthesis techniques to compile programs written in Verilog [22] or C [8, 21] into equivalent homomorphic circuits over logic gates (or instructions over logic circuits in the case of [8]). Since programs can be easily transformed into Boolean circuits, some logic-circuit FHE compilers, e.g. Transpiler [21], are functionally complete and readily deployable. However, we point out that many existing logic FHE compilers are functionally incomplete as well. For example, as shown in Table I, Cingulata, Marble and E^3 can only support circuit with limited multiplicative depth due to the lack of the bootstrapping operator. Furthermore, it is commonly known that representing general programs as Boolean circuits are inefficient [42]. Hence, programs produced by logic-circuit FHE compilers suffer from degraded latency performance, especially on arithmetic-heavy tasks (e.g., homomorphic convolution).

Hybrid-Circuit FHE Compilers: Due to the usability and performance limitations of arithmetic-only and logic-only operators, some works explore how to efficiently combine arithmetic and logic FHE operations to generate one FHE program over the hybrid circuit. For example, E^3 can convert bit-level logic values to integers over finite fields and *vice versa*, enabling arithmetic and logic operators to co-exist in the generated FHE program. Unfortunately, the cost of reversing the logic-to-arithmetic conversion is prohibitive in E^3 due to the need for a huge number of homomorphic multiplications. As a result, E^3 is still mostly an arithmetic-orient compiler framework.

Remark on DSL: By explicitly defining domain-specific lexica, DSL-based FHE compiler frameworks [23, 58, 62] permit simpler compiler designs as FHE-specific operators can be directly embedded into the target program. However, coding over DSLs can be as challenging as that directly over FHE primitives [53], for choosing the correct domain-specific lexica still requires a deep understanding of fundamental FHE mechanics.

II. Preliminaries and Background

In this work, we use two types of lattice-based FHE schemes categorized according to their functionalities, namely, arithmetic FHE (Section II-C) and logic FHE (Section II-B. Both FHE categories follow the standard FHE hardness assumptions based on the learning with errors (LWE) and ring learning with errors (RLWE) problems. In what follows, we describe constructions as well as optimizations of logic and arithmetic FHE schemes in Section II-B and Section II-C, respectively. In addition, we also discuss some of the scheme conversion techniques proposed recently in Section II-D.

A. Notation

For simplicity, we use the symmetric version of the CKKS scheme [65] as an example of FHE throughout this work. Note that the basic encryption mechanism is the same for CKKS and TFHE [18]. We use λ to denote the security parameter, p the plaintext modulus, q/Q the ciphertext moduli. $\mathbb{Z}_q$ refers to the set of integers modulo q. For some lattice dimension n/N, we write $R_{N,q} = \mathbb{Z}_q[X]/(X^N+1)$. In this paper, we use tilde lower-case letters such as $\tilde{a}$ to depict polynomials and bold lower-case letters such as $\mathbf{a}$ to represent vectors. In particular, $\mathbf{a}[i]$ refers to the i-th element in $\mathbf{a}$.

B. Logic FHE

We consider FHE schemes such as FHEW [19, 67], TFHE [18] as logic FHE. For logic FHE, each ciphertext only encrypts one plaintext message, such as a single logic value of 0 or 1. As a result, ciphertext bootstrapping over logic FHE is extremely fast. By evaluating logic circuits over such ciphertexts, logic FHE schemes can easily be made functional complete. Homomorphic gates in logic FHE are mainly evaluated on LWE ciphertexts, which are composed as follows. Given a plaintext message $m \in \mathbb{Z}_q$ and the secret key $\mathbf{s} \in \mathbb{Z}_q^n$, an LWE ciphertext is

$$\mathsf{LWE}_{\mathbf{s}}^{n,q}(m) = (b, \mathbf{a}) = (-\mathbf{a}^{\mathrm{T}}\mathbf{s} + \Delta m + e, \mathbf{a}).$$

where $\mathbf{a} \in \mathbb{Z}_q^n$ is chosen uniformly at random, the error e is chosen from some distribution χ_{noise} with standard deviation $noise$, and Δ is a scaling factor to protect the least significant bits of the message from the noises. The main drawback for logic FHE is that arithmetic operators such as large-precision additions and multiplications are extremely slow due to the Boolean nature of the computation.

Programmable bootstrapping: The main optimization technique for logic FHE is programmable bootstrapping (PBS), also known as functional bootstrapping. PBS is one of the most important techniques that improve the efficiency of logic FHE schemes [66, 68]. PBS can evaluate arbitrary functions through Look-Up Tables (LUTs) while refreshing the noise of the ciphertext. While earlier PBS requires the MSB of the plaintext to be set to zero, methods are proposed to utilize the full input plaintext domain [69–71]. Meanwhile, multi-value PBS is devised to allow for a much smaller cost when evaluating multiple homomorphic functions with the same inputs [72–74]. Equipped with PBS, logic FHE becomes more capable of efficiently evaluating both logic and arithmetic circuits [73, 74].

C. Arithmetic FHE

FHE schemes that heavily rely on the single instruction multiple data (SIMD) technique of RLWE ciphertexts [20] are categorized as arithmetic FHE, e.g., BFV [64, 75], BGV [76], and CKKS [65]. Given an encoded plaintext polynomial $\tilde{m} \in R_{N,Q}$ and a secret key polynomial $\tilde{s} \in R_{N,Q}$, an RLWE ciphertext is encrypted as

$$\mathsf{RLWE}_{\tilde{s}}^{N,Q}(\tilde{m}) = (\tilde{b}, \tilde{a}) = (-\tilde{a} \cdot \tilde{s} + \Delta\tilde{m} + \tilde{e}, \tilde{a}).$$

where $\tilde{a} \in R_{N,Q}$ is chosen uniformly at random, the error $\tilde{e}$ is chosen from some distribution χ_{noise}, and Δ is a scaling factor. While arithmetic FHE is dominant in implementing linear and polynomial functions [6, 77–79], due to the limitation of arithmetic computations on quotient (polynomial) rings, it is generally inefficient to evaluate non-polynomial or logic functions over arithmetic FHE schemes. For example, evaluating privacy-preserving sorting over CKKS can be up to 6× slower than that on TFHE [3].

Encoding Methods: For logic FHE schemes, plaintext messages usually do not need complex encoding since each ciphertext can only encrypt one plaintext. However, for arithmetic FHE schemes, a message vector is first encoded in a polynomial before being encrypted to an RLWE ciphertext [80, 81]. Slot encoding [20] and coefficient encoding [2, 4] are widely used in the existing FHE algorithms. The concrete definitions are as follows.

- **Slot Encoding**: The CKKS-variant slot encoding maps input messages from $\mathbf{m} \in \mathbb{C}^{N/2}$ and scaling factor $\Delta \in \mathbb{R}$ to a polynomial $\tilde{m} \in R_N$. $\tilde{m}(\zeta_j) = \Delta \cdot \mathbf{m}[j]$ where ζ_j is $2N$-th root of unity that satisfies $X^N + 1 = 0$. Actually, slot encoding can be viewed as the number theoretic transform (NTT) in BFV [64] or discrete Fourier transform (DFT) in CKKS [65]. Slot encoding supports efficient element-wise SIMD addition and multiplication between plaintexts ciphertexts [20].
- **Coefficient Encoding**: In coefficient encoding, the encoded plaintext polynomial $\tilde{m} \in R_{N,Q}$ of messages $\mathbf{m} \in \mathbb{Z}_Q^N$ is given as:

$$\tilde{m}(X) = \tilde{m}_0 + \tilde{m}_1 X + ... + \tilde{m}_{N-1} X^{N-1}.$$

where $\tilde{m}_i = \mathbf{m}[i]$ for $i = 1, 2, .., N-1$. Coefficient encoding is commonly used in sample extraction [18] and CKKS bootstrapping [15, 16, 82–85]. Besides, coefficient representation is found to be more efficient than slot representation in implementing homomorphic inner products and convolutions [2, 4].

Residue Number System (RNS): The RLWE ciphertext modulus Q is often set to be as large as hundreds of bits to support deeper homomorphic circuits and accommodate larger noises. Since large integers do not fit into the 64-bit CPU word size, Chinese remainder theorem (CRT) is used to represent the quotients of $R_{N,Q}$ as a product of elements from smaller quotient rings. In other words, we can decompose Q into a product of k co-prime factors $Q = \prod_{i=0}^{k-1} q_i$. Therefore, for an RLWE ciphertext $ct = (\tilde{b}, \tilde{a}) \in R_{N,Q}^2$, arithmetic operations can be performed separately on each of the CRT-decomposed (i.e. RNS [86]) vectors $\mathbf{b}, \mathbf{a} \in \prod_i R_{N,q_i}$, eliminating the need of large-integer arithmetic [87–89].

D. Ciphertext Conversions in FHE

Conversion between arithmetic and logic FHE. As mentioned above, arithmetic FHE schemes such as BFV [64, 75], BGV [76], and CKKS [65] pack multiple messages into one ciphertext and support efficient SIMD operations [20]. However, due to the limitation of arithmetic on polynomial rings, it can be difficult for RLWE-based schemes to evaluate non-polynomial functions such as comparison and floating-point division. On the contrary, while logic FHE schemes such as FHEW [19, 67] and TFHE [18] can only encrypt one message in one ciphertext, logic FHE has lightweight bootstrapping and can evaluate multiple non-polynomials functions simultaneously via TFHE PBS.

To support the evaluation of both logic and arithmetic functions, a number of ciphertext conversion techniques have been proposed to achieve the best of both worlds. Recent developments explore how to combine logic and arithmetic FHE by developing a series of conversion algorithms between the LWE and RLWE ciphertext formats. For instance, CHIMERA [90] introduces the first hybrid solution for switching between TFHE and a torus variant of BFV and CKKS. This conversion method enables CHIMERA to perform efficient SIMD arithmetic operations under BFV/CKKS scheme and then evaluate non-polynomial functions with TFHE PBS after converting from arithmetic FHE ciphertext to logic FHE ciphertext. Hence, CHIMERA is later adopted by privacy-preserving neural networks inference [6, 66, 68, 91] to implement non-linear activation layers. Unfortunately, the conversion from TFHE to BFV/CKKS in CHIMERA is extremely costly in terms of computation and evaluation key sizes. As a solution, PEGASUS [3] addressed the drawback of CHIMERA through approximate decryption, and proposed a practical framework for switching back and forth between TFHE and CKKS ciphertexts. Subsequently, HEDA [7] proposed the parameter lift bootstrapping technique to efficiently convert encryption parameters for the accurate TFHE-CKKS conversion required by database applications. Through the process, two key algorithms are developed for the conversions between arithmetic and logic circuits, namely, SampleExtract and Repack.

- **SampleExtract**: First proposed in [18], SampleExtract can extract an element of the encoded vector in an arithmetic FHE ciphertext $\mathsf{ct} \in \mathsf{RLWE}_{\tilde{s}}^{N,Q}(\tilde{m})$ into a logic FHE ciphertext:

$$\textsc{SampleExtract}(\mathsf{ct}, i) \to \mathsf{LWE}_{\mathbf{s}}^{N,Q}(\tilde{m}_i), i \in \langle N \rangle.$$

  Here, the equation extracts the i-th plaintext message $\tilde{m}_i$ encrypted in the RLWE ciphertext ct.
- **Repack**: Repack can pack a set of logic FHE ciphertexts $\{\mathsf{ct}_i\}_{i\in\langle N\rangle}$, $\mathsf{ct}_i \in \mathsf{LWE}_{\mathbf{s}}^{N,Q}(\mathbf{m}[i])$ into a single RLWE ciphertext:

$$\textsc{Repack}(\{\mathsf{ct}_i\}_{i\in\langle N\rangle}) \to \mathsf{RLWE}_{\tilde{s}}^{N,Q}(\tilde{m}),$$

  where the set of LWE ciphertexts encrypt each element of the plaintext vector $\mathbf{m}$, and the output RLWE ciphertext encrypts $\tilde{m}$ such that $\tilde{m}_i = \mathbf{m}[i]$.

We see two distinct methods for implementing the REPACK operator decryption-based [3, 18, 92], and automorphism-based [93]. Though decryption-based Repack runs faster when packing a large number of ciphertexts, the method suffers from the fact that the repacked plaintexts are low in bit precision. On the other hand, automorphism-based REPACK achieves a high level of message precision but results in incompatible ciphertext encodings. In this work, we improve high-precision automorphism-based Repack when packing a small number of LWE ciphertexts, which will be further discussed in Section IV-B2.

Conversion between slot encoding and coefficient encoding. In addition to scheme-switching conversions, we also need the homomorphic conversion between the slot and coefficient encoding, for different plaintext encoding methods have different applications. For example, by using coefficient encoding, some types of linear transformations such as homomorphic inner products and homomorphic convolutions are faster than that using slot encoding [2, 4]. To fully leverage the benefits of different encoding methods, encoding-switching techniques are proposed in [82, 94]. Lastly, we note that, while the above conversion methods can solve the ciphertext compatibility problem, many conversion operators incur a significant computational burden per se. Consequently, we believe that a compiling framework that "understands" how and when to use different FHE schemes as well as scheduling necessary conversion operators is crucial to both the usability and efficiency of FHE-based secure computing systems.

III. The HEIR Compiler Framework

In this section, we present HEIR, a compiler framework that takes advantage of both arithmetic and logic FHE schemes with different plaintext encodings. We first formulate the problems and challenges faced by our framework in Section III-A. Then, we summarize the overall framework and threat model in Section III-B and Section III-C. Finally, we discuss details on certain aspects of our design pipeline in Section III-D.

A. Problem Formulation and Challenges

The main goal of this work can be formulated as: given a plaintext program and its input data, we wish to generate a homomorphic program with FHE-friendly data structures and appropriate data encoding. In what follows, we identify and summarize the key barriers confronting the design of a cross-scheme FHE compiler.

Challenge 1: The first barrier is the lack of a unified intermediate representation for the various ciphertext formats and homomorphic operators in cross-scheme FHE. For instance, as shown in Table I, existing compilers such as EVA [23] and HECO [58], which mainly focus on optimizing arithmetic FHE, only supports operations on RLWE ciphertext and do not accommodate non-polynomial function evaluations. Similarly compilers like Transpiler [21] decompose all input data into Boolean representations and implement every arithmetic operation using logic circuit, which is clearly inefficient. To enable hybrid-circuit FHE, the compiler needs a unified IR infrastructure to correctly identify the lowering layers and perform key optimization passes.

Challenge 2: The second barrier lies in the fact that, since HEIR adopts cross-scheme compilation, it becomes much harder to schedule conversion operations and select of encryption parameters. In particular, to select the appropriate parameters for the input program, the first step is to determine the *level* for each ciphertext. Although EVA [23] can automatically maintain the ciphertext level for leveled homomorphic encryption (LHE), to the best of our knowledge, there is no compiler framework that can automate the scheduling of ciphertext bootstrapping along with level management involving arithmetic-logic conversions.

B. HEIR Compiler Overview

To better support and make full use of optimizations provided by existing compilation toolchains, HEIR is constructed over the MLIR framework [95]. In MLIR, custom-defined IRs are known as dialects, which constitute the middle-end of the compiler stack. The key advantage of installing dialects over MLIR is the ability to maintain the desired level of semantics while performing various types of graph-based optimizations.

We provide an overview of the processing flow of HEIR in Figure 1. Here, the input C program is first transformed into an MLIR module composed of high-level IRs with built-in MLIR dialects using the Front-End module based on Polygeist [96]. Then, the Middle-End module of HEIR consists of a number of FHE-specific transformation and optimization pass. We group these lowering passes into two main stages: i) the transformation stage, and ii) the optimization stage. First, in the transformation stage, we gradually lower programs written in high-level IR and the associated plaintext data types into a series of FHE-friendly dialects, and produce programs written in `fhe` dialect. Within the transformation stage (Section III-D5), the core functionality is program segmentation and automated data type as well as operator conversions. Second, in the optimization stage, we lower the `fhe` dialect into the `lwe` and `rlwe` dialects, where logic and arithmetic operators can be concretely instantiate over the compatible ciphertext IR types. In addition, we need to perform lower-level FHE optimizations such as level management and parameter tuning, which are the essential steps in instantiating the underlying FHE

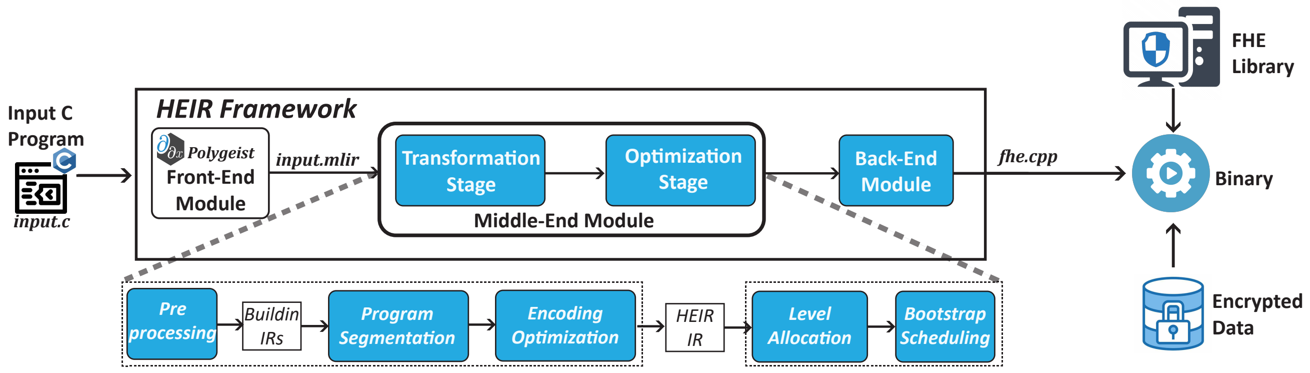


Fig. 1. Compilation flow of the HEIR framework.

schemes. For instance, we need to maintain a level management system equipped with ciphertext bootstrapping to guarantee security while improving efficiency. Finally, the Back-End module lowers `lwe` and `rlwe` IRs into concrete homomorphic operators, which we refer to as homomorphic instructions, that are supported by the unified bottom-level FHE library. Here, we formulate a unified set of homomorphic instructions such that HEIR can support a large variety of cross-scheme homomorphic operations with high efficiency and usability.

C. Threat Model

In this work, we assume a secure two-party computation setting which is typical for homomorphic encryption. Here, the client has some private data, and wishes that the server could perform certain computations on the data. Meanwhile, we assume that the server is semi-honest in that the server will indeed carry out the prescribed computation in the protocol, but is curious about the encrypted data.

Different from data privacy, we point out that, in most FHE-based protocols, the program is public to the server. In fact, for the server to execute a program on the private inputs from the client, the server has to know exactly what kind of computations need to be performed. Therefore, while the FHE compiler can be held either by the server or the client, the compiler needs to ensure that no plaintext data are exposed to the server during compile time. As for the target program to be compiled, we consider all of its inputs as data that are to be encrypted. However, one of the subtle complexity here is that constant values declared in the program are considered to be a part of the program rather than the data, and are thus public to the server. Lastly, we note that a minimum set of DSL lexica can be integrated into our framework to permit a higher level of flexibility in terms of public-private data declarations.

D. HEIR Framework Details

1) Design of HEIR Dialect: As illustrated in Figure 2, once the program is processed by the Front-End module of HEIR, the input program is transformed into the surface-layer IR representation consisting of a variety of high-level built-in dialects that can be further lowered to our HEIR dialect. Owing to the multi-level nature of MLIR, we are able to define additional dialects consisting of FHE-specific data types and

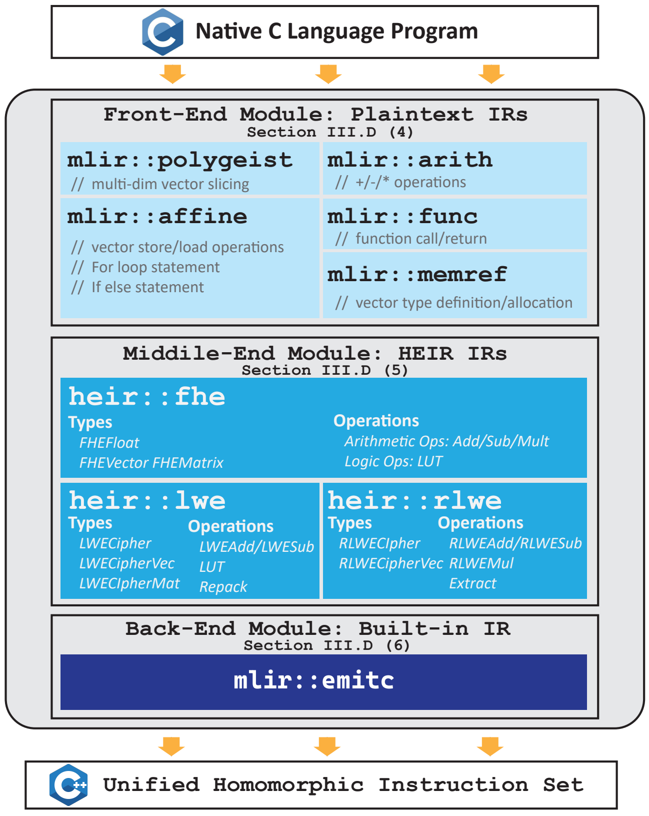


Fig. 2. An overview of the HEIR dialect structure.

operations to support transformations and optimizations passes in the Middle-End module, referred to as the `heir` IR. The `heir` IR is also a multi-level dialect consisting of two main dialect levels, namely, the `fhe` dialect and the `lwe-rlwe` dialect. On the upper layer, the `fhe` dialect first characterizes both the operations and the semantics of programs over FHE, while abstracting away lower cryptographic subtleties. Next, on the lower layer, `lwe` and `rlwe` dialects make explicit the low-level semantics of arithmetic FHE and logic FHE schemes and encryption parameters. Finally, the `lwe` and `rlwe` dialects are compiled into concrete operators implemented by the bottom-level FHE libraries. Besides, different from the IR defined in EVA [23], the intermediate representations of a given input program in HEIR are in the static single-assignment (SSA) form, where each operator represents an operation performed in the program and each operand represent a program argument or a piece of data generated inside the program.

Here, we explain some notations for types and values proposed in the `heir` IR. First of all, data appearing in a program can be classified into three types: *Inputs*, *Consts* and *Variables*. The value of *Inputs* are infeasible in compile-time and only be initialized in run-time. On the contrary, the value of *Consts* are constant values hard-coded into the program. *Variables* are defined inside the program or generated from the existing *Inputs* and *Consts*. For each defined *Variable* u in the program, we write $u.parm$ as its attributes list and $u.parm_i$ as the i-th attribute in $u.parm$. To reflect the relationships between operators and their operands, we write $u.op_i$ to denote the i-th operations to be performed on u as a *Variable* can be used by multiple operators.

2) Types in HEIR: HEIR has two main type systems, namely, the plaintext types and the ciphertext types (details can be found in Table A2 in the appendix). For plaintext types, to interface with the program generated by Polygeist, some MLIR built-in types such as `Float` and `MemRef` are included in HEIR plaintext types for plaintext-ciphertext transformation. Meanwhile, for the ciphertext types, attributes like the ciphertext level and the data size is included for better semantics abstraction. To strengthen the expressiveness, we add unique types such as vectors and matrices composed of LWE and RLWE ciphertexts as elements.

3) Operators in HEIR: Due to the single-scheme nature of existing FHE compilers, the number of available FHE operators and operations is severely restricted. In contrast, HEIR includes both arithmetic and logic operators as well as conversion operators for scheme- and encoding-switching. We defer the full operator list into the appendix (i.e., Table A3), and provide a brief summary as follows.

- **RLWE Operators**: include addition, subtraction, and multiplication between RLWE ciphertexts and RLWE encoded plaintexts.
- **LWE Operators**: include basic addition, subtraction, and functional bootstrapping operations for LWE ciphertexts.
- **Conversion Operator**: include two types of operations. The first type is scheme-switching operators to convert one or more ciphertexts from LWE formats to RLWE formats and *vice versa*. The second type is encoding-switching operators that can change the encoding method of an RLWE ciphertext homomorphically for better overall computation efficiency.
- **Maintenance Operators**: which mainly include vector data transfer operators like LOAD and STORE.

4) Front-End and Program Preprocessing: Existing fully homomorphic encryption (FHE) compilers use different DSLs to restrict the input types and operations available, where the main objective is to eliminate the need of complex type conversions. However, we argue that a non-DSL front-end FHE compiler is of great importance to enhance the usability and expressiveness of the input plaintext programs. Therefore, in HEIR, we allow developers to code their programs in the native C language, and the list of supported operations is summarized in the appendix (Table A1). In addition to the C built-in operators such as addition and multiplication, HEIR also supports comparison, array operations, the `if` statement and the `for` statement. Moreover, for other complicated operations not listed in Table A1 such as taking reciprocals and computing absolute values, they can be re-formulated as a standalone function call and implemented by the FHE LUT operator.

To transform the C program into SSA-style IR, we adopt Polygeist [96] as the front-end of HEIR framework. Polygeist can lower a C program to a variety of high-level built-in IRs in the MLIR framework. The key advantage of directly employing Polygeist is that generic program optimizations can be decoupled with the subsequent FHE-specific transformations and optimizations. Consequently, different from existing FHE compilers like [62], we do not need to manually re-implement generic program optimizations such as common sub-expression elimination (CSE) and dead-code elimination and maximize the utilization of the existing tools.

5) Transformations and Optimizations in HEIR: After going through the Front-End module, the input program enters the Middle-End module, and FHE-specific transformations passes are applied to convert plaintext operations into homomorphic operations. We elaborate on the following two main stages in the Middle-End module.

Program Transformation: In this stage, operation conversion, as well as type conversion, is necessary to map the plain operations on plain data to the homomorphic operations on encrypted data. The transformation stage is composed of three phases: i) preprocessing, ii) program segmentation, and iii) encoding optimization. In the beginning, preprocessing converts the program expressed by the built-in plaintext IRs into our `fhe` dialect. Additionally, statements such as `if` and `for` are converted into a sequential structure as long as the loop condition is known at compile time. Note that the number of loop iterations can be unknown at compile time but known at run time, e.g., if it is decided by the server. Next, In the program segmentation stage, HEIR will segment the input program into arithmetic or logic regions, assign different homomorphic primitives to each region, and insert proper scheme-switching conversions to bridge different regions. During this stage, lexica in the `fhe` dialect is gradually replaced by the `lwe-rlwe` dialect. Hence, encrypted data types like `LWECipher` and `RLWECipher` are allocated to each of the *Variable*. Finally, in the encoding optimization pass, the encoding method of each of the `RLWECipher` in the program is determined to minimize the overall computation latency considering all the necessary encoding-switching conversions. More details on program transformations is discussed in Section IV.

Cryptographic Optimization. In this stage, we concentrate on how to initialize the underlying FHE schemes in an efficient way. The biggest challenge here is the parameter selection for different FHE schemes. At its current state, automatic parameter selection strategies in existing compilers [23, 24, 62] are designed for leveled homomorphic encryption schemes.

In HEIR, we propose a bootstrapping-aware ciphertext level management technique to instantiate the bottom-level FHE schemes with the smallest possible parameters while ensuring proper security levels and correct ciphertext decryption. One of the most important technique in achieving the above goal is the multi-modulus bootstrapping technique, which transforms the input ciphertext with a small ciphertext modulus into a ciphertext with a large modulus while retaining small ciphertext noises. More details on the optimization stage is further discussed in Section V.

6) Back-End and Code Generation in HEIR.: Last but not least, in the Back-End module, HEIR translates a program represented using the `lwe-rlwe` dialect into an executable. Here, HEIR needs to lower the program from `lwe-rlwe` dialect to the `EmitC` dialect. Then, the `EmitC` program will be compiled into a final C-language program written in the bottom-level FHE application programming interface (API), which are essentially operators implemented by the underlying cross-scheme FHE library. Due to the fact that no existing FHE libraries implement both arithmetic and logic FHE schemes with proper conversion operators, HEIR targets the *emitc* program to our dedicated bottom-level FHE library, which has sufficient support for arithmetic and logic FHE schemes. However, note that the multi-layer property of the MLIR representation allows HEIR to easily target additional bottom-level FHE libraries regardless of the specific implementation details.

IV. Automated Program Transformation

Since program transformation is the core of our HEIR frameowrk, in this section, we provide further details on the exact procedures for program segmentation and operator-operand mapping. To illustrate the optimization method used in this section, we first use an application program composed of both arithmetic and logic operations as a simple demonstration.

Example Application - Minimum Distance: In Figure 3(a), we consider a simple program sketched in Figure 3(a) that, among a number of input data points, outputs the data point that is closest to the input center point. Here, we first need to measure the distance based on some L_2 distance calculations, which are essentially arithmetic computations. Then, we need to identify the minimal value from the calculated distances, which is clearly a logic comparison. Hence, the example program serves as an illustration of real-world applications that often contain both arithmetic and logic operations, and cannot be compiled using the existing arithmetic-circuit FHE compilers. Moreover, the use of nested loops accessing a complex set of indices creates the opportunity to leverage the SIMD batching properties of arithmetic-circuit FHE schemes, which are not available in logic-circuit FHE compilers.

Using the state-of-the-art FHE techniques, we manually crafted an efficient FHE program that implements Figure 3(a) in Figure 3(b). The key observation is that, using the coefficient SIMD encoding, the Euclidean distance can be implemented using a single ciphertext-ciphertext multiplication. Furthermore, the minimal value selection of encrypted distances can be realized by a scheme-switching to LWE ciphertext and a series of homomorphic LUT evaluation through PBS. A detailed summary of the distinction between arithmetic instructions and Boolean instructions will be included in Appendix A. In what follows, we explain how we can transform Figure 3(a) into Figure 3(b) automatically using our framework.

A. Program Conversion from Plaintext to Ciphertext

Since the compilation output of the Front-End module in HEIR is a program represented in a mix of high-level built-in dialects as shown in Figure 2, the main task is to transform such plaintext program into a unified homomorphic representation, i.e., the `fhe` dialect. At this stage, HEIR does not care if a plaintext is mapped to an RLWE or LWE ciphertext. Instead, we define three unified encrypted data types, namely, `FHEFloat`, `FHEVector`, `FHEMatrix`, in `fhe` dialect to transform the plaintext variables into encrypted forms. Through the transformation from built-in dialects to `fhe` dialect, HEIR mainly address the following two problems.

First, we need to remove the incompatible statements for homomorphic computation. For example, since the branch structure is incompatible with the FHE computation paradigm, every `if/else` statement shall be emulated by evaluating the result of all branches and performing an multiplexed aggregation. Similarly, the `For` statement with constant length needs to be unrolled for subsequent vectorized optimizations. Loop statements with encrypted length are currently not supported by most FHE schemes, except [8]. The main challenge here is that the server cannot judge if an encrypted loop has finished without interacting with the client. Therefore, HEIR will throw an error when dynamic-length loops are detected.

Second, we need to check all variables if they can remain as plaintexts or need to be encrypted as ciphertexts. As described in Section III-C, we only consider the program inputs as confidential data that must be protected from the server. Therefore, all program input arguments and variables derived from these arguments are mapped to encrypted variables, which can be easily identified through the *Def-Use* chain of the program. Meanwhile, constants defined inside a program are mostly matrix indices or operands. If the constant is used for computation with encrypted data, this constant is essential to be encoded. The complete program conversion algorithm can be found in Algorithm 3 of the appendix.

B. Program Segmentation

Here, HEIR separates the program into arithmetic and logic regions in order to map the operators and data types into the corresponding homomorphic representations. During this process, HEIR will *lower* `fhe` dialect to `lwe` and `rlwe` dialects. There are two important procedures involved in the segmentation of a program: operator-based conversion (Section IV-B1) and mini-repacking (Section IV-B2).

1) Operator-based Conversion: As mentioned in Section II, arithmetic FHE is good at performing SIMD addition and multiplication operations, while logic FHE is efficient in evaluating non-polynomial functions. Therefore, it is actually easy to segment the program into different regions. However,

```
#include<stdio.h>
int min_dist(int data[5][4], cent[4])
{
  int min;
  int dist[5];
  for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 4; j++)
      dist[i]+=(data[i][j]-cent[j]) * (data[i][j]-
    cent[j])
  }
  min = dist[0];
  for (int i = 1; i < 5; i++) {
    if (min > dist[i]) min = dist[i];
  }
  return min;
}
```

(a) The original handwriting implementation of a minimal distance program

```
LWECipher min_dist(vector<RLWECipher> data,
    RLWECipher cent)
{
  LWECipher min;
  vector<LWECipher> dist;
  for (int i=0; i<5; i++)
    dist[i] = inner_prod(data[i], data[i])+
    inner_prod(cent,cent)-2*inner_prod(data[i],
    cent);
  min = dist[0]
  for (int i = 1; i < 5; i++) {
    LWECipher msb = dist[i] - min;
    msb = LUT(msb, x < 0 ? 1 : 0);
    min = msb * dist[i] + (1 - msb) * min
  }
}
```

(b) Optimized batched homomorphic solution of the same program

Fig. 3. Two programs calculating the minimum distance between each data point and the center point. a) the input program for HEIR, which calculates element-wise multiplication individually. b) the output of HEIR, which transforms the original program into a homomorphic program and uses batched multiplication for RLWE ciphertexts to simplify the computation of vector inner product.

$$\mathrm{FHE-LWE}\,\frac{u \in \{Inputs, Variables\} \quad u.type \in \{\texttt{FHEVector, FHEMatrix}\}}{u.type \leftarrow \{\texttt{LWECipherVec, LWECipherMat}\}}$$

$$\mathrm{LWE-RLWE}\,\frac{u \in \{Inputs, Variables\} \quad u.type \in \{\texttt{LWECipherVec, LWECipherMat}\} \quad u.op \in \{\textsc{Multiply}, \textsc{Slice}\}}{u.type \leftarrow \{\texttt{RLWECipher, RLWECipherVec}\} \quad u \leftarrow \textsc{Repack}(u)}$$

$$\mathrm{RLWE-LWE}\,\frac{u \in \{Inputs, Variables\} \quad u.type \in \{\texttt{RLWECipher, RLWECipherVec}\} \quad u.op \in \{\textsc{Lut}, \textsc{Load}, \textsc{Store}\}}{u.type \leftarrow \{\texttt{LWECipherVec, LWECipherMat}\} \quad u \leftarrow \textsc{SampleExtract}(u)}$$

Fig. 4. Graph rewriting rules for program segmentation in HEIR.

it is not always the case that arithmetic regions should be implemented by arithmetic FHE. For example, if we need to perform the addition between a small amount (e.g., 2 to 4) of low-precision (e.g., 2 to 4 bits) ciphertexts, we do not need to execute a scheme conversion to convert from logic FHE to SIMD-compatible arithmetic FHE. The addition can simply be carried out over logic FHE ciphertexts. Therefore, we argue that both scheme and encoding conversions need to be carefully scheduled by the compiler for each and every homomorphic operator in the program. To enable operator-based operand conversion, we take a simple yet effective approach: we first initialize all variables to be non-vectorized (i.e., LWE) ciphertext, and then scan the entire program to see if there exists regions we can schedule a scheme-switching conversion to leverage the SIMD capability of arithmetic FHE using an RLWE ciphertext.

For programs without vectorized inputs and variables, we can directly transform the plaintext operations to the corresponding homomorphic operations since there is no chance to leverage the SIMD capability. As demonstrated in FHE-LWE rule of Figure 4, HEIR transformed all encrypted data types and operators into the corresponding formats in `lwe` dialect. Afterwards, variables with `LWECipher` type used as operands of a MULTIPLY operator will be converted to a RLWE ciphertext for further arithmetic operations. To achieve this goal, SAMPLEEXTRACT and REPACK operators are essential to be inserted in the conversion process as demonstrated in LWE-RLWE and RLWE-LWE rules.

On the other hand, for programs with vectorized inputs or intermediate variables, we integrated the batching pass (Algorithm 2 in HECO [58]) to transform individual element-wise operations into a batched computation paradigm. To improve the generality, we extended the algorithm also for two-dimensional matrices. In the case of two-dimensional batching pass, we first parse the matrix into a set of row vectors, and each row is seen as a potential RLWE ciphertext to be batched. Then, the batching pass is performed on each of the row vectors. If a scheme-switching conversion can be scheduled as judged by the compiler, an `RLWECipher`-type (can also be `RLWECipherVec` and `RLWECipherMat` depending on the exact computations) variable will be generated, and the subsequent computations can be batched on the associated ciphertexts.

2) Mini-Repack Algorithm: As demonstrate in Figure 4, REPACK operator is essential for performing LWE-RLWE conversion. We observe that a lightweight REPACK algorithm for packing a small number of ciphertexts is valuable, as further discussed in Appendix C. To meet the above needs, we proposed a *Mini-Repack* algorithm which is tailored for packing a small number of LWE ciphertexts as demonstrated in Algorithm 1. To start with, on Line 3-4, a single LWE ciphertext is packed into the 0-th coefficient of an RLWE ciphertext using the *EvalTr* algorithm introduced in [93]. Next, on Line 5, the ciphertext is first multiplied with a monomial to rotate the encrypted plaintext into the i-th coefficient, and then accumulated into the ACC ciphertext. As mentioned,

```
Algorithm 1: Mini-Repack
Input    : $t$ LWE ciphertexts $(ct_0, ct_1, ..., ct_{t-1})$ where
           $ct_i = \mathsf{LWE}^{n,Q}_{\mathbf{s}}(m_i) = (b_i, \mathbf{a_i})$.
Input    : Automorphism Keys $\mathsf{KsK}$
Output   : Packed RLWE Cipher $\mathsf{RLWE}^{n,Q}_{\tilde{s}}(\tilde{m})$, where $\tilde{m}_i = m_i$,
           for $i \in \mathbb{Z}_t$
$ACC \leftarrow (0, 0) \in R_Q^2$
for $i = 0$ to $t - 1$ do
    $\tilde{ct}_i \leftarrow (\tilde{b}^{(i)}, \tilde{a}^{(i)}) \in R_Q^2$, where $\tilde{b}^{(i)}_0 = b_i$, $\tilde{a}^{(i)}_j = \mathbf{a}[j]$.
    $\tilde{ct}_i \leftarrow \textsc{EvalTr}_{n/1}(\tilde{ct}_i, \mathsf{KsK})$
    $ACC \leftarrow ACC + \tilde{ct}_i \cdot X^i$
Return   : $ACC$
```

while *Mini-Repack* can be slower than both [3] and [93] when packing a large number of ciphertexts, we find the technique practical when dealing with real-world programs that often do not have large-scale parallelism.

C. Encoding Optimization

Most existing arithmetic-circuit FHE compilers only support RLWE ciphertexts with slot encoding. In spite the fact that all linear transformations can be implemented on slot encoding [97, 98], it is shown that coefficient encoding can be much faster in certain cases [2, 4]. To further complicate the situation, we note that the SampleExtract algorithm [3] and the Repack algorithm [93] used for scheme-switching conversion can only be applied to coefficient-encoding ciphertexts. Therefore, any slot-encoding RLWE ciphertexts that need to be extracted to LWE ciphertexts must first go through an extra slot-to-coefficient encoding-switching conversion, which is in fact computationally expensive.

To correctly capture the encoding characteristics in general programs, we proposed a novel encoding pass such that RLWE ciphertexts can be set in the most efficient encoding methods throughout the program. Similar to that of scheme-switching scheduling, in the encoding pass, we first set all RLWE ciphertexts to be in the slot representation, and then execute Algorithm 4 in Appendix D to identify program fragments that are more suitable to be in the coefficient encoding formats.

One cryptographic subtlety here is that, due to the nature of polynomial multiplication (i.e., convolution), to produce a correct inner product between two RLWE ciphertexts in the coefficient encoding, one of the ciphertexts needs to have its plaintext coefficients be in the *reverse order*. Homomorphically reversing the order of the plaintext coefficients in a ciphertext can be achieved by first converting the RLWE ciphertext into a set of LWE ciphertext and then pack the LWE ciphertexts in the reverse order. One slight optimization here is that, if we wish to take the inner product between two input RLWE ciphertexts, we can let the client encrypt one of the input RLWE ciphertext directly in the reversed order to avoid the above two encoding-switching conversions.

V. Ciphertext Level Management

Once transformed, the program is compiled into the `rlwe-lwe` dialects with proper ciphertext types and the associated homomorphic operators. However, this representation is still too abstract to be directly mapped to low-level FHE library

```
Algorithm 2: Level Management Algorithm
Input    : DAG of the program $(\mathcal{V}, \mathcal{E}) \in \mathcal{G}$
Input    : Maximum ciphertext level $\gamma$
Output   : Transformed DAG of the program $(\mathcal{V}, \mathcal{E}) \in \mathcal{G}$
▷ Initial Level Allocation
for $u \in \mathcal{V}$ do
    $u.level = \textsc{LevelIterator}(u)$
▷ Bootstrap Scheduling
for $u \in \mathcal{V}$ do
    if $u.level == i \cdot (\gamma + 1)$ for $i \in N_+$ then
        $u.level \leftarrow 1$
        ▷ Insert Extra Bootstrap
        Insert $u \leftarrow \textsc{MultModulusBootstrap}(u, \gamma)$
    else if $u.level > \gamma + 1$ then
        $u.level \leftarrow u.level \mod \gamma$
Procedure : $\textsc{LevelIterator}(u)$
Initialize $nextLvl \leftarrow 0$.
Initialize a list of level $\mathsf{subList}$.
for $op \in u.Uses$ do
    switch $op$ do
        case $RLWEMul$ do $nextLvl \leftarrow 1$;
        case $S2C$ do $nextLvl \leftarrow 2$;
        case $C2S$ do $nextLvl \leftarrow 2$;
        case $LUT$ do $nextLvl \leftarrow 0$;
    if $op! = LUT$ then
        $res \leftarrow op.Result$
        $res.level \leftarrow \textsc{LevelIterator}(res)$
        $\mathsf{subList}.append(res.level + nextLvl)$
$u.level \leftarrow max(\mathsf{subList})$
Return   : $u.level$
```

API, since there is no hint for the selections of encryption parameters and generations of evaluation keys. In what follows, we introduce our general level management procedure that allows HEIR to compile unbounded-depth FHE programs.

In the HEIR framework, we use the leveled [76] property of FHE ciphertexts to carry out arithmetic operations. The main drawback of leveled homomorphic encryption scheme is that, after performing heavy computations such as a homomorphic multiplication between two ciphertexts, the output ciphertext needs to receive a *rescaling* operator that drops a modulus in the residual number system (RNS) modulus chain to maintain a low ciphertext noise size. The total number of moduli in the RNS modulus chain is referred to as levels, and if only one modulus is left in the RNS modulus chain, we say that all levels are consumed. At this stage, no more ciphertext-to-ciphertext multiplications can be further performed. Hence, a severe limitation to the compilation over leveled homomorphic encryption schemes is that the compiler needs to know the depth of the circuit *a priori*, or the levels cannot be properly determined at compile time. To solve such level problem, we propose to schedule the ciphertext bootstrapping operator during the compilation FHE programs, such that the HEIR framework can compile unbounded-depth programs.

The real challenge for properly bootstrapping FHE ciphertexts is how to schedule the bootstrapping operator. On one hand, scheduling bootstrapping too frequently may severely degrade the performance of the FHE program. On the other hand, the program cannot be correctly evaluated with an insufficient amount of bootstrapping operators.

In HEIR, we propose a level management procedure sketched in Algorithm 2 to properly assign levels to the FHE ciphertexts and insert bootstrapping operators when necessary. In the

TABLE II
The Encryption Parameters

Ciphertext	Parameters	
LWE	n	4096
	$\log_2(q)$	48
RLWE	N	8192
	$\log_2(q_i)$	48
	Max$(\log_2(Q))$	192

beginning, we need to decide a maximum ciphertext level γ which can be seen as a hyperparameter to the compiler framework. To determine γ, we characterized the average gate bootstrapping frequency in a set of benchmark FHE applications that include sorting, image processing algorithms and K-Means clustering. We discover that a γ of 4 is sufficient in compiling most of the benchmarks without incurring extra bootstrapping. Since γ can be program-dependent in practice, we set the value of γ to be user-defined in HEIR. Then, since the number of each type of homomorphic operators has mostly been fixed at compile time through the transformation passes, we can simply go through the entire program and calculate the necessary levels for each of the ciphertext operators. According to Algorithm 2, on Line 1-3, we start a traversal for all variables in this program based on *Def-Use* chain by calling LevelIterator function. On Line 9-23, we first initialize the $nextLevel$ with 0 and update $nextLvl$ based on the next operators. For example, the C2S (i.e., the coefficient-to-slot encoding-switching [94]) and the S2C (i.e., the slot-to-coefficient encoding-switching) operators always consume two RNS levels [3] and the RlweMul operator (i.e., ciphertext-ciphertext multiplication) consumes one level. Each time some operator uses a variable (e.g., u) as its operand, we capture the result of the operator and call the LevelIterator function over the variable u. We increment the levels of u until u has no more uses or is only used by the LUT operator (since LUT is essentially a PBS). Note that there may be multiple independent uses of u. Hence, on Line 24, the level of u is determined by the maximum level amongst all of its independent uses. Finally, after the levels of all variables are determined, we insert the LUT operator based on our multi-modulus bootstrapping technique described in the appendix (Appendix E) to reset the levels of the ciphertext variables. As depicted on Line 7, if the level of a variable presents to be too large (i.e., $\geq \gamma$), we will schedule another LUT to ensure that all levels in the program is less than the pre-determined threshold.

VI. Evaluation

Throughout the evaluation, we mainly focus on comparing the effectiveness and performance of the proposed HEIR architecture. We mainly compare our HEIR framework with the state-of-the-art arithmetic-circuit and logic-circuit FHE compilers, i.e., EVA [23, 62], HECO [58] and Transpiler [21].

A. Evaluation Setup

The entire HEIR compiler framework is implemented using C++17 and compiled with clang 14.0.0 based on MLIR framework. Our bottom-level FHE library is an in-house

TABLE III
Comparison of homomorphic K-Means algorithm among HEIR, Transpiler and a hand-tuned implementation, where N denotes the number of data points, K denotes the number of centroids and d denotes dimensions of the data point

N	K	d	Implementaions	Latency (s)	
5	2	3	Transpiler [21]	24321.191	179×
			HEIR	135.634	1×
			Hand-Tuned	59.338	0.44×
10	2	3	Transpiler [21]	*Compilation fails*	–
			HEIR	248.823	1×
			Hand-Tuned	105.621	0.42×

TABLE IV
Running Time of homomorphic K-Means algorithm

Number of Data	Number of Centroids	Running Time (min)
128	2	49.89
	3	92.85
512	2	198.28
	3	369.98
1024	2	399.53
	3	798.21

TABLE V
Comparison of compile time and memory consumption between HEIR and other FHE compilers

Benchmarks	Compilers	Compilation Time	Memory Usage
Inner Product	EVA [23]	0.007 s	135 MB
	HECO [58]	0.078 s	74 MB
	HEIR	0.15 s	53 MB
Boxblur	EVA [23]	0.008 s	272 MB
	HECO [58]	0.078 s	98 MB
	HEIR	0.15 s	116 MB
Fibonacci	Transpiler [21]	39.28 s	308 MB
	HEIR	0.16 s	6.1 GB
K-Means	Transpiler [21]	974.98 s	718 MB
	HEIR	0.18 s	12.2 GB

implementation of the CKKS and TFHE schemes written in C++17 and compiled using clang 14.0.0. We conducted experiments for microbenchmarks and end-to-end applications on a single core of an Intel Xeon Gold 5318Y processor with 512GB of RAM.

The parameters used in HEIR for these evaluations are summarized in Table II, which achieves 128-bit security estimated by lwe-estimator [99]. To allow efficient CPU execution, we set the upper bound of the modulus $Q = \prod_{i=0}^{k-1} q_i$ used in RLWE ciphertexts to be 192 bits.

B. Microbenchmarks

Here, we first test HEIR on a set of selected microbenchmarks over both arithmetic and logic circuits. The results for arithmetic circuits are summarized in Section VI-B1 and that for logic circuit in Section VI-B2.

1) Arithmetic Circuit Evaluation: We demonstrate the speedup achieved by HEIR on three applications that are commonly used as building blocks in many practical ap-

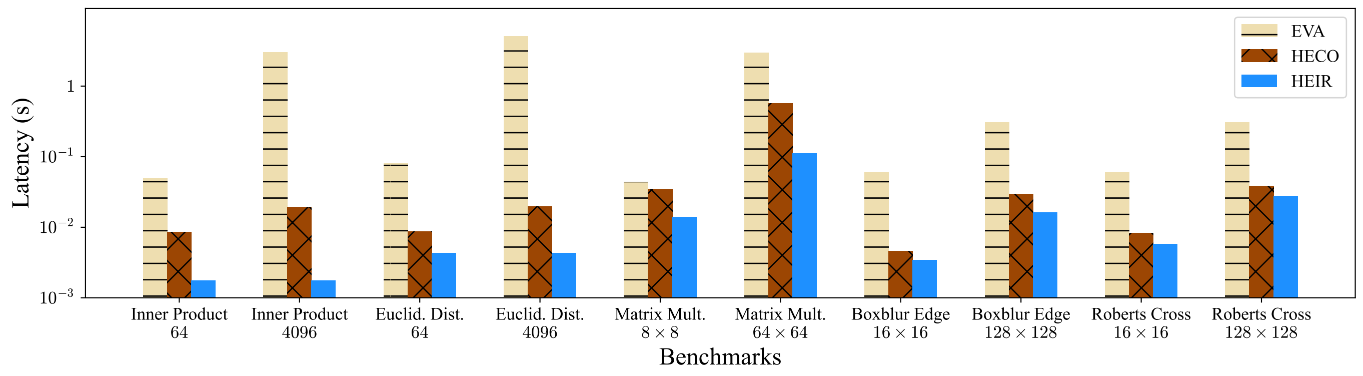


Fig. 5. Microbenchmark results for arithmetic-circuit programs with different vector/matrix/image size.

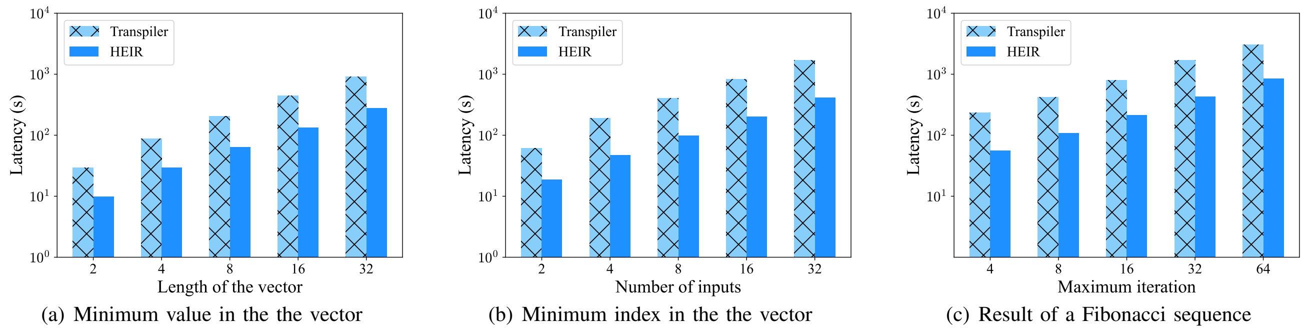


Fig. 6. Microbenchmark results for logic-circuit programs.

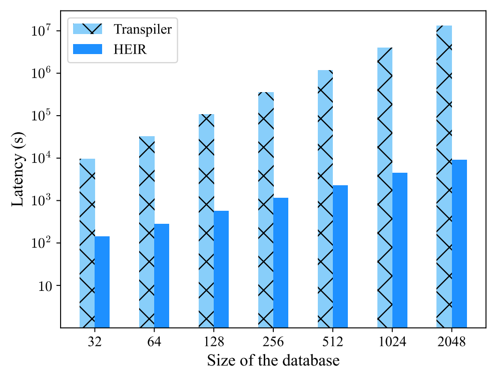


Fig. 7. Runtime of data analysis program for an encrypted database.

plications: inner product, Euclidean distance, matrix-vector multiplication, Boxblur filtering and Roberts Cross filtering. Note that EVA does not support the automatic vector batching. Therefore, the programmer has to manually implement the basic rotate-and-add approach (i.e., Appendix B) for most arithmetic benchmarks.

Although Transpiler [21] also supports the compilation of all three benchmarks, we note that logic FHE circuits are not good at handling arithmetic applications involving a large number of multiplication operations. In our experiments, we find that the program generated by Transpiler required approximately 6000 seconds to evaluate an inner product between two 32-element 16-bit vectors, which is unacceptable for practical uses. Therefore, we omit Transpiler in the benchmark comparisons over arithmetic circuit evaluations.

Runtime Latency: As shown in Figure 5 and Appendix Figure A2, using the coefficient encoding technique, the runtime latency of HEIR on *inner product* and *euclidean distance* is independent of the number of vector elements (up to the lattice dimension n). In comparison, the computation latency growths for EVA programs are linear to the number of vector elements. Overall, HEIR achieves a speedup of 3.4×–11× over the advanced compilation result for vector inner product, 1.4×–4.5× for vector Euclidean distance and 2×–5× for matrix-vector multiplication. In applications such as Boxblur and Roberts Cross, HEIR cannot use the coefficient encoding technique to optimize the programs. Nonetheless, HEIR achieves comparable latency with HECO through the implementation of SIMD batching optimizations, resulting in a speedup of 21-36× over EVA.

2) ***Logic Circuit Evaluation:*** We evaluated the efficiency of different compilers in evaluating non-polynomial functions in the logic circuit benchmarks. As existing arithmetic FHE compilers can only generate programs consisting of polynomial addition and multiplication operators, they cannot be used in evaluating logic circuits. Hence, the performance comparison is conducted between HEIR and Transpiler [21]. We demonstrate the speedup achieved by HEIR on three logic circuit benchmarks including vector min value, vector min index and Fibonacci sequence. Note that the application of the

Fibonacci sequence also exemplifies the capability of HEIR to incur extra TFHE multi-modulus bootstrapping for level management. When evaluating a Fibonacci sequence with a maximum iteration of 64, HEIR invokes the bootstrapping operator 44 times to increase the level of the ciphertexts, which is reduced by the repeated evaluation of the `if` statements.

Runtime Latency. As demonstrated in Figure 6, we show the runtime latency of the above three logic-circuit programs. Though the latency of the programs generated by both Transpiler and HEIR has linear complexity, the bit-level FHE computation paradigm used in Transpiler is still slower than the cross-scheme paradigm used in HEIR for evaluating the microbenchmarks. In summary, compared to Transpiler, HEIR achieves a speedup of $3.2\times$ over the min-value evaluation, $4.1\times$ over the min-index evaluation, and $3.6\times$ over the Fibonacci sequence program.

C. Evaluation on End-to-End Applications

Lastly, we evaluate two end-to-end programs that are frequently used in real-world applications.

1) Data Analysis in Homomorphic Database: Filter-aggregation is a common program for data analysis [7, 100, 101] which includes both arithmetic and logic operations. Here, we consider an encrypted database storing personal information in a company. Our goal is to calculate the overall salary of employees earning more than 4000 dollars per month. To achieve this, we need first to filter the employees whose salaries are more than 4000 dollars from the database, and then accumulate the filtered values to generate the sum. Figure 7 demonstrates the running latency of the compiled programs mentioned above. For a database recording information of 32 employees, Transpiler requires 2.85 hours to compute the result while HEIR achieves a speedup of 72× with 2.3 minutes.

2) End-to-end Homomorphic K-Means Evaluation: K-Means clustering [102] is a popular unsupervised machine-learning algorithm that also require both arithmetic and logic operations. The overall process can be characterized into two main steps. First, for each input data point, we need to homomorphically calculate the closest centroid associated with the point, i.e., the minimum distance function depicted in Figure 3(a). We need an arithmetic-logic conversion here because distances can be better calculated using arithmetic circuit, while determining the minimum distance is a logic operation. Second, we need to convert the logical comparison result into an arithmetic representation, as updating the centroids requires calculating the mean values of a certain number of points. A toy example of single-round K-Means evaluation is demonstrated in Table III. As observed in Table III, HEIR is 179× faster than Transpiler on the setting with 5 data points and 2 centroids. Furthermore, Transpiler cannot compile the K-Means algorithm on 10 data points with 2 centroids, for the overall circuit is too complicated for Transpiler to synthesize and optimize. In contrast, as demonstrated in Table IV, HEIR has the ability to compile K-Means on large datasets and achieve competitive latency to group 1024 data into 3 clusters within 14 hours. Under the setting of Table IV, HEIR invokes the conversion operators 390–3078 times per K-Means iteration under input data sizes of 128–1024 and 3 centroids.

In addition to Transpiler, we also compare HEIR with an in-house version of K-Means developed based on Open-PEGASUS [3]. Here, we hand-tuned the K-Means program and manually integrated various advanced FHE optimization techniques, such as flexible encoding methods and the identity key switching method [18], to accelerate program execution. We observe that HEIR still achieves comparable runtime latency against this hand-tuned version of the K-Means program.

Compilation Time and Memory Usage: As shown in Table V, we compare the compilation time of different FHE compilers and the memory usage of the generated programs on various benchmarks. The result demonstrates that HEIR achieves comparable efficiency with EVA and HECO to compile the input program in less than one second. On the contrary, due to the complicated Boolean circuit synthesis process, Transpiler takes tens to hundreds of seconds to compile a program. As for memory usage, HECO and HEIR achieve better memory overhead than EVA due to batch-related optimizations on arithmetic circuits. For logic and hybrid circuit evaluation, HEIR consumes more memory than Transpiler, for HEIR utilizes much more utility keys to speedup program evaluation (e.g., the Galois keys and relinearization keys). In other words, comparing to Transpiler, HEIR trades memory efficiency for better runtime latency.

Remark: While HEIR makes a significant step in hybrid circuit compilation, the challenge in homomorphic bit extraction for plaintext encrypted using $\mathbb{Z}_p$ for $p > 2$ remains to be addressed. For example, if two integers `a` and `b` are encrypted just as integers in ciphertexts, homomorphic bit-wise operations like `a & b` cannot be compiled by HEIR. At the current stage, `a` and `b` need to be encrypted bit-by-bit to carry out homomorphic bit-wise operations, which adds significant communication and computation burdens. In the future, advanced digit decomposition techniques such as [103] will be integrated to extend the compilation capability of HEIR for bit-level operations.

VII. Conclusion

In this work, we proposed HEIR, a compiler framework that can translate unbounded-depth FHE programs containing both arithmetic and logic operations. By introducing the HEIR IR consisting of a set of FHE-specific dialects, we develop multiple lowering passes to directly convert high-level programs into bottom-level FHE operators without the assistance of DSLs. In the experiment, we show that, by properly scheduling conversion and bootstrapping operators, HEIR can compile programs that run up to 72–179× faster than that generated by existing compilers on end-to-end real-world applications.

Acknowledgement

This work was partially supported by the National Key R&D Program of China (2021YFB2700200), the National Natural Science Foundation of China (62002006, 62202028, 62172025, U21B2021, 61932014, 61972018, 61972019, U2241213), and the CCF-Huawei Populus Euphratica project. Zhenyu Guan is the corresponding author.

References

[1] L. Folkerts, C. Gouert, and N. G. Tsoutsos, "Redsec: Running encrypted discretized neural networks in seconds," in *NDSS*, 2023.

[2] S. Bian, D. Kundi, K. Hirozawa, W. Liu, and T. Sato, "APAS: application-specific accelerators for rlwe-based homomorphic linear transformations," *IEEE Trans. Inf. Forensics Secur.*, vol. 16, pp. 4663–4678, 2021.

[3] W. Lu, Z. Huang, C. Hong, Y. Ma, and H. Qu, "PEGASUS: bridging polynomial and non-polynomial evaluations in homomorphic encryption," in *S&P*, pp. 1057–1073, 2021.

[4] Z. Huang, W. Lu, C. Hong, and J. Ding, "Cheetah: Lean and fast secure two-party deep neural network inference," in *USENIX Security*, pp. 809–826, 2022.

[5] K. Cong, D. Das, J. Park, and H. V. L. Pereira, "Sortinghat: Efficient private decision tree evaluation via homomorphic encryption and transciphering," in *CCS*, pp. 563–577, 2022.

[6] Q. Lou and L. Jiang, "HEMET: A homomorphic-encryption-friendly privacy-preserving mobile neural network architecture," in *ICML*, pp. 7102–7110, 2021.

[7] X. Ren, L. Su, Z. Gu, S. Wang, F. Li, Y. Xie, S. Bian, C. Li, and F. Zhang, "HEDA: multi-attribute unbounded aggregation over homomorphically encrypted database," *Proc. VLDB Endow.*, pp. 601–614, 2022.

[8] K. Matsuoka, R. Banno, N. Matsumoto, T. Sato, and S. Bian, "Virtual secure platform: A five-stage pipeline processor over TFHE," in *USENIX Security*, pp. 4007–4024, 2021.

[9] E. Lee, J.-W. Lee, J. Lee, Y.-S. Kim, Y. Kim, J.-S. No, and W. Choi, "Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions," in *ICML*, pp. 12403–12422, 2022.

[10] "Microsoft SEAL (release 3.7)." https://github.com/Microsoft/SEAL, Sept. 2021. Microsoft Research.

[11] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference service for neural networks," in *USENIX Security*, pp. 2505–2522, 2020.

[12] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow2: Practical 2-party secure inference," in *CCS*, pp. 325–342, 2020.

[13] J. H. Cheon, D. Kim, D. Kim, H. Lee, and K. Lee, "Numerical method for comparison on homomorphically encrypted numbers," in *ASIACRYPT*, pp. 415–445, 2019.

[14] J. H. Cheon, D. Kim, and D. Kim, "Efficient homomorphic comparison methods with optimal complexity," in *ASIACRYPT*, pp. 221–256, 2020.

[15] J. Lee, E. Lee, Y. Lee, Y. Kim, and J. No, "High-precision bootstrapping of RNS-CKKS homomorphic encryption using optimal minimax polynomial approximation and inverse sine function," in *EUROCRYPT*, pp. 618–647, 2021.

[16] H. Chen, I. Chillotti, and Y. Song, "Improved bootstrapping for approximate homomorphic encryption," in *EUROCRYPT*, pp. 34–54, 2019.

[17] Y. Bae, J. H. Cheon, W. Cho, J. Kim, and T. Kim, "META-BTS: bootstrapping precision beyond the limit," in *CCS*, pp. 223–234, 2022.

[18] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: fast fully homomorphic encryption over the torus," *J. Cryptol.*, vol. 33, no. 1, pp. 34–91, 2020.

[19] L. Ducas and D. Micciancio, "FHEW: bootstrapping homomorphic encryption in less than a second," in *EUROCRYPT*, pp. 617–640, 2015.

[20] N. P. Smart and F. Vercauteren, "Fully homomorphic SIMD operations," *Des. Codes Cryptogr.*, vol. 71, no. 1, pp. 57–81, 2014.

[21] S. Gorantala, R. Springer, and S. P. , et. al., "A general purpose transpiler for fully homomorphic encryption," *IACR ePrint*, 2021.

[22] C. Gouert and N. G. Tsoutsos, "Romeo: Conversion and evaluation of HDL designs in the encrypted domain," in *DAC*, pp. 1–6, 2020.

[23] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "EVA: an encrypted vector arithmetic language and compiler for efficient homomorphic computation," in *PLDI*, pp. 546–561, 2020.

[24] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. E. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "CHET: an optimizing compiler for fully-homomorphic neural-network inferencing," in *PLDI*, pp. 142–156, 2019.

[25] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski, "nGraph-HE: a graph compiler for deep learning on homomorphically encrypted data," in *CF*, pp. 3–13, 2019.

[26] T. van Elsloo, G. Patrini, and H. Ivey-Law, "Sealion: a framework for neural network inference on encrypted data," *CoRR*, 2019.

[27] I. Damgård, C. Orlandi, and M. Simkin, "Yet another compiler for active security or: Efficient mpc over arbitrary rings," in *CRYPTO*, pp. 799–829, 2018.

[28] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof, "Sublinear gmw-style compiler for mpc with preprocessing," in *CRYPTO*, pp. 457–485, 2021.

[29] M. Abspoel, A. Dalskov, D. Escudero, and A. Nof, "An efficient passive-to-active compiler for honest-majority mpc over rings," in *ACNS*, pp. 122–152, 2021.

[30] D. Malkhi, N. Nisan, B. Pinkas, Y. Sella, *et al.*, "Fairplay-secure two-party computation system.," in *USENIX Security*, p. 9, 2004.

[31] A. Ben-David, N. Nisan, and B. Pinkas, "FairplayMP: a system for secure multi-party computation," in *CCS*, pp. 257–266, 2008.

[32] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen, "Asynchronous multiparty computation: Theory and implementation," in *PKC*, pp. 160–179, 2009.

[33] Y. Zhang, A. Steele, and M. Blanton, "PICCO: a general-purpose compiler for private distributed computation," in *CCS*, pp. 813–826, 2013.

[34] M. Franz, A. Holzer, S. Katzenbeisser, C. Schallhart, and H. Veith, "CBMC-GC: An ansi c compiler for secure two-party computations," in *CC*, pp. 244–249, 2014.

[35] A. Rastogi, M. A. Hammer, and M. Hicks, "Wysteria: A programming language for generic, mixed-mode multiparty computations," in *S&P*, pp. 655–670, 2014.

[36] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," in *S&P*, pp. 359–376, 2015.

[37] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor, "Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation," in *Euro S&P*, pp. 112–127, 2016.

[38] X. Wang, A. J. Malozemoff, and J. Katz, "Emp-toolkit: Efficient multiparty computation toolkit," 2016.

[39] D. Demmler, S. Katzenbeisser, T. Schneider, T. Schuster, and C. Weinert, "Improved circuit compilation for hybrid mpc via compiler intermediate representation," *IACR Cryptol. ePrint Arch.*, 2021.

[40] T. Araki, A. Barak, J. Furukawa, M. Keller, Y. Lindell, K. Ohara, and H. Tsuchida, "Generalizing the spdz compiler for other protocols," in *CCS*, pp. 880–895, 2018.

[41] D. Demmler, T. Schneider, and M. Zohner, "ABY-A framework for efficient mixed-protocol secure two-party computation.," in *NDSS*, 2015.

[42] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic, "Sok: General purpose compilers for secure multi-party computation," in *S&P*, pp. 1220–1237, 2019.

[43] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos, "Sepia: Privacy-preserving aggregation of multi-domain network events and statistics," *Network*, vol. 1, no. 101101, pp. 15–32, 2010.

[44] P. Mohassel and P. Rindal, "ABY3: A mixed protocol framework for machine learning," in *CCS*, pp. 35–52, 2018.

[45] S. Bian, W. Jiang, Q. Lu, Y. Shi, and T. Sato, "NASS: Optimizing secure inference via neural architecture search," in *ECAI*, 2020.

[46] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, and T. Schneider, "HyCC: Compilation of hybrid protocols for practical secure computation," in *CCS*, pp. 847–861, 2018.

[47] T. Heldmann, T. Schneider, O. Tkachenko, and et. al., "llvm-based circuit compilation for practical secure computation," in *ACNS*, pp. 99–121, 2021.

[48] M. Keller, "Mp-spdz: A versatile framework for multi-party computation," in *CCS*, pp. 1575–1590, 2020.

[49] L. Braun, D. Demmler, T. Schneider, and O. Tkachenko, "Motion–a framework for mixed-protocol multi-party computation," *ACM Transactions on Privacy and Security*, vol. 25, no. 2, pp. 1–35, 2022.

[50] F. Kerschbaum, T. Schneider, and A. Schröpfer, "Automatic protocol selection in secure two-party computations," in *ACNS*, pp. 566–584, 2014.

[51] E. Pattuk, M. Kantarcioglu, H. Ulusoy, and B. Malin, "CheapSMC: A framework to minimize secure multiparty computation cost in the cloud," in *DBSec*, pp. 285–294, 2016.

[52] S. Carpov, P. Dubrulle, and R. Sirdey, "Armadillo: A compilation chain for privacy preserving applications," in *SCC@ASIACCS*, pp. 13–19, 2015.

[53] E. Crockett, C. Peikert, and C. Sharp, "ALCHEMY: A language and compiler for homomorphic encryption made easy," in *CCS*, pp. 1020–1037, 2018.

[54] E. Chielle, O. Mazonka, H. Gamil, and M. Maniatakos, "Accelerating fully homomorphic encryption by bridging modular and bit-level arithmetic," in *ICCAD*, 2022.

[55] A. Viand and H. Shafagh, "Marble: Making fully homomorphic encryption accessible to all," in *WAHC*, pp. 49–60, 2018.

[56] D. Archer, J. Trilla, J. Dagit, A. Malozemoff, Y. Polyakov, K. Rohloff, and G. Ryan, "RAMPARTS: A programmer-friendly system for building homomorphic encryption applications," in *WAHC*, pp. 57–68, 2019.

[57] M. Cowan, D. Dangwal, A. Alaghi, C. Trippel, V. T. Lee, and B. Reagen, "Porcupine: a synthesizing compiler for vectorized homomorphic encryption," in *PLDI*, pp. 375–389, 2021.

[58] A. Viand, P. Jattke, M. Haller, and A. Hithnawi, "HECO: automatic code optimizations for efficient fully homomorphic encryption," *CoRR*, 2022.

[59] Y. Lee, D. Kim, D. Lee, and H. Kim, "Elasm: Error-latency-aware scale management for fully homomorphic encryption," in *USENIX Security*, pp. 1–15, 2023.

[60] A. Patra, T. Schneider, A. Suresh, and H. Yalame, "Aby2. 0: Improved mixed-protocol secure two-party computation.," in *USENIX Security*, pp. 2165–2182, 2021.

[61] E. Chielle, O. Mazonka, N. G. Tsoutsos, and M. Maniatakos, "E^3: A framework for compiling C++ programs with encrypted operands," *IACR ePrint*, 2018.

[62] S. Chowdhary, W. Dai, K. Laine, and O. Saarikivi, "EVA improved: Compiler and extension library for CKKS," in *WAHC*, pp. 43–55, 2021.

[63] Y. Lee, S. Heo, S. Cheon, S. Jeong, C. Kim, E. Kim, D. Lee, and H. Kim, "HECATE: performance-aware scale optimization for homomorphic encryption compiler," in *CGO*, pp. 193–204, 2022.

[64] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR ePrint*, 2012.

[65] J. H. Cheon, A. Kim, M. Kim, and Y. S. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *ASIACRYPT*, pp. 409–437, 2017.

[66] I. Chillotti, M. Joye, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks," in *CSCML*, pp. 1–19, 2021.

[67] J. Alperin-Sheriff and C. Peikert, "Faster bootstrapping with polynomial error," in *CRYPTO*, pp. 297–314, 2014.

[68] C. Boura, N. Gama, M. Georgieva, and D. Jetchev, "Simulating homomorphic evaluation of deep learning predictions," in *CSCML*, pp. 212–230, 2019.

[69] P. Clet, M. Zuber, A. Boudguiga, R. Sirdey, and C. Gouy-Pailler, "Putting up the swiss army knife of homomorphic

calculations by means of TFHE functional bootstrapping," *IACR ePrint*, 2022.

[70] K. Kluczniak and L. Schild, "FDFB: full domain functional bootstrapping towards practical fully homomorphic encryption," *IACR CHES*, vol. 2023, no. 1, pp. 501–537, 2023.

[71] Z. Yang, X. Xie, H. Shen, S. Chen, and J. Zhou, "TOTA: fully homomorphic encryption with smaller parameters and stronger security," *IACR ePrint*, 2021.

[72] S. Carpov, M. Izabachène, and V. Mollimard, "New techniques for multi-value input homomorphic evaluation and applications," in *CT-RSA*, pp. 106–126, 2019.

[73] I. Chillotti, D. Ligier, J. Orfila, and S. Tap, "Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE," in *ASIACRYPT*, pp. 670–699, 2021.

[74] A. Guimarães, E. Borin, and D. F. Aranha, "Revisiting the functional bootstrap in TFHE," *IACR CHES*, vol. 2021, no. 2, pp. 229–253, 2021.

[75] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *CRYPTO*, pp. 868–886, 2012.

[76] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *ITCS*, pp. 309–325, 2012.

[77] H. Chabanne, A. de Wargny, J. Milgram, C. Morel, and E. Prouff, "Privacy-preserving classification on deep neural network," *IACR ePrint*, 2017.

[78] E. Hesamifard, H. Takabi, and M. Ghasemi, "Cryptodl: Deep neural networks over encrypted data," *CoRR*, 2017.

[79] X. Jiang, M. Kim, K. E. Lauter, and Y. Song, "Secure outsourced matrix computation and application to neural networks," in *CCS*, pp. 1209–1222, 2018.

[80] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," in *CRYPTO*, pp. 850–867, 2012.

[81] C. Gentry, S. Halevi, and N. P. Smart, "Fully homomorphic encryption with polylog overhead," in *EUROCRYPT*, pp. 465–482, 2012.

[82] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption," in *EUROCRYPT*, pp. 360–384, 2018.

[83] J. Bossuat, C. Mouchet, J. R. Troncoso-Pastoriza, and J. Hubaux, "Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys," in *EUROCRYPT*, pp. 587–617, 2021.

[84] Y. Lee, J. Lee, Y. Kim, Y. Kim, J. No, and H. Kang, "High-precision bootstrapping for approximate homomorphic encryption by error variance minimization," in *EUROCRYPT*, pp. 551–580, 2022.

[85] C. S. Jutla and N. Manohar, "Sine series approximation of the mod function for bootstrapping of approximate HE," in *EUROCRYPT*, pp. 491–520, 2022.

[86] H. L. Garner, "The residue number system," *IRE Trans. Electron. Comput.*, vol. 8, no. 2, pp. 140–147, 1959.

[87] J. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, "A full RNS variant of FV like somewhat homomorphic encryption schemes," in *SAC*, pp. 423–442, 2016.

[88] S. Halevi, Y. Polyakov, and V. Shoup, "An improved RNS variant of the BFV homomorphic encryption scheme," in *CT-RSA*, pp. 83–105, 2019.

[89] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full RNS variant of approximate homomorphic encryption," in *SAC*, pp. 347–368, 2018.

[90] C. Boura, N. Gama, M. Georgieva, and D. Jetchev, "CHIMERA: combining ring-lwe-based fully homomorphic encryption schemes," *J. Math. Cryptol.*, pp. 316–338, 2020.

[91] Q. Lou, B. Feng, G. C. Fox, and L. Jiang, "Glyph: Fast and accurately training deep neural networks on encrypted data," in *NeurIPS*, pp. 9193–9202, 2020.

[92] D. Micciancio and J. Sorrell, "Ring packing and amortized FHEW bootstrapping," in *ICALP*, pp. 100:1–100:14, 2018.

[93] H. Chen, W. Dai, M. Kim, and Y. Song, "Efficient homomorphic conversion between (ring) LWE ciphertexts," in *ACNS*, pp. 460–479, 2021.

[94] K. Han, M. Hhan, and J. H. Cheon, "Improved homomorphic discrete fourier transforms and FHE bootstrapping," *IEEE Access*, vol. 7, pp. 57361–57370, 2019.

[95] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. A. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: scaling compiler infrastructure for domain specific computation," in *CGO*, pp. 2–14, 2021.

[96] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, "Polygeist: Raising C to polyhedral MLIR," in *PACT*, pp. 45–59, 2021.

[97] S. Halevi and V. Shoup, "Algorithms in helib," in *CRYPTO*, pp. 554–571, 2014.

[98] S. Halevi and V. Shoup, "Design and implementation of helib: a homomorphic encryption library," *IACR ePrint*, 2020.

[99] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," *J. Math. Cryptol.*, vol. 9, no. 3, pp. 169–203, 2015.

[100] Y. Wang and K. Yi, "Secure yannakakis: Join-aggregate queries over private data," in *ICMD*, pp. 1969–1981, 2021.

[101] T. Hackenjos, F. Hahn, and F. Kerschbaum, "SAGMA: secure aggregation grouped by multiple attributes," in *SIGMOD*, pp. 587–601, 2020.

[102] J. MacQuuen, "Some methods for classification and analysis of multivariate observation," in *Berkley Symposium on Mathematical Statistics and Probability*, pp. 281–297, 1967.

[103] S. Ma, T. Huang, A. Wang, and X. Wang, "Fast and accurate: Efficient full-domain functional bootstrap and digit decomposition for homomorphic computation," *IACR ePrint*, 2023.

[104] C. Juvekar, V. Vaikuntanathan, and A. P. Chandrakasan, "GAZELLE: A low latency framework for secure neural network inference," in *USENIX Security*, pp. 1651–1669, 2018.

Appendix Table A1
NATIVE C INSTRUCTIONS SUPPORTED IN HEIR

Instructions	Description
`+ +=`	Addition
`- -=`	Substraction
`*/* =`	Multiplication
`=`	Assignment
`==`	Equality comparison
$> \geq$	Greater than operator
$< \leq$	Less than operator
`a[i]`	Index for single/two-dimensional array
`if else`	If... Else statement
`for`	For Loop statement
`Function`	Other instructions can be supported by function call

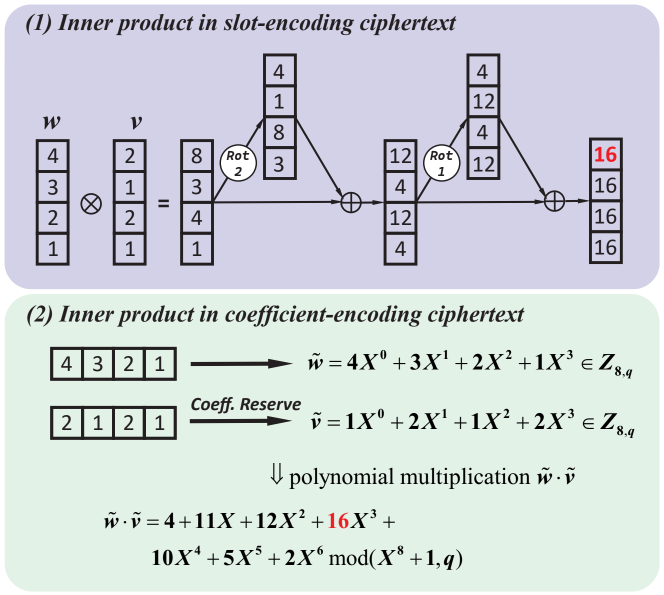
Algorithm 3: Plaintext to Ciphertext Transformation Pass

```
Input  : DAG of the program $(\mathcal{V}, \mathcal{E}) \in \mathcal{G}$
Output : transformed DAG of the program $(\mathcal{V}, \mathcal{E}) \in \mathcal{G}$
1  for $arg \in Inputs$ do
2      $arg.type \leftarrow$ TypeTransform($arg$)
3      OpIterator($arg$)
PROCEDURE : OpIterator($u$)
4  for $op \in u.Uses$ do
5      if $op.getResult \notin \emptyset$ then
6          $res \leftarrow op.getResult$
7          $res \leftarrow$ TypeTransform($res$)
           ▷ Encode Constants
8          for $u' \in op.Operands$ do
9              if $u'.type ==$ Float then
10                 $u'.type \leftarrow$ PlainInt
11                 Insert fhe.Encode for $u'$
12             else if $u'.type ==$ MemRef then
13                 $u'.type \leftarrow$ PlainPoly
14                 Insert fhe.Encode for $u'$
15         if $res.Uses \notin \emptyset$ then
16             $res \leftarrow$ OpIterator($res$)
▷ Transform to FHE Types
PROCEDURE : TypeTransform($u$)
17 if $u.type ==$ Float then
18     $u.type \leftarrow$ FHEFloat
19 else if $u.type ==$ MemRef then
20     if $u.dim == 1$ then $u.type \leftarrow$ FHEVector;
21     else if $u.dim == 2$ then $u.type \leftarrow$ FHEMatrix;
```

APPENDIX

A. Type and Operator Definition in HEIR framework

This section summarizes the types and operators supported and defined in HEIR framework. Table A1 lists the available native C instructions for the input programs for HEIR. In general, in an input program, only additions, subtractions and multiplications are considered arithmetic instructions which will be transformed further SIMD optimizations. Meanwhile, other instructions such as comparisons and `if` statements are considered as logic instructions that will be compiled and implemented using TFHE PBS. For operators not listed in Table A1, developers can define an external function call to implement the particular operator, and HEIR will translate this function call into a LUT implemented by TFHE PBS. Table A2 and Table A3 lists the types and operators defined in HEIR.


(1) Inner product in slot-encoding ciphertext
w
v
Rot 2
Rot 1
(2) Inner product in coefficient-encoding ciphertext
Coeff. Reserve
$\tilde{w} = 4X^0 + 3X^1 + 2X^2 + 1X^3 \in Z_{8,q}$
$\tilde{v} = 1X^0 + 2X^1 + 1X^2 + 2X^3 \in Z_{8,q}$
⇓ polynomial multiplication $\tilde{w} \cdot \tilde{v}$
$\tilde{w} \cdot \tilde{v} = 4 + 11X + 12X^2 + 16X^3 + 10X^4 + 5X^5 + 2X^6 \bmod (X^8 + 1, q)$

Appendix Figure A1. Toy example of how inner product of two ciphertexts is performed in slot and coefficient encoding formats.

B. Illustration of the Rotate-and-Add Pattern

we illustrated that the use of coefficient encoding ciphertexts can significantly accelerate the efficiency of the program, especially in a mix-scheme paradigm.

Rotate-and-Sum *vs* **Coeff-Convolution:** Figure A1 presents different evaluation methods for inner product evaluation in slot and coefficient encoding ciphertexts. For slot-encoding ciphertext, slot-wise multiplication can be performed between RLWE ciphertexts. Afterwards, using the rotation technique of AFHE, the result is evaluated through an accumulation procedure. For a developer familiar with FHE, the number of rotations in this program can be optimized [58, 97] from $O(N)$ to $O(\log N)$ as demonstrated. Conversely, the efficiency of inner product can be further optimized through coefficient encoding methods [2, 4]. Based on the convolution property of polynomial multiplication, the result of two l-element vectors is encoded in the l-th coefficient of the product polynomial, which can be easily extracted. Note that the plaintext order must be preserved in one of the vectors through the SAMPLEEXTRACT and REPACK algorithm to ensure correctness.

C. Motivation for Mini-Repack Algorithm

As demonstrated in Figure 4, operators including SAMPLE-EXTRACT (transform an RLWE ciphertext to a set of LWE ciphertexts) and REPACK (convert a set of LWE ciphertexts back to an RLWE ciphertext) are essential to be inserted into the program when performing scheme-switching conversions between the LWE and RLWE ciphertexts. Although the SAMPLEEXTRACT operator is relatively lightweight, REPACK can be much heavier in terms of computational costs, depending on the exact parameters and implementation. More specifically, most (if not all) existing REPACK algorithms become fast only when the number of items to be packed is large [93]. However, what we observe is that many practical application

Appendix Table A2
TYPES OF VALUES DEFINED IN HEIR

Dialect	Type	Attributes	Description
FHE	`Float`	—	A floating point value to interface with high-level `Arith` IR.
	`MemRef`	Size/ElementType	Type for multi-dimensional vector to interface with high-level `MemRef` IR.
	`PlainInt`	Level	An encoded plaintext for scalar value.
	`PlainPoly`	Level	An encoded plaintext vector.
	`FHEFloat`	—	An encrypted floating point value.
	`FHEVector`	Size	An encrypted one-dimensional vector.
	`FHEMatrix`	Shape	An encrypted two-dimensional matrix.
LWE	`LWECipher`	Level	A LWE ciphertext which encrypts a single plaintext.
	`LWECipherVec`	Size	A one-dimensional vector, each element is composed of `LWECipher` .
	`LWECipherMat`	Shape	A two-dimensional matrix, each element is composed of `LWECipher` .
RLWE	`RLWECipher`	Level/Size/Encoding	A RLWE ciphertext which encrypts a plaintext vector.
	`RLWECipherVec`	Size	A two-dimensional matrix, each row comprises `RLWECipher` .

Appendix Table A3
OPERATORS DEFINED IN HEIR

Dialect	Operator	Signature
fhe	*Encode*	`PlainInt` → `FHEFloat`
	Add	`FHEFloat` → `FHEFloat`
	Sub	`FHEFloat` → `FHEFloat`
	Mul	`FHEFloat` → `FHEFloat`
	FunCall	(`FHEFloat`, ..., `FHEFloat`) → `FHEFloat`
	Load	`FHEVector` , `index` → `FHEFloat`
	Store	`FHEFloat`, `FHEVector` , `index` → `FHEVector`
lwe	*LWEEncode*	`Float` → `PlainInt`
	LWEAdd	`LWECipher`, `LWECipher` → `LWECipher`
	LWEAddPlain	`LWECipher`, `PlainInt` → `LWECipher`
	LWESub	`LWECipher`, `LWECipher` → `LWECipher`
	LWESubPlain	`LWECipher`, `PlainInt` → `LWECipher`
	LUT	`LWECipher` → `LWECipher`
	Load	`LWECipherMat`, `index` → `LWECipher`
	Store	`LWECipher`, `LWECipherVec`, `index` → `LWECipherVec`
	Repack	`LWECipherVec` → `RLWECipher`
rlwe	*RLWEEncode*	`Float` → `PlainPoly`
	RLWEAdd	`RLWECipher`, `RLWECipher` → `RLWECipher`
	RLWEAddPlain	`RLWECipher`, `RLWECipher` ← `RLWECipher`
	RLWESub	`RLWECipher`, `RLWECipher` → `RLWECipher`
	RLWESUbPlain	`RLWECipher`, `RLWECipher` ← `RLWECipher`
	RLWEMul	`RLWECipher`, `RLWECipher`, → `RLWECipher`
	RLWEMulPlain	`RLWECipher`, `PlainPoly` → `RLWECipher`
	S2C	`RLWECipher` → `RLWECipher`
	C2S	`RLWECipher` → `RLWECipher`
	Reverse	`RLWECipher` → `RLWECipher`
	Load	`RLWECipherVec`, `index` → `RLWECipher`
	Store	`RLWECipher`, `RLWECipherVec`, `index` → `RLWECipherVec`
	Extract	`RLWECipher` → `LWECipherVec`

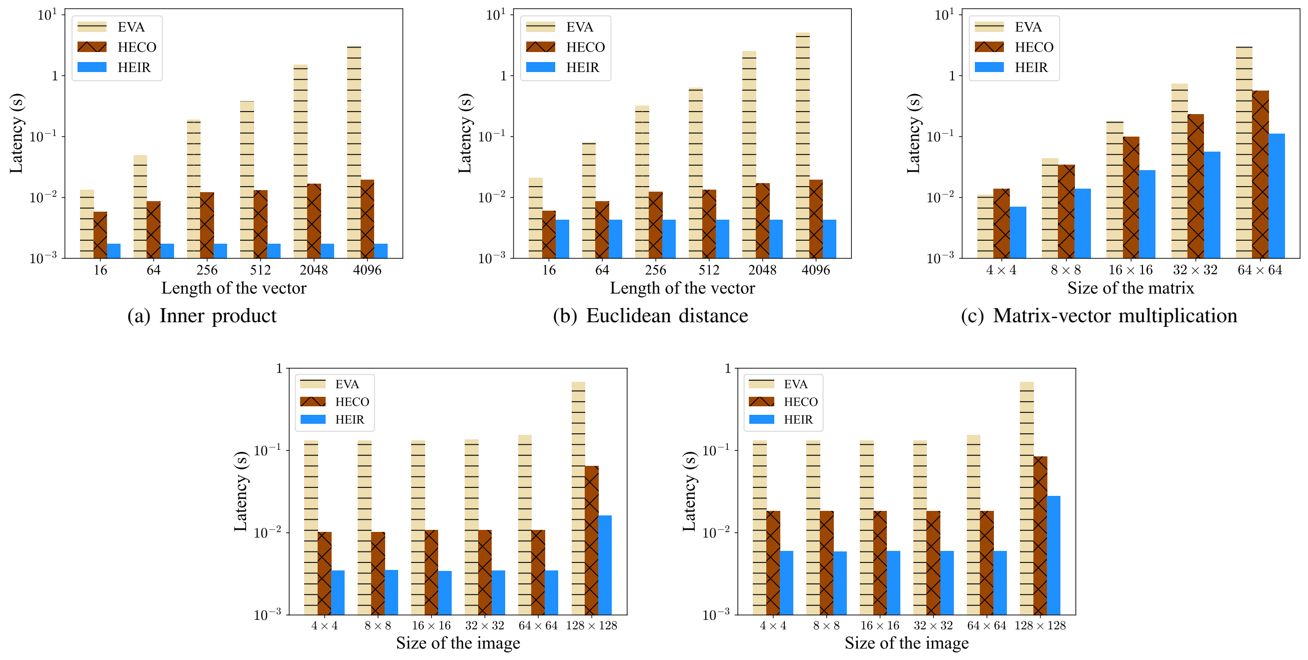
programs are not massively parallel, and cannot fully leverage the SIMD properties of the underlying RLWE ciphertext (tens of thousands). Therefore, in the process of designing HEIR, we see a need of small-number REPACK algorithm, i.e., an LWE to RLWE switching algorithm that is efficient when packing a small number of ciphertexts.

D. *The Algorithm of Encoding Pass*

The algorithm is fairly easy: as shown on Line 1–4 in Algorithm 4, we focus on matching patterns of accumulations and inner products in the target programs. We point out that, when the RLWE ciphertext is in the slot encoding, both accumulations and inner products need to be implemented through the standard rotate-and-add algorithm [94, 104]. Hence, once we detect the pattern of rotate-and-add (e.g., on Line 5 and 12 in Algorithm 4), we can replace the entire code block by a simple multiplication between RLWE ciphertexts in the coefficient encoding, and schedule encoding-switching conversion operators if needed. Here, recall that accumulating the encrypted elements in the given RLWE ciphertext is equivalent to taking the inner product between the given RLWE ciphertext all-1 vector.

E. *The Multi-Modulus Bootstrapping Algorithm*

Here, we describe out multi-modulus bootstrapping algorithm. In multi-modulus bootstrapping, the input LWE ciphertext with large noise in a smaller modulus q is transformed to a



(a) Inner product

(b) Euclidean distance

(c) Matrix-vector multiplication

(d) Boxblur filtering

(e) Roberts cross filtering

Appendix Figure A2. Complete experiment results for arithmetic-circuit benchmarks with different vector/matrix/image size.

Algorithm 4: Encoding Pass

```
Input     : DAG of the program $(\mathcal{V},\mathcal{E}) \in \mathcal{G}$
Output    : transformed DAG of the program $(\mathcal{V},\mathcal{E}) \in \mathcal{G}$
1  for $u \in \mathcal{V} \wedge u.type = \texttt{RLWECipher}$ do
2      AccumulationPattern($u$)
3      InnerProductPattern($u$)
Procedure : AccumulationPattern($u$)
4  if ExistRotAddSuquence($u$) then
5      Set Encoding Method of $u$ to coefficient
6      $pt \leftarrow CoeffEncode([1,1,1,...,1])$
7      Insert $res \leftarrow RLWEMulPlain(u, pt)$
8      Delete Rotate-Add sequence
Procedure : InnerProductPattern($u$)
9  $op \leftarrow u.DefiningOp$
10 if $op == RLWEMul$ then
11     if ExistRotAddSuquence($u$) then
12         $ct1, ct2 \leftarrow op.Operands$
13         Set Encoding Method of $ct1, ct2$ to coefficient
14         $res \leftarrow Extract(n.Vecsize)$
15         Delete Rotate-Add sequence
```

Algorithm 5: Multi-Modulus Bootstrapping

```
Input     : A LWE Ciphertext $(b,\mathbf{a}) = \mathsf{LWE}^{n,q}_{\mathbf{s}}(\Delta m)$, where
            $||\lfloor \Delta m \rceil|| < q/4$ and $\mathbf{s} \in \{0,1\}^n$.
Input     : A look-up table function $T(x) = \mathbb{R} \rightarrow \mathbb{R}$.
Input     : Pre-determined output ciphertext level $k$.
Output    : A LWE Cipher $ct = \mathsf{LWE}^{n,Q}_{\mathbf{s}}\{T(\lfloor \Delta m \rceil)\}$.
1  Key Generation Stage: Generate bootstrapping keys $BK_i$, for
   $i \in \mathbb{Z}_n$, where $BK_i = \mathsf{RGSW}^{n,Q}_{\tilde{s}}(\mathbf{s}[i])$, where $Q = \prod_{i=0}^{k-1} q_i$.
   ▷ FHE Computation Stage
2  Define $\eta_j = \frac{jq}{2n\Delta}$ and a look-up table polynomial $\tilde{f} \in R_{n,q}$ where
   $f_j = \begin{cases} \lfloor \Delta T(0) \rceil & j = 0 \\ \lfloor \Delta T(\eta_j) \rceil & 1 \leq j \leq \frac{n}{2} \\ \lfloor \Delta - T(\eta_{j-n}) \rceil & \frac{n}{2} < j < n \end{cases}$
3  Define $b \leftarrow \lfloor \frac{2n}{q} b \rceil$ and $\mathbf{a} \leftarrow \lfloor \frac{2n}{q} \mathbf{a} \rceil$.
4  Initialize $tv \in R_{n,Q}$ and RNS representation of $tv$ is
   $(tv^{(0)}, tv^{(1)}, ..., tv^{k-1})$, where $tv^{(i)} = \sum_{j=0}^{n-1} \hat{f}_j X^j$ and
   $\hat{f}_j = f_j \bmod q_i$.
5  Initialize $ACC \leftarrow (tv \cdot X^{b \bmod n}, 0) \in \mathsf{RLWE}^{N,Q}_{\tilde{S}}(.)$.
6  for $i = 0$ to $n-1$ do
7      $ACC \leftarrow ACC + (X^{-\mathbf{a}[i]} - 1) \cdot (ACC \odot BK_i)$
Return    : SampleExtract($ACC, 0$)
```

ciphertext with smaller noise in larger modulus $Q = \prod_{i=0}^{k-1} q_i$. The concrete algorithm is demonstrated in Algorithm 5. On Line 2-3, as the original FHEW/TFHE bootstrapping, we generate a look-up table polynomial $\tilde{f}$ based on a given function $T(x)$. Specifically on Line 4-5, the test vector tv is initialized modulus $Q = \prod_{i=0}^{k-1} q_i$ based on a given level k rather than the original q. Subsequently, on Line 6-8, blind rotation is performed on this multi-modulus accumulator and eventually output a high level ciphertext.

F. Conversion methods in E^3

Conversion between modular and bit-level arithmetic. An alternative method to support both arithmetic and logic operations on FHE is through modular and bit-level conversion. To evaluate non-polynomial functions using arithmetic FHE, E^3 [54] introduced a bridging algorithm based on Fermat's little theorem. Roughly speaking, the method extracts individual bits from an encrypted plaintext in $\mathbb{Z}_p$ for $p > 2$. This enables the conversion of non-polynomial operations into Boolean circuits, allowing them to be operated on bit-value ciphertexts. However, due to its high computational cost, this bridging algorithm can be impractical in real-world applications. For instance, when using a plaintext modulus $p = 2^4+1$, E^3 takes about 20 minutes to convert the plaintext encrypted in $\mathbb{Z}_p$ to the equivalent bit-level ciphertexts. In contrast, using the arithmetic-to-logic FHE conversion paradigm adopted by HEIR, the overall conversion latency from arithmetic FHE ciphertexts to logic FHE is under 1 ms (essentially a single application of SAMPLEEXTRACT).