Check for updates

ACM DIGITAL LIBRARY | Association for Computing Machinery | acm open

Latest updates: https://dl.acm.org/doi/10.1145/3696443.3708924

RESEARCH-ARTICLE

ANT-ACE: An FHE Compiler Framework for Automating Neural Network Inference

LONG LI, Ant group, Hangzhou, Zhejiang, China

JIANXIN LAI, Ant group, Hangzhou, Zhejiang, China

PENG YUAN, Ant group, Hangzhou, Zhejiang, China

TIANXIANG SUI, Ant group, Hangzhou, Zhejiang, China

YAN LIU, Ant group, Hangzhou, Zhejiang, China

QING ZHU, Ant group, Hangzhou, Zhejiang, China

View all

Open Access Support provided by:

UNSW Sydney

Ant group

Tsinghua University

PDF Download
3696443.3708924.pdf
27 December 2025
Total Citations: 2
Total Downloads: 1257

Published: 01 March 2025

Citation in BibTeX format

CGO '25: 23rd ACM/IEEE International Symposium on Code Generation and Optimization
March 1 - 5, 2025
NV, Las Vegas, USA

Conference Sponsors:
SIGPLAN
SIGMICRO

ANT-ACE: An FHE Compiler Framework for Automating Neural Network Inference

Long Li
ll398708@antgroup.com
Ant Group
Shanghai, China

Jianxin Lai
laijianxin.ljx@antgroup.com
Ant Group
Shanghai, China

Peng Yuan
yp398707@antgroup.com
Ant Group
Beijing, China

Tianxiang Sui
suitianxiang.stx@antgroup.com
Ant Group
Shanghai, China

Yan Liu
ly409648@antgroup.com
Ant Group
Shanghai, China

Qing Zhu
zq398709@antgroup.com
Ant Group
Shanghai, China

Xiaojing Zhang
zxj398711@antgroup.com
Ant Group
Shanghai, China

Linjie Xiao
xiaolinjie.xlj@antgroup.com
Ant Group
Shenzhen, China

Wenguang Chen
yuanben.cwg@antgroup.com
Tsinghua University, Ant Group
Beijing, China

Jingling Xue
j.xue@unsw.edu.au
UNSW, Ant Group
Sydney, Australia

Abstract

Fully Homomorphic Encryption (FHE) facilitates computations on encrypted data without requiring access to the decryption key, offering substantial privacy benefits for deploying neural network applications in sensitive sectors such as healthcare and finance. Nonetheless, programming these applications within the FHE framework is complex and demands extensive cryptographic expertise to guarantee correctness, performance, and security.

In this paper, we present ANT-ACE, a production-quality, open-source FHE compiler designed to automate neural network inference on encrypted data. ANT-ACE accepts ONNX models and generates C/C++ programs, leveraging its custom open-source FHE library. We explore the design challenges encountered in the development of ANT-ACE, which is engineered to support a variety of input formats and architectures across diverse FHE schemes through a novel Intermediate Representation (IR) that facilitates multiple levels of abstraction. Comprising 44,000 lines of C/C++ code, ANT-ACE efficiently translates ONNX models into C/C++ programs for encrypted inference on CPUs, specifically utilizing the RNS-CKKS scheme. Preliminary evaluations on a single CPU indicate that ANT-ACE achieves significant speed enhancements in ResNet models, surpassing expert manual implementations and fulfilling our design goals.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Security and privacy** → **Software and application security**; • **Computing methodologies** → *Artificial intelligence.*

Keywords: FHE, Compilers, Neural Network Inference

ACM Reference Format:
Long Li, Jianxin Lai, Peng Yuan, Tianxiang Sui, Yan Liu, Qing Zhu, Xiaojing Zhang, Linjie Xiao, Wenguang Chen, and Jingling Xue. 2025. ANT-ACE: An FHE Compiler Framework for Automating Neural Network Inference. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3696443.3708924

1 Introduction

Fully Homomorphic Encryption (FHE) [24] enables computations on encrypted data, allowing privacy-preserving applications to be outsourced to untrusted cloud providers without compromising data confidentiality. This allows organizations to leverage external computational resources while maintaining control over their information security, eliminating the need to trust cloud software or hardware vendors, which is critical in sectors like healthcare and finance. For example, FHE has been used in genotype imputation [29], ensuring that both the data and results remain secure and unreadable to unauthorized parties.

This work is licensed under a Creative Commons Attribution 4.0 International License.
CGO '25, March 01–05, 2025, Las Vegas, NV, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708924

Table 1. Comparing capabilities of FHE compilers for supporting NN inference.

Compiler	Auto NN Operator Handling		Auto Parameter Selection	Bootstrapping	Fixed-Point Support	Not-DSL-Based	Open-Source
	Linear	Nonlinear					
E3 [15]	✗		✗	✗	✗	✗	✓
nGraph-HE [9]	✓	✗	✗	✗	✓	✓	✓
CHET [19]	✗		✓	✗	✓	✗	✗
EVA [18]	✗		✓	✗	✓	✗	✓
Transpiler [28]	✗		✓	✓	✗	✓	✓
HECO [50]	✗		✗	✗	✓	✗	✓
Fhelipe [33]	✓		✓	✓	✓	✗	✓
ACE (This Paper)	✓		✓	✓	✓	✓	✓

Recent advances in FHE theory and library improvements have made FHE more practical. Initial schemes like TFHE [16], which supports Boolean operations, and BGV [12], BFV [20], and GSW [25], which support integer operations, laid the groundwork. More recent developments, such as CKKS [14] and its variant RNS-CKKS [13], support fixed-point arithmetic by scaling floating-point values to integers, making them particularly well-suited for machine learning tasks.

Efficient libraries like SEAL [47] and OpenFHE [6] provide comprehensive APIs that simplify FHE operations, increasing accessibility. However, programming FHE applications, including machine learning models, remains challenging and requires deep cryptographic knowledge. Programmers must select appropriate security parameters and perform scheme-specific optimizations, such as configuring tensor data layouts and improving bootstrapping efficiency [24]. For example, Microsoft's Edge browser uses FHE to secure user passwords [34], requiring developers to be proficient in FHE schemes to ensure correctness, performance, and security. Creating a correct and efficient FHE implementation often necessitates repeated manual tuning.

To address the challenges of FHE programming, we introduce ANT-ACE, a production-quality open-source FHE compiler to fully automate neural network (NN) inference on encrypted data. As highlighted in Table 1, ANT-ACE differentiates itself from existing solutions like E3 [15], nGraph-HE [9], and HECO [50], which lack automatic security parameter selection and bootstrapping. Unlike CHET [19] and EVA [18], which require manual optimization of individual NN operators through a DSL, and nGraph-HE, which automates only linear NN operators, ANT-ACE offers a comprehensive and automated solution. E3 and Transpiler [28] are designed for C/C++ programs, while nGraph-HE, CHET, and HECO specifically target neural network inference. EVA attempts to cater to both general-purpose and machine learning inference tasks but offers only a limited number of APIs. Fhelipe [33] exceeds CHET and EVA in terms of support scope; however, like these compilers, it also requires manual optimization of NN operators through a DSL.

ANT-ACE is a compiler infrastructure that fully automates neural network inference on encrypted data using FHE, providing a robust open-source platform. It transforms ONNX inference models into C/C++ programs with its custom open-source FHE library. Designed with a versatile compiler infrastructure supporting multiple levels of abstraction, ANT-ACE aims to support various input formats and computer architectures across different FHE schemes. Currently, it operates with the RNS-CKKS scheme [13, 14] on CPU platforms. The ANT-ACE codebase comprises 44K lines of source code, 11K lines dedicated to unit and integration tests, and 15K lines of comments to support open-source development. The development of this project required 44 man-months of effort from several compiler experts by the time of this paper's submission. ANT-ACE efficiently translates ONNX models into C/C++ for encrypted data execution on CPUs, leveraging a custom open-source FHE library to expose advanced optimizations with the RNS-CKKS scheme to the ANT-ACE compiler framework. Preliminary performance assessments on single CPUs show that ANT-ACE achieves 2.24× performance speedups in ResNet models, surpassing manual expert implementations and meeting its design goals.

Our work makes the following main contributions:

- **An End-to-End FHE Compiler Framework:** Automatically converts ONNX models into C/C++ programs that perform encrypted inference via FHE.
- **Production-Quality Multi-Level IR Compiler Infrastructure:** Supports detailed analysis and optimization specific to FHE, enhancing the framework's effectiveness, extensibility and portability.
- **Open-Source Implementation:** Facilitates future expansions of ANT-ACE to accommodate additional input formats and extend compatibility to various computing architectures and FHE schemes.

- **Superior Performance:** ANT-ACE demonstrates remarkable efficiency by converting small ResNet models into their FHE equivalents in seconds—a task that would typically take weeks for an FHE expert to complete manually. It also achieves a performance speedup of 2.24× compared to expert-crafted versions [35].

2 Background

We begin with the fundamentals of FHE schemes in Section 2.1, followed by an introduction to CKKS [14] and RNS-CKKS [13] in Section 2.2. Next, we outline the challenges in developing FHE compilers, specifically for CKKS and RNS-CKKS, in Section 2.3. Finally, we present the threat model for our ANT-ACE compiler framework in Section 2.4.

2.1 Fully-Homomorphic Encryption

Consider a plaintext message p. The encryption function *Enc* encrypts p into a *ciphertext* $Enc(p)$, which is decrypted by the decryption function *Dec* such that $Dec(Enc(p)) = p$. All FHE schemes operate on integers within ring structures. For two integers x and y, where + represents the addition operator and × the multiplication operator, an FHE scheme defines $\oplus$ as the homomorphic equivalent of addition and $\otimes$ as the homomorphic equivalent of multiplication. Therefore, the FHE scheme ensures the following equations hold:

$$Dec(Enc(x) \oplus Enc(y)) = x + y$$
$$Dec(Enc(x) \otimes Enc(y)) = x \times y$$

Additionally, an FHE scheme usually supports operations where ciphertexts can be added to or multiplied by plaintexts.

To ensure quantum safety [43], FHE schemes introduce noise during encryption, which must be removed during decryption [22]. This noise increases linearly with homomorphic addition and exponentially with homomorphic multiplication, limiting the maximum multiplicative depth. If the noise exceeds a threshold set by the security parameters, the ciphertext becomes undecryptable. *Bootstrapping* [24] is essential for reducing noise, allowing unlimited homomorphic operations and achieving true FHE. However, bootstrapping is the most computationally intensive FHE operation. Similar to Transpiler [28] (supporting TFHE [16] only) and Fhelipe [33], ANT-ACE aims to fully optimize this crucial noise-reduction process (Table 1).

2.2 CKKS

Currently, CKKS [14] and its RNS-CKKS variant [13] uniquely support fixed-point arithmetic for floating-point numbers. Consequently, they are widely supported by FHE libraries [6, 47] and various FHE compilers [9, 18, 19, 50] for enabling homomorphic computations on encrypted data.

Figure 1 illustrates the CKKS workflow. Initially, a *cleartext* message m, typically a vector of floating-point numbers,

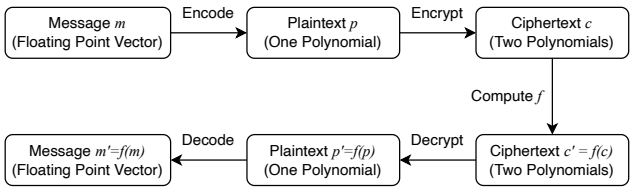

Figure 1. Workflow of CKKS.

is encoded into a *plaintext* polynomial p with integer coefficients under the polynomial modulus $X^N + 1$, where the ring degree N typically ranges from 2^{12} to 2^{15} for moderate computations. This plaintext is then encrypted into a *ciphertext* c, represented as a pair of polynomials with integer coefficients modulo Q, the ciphertext modulus, which generally ranges from 2^{180} to 2^{300}. After performing the computation f homomorphically on c, the resulting ciphertext c' is decrypted back to the plaintext p'. Finally, p' is decoded into the message m', representing the outcome of $f(m)$.

CKKS leverages SIMD parallelism through *batching*, packing $N/2$ floating-point values into a single ciphertext. This enhances efficiency and throughput by allowing simultaneous operations on multiple values within one cryptographic operation, significantly speeding up complex computations like those in machine learning models.

Homomorphic operations like addition or multiplication on ciphertexts correspond to element-wise addition or multiplication of their respective cleartext messages.

The RNS-CKKS variant [13] enhances computational efficiency by decomposing Q into a chain of r smaller co-prime moduli $Q_0, \ldots, Q_{r-1}$. This allows ciphertexts, typically modulo Q, to be expressed as sets of smaller polynomials modulo each Q_i, where Q_i can be represented by a 64-bit integer. Operations on these smaller polynomials can be performed in parallel and recombined using the Chinese Remainder Theorem, reducing computational complexity and memory usage. Given these advantages, ANT-ACE supports RNS-CKKS. For simplicity, we will refer to both CKKS and RNS-CKKS interchangeably unless a distinction is explicitly required.

CKKS uses rescaling to manage the scale of ciphertexts and prevent overflow. For example, a ciphertext c that contains the encoding 0.18 multiplied by the scale 2^{40} will encode 0.0324 at a scale of 2^{80} when squared. Rescaling back to 2^{40} truncates the fixed-point representation but maintains a consistent scale and shifts the modulus down one level in the modulus chain [13, 14]. Operands in addition operations must have the same scale. Although rescaling is less resource-intensive than bootstrapping, minimizing its frequency is beneficial. Modulus switching, on the other hand, reduces the modulus of a ciphertext without altering its scale.

When multiplying two ciphertexts with k_1 and k_2 polynomials, the result contains $k_1 + k_2 + 1$ polynomials. To prevent

unmanageable growth in polynomial count, *relinearization* in CKKS is applied to reduce this number back to 2.

Rotations enable cyclic shifts of cleartext vectors encrypted within a ciphertext by a constant k. This requires special rotation keys, generated during key setup to manipulate encrypted data without decryption. Generating rotation keys for every possible k is impractical due to the large ring degree N used even for moderate computations. Thus, FHE libraries [6, 47] typically create rotation keys for powers of 2 and combine multiple rotations for the desired shift. Rotations are as computationally costly as multiplications [14].

2.3 Compiler Design Challenges

We discuss the development challenges of the ANT-ACE compiler aimed at supporting privacy-preserving computations with FHE, particularly for encrypted inference in machine learning. While these challenges are presented in the context of CKKS, they are broadly applicable to other FHE schemes like TFHE [16], BGV [12], BFV [20], and GSW [25].

• **Security Parameter Selection.** In CKKS, the security parameters Q and N are chosen to meet desired security levels, such as 128-bit security [7, 14]. Q is influenced by the input scale Δ for fixed-point arithmetic, the output scale Q_0 for desired output precision, and the maximum multiplicative depth. The polynomial ring degree N is determined by Q and the required SIMD vector width. Higher Q enables more operations before bootstrapping, and greater N improves parallelism, both raising computational costs. Compiler optimizations further complicate their optimal settings due to their interdependent impact on performance.

• **Key Generation.** In addition to the secret and public keys for encryption and decryption, CKKS [14], like other FHE schemes [16, 20], requires various *evaluation keys* such as relinearization and rotation keys (generated using the secret key) for encrypted computations. These keys can occupy hundreds of gigabytes of memory [35]. Optimizing them can significantly reduce memory usage and generation time.

• **Compiler-Time Analysis and Optimization for Supporting Fast Encrypted Computations.** Homomorphic operations, with complexities such as $O(N \log Nr^2)$ for multiplications and rotations (where r is the number of ciphertext moduli, Section 2.2), are significantly more expensive than plaintext operations. These complexities necessitate a specialized FHE compiler like ANT-ACE, designed to efficiently manage these challenges. ANT-ACE automates security parameter selection (as discussed above), optimizes homomorphic operations, and strategically manages rescaling, relinearization, rotation, and bootstrapping. In particular, ANT-ACE focuses on reducing the maximum multiplicative depth and improving bootstrapping efficiency, thereby enhancing performance.

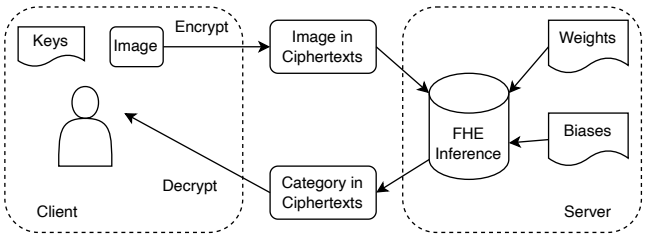

Figure 2. Threat model.

• **Fast Encrypted Inference.** Accelerating encrypted inference for neural network models requires the ANT-ACE compiler to handle all NN operators supported by ONNX and other models, including non-linear functions like `exp`, `log`, and `tanh`. These functions are approximated using polynomials [27, 36], increasing homomorphic multiplications and bootstrapping operations. The precision of these approximations directly impacts multiplicative depth, so ANT-ACE must balance efficiency and accuracy. Additionally, ANT-ACE must optimize data layouts for encrypted tensors, such as user-provided images, as the layout choice affects efficiency through SIMD parallelism and influences security by altering the number of required homomorphic multiplications [18, 19].

• **Extensibility and Portability.** An effective FHE compiler should provide portable performance, adaptable to various input formats like PyTorch and C/C++ and multiple computing architectures within different FHE schemes. Modern optimizations allow encrypted fixed-point multiplications to occur in microseconds, comparable to early 8086 processors' speeds. To enhance practicality in FHE, adoption of domain-specific architectures (DSAs) to expedite encrypted computations is expected. The ANT-ACE compiler framework focuses on maximizing hardware capabilities through high-level FHE library APIs, aiming to streamline encrypted computations.

To address these challenges, a more flexible domain-specific compiler is required. This compiler should focus on tailored IR capabilities to support both general and FHE-specific analyses and optimizations across various FHE schemes and architectures. Additionally, it should facilitate software-hardware co-design for FHE accelerators.

2.4 Threat Model

Figure 2 depicts a scenario where a user benefits from encrypted machine learning inference on an untrusted server. The user encodes and encrypts an image, then sends this encrypted data to the server. The server performs inference on the encrypted image without accessing its content. The encrypted result is sent back to the user, who decrypts and decodes it with their secret key to obtain the final result.

We adopt a semi-honest threat model similar to that used in CryptoNets [27], where both the server and the compiler are considered semi-honest. This means they are expected to perform computations accurately but might be tempted to learn about the user's data. The integrity of the computation is maintained, yet the privacy of the user or client data is crucial and must be rigorously protected under this model.

3 ANT-ACE: Compiler Framework

To tackle these challenges (Section 2.3), we developed the ANT-ACE compiler framework, illustrated in Figure 3, to support encrpyed machine learning inference. The process begins with an NN inference program being parsed by our front-end (Section 3.1). It is then processed through a multi-level IR (Section 3.2) and managed by our custom runtime library, named ACEfhe (Section 3.3). Finally, an equivalent FHE program is output by the code generator (Section 3.4).

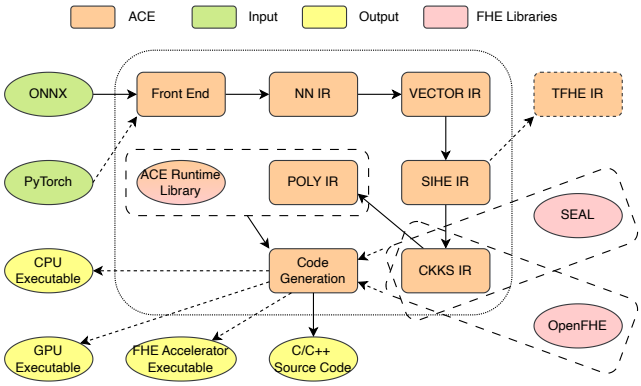

Figure 3. The ANT-ACE compiler framework.

To deploy their machine learning models homomorphically (e.g., on a cloud provider), developers can simply input their existing models into ANT-ACE. This approach bypasses the need for the DSL implementations of NN operators, as required by compilers such as CHET [19], EVA [18], and Fhelipe [33], and eliminates the necessity for re-training even when incorporating non-linear operators mandated by existing FHE compilers (Table 1).

Currently, ANT-ACE supports encrypted inputs (e.g., images) encapsulated within single ciphertexts. To accommodate larger inputs that may require multiple ciphertexts, developers must encrypt inputs using an ANT-ACE-generated encryptor, which encodes the selected parameters, including data layouts. The encrypted data is subsequently decrypted using a corresponding ANT-ACE-generated decryptor. In future work, both tools—which handle all necessary configurations similar to those in CHET [19]—will be integrated to further streamline the deployment process.

3.1 Front End

Currently, ANT-ACE accepts machine learning models defined in the ONNX format, as ONNX encompasses all essential operations required for machine learning models. Additionally, tools are available for converting both TensorFlow [1] and PyTorch [17] models into the ONNX format.

We start by supporting a subset of ONNX operations for experimenting with encrypted inference on small ResNet models, with plans to incrementally add more operations. Additionally, we aim to directly accept PyTorch models, leveraging the latest machine learning advancements and avoiding potential optimization losses from ONNX transformations.

The frontend converts ONNX models into the `NN` IR, the first of multiple abstraction levels designed to support encrypted computations through FHE, as detailed below.

3.2 Multi-Level Intermediate Representation

We began developing our ANT-ACE compiler with a focus on CKKS [13, 14] to support encrypted inference for machine learning tasks on CPUs. Thus, ANT-ACE, as outlined in Table 2, is structured with five abstraction levels: `NN`, `VECTOR`, `SIHE`, `CKKS`, and `POLY`. Most analysis and optimization tasks aim to enhance performance, while `SIHE` IR focuses on code translation and `CKKS` IR on security parameter selection, targeting both performance and automation. In Section 8, we discuss how to extend ANT-ACE to support other FHE schemes on additional platforms.

Table 2. Multiple abstracion levels (for supporting CKKS).

IR	Homomorphc Operations
NN	NN Operator Fusion[1]
VECTOR	Data Layout Selection[1], Batching[1], Matrix Multiplication Optimization[1], Convolution Optimization[1],
SIHE	FHE Computation Recognition[2], Nonlinear Function Approximation[2]
CKKS	Parameter Selection[1,2], Rescaling Placement[1], Multiplication Depth Reduction[1], Bootstrapping Placement[1], Relinearization Placement[1], Rotation Optimization[1], CKKS Operator Fusion[1], Key Generation[1]
POLY	Polynomial Operator Fusion[1], Loop Fusion[1], FHE-Accelerator-Specific Optimizations [1]

[1] Performance-Focused, [2] Translation-Focused

We summarize the functionalities of these IRs below. All traditional compiler optimizations are omitted in Table 2.

• **`NN` (Neural Network) IR.** This IR is the ONNX-equivalent representation in ANT-ACE, where each ONNX operator operating on tensors is mirrored. Operator fusion is supported to enhance performance. While ONNX models might already be optimized through operator fusion, PyTorch programs as input can significantly benefit from this optimization.

• **VECTOR (VECTOR IR).** The tensor data types in the NN IR are converted to vector data types in this IR, enabling operations like GEMM to be performed as vector multiplications. During this phase, data layouts for different tensors are determined, and batching is managed. Critical performance optimizations, especially for matrix multiplication and convolution, are conducted at this stage.

• **SIHE (Scheme-Independent Homomorphic Encryption).** VECTOR IR computations are transformed into operations on ciphertext (type Cipher), plaintext (type Plain), and cleartext (type Vector) within the SIHE IR. Nonlinear functions are identified and approximated using polynomials [36]. By keeping SIHE independent of a particular FHE scheme, ANT-ACE can easily adapt to other schemes like TFHE [16] and BGV [12], accommodating future advancements.

• **CKKS (CKKS IR).** In this IR, SIHE IR operations are tailored for CKKS. ANT-ACE automates security parameter selection and optimizes processes like reducing multiplicative depth, fusing operators, and managing rescaling, relinearization, rotation, and bootstrapping. It also determines optimal key generation at this stage.

• **POLY (POLY IR).** In this IR, CKKS operations from the CKKS IR are decomposed into fundamental polynomial operations. FHE-specific techniques like loop fusion and polynomial operator fusion are employed. Additionally, with future FHE accelerators, optimizations considering both software and hardware aspects will be implemented at this stage.

3.3 Runtime Library

Despite available FHE libraries like SEAL [47] and OpenFHE [6] that support CKKS, we have developed ACEfhe, a custom open-source FHE library, to (1) expose optimization opportunities within RNS-CKKS operations to the ANT-ACE compiler framework; (2) enhance polynomial operations; (3) customize the library for various hardware accelerators, boosting portability and performance; and (4) facilitate software-hardware co-design across multiple IRs.

We have enhanced ACEfhe with optimizations absent in SEAL [47], OpenFHE [6], and FullRNS-HEAAN [13], improving performance for CKKS operations. Specifically, we will detail our advancements in bootstrapping in Section 4.4.

3.4 Code Generation

ANT-ACE's code generator is designed to adapt input programs for encrypted execution across various platforms, including CPUs, GPUs, and dedicated FHE accelerators. It currently converts ONNX models into C/C++ for CPU execution using the POLY IR and our ACEfhe library, specifically optimizing RNS-CKKS operations. To minimize compile time and reduce file size, weights and biases are stored externally. For instance, with ResNet-20, external storage reduces compile time from 30 seconds to about 2 seconds and decreases the C/C++ file size from 621 MB to 384 KB, with the weights and biases taking up an additional 215 MB in an external file. Looking ahead, we plan to introduce options for utilizing the CKKS IR with both SEAL [47] and OpenFHE [6], as illustrated in Figure 3. This versatility will enable ANT-ACE's researchers and users to explore various performance optimization strategies more effectively.

4 Multi-Level Intermediate Representation

We describe the design of our multi-level abstractions using a simplified ONNX model in Figure 4. This model, named `linear_infer`, features a single `gemv` operation—a special case of `gemm` that resembles a Linear Regression model—and is used to demonstrate the transformations at each IR level.

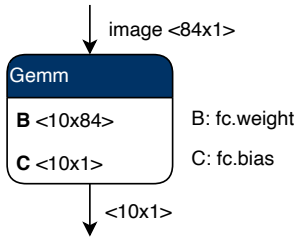

Figure 4. An ONNX model.

In this particular `gemm` example, the implementation details come from ACE's open-source implementation. Our focus, however, is on illustrating how GEMV is successively lowered across the five IRs.

When introducing an instruction at each IR, we omit the explicit return value, assuming it resides in `res`.

4.1 NN IR

The NN IR is a tree node-based representation aligned with the graph structure defined in ONNX. When ANT-ACE loads a neural network model from a file, it constructs this representation by mapping each operator to one or more nodes. The NN IR is strongly typed, with the input and output of each node explicitly designated with tensor types.

Table 3. Operators in the NN IR.

Operator	Types
`average_pool x`	`x, res ∈ Tensor`
`conv x w b`	`x, w, b, res ∈ Tensor`
`flatten x`	`x, res ∈ Tensor`
`gemm a b c`	`a, b, c, res ∈ Tensor`
`global_average_pool x`	`x, res ∈ Tensor`
`relu x`	`x, res ∈ Tensor`
`reshape d s`	`d, s, res ∈ Tensor`
`strided_slice d i l t`	`d, i, l, t, res ∈ Tensor`

Table 3 lists key supported operators that conform to the semantics of their ONNX counterparts [4], with the exception of `strided_slice`. For `strided_slice`, `d` represents the input, `i` the start index, `l` the slice size, and `t` the stride. Additionally, `gemm` includes `gemv` as a special case. For simplicity, all tensor types are uniformly referred to as `Tensor`.

The `linear_infer` model in Figure 4 is translated into a function with the prototype "`Tensor linear_infer(Tensor& image)`" in this IR, containing the following statements:

```
st "output"
  gemm
    ld "image"
    ld "fc.weight"
    ld "fc.bias"
retv
  ld "output"
```

Listing 1. NN IR for `linear_infer` in Figure 4.

Here, `ld` and `st` represent nodes for memory load and store, respectively, and `retv` is a node used for returning a value. We have omitted code handling the weight and bias of the FC layer, as it is not relevant to this discussion.

4.2 VECTOR IR

The VECTOR IR specifies types, transformations, and operations tailored to vectors, facilitating the development of optimization strategies for vector operations. This IR abstracts multi-dimensional tensors into one-dimensional vectors, encapsulating their multi-dimensional characteristics within the Vector type. This approach effectively bridges the semantic gap between tensors and their lower-dimensional operations, with vectors internally implemented as arrays.

This design successfully achieves our primary objective: to enable progressive lowering from the NN IR to the VECTOR IR, and further into lower IRs essential for executing FHE-specific operations and machine-native SIMD operations.

Table 4. Operators in VECTOR IR.

Operator	Types
`add x y`	`x, y, res ∈ Vector`
`broadcast x y`	`x, res ∈ Vector, y ∈ Int`
`mul x y`	`x, y, res ∈ Vector`
`pad x y`	`x, res ∈ Vector, y ∈ Int`
`reshape d s`	`d, s, res ∈ Vector`
`roll x y`	`x, res ∈ Vector, y ∈ Int`
`slice d i s`	`d, res ∈ Vector, i, s ∈ Int`
`tile x y`	`x, res ∈ Vector, y ∈ Int`

Table 4 lists the operators introduced in the VECTOR IR, providing NumPy-like semantics and facilitating the abstraction of higher-level tensor operations into canonical patterns. For simplicity, we use `Vector` to denote any vector, irrespective of its underlying tensor type and dimensions.

Consider `linear_infer` in Figure 4. The VECTOR IR function prototype is "`Vector linear_infer(Vector& image)`". Its corresponding NN IR in Listing 1, described by `output = gemm(image, fc.weights, fc.biases)`, is translated into the following pseudo loop form in VECTOR IR:

```
...
  do_loop
    ...
    block
      st "output"
        VECTOR.add
          ld "output"
          VECTOR.mul
            VECTOR.roll
              ld "image"
              ...
            VECTOR.slice
              ld "fc.weight"
...
```

Listing 2. VECTOR IR for `linear_infer` in Figure 4.

In Listing 2, we have omitted handling `fc.bias` in Listing 1 for simplicity. Each operator in the VECTOR IR starts with "VECTOR" for clarity. The `do_loop` loop handles matrix multiplication, where `Vector.mul` will be translated to homomorphic multiplication during the VECTOR-to-SIHE phase. `VECTOR.roll` shifts parameters along an axis, and `VECTOR.slice` functions like slicing in NumPy. This IR focuses on data-flow analysis and optimization for key operations like matrix multiplication and convolution, determining tensor data layouts for encryption into CKKS ciphertexts. It leverages CKKS's batching vectorization, balancing security and performance considerations like vector widths, N, and multiplicative depth.

4.3 SIHE IR

Table 5. Operators in the SIHE IR.

Operator	Types
`rotate x y`	`x, res ∈ Cipher, y ∈ Int`
`add x y`	`x, res ∈ Cipher, y ∈ Cipher|Plain`
`sub x y`	`x, res ∈ Cipher, y ∈ Cipher|Plain`
`mul x y`	`x, res ∈ Cipher, y ∈ Cipher|Plain`
`neg x`	`x, res ∈ Cipher`
`encode x`	`x ∈ Vector, res ∈ Plain`
`decode x`	`x ∈ Plain , res ∈ Vector`

The SIHE IR is designed to analyze and optimize FHE operations independently of any specific FHE scheme. Being scheme-independent, it supports only the common FHE operations outlined in Table 5. For neural networks, it distinguishes three data types: ciphertext (Cipher) abstracting a sequence (e.g., for representing part of a private image), plaintext (Plain) abstracting a sequence (e.g., for weights and biases), and cleartext (Vector) inherited from the VECTOR IR. The `encode` operator is used by both the ANT-ACE-generated encryptor and the ANT-ACE compiler, whereas the `decode` operator is employed exclusively by the ANT-ACE-generated decryptor. Recall that the encryptor and decryptor are described in Section 2.4

In the SIHE IR, a key task is to automatically detect computations on encrypted data that require homomorphic execution. This is accomplished by using type inference to identify ciphertext variables and replace their operations with homomorphic equivalents via data-flow analysis.

For `linear_infer` in Figure 4, the `do_loop` loop in the VECTOR IR in Listing 2 is now translated into the SIHE IR:

```
st "output"
  SIHE.add
    ld "output"
    SIHE.mul
      SIHE.rotate
        ld "image"
        ...
      SIHE.encode
        VECTOR.slice
          ld "fc.weight"
...
```

Listing 3. SIHE IR for `linear_infer` in Figure 4.

Initially, "image" is a ciphertext by input. Therefore, the result of `VECTOR.roll`, which operates on "image", as shown in Listing 2, is recognized as a ciphertext. Since this result is the first operand in `VECTOR.mul`, this operation is also identified as a ciphertext operation. The second operand of `VECTOR.mul`, `VECTOR.slice`, accesses constant (cleartext) data ("`fc.weight`"). Consequently, `SIHE.encode` is employed to convert the result of `VECTOR.slice` into plaintext. Moreover, `VECTOR.add`, which also operates on ciphertext, does not require `SIHE.encode` as its `output` operand remains a ciphertext, necessitating encrypted execution.

In real-world models, code blocks like those in Figure 4 are typically linked by ReLU operators, which are approximated by polynomials in the SIHE IR, as noted in [27]. Specifically, we use the approximation algorithm from [36] to approximate these nonlinear functions automatically in ANT-ACE.

4.4 CKKS IR

The CKKS IR, detailed in Table 6, includes six new operators and retains all operators from the SIHE IR in Table 5, with some semantic differences. First, `Cipher` and `Plain` now represent a pair of polynomials and one polynomial, respectively. For example, `Cipher-Plain` multiplication (`mul`) produces a `Cipher` containing two polynomials. Second, `Cipher-Cipher` multiplication (`mul`) results in a `Cipher3`, representing three polynomials. A relinearisation operation (`relin`) can convert a `Cipher3` back to a `Cipher` [14].

When transitioning from SIHE to CKKS, all arithmetic operations are mapped to their CKKS equivalents, maintaining the same types except for the `Cipher3` adjustment.

In CKKS [14], operands for homomorphic operations must conform to particular constraints on scales (fixed-point arithmetic) and levels (available multiplicative depths). For ciphertext-ciphertext additions and subtractions, operands must have identical scales and levels, and the operations preserve these attributes. For multiplications, operands need only match levels, with the result's scale being the product of the operand scales, resembling fixed-point multiplication.

Table 6. Operators in the CKKS IR.

Operator	Types
rotate, add, sub, neg, encode, and decode in Table 5	
mul x y	x, res ∈ Cipher, y ∈ Plain
mul x y	x, y ∈ Cipher, res ∈ Cipher3
modswitch x	x, res ∈ Cipher
upscale x y	x, res ∈ Cipher, y ∈ Int
rescale x	x, res ∈ Cipher
downscale x	x, res ∈ Cipher
bootstrap x	x, res ∈ Cipher
relin x	x ∈ Cipher3, res ∈ Cipher

Ciphertext polynomial coefficients in CKKS are represented modulo Q. Exceeding Q causes an overflow, corrupting the underlying message. To prevent this, values must be periodically rescaled, as discussed in Section 2.2. ANT-ACE automatically includes operations like `rescale`, `downscale`, or `upscale` in the CKKS program, each reducing the ciphertext scale by one factor or adjusting it to a specified scale. Additionally, CKKS employs modulus switching to reduce the ciphertext level without altering the message scale.

Effective scale management through `rescale` and `downscale` reduces ciphertext levels, enhancing efficiency in subsequent operations. ANT-ACE adopts a similar approach to EVA [18], focusing on post-multiplication scale adjustments and strategically delaying `rescale` and `downscale` within loops to balance scale management, computational efficiency, and compile-time considerations.

For our motivating example in Figure 4, the code from the SIHE IR (Listing 3) is translated into the CKKS IR as follows:

```
...
st "_image_tmp"
  CKKS.rotate
    ld "image"
    ...
st "output"
  CKKS.add
    ld "output"
    CKKS.mul
      ld "_image_tmp"
      CKKS.encode
        VECTOR.slice
          ld "fc.weight"
          ...
        intconst #0x5a
        CKKS.scale
          ld "_image_tmp"
        CKKS.level
          ld "_image_tmp"
...
```

Listing 4. CKKS IR for `linear_infer` in Figure 4.

SIHE.encode is lowered to CKKS.encode, matching the scale and level of CKKS.rotate in CKKS.mul. To prevent redundant CKKS.rotate computations, it is pre-computed and hoisted , allowing reuse of _image_tmp in CKKS.mul, reducing the need for three CKKS.rotate operations to one. All SIHE operators have been replaced with their CKKS equivalents, and intconst serves to return a constant directly.

Given a ciphertext, bootstrapping latency is influenced by the refreshed ciphertext's level. Higher levels increase the cost of both the bootstrapping and subsequent homomorphic operations [13, 31]. Ideally, the level should match the multiplication depth required until the next bootstrapping or program exit. In line with [35], we position bootstrapping operations before ReLU operations, but only bootstrap a ciphertext to the minimal levels needed based on the number of homomorphic multiplications before the next bootstrapping point. Recently, we have developed a new method to optimize both rescaling and bootstrapping operations through a region-based approach [38], which will soon be integrated into the ANT-ACE compiler framework.

Ciphertext operations need evaluation keys for tasks like relinearization and rotation, which are memory-intensive, consuming over 1 GB each [35]. It is advisable to generate only the necessary keys. For example, ANT-ACE can identify the required rotation keys by analyzing all ciphertext rotations during the SIHE-to-CKKS lowering phase, using Vector.roll operators from the VECTOR IR.

Finally, ANT-ACE automatically selects the security parameters Q and N to meet the required security level, such as 128-bit security. It calculates Q using the scaling factor Δ and the maximum multiplicative depth, as outlined in Section 2.3 [7, 10]. ANT-ACE then establishes the upper bound for N_1, the polynomial modulus for CKKS ($X^{N_1}+1$), to align with the necessary security level [6, 7, 10, 47]. Subsequently, it assesses N_2, the maximum SIMD width required by the VECTOR IR, and sets $N = \max(N_1, N_2)$. While selecting an N larger than N_1 does not compromise security, it can lead to decreased efficiency [19]. To mitigate this, ANT-ACE can incorporate data layout algorithms to enhance performance. Thus, ANT-ACE serves as a robust platform that effectively balances correctness, performance, and security.

4.5 POLY IR

All homomorphic operations in the CKKS IR are further lowered to the POLY IR, preserving their data types (Cipher, Cipher3, Plain and Vector). In the RNS-CKKS model [13], each polynomial is divided into r smaller *RNS polynomials*, each modulo Q_i, where $Q = \prod_{i=0}^{r-1} Q_i$ (Section 2.2).

Table 7 presents key operators from 40 available, categorized into two groups. High-level operations like decomp, mod_up, and mod_down focus on overall polynomial granularity and do not distinguish individual RNS components [13, 31], enabling coarse-grained operator fusion. In contrast,

Table 7. Operators in the POLY IR.

Operator	Types
coeff x y	x ∈ Poly, y ∈ Int, res ∈ Int*
set_coeffs x y z	x ∈ Poly, y ∈ Int, z ∈ Int*
num_q x	x ∈ Poly, res ∈ Int
num_p x	x ∈ Poly, res ∈ Int
num_decomp x	x ∈ Poly, res ∈ Int
decomp x y	x,res ∈ Poly, y ∈ Int
mod_up x y	x,res ∈ Poly, y ∈ Int
mod_down x	x,res ∈ Poly
rescale x	x,res ∈ Poly
hw_ntt x	x,res ∈ Int*
hw_intt x	x,res ∈ Int*
hw_modadd x y z	x,y,res ∈ Int*, z ∈ Modulus
hw_modmul x y z	x,y,res ∈ Int*, z ∈ Modulus
hw_rotate x y z	x,y,res ∈ Int*, z ∈ Modulus

operations prefixed with "hw_" specifically address RNS polynomial granularity. Polynomial functions such as add, mul, and rotate from the CKKS IR are adapted into *RNS* loops in the POLY IR, cycling through moduli, with "hw_" operations incorporated within these loops, as illustrated by a ciphertext-ciphertext addition example given below:

```
// ciph3 = ciph1 + ciph2
for (int i = 0; i < num_q(ciph1.c0); i++)
  ciph3.c0[i] = hw_modadd(ciph1.c0[i], ciph2.c0[i]);
for (int i = 0; i < num_q(ciph1.c1); i++)
  ciph3.c1[i] = hw_modadd(ciph1.c1[i], ciph2.c1[i]);
```

Here, num_q returns r, i.e., the count of remaining RNS polynomials. The "hw_" prefix imlies these operators are optimized for conversion into specialized hardware instructions.

Optimizations in the POLY IR leverage traditional techniques specifically tailored for polynomial operations that constitute CKKS functions. Operator fusion, aimed at minimizing memory duplication and movement across library calls, is implemented through optimized APIs in our ACEfhe library. These include decomp_modup (fusing decomp and modup) and hw_modmuladd (combining hw_modmul and hw_modadd). Our ANT-ACE compiler utilizes data-flow analysis to identify and apply these fused operators effectively.

RNS-loop fusion is beneficial in RNS-CKKS since RNS loops have compile-time constant trip counts. Loops with identical trip counts can be fused, reducing redundant memory operations and enhancing data locality.

Consider the following code snippet:

```
for (int i = 0; i < num_q(poly1); i++)
  poly3[i] = hw_modadd(poly1[i], poly2[i]);
for (int i = 0; i < num_q(poly3); i++)
  poly5[i] = hw_modmul(poly3[i], poly4[i]);
```

With $N = 2^{16}$ and a remaining multiplicative depth of 20 for five ciphertexts (poly1 to poly5), where num_q(poly1) = num_q(poly3) = 20, each polynomial necessitates 10MB of

Table 9. Detailed comparison of ANT-ACE with other FHE compilers from Table 1 on the application of compiler technology in automating encrypted inference.

	FHE Scheme	Infrastructure	Frontend	Backend	IR	Optimizations
E3 [15]	BFV, BGV, TFHE	Synopsys Compiler	C++	SEAL, TFHE	Circuit	Circuit
nGraph-HE [9]	BFV, CKKS	nGraph Compiler	TensorFLow	SEAL	nGraph IR	SIMD Packing, Operator Fusion
CHET [19]	CKKS	In-House DAG	Tensor Circuit DSL	SEAL, HEAAN	Homo Tensor Circuit, Homo ISA	FHE Vectorization, Data Layout
EVA [18]	CKKS	In-House DAG	Python DSL	SEAL	Abstract Semantic Graph	Rescale, ModSwtich
Transpiler [28]	TFHE	XLS	C++	TFHE	XLS IR	Circuit
HECO [50]	BFV, BGV, CKKS	MLIR	Python DSL	SEAL	HIR, SIR, PIR, RIR	Batching
Fhelipe [33]	CKKS	In-House DAG	Python DSL	Lattigo	DFG of tensor operators, DAG of CKKS operators	Data Layout, Rescale, Bootstrap
ANT-ACE	CKKS	In-House IR	ONNX	Custom Library	NN, Vector, SIHE, CKKS, POLY	All the Operations in Table 2

storage. Following the loop fusion technique described in [42], `poly3` is substituted by a temporary array, `tmp`, which requires only 512KB for storage.

```
for (int i = 0; i < num_q(poly1); i++) {
  int64_t *tmp = hw_modadd(poly1[i], poly2[i]);
  poly5[i] = hw_modmul(tmp, poly4[i]);
}
```

In ANT-ACE, a simple model such as the one depicted in Figure 4 expands to 331 lines of code in the `POLY` IR, reducing to 68 lines when translated into C. This code is automatically generated through a series of analyses, transformations, and optimizations across our five IRs. Manually performing these tasks would be error-prone and labor-intensive, even for users with deep knowledge and expertise in FHE.

5 Implementation

We are developing ANT-ACE as an FHE compiler that automates the transformation of general-purpose programs into formats capable of performing computations on encrypted data, suitable for various FHE schemes and architectures. As of this paper's submission, the project has involved multiple compiler specialists, dedicating 44 man-months of effort. It comprises over 44K+ lines of C++ code, including more than 11K+ lines dedicated to testing and over 15K+ lines of comments to ensure clarity, as detailed in Table 8.

Currently, ANT-ACE is capable of automatically translating ONNX models to CKKS [13, 14] on CPUs. To aid in debugging, ANT-ACE includes instrumentation capabilities at both the `NN` and `VECTOR` IR levels, enabling support for machine learning inference in both unencrypted and encrypted modes.

In Table 9, we enhance the comparison of ANT-ACE with the FHE compilers from Table 1, focusing on their application of compiler technology to support encrypted computations. ANT-ACE excels as the most advanced compiler for automating encrypted inference, enabled by multiple abstraction levels.

Table 8. ANT-ACE's component breakdown by LOC.

ANT-ACE	**LOC**	**Tests**	**Comments**
Infrastructure	17279	3544	4628
NN IR	1902	186	406
VECTOR IR	3214	405	654
SIHE IR	4087	190	803
CKKS IR	1255	369	354
POLY IR	2975	1163	547
Run-Rime Library (ACEFHE)	14101	5551	7860
Total	44813	11390	15252

6 Evaluation

Despite advances [14, 26, 40], FHE remains up to $10,000\times$ slower for small machine learning models. ANT-ACE aims to push forward compiler research to address this challenge.

ANT-ACE currently supports a limited range of NN operators for smaller models, but it uniquely automates the translation of ONNX models for encrypted CPU execution, including both linear and non-linear operators like ReLU. This automation distinguishes ANT-ACE from other FHE compilers listed in Tables 1 and 9, which do not fully automate the encryption process, complicating direct comparisons. Consequently, we evaluate ANT-ACE using small ResNet models, given the significant demands of FHE (e.g., several hours per encrypted inference for `ResNet-110`), and compare it against expert hand-tuned implementations. Although the focus on smaller models is typical for FHE compilers, ANT-ACE's distinctive automation capability effectively achieves our design objectives and addresses four research questions.

- **RQ1:** Can ANT-ACE compile small ResNet models for encrypted execution both automatically and efficiently?
- **RQ2:** How does ANT-ACE compare to expert hand-tuned implementations in executing encrypted NN inference?
- **RQ3:** Is ANT-ACE capable of selecting security parameters that effectively balance correctness, performance, and security?

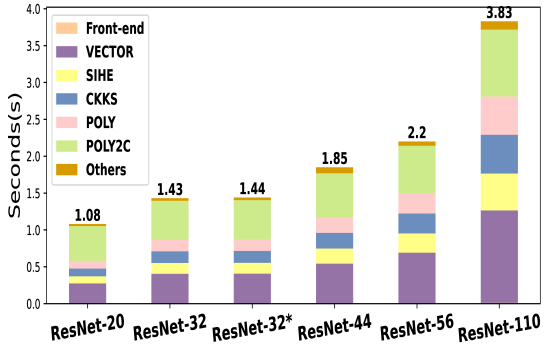

Figure 5. ANT-ACE's compile times.

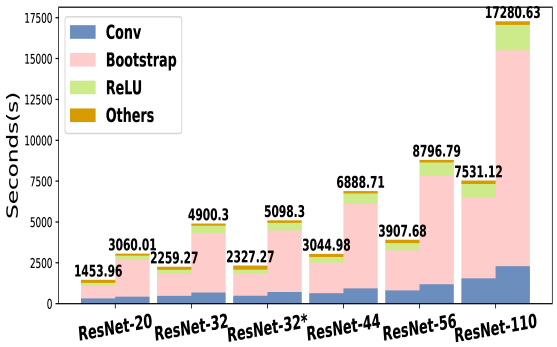

Figure 6. Per-image inference time comparison between ANT-ACE (left bar) and "Expert" (right bar) for each model.

• **RQ4:** Does ANT-ACE maintain the accuracy of NN models during encrypted inference?

• **Experimental Setup.** All experiments were in a Docker container (version 25.0.1) on a Linux server with an Intel Xeon Platinum 8369B CPU @2.70GHz and 512 GB memory.

We evaluated six ResNet models using expert hand-tuned C++ implementations and the SEAL library [35, 47], which includes `ResNet-20, 32, 44, 56,` and `110` on `CIFAR-10`, as well as `ResNet-32` on CIFAR-100 (`ResNet-32*`). Recognized for its performance, SEAL is highly regarded among FHE libraries [53]. All models, except for `ResNet-110` which we trained due to its unavailability, were pre-trained. ANT-ACE compiles these ONNX models to C using GCC (version 10.2.1) with our custom FHE library, ACEfhe, setting input and output scales at $\Delta = 2^{59}$ and $Q_0 = 2^{60}$ respectively. The ring degree N is automatically selected, as will be discussed later. We opted for ACEfhe over SEAL, because SEAL does not support bootstrapping, whereas ACEfhe does. Unlike manual tuning in expert implementations, ANT-ACE automatically integrates bootstrapping calls into ACEfhe.

• **RQ1: Compile Times.** Figure 5 shows the compile times ANT-ACE requires to translate six models into C programs, with percentage time breakdowns for each model. ANT-ACE spent more time on the `VECTOR` IR than the other IRs, transforming cleartext data to vector form. In "Others", most of the time was spent writing model weights to external files.

GCC compiles these encrypted C versions in just a few seconds, faster than ANT-ACE's translation to C. In contrast, experts may need weeks to optimize small networks manually [19]. ANT-ACE significantly enhances developer productivity by automatically translating machine learning models for encrypted execution on untrusted cloud providers.

• **RQ2: Performance and Memory Advantages.** Figure 6 shows that ANT-ACE (left bars) consistently outperforms "Expert" implementations (right bars) across all six models, providing detailed comparisons of per-image inference times for `Conv` (convolution), `Bootstrap`, and `ReLU` in single-thread mode. ANT-ACE significantly reduces the time spent in `Conv` by 31.5%, in `Bootstrap` by 63.3%, and in `ReLU` by 44.6%, resulting in an average performance speedup over the "Expert" implementations.

For context, unencrypted inference with ONNX Runtime 1.18.1 [5] in the same setup ranges from 1.60 ms for `ResNet-20` to 9.73 ms for `ResNet-110` per image in single-thread mode.

We describe how ANT-ACE boosts `Conv`, `Bootstrap`, and `ReLU` performance through sophisticated analyses and optimizations, emphasizing its role in integrating these enhancements effectively within our compiler framework:

– **`Conv`.** ANT-ACE boosts the performance of `Conv` across all five IRs—`NN`, `VECTOR`, `SIHE`, `CKKS`, and `POLY` —by optimizing data layout selection in the `VECTOR` IR and refining it through the subsequent lower-level IRs. In the `CKKS` IR, placement of rescaling and rotation keys is optimized, and in the `POLY` IR, polynomial operations are further enhanced. These sophisticated analyses and optimizations lead to the notable performance improvements ANT-ACE achieves over the "Expert" implementations.

– **`Bootstrap`.** ACEfhe enhances bootstrapping in the `CKKS` IR by strategically placing operations and setting levels for refreshed ciphertexts according to the program's multiplicative depth constraints. This optimization enables ANT-ACE to surpass expert implementations through faster bootstrapping. We have also developed a new method to simultaneously optimize rescaling and bootstrapping [38]. Despite these advancements, bootstrapping continues to be a major bottleneck, highlighting the persistent need for efficiency improvements in cryptographic and compiler technologies.

– **`ReLU`.** ANT-ACE facilitates polynomial approximations of nonlinear functions like ReLU in the `SIHE` IR, allowing precise monitoring of homomorphic multiplication depths consumed in the `CKKS` IR. This capability not only positions ANT-ACE for future integration of advanced approximation algorithms but also enhances the accuracy of maximum multiplicative depth obtained. Such detailed information aids in the optimal placement of bootstrapping in the `CKKS` IR, improving computational efficiency. To handle $ReLU(x) = 0.5 * x * (1 + \texttt{sign}(x))$ in ResNet models, we

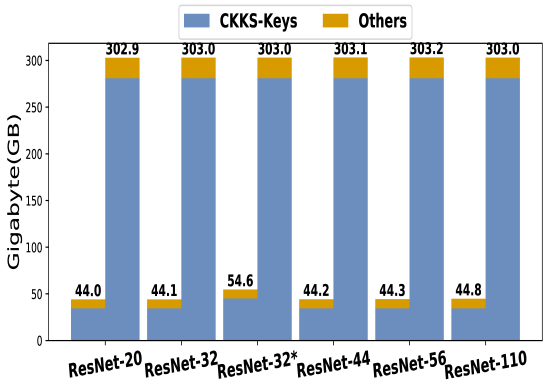

Figure 7. Memory usage comparison between ANT-ACE (left bar) and "Expert" (right bar) for each model.

Table 10. Security parameters selected automatically for CKKS by the ANT-ACE compiler.

Model	$\log_2(N)$	$\log_2(Q_0)$	$\log_2(\Delta)$
ResNet-20	16	60	56
ResNet-32	16	60	56
ResNet-32*	16	60	56
ResNet-44	16	60	56
ResNet-56	16	60	56
ResNet-110	16	60	56

approximate $\texttt{sign}(x)$ using a composition of polynomials with odd-degree terms [36]. The comprehensive workflow provided by ANT-ACE for nonlinear function approximation significantly enhances its performance, enabling it to surpass the "Expert" implementations.

Figure 7 shows further that ANT-ACE surpasses the "Expert" implementations in memory efficiency, highlighting memory usage percentages for key optimization (`CKKS-Keys`). On average, ANT-ACE reduces memory consumption by 84.8% by generating only the necessary keys in the `CKKS` IR through data-flow analysis, after completing all optimizations from the `NN` IR to the `CKKS` IR.

However, CKKS evaluation keys still represent a significant portion of total memory. For example, in `ResNet-20`, ANT-ACE allocates 34.5 GB for keys, with evaluation keys alone accounting for 34.3 GB. This highlights the need for more efficient management of evaluation keys and their associated homomorphic operations, such as relinearization and rotation, through advanced memory-centric optimizations and potentially software-hardware co-design.

• **RQ3: Automatic Security Parameter Selection.** Table 10 lists the security parameters N and Q chosen by ANT-ACE, based on the input scale $\Delta = 2^{56}$ and the output scale $\Delta = 2^{60}$ given for these models, to ensure 128-bit security.

• **RQ4: Inference Accuracy.** Table 11 shows that ANT-ACE nearly maintains the accuracy of all six ResNet models during encrypted inference, aligning with their original pre-trained performance levels. Despite the substantial computational demands of FHE, which limited testing to 1,000 images, ANT-ACE only exhibits a minor average accuracy decrease of 0.43%. This reduction is due to the inherent precision limitations of the CKKS scheme [13, 14] and the approximation of nonlinear functions like ReLU [36]. For comparison purposes, corresponding expert hand-tuned implementations exhibit a slight accuracy drop, ranging from 0.1% to 0.6%.

Table 11. Comparing inference accuracy between unencrypted and ANT-ACE-encrypted ResNet models.

Model	Unencrypted	Encrypted	Accuracy Loss
ResNet-20	90.6%	91.2%	-0.7%
ResNet-32	92.8%	92.8%	0.0%
ResNet-32*	66.4%	65.9%	0.8%
ResNet-44	92.5%	90.7%	1.9%
ResNet-56	93.9%	93.8%	0.1%
ResNet-110	93.9%	93.4%	0.5%

7 Relate Work

Gentry's initial FHE scheme, based on ideal lattices [22], was impractical, taking 31 minutes for a bootstrapping operation [23]. The BGV [12] and BFV [20] schemes, based on the LWE [44] and ring-LWE (RLWE) [41] problems, supported a fixed number of "leveled" homomorphic operations without bootstrapping by using large parameters. They introduced SIMD-style "batching" to pack multiple messages into a single ciphertext, enhancing throughput. The GSW [25] scheme and its successor, TFHE [16], also based on LWE, drastically reduced bootstrapping times to milliseconds but lacked batching, creating a latency-throughput trade-off. The CKKS scheme [13, 14], supporting efficient fixed-point arithmetic, has broadened FHE applications in machine learning.

Advancements in FHE schemes have led to the development of libraries that streamline operations via APIs. HElib [30], the first such library, initially supported only BGV. SEAL [47], supporting BGV, BFV, and CKKS without bootstrapping, is now widely used in FHE research. Recently, OpenFHE [6], which includes support for BGV, BFV, CKKS, and TFHE with bootstrapping, has gained popularity among researchers.

Efforts to minimize FHE expertise in development have led to the exploration of programming language extensions and compiler frameworks. E3 [15] offers C++ extensions for encrypted data handling but requires manual configuration of FHE libraries, schemes, and parameters. CHET [19] uses a DSL to convert tensor operations into FHE operations with basic optimizations. EVA [18] extends and improves upon CHET by embedding a DSL into Python for Neural Network (NN) inference using CKKS in the SEAL library [47], designed for general-purpose use. nGraph-HE [9] enables TensorFlow computations to be converted into BFV or CKKS using SEAL

with minimal code changes but also requires user-defined FHE parameters. Transpiler [28] specifically translates C++ into TFHE, while HECO [50] provides a Python DSL for general-purpose programs targeting the SEAL library. Fhelipe [33] provides a numpy-style programming interface, specializing in automatic data packing and bootstrap placement. It matches the performance of the "Expert" implementations by Lee et al. [35] in `ResNet-20` per-image inference, utilizing the Lattigo FHE library [3]. Meanwhile, HEIR [8] focuses on its cross-scheme compilation capabilities but has not yet demonstrated its effectiveness in NN inference.

Compared to other expert-written and highly-tuned NN inference programs in FHE [9, 11, 18, 27, 39], Lee et al. [35] offer a relatively fair comparison point for ANT-ACE in terms of using the CKKS scheme, particularly with bootstrapping enabled. They developed and optimized a set of ResNet models with significant latencies, overcoming limitations in the SEAL library's [47] bootstrapping capabilities.

8 Future Work

The ANT-ACE compiler framework represents the initial phase of our research into FHE compiler technology. We have developed key functionalities for an FHE compiler aimed at privacy-preserving machine learning inference, showcasing the automation of ONNX model inference with CKKS-encrypted data on CPUs across multiple abstraction levels. Future developments of ANT-ACE will expand support to various input formats and FHE schemes on diverse computing architectures, including GPUs. These enhancements will be further strengthened by contributions from the open-source community, as outlined below.

• **Enhancing Large Language Models (LLMs).** In response to the growing interest in LLMs and the leading role of PyTorch in LLM research, we aim to upgrade the ANT-ACE compiler front-end to support direct compilation of PyTorch models via the TORCH.FX [2] toolkit. This enhancement will involve integrating additional NN operators currently unsupported by ANT-ACE, as listed in Table 3, to fully leverage this capability.

• **Integrating Diverse FHE Schemes.** While CKKS [13, 14] is preferred for machine learning applications, alternative schemes like TFHE [16] have gained attention for features like lossless comparison and fast bootstrapping. We aim to integrate these schemes into the ANT-ACE framework through the SIHE IR and plan to introduce a TFHE IR similar in scale to the CKKS IR, which comprises approximately 1300 lines of code (Table 8). In the POLY IR, existing polynomial operators like `POLY.add` and `POLY.mul` will be utilized, with the addition of new operators such as `POLY.hw_fft` and `POLY.hw_ifft` to accommodate TFHE's requirements.

• **Exploring Parallelism on GPUs.** We are considering several strategies to harness GPU capabilities for enhancing FHE performance. One approach is to generate CUDA code from the `CKKS` IR using a CUDA-based FHE library, which would support coarse-grained homomorphic operations such as modulus switching and bootstrapping. Alternatively, we could translate the `POLY` IR directly into Triton [48]. Additionally, leveraging the advanced parallelization features of GPUs could significantly enhance ciphertext-level parallelism [52] and accelerate FHE operations [21, 51].

• **Embracing Hardware Acceleration.** Hardware acceleration is crucial for transitioning FHE from theory to practice [45]. Efforts to accelerate multiplications [46], key reuse [32], and bootstrapping [49] exist. We plan to integrate an FHE accelerator into our ACE framework via the `POLY` IR.

• **Developing Aggressive Optimizations for speed.** We will continuously explore optimization opportunities across all IR abstractions, targeting both general and FHE-specific enhancements. For example, optimizing data layouts for encrypted data like tensors and enhancing bootstrapping efficiency are vital for improving performance.

• **Innovating Memory Management.** FHE computing is known for its high memory demands but predictable access patterns. We aim to exploit these patterns for targeted memory optimization through the ANT-ACE compiler framework and its runtime, enhancing efficiency and scalability with fine-grained memory management strategies.

9 Conclusion

In this paper, we introduce ANT-ACE, an open-source compiler framework designed to facilitate privacy-preserving computations using Fully Homomorphic Encryption (FHE). Currently, ANT-ACE is capable of automatically converting ONNX models for efficient homomorphic execution on CPUs. Preliminary evaluations using small machine learning models have verified that ANT-ACE achieves its design objectives. Moving forward, we plan to enhance ANT-ACE to overcome existing barriers and make FHE more practical, with contributions from the open-source community. Future efforts will focus on broadening support for various FHE schemes and incorporating hardware acceleration to enhance performance. Through continuous improvements to ANT-ACE, our goal is to make privacy-preserving machine learning more accessible and efficient.

10 Data-Availability Statement

The ANT-ACE compiler framework is now open-source and available at https://github.com/ant-research/ace-compiler. The corresponding artifact is also publicly available at [37].

Acknowledgments

We would like to thank the anonymous reviewers for their valuable suggestions. This work was supported by National Key R&D Program of China (Grant No. 2023YFB4503204).

A Artifact

A.1 Abstract

ANT-ACE is available as open-source software at https://github.com/ant-research/ace-compiler.

A.2 Artifact check-list (meta-information)

- **Binary:** Source code and scripts included to regenerate binaries.
- **Model:** ResNet-[20|32|44|56|110].
- **Data set:** CIFAR-[10|100].
- **Run-time environment:** Docker container version 25.0.1, dependencies detailed in https://github.com/ace-compiler/ace-compiler/blob/main/Dockerfile.
- **Output:** Figure5.pdf, Figure6.pdf, Figure7.pdf, Table10.pdf and Table11.pdf or Table11-10-ImagesOnly.pdf.
- **Experiments:** Detailed in https://github.com/ace-compiler/ace-compiler/blob/main/README.md.
- **How much disk space required (approximately)?:** 20 GB.
- **How much time is needed to complete experiments (approximately)?:** 25+ hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache-2.0 WITH LLVM-exception.

A.3 Description

A.3.1 How to Access.

- Source code: https://github.com/ace-compiler/ace-compiler
- Docker Hub: opencc/ace:latest

A.3.2 Hardware Dependencies.

- To match configurations in this paper: Intel Xeon Platinum 8369B CPU @ 2.70 GHz with 512 GB memory.

A.3.3 Software Dependencies.

- Docker container version 25.0.1, other dependencies detailed in https://github.com/ace-compiler/ace-compiler/blob/main/Dockerfile.

A.4 Installation

There are two options to setup the artifact environments which are described on https://github.com/ace-compiler/ace-compiler. It is recommended to pull the pre-built docker image (opencc/ace:latest) from Docker Hub:

```
cd [YOUR_DIR_TO_DO_AE]
mkdir -p ace_ae_result
docker pull opencc/ace:latest
docker run -it --name ace -v "$(pwd)"/ace_ae_result:/app/
    ace_ae_result --privileged opencc/ace:latest bash
```

Alternatively, if you encounter issues pulling the pre-built image, you can build the image from the Dockerfile:

```
cd [YOUR_DIR_TO_DO_AE]
git clone https://github.com/ace-compiler/ace-compiler.git
cd ace-compiler
mkdir -p ace_ae_result
docker build -t ace:latest .
docker run -it --name ace -v "$(pwd)"/ace_ae_result:/app/
    ace_ae_result --privileged ace:latest bash
```

A local directory "ace_ae_result" is created and mounted in the docker container to collect the generated figures and tables.

A.5 Experiment workflow

- Build the ACE compiler, navigate to the /app directory within the container and run:

```
/app/scripts/build_cmplr.sh Release
```

- Build EXPERT hand-tuned implementations, in the /app directory of the container, run:

```
python3 /app/FHE-MP-CNN/build_cnn.py
```

- Run test, in the /app directory of the container, run:

```
python3 /app/scripts/perf.py -a
```

  A log file named with the date and time the command was launched will be generated, such as 2024_05_26_13_18.log. It will take approximately 18 hours for this command to complete in given hardware configurations.
- Reproduce Figure 5–7 and Table 10, in the /app directory of the container, run:

```
python3 /app/scripts/generate_figures.py -f 2024
    _05_26_13_18.log
```

- Build the ACE compiler with OpenMP support, in the /app directory of the container, run:

```
/app/scripts/build_cmplr_omp.sh Release
```

- Reproduce Table 11 with 10 images per model, in the /app directory of the container, run:

```
python3 /app/scripts/accuracy_all.py -n 10
```

  With 10 cores assumed to be available, the expected completion time is approximately 7 hours.
  Alternatively, to reproduce Table 11 with 1000 images per model, execute the following command in the /app directory of the container:

```
python3 /app/scripts/accuracy_all.py -n 1000
```

  This process will take over 140 hours to complete on the recommended computing platform, utilizing 64 threads.

A.6 Evaluation and Expected Results

During artifact evaluation, the newly generated Figures 5 and 6 may differ slightly from those in our paper due to variations in workload and I/O on the hardware. Similarly, the generated Figure 7 may exhibit minor differences compared to the one in our paper. However, the generated Table 10 should match exactly with the published version. Finally, the generated Table 11, which uses 10 images, should closely resemble the table at https://github.com/ace-compiler/ace-compiler/blob/main/scripts/Table11-10-ImagesOnly.png.

References

[1] 2021. Getting Started Converting TensorFlow to ONNX. Online: https://onnxruntime.ai/docs/tutorials/tf-get-started.html.

[2] 2023. torch.fx. Online: https://pytorch.org/docs/stable/fx.html.

[3] 2024. Lattigo v6. Online: https://github.com/tuneinsight/lattigo. EPFL-LDS, Tune Insight SA.

[4] 2024. ONNX Operators. Online: https://onnx.ai/onnx/operators/index.html.

[5] 2024. ONNX Runtime. Online: https://onnxruntime.ai/.

[6] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. 2022. OpenFHE: Open-Source Fully Homomorphic Encryption Library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (Los Angeles, CA, USA) *(WAHC'22)*. Association for Computing Machinery, New York, NY, USA, 53–63. https://doi.org/10.1145/3560827.3563379

[7] Martin R. Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin E. Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. 2019. Homomorphic Encryption Standard. *IACR Cryptol. ePrint Arch.* (2019), 939. https://eprint.iacr.org/2019/939

[8] Song Bian, Zian Zhao, Zhou Zhang, Ran Mao, Kohei Suenaga, Yier Jin, Zhenyu Guan, and Jianwei Liu. 2023. HEIR: A Unified Representation for Cross-Scheme Compilation of Fully Homomorphic Computation. Cryptology ePrint Archive, Paper 2023/1445. https://doi.org/10.14722/ndss.2024.23067

[9] Fabian Boemer, Yixing Lao, and Casimir Wierzynski. 2018. nGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data. *CoRR* abs/1810.10121 (2018). arXiv:1810.10121 http://arxiv.org/abs/1810.10121

[10] Jean-Philippe Bossuat, Rosario Cammarota, Jung Hee Cheon, Ilaria Chillotti, Benjamin R. Curtis, Wei Dai, Huijing Gong, Erin Hales, Duhyeong Kim, Bryan Kumara, Changmin Lee, Xianhui Lu, Carsten Maple, Alberto Pedrouzo-Ulloa, Rachel Player, Luis Antonio Ruiz Lopez, Yongsoo Song, Donggeon Yhee, and Bahattin Yildiz. 2024. Security Guidelines for Implementing Homomorphic Encryption. Cryptology ePrint Archive, Paper 2024/463. https://eprint.iacr.org/2024/463

[11] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. 2017. Fast Homomorphic Evaluation of Deep Discretized Neural Networks. Cryptology ePrint Archive, Paper 2017/1114. https://eprint.iacr.org/2017/1114

[12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2011. Fully Homomorphic Encryption without Bootstrapping. Cryptology ePrint Archive, Paper 2011/277. https://eprint.iacr.org/2011/277

[13] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2019. A Full RNS Variant of Approximate Homomorphic Encryption. In *Selected Areas in Cryptography – SAC 2018*, Carlos Cid and Michael J. Jacobson Jr. (Eds.). Springer International Publishing, Cham, 347–368. https://doi.org/10.1007/978-3-030-10970-7_16

[14] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017*. Springer International Publishing, Cham, 409–437. https://doi.org/10.1007/978-3-319-70694-8_15

[15] Eduardo Chielle, Oleg Mazonka, Homer Gamil, Nektarios Georgios Tsoutsos, and Michail Maniatakos. 2018. E3: A Framework for Compiling C++ Programs with Encrypted Operands. Cryptology ePrint Archive, Report 2018/1013. https://ia.cr/2018/1013.

[16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2018. TFHE: Fast Fully Homomorphic Encryption over the Torus. Cryptology ePrint Archive, Paper 2018/421. https://eprint.iacr.org/2018/421

[17] Thiago Crepaldi. 2024. Export a PyTorch model to ONNX. Online: https://pytorch.org/tutorials/beginner/onnx/export_simple_model_to_onnx_tutorial.html.

[18] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madanlal Musuvathi. 2020. EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 546–561. https://doi.org/10.1145/3385412.3386023

[19] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madan Musuvathi, and Todd Mytkowicz. 2019. CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing. In *PLDI 2019*. ACM, 142–156. https://www.microsoft.com/en-us/research/publication/chet-an-optimizing-compiler-for-fully-homomorphic-neural-network-inferencing/

[20] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Paper 2012/144. https://eprint.iacr.org/2012/144

[21] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. 2023. TensorFHE: Achieving Practical Computation on Encrypted Data Using GPGPU. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 922–934. https://doi.org/10.1109/HPCA56546.2023.10071017

[22] Craig Gentry. 2009. *A fully homomorphic encryption scheme*. Ph. D. Dissertation. Stanford University. https://dl.acm.org/doi/10.5555/1834954

[23] Craig Gentry and Shai Halevi. 2010. Implementing Gentry's Fully-Homomorphic Encryption Scheme. Cryptology ePrint Archive, Paper 2010/520. https://eprint.iacr.org/2010/520

[24] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2012. Homomorphic Evaluation of the AES Circuit. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7417)*, Reihaneh Safavi-Naini and Ran Canetti (Eds.). Springer, 850–867. https://doi.org/10.1007/978-3-642-32009-5_49

[25] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Advances in Cryptology – CRYPTO 2013*, Ran Canetti and Juan A. Garay (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 75–92. https://doi.org/10.1007/978-3-642-40041-4_5

[26] Armin Gerami, Monte Hoover, Pranav S. Dulepet, and Ramani Duraiswami. 2024. FAST: Factorizable Attention for Speeding up Transformers. *CoRR* abs/2402.07901 (2024). https://doi.org/10.48550/ARXIV.2402.07901 arXiv:2402.07901

[27] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings, Vol. 48)*, Maria-Florina Balcan and Kilian Q. Weinberger (Eds.). JMLR.org, 201–210. http://proceedings.mlr.press/v48/gilad-bachrach16.html

[28] Shruthi Gorantala, Rob Springer, Sean Purser-Haskell, William Lam, Royce Wilson, Asra Ali, Eric P. Astor, Itai Zukerman, Sam Ruth, Christoph Dibak, Phillipp Schoppmann, Sasha Kulankhina, Alain Forget, David Marn, Cameron Tew, Rafael Misoczki, Bernat Guillen, Xinyu Ye, Dennis Kraft, Damien Desfontaines, Aishe Krishnamurthy, Miguel Guevara, Irippuge Milinda Perera, Yurii Sushko, and Bryant Gipson. 2021. A General Purpose Transpiler for Fully Homomorphic Encryption. Cryptology ePrint Archive, Paper 2021/811. https://eprint.iacr.org/2021/811

[29] Gamze Gürsoy, Eduardo Chielle, Charlotte M. Brannon, Michail Maniatakos, and Mark Gerstein. 2020. Privacy-preserving genotype imputation with fully homomorphic encryption. *bioRxiv* (2020). https://doi.org/10.1101/2020.05.29.124412

[30] Shai Halevi and Victor Shoup. 2014. Algorithms in HElib. In *Advances in Cryptology – CRYPTO 2014*, Juan A. Garay and Rosario Gennaro (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 554–571. https://doi.org/10.1007/978-3-662-44371-2_31

[31] Kyoohyung Han and Dohyeong Ki. 2020. Better Bootstrapping for Approximate Homomorphic Encryption. In *Topics in Cryptology – CT-RSA 2020*, Stanislaw Jarecki (Ed.). Springer International Publishing, Cham, 364–390. https://doi.org/10.1007/978-3-030-40186-3_16

[32] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, and Jung Ho Ahn. 2022. ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1237–1254. https://doi.org/10.1109/MICRO56248.2022.00086

[33] Aleksandar Krastev, Nikola Samardzic, Simon Langoski, Srinivas Devadas, and Daniel Sanchez. 2024. A Tensor Compiler with Automatic Data Packing for Simple and Efficient Fully Homomorphic Encryption. In *Proc. ACM Program. Lang. 8, PLDI, Article 152, 25 pages*. https://doi.org/10.1145/3656382

[34] Kristin Lauter, Sreekanth Kannepalli, Kim Laine, and Radames Cruz Moreno. 2021. Password Monitor: Safeguarding passwords in Microsoft Edge. Online: https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge/.

[35] Eunsang Lee, Joon-Woo Lee, Junghyun Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and Woosuk Choi. 2021. Low-Complexity Deep Convolutional Neural Networks on Fully Homomorphic Encryption Using Multiplexed Parallel Convolutions. Cryptology ePrint Archive, Paper 2021/1688. https://eprint.iacr.org/2021/1688

[36] Eunsang Lee, Joon-Woo Lee, Jong-Seon No, and Young-Sik Kim. 2021. Minimax Approximation of Sign Function by Composite Polynomial for Homomorphic Comparison. *IEEE Transactions on Dependable and Secure Computing, 19(6)*, 3711–3727. https://doi.ieeecomputersociety.org/10.1109/TDSC.2021.3105111

[37] Long Li. 2024. ANT-ACE: an FHE Compiler Framework. https://doi.org/10.5281/zenodo.14625994

[38] Yan Liu, Jianxin Lai, Long Li, Tianxiang Sui an d Linjie Xiao, Peng Yuan, Xiaojing Zhang, Qing Zhu, Wenguang Chen, and Jingling Xue. 2025. ReSBM: Region-based Scale and Minimal-Level Bootstrapping Management for FHE via Min-Cut. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. https://doi.org/10.1145/3669940.3707276

[39] Qian Lou and Lei Jiang. 2021. HEMET: A Homomorphic-Encryption-Friendly Privacy-Preserving Mobile Neural Network Architecture. arXiv:2106.00038 [cs.CR]

[40] Guiwen Luo, Shihui Fu, and Guang Gong. 2023. Speeding Up Multi-Scalar Multiplication over Fixed Points Towards Efficient zkSNARKs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2023, 2 (2023), 358–380. https://doi.org/10.46586/TCHES.V2023.I2.358-380

[41] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2010. On Ideal Lattices and Learning with Errors over Rings. In *Advances in Cryptology – EUROCRYPT 2010*, Henri Gilbert (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–23. https://doi.org/10.1007/978-3-642-13190-5_1

[42] Steven Stanley Muchnick. 1998. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 340 Pine Street, Sixth Floor, San Francisco, CA, United States. https://dl.acm.org/doi/10.5555/286076

[43] Chris Peikert. 2016. A Decade of Lattice Cryptography. *Found. Trends Theor. Comput. Sci.* 10, 4 (2016), 283–424. https://doi.org/10.1561/0400000074

[44] Oded Regev. 2024. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. arXiv:2401.03703 [cs.CR]

[45] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Sahana Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. *IEEE/ACM International Symposium on Microrachitecture*, 238–252. https://doi.org/10.1145/3466752.3480070

[46] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. 2022. CraterLake: a hardware accelerator for efficient unbounded computation on encrypted data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 173–187. https://doi.org/10.1145/3470496.3527393

[47] SEAL 2020. Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA..

[48] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3315508.3329973

[49] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Furkan Turan, and Ingrid Verbauwhede. 2023. FPT: A Fixed-Point Accelerator for Torus Fully Homomorphic Encryption. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (, Copenhagen, Denmark,) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 741–755. https://doi.org/10.1145/3576915.3623159

[50] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. 2022. HECO: Automatic Code Optimizations for Efficient Fully Homomorphic Encryption. (2022). https://arxiv.org/abs/2202.01649

[51] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. 2012. Accelerating fully homomorphic encryption using GPU. In *2012 IEEE Conference on High Performance Extreme Computing*. 1–5. https://doi.org/10.1109/HPEC.2012.6408660

[52] Zhongcheng Zhang, Ying Li, Yuyang Zhang, Zhenchuan Chen, Jiacheng Zhao, XiaoBing Feng, Huimin Cui, and Jingling Xue. 2025. Qiwu: Exploiting Ciphertext-Level SIMD Parallelism in Homomorphic Encryption Programs. In *2025 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1145/3696443.3708917

[53] H. Zhu, T. Suzuki, and H. Yamana. 2023. Performance Comparison of Homomorphic Encrypted Convolutional Neural Network Inference Among HElib, Microsoft SEAL and OpenFHE. In *2023 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–7. https://doi.org/10.1109/CSDE59766.2023.10487709

Received 2024-05-29; accepted 2024-07-22