# HECATE: Performance-Aware Scale Optimization for Homomorphic Encryption Compiler

Yongwoo Lee
*Yonsei University*
Seoul, Republic of Korea
dragonrain96@yonsei.ac.kr

Seonyeong Heo
*ETH Zurich*
Zurich, Switzerland
seoheo@ethz.ch

Seonyoung Cheon
*Yonsei University*
Seoul, Republic of Korea
sunyoung7708@yonsei.ac.kr

Shinnung Jeong
*Yonsei University*
Seoul, Republic of Korea
shin0403@yonsei.ac.kr

Changsu Kim
*Seoul National University*
Seoul, Republic of Korea
kcs9301@snu.ac.kr

Eunkyung Kim
*Samsung SDS*
Seoul, Republic of Korea
ek41.kim@samsung.com

Dongyoon Lee
*Stony Brook University*
Stony Brook, USA
dongyoon@cs.stonybrook.edu

Hanjun Kim
*Yonsei University*
Seoul, Republic of Korea
hanjun@yonsei.ac.kr

***Abstract*—Despite the benefit of Fully Homomorphic Encryption (FHE) that supports encrypted computation, writing an efficient FHE application is challenging due to magnitude scale management. Each FHE operation increases scales of ciphertext and leaving the scales high harms performance of the following FHE operations. Thus, rescaling ciphertext is inevitable to optimize an FHE application, but since FHE requires programmers to match the rescaling levels of operands of each FHE operation, programmers should rescale ciphertext reflecting the entire FHE application. Although recently proposed FHE compilers reduce the programming burden by automatically manipulating ciphertext scales, they fail to fully optimize the FHE application because they greedily rescale the ciphertext without considering their performance impacts throughout the entire application.**

**This work proposes HECATE, a new FHE compiler framework that optimizes scales of ciphertext reflecting their rescaling levels and performance impact. With a new type system that embeds the scale and rescaling level, and a new rescaling operation called `downscale`, HECATE makes various scale management plans, analyzes their expected performance, and finds the optimal rescaling points throughout the entire FHE application. This work implements HECATE on top of the MLIR framework with a Python frontend and shows that HECATE achieves 27% speedup over the state-of-the-art approach for various FHE applications.**

***Index Terms*—Homomorphic encryption, compiler, privacy-preserving machine learning, deep learning**

## I. INTRODUCTION

Fully Homomorphic Encryption (FHE) [1]–[15] allows computations on a ciphertext without decrypting it first, enabling privacy-preserving computation offloading. In the form of FHE, decrypting the computation result on a ciphertext is identical to the computation result on a plaintext. Since FHE allows an application to safely outsource computation without exposing its data, it removes privacy barriers inhibiting data sharing and enables new services in highly regulated industries such as healthcare and financial business. Especially, the RNS-CKKS FHE scheme [7], one of the best performing state-of-the-art FHE schemes, is widely used in FHE libraries such as Microsoft's SEAL [16] and HEAAN [17].

Despite the benefits of FHE, existing FHE schemes [7]–[10] require in-depth knowledge about the FHE internals such as scale management to achieve desired performance. For example, RNS-CKKS [7] encodes a real number as an integer with a scale representing the position of decimal point before encryption. The scale exponentially increases with the degree of multiplications. Since a ciphertext with a larger scale requires more memory space and execution time to store and operate the ciphertext, to keep the scale under control without slowing down an FHE application, programmers should manually insert a parameter switching operation called *rescale* that reduces the scale by a fixed amount and increases its rescaling level. Moreover, since RNS-CKKS requires programmers to match the rescaling levels of the operands of each operation, scale manipulation requires huge programming efforts.

Recently proposed FHE compilers [18]–[21] reduce the programming burden by automatically managing ciphertext scales, but they fail to fully optimize FHE applications. The compilers trace the scale growth, automatically insert `rescale` operations where the scale of rescaled value becomes larger than the minimum scale, called *waterline*, and match the rescaling levels of FHE operands. However, the waterline rescaling does not consider its performance impact. In RNS-CKKS, the latency of an operation depends on the recaling levels of the operands, but existing scale management schemes do not consider them, losing performance optimization opportunities.

This work proposes HECATE, a new FHE compiler framework that optimizes scales of ciphertexts reflecting rescaling levels and their performance impacts. First, HECATE proposes a new parameter switching operation called `downscale` that rescales a ciphertext even though its scale is smaller than the sum of rescaling factor and *waterline*, thus enabling proactive rescaling. Second, HECATE analyzes ciphertexts and their FHE operations and generates scale management units by grouping ciphertexts that have the same scale and rescaling level. Third, HECATE constructs various scale management plans, estimates their performance, and finds the optimal scale management plan. Furthermore, to respect the constraints on scales and rescaling levels of FHE operands, HECATE introduces a new type system that verifies the scales and levels.

This work implements the HECATE framework on top of the MLIR framework [22] with a Python frontend. HECATE is evaluated with various machine learning and deep learning algorithms in terms of performance. The evaluation results show that the HECATE framework outperforms EVA, the state-of-the-art approach, with 27.38% speedup for the machine learning-based FHE applications, including neural networks such as multi-layer perceptron (MLP) and LeNet.

Followings are the contributions of this work.

- The HECATE language compiler that supports performance-aware scale optimization for FHE applications;
- the new parameter switching operation called `downscale` that enables proactive rescaling;
- the scale management space exploration scheme that optimizes ciphertext scales reflecting their performance impact;
- the new type system that reflects constraints on FHE operations and rescaling operations.

## II. BACKGROUND AND MOTIVATION

Homomorphic Encryption (HE) allows arithmetic operations such as addition and multiplication over encrypted data. HE guarantees that when decrypted, the result of the computation is the same as if the operations had been applied on the unencrypted data. While Leveled Homomorphic Encryption schemes [23]–[25] limit the maximum depth of multiplication, Fully Homomorphic Encryption (FHE) schemes [7]–[10] support an arbitrary number of arithmetic computations. Among the FHE schemes, this work targets RNS-CKKS [7] that is widely used in FHE libraries [16] and compilers [18]–[21].

### A. RNS-CKKS: The State-of-the-Art FHE Scheme

RNS-CKKS [7] is one of the best performing FHE schemes. RNS-CKKS is constructed on the Ring Learning with Errors (RLWE) problem [26] which adds and removes a small random error to a ciphertext during encryption and decryption. It is well suited to machine learning applications that include a large number of integer/fixed-point multiplications yet are inherently tolerant to some error noise. Thus, existing FHE libraries (*e.g.,* SEAL [16]) and FHE compilers (*e.g.,* CHET [18], EVA [19], and HECATE) mainly target RNS-CKKS.

**Encryption parameters:** RNS-CKKS require programmers to (manually) set two encryption parameters, *coefficient modulus* $Q$ and *polynomial modulus* $N$, which affect the performance, correctness, and security of an FHE application. Unfortunately, determining optimal $Q$ and $N$ parameters require in-depth knowledge about RNS-CKKS internals and constraints. For performance, $Q$ and $N$ are preferred to be set as small as possible because larger values of them increase the cost of FHE operations. However, they have to be set reasonably large for security and correctness reasons. For security, in theory, a larger $N$ provides a stronger security guarantee for a given $Q$. In practice, the minimum value of $N$ (given $Q$) against currently known attacks is available in [27]. For correctness, $Q$ should be set to be large enough to avoid so-called "scale overflow".

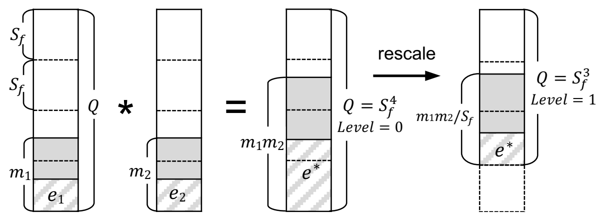


Fig. 1: FHE computation with RNS-CKKS. A ciphertext initially has coefficient modulus $Q$ and stores a message which has scale $m_i$ with noise $e_i$. After multiplication, the scale of the resulting ciphertext grows to $m_1m_2$ and noise grows to $e^*$. To keep the result scale at the same level, the rescaling operation is necessary that divides the coefficient modulus by the rescale factor that is predefined as $S_f$. After rescaling, the scale is reduced by $S_f^{-1}$. The level increases from 0 to 1.

**Scale:** RNS-CKKS encodes a vector of raw data into a plaintext cyclotomic polynomial [28] and encrypts the plaintext to a ciphertext using RLWE. More precisely, RNS-CKKS encodes a real number as an integer (representing the significand) with a *scale* (representing the decimal position). Then RNS-CKKS encrypts the integer as a ciphertext and stores the scale as its property. As shown in Fig. 1, coefficient modulus $Q$ represents an (initial) maximum scale capacity of a ciphertext. On multiplication of two ciphertexts with the scales of $m_1$ and $m_2$, the scale of the resulting ciphertext increases to $m_1m_2$.

RNS-CKKS requires that the scale remains less than $Q$ (Constraint 1). Otherwise, the scale overflow leads to a corrupted, unrecoverable result. To avoid the overflow (and to use smaller $Q$), RNS-CKKS provides a `rescale` operation that reduces the scale of a ciphertext by predefined rescaling factor $S_f$, as shown in Fig. 1. In this sense, $Q$ is represented as a product of small prime moduli $S_f$. On the other hand, the scale (after rescaling) should be larger than the minimum scale called *waterline* (Constraint 2). Otherwise, a message component (beyond an error) will lose its least significant bits and may be corrupted in the worst case.

**Level:** Each `rescale` operation consumes a modulus $S_f$. The rescaling *level* of a ciphertext indicates the number of consumed $S_f$, growing from zero and to $log_{S_f}(Q)$. RNS-CKKS requires that the operands of multiplication and addition be at the same level (Constraint 3). To manage the level, RNS-CKKS provides a `modswitch` operation that consumes $S_f$ without affecting the scale, and thus increases the level. If the level does not match between two operands, programmers should insert `modswitch` (or `rescale`) operations to adjust the level of ciphertexts.

In sum, Table I summarizes the semantics of the `rescale` and `modswitch` operations. In RNS-CKKS, each ciphertext is associated with two properties: scale and level. The `rescale` and `modswitch` operations should be (manually and correctly) used to meet the three constraints on the properties:

- (C1) The scale of a ciphertext should be less than coefficient modulus $Q$;
- (C2) The scale should be larger than waterline to avoid message corruption; and

TABLE I
FHE SCALE MANAGEMENT OPERATIONS CHANGE SCALE $j$ AND LEVEL $k$. $S_f$: RESCALING FACTOR. $S_w$: WATERLINE.

| Scale Mgmt. Operator | Scale $j$ | Level $k$ |
|---|---|---|
| `rescale` | $j/S_f$ (reduce) | $k+1$ |
| `modswitch` | $j$ (no change) | $k+1$ |
| `downscale` (new) | $S_w$ (reduce) | $k+1$ |

- (C3) The level of operands of multiplications and additions should be the same. (The addition also requires the same scale operands.)

## B. EVA: The State-of-the-Art FHE Compiler

To ease programmers' burden, FHE compilers [18]–[21] (for RNS-CKKS) have been proposed. Among them, the most recently proposed compiler, Encrypted Vector Arithmetic (EVA) [19] introduces two new concepts: *waterline rescaling* and *rescale chain*, to offer automatic scale management and parameter selection.

First, EVA automatically places `rescale` as needed to keep the scale of a ciphertext less than coefficient modulus $Q$ (C1), yet does so only when the rescaled scale remains higher than the *waterline* (C2). In EVA, the waterline is set to be the maximum scale of input ciphertexts. In other words, EVA attempts to ensure that the scales of intermediate and result ciphertexts always remain higher than the scales of input ciphertexts.

Second, EVA proposes the concept of *rescale chain* to satisfy the constraint (C3) that the operands of binary operations must be at the same level. The rescale chain encodes the sequence of `rescale` and `modswitch` operations from the root ciphertext to the target ciphertext. Since the `rescale` and `modswitch` operations increase the level of a result ciphertext, the number of the operations in a rescale chain represents the level of the ciphertext. EVA tracks the levels of ciphertexts along the rescale chains to satisfy the constraint (C3).

Figure 2a illustrates how EVA manipulates the scale of ciphertexts for example code that computes $(x^2+y^2)^3$. Since the input scale is $2^{20}$, EVA sets the waterline to be $2^{20}$. Multiplication such as $x^2$ and $y^2$ increases its scale to the product of the scales of its operands while addition keeps the same scale. Since the scale of $z^2$ becomes $2^{80}$ whose rescaled value $2^{20}$ become larger than the waterline, EVA places a `rescale` operation, which reduces its scale to $2^{20}$ and increases its level to 1. Since $z^3$ is the multiplication result of $z^2$ and $z$, and the level of the operands are different, EVA inserts a `modswitch` operation to increase the level of $z$ to 1.

## C. Motivation: Limitations of EVA

We discuss three limitations of EVA's scale management scheme, leading to suboptimal FHE performance.

**Limitation 1: A reactive fixed-factor scale management.** EVA's waterline rescaling scheme places a `rescale` operation if the scale remains larger than the waterline ($S_w$) after rescaling (to meet C1 and C2 constraints). EVA's rescaling is *reactive* in the sense that it reduces the scale after a multiplication increases the scale of the resulting ciphertext. Moreover, since the `rescale` operation reduces a scale only by the *fixed* factor $S_f$, EVA can insert the `rescale` operation only after the scale is larger than $S_f S_w$, thus losing a proactive optimization opportunity. We later show that with the new rescaling operator which can *proactively* change a scale by an *arbitrary* amount, thus enabling more efficient scale management.

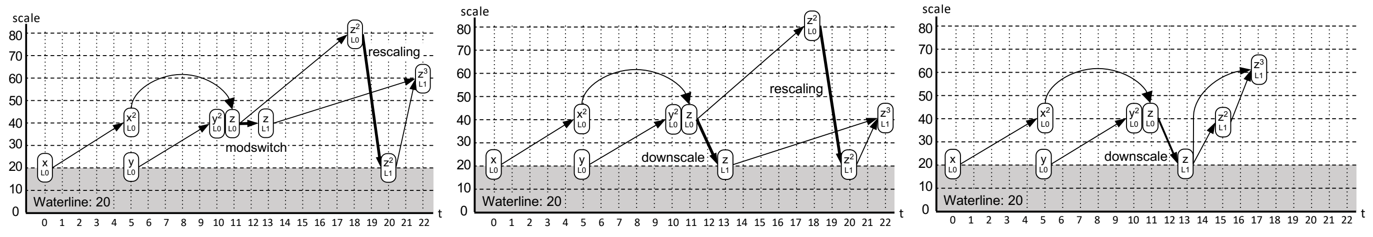
**Limitation 2: A separated scale and level analysis.** To meet both the scale (C1&C2) and level (C3) constraints, EVA employs a separated two-phase approach. EVA first performs waterline rescaling, placing `rescale` operations. Given the code instrumented with `rescale` operations, EVA then places `modswitch` operations as needed. In other words, EVA determines the scale first to meet the scale constraints, then adjusts the level separately to meet the level constraint. We later show that considering both scale and level holistically opens up a new scale management opportunity, and an FHE compiler can explore more optimal scale management alternatives.

**Limitation 3: A performance-oblivious scale management.** Given the assumption that the fixed-factor `rescale` is the only rescaling operator, the lower scale leads to better performance, and thus EVA solely focuses on reducing the scale. However, we later show that when a flexible rescaling becomes available, the lower scale does not always improve the performance. The cost of an FHE operation decreases as the rescaling level increases (*e.g.,* level 1 multiplication is 2.25× faster than level 0 multiplication). To achieve better performance, an FHE compiler should consider an alternative scale management option that may lead to a higher scale yet allow more computations to be performed at a higher level.

## III. OVERVIEW OF HECATE

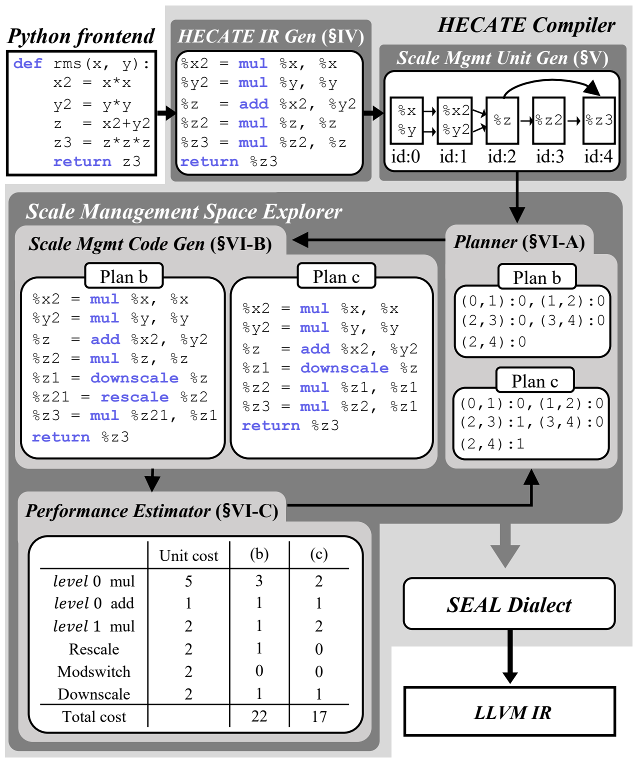
This work presents HECATE, a new FHE compiler, which supports performance-aware scale management of RNS-CKKS applications. HECATE enables performance-aware scale management by introducing (1) a new scale management operator called `downscale` and a new proactive rescaling scheme; (2) a new type system that analyzes scale and level together; (3) a new scale management space exploration (SMSE) with performance estimation.

**Solution 1: A proactive flexible-factor scale management.** HECATE introduces a new rescaling operator called `downscale` that can adjust the scale of a ciphertext by an *arbitrary* amount. Table I shows the semantic of `downscale` operation, which reduces the scale of a ciphertext exactly to the waterline ($S_w$) and increases the level by one. The `downscale` operation enables *proactive* scale management. Unlike EVA's waterline rescaling that reactively reduces the scale after a multiplication, HECATE can proactively downscale the scale of a ciphertext before a multiplication. For instance in Fig. 2a, to multiply the level one $z^2$ (after rescaling) and the level zero $z$, EVA introduces `modswitch` and increases the level of $z$. After the multiplication, the scale of $z^3$ becomes $2^{60}$. In Fig. 2b, HECATE uses `downscale` before the multiplication, reducing the scale of $z$ to the waterline $2^{20}$. In the end, the scale of $z^3$ becomes $2^{40}$ that is lower than EVA.



(a) Existing HE compiler (EVA) (b) Proactive rescaling (PARS) (c) Scale management space explorer (SMSE)

Fig. 2: Comparison of the scale management schemes of EVA and HECATE for the example program which calculates $(x^2 + y^2)^3$ which is a part of root mean square. `rescale` reduces the scale by $2^{60}$, and increases the level by one. `modswitch` only increases the level, and `downscale` reduces the scale to the waterline while increasing its level.



```
def rms(x, y):
    x2 = x*x
    y2 = y*y
    z  = x2+y2
    z3 = z*z*z
    return z3
```

```
%x2 = mul %x, %x
%y2 = mul %y, %y
%z  = add %x2, %y2
%z2 = mul %z, %z
%z3 = mul %z2, %z
return %z3
```

Plan b

```
%x2 = mul %x, %x
%y2 = mul %y, %y
%z  = add %x2, %y2
%z2 = mul %z, %z
%z1 = downscale %z
%z21 = rescale %z2
%z3 = mul %z21, %z1
return %z3
```

Plan c

```
%x2 = mul %x, %x
%y2 = mul %y, %y
%z  = add %x2, %y2
%z1 = downscale %z
%z2 = mul %z1, %z1
%z3 = mul %z2, %z1
return %z3
```

Plan b

```
(0,1):0,(1,2):0
(2,3):0,(3,4):0
(2,4):0
```

Plan c

```
(0,1):0,(1,2):0
(2,3):1,(3,4):0
(2,4):1
```

| | Unit cost | (b) | (c) |
|---|---|---|---|
| *level* 0 mul | 5 | 3 | 2 |
| *level* 0 add | 1 | 1 | 1 |
| *level* 1 mul | 2 | 1 | 2 |
| Rescale | 2 | 1 | 0 |
| Modswitch | 2 | 0 | 0 |
| Downscale | 2 | 1 | 1 |
| Total cost | | 22 | 17 |

Fig. 3: Design of the HECATE framework. The example code uses the same program with Fig. 2. Plan (b) and (c) are matched with Figures 2b and 2c.

**Solution 2: A new type system for scale and level management.** HECATE proposes a new type system that analyzes the scale and the level of a ciphertext together to enable better scale management. Unlike EVA, which adjusts the scales first and addresses the level differences later, HECATE considers the scale and level holistically to explore various scale management options.

For the same $(x^2 + y^2)^3$ computation, Fig. 2c illustrates yet another scale management option in which `downscale` is used even before the very first multiplication of $z * z * z$, where $z = (x^2 + y^2)$. The three scale management options in Fig. 2 demonstrate that different ways of using `rescale`, `modswitch`, and `downscale` may lead to different accumulative scales (and even the performance) The type system allows HECATE to safely explore the scale management space while obeying the RNS-CKKS constraints (C1-C3).

**Solution 3: A new scale management space exploration (SMSE) with performance estimation.** To achieve better performance, HECATE automatically explores the scale management space, determining where and which one of `rescale`, `modswitch`, and/or `downscale` to place. As the exploration space could be huge for large FHE applications, HECATE performs static analysis to identify scale management units where the scale and level can be managed together so that the scale management operators do not need to be placed in the middle of a unit. HECATE explores the scale management space by the hill-climbing method. Given the scale management units, HECATE constructs scale management plans by changing one step from the previous iteration. Then it generates FHE codes with scale management operators, which meet the constraints (C1-C3). Finally, it estimates the performance of each plan. The one with the best-estimated performance is passed to the next iteration until the hilltop is reached.

Fig. 2c shows the scale management plan that is estimated to perform the best. Note that the plan (c) has the accumulative scale of $2^{60}$ higher than the plan (b). However, the early introduction of `downscale` allows the two multiplications of $z*z*z$ (where $z = (x^2+y^2)$) to occur all at level one (instead of zero in the plan (b)), leading to better performance.

**Overview.** Fig. 3 shows the design of the HECATE framework. We build HECATE on the top of MLIR [22] for extensibility: to support various frontends (e.g. ONNX-MLIR, NumPy) and backends (other than SEAL) in the future. HECATE provides Python frontend to write an FHE program easily. HECATE IR generator produces the program written in HECATE IR (§IV) from the program written in Python frontend. Then, HECATE generates scale management units (§V). To explore the scale management space with the hill-climbing method, the planner (§VI-A) produces a set of next plans to explore from the best plan of the previous iteration. Given the plans HECATE generates correct and fast FHE codes considering the RNS-CKKS constraints (§VI-B). Finally, the performance estimator (§VI-C) calculates the estimated cost of the generated codes for the next iteration. For the final FHE code, HECATE provides SEAL dialect which represents the backend of an FHE program and optimizes memory usage by analyzing the liveness.

$$
\begin{aligned}
&Prg ::= \overline{F}\\
&F ::= \texttt{func}\,\mathit{fid}\,(\overline{v : \boxed{\mathrm{T}}})\,\{s; \overline{e}\}\\
&s ::= v := e \mid s; s\\
&e ::= \boxed{\mathit{homomorphic} \mid \mathit{opaque}}\\
&\boxed{\mathit{homomorphic}} ::= c \mid v \mid e \oplus e \mid -e \mid \texttt{rotate}\,(e, i)\\
&\boxed{\mathit{opaque} ::= \texttt{rescale}\,(e) \mid \texttt{modswitch}\,(e) \mid}\\
&\boxed{\qquad\qquad \texttt{upscale}\,(e, i) \mid \texttt{downscale}\,(e)}\\
&\oplus ::= + \mid \times\\
&T ::= \texttt{free} \mid \boxed{\texttt{scaled}\,(j, k)}\\
&\texttt{scaled} ::= \texttt{cipher} \mid \boxed{\texttt{plain}}
\end{aligned}
$$

$v \in$ variables, $c \in$ constants, $i, j, k \in \mathbb{Z}^+$, $\mathit{fid}$ : function id

Fig. 4: The formal syntax of HECATE IR. The syntax with gray box shows additional syntax for HECATE IR over its language. $\overline{A}$ means a list of $A$.

$$\frac{\Gamma \vdash e_1 : \texttt{cipher}\,(j, k) \quad \Gamma \vdash e_2 : \texttt{scaled}\,(j', k)}{\Gamma \vdash e_1 \times e_2 : \texttt{cipher}\,(j * j', k)} \tag{1}$$

$$\frac{\Gamma \vdash e_1 : \texttt{cipher}\,(j, k) \quad \Gamma \vdash e_2 : \texttt{scaled}\,(j, k)}{\Gamma \vdash e_1 + e_2 : \texttt{cipher}\,(j, k)} \tag{2}$$

$$\frac{\Gamma \vdash e : \texttt{cipher}\,(j, k) \quad j/S_f > S_w}{\Gamma \vdash \texttt{rescale}\,(e) : \texttt{cipher}\,(j/S_f, k+1)} \tag{3}$$

$$\frac{\Gamma \vdash e : \texttt{scaled}\,(j, k)}{\Gamma \vdash \texttt{modswitch}\,(e) : \texttt{scaled}\,(j, k+1)} \tag{4}$$

$$\frac{\Gamma \vdash e : \texttt{scaled}\,(j, k) \quad j' > j}{\Gamma \vdash \texttt{upscale}\,(e, j') : \texttt{scaled}\,(j', k)} \tag{5}$$

$$\frac{\Gamma \vdash e : \texttt{cipher}\,(j, k) \quad j < S_w \cdot S_f}{\Gamma \vdash \texttt{downscale}\,(e) : \texttt{cipher}\,(S_w, k+1)} \tag{6}$$

Fig. 5: Parts of HECATE type system.

## IV. HECATE IR AND TYPE SYSTEM

This section describes HECATE's intermediate representation (IR) (§IV-A) and scale/level-aware type system (§IV-B).

### A. HECATE *Intermediate Representation (IR)*

Fig. 4 presents the formal syntax of IR that HECATE compiler uses for scale management and type system. HECATE compiler takes as input an HE program written in Python language and compiles it down to HECATE IR. The syntax with a gray box represents HECATE's scale management operations that do not appear in the input program.

HECATE IR defines a function with typed arguments ($T$). Expressions in HECATE IR are classified into *homomorphic* and *opaque* expressions. *Homomorphic* expressions have the same semantics as their homomorphic counterparts. By the homomorphism of HE, the program should produce the same result when we change every ciphertext and its operations to plaintext and its homomorphic counterpart operations. An input program includes *homomorphic* expressions only.

On the other hand, *opaque* operations do not have a homomorphic counterpart. They only affect the data and properties of a ciphertext for scale management operations and the type system. The `rescale` and `modswitch` operations are directly mapped to the scale management operations in the RNS-CKKS scheme. The `upscale` $(e, i)$ operation is syntactic sugar for multiplying one with an arbitrary scale to increase the scale of $e$ by $i$ without changing the level.

HECATE introduces the new `downscale` $(e)$ operation which reduces the scale of $e$ to the waterline ($S_w$). HECATE implements the `downscale` operation by (1) multiplying the target ciphertext of scale $j$ with a plaintext filled with 1 of scale $S_f * S_w/j$ (increasing the scale to $S_f * S_w$) and (2) applying the `rescale` operation (reducing the scale to $S_w$ and increasing the level by one). Table I summarises the effects of the scale management operations on the scale and the level.

### B. HECATE *Type System*

HECATE IR uses Type $T$, representing the scale and level of a value. The type system allows HECATE to consider both scale and level during scale management and to produce a correct FHE program obeying the RNS-CKKS constraints. The `free` type represents a message that is not encoded or encrypted. The `plain` type represents a plaintext encoded but not encrypted. The `cipher` type represents a ciphertext encoded and encrypted. Both `plain` and `cipher` types have the scale $j$ and level $k$ properties. We refer to `plain` and `cipher` types with these properties as a `scaled` type because HE allows computation between plaintext and ciphertext.

**Homomorphic operations.** Homomorphic operations take an effect on the plain counterpart of the HE program. Binary operations between `cipher` type and `scaled` type operands have complex rules because of their constraints on the scale and level properties of the operands. For multiplication (Eq. 1), the level of the operands should be the same and the result scale becomes a product of scales of the operands. For addition (Eq. 2), the level and scale of the operands should be the same and also the result scale remains the same.

**Scale management operations.** The opaque scale management operations change the scale and level of the HE program. Eq. 3 describes the typing rule for the `rescale` operation, which only takes a `cipher` type operand. A `free` type operand does not have a scale. A `plain` type operand is only generated from the `free` type which does not decrease a scale. Thus, both do not need rescaling. A `rescale` operation reduces scale of an operand by a fixed factor $S_f$. The level increases by one. Because the waterline denotes the minimum required scale of ciphertext, HECATE allows rescaling only if the scale of ciphertext $j/S_f$ is larger than the waterline $S_w$. A `modswitch` operation takes a `scaled` type operand and increases only the level of ciphertext without affecting the other properties, as shown in Eq. 4. As described in Eq. 5, an `upscale` operation takes as operands a `scaled` type expression and the desired scale in integer. This operation is syntactic sugar that multiplies plaintext filled with 1 which has $j'/j$ scale. As in Eq. 6, a newly proposed `downscale` operation takes only a `cipher` type operand like the `rescale` operation, reduces the scale of ciphertext to the waterline, and increases the level by one.

**Algorithm 1:** Scale management unit analysis

```
Input: Func: Function of an HE application
Output: Group: Mapping from an ciphertext to SMU
Function ScaleMgmtUnitGrouping (Func) :
    Group ← {}
    MergeDef ← {} // Definition-aware Merge Step
    foreach (op, arg0, arg1, res) ∈ Func.getBody() do
        G ← Group.insert(res)
        G0 ← Group[arg0], G1 ← Group[arg1]
        if op ∈ {$+_p$} ∨ (op ∈ {$+_c$} ∧ G0 = G1) then
            Group.merge(G, G0)
        else
            def ← (op, G0, G1)
            if def ∈ MergeDef then
                Group.merge(G, MergeDef[def])
            else MergeDef[def] ← G
    end
    OpSplitDef ← {} // Operation-aware Split Step
    foreach G ∈ Group do
        foreach (op, arg0, arg1, res) ∈ G do
            Gres ← Group.split(G, res)
            def ← (op, G)
            if def ∈ OpSplitDef then
                Group.merge(Gres, OpSplitDef[def])
            else OpSplitDef[def] ← Gres
        end
    end
    UserSplitDef ← {} // Use-aware Split Step
    foreach G ∈ reverse(Group) do
        foreach (op,arg0,arg1,res) ∈ G do
            Gres ← Group.split(G,res)
            Guse ← {}
            foreach user ∈ res.getUsers() do
                Guse ← Group[user]
            end
            def ← (G, Guse)
            if def ∈ UserSplitDef then
                Group.merge(Gres, UserSplitDef[def])
            else
                UserSplitDef[def] ← Gres
                UserSplitDef[(G, Gres] ← Gres
        end
    end
end
```

Because downscale is done by `upscale` and `rescale`, it does not have meaning when `rescale` is possible. The type system only allows `downscale` when `rescale` cannot be applied.

## V. Scale Management Unit Generator

For scale management space exploration, HECATE analyzes a HECATE IR program and generates *scale management units* within which the scale and level can be managed together, reducing the search space. Algorithm 1 presents HECATE's three-phase scale management unit analysis algorithm with a custom *Group* data structure. *Group*.insert($res$) adds and returns a new set which can be indexed from "$res$". *Group*[$v$] finds a corresponding set from a value $v$. *Group*.merge($A$, $B$) merges sets $A$ and $B$ while keeping the key of $A$. *Group*.split($A$, $v$) splits and returns $\{v\}$ from $A$.

The first definition-aware merge step (Lines 4-14) performs a forward program analysis and groups a set of values sharing

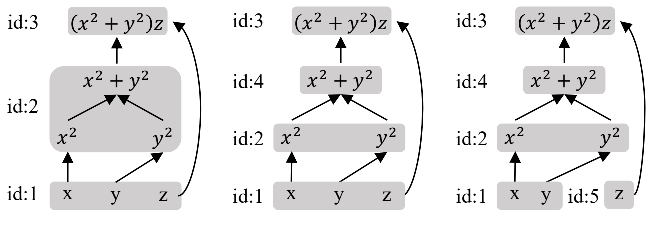


Fig. 6: Scale management unit analysis example for $(x^2 + y^2)z$

the same scale and level into the same unit. A plaintext addition ($+_p$) does not change the scale and level of a ciphertext. A ciphertext addition ($+_c$) does not alter them, if the two operands have the same scale and level. For the above two cases (Lines 7-9), HECATE puts the resulting plaintext/ciphertext and the operands in the same unit. On the other hand, for a ciphertext addition with the operands at different scales/levels that require a scale management operation, and a ciphertext multiplication that changes the scale of resulting ciphertext, HECATE introduces a new scale management unit if there is no previous case with the same operator and operand group combination (Lines 10-13). For the example in Fig. 6a, the definition-aware merge step puts ($x^2$, $y^2$, and $x^2+y^2$) with the same scale/level into one unit. The next operation $(x^2+y^2)z$ increases the scale, so it goes to another unit.

The second step (Lines 16-24) splits the scale management units constructed by the first step into smaller units based on the operation type (See Line 19 that uses $op$ as a key). The second split step tries to split the multiplication prefix from the rest non-multiplication suffix. The rationale behind this split pass is that the prefix multiplications always generate the result with a scale larger than ${S_w}^2$, so there is room for proactive scale management. As the scale of multiplication operand is larger than $S_w$, the scale of the prefix multiplication is always larger than $S_w * S_w$. For the same example in Fig. 6a, since the first two operations are $mul$ and the third is $add$, the operation-aware split step splits the unit ($x^2$, $y^2$, and $x^2+y^2$) into two units ($x^2$, $y^2$) and ($x^2+y^2$), as shown in Fig. 6b.

The last user-aware split step (Lines 26-40) further partitions the scale management units. If ciphertexts are used with different scales and levels, it is worth considering different scale management plans for them. This step analyzes how each ciphertext is "used" in a backward way, and places ciphertexts used in different units separately. In the same example in Fig. 6b, the use of $x$ and $y$ is $x^2$ and $y^2$; and the use of $z$ is $(x^2+y^2)z$. The third step splits $(x, y)$ and $(z)$. Fig. 6c shows the final scale management units.

## VI. Scale Management Space Explorer

As shown Fig. 3, HECATE's Scale Management Space Explorer (SMSE) consists of three components: scale management planner (§VI-A), code generator (§VI-B), and performance estimator (§VI-C).

## A. Scale Management Planner

The planner takes as input scale management units (§V) and the best plan in a previous iteration and produces a set of new scale management plans. Each management plan maps an edge between scale management units to *an optimization degree*, representing the number of additional scale management operations such as `rescale`, `downscale`, and `modswitch`.

The planner produces new scale management plans using the steepest ascent hill-climbing method. Given the best plan from the previous iteration, the planer generates a set of new plans by increasing the optimization degree by one at each different location. Suppose a program consists of three scale management units 0, 1, and 2 with two edges (0,1) and (1,2). Given the best plan {(0, 1): 1, (1, 2): 0}, the new plans are: {(0, 1): 2, (1, 2): 0} and {(0, 1): 1, (1, 2): 1}. In other words, the planner attempts to introduce one more scale management operation than the prior iteration.

Each plan gives (a) the locations to place scale management operations and (b) the numbers of operations to place. Given a scale management plan, the planner determines (c) the type of scale management operations to place, based on the scale of an operand. If the scale is larger than the waterline after `rescale`, the planner applies `rescale` to the operand. If `rescale` cannot be applied but the scale can be reduced by `downscale`, the planner applies `downscale`. In the other case, `modswitch` is applied. Because this algorithm needs the scale information of the operand and should meet the RNS-CKKS constraints, the planner uses the proactive rescaling algorithm (PARS, Algorithm 2), described in the next section, to generate the correct FHE codes.

**Algorithm 2:** Proactive rescaling algorithm (PARS)

```
Parameter: $S_f$: Rescale Value
Parameter: $S_w$: Waterline Value
Input: Op: The operation type of HE operation.
Input: arg0, arg1: Argument ciphertext of an HE operation.
Input: res: Result ciphertext of an HE operation.
1  Function PARS (op, arg0, arg1, res) :
2      // (a) Encode
3      if $op \in \{+, \times\}$ then
4          // Without Loss of Generality
5          if arg0.type = free then
6              arg0.type ← plain (arg0, $S_w$)
7      // (b) Rescale Analysis
8      // Without Loss of Generality
9      if arg0.scale $> S_w \cdot S_f$ then
10         arg0 ← rescale (arg0)
11     // (c) Level Match
12     // Without Loss of Generality
13     if $op \in \{+, \times\} \wedge$ arg0.level < arg1.level then
14         if arg0.scale = $S_w$ then
15             arg0 ← modswitch (arg0)
16         else if arg0.scale $> S_w$ then
17             arg0 ← downscale (arg0)
18     // (d) Scale Match
19     // Without Loss of Generality
20     if $op \in \{+\} \wedge$ arg0.scale < arg1.scale then
21         arg0 ← upscale (arg0, arg1.scale)
22     // (e) Downscale Analysis
23     if $op \in \{\times\} \wedge$ arg0.scale * arg1.scale $> S_w^2 \cdot S_f$ then
24         arg0 ← downscale (arg0)
25         arg1 ← downscale (arg1)
26 end
```

## B. Scale Management Code Generation

Algorithm 2 describes how HECATE inserts scale management operations. HECATE greedily reduces the cumulative scale of each ciphertext. The key idea is to apply a scale management operation to minimize the cumulative scale of operands for each operation, leading to the minimum cumulative scale of the result of that operation. The algorithm consists of five steps: (a) encode, (b) `rescale` analysis, (c) level match, (d) scale match, and (e) `downscale` analysis. Additionally, HECATE applies early modswitch optimization of EVA to move `modswitch` to the earlier position.

**(a) Encode.** This step specifies the scale of a `free` type operand of binary expressions. If an operand is `free`-type, HECATE transforms it to `plain`-type with the waterline $S_w$.

**(b) Rescale analysis.** This step rescales the operands of each operation if the new scale remains larger than the waterline. The `rescale` operation reduces the scale from $j$ to $j/S_f$, given a rescale factor of $S_f$. Thus, if the new scale is larger than the waterline $S_w$: i.e., $j > S_w S_f$, HECATE inserts `rescale` to reduce the scale of an operand in binary expressions (Line 9-10). This transformation can be commutatively and recursively applied to generate the minimal scale as long as the waterline is ensured.

**(c) Level match.** This step satisfies the level constraint of binary expressions: i.e., the levels of two operands should be the same. If the scale of the smaller level operand is the waterline, HECATE inserts `modswitch` to increase the level of the operand as EVA does (Line 14-15). If the waterline $S_f$ is less than scale $j$ as well, If not, HECATE uses `downscale` to minimize the scale while increasing the level (Line 16-17). Note that after steps (a) to (c), HECATE guarantees that two operands of every binary expression have the types with the same level: i.e., $(j, k)$ and $(j', k)$.

**(d) Scale match.** This step satisfies the scale constraint of add operations: i.e., the scale of two operands should be the same. To match the scale, HECATE inserts the `upscale` operation to the operand with a smaller scale.

**(e) Downscale analysis.** The last step is to analyze if applying `downscale` on two operands of multiplication is beneficial. Give two operands of types $\texttt{scaled}(j, k)$ and $\texttt{scaled}(j', k)$ for multiplication (after steps (a) to (d)), there are two possible ways to manage the scale of the multiplication. One method is to simply multiply them first, and then apply `downscale`. The type of the result becomes $(jj'/S_f, k+1)$. An alternative method is to apply `downscale` on both operands and then to multiply them. In this case, after `downscale`, the types of two operands will be $(S_w, k+1)$ and $(S_w, k+1)$, respectively. After the multiplication, the type of the result becomes $(S_w^2, k+1)$. HECATE applies `downscale` if $jj'/S_f > S_w^2$ (Line 23-25).

TABLE II
RMS ERROR OF THE PROGRAMS

| Benchmark | EVA | PARS | SMSE | HECATE |
|---|---|---|---|---|
| SF | 9.738E-04 | 3.799E-03 | 2.503E-04 | 3.680E-03 |
| HCD | 3.313E-03 | 1.675E-03 | 8.102E-04 | 3.265E-03 |
| MLP | 4.634E-04 | 6.040E-05 | 5.521E-05 | 3.257E-04 |
| Lenet | 1.126E-03 | 2.584E-04 | 5.004E-04 | 2.183E-03 |
| LR E2 | 2.296E-04 | 1.654E-04 | 6.493E-06 | 2.525E-03 |
| LR E3 | 2.742E-05 | 1.784E-03 | 7.107E-08 | 2.716E-04 |
| PR E2 | 3.266E-03 | 1.748E-04 | 7.566E-04 | 1.333E-03 |
| PR E3 | 1.721E-04 | 5.713E-04 | 2.613E-03 | 1.788E-03 |
| Gmean | 5.271E-04 | 4.823E-04 | 9.195E-05 | 1.390E-03 |

### C. Performance Estimator

The performance estimator statically estimates an expected execution time of a HECATE IR program. Dynamically executing all candidate programs easily becomes too expensive. The latency of an FHE operation in RNS-CKKS is determined by the level of the operands and the polynomial modulus $N$. The time complexity of the operation is linear or quadratic to the level of the operands, and linear or log-linear to $N$, depending on the operation type. Based on the observation, we profile the execution time of each FHE operation at different levels and $N$. as in Fig. 3. Given the profiled per-level latency of each FHE operation, the performance estimator can estimate the total execution time of a HECATE IR program. For each operation, the level information is readily available thanks to the HECATE's type system.

## VII. EVALUATION

### A. Experimental Setup

For performance evaluation, this work compares the performance of HECATE with the state-of-the-art EVA [19] using a set of benchmarks. HECATE employs proactive rescaling (PARS, §VI-B) and scale management space exploration (SMSE, §VI). Thus, this work also evaluates the benefits of an individual part. In sum, this work considers four different scale management schemes:

- **EVA** uses the fixed-factor scale management and the *waterline rescaling* algorithm. These two components of EVA are reimplemented on top of the HECATE framework.
- **PARS** uses the proactive rescaling scheme but does not use the scale management space exploration.
- **SMSE** uses the scale management space exploration but does not use the proactive rescaling. Instead, it uses EVA's waterline rescaling algorithm.
- **HECATE** uses both proactive rescaling and scale management space exploration.

For benchmarks, we implemented and tested the following six applications. We tested the same benchmark set (except SqueezeNet) used in EVA and CHET (e.g., LR, SF, LeNet). We additionally tested MLP.

- **Sobel Filter (SF)** is a classic edge detection algorithm, which calculates the variation of an image in the vertical and horizontal directions using $3 \times 3$ image gradient filters.

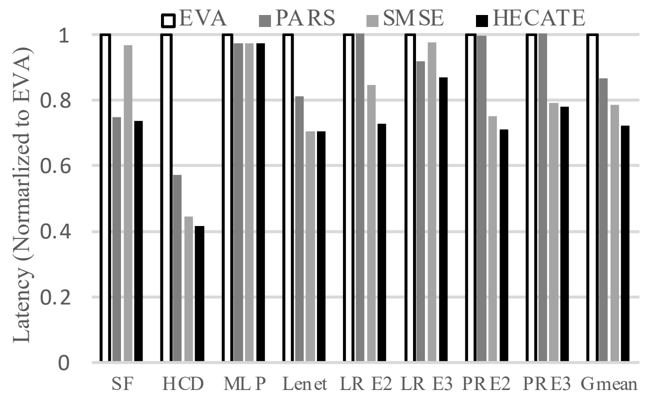


Fig. 7: Performance of 8 benchmarks with different scale management schemes: Sobel Filter, Harris Corner Detection, Multi-layer Perceptron, LeNet-5, Linear Regression and Polynomial Regression,. This work executes regression benchmarks with two and three epochs (denoted as E2 and E3).

- **Harris Corner Detection (HCD)** detects corner points of an image. HCD calculates the difference of pixels in the window and detects corner points.
- **Multi-layer Perceptron (MLP)** is a feed-forward neural network for image classification. This benchmark uses $784 \times 100$ and $100 \times 10$ layers with square activation.
- **LeNet-5** is a convolutional neural network for image classification. This benchmark uses the network presented in [29], modifying the output channel of the second fully connected layer to 64 and activation function to square function.
- **Linear Regression (LR)** models the relationship between a dependent variable and independent variables by fitting a linear equation for given input data.
- **Polynomial Regression (PR)** is one of the regression methods for nonlinear data using the $n$-th degree polynomial equation. This work uses the quadratic equation.

This work uses Microsoft SEAL (Release 3.5.9) and conducts all the experiments with Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz CPU with 6 physical cores and 64GB RAM. This work sets the security level as 128-bit for all the experiments. With the software and hardware specification, this work writes the benchmarks assuming a packed ciphertext with $2^{14}$ slots. The image processing benchmarks only use 4096 pixels of 64 $\times$ 64 images, the regression benchmarks use 16384 randomly generated inputs for each variable, and the deep learning benchmarks use a random input from the MNIST dataset. This work uses the gradient descent algorithm for the regression benchmarks with 2 and 3 epochs.

### B. Performance Evaluation

Figure 7 shows the minimum latency of each benchmark when setting the maximum error bounds as $2^{-8}$. For all the schemes, this work tries 36 different waterlines and finds the best waterline in terms of latency among the cases that do not exceed the maximum error bound. The exact root-mean-square error of the compiled program is presented in Table II. Note that smaller error does not imply a better exploration

TABLE III
SEARCH SPACE REDUCTION. (USES: # OF USES, SMU: # OF SCALE MANAGEMENT UNITS, EPOCH: # OF EXPLORATION ITERATIONS, PLAN: # OF EXPLORED SPACES)

| Bench marks | uses | SMU | Naive | | Hecate | | Reduction Ratio |
|---|---|---|---|---|---|---|---|
| | | | epoch | plan | epoch | plan | |
| SF | 91 | 19 | 5 | 1093 | 5 | 229 | 4.773 |
| HCD | 164 | 19 | 34 | 16237 | 7 | 343 | 47.34 |
| MLP | 575 | 12 | 2 | 1726 | 2 | 37 | 46.65 |
| Lenet | 11735 | 48 | 43 | 1.48E6 | 6 | 721 | 2050 |
| LR E2 | 186 | 44 | 13 | 6697 | 10 | 1189 | 5.632 |
| LR E3 | 278 | 66 | 12 | 9175 | 11 | 1981 | 4.631 |
| PR E2 | 284 | 71 | 13 | 10225 | 10 | 1918 | 5.331 |
| PR E3 | 424 | 106 | 18 | 21625 | 12 | 3499 | 6.180 |

because it means there is no better optimization option found that sacrifices the precision to get better performance.

The evaluation results show that HECATE can enhance the performance of various HE applications with PARS and SMSE. Performance improvement from PARS and SMSE is 13.38% and 21.36% on average, respectively. Overall performance improvement of HECATE is 27.85% on average.

**Speedup of PARS.** Although PARS always achieves a smaller cumulative scale which defines the initial level of the program, PARS does not always show the speedup over EVA. For SF, HCD, and Lenet, PARS improves the performance by reducing the cumulative scales and their initial levels of ciphertexts. However, PARS does not improve the performance of MLP, LR E2 and E3, and PR E2 and E3 because their cumulative scale reduction does not exceed a program-specific tipping point of level reduction, and thus their initial levels of ciphertexts are not reduced. PARS averagely shows 13.38% speedup over EVA on average.

**Speedup of SMSE.** SMSE always shows the speedup over EVA. Because SMSE rejects the optimization plan when the estimated latency is slower than the previous plan, it is clear that SMSE always introduces the speedup if the performance estimation is accurate. As described in §VII-D, the estimation is accurate enough to explain the 21.35% speedup of SMSE.

However, SMSE shows a little speedup on SF, MLP, and LR E3. Other schemes also do not achieve meaningful speedup on MLP and LR E3 which implies that EVA already finds good parameters for the program. For SF, its number of SMU edges is smaller than others, thus limiting scale management space. Since the impact of SMSE is limited, its performance speedup largely relies on the code generation scheme such as PARS.

**Speedup of HECATE.** HECATE is superior to the other optimization schemes. For SF, HCD, and LR E3, HECATE achieves synergistic performance improvement by using both SMSE and PARS. Results of LR E2, PR E2, and PR E3 show how code generation can affect the speedup of HECATE. Although solely using PARS does not improve the performance of the benchmarks, HECATE achieves better latency than SMSE with waterline rescaling, because the code generation can affect the effectiveness of the optimization plan. The result of Lenet shows the impact of SMSE. Not surprisingly, SMSE can add the `downscale` operation on the proper place and may optimize

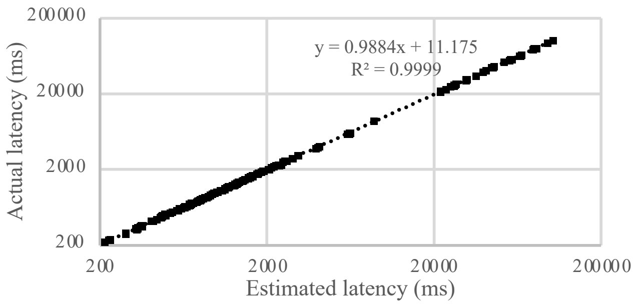


Fig. 8: Comparison between estimated and actual latencies. The comparison plots the data of 1152 different settings that uses 36 different waterlines for 8 benchmarks and 4 optimization schemes. The maximum relative error is 4.8%

the same places that PARS optimizes. Thus, even with waterline rescaling, SMSE can find a similar scale management plan like HECATE. On average, HECATE shows the best performance speedup of 27.85%.

Hecate achieves a meaningful performance improvement via compiler-only optimizations (without additional hardware nor algorithmic change). Our optimization can be synergistically applied and further improve the performance of recent hardware acceleration schemes such as HEAX [30].

## *C. Search Space Reduction*

This section demonstrates how the scale management unit generation reduces the search space. Without scale management unit generation, the optimization plan should be applied to every use of the ciphertexts in a program. The number of explored plans is increased not only by the number of the edges, but also by the epoch of the hill-climbing algorithm. To analyze how effectively scale management unit generation reduces the exploration space of SMSE, this work implements a naïve exploration scheme that explores scale management plans with the same hill-climbing algorithm but without scale management units. Thus, the naïve scheme directly uses use-def edges in the program.

Table III shows the comparison results of the naïve scheme and HECATE. For SF and MLP, using SMU does not reduce the number of epochs, which means the optimized plan only operates on a single-use SMU edge. However, these benchmarks take advantage of the reduction of the number of units. SMU generation of HECATE slightly reduces the number of the epoch of the regression benchmarks such as LR E2, LR E3, PR E2, and PR E3. The regression benchmarks have a few parallel operations that can share the schedule, so the reduction of the epoch is small. For Lenet and HCD, the epochs are dramatically reduced because the benchmark exposes the parallel operations a lot. The result of Lenet shows that the SMU scales well with the size of the program. The compilation time depends on the number of iterations during SMSE. Thanks to the proposed search space reduction (Table III), the longest compilation time of HECATE only takes 340 seconds compared to the 649 hours of the naïve scheme.

## D. Performance Estimation

Performance estimation largely affects the optimization of SMSE. As shown in Fig. 8, the estimated latency of various settings is almost identical to the actual latency of the compiled program. The geometric average of relative error is 1.3% and the maximum error is 4.8%. This result shows that the simple estimation explained in §VI-C is sufficient for SMSE. Inherently, the latency of each HE operation has a very small variance, because HE operations execute long regular loops enough to reduce the variance on the latency of each iteration by a statistical effect. The performance characteristic of HE operations explains why the proposed estimation accurately estimates the latency.

## VIII. Related Work

Previous work [18]–[21], [31]–[37] proposes various languages and optimizing compilers for HE to enhance the performance and programmability of HE applications. Survey [38] on FHE compilers and libraries provides an extensive survey and experimental evaluation. In general, the existing languages and compilers hide the complexities of HE schemes by providing high-level languages or automating encryption parameter selection. Moreover, the compilers introduce various optimization techniques for HE applications.

**General-purpose HE compilers:** Some work [19], [31]–[35], [39]–[41] supports implementing general-purpose HE applications either for existing programming languages or with proposing new programming languages.

The HE compilers for existing programming languages [31]–[34], [39]–[41] enable programmers to implement HE applications in a friendly way. Cingulata [39], [40], $E^3$ [33] and Marble [32] provides open-source compiler and runtime supports to executing C++ programs over encrypted data. Lobster [41] is an optimizing compiler for Cingulata, which optimizes boolean circuits generated by Cingulata through program synthesis and term rewriting. RAMPARTS [31] automatically transforms an imperative program in Julia into an optimized computation circuit for HE, using the PALISADE [42] library as its back-end. ALCHEMY [34] supports Haskell front-end and automatically selects suitable parameters. Additionally, Porcupine [35] proposes a Quill DSL for data layout optimizations in FHE using program synthesis.

While the compilers either target non-CKKS schemes [9], [10], [43] only or lack consideration of scale management, this work focuses on providing optimal scale management for the state-of-the-art CKKS and RNS-CKKS schemes.

Recently, EVA (Encrypted Vector Arithmetic) [19] introduces a new FHE language, which can be an intermediate representation for other domain-specific languages. EVA supports arithmetic operations on fixed-width vectors and scalars, facilitating encrypted SIMD computations. EVA also includes an optimizing compiler, which provides FHE-specific optimizations such as inserting `rescale` and `modswitch` operations optimally. EVA reduces the number of `rescale` operations based on the insights that reusing the same rescale value can lead to smaller encryption parameters.

Similar to EVA, the proposed HECATE framework supports general-purpose FHE applications with performance-aware scale management. This work discovers three limitations in EVA in terms of scale management: a reactive fixed-factor scale management, separated scale and level analysis and the performance-oblivious scale management. The HECATE framework provides better scale management than EVA with a new operation and type system and a design space exploration approach for performance-aware scale management.

**Domain-specific HE compilers:** Other work [18], [20], [21], [37] targets specific domains like DNN inference.

CHET [18] is an optimizing compiler for homomorphic DNN inference. CHET includes a domain-specific language for specifying tensor circuits that implement DNN inference. From domain-specific information such as the dimensions of input tensors, CHET transforms an input tensor circuit with FHE operations. To hide the complexities of FHE schemes, CHET automates parameter selection and guarantees the security and accuracy of the target tensor circuit. Furthermore, CHET provides layout selection and other HE-specific optimizations to enhance the performance of the target circuit.

nGraph-HE [20], [21] enables homomorphic deep learning by extending an existing deep learning graph compiler, Intel's nGraph [44]. By implementing an HE backend for nGraph, the compilers facilitate deploying neural network models with popular deep learning frameworks such as TensorFlow [45]. In addition, the compilers apply various HE-aware optimizations like exploiting SIMD operations that HE schemes support. Especially, nGraph-HE provides a CKKS-specific optimization called *lazy rescaling*, which inserts `rescale` operations only after fully-connected and convolutional layers.

AHEC [37] supports nGraph and Tensorflow as front-end and CKKS scheme of SEAL library as a backend. Moreover, AHEC also provides multiple hardware backend with GPU-accelerated HE library. AHEC mainly supports automated kernel generation with vectorization, tiling, and tensor layout selection. AHEC also provides Tile DSL to describe the HE kernel and hardware abstraction layer for parallelization.

Since the scale management scheme of HECATE can apply to the existing domain-specific compilers, the compilers can further enhance the performance of their target applications by employing HECATE's performance-aware scale management.

**Scale management of (non-FHE) fixed-point arithmetic:** [46]–[51] optimize the bitwidth of fixed-point numbers and reduce the complexity of arithmetic operations. However, they cannot be directly applied to RNS-CKKS without considering the impact of scale management on performance.

## IX. Conclusion

This work proposes the HECATE compiler framework that allows performance-aware scale management with a new type system, scale management operations including a new `downscale` operation, proactive rescaling algorithm, and scale management space explorer. With the proposed proactive rescaling and scale management space exploror, HECATE achieves 27.38% speedup over the state-of-the-art approach.

## Acknowledgment

We thank the anonymous reviewers for their valuable feedback. We also thank the CoreLab members for their support and feedback during this work. This work is supported by IITP-2020-0-01847, IITP-2020-0-01361 and IITP-2021-0-00853 through the Institute of Information and Communication Technology Planning and Evaluation (IITP) funded by the Ministry of Science and ICT. This work is also supported by Samsung SDS. *(Corresponding author: Hanjun Kim)*

## References

[1] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford, CA, USA, 2009.

[2] ——, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, ser. STOC '09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 169–178. [Online]. Available: https://doi.org/10.1145/1536414.1536440

[3] C. Gentry and S. Halevi, "Implementing gentrys fully-homomorphic encryption scheme," vol. 6632, 05 2011, pp. 129–148.

[4] J.-S. Coron, A. Mandal, D. Naccache, and M. Tibouchi, "Fully homomorphic encryption over the integers with shorter public keys," in *Advances in Cryptology – CRYPTO 2011*, vol. 6841, 08 2011, pp. 487–504.

[5] J.-S. Coron, D. Naccache, and M. Tibouchi, "Public key compression and modulus switching for fully homomorphic encryption over the integers," in *Advances in Cryptology – EUROCRYPT 2012*, 04 2012, pp. 446–464.

[6] J. H. Cheon, J.-S. Coron, J. Kim, M. S. Lee, T. Lepoint, M. Tibouchi, and A. Yun, "Batch fully homomorphic encryption over the integers," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2013, pp. 315–335.

[7] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full rns variant of approximate homomorphic encryption," in *Selected Areas in Cryptography – SAC 2018*, C. Cid and M. J. Jacobson Jr., Eds. Cham: Springer International Publishing, 2018, pp. 347–368.

[8] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology – ASIACRYPT 2017*, T. Takagi and T. Peyrin, Eds. Cham: Springer International Publishing, 2017, pp. 409–437.

[9] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Report 2012/144, 2012, https://eprint.iacr.org/2012/144.

[10] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ser. ITCS '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 309–325. [Online]. Available: https://doi.org/10.1145/2090236.2090262

[11] J.-S. Coron, T. Lepoint, and M. Tibouchi, "Scale-invariant fully homomorphic encryption over the integers," in *International Workshop on Public Key Cryptography*. Springer, 2014, pp. 311–328.

[12] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) lwe," *SIAM Journal on Computing*, vol. 43, no. 2, pp. 831–871, 2014.

[13] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *Annual Cryptology Conference*. Springer, 2012, pp. 868–886.

[14] C. Gentry, S. Halevi, and N. P. Smart, "Fully homomorphic encryption with polylog overhead," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2012, pp. 465–482.

[15] ——, "Better bootstrapping in fully homomorphic encryption," in *International Workshop on Public Key Cryptography*. Springer, 2012, pp. 1–16.

[16] "Microsoft SEAL (Release 3.5.9)," https://github.com/microsoft/SEAL, 2020.

[17] "HEAAN Open-Source HE Library," https://github.com/snucrypto/HEAAN, 2020.

[18] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "CHET: An Optimizing Compiler for Fully-homomorphic Neural-network Inferencing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. ACM, 2019, pp. 142–156. [Online]. Available: http://doi.acm.org/10.1145/3314221.3314628

[19] 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 *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. Association for Computing Machinery, 2020, pp. 546–561. [Online]. Available: https://doi.org/10.1145/3385412.3386023

[20] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski, "nGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, ser. CF '19. ACM, 2019, pp. 3–13. [Online]. Available: http://doi.acm.org/10.1145/3310273.3323047

[21] F. Boemer, A. Costache, R. Cammarota, and C. Wierzynski, "nGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data," in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. WAHC'19. ACM, 2019, pp. 45–56. [Online]. Available: http://doi.acm.org/10.1145/3338469.3358944

[22] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, pp. 2–14. [Online]. Available: https://ieeexplore.ieee.org/document/9370308/

[23] Y. Ishai and A. Paskin, "Evaluating branching programs on encrypted data," in *Proceedings of the 4th Conference on Theory of Cryptography*, ser. TCC'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 575–594.

[24] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-dnf formulas on ciphertexts," in *Proceedings of the Second International Conference on Theory of Cryptography*, ser. TCC'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 325–341. [Online]. Available: https://doi.org/10.1007/978-3-540-30576-7_18

[25] T. Sander, A. Young, and Moti Yung, "Non-interactive cryptocomputing for nc/sup 1/," in *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, 1999, pp. 554–566.

[26] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Advances in Cryptology – EUROCRYPT 2010*, H. Gilbert, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[27] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan, "Homomorphic encryption security standard," HomomorphicEncryption.org, Toronto, Canada, Tech. Rep., November 2018.

[28] Z. Brakerski, C. Gentry, and S. Halevi, "Packed Ciphertexts in LWE-Based Homomorphic Encryption," in *Public-Key Cryptography PKC 2013*, ser. Lecture Notes in Computer Science, K. Kurosawa and G. Hanaoka, Eds. Springer, 2013, pp. 1–13.

[29] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[30] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An Architecture for Computing on Encrypted Data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, pp. 1295–1309. [Online]. Available: https://dl.acm.org/doi/10.1145/3373376.3378523

[31] D. W. Archer, J. M. Caldern Trilla, J. Dagit, A. Malozemoff, Y. Polyakov, K. Rohloff, and G. Ryan, "RAMPARTS: A Programmer-Friendly System for Building Homomorphic Encryption Applications," in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. WAHC'19. ACM, 2019, pp. 57–68. [Online]. Available: http://doi.acm.org/10.1145/3338469.3358945

[32] A. Viand and H. Shafagh, "Marble: Making fully homomorphic encryption accessible to all," in *Proceedings of the 6th Workshop on Encrypted Computing amp; Applied Homomorphic Cryptography*, ser. WAHC '18. Association for Computing Machinery, 2018.

[33] E. Chielle, O. Mazonka, H. Gamil, N. G. Tsoutsos, and M. Maniatakos, "E3: A framework for compiling c++ programs with en-

crypted operands," Cryptology ePrint Archive, Report 2018/1013, 2018, https://ia.cr/2018/1013.

[34] E. Crockett, C. Peikert, and C. Sharp, "Alchemy: A language and compiler for homomorphic encryption made easy," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. Association for Computing Machinery, 2018.

[35] *Porcupine: A Synthesizing Compiler for Vectorized Homomorphic Encryption*. Association for Computing Machinery, 2021, p. 375389.

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

[37] H. Chen, R. Cammarota, F. Valencia, F. Regazzoni, and F. Koushanfar, "Ahec: End-to-end compiler framework for privacy-preserving machine learning acceleration," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020.

[38] A. Viand, P. Jattke, and A. Hithnawi, "Sok: Fully homomorphic encryption compilers," in *2021 2021 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, may 2021, pp. 1092–1108.

[39] "Cingulata," https://github.com/CEA-LIST/Cingulata, 2020.

[40] S. Carpov, P. Dubrulle, and R. Sirdey, "Armadillo: a compilation chain for privacy preserving applications," in *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, 2015, pp. 13–19.

[41] D. Lee, W. Lee, H. Oh, and K. Yi, "Optimizing homomorphic evaluation circuits by program synthesis and term rewriting," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 503–518. [Online]. Available: https://doi.org/10.1145/3385412.3385996

[42] "PALISADE Lattice Cryptography Library," https://palisade-crypto.org/, Oct. 2020.

[43] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: Fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[44] "nGraph Deep Learning Compiler," https://www.ngraph.ai, 2020.

[45] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 265–283.

[46] D. Williamson, "Dynamically scaled fixed point arithmetic," in *[1991] IEEE Pacific Rim Conference on Communications, Computers and Signal Processing Conference Proceedings*. IEEE, 1991, pp. 315–318.

[47] M. Gao, Q. Wang, and G. Qu, "Energy and error reduction using variable bit-width optimization on dynamic fixed point format," in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2019, pp. 152–157.

[48] T. Na and S. Mukhopadhyay, "Speeding up convolutional neural network training with dynamic precision scaling and flexible multiplier-accumulator," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, 2016, pp. 58–63.

[49] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," *arXiv preprint arXiv:1604.03168*, 2016.

[50] A. B. Kinsman and N. Nicolici, "Bit-Width Allocation for Hardware Accelerators for Scientific Computing Using SAT-Modulo Theory," vol. 29, no. 3, pp. 405–413. [Online]. Available: http://ieeexplore.ieee.org/document/5419231/

[51] S. Purini, V. Benara, Z. Choudhury, and U. Bondhugula, "Bitwidth customization in image processing pipelines using interval analysis and SMT solvers," in *Proceedings of the 29th International Conference on Compiler Construction*. ACM, pp. 167–178. [Online]. Available: https://dl.acm.org/doi/10.1145/3377555.3377899