ELASM: Error-Latency-Aware Scale Management for Fully Homomorphic Encryption

Yongwoo Lee[1], Seonyoung Cheon[1], Dongkwan Kim[1], Dongyoon Lee[2], and Hanjun Kim[1]

[1]Yonsei University
[2]Stony Brook University

Abstract

Thanks to its fixed-point arithmetic and SIMD-like vectorization, among fully homomorphic encryption (FHE) schemes that allow computation on encrypted data, RNS-CKKS is widely used for privacy-preserving machine learning services. Prior works have partly automated a daunting scale management task required for RNS-CKKS fixed-point arithmetic, yet none takes an output error into consideration, preventing users from exploring a better error-latency trade-off.

This work proposes a new *error- and latency-aware scale management (ELASM)* scheme for the RNS-CKKS FHE scheme. By actively controlling the scale of a ciphertext, one can effectively make the impact of noise on an error smaller because an error is a scaled noise introduced by an RNS-CKKS operation. ELASM explores different scale management plans that repurpose an `upscale` operation as an error reduction operation, estimates the output error and latency of each plan, and iteratively finds the best plan that minimizes the error-latency cost function. In addition, this work proposes a new *scale-to-noise ratio (SNR)* parameter and introduces fine-grained noise-aware waterlines (a minimum scale requirement) for different RNS-CKKS operations, opening a new opportunity to further improve an error-latency trade-off.

This work implements the proposed ideas in the ELASM compiler along with a new FHE language and type system that enforces the RNS-CKKS constraints including SNR-based noise-aware waterlines. For ten machine and deep learning benchmarks, ELASM finds the better error and latency trade-offs (lower Pareto curves) than the state-of-the-art solutions such as EVA and Hecate.

1 Introduction

Fully homomorphic encryption (FHE) [2] allows analytical functions to be applied to encrypted data and produces the same encrypted results as if the computations were performed without encryption. The homomorphic property of FHE enables users to offload heavy computation to a third party safely (*e.g.,* cloud service) and promises privacy-preserving machine learning (ML) services in highly regulated fields such as healthcare, financial, and insurance [4, 41, 42, 47]. For example, Microsoft and IBM have demonstrated their practice use cases in bioinformatics and finance [29, 50].

Among various FHE schemes [8, 10, 11, 14–16, 21–23, 30, 32–36], the state-of-the-art RNS-CKKS [15] scheme is widely used for ML applications thanks to two distinct features, supporting (1) *fixed-point arithmetic* by scaling fixed-point numbers as integers and (2) *SIMD-like vectorization* (also referred to as batching) for data parallelism. These features make RNS-CKKS well-suited to ML applications that require a large number of floating-point operations yet are inherently tolerant to some errors [53]. Thus, RNS-CKKS has become the main target of many FHE libraries (*e.g.,* SEAL [51], HElib [39], HEAAN [31]) and FHE compilers (*e.g.,* EVA [26], CHET [27], Hecate [45]).

However, developing a correct (recoverable), precise (smaller error), and efficient (less latency) privacy-preserving ML application with RNS-CKKS is intractably challenging due to the manual ciphertext scale management requirements in RNS-CKKS. In the fixed-point arithmetic of RNS-CKKS, each multiplication increases the scale of a result ciphertext, and the *scale overflow* leads to an unrecoverable result. On the other hand, since some RNS-CKKS operations introduce a scale-insensitive noise to a ciphertext, keeping the scale low may cause the *scale underflow* problem, rendering the relative error (the amount of noise over scale) in a ciphertext large. To avoid overflow and underflow by keeping the ciphertext scale between the maximum and minimum scales, developers should carefully insert a scale management operation such as `rescale` that reduces the scale of a ciphertext.

Besides correctness, developers should consider the impacts of scale management on error and latency. The result error accumulated from each RNS-CKKS operation affects the Quality of Service (QoS) of an application. For example, the prediction accuracy of ML applications drops if the resulting error becomes larger. Thus, developers should keep the result error lower to achieve better QoS. On the other hand, RNS-CKKS operations have different latencies depending on the rescaling levels of their operands. Thus, manual scale management becomes even harder because scale management operations (*e.g.,* `rescale`) at different locations have non-trivial cascading effects on the program latency and the noise introduced by the subsequent operations.

To address the problem, EVA [26] and Hecate [45] proposed automatic scale management schemes. However, there are two critical limitations that prevent users from exploring better (Pareto-optimal) error-latency trade-offs. First, they both do not take into account the effect of scale management on the resulting error. EVA adds `rescale` operations if the scale after rescaling remains higher than a minimum scale threshold called *waterline*, which is (blindly) fixed as the maximum scale of input ciphertexts without considering error and latency. Hecate proposes a new scale management operation called `downscale`, which eagerly reduces the scale of ciphertext to the fixed waterline and explores different scale management options, but the exploration only considers latency optimization. In sum, none considers an error during scale management.

Second, they both rely on a coarse-grained noise-oblivious waterline to control scale underflow. Each RNS-CKKS operation introduces a different amount of noise into a ciphertext. As a result, their noise-oblivious waterline (which is set based on the input scales) may be unnecessarily large for a low-noise operation, or too small for a high-noise operation. Though a very conservative waterline can prevent scale underflow, it may lead to a worse latency-error trade-off.

This work proposes three new solutions that enable users to explore better error and latency trade-offs. First, this work proposes the first error estimation model for RNS-CKKS that estimates the result error between plain and FHE computation results. Each RNS-CKKS operation introduces a certain noise depending on the operation type and the rescale level of its operands. Since the error is a scaled noise and propagated to the next operations, the error estimation model estimates the result error from operation noises, considering a ciphertext's scale and a cascaded impact along the data flow.

Second, this work presents a new *Error-Latency-Aware Scale Management (ELASM)* scheme that finds the best scale management plan that minimizes the user-defined cost function about the error and latency. First, ELASM generates various scale management plans by differently inserting scale management operations. Then, ELASM estimates the result error with the error estimation model and the latency by accumulating the latency of each RNS-CKKS operation. Based on the estimation results, ELASM calculates the user-defined cost function and finds the best plan with minimal cost. Unlike prior work, since the cost function includes not only latency but also error, ELASM may decide to *increase* the scale of a ciphertext if its estimated cost is profitable.

Third, this work proposes a new *scale-to-noise ratio* (*SNR*) parameter and introduces fine-grained noise-aware waterlines for different RNS-CKKS operations. *SNR* allows the user to specify the waterline management requirement such that the ratio between the scale m of a ciphertext and the noise n introduced by an operation must be greater than or equal to *SNR* (*i.e.,* $m/n \geq SNR$), akin to the traditional signal-to-noise ratio in signal processing. Given *SNR*, ELASM sets different waterlines for different noisy operations such as `rescale` and `rotate`, enabling a better error-latency trade-off.

This work implements the proposed ideas in the ELASM compiler along with a new noise-aware FHE intermediate representation (IR) and type system that reflect the RNS-CKKS constraints and the proposed error-proportional *SNR* parameter. The type system ensures that the scale management always satisfies the FHE operation and noise constraints during exploration.

This work evaluates the ELASM compiler with 10 benchmarks consisting of various machine and deep learning algorithms. ELASM shows a better error-latency trade-off compared to the existing works. When average error and latency values among explored results are used as constraints, ELASM shows 16.7% and 24.9% lower latency and 4.2 bits (18.1$\times$) and 5.7 bits (51.2$\times$) lower error than Hecate and EVA. Additionally, we evaluate the error-proportionality of error estimation and *SNR* parameter as the R^2 value of linear fitting with real error. The error proportionality of error estimation and *SNR* parameter is 0.948 and 0.986, respectively. Finally, the case study on an end-to-end application shows that ELASM and its high error proportionality allow users to control the error-latency trade-off with the *SNR* parameter.

The followings are the contributions of this work;

- the error-latency-aware scale management (ELASM) with a new error estimation model;
- the new error-proportional parameter SNR that allows a user to trade-off between error and latency;
- the new fine-grained noise-aware waterline management scheme based on the SNR parameter;
- the ELASM IR and type system with the formal semantics of the SNR parameter on RNS-CKKS; and
- the ELASM compiler framework that automates error-latency-aware scale management.

2 Background on RNS-CKKS

Since Gentry's breakthrough work in 2009 [32, 33], various FHE schemes have been proposed such as BGV/BFV [10, 30], CKKS [15, 16], and GSW [37]. This paper focuses on RNS-CKKS [15] which has shown to fit well to ML workloads [53], thanks to its efficient support for fixed-point operations and vectorization (data parallelism). While BGV/BFV support vectorization, they are mainly designed for integer operations. GSW is highly optimized for fast bootstrapping yet does not support vectorization.

Table 1: Time complexity [27] and noise [40] of RNS-CKKS operations. ELASM implicitly performs `relinearize` after multiplication (so multiplication adds noise in ELASM). N: polynomial modulus, l: level, σ: std. deviation.

RNS-CKKS Ops	Time Complexity	Noise
negate, add, mul	$O(N \cdot l)$	0
`modswitch`	$O(N \cdot l)$	0
`rotate`, `relinearize`	$O(N \log N \cdot l^2)$	$\frac{8\sqrt{3}}{3}\sigma l N + \frac{8\sqrt{2}}{3}N + \sqrt{3N}$
`rescale`	$O(N \log N \cdot l^2)$	$\frac{8\sqrt{2}}{3}N + \sqrt{3N}$

2.1 Encryption Parameters

RNS-CKKS encodes a vector of raw data into a plaintext cyclotomic polynomial [9] and encrypts the plaintext to a ciphertext using the Ring Learning with Errors (RLWE) [46]. RNS-CKKS ciphertext requires users to manually determine these two encryption parameters: *polynomial modulus N* and *coefficient modulus Q*.

The polynomial modulus N affects security and performance. The minimum value of N for a given Q is known for a certain security level [3]: *e.g.*, from $N = 2^{10}$ for $Q = 2^{27}$ to $N = 2^{15}$ for $Q = 2^{881}$ to achieve 128-bit security. Thus, N is often determined by Q in practice. For performance, smaller N (and equivalently smaller Q) is preferred as the latency of RNS-CKKS operations linearly or log-linearly depends on N [27] (See Table 1). On the other hand, coefficient modulus Q is related to correctness: it should be set large enough to avoid *scale overflow*, which we describe in the next section.

2.2 Scale Management

To efficiently support fixed-point arithmetic, RNS-CKKS encodes a real number as an integer with a *scale*, and encrypts the integer as a ciphertext while storing the scale as its property. For instance, the value $x = 1.23$ represents an integer $v = 123$ with the scale $m = 10^2$. The encryption with RLWE introduces some noise n to v, *i.e.*, $v = m \cdot x \pm n$. The problem is that each multiplication of two ciphertexts makes the resulting ciphertext value larger, as shown in Figure 1. When it overflows the encryption parameter Q, the result becomes unrecoverable. In other words, Q should be set large enough for correctness.

To avoid the overflow and to keep Q (and N) small for performance, RNS-CKKS supports a scale management operation `rescale` that reduces the scale of a ciphertext: *e.g.*, rescaling 1230 at scale 10^3 to 123 at scale 10^2. As Q should be strictly divisible by the rescaling factor, EVA [26] and Hecate [45] use a single rescaling factor, say R, and determine $Q \approx R^l$ as the power of the rescaling factor, where the exponent is referred to as *level l*. Starting from the initial (maximum) level L, each `rescale` consumes R and reduces the level l by one. Figure 1 illustrates that `rescale` reduces the scale $m1 \cdot m2$ to $m1 \cdot m2/R$ and the level l from 4 to 3.

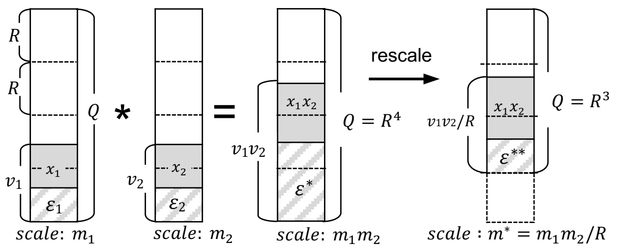


Figure 1: RNS-CKKS multiplication and rescaling. The real value x is represented as integer v at scale m. The scaled integer v includes some noise n. Simply put, $v = m \cdot x \pm n$. Multiplication (due to implicit `relinearize` operation) and `rescale` operation introduce additional noises, $n_{relinearize}$ and $n_{rescale}$, respectively. Thus, $n^* = m_1x_1n_2 \pm m_2x_2n_1 \pm n_1n_2 \pm n_{relinearize}$ and $n^{**} = n^*/R \pm n_{rescale}$ where n^* and n^{**} refer to the noise after multiplication and rescale, respectively. The expectation of sum of noise $\|n_1 \pm n_2\|$ is $(\|n_1\|^2 + \|n_2\|^2)^{1/2}$

2.3 Noise and Error

RNS-CKKS is constructed on RLWE [46] that adds and removes a small random noise to a ciphertext during encryption and decryption. Besides the initial noise added by encryption, three RNS-CKKS operations (1) `rescale`, (2) `rotate`, and (3) `relinearize` introduce additional noise into the resulting ciphertext [40]. Table 1 lists its variance.

(1) `rescale`: `rescale` divides the integer value, scale, and Q of ciphertext by the rescaling factor R and introduces the rounding noise $\|n_{rescale}\| = \frac{8\sqrt{2}}{3}N + \sqrt{3N}$ that depends only on N. In Figure 1, the noise after rescaling becomes $n^{**} = n^*/R + n_{rescale}$ where n^* is the noise of input ciphertext.

(2) `rotate`: RNS-CKKS supports vectorization (for data parallelism) in which a vector of $N/2$ values is encrypted as a single ciphertext. Besides vector addition and multiplication, it provides the `rotate` operation that rotates a vector by a given amount. The rotation requires a key-switching process that introduces a noise $\|n_{rotate}\| = \frac{8\sqrt{3}}{3}\sigma l N + \frac{8\sqrt{2}}{3}N + \sqrt{3N}$, depending on both N and level l.

(3) `relinearize`: A ciphertext multiplication in RNS-CKKS increases the number of polynomials in a ciphertext, which in turn increases the computation cost of FHE operations. RNS-CKKS supports the `relinearize` operation that reduces the number of polynomials inside a ciphertext. Relinearization requires a key-switching process like `rotate`, and thus introduces the same amount of noise. Existing RNS-CKKS compilers [26, 27, 45] including ours apply `relinearize` operation on each ciphertext multiplication. Thus, a ciphertext multiplication in effect is considered to introduce additional noise $n_{relinearize}$, as depicted in Figure 1.

The computation error ε is defined to be the difference between plain and FHE computation results. Note that RNS-CKKS operations add noises regardless of the scale (See Table 1). The amount of error ε is then determined by the

$$0 < l \quad (1) \qquad l \leq L \quad (2)$$

$$|v_i| \leq R^l \quad (3) \qquad R^L \leq f_{sec}(k) \quad (4)$$

$$l_1 = l_2 \text{ for } v_1 \oplus v_2 \quad (5) \qquad m_1 = m_2 \text{ for } v_1 + v_2 \quad (6)$$

$$m_w \leq m \quad (7) \qquad m_{op} = SNR * n_{op} \leq m \quad (8)$$

Figure 2: RNS-CKKS Constraints. m_i and l_i are the scale and the level of v_i. $\oplus$ represents binary operators $+$ and $\times$. This work proposes to use a new find-grained noise-aware waterline (8), instead of (7).

amount of noise over scale: *i.e.,* $\varepsilon = n/m$. For example, noise $n = 10$ introduces error 0.01 for scale $m = 10^3$, and error 0.001 if $m = 10^4$. This relation implies that for a given noise, a minimal scale requirement restricts the maximum error of an FHE operation.

2.4 RNS-CKKS Constraints

The encryption parameters and a scale management scheme, deciding where to place scale management operations (*e.g.,* `rescale`, `modswitch`, etc.) and which operations to use, should obey the constraints in Figure 2 for functional correctness and security guarantee.

Given the rescaling factor R, for a ciphertext with the scale m, the level l (with the initial maximum L), and the vectorization slots $k = N/2$: Equations 1 and 2 are natural minimum/maximum constraints on the level of a ciphertext: *e.g.,* the level l monotonically decreases from initial maximum L. Equation 3 prevents scale overflow: *i.e.,* the scaled value $|v|$ should be always smaller than or equal to the coefficient modulus $Q = R^l$ at level l. Equation 4 ensures x-bit security guarantee: *i.e.,* the minimum $N = 2k$ is determined by $Q = R^L$ at level L according to [3].

RNS-CKKS further requires two more constraints on the operands of binary operations. Equation 5 requires that the operands of multiplication and addition should be at the same level l (*i.e.,* the same $Q = R^l$). If the level does not match, developers should insert a `modswitch` operation that consumes R without affecting the scale and only decreases the level by one. Equation 6 enforces that the operands of addition should have the same scale. If not, developers should add an `upscale` operation, syntactic sugar for multiplying one with an arbitrary scale, to increase the scale of an operand ciphertext without changing the level.

Existing FHE compilers [26, 45] introduce the waterline constraint (Equation 7) that specifies the minimum required scale. In contrast, this work proposes a new minimum scale constraint (Equation 8) based on the noise n, which will be introduced in §4.

3 Motivation

This section discusses the existing scale management schemes [26, 45] and their limitations, motivating the need for a new error-aware scale management scheme. The scale management scheme used in EVA [26], called waterline rescaling, aims to keep the scale small. EVA keeps track of the scale growth and inserts the `rescale` operation if the scale after rescaling remains higher than the coarse-grained waterline m_w (See Equation 7 in Figure 2). The waterline is fixed as the maximum scale of input ciphertexts, and users can alter the input scale to adjust the waterline. Hecate [45] improves waterline rescaling with a new rescaling operation called `downscale` that can reduce the arbitrary amount of scale, but follows the same waterline constraint.

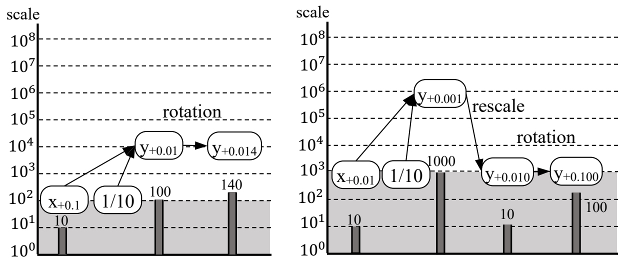
Figures 3a and 3b show how EVA (and Hecate also) works when computing $rotate(0.1x)$, a part of convolution, for given different input scales (waterlines) 10^2 and 10^3, and rescaling factor $R = 10^3$. For waterline 10^2 (Figure 3a), EVA does not add `rescale` as the scale after rescaling becomes smaller than the waterline: *i.e.,* $10^4/10^3 < 100$. On the other hand, for waterline 10^3 (Figure 3b), EVA inserts `rescale` to reduce the scale from 10^6 to 10^3 between multiplication and rotation.

There are two critical limitations in the existing scale management schemes:

Limitation 1: Existing solutions do not control errors during scale management. Increasing the waterline parameter does not necessarily lead to a low error. Let us illustrate the problem with the same $rotate(0.1x)$ example. Consider Figure 3a (input scale = 10^2) in which the bar graph represents noise and the subscript number represents error. Suppose the encryption adds an initial noise 10. As the input scale is 10^2, the error of the first ciphertext x is 0.1 (See the subscript ± 0.1 next to x.) Multiplication $y = x \cdot 1/10$ increases the scale of y to 10^4 ($= 10^2 \cdot 10^2$). Note that the new noise after multiplication is $m_1 x_1 n_2 \pm m_2 x_2 n_1 \pm n_1 n_2 \pm n_{relinearize}$ (See Figure 1). Here, $\|n_2\|$ and $\|n_{relinearize}\|$ are 0 because $1/10$ is plaintext. Thus, the noise becomes 100 ($= 10^2 \cdot 10 \cdot 1/10$) and the error is 0.01 ($= 100/10^4$). Finally, `rotate` introduces an additional noise $n_{rotate} = 100$, and the final noise and error become 140 ($= \sqrt{100^2 + 100^2}$) and 0.014 ($= 140/10^4$).

Figure 3b (input scale = 10^3) shows that a higher input scale unexpectedly leads to a higher error. After multiplication, the scale of y becomes 10^6 ($= 10^3 \cdot 10^3$), the noise grows to 1000 ($= 10^3 \cdot 10 \cdot 1/10$), and the error is 0.001 ($= 1000/10^6$). In this case, EVA applies to `rescale` that divides the noise by scale and adds $\|n_{rescale}\| = 10$. After rescaling, the scale is 10^3, the noise is 10 ($\approx \sqrt{(1000/10^3)^2 + 10^2}$) and the error is 0.01 ($= 10/10^3$). Finally, `rotate` increases the noise to 100 ($\approx \sqrt{100^2 + 10^2}$) after adding $\|n_{rotate}\| = 100$. The final error becomes 0.1 ($= 100/10^3$), which is higher than in Figure 3a.

Limitation 2: They neglect noise differences among operations and use a fixed noise-oblivious waterline. As discussed in §2.3, RNS-CKKS operations introduce a different



(a) Waterline=100 (b) Waterline=1000

Figure 3: Comparison of the scale management of EVA with different waterlines on the parts of convolution that compute $rotate(0.1x)$. A gray bar and a subscript represent the accumulated noise and error in a ciphertext, respectively. A rescale operation divides the scale and the encrypted value by 1000. The noises of rescale and rotation are 10 and 100, respectively.

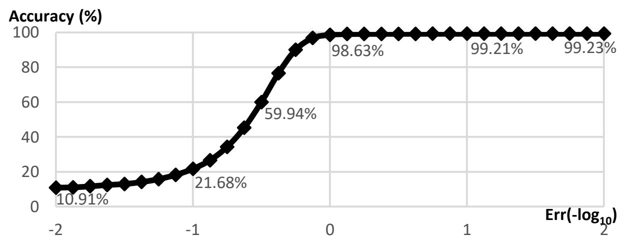


Figure 4: Inference accuracy of LeNet-5 for different errors.

amount of noise into the resulting ciphertext. For instance, the `rotate` operation adds the noise $\|n_{rotate}\| = 100$ in both cases of Figure 3 and thus increases the resulting error. Yet, EVA does not do anything special to tame those additional noises although increasing the scale can reduce the impact of noise on the error. Furthermore, the single fixed waterline unnecessarily restricts the scale of an operation that has little impact on the resulting error, preventing users from exploring better latency-error trade-offs.

Importance of error management. An excessive output error can have a negative impact on the quality of service (QoS), even for error-tolerant ML applications. Figure 4 shows the prediction accuracy (QoS) of LeNet-5 on MNIST dataset [28] when varying output errors. An unacceptable accuracy drop occurs for large errors ($-\log\|\varepsilon\| \leq -1$). For smaller errors ($-\log\|\varepsilon\| \geq -1$), the accuracy slowly increases as the output error decreases, opening an opportunity to take different error-latency trade-off. The lack of error-aware scale management and fine-grained waterline implies that existing solutions provide no guarantee on output errors, and thus may produce RNS-CKKS programs with an arbitrary amount of errors. For example, Figure 5 shows that EVA's compilation parameter (*i.e.,* waterline) fails to control the output error, yielding an arbitrary variation in the output error. It makes exploring the trade-off between latency and error non-trivial.

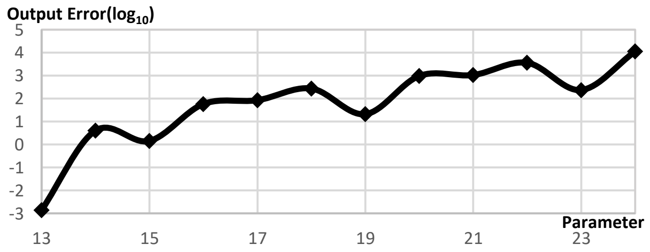


Figure 5: The input scale parameter of EVA leads to an arbitrary variation in the output error in LeNet-5.

4 Overview

This work proposes (1) the new error-latency-aware scale management (ELASM, §4.1) and (2) the new fine-grained noise-aware waterline management with the new error-proportional compile parameter called SNR (§4.2). Combined, they enlarge scale management space and offer improved error and latency trade-offs. Moreover, this work implements the proposed ideas in the ELASM compiler along with a new noise-aware FHE IR and type system (§4.3).

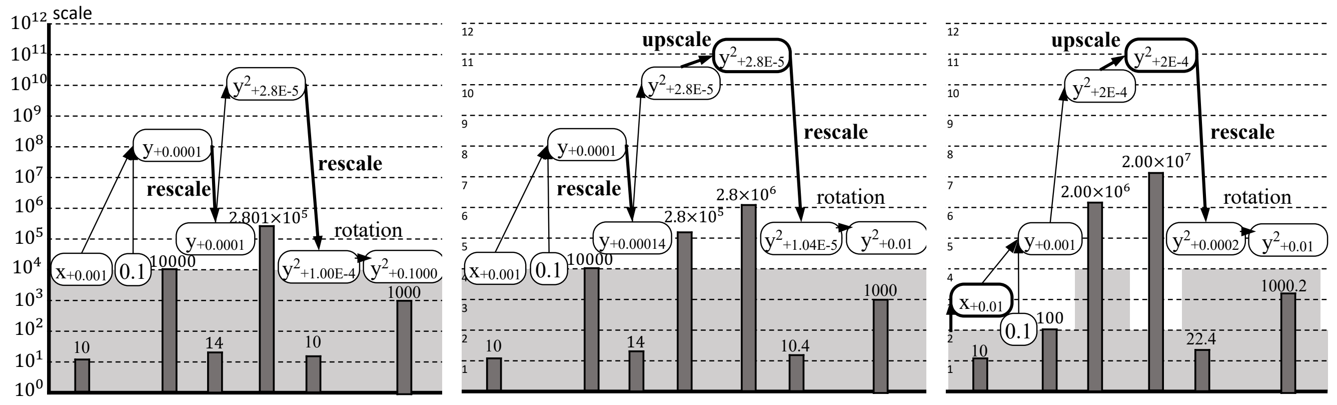
4.1 Error-Latency-Aware Scale Management

The error-latency-aware scale management (ELASM) is designed on the key observation that by increasing a scale, one can effectively make the impact of noise on error smaller (as the amount of error ε is determined by the amount of noise over scale: *i.e.,* $\varepsilon = n/m$). Unlike prior work which uses `upscale` operation (which increases the scale of a ciphertext) only to match the scales of addition operands, ELASM proposes to use `upscale` *as an error reduction operation* also.

Consider the new example in Figure 6 that computes $rotate(0.1x)^2$. Suppose the input scale (or waterline) is 10^4; the rescaling factor $R = 10^3$; and the noises of the `rescale`, `rotate`, and `relinearize` (ciphertext multiplication) operations are 10, 1000, and 1000, respectively. Figure 6a shows that existing work results in the error 0.1000.

Figure 6b illustrates that actively increasing a scale can improve the output error (0.01). By applying `upscale` on y^2, the active upscaling scheme can effectively reduce the impact of the noise introduced by `rotate`. Note that Figure 6b introduces the same number (2) of `rescale` operations as Figure 6a, leading to the same rescaling level. As the latency largely depends on the level l (See Table 1), this implies that two codes would show similar performance. As a result, Figure 6b can offer a better error-latency trade-off.

Based on the insight, ELASM explores various scale management plans that place `upscale` operations at different locations and use different scaling factors. In particular, ELASM employs Markov Chain Monte-Carlo (MCMC) sampling [38] to iteratively find a better plan and optimize a latency-error cost function. Users can define a custom cost function for



(a) Existing Work (EVA) (b) Active Upscaling (c) Noise-aware Waterline

Figure 6: Comparison of the scale management schemes of EVA and ELASM on the program that computes $rotate(0.1x)^2$. A gray bar (*e.g.,* 10 on the leftmost bar) and a subscript (*e.g.,* +0.001 next to the leftmost x) represent the accumulated noise and error in a ciphertext, respectively. Rescaling factor R is 10^3. The noises introduced by `rescale`, `rotate`, and `relinearize` (ciphertext multiplication) are 10, 1000, and 1000, respectively. For (a) and (b), the waterline is fixed as $m_w = 10^4$. For (c), the waterline is noise-aware and computed by $m_{op} = SNR \times \|n_{op}\|$ where SNR is set to be 10.

ELASM as needed with the estimated latency and error. To reduce the exploration overhead, ELASM estimates the error and latency of each scale management plan (instead of actually running the generated code and measuring the metrics, which may be slow). §5 discusses ELASM in depth.

4.2 Fine-grained Noise-aware Waterline

This work proposes a new *scale-to-noise ratio* (*SNR*) parameter and introduces fine-grained noise-aware waterlines. Similar to the traditional signal-to-noise ratio in signal processing, the *SNR* parameter allows users to specify the minimum ratio between the scale m of a ciphertext and the noise n introduced by an RNS-CKKS operation: *i.e.,* $SNR \leq m/n$. With SNR, the waterline m_w is no longer a fixed value set by the input scale. The waterline becomes a function of a noise: *i.e.,* $m_{op} = SNR \times \|n_{op}\| \leq m$ (Equation 8 in Figure 2).

Figures 6a and 6b illustrate the downside of using a fixed noise-oblivious waterline (set by the maximum input scales as in EVA). The resulting error is mostly dominated by a large-noise operation like `rotate`. As listed in Table 1, three RNS-CKKS operations introduce different amounts of noise, and the others do not. The single fixed waterline may sub-optimally limit the scales of the other operations that produce small or no noises and thus have little impact on the errors.

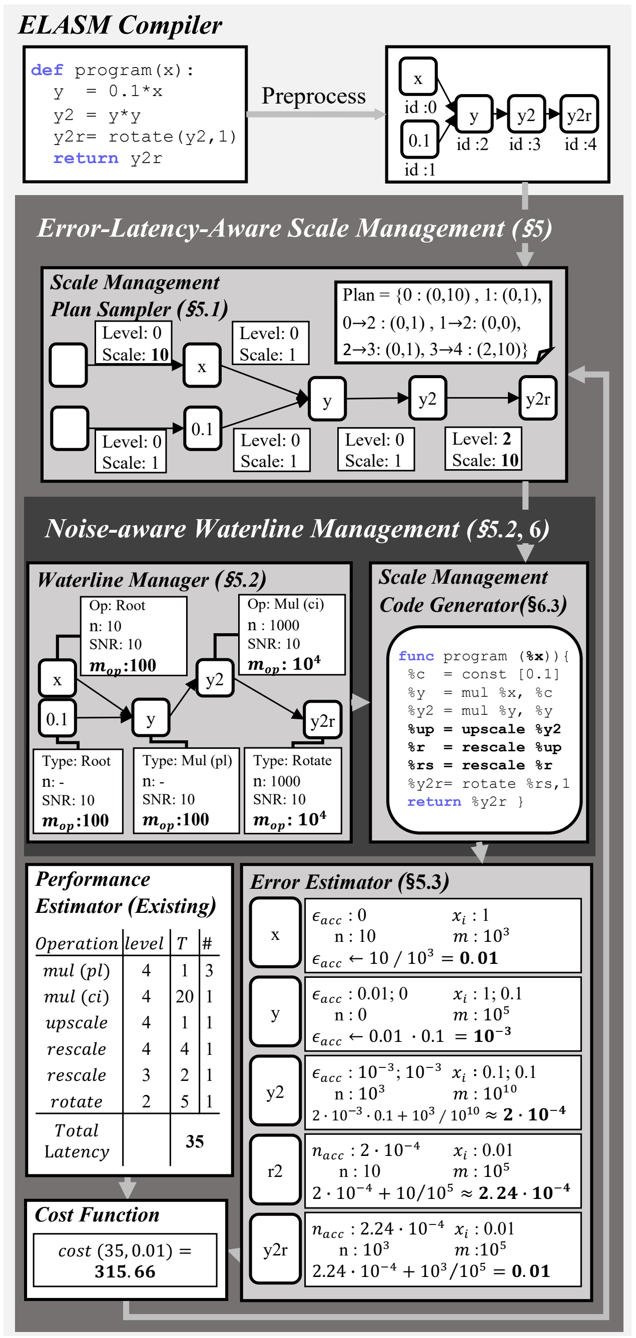
On the other hand, Figure 6c demonstrates that the noise-aware flexible waterline can enlarge the scale management space, allowing better performance without sacrificing the resulting error. Suppose in this example, *SNR* is 10; the noise of encryption and `rescale` is 10; and the noise of `rotate` and `relinearize` (ciphertext multiplication) is 1000. Then, the waterline for `rotate` and ciphertext multiplication becomes $10 \cdot 1000 = 10000$, and the waterline for all the rest operations is $10 \cdot 10 = 100$. The waterline for plaintext values is not defined by noise, but it is set to the minimum waterline for the other values. The waterline for `rotate` is also applied to its operand (unlike ciphertext multiplications) because the scales of the operand and result ciphertexts remain the same.

Figure 6c shows that the flexible waterline does not restrict the scale of the ciphertexts including x, 0.1, and y which have little impact on the scale. As a result, Figure 6c could introduce only one `rescale` operation (compared to two in the other cases (a) and (b)). The lower number of `rescale` implies that one can use a smaller coefficient modulus $Q = R^L$, leading to lower latency (See §2.1 and Table 1). Compared to Figure 6b, Figure 6c offers a better latency-error trade-off: *i.e.,* a lower latency at a similar error (0.01 vs. 0.01).

4.3 ELASM Compiler Design

This work proposes the ELASM compiler that supports ELASM and the fine-grained noise-aware waterline with its IR and type system. The ELASM IR and type system enforce that each ciphertext type embeds the scale and rescaling level information, and each RNS-CKKS operation respects its SNR-based waterlines (Equation 8).

Figure 7 shows the design of the ELASM compiler. The ELASM compiler translates an input FHE program written in Python into the program with the ELASM IR. The scale management plan sampler (§5.1) samples a set of optimization plan candidates, each of which specifies a different amount of level reduction (*e.g.,* using `rescale`) and scale increase (*e.g.,* using `upscale`). Given the SNR parameter, ELASM calculates noise-aware waterlines for each FHE operation as


ELASM Compiler
Preprocess
x
id :0
0.1
id :1
y
id :2
y2
id :3
y2r
id :4
Error-Latency-Aware Scale Management (§5)
Scale Management Plan Sampler (§5.1)
Plan = {0 : (0,10) , 1: (0,1), 0→2 : (0,1) , 1→2: (0,0), 2→3: (0,1), 3→4 : (2,10)}
Level: 0 Scale: 10
Level: 0 Scale: 1
Level: 0 Scale: 1
Level: 0 Scale: 1
Level: 0 Scale: 1
Level: 2 Scale: 10
Noise-aware Waterline Management (§5.2, 6)
Waterline Manager (§5.2)
Op: Root n: 10 SNR: 10 m_{op}:100
Op: Mul (ci) n : 1000 SNR: 10 m_{op}: 10^4
Type: Root n: - SNR: 10 m_{op}:100
Type: Mul (pl) n: - SNR: 10 m_{op}:100
Type: Rotate n: 1000 SNR: 10 m_{op}: 10^4
Scale Management Code Generator(§6.3)
Performance Estimator (Existing)
Cost Function
$cost\ (35, 0.01) =$ 315.66
Error Estimator (§5.3)
x
ϵ_{acc} : 0 x_i : 1 n : 10 m : 10^3 $\epsilon_{acc} \leftarrow 10\ /\ 10^3 = \mathbf{0.01}$
y
ϵ_{acc} : 0.01; 0 x_i : 1; 0.1 n : 0 m : 10^5 $\epsilon_{acc} \leftarrow 0.01 \cdot 0.1 = \mathbf{10^{-3}}$
y2
ϵ_{acc} : 10^{-3}; 10^{-3} x_i : 0.1; 0.1 n : 10^3 m : 10^{10} $2 \cdot 10^{-3} \cdot 0.1 + 10^3\ /\ 10^{10} \approx \mathbf{2 \cdot 10^{-4}}$
r2
n_{acc} : $2 \cdot 10^{-4}$ x_i : 0.01 n : 10 m : 10^5 $2 \cdot 10^{-4} + 10/10^5 \approx \mathbf{2.24 \cdot 10^{-4}}$
y2r
n_{acc} : $2.24 \cdot 10^{-4}$ x_i : 0.01 n : 10^3 m :10^5 $2.24 \cdot 10^{-4} + 10^3/10^5 = \mathbf{0.01}$

```
def program(x):
  y  = 0.1*x
  y2 = y*y
  y2r= rotate(y2,1)
  return y2r
```

```
func program (%x)){
 %c  = const [0.1]
 %y  = mul %x, %c
 %y2 = mul %y, %y
 %up = upscale %y2
 %r  = rescale %up
 %rs = rescale %r
 %y2r= rotate %rs,1
 return %y2r }
```

Operation	*level*	*T*	*#*
mul (pl)	4	1	3
mul (ci)	4	20	1
upscale	4	1	1
rescale	4	4	1
rescale	3	2	1
rotate	2	5	1
Total Latency		**35**	

Figure 7: Design of the ELASM compiler. The example code and scale management plan are the same program and plan in Figure 6c. m_{op} means waterline for each operation.

a part of ELASM type system (§6.2). For each scale management plan, the scale management code generator (§6.3) instruments scale management operations (*e.g.,* `rescale`, `modswitch`, `upscale`) to satisfy the RNS-CKKS constraints including noise-aware waterlines (Figure 2) and generates a legal RNS-CKKS program. For each generated program, ELASM estimates its error (§5.3) and latency, and then computes a user-defined error-latency cost function, which is iteratively fed back to the scale management plan sampler. Besides, ELASM employs the scale management group generation optimization, latency estimation, and backend code implementation of Hecate [45].

5 Error-Latency-Aware Scale Management

This section describes Error-Latency-Aware Scale Management (ELASM) which actively manages the scale of ciphertext with an `upscale` operation for error control. ELASM consists of a sampling of the scale management space (§5.1), a noise-aware waterline management and code generation (§5.2), and an error estimation that enables fast iteration of the exploration (§5.3).

5.1 Sampling of Scale Management Space

ELASM employs the Metropolis-Hastings algorithm [38], a widely used Markov-Chain Monte-Carlo (MCMC) sampling algorithm, to iteratively find a better error-latency trade-off option for a given SNR parameter. The Metropolis-Hastings algorithm starts from a plan P, and derives a new plan P^* from P. The algorithm then decides whether to accept the new plan or not, meaning that the next new plan is derived from P^* or P, respectively. The algorithm accepts the new plan when $U \leq \alpha(P^*|P)$, where $U \sim Uniform(0,1)$ and $\alpha(P^*|P) = min(1, cost(P)/cost(P^*))$. Hence, the cost function affects the error and latency of an optimized program. ELASM allows a user to design the cost function (*e.g.,* $cost(P) = T \cdot E$ for latency T and error E of plan P).

Each sample in ELASM represents a scale management plan that specifies the position of a scale management operation and the amount of scale increment and/or level decrement. Consider the $\texttt{rotate}(0.1x)^2$ program in Figure 7. A candidate position of scale management operations includes the edges between values (x, y), (0.1, y), (y, y^2), and (y^2, $\texttt{rotate}(y^2)$). In simple, the edge representation based on the group id is (0→2), (1→2), (2→3), (3→4). The initial edge (*null*, x) is also a target of a scale management plan, mimicking a change on the scale of an encrypt operation. Then, a scale management plan in ELASM can be represented as a map between an edge (as a key) and a level-scale-change pair (as a value). For instance, the plan in Figure 7 $\{(0:(0,10); 1:(0,1); 0 \rightarrow 2:(0,1); 1 \rightarrow 2:(0,1); 2 \rightarrow 3:(0,1); 3 \rightarrow 4:(2,10)\}$ stands for the plan that decreases the level by 2 and increases the scale by 10 between y^2 and $\texttt{rotate}(y^2)$ (*i.e.,* before `rotate`).

To propose a new plan, ELASM randomly selects the target positions, and changes the amount of level and scale.

5.2 Noise-aware Waterline Management

Inserting scale management operations as suggested by a scale management plan is not sufficient to generate a legal program

Table 2: The estimated value and error of each FHE operation. For operation names, the suffix 'c' means ciphertext and 'p' means plaintext: *e.g., mulcp* stands for a multiplication between ciphertext and plaintext. Unlisted operations preserve the error and value. An operation operates on an encoded integer $v = mx \pm n = m(x \pm \varepsilon)$ with scale m, value x, noise n, and error ε. $*$ implies an estimated value. The addition of errors means root-sum-square ($\|\varepsilon_0 \pm \varepsilon_1\| = (\|\varepsilon_0\|^2 \pm \|\varepsilon_1\|^2)^{1/2}$).

Operation	Est. Value	Estimated Error
mulcc(v_1,v_2)	$x_1^* x_2^*$	$\varepsilon_1^* x_2^* \pm \varepsilon_2^* x_1^* \pm \varepsilon_1^* \varepsilon_2^* \pm n_{relinearize}/m$
mulcp(v_1,v_2)	$x_1^* x_2$	$\varepsilon_1^* x_2$
addcc(v_1,v_2)	1	$\varepsilon_1^* \pm \varepsilon_2^*$
addcp(v_1,v_2)	1	ε_1^*
rotate(v)	x^*	$\varepsilon^* \pm n_{rotate}/m$
rescale(v)	x^*	$\varepsilon^* \pm n_{rescale}/(m/R)$
downscale(v)	x^*	$\varepsilon^* \pm n_{rescale}/m_w$
upscale(v)	x^*	ε^*

$$
\begin{array}{rcl}
Prg & ::= & \overline{F} \\
F & ::= & \texttt{func}\, \mathit{fid}\,(\overline{v:T})\,\{s;\overline{h}\} \\
S & ::= & \varepsilon \,|\, v := h \,|\, S;S \\
h & ::= & c \,|\, v \,|\, h+h \,|\, h \times h \,|\, -h \,|\, \texttt{rotate}(h,i) \;|\texttt{rescale}(h) \\
 & & |\texttt{modswitch}(h)|\texttt{upscale}(h,m)|\texttt{downscale}(h) \\
T & ::= & \texttt{re}\,|\,\texttt{ci}\,(m,d)\,|\; \texttt{pl}\,(m,d)\; |\overline{T} \to \overline{T}
\end{array}
$$

v : variable id, fid : function id, $c \in$ constants
$i, l \in \mathbb{Z}^+, m \in \mathbb{R}^+$

Figure 8: The formal syntax of the ELASM IR. The syntax with a gray box shows the scale management operations which is not used by a programmer. $\overline{A}$ means a list of A.

that obeys all the RNS-CKKS constraints in Figure 2. The ELASM compiler uses a type system (which will be discussed in §6.2) to enforce the RNS-CKKS constraints including the proposed SNR-based noise-aware waterlines. This work also defines a set of rewriting rules (§6.3) to generate an RNS-CKKS constraint-compliant program.

For example, in Figure 7 where *SNR* is set to 10, ELASM computes waterlines for each ciphertext based on the operation type. For y^2 and $rotate(y^2)$, the waterlines are set to $10 \times 10^3 = 10^4$. The waterlines for the others are set to $10 \times 10 = 100$. Then, ELASM inserts scale management operations `upscale` and `rescale`. The generated program is guaranteed to meet all the RNS-CKKS constraints based on the ELASM's type system. (§6.2)

5.3 Error Estimation

Given candidate programs, ELASM statically estimates their error and latency to compare the cost instead of dynamically running and measuring the actual metrics, which is very costly. ELASM estimates a latency by simply accumulating the time latency of each RNS-CKKS operation like previous work [45]. As listed in Table 1, the time complexity of an RNS-CKKS operation depends on polynomial modulus N and the current level l, which can be computed once a program is given.

On the other hand, estimating an absolute amount of error is challenging because reasoning about an error of multiplication between ciphertexts requires an unencrypted value, which is not available. As illustrated in Figure 1, given two ciphertexts $v_1 = m_1(x_1 \pm \varepsilon_1)$ and $v_2 = m_2(x_2 \pm \varepsilon_2)$ with scale m, value x, and error ε, the resulting error after multiplication becomes $\varepsilon_1 x_2 \pm \varepsilon_2 x_1 \pm \varepsilon_1 \varepsilon_2 \pm n_{relinearize}/m$, which includes the value-related terms like $\varepsilon_1 x_2$.

To address the problem, ELASM proposes a simple value estimation-based approach that is sufficient to compare two different candidate programs, but may not be able to estimate the absolute amount of an error. Table 2 summarizes ELASM's value and error estimation formulas for each FHE operation. Using the formulas, ELASM estimates values and errors by propagating the estimation results along their data flow. For most operations, the error estimation does not require an estimated value. On the other hand, for *mulcc*, ELASM uses the estimated value to estimate an error. Since estimating precise values is hard, this work postulates that feeding the same estimated value at the same program point across different candidate programs would be sufficient to reason about the difference in errors. Based on the idea, ELASM estimates the resulting value of an addition to be 1, the multiplicative identity that simplifies the error estimation on *mulcc* and *mulcp*. The proposed approach attempts to precisely track the impacts of multiplications, while giving up the precision for additions. Resetting values to 1 also implies that ELASM in effect analyzes FHE programs in a piece-wise manner. Recall that the idea here is to enable comparison, not to estimate a precise value. We later show that the proposed error estimation is proportional to the actual resulting error in §7.2.

For example, in Figure 7 that computes `rotate`$((0.1 * x)^2)$, ELASM estimates (assumes) the initial value of x is 1. Note that the plaintext 0.1 does not need estimation. Then, ELASM estimates y at 0.1 and y^2 at 0.01, and propagates the same estimated value to the `rescale` and `rotate` operations. Along the program code, based on these estimated values, ELASM computes the noises (and equivalently errors) using the method in Table 2, yielding the final error 0.01 (Note that example error calculation uses simple addition instead of root-sum-square for simplicity). If the scale management plan is changed, the error will be also changed. Given two different plans, the proposed error estimation is sufficient enough to compare the impacts of scale management operations at different places.

6 ELASM IR and Type System

This section describes the ELASM IR (§6.1), the ELASM type system that enforces RNS-CKKS constraints including the proposed noise-aware waterline (§6.2), and the ELASM code generation component based on rewriting rules (§6.3). Discussions on the formal operational semantics can be found in Appendix A.3.

6.1 ELASM IR

Figure 8 shows the formal syntax of the ELASM IR (intermediate representation), which the ELASM compiler generates and uses during optimization. The syntax with the gray box represents the scale management operations that do not appear in the input program. The scale management algorithm inserts the operations.

An ELASM program *Prg* consists of a list of functions $\overline{F}$ which can be called by an external driver. A function is composed of a list of statements S that ends with a return expression. For a function body, ELASM only defines a list of assignment statements $v := e$. An RNS-CKKS program does not include branches such as if-else and for-loop [17].

ELASM requires specifying the type T for each function argument v. Type T of a variable is a real vector (`re`), plain (`pl`), or cipher (`ci`). The `re` type represents a raw message that is not encrypted. The `pl` type represents an integer vector that contains the encoded data for a raw message. The `ci` type represents a ciphertext with scale m, and depth d properties. The depth d represents the number of `rescale`, `downscale` and `modswitch` operations applied to the ciphertext. The depth is simply another way to reason about the rescaling level l discussed in §2.2, which decreases from the initial level L upon `rescale`, `downscale` and `modswitch` operations: *i.e.,* $d = L - l$. Because the initial level L is selected by Equation 3, the exact L is unknown before compilation. Thus, the ELASM compiler uses depth d (increasing from 0) instead of level l (decreasing from L).

An FHE expression h could be constant, variable, binary expressions, negation and rotation. Binary expressions only contain addition and multiplication. Subtraction is implemented with negation, and division is implemented by multiplying the inverse of the divisor. The `rotate` (h,i) operation only takes a ciphertext h as its operand, and shifts vectored data in the ciphertext by an offset i.

ELASM does not ask programmers to use low-level scale management operations such as `rescale`, `modswitch`, `upscale`, and `downscale`. The `rescale` (h) operation decreases the scale by the predefined rescaling factor R, and increases the depth by one. The `modswitch` (h) operation does not alter the scale but increases the depth by one. The `upscale` (h,m) operation, syntactic sugar for multiplying one with an arbitrary scale, increases the scale of h by m but does not change the level. The `downscale` (h) operation, syntactic sugar for applying `upscale` and `rescale` in sequence, reduces the scale to a waterline. Formal operational semantics of the ELASM IR can be found in Appendix A.3.

$$\frac{\Gamma \vdash h_1 : \texttt{ci}(m,d) \quad \Gamma \vdash h_2 : \texttt{ci}(m',d) \quad mm' \geq m_{relinearize}}{\Gamma \vdash h_1 \times h_2 : \texttt{ci}(mm',d)} \quad (\text{Mul}_{CC})$$

$$\frac{\Gamma \vdash h : \texttt{ci}(m,d) \quad m \geq m_{rotation}}{\Gamma \vdash \texttt{rotate}(h,l) : \texttt{ci}(m,d)} \quad (\text{Rot})$$

$$\frac{\Gamma \vdash h : \texttt{ci}(m,d) \quad m_{rescale} \leq m \leq m_{rescale} \cdot R}{\Gamma \vdash \texttt{downscale}(h) : \texttt{ci}(m_{rescale}, d+1)} \quad (\text{DS})$$

$$\frac{\Gamma \vdash h : \texttt{ci}(m,d) \quad \frac{m}{R} \geq m_{rescale}}{\Gamma \vdash \texttt{rescale}(h) : \texttt{ci}(\frac{m}{R}, d+1)} \quad (\text{RS})$$

Figure 9: Parts of new typing rules of the ELASM IR. $m_{rescale}$ means the minimal scale required by a rescale operation in Equation 8 and $m_{rotation}$ means the minimal scale required by a rotate operation.

$$\frac{\Gamma \vdash h_1 : \texttt{ci}(m,d) \quad \Gamma \vdash h_2 : \texttt{re}}{h_1 \times h_2 \xrightarrow{rewrite} h_1 \times \texttt{upscale}(h_2, m_{rescale})} \quad (\text{EncodeMul})$$

$$\frac{\Gamma \vdash e : \texttt{ci}(m,d) \quad m < m_{rotation}}{\texttt{rotate}(e,i) \xrightarrow{rewrite} \texttt{rotate}(\texttt{upscale}(e, m_{rotation}/m), i)} \quad (\text{URot})$$

Figure 10: Parts of rewriting rules for scale management code generation. $m_{rescale}$ means the minimal scale required by a rescale operation by Equation 8, and $m_{rotation}$ means the minimal scale required by a rotate operation.

6.2 Type System for Waterline Management

Figure 9 shows a subset of the ELASM typing rules that presents the waterline constraints. The full typing rules can be found in Appendix A.1. The type system is designed to satisfy/enforce the RNS-CKKS constraints in Figure 2 including the SNR-based noise-aware waterlines described in §4.2. The type soundness of ELASM IR guarantees that a well-typed program does not violate the RNS-CKKS constraints.

The ELASM type system reflects the SNR constraint. Notably, four rules such as Equations Mul_{CC}, Rot, DS and RS require some minimal scales (waterlines): $m_{relinearize}$ for ciphertext multiplication, $m_{rotation}$ for `rotate`, and $m_{rescale}$ for `rescale` and `downscale`. The $m_{relinearize}$ waterline is easily satisfied as a ciphertext multiplication increases the scale by itself. On the other hand, to satisfy the SNR-based noise-aware waterline constraint in Equation 8, the waterlines $m_{rotation}$ and $m_{rescale}$ should be defined as $\|n_{rotate}\| \cdot SNR$ and $\|n_{rescale}\| \cdot SNR$ for a given SNR. From Table 1, $\|n_{rotate}\|$ is $\frac{8\sqrt{3}}{3}\sigma lN + \frac{8\sqrt{2}}{3}N + \sqrt{3N}$ and $\|n_{rescale}\|$ is $\frac{8\sqrt{2}}{3}N + \sqrt{3N}$. Because $\|n_{rotate}\|$ is affected by the level of ciphertext that is unknown before the scale management, ELASM uses the worst-case value.

6.3 Scale Management Code Generation

Given a program that is potentially illegal without all the necessary scale management operations, ELASM generates a *well-typed program for a given SNR parameter* that meets all the RNS-CKKS constraints. For this purpose, ELASM defines a set of code rewriting rules that add scale management operations such as `downscale`, `upscale` and `rescale`.

ELASM introduces the new rewriting rules (partly on Figure 10) that rewrite the expression to meet the required conditions of the typing rules (Figure 9). The full rewriting rules for ELASM can be found in Figure 17. Notably, Equation URot, the newly proposed rule for rotation, inserts `upscale` if the operand's scale is less than the noise-aware waterline for rotation ($m_{rotation}$) to satisfy the typing rule Equation Rot in Figure 9. Equation EncodeMul also inserts `upscale` for the casting, but the encoding scale is different from the rotation. For the multiplication, the encoding scale is the same as the waterline of `rescale` $m_{rescale}$, because the scale of all input data for a program is set to $m_{rescale}$.

After the scale management code generation is finished and the generated code is selected for the optimal program, the program is translated to LLVM IR which calls the FHE library functions. We use Microsoft SEAL [51] which implements the RNS-CKKS scheme as a backend.

7 Evaluation

This work compares ELASM with the state-of-the-art FHE compilers such as EVA [26] and Hecate [45] to evaluate the benefits of the proposed error-latency aware scale management and the noise-aware waterlines. For the benchmarks, we implemented and tested the seven machine learning and deep learning applications listed in Figure 11k. The benchmark sets are the same as those used in Hecate except for newly added multivariate regression (MR) for epoch 2 and 3. The benchmark sets also include the benchmark sets of EVA except SqueezeNet. The image processing benchmarks (SF, HCD) 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 MNIST dataset. This work uses the gradient descent algorithm for the regression benchmarks with 2 and 3 epochs. Note that the regression benchmarks perform training workload that calculates the corresponding function and deep learning benchmarks perform inference workload. The benchmarks assume a packed ciphertext with 16384 slots.

This evaluation uses Microsoft SEAL [51] (Release 3.5.9) for RNS-CKKS backend library and runs experiments on Intel(R) Core(TM) i7-8700 @ 3.20GHz with 64GB RAM. All of the results are the measured values from the real execution, not from the estimation. For ELASM, EVA, and Hecate, the same RNS-CKKS settings are used. The rescaling factor $R = 2^{60}$ and polynomial modulus $N = 2^{15}$. This work sets the security level as 128-bit for all the experiments. ELASM explores 12000 scale management plan samples with 12 parallel threads. ELASM samples the level decrement $l_{dec} \sim U_{[0,2]}$ and scale increment $m_{inc} = ReLU(X)$ for $X \sim U_{[-10,10]}$ where $U_{[a,b]}$ is a uniform distribution over $[a,b]$. The number of the newly sampled position for a new plan is $\sqrt{\text{\# of candidate positions}}$. We use $\sqrt{T} \cdot (60 + \log_2 E)$ as the cost function of ELASM. The cost function adjusts the effect of compilation parameter (SNR) on latency T(quadratic) and result error E(inversely exponential). Therefore, we apply square root to T and logarithm to E and then multiply them. Note that we add 60 to $\log_2 E$ to make the value always positive. In addition, the maximum compilation time of ELASM is 312 seconds from LeNet-5 and the others are smaller than 15 seconds on parallel implementation with 12 threads.

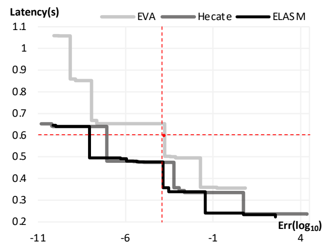
7.1 Pareto Curve of Error-Latency Trade-off

Figure 11 shows Pareto-optimal error and latency trade-off options for all the benchmarks. This work uses the compilation parameter (the waterline in EVA and Hecate, and the waterline of `rescale` derived from SNR in ELASM) from 2^{15} to 2^{50}. SF, MR E2 (epoch 2), MR E3 (epoch 3), and MLP clearly show that their Pareto curves are shifted to the left, which means the error for a given latency is reduced. LR E2, LR E3, PR E2 and PR E3 show that their Pareto curves are shifted to the left-and-downward, which means that ELASM improves both the latency and error at the same time. For the other benchmarks (HCD and Lenet), the shapes of curves are not similar enough to compare how their curves are shifted, but the Pareto curves show better performance and error in general. EVA and Hecate finds a slightly better performance and error trade-off point in MR E2 and Lenet, respectively. ELASM expects better error for its scale management plan but the imprecise error estimation results in an inoptimal scale management plan. To quantify the improvement of the pareto curve of error-latency trade-off, we plot the best error and latency for a given constraint shown in red lines in Figure 11. Here, as the constraint points, we selected average error and latency values among explored results.

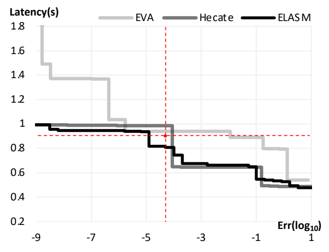
Figure 12 shows the error and latency comparison among EVA, Hecate, and ELASM. ELASM focuses on both error and latency, so the improvement can be observed in both dimensions. Figure 12a shows that given the same latency for each application, on average, ELASM has 5.7 bits (51.2$\times$) and 4.2 bits (18.1$\times$) smaller errors than EVA and Hecate, respectively. Unlike the prior works, ELASM actively manages the scale of ciphertext to control the error. On the other hand, Figure 12b shows that at the same error for each application, ELASM is on average 24.9% and 16.7% faster than EVA and Hecate, respectively. The scale management search space of ELASM is larger than that of Hecate thanks to the proposed noise-aware waterlines. ELASM successfully finds a better scale management plan with lower latency than Hecate.

7.2 Error Estimation

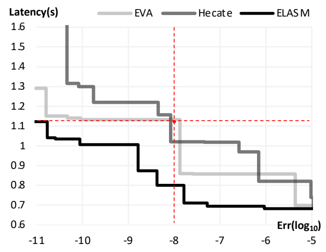
This work evaluates the precision of the error estimation used in ELASM. As discussed in §5.3, ELASM does not aim to



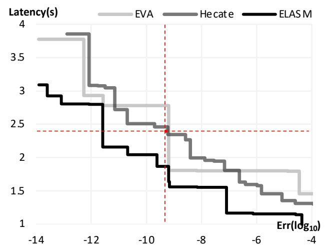
(a) SF



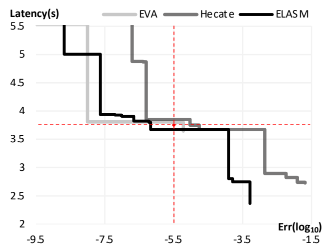
(b) HCD



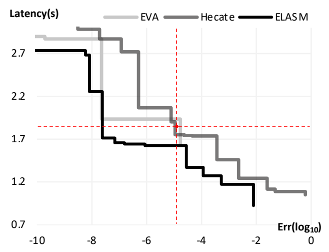
(c) LR E2



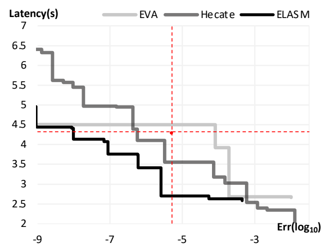
(d) PR E2



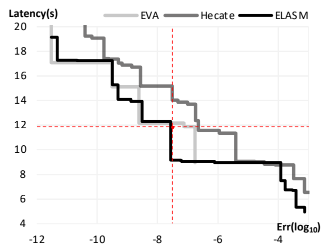
(e) MR E2



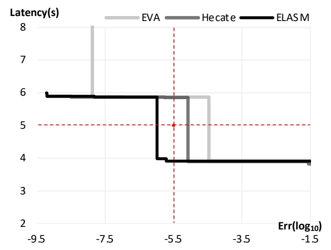
(f) LR E3



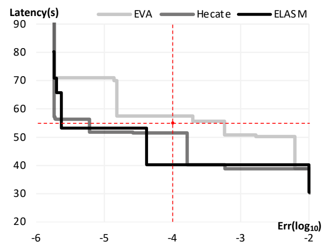
(g) PR E3



(h) MR E3



(i) MLP



(j) Lenet

Name	Description
Sobel filter (SF)	A classic edge detection algorithm
Harris corner detection(HCD)	A classic corner detection algorithm
Linear regression (LR)	Fitting a linear equation
Polynomial regression (PR)	Fitting a quadratic equation
Multivariate regression (MR)	Fitting a multiple linear equation
Multi-layer perceptron (MLP)	A feed-forward neural network for image classification with 784×100 and 100×10 layer and square activation
LeNet-5 (Lenet)	A convolutional neural network for image classification presented in [43] with square activation

(k) Benchmark program description

Figure 11: Pareto-frontier of error-latency trade-offs (a-j) of the tested FHE benchmark programs (k). E2 and E3 stands for two and three epoch of the gradient descent algorithm in regression benchmarks. The red line shows the representative values among the trade-off options. We select the representative values from the average on the usable ranges of error and latency.

estimate the absolute amount of errors for a given program. Instead, the error estimation method of ELASM strives to produce an error-proportional estimation on which the cost functions of different scale management plans can be computed and compared. Thus, the precision metric on the error estimation becomes how much the estimated errors are proportional to measured errors.

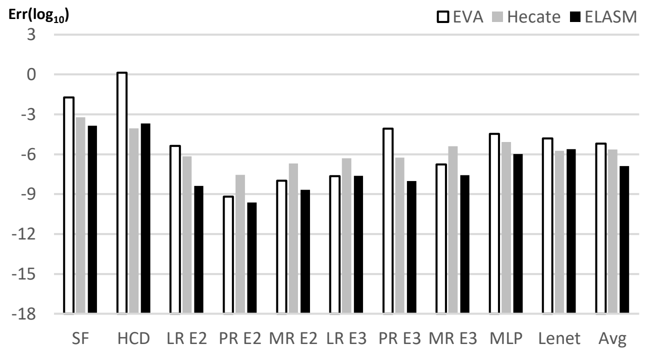
Figure 13 shows the R^2 value of the linear fitting between the estimated and measured errors. ELASM's error estimation method shows a high R^2 value of 0.948 on average, implying that the proposed error estimation is proportional enough to be used to compare different scale management plans. Lenet shows the least R^2 value of 0.884 because the simple value estimation is not precise, and the delta between the assumed values and the exact values becomes amplified as a result of multiple stages of multiplication and addition.

7.3 Error-proportionality of *SNR* parameter

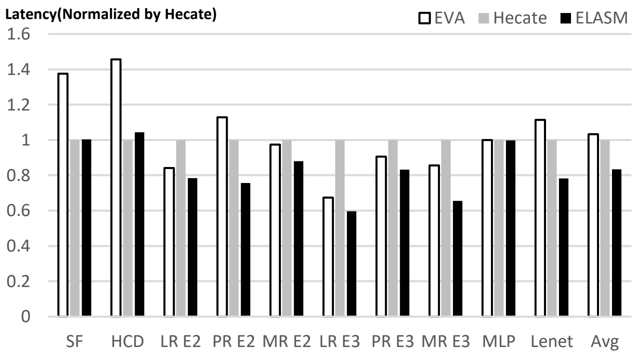
This section evaluates how much the compiler parameter used in EVA, Hecate, and ELASM is proportional to the resulting error. An error-proportional compiler parameter enables users easily to explore different error-latency trade-offs. Given one parameter and corresponding error-latency along the Pareto-curve, users can reason about how the error-latency trade-off would change when increasing or decreasing the parameter. Therefore, users do not need to explore all the parameters.

Figure 14 shows R^2 of linear fitting between the parameter and output error that represents the relation between the error and the compile parameter for each application. Because EVA, Hecate, and ELASM use their parameters with different semantics, we place the parameters on the same x-axis if their upper bound of the operation-wise error is the same.

Overall, ELASM shows much better error-proportionality than EVA which appears to be ineffective in controlling the errors. EVA shows a very low proportionality in PR E3 because its inefficient scale management and the large multiplication



(a) Best error for a given latency (Lower is better)



(b) Best latency for a given error (Lower is better)

Figure 12: Error and latency for a given constraint. The constraints are plotted on Figure 11.

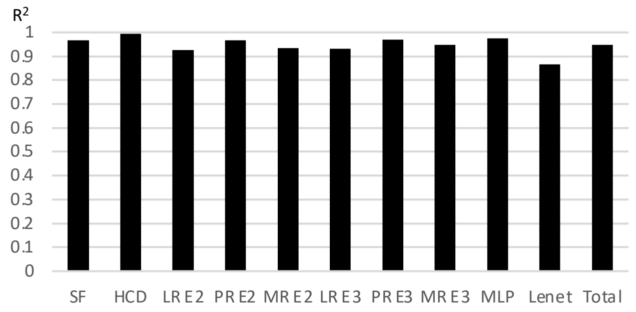


Figure 13: R^2 of linear fitting between estimated error and measured error

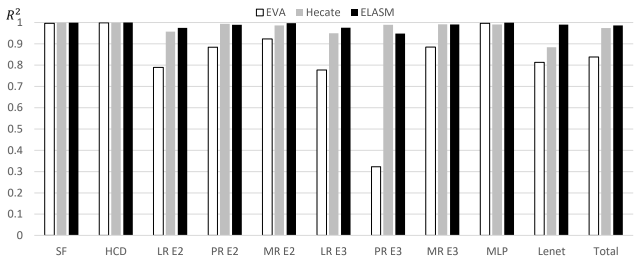


Figure 14: R^2 of linear fitting between parameter and error

depth of PR E3 requires large coefficient moduli for some parameter values that are infeasible.

Interestingly, in PR E3, EVA shows (not significant yet) higher R^2 value than ELASM. Note that ELASM aims to optimize the cost function that incorporates both the latency and the error. As a result, ELASM may choose a scale management plan in which an error is higher but the error-latency cost function is smaller, leading to some minor fluctuation.

Nonetheless, Figure 14 shows that ELASM's R^2 values are consistently high across all the applications with the minimum R^2 of 0.948 as well as the mean R^2 values of 0.986.

7.4 Case Study: End-to-end DNN Application

We prototype the end-to-end image classification inference scenario with LeNet-5. The client encrypts the image from MNIST evaluation dataset and sends it to the server. The server processes the encrypted image with LeNet-5 and sends it back to the client. Finally, the client decrypts the image and checks the correctness. The network bandwidth between the client and server is 1Gbps. The client and server have the same configuration of evaluation setup.

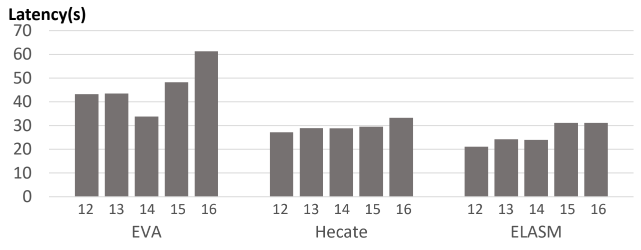
Figure 15a shows the end-to-end latency results. EVA does not consider the latency during scale management, so there does not exist a consistent correlation between its parameter and latency. On the other hand, Hecate and ELASM reflect the latency estimation in scale management and show monotonic relation between their parameters and latency.

Figure 15b shows the inference accuracy of the existing works and ELASM. The figure shows the existence of threshold that significantly increases the accuracy, supported by Figure 4. Notably, as Hecate aggressively optimizes for latency while ignoring error, it leads to an unpredictable accuracy drop on waterline 2^{15}. On the other hand, ELASM applies the error estimation to scale management and shows monotonic relation between parameter and accuracy.

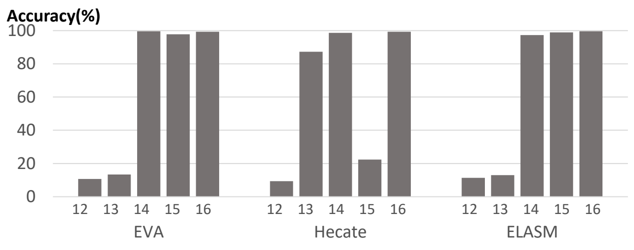
In overall, EVA and Hecate cannot support efficient exploration between accuracy and latency. On the other hand, ELASM shows monotonic relation both on accuracy and latency, showing the effectiveness of the error-proportional parameter. Furthermore, for the similar error level, ELASM shows the smallest end-to-end latency. As a result, a user can easily explore the most efficient accuracy-latency trade-off with ELASM.

7.5 Discussion: Larger Applications

The benchmark suite does not include larger applications like ResNet-20, because SEAL library does not support bootstrap operation which allows for restoring the ciphertext level.



(a) Latency for a given parameter (Lower is better)



(b) Accuracy for a given parameter (Higher is better)

Figure 15: End-to-end inference case study with LeNet-5. Latency includes network and encryption/decryption latency. Accuracy is calculated with a subset of MNIST evaluation dataset.

ELASM can support the larger application with another FHE library that supports the bootstrap operation. When other libraries that support bootstrapping are used, one can partition the program across bootstrapping, apply ELASM's performance and error-aware scale management ideas to subprograms, and recombine them.

8 Related Work

To improve the performance and programmability of HE applications, prior works [5–7, 13, 18, 24–27, 52, 54] proposed various languages and optimizing compilers for HE. In [53], the survey of FHE compilers and libraries provides an extensive survey and experimental evaluation. Typically, the existing compilers and languages automate encryption parameter selection or conceal the complexity of the HE scheme behind a high-level language. In addition, the compilers present various optimization techniques for HE applications.

General-purpose HE compilers: Several works [5, 12, 18, 20, 24–26, 44, 54] propose new programming languages or the implementation of general-purpose HE applications for existing programming languages.

The HE compilers [5, 12, 18, 20, 25, 44, 54] for existing programming languages ease the implementation of FHE applications. Cingulata [12, 20], E^3 [18] and Marble [54] provide open source compiler and runtime supports for running C++ programs with encrypted data. Lobster [44] is an optimizing compiler for Cingulata and it performs term rewriting and program synthesis on boolean circuits generated by Cingulata to optimize the program. RAMPARTS [5] provides an environment for developing HE applications in Julia, and automatically converts an imperative program into an optimized computation circuit for HE by using PALISADE [49] library. ALCHEMY [25] supports the Haskell front-end and automatically selects the appropriate parameters. In addition, Porcupine [24] proposes HE DSL named Quill for data layout optimization in FHE using program synthesis. However, the compilers only support non-CKKS schemes [10, 19, 30] instead of the CKKS and RNS-CKKS schemes which can support fixed-point arithmetic.

Recently proposed compilers [26, 45] support the CKKS and RNS-CKKS schemes. Encrypted Vector Arithmetic (EVA) [26] introduces a new language for FHE computation, which is designed to be an intermediate representation of other domain-specific languages. EVA supports arithmetic operations on fixed-width vectors and facilitates encrypted SIMD computations. Moreover, the EVA compiler automatically manages scales of fixed point ciphertexts. Assuming that the program latency is proportional to the output scale, its scale management scheme optimizes the output scales. On the other hand, Hecate [45] addresses that the minimal scale does not minimize latency, and introduces a new scale management space exploration that optimizes latency of the program. With a proactive rescaling algorithm, Hecate directly optimizes latency with performance estimation.

This work discovers two limitations in EVA and Hecate in terms of scale management: an absence of error-aware scale management and a single-fixed fixed-waterline without considering FHE operation noises. With a new error- and latency-aware scale management scheme and a new error-proportional parameter, ELASM framework provides better error and latency results than EVA and Hecate.

Domain-specific HE compilers: Other works [6, 7, 13, 27] support CKKS schemes, but their applications are limited to specific domains like DNN inference

CHET [27] is an optimized compiler for encrypted DNN inference. CHET provides a domain-specific language and transforms an input tensor circuit into a sequence of FHE operations. CHET also automates parameter selection. Moreover, CHET provides layout selection and other HE-specific optimizations to improve the latency of the HE program.

nGraph-HE [6, 7] extends Intel's nGraph [48], an existing deep learning graph compiler, to enable encrypted deep learning. The compiler implements the HE backend for nGraph to deploy neural network models with popular deep learning frameworks like TensorFlow [1]. The compilers apply various HE-specific optimizations including data layout optimization. Furthermore, nGraph-HE provides *lazy rescaling* and it inserts `rescale` operation only after linear layers.

AHEC [13] supports nGraph and Tensorflow as front-end and SEAL (RNS-CKKS) as back-end with supporting multiple hardware back-end through GPU-accelerated HE library. AHEC enables the automated generation and optimization of HE kernels through vectorization, tiling, and tensor layout selection. To exploit parallelism, AHEC provides Tile DSL that describes the HE kernel and hardware abstraction layer.

ELASM's scale management scheme can be applied to existing domain-specific compilers, so the compiler can use ELASM's error-aware scale management to further improve the performance of the target application.

9 Conclusion

This work proposes a new error- and latency-aware scale management (ELASM) for RNS-CKKS fully homomorphic encryption. ELASM explores different scale management plans, efficiently estimates the output error and latency of each plan, and iteratively finds the best plan with the least estimated error and latency. ELASM introduces a new error-proportional parameter called scale-to-noise ratio (SNR) and supports the new fine-grained noise-aware waterlines, thus enlarging scale management exploration space. The proposed ideas are implemented in the ELASM compiler along with the noise-aware ELASM IR and type system. This work evaluates ELASM with ten machine learning and deep learning benchmarks and demonstrates that ELASM offers better latency and error trade-offs than the state-of-the-art RNS-CKKS compilers such as EVA and Hecate.

Acknowledgments

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, IITP-2021-0-00853 and IITP-2022-11-2032 funded by the Ministry of Science and ICT. This work is in part supported by the National Science Foundation under Grant No. CCF-2153747 and CNS-2135157. This work is also supported by CryptoLab. (Corresponding author: Hanjun Kim)

Availability

The source code of ELASM compiler will be publicly available on `https://github.com/corelab-src/elasm`.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA, November 2016. USENIX Association.

[2] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Comput. Surv.*, 51(4), July 2018.

[3] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[4] David Archer, Lily Chen, Jung Hee Cheon, Ran Gilad-Bachrach, Roger A Hallman, Zhicong Huang, Xiaoqian Jiang, Ranjit Kumaresan, Bradley A Malin, Heidi Sofia, et al. Applications of homomorphic encryption. *HomomorphicEncryption. org, Redmond WA, Tech. Rep.*, 2017.

[5] David W. Archer, José Manuel Calderón Trilla, Jason Dagit, Alex Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard Ryan. RAMPARTS: A Programmer-Friendly System for Building Homomorphic Encryption Applications. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. ACM, 2019.

[6] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir 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*. ACM, 2019.

[7] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. nGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*. ACM, 2019.

[8] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Annual Cryptology Conference*. Springer, 2012.

[9] Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed Ciphertexts in LWE-Based Homomorphic Encryption. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *Public-Key Cryptography - PKC 2013*. Springer, 2013.

[10] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, New York, NY, USA, 2012. ACM.

[11] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on Computing*, 43(2), 2014.

[12] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. Armadillo: a compilation chain for privacy preserving applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, 2015.

[13] Huili Chen, Rosario Cammarota, Felipe Valencia, Francesco Regazzoni, and Farinaz Koushanfar. Ahec: End-to-end compiler framework for privacy-preserving machine learning acceleration. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020.

[14] Jung Hee Cheon, Jean-Sébastien Coron, Jinsu Kim, Moon Sung Lee, Tancrede Lepoint, Mehdi Tibouchi, and Aaram Yun. Batch fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2013.

[15] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full rns variant of approximate homomorphic encryption. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography – SAC 2018*, Cham, 2018. Springer.

[16] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, Cham, 2017. Springer.

[17] Jung Hee Cheon, Dongwoo Kim, Duhyeong Kim, Hun Hee Lee, and Keewoo Lee. Numerical method for comparison on homomorphically encrypted numbers. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology – ASIACRYPT 2019*, Cham, 2019. Springer.

[18] Eduardo Chielle, Oleg Mazonka, Homer Gamil, Nektarios Georgios Tsoutsos, and Michail Maniatakos. E3: A framework for compiling c++ programs with encrypted operands. Cryptology ePrint Archive, Report 2018/1013, 2018. https://ia.cr/2018/1013.

[19] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1), 2020.

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

[21] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Scale-invariant fully homomorphic encryption over the integers. In *International Workshop on Public Key Cryptography*. Springer, 2014.

[22] Jean-Sébastien Coron, Avradip Mandal, David Naccache, and Mehdi Tibouchi. Fully homomorphic encryption over the integers with shorter public keys. In *Advances in Cryptology – CRYPTO 2011*, volume 6841, 08 2011.

[23] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. In *Advances in Cryptology – EUROCRYPT 2012*, 04 2012.

[24] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T. Lee, and Brandon Reagen. Porcupine: A synthesizing compiler for vectorized homomorphic encryption. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, 2021.

[25] Eric Crockett, Chris Peikert, and Chad Sharp. Alchemy: A language and compiler for homomorphic encryption made easy. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018.

[26] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan 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*. ACM, 2020.

[27] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd 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*. ACM, 2019.

[28] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6), 2012.

[29] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Manual for using homomorphic encryption for bioinformatics. Technical Report MSR-TR-2015-87, November 2015.

[30] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. https://eprint.iacr.org/2012/144.

[31] FullRNS-HEAAN. https://github.com/KyoohyungHan/FullRNS-HEAAN, 2018.

[32] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford, CA, USA, 2009.

[33] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, New York, NY, USA, 2009. ACM.

[34] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. volume 6632, 05 2011.

[35] Craig Gentry, Shai Halevi, and Nigel P Smart. Better bootstrapping in fully homomorphic encryption. In *International Workshop on Public Key Cryptography*. Springer, 2012.

[36] Craig Gentry, Shai Halevi, and Nigel P Smart. Fully homomorphic encryption with polylog overhead. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2012.

[37] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Annual Cryptology Conference*. Springer, 2013.

[38] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. 57(1).

[39] HElib Open-Source HE Library. https://github.com/homenc/HElib, 2020.

[40] Andrey Kim, Antonis Papadimitriou, and Yuriy Polyakov. Approximate homomorphic encryption with reduced approximation error. *IACR Cryptol. ePrint Arch.*, 2020, 2020.

[41] Övünç Kocabaş and Tolga Soyata. Medical data analytics in the cloud using homomorphic encryption. In *E-Health and Telemedicine: Concepts, Methodologies, Tools, and Applications*. IGI Global, 2016.

[42] Ovunc Kocabas, Tolga Soyata, Jean-Philippe Couderc, Mehmet Aktas, Jean Xia, and Michael Huang. Assessment of cloud-based health monitoring using homomorphic encryption. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013.

[43] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 1998.

[44] DongKwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun 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*, New York, NY, USA, 2020. ACM.

[45] Yongwoo Lee, Seonyeong Heo, Seonyoung Cheon, Shinnung Jeong, Changsu Kim, Eunkyung Kim, Dongyoon Lee, and Hanjung Kim. HECATE: Performance-Aware Scale Optimization for Homomoprhic Encryption Compiler. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2022.

[46] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, Berlin, Heidelberg, 2010. Springer.

[47] Oliver Masters, Hamish Hunt, Enrico Steffinlongo, Jack Crawford, Flavio Bergamaschi, Maria Eugenia Dela Rosa, Caio Cesar Quini, Camila T Alves, Fernanda de Souza, and Deise Goncalves Ferreira. Towards a homomorphic machine learning big data pipeline for the financial services sector. *IACR Cryptol. ePrint Arch.*, 2019, 2019.

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

[49] PALISADE Lattice Cryptography Library. https://palisade-crypto.org/, October 2020.

[50] Jim Salter. Ibm completes successful field trials on fully homomorphic encryption. https://arstechnica.com/gadgets/2020/07/ibm-completes-successful-field-trials-on-fully-homomorphic-encryption/, July 2020.

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

[52] Tim van Elsloo, Giorgio Patrini, and Hamish Ivey-Law. Sealion: a framework for neural network inference on encrypted data, 2019.

[53] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. Sok: Fully homomorphic encryption compilers. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, may 2021.

[54] Alexander Viand and Hossein Shafagh. Marble: Making fully homomorphic encryption accessible to all. In *Proceedings of the 6th Workshop on Encrypted Computing; Applied Homomorphic Cryptography*. ACM, 2018.

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash v := e : \Gamma, v : T}\ \text{(Asn)} \qquad \frac{\Gamma \vdash s : \Gamma' \quad \Gamma' \vdash s' : \Gamma''}{\Gamma \vdash s; s' : \Gamma''}\ \text{(Stm)} \qquad \frac{\Gamma, \overline{v : T} \vdash s : \Gamma' \quad \overline{T} \in \{\mathtt{re}, \mathtt{ci}(m,0)\} \quad \overline{\Gamma' \vdash e : U}}{\Gamma \vdash \mathtt{func}\, \mathit{fid}\, (\overline{v : T})\, \{s; \overline{e}\} : \overline{T} \to \overline{U}}\ \text{(Fun)}$$

$$\frac{}{\Gamma \vdash c : \mathtt{re}}\ \text{(Const)} \qquad \frac{\Gamma \vdash h : \mathtt{re}}{\Gamma \vdash -h : \mathtt{re}}\ (\text{Neg}_R) \qquad \frac{\Gamma \vdash h : \mathtt{ci}(m,d)}{\Gamma \vdash -h : \mathtt{ci}(m,d)}\ (\text{Neg}_C) \qquad \frac{\Gamma \vdash h_1 : \mathtt{re} \quad \Gamma \vdash h_2 : \mathtt{re}}{\Gamma \vdash h_1 \oplus h_2 : \mathtt{re}}\ (\text{Bin}_R)$$

$$\frac{\Gamma \vdash h_1 : \mathtt{ci}(m,d) \quad \Gamma \vdash h_2 : \mathtt{sc}(m,d)}{\Gamma \vdash h_1 + h_2 : \mathtt{ci}(m,d)}\ \text{(Add)} \qquad \frac{\Gamma \vdash h_1 : \mathtt{ci}(m,d) \quad \Gamma \vdash h_2 : \mathtt{pl}(m',d)}{\Gamma \vdash h_1 \times h_2 : \mathtt{ci}(mm',d)}\ (\text{Mul}_{CP})$$

$$\frac{\Gamma \vdash h_1 : \mathtt{ci}(m,d) \quad \Gamma \vdash h_2 : \mathtt{ci}(m',d) \quad mm' \geq m_{relinearize}}{\Gamma \vdash h_1 \times h_2 : \mathtt{ci}(mm',d)}\ (\text{Mul}_{CC}) \qquad \frac{\Gamma \vdash h : \mathtt{ci}(m,d) \quad m \geq m_{rotation}}{\Gamma \vdash \mathtt{rotate}(h,l) : \mathtt{ci}(m,d)}\ \text{(Rot)}$$

$$\frac{\Gamma \vdash h : \mathtt{ci}(m,d) \quad m_{rescale} \leq m \leq m_{rescale} \cdot R}{\Gamma \vdash \mathtt{downscale}(h) : \mathtt{ci}(m_{rescale}, d+1)}\ \text{(DS)} \qquad \frac{\Gamma \vdash h : \mathtt{ci}(m,d) \quad \frac{m}{R} \geq m_{rescale}}{\Gamma \vdash \mathtt{rescale}(h) : \mathtt{ci}(\frac{m}{R}, d+1)}\ \text{(RS)}$$

$$\frac{\Gamma \vdash h : \mathtt{sc}(m,d)}{\Gamma \vdash \mathtt{modswitch}(h) : \mathtt{sc}(m,d+1)}\ \text{(MS)} \qquad \frac{\Gamma \vdash h : \mathtt{re}}{\Gamma \vdash \mathtt{upscale}(h,m) : \mathtt{pl}(m,0)}\ (\text{US}_R) \qquad \frac{\Gamma \vdash h : \mathtt{ci}(m,d) \quad m' \geq m}{\Gamma \vdash \mathtt{upscale}(h,m') : \mathtt{ci}(m',d)}\ (\text{US}_C)$$

Figure 16: Typing rules of the ELASM IR. $m_{rescale}$ means the minimal scale required by a rescale operation in Equation 8 and $m_{rotation}$ means the minimal scale required by a rotate operation. `sc` includes `ci` and `pl` types, and $\oplus$ includes $+$, $\times$.

Appendix A ELASM IR and Type System

A.1 Type Systems of ELASM

Figure 16 shows the typing rules of the ELASM IR. The type system is designed to satisfy/enforce the RNS-CKKS constraints in Figure 2 including the SNR-based noise-aware waterlines described in §4.2. The type soundness of ELASM IR guarantees that a well-typed program does not violate the RNS-CKKS constraints.

The ELASM type system reflects the SNR constraint. Notably, three rules such as Equation Mul_{CC}, Equation Rot, and Equation RS require some minimal scales (waterlines): $m_{relinearize}$ for ciphertext multiplication, $m_{rotation}$ for `rotate`, and $m_{rescale}$ for `rescale`. The first $m_{relinearize}$ waterline is easily satisfied as a ciphertext multiplication increases the scale by itself. On the other hand, to satisfy the SNR-based noise-aware waterline constraint in Equation 8, the waterlines $m_{rotation}$ and $m_{rescale}$ should be defined as $n_{rotate} \cdot SNR$ and $n_{rescale} \cdot SNR$ for a given SNR and the noises of `rotate` and `rescale` operations, respectively. Because n_{rotate} is affected by the level of ciphertext that is unknown before the rescaling, ELASM uses the worst-case value.

A.2 Rewriting Rules of ELASM

The rewriting rules (Figure 10) rewrite the expression to meet the required conditions of the typing rules (Figure 9). Equation DScale inserts `downscale` for both operands when it is better than multiplication and then rescale. Equation DMatch and Equation LMatch insert `downscale` and `modswitch` to match the level of operands of binary operations for satisfying typing rules Equations Add to Mul_{CC}, respectively. Equation SMatch inserts `upscale` to match the scale of operands of binary operations for satisfying a typing rule Equation Add. Equation EncodeAdd inserts `upscale` to cast `re` type operand to `pl` type value, because Equation Add does not allow `re` type operand. Moreover, the encoding scale is set to the scale of another operand. Equation EncodeMul also inserts `upscale` for the casting, but the encoding scale is different from the addition. For the multiplication, the encoding scale is the same as the waterline of `rescale` $m_{rescale}$, because the scale of all input data for a program is set to $m_{rescale}$. Equation Rescale defines the position of `rescale`.

Notably, Equation URot, the newly proposed rule for rotation, inserts `upscale` if the operand's scale is less than the noise-aware waterline for rotation $m_{rotation}$, which is computed by $n_{rotate} \cdot SNR$, given the SNR parameter, to satisfy the typing rule Equation RS in Figure 9.

After the scale management code generation is finished and the generated code is selected for the optimal program, the program is translated to LLVM IR which calls the FHE library functions. We use Microsoft SEAL [51] which implements the RNS-CKKS scheme as a backend.

A.3 Operational Semantics of ELASM

The runtime systems of ELASM defines big-step operational semantics

HE semantics function $\mathcal{H}$ represents the semantics of HE expressions h listed in Figure 8.

$$\frac{}{\langle v := h, s \rangle \mapsto s[v \mapsto \mathcal{H}[\![h]\!]s]} \tag{9}$$

$$\frac{\langle S_1, s \rangle \mapsto s'}{\langle S_1; S_2, s \rangle \mapsto \langle S_2, s' \rangle} \tag{10}$$

$$\frac{\langle S, s \rangle \mapsto s' \quad \mathcal{H}[\![\overline{h}]\!]s' = \overline{o}}{\langle S; \overline{h}, s \rangle \mapsto halt(\overline{o})} \tag{11}$$

Operand space: A constant $c \in Vector$ represents the constant vector of floating point numbers and a variable $x \in Var$ represents a variable name which can store an operand of

$$\frac{\Gamma \vdash h : \mathtt{sc}(m,d) \quad \Gamma \vdash h' : \mathtt{sc}(m',d) \quad m \cdot m' < {m_{rescale}}^2 \cdot R}{h \times h' \xrightarrow{rewrite} \mathtt{downscale}(h) \times \mathtt{downscale}(h')} \text{(DScale)} \qquad \frac{\Gamma \vdash h : \mathtt{sc}(m,d) \quad m \geq Rm_{rescale}}{h \xrightarrow{rewrite} \mathtt{rescale}(h)} \text{(Rescale)}$$

$$\frac{\Gamma \vdash e : \mathtt{sc}(m,d) \quad \Gamma \vdash e' : \mathtt{sc}(m',d') \quad m > m_{rescale} \quad d < d'}{e \oplus e' \xrightarrow{rewrite} \mathtt{downscale}(e) \oplus e'} \text{(DMatch)} \qquad \frac{\Gamma \vdash h : \mathtt{ci}(m,d) \quad \Gamma \vdash h' : \mathtt{re}}{h + h' \xrightarrow{rewrite} h + \mathtt{upscale}(h',m)} \text{(EncodeAdd)}$$

$$\frac{\Gamma \vdash e : \mathtt{sc}(m,d) \quad \Gamma \vdash e' : \mathtt{sc}(m',d') \quad m = m_{rescale} \quad d < d'}{e \oplus e' \xrightarrow{rewrite} \mathtt{modswitch}(e) \oplus e'} \text{(LMatch)} \qquad \frac{\Gamma \vdash h : \mathtt{ci}(m,d) \quad \Gamma \vdash h' : \mathtt{re}}{h \times h' \xrightarrow{rewrite} h \times \mathtt{upscale}(h',m_{rescale})} \text{(EncodeMul)}$$

$$\frac{\Gamma \vdash h : \mathtt{sc}(m,d) \quad \Gamma \vdash h' : \mathtt{sc}(m',d) \quad m < m'}{h + h' \xrightarrow{rewrite} \mathtt{upscale}(h,m'/m) + h'} \text{(SMatch)} \qquad \frac{\Gamma \vdash e : \mathtt{ci}(m,d) \quad m < m_{rotation}}{\mathtt{rotate}(e,i) \xrightarrow{rewrite} \mathtt{rotate}(\mathtt{upscale}(e,m_{rotation}/m),i)} \text{(URot)}$$

Figure 17: Rewriting rules for scale management code generator. $m_{rescale}$ means the minimal scale required by a rescale operation by Equation 8, and $m_{rotation}$ means the minimal scale required by a rotate operation. `sc` includes `ci` and `pl` type.

Table 3: The operational semantics of HE operations. Case represents the abbreviation of each operation. Semantics describes how the HE semantics function $\mathcal{H}$ maps an HE expression h and a state s to operand space O. Condition restricts the application of the semantics function $\mathcal{H}$ to satisfy the interface of an HE library.

Case	Semantics	Condition
	$*$	$m \geq 1, 0 < l < L, \lvert v_i \rvert \leq R^l, m/n \geq SNR$
const	$\mathcal{H}\llbracket c \rrbracket s = \mathbb{R}^k\llbracket c \rrbracket$	$c \in Vector$
var	$\mathcal{H}\llbracket x \rrbracket s = o$	$x \in Var, sx = o \in O$
encode	$\mathcal{H}\llbracket encode(h,m) \rrbracket s = (\mathcal{P}\llbracket mv \rrbracket, m, L)$	$\mathcal{H}\llbracket h \rrbracket s = \mathbb{R}^k\llbracket v \rrbracket, m \geq 1$
encrypt	$\mathcal{H}\llbracket encrypt(h) \rrbracket s = (\mathcal{C}\llbracket v + n_{rescale} \rrbracket, m, l)$	$\mathcal{H}\llbracket h \rrbracket s = (\mathcal{P}\llbracket v \rrbracket, m, l)$
negate	$\mathcal{H}\llbracket -h \rrbracket s = (\mathcal{C}\llbracket -v \rrbracket, m, l)$	$\mathcal{H}\llbracket h \rrbracket s = (\mathcal{C}\llbracket v \rrbracket, m, l)$
addcp	$\mathcal{H}\llbracket h_1 + h_2 \rrbracket s = (\mathcal{C}\llbracket v_1 + v_2 \rrbracket, m, l)$	$\mathcal{H}\llbracket h_1 \rrbracket s = (\mathcal{C}\llbracket v_1 \rrbracket, m, l), \mathcal{H}\llbracket h_2 \rrbracket s = (\mathcal{P}\llbracket v_2 \rrbracket, m, l)$
addcc	$\mathcal{H}\llbracket h_1 + h_2 \rrbracket s = (\mathcal{C}\llbracket v_1 + v_2 \rrbracket, m, l)$	$\mathcal{H}\llbracket h_1 \rrbracket s = (\mathcal{C}\llbracket v_1 \rrbracket, m, l), \mathcal{H}\llbracket h_2 \rrbracket s = (\mathcal{C}\llbracket v_2 \rrbracket, m, l)$
mulcp	$\mathcal{H}\llbracket h_1 \times h_2 \rrbracket s = (\mathcal{C}\llbracket v_1 v_2 \rrbracket, m_1 m_2, l)$	$\mathcal{H}\llbracket h_1 \rrbracket s = (\mathcal{C}\llbracket v_1 \rrbracket, m_1, l), \mathcal{H}\llbracket h_2 \rrbracket s = (\mathcal{P}\llbracket v_2 \rrbracket, m_2, l)$
mulcc	$\mathcal{H}\llbracket h_1 \times h_2 \rrbracket s = (\mathcal{C}\llbracket v_1 v_2 + n_{relinearize} \rrbracket, m_1 m_2, l)$	$\mathcal{H}\llbracket h_1 \rrbracket s = (\mathcal{C}\llbracket v_1 \rrbracket, m_1, l), \mathcal{H}\llbracket h_2 \rrbracket s = (\mathcal{C}\llbracket v_2 \rrbracket, m_2, l)$
rotate	$\mathcal{H}\llbracket \mathtt{rotate}(h,i) \rrbracket s = (\mathcal{C}\llbracket v' + n_{rotate} \rrbracket, m, l)$	$\mathcal{H}\llbracket h \rrbracket s = (\mathcal{C}\llbracket v \rrbracket, m, l), v'_j = v_{(i+j)\%k}, 1 \leq j \leq k$
rescale	$\mathcal{H}\llbracket \mathtt{rescale}(h) \rrbracket s = (\mathcal{C}\llbracket v + n_{rescale} \rrbracket, m/R, l-1)$	$\mathcal{H}\llbracket h \rrbracket s = (\mathcal{C}\llbracket v \rrbracket, m, l)$
downscale	$\mathcal{H}\llbracket \mathtt{downscale}(h,m') \rrbracket s = (\mathcal{C}\llbracket v + n_{rescale} \rrbracket, m, l-1)$	$\mathcal{H}\llbracket h \rrbracket s = (\mathcal{C}\llbracket v \rrbracket, m, l), m/R \leq m' \leq m$
modswitchp	$\mathcal{H}\llbracket \mathtt{modswitch}(h) \rrbracket s = (\mathcal{P}\llbracket v \rrbracket, m, l-1)$	$\mathcal{H}\llbracket h \rrbracket s = (\mathcal{P}\llbracket v \rrbracket, m, l)$
modswitchc	$\mathcal{H}\llbracket \mathtt{modswitch}(h) \rrbracket s = (\mathcal{C}\llbracket v \rrbracket, m, l-1)$	$\mathcal{H}\llbracket h \rrbracket s = (\mathcal{C}\llbracket v \rrbracket, m, l)$
upscaler	$\mathcal{H}\llbracket \mathtt{upscale}(h,m) \rrbracket s = \mathcal{H}\llbracket encode(h,m) \rrbracket s$	$\mathcal{H}\llbracket h \rrbracket s = \mathbb{R}^k\llbracket v \rrbracket, m \geq 1$
upscalec	$\mathcal{H}\llbracket \mathtt{upscale}(h,m) \rrbracket s = \mathcal{H}\llbracket ho \rrbracket s$	$\mathcal{H}\llbracket h \rrbracket s = (\mathcal{C}\llbracket v \rrbracket, m', l), o = (\mathcal{P}\llbracket [1] \rrbracket, m, l), m \geq 1$

HE operations. An operand o is a element of operand space O which is defined by $O : \mathbb{R}^k \cup \mathcal{P} \times \mathbb{R} \times \mathbb{Z}^+ \cup \mathcal{C} \times \mathbb{R} \times \mathbb{Z}^+$. The elements of $\mathbb{R}^n$ is a real value vectors where n means the number of slots in a packed ciphertext. $\mathcal{P}$ and $\mathcal{C}$ mean the plaintext and ciphertext space defined by encryption parameters, respectively. Note that plaintext and ciphertext need scale $m \in \mathbb{R}$ and level $l \in \mathbb{Z}^+$ properties as described in Section §2.1. The result of an HE operation is also an element of operand space O.

HE semantics function: Table 3 shows the semantics of HE operations. Homomorphic expression (h) represents the HE operations. To describe the semantics of HE operations, we define the state function $s \in \mathcal{S} : Var \to O$ and HE semantics function $\mathcal{H} : h \to \mathcal{S} \to O$. $\mathbb{R}^k\llbracket c \rrbracket$ embeds vector representations c to a k-dimensional real-valued vector. Moreover, $\mathcal{P}\llbracket v \rrbracket$ and $\mathcal{C}\llbracket v \rrbracket$ embeds real vector v to a plaintext and ciphertext, respectively.

Although encode and encrypt operations are essential to use other HE operations, the operations do not appear in the program because encryption should be done before its execution and encoding is performed in `upscale`.

The scale and level of result ciphertext are related to the operands and the operation. A negate and rotate operation preserves the scale and level of ciphertext operand. Furthermore, an addition operation preserves the scale and level of operands. The operands of an addition operation should have the same scale and level, and the result of the operation is the same. For a multiplication operation, the result scale is the product of the scale of operands.

`rescale`, `modswitch` and `downscale` operations manipulate the encryption parameter of an operand. A rescale operation reduces the scale of the operand by rescaling value Rand decreases the level. A `modswitch` operation simply decreases the level of an operand without changing scale. A `downscale` operation proposed by [45] reduces the scale as arbitrary amount.

An `upscale` operation works differently for a ciphertext and a constant vector. For a ciphertext, `upscale` changes the scale of the ciphertext by multiplying a identity value which encodes 1 with a specified scale. For a constant vector, `upscale` operation encodes the vector to a plaintext value.