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

Yongwoo Lee¹, Seonyoung Cheon¹, Dongkwon Kim¹, Dongyoon Lee², and Hanjun Kim¹

¹Yonsei University
²Stony 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 downside, 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×) and 5.7 bits (51.2×) 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 relinearization after multiplication (so multiplication adds noise in ELASM). \(N\): polynomial modulus, \(l\): level, \(\sigma\): std. deviation.

<table>
<thead>
<tr>
<th>RNS-CKKS Ops</th>
<th>Time Complexity</th>
<th>Noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>negate, add, mul</td>
<td>(O(N\cdot l))</td>
<td>0</td>
</tr>
<tr>
<td>modswitch</td>
<td>(O(N\cdot l))</td>
<td>0</td>
</tr>
<tr>
<td>rotate, relinearize</td>
<td>(O(\log N \cdot l^2))</td>
<td>(8\sqrt{\pi}\sigma N + 8\sqrt{\pi}N + \sqrt{3N})</td>
</tr>
<tr>
<td>rescale</td>
<td>(O(\log N \cdot l^2))</td>
<td>(6\sqrt{\pi}N + \sqrt{3N})</td>
</tr>
</tbody>
</table>

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., \(v = m \cdot x + n\). 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 10³ to 123 at scale 10². 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 \(m_1 \cdot m_2\) to \(m_1 \cdot m_2/R\) 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, \(v = m \cdot x + n\). Multiplication (due to implicit relinearization operation) and rescale operation introduce additional noises, \(n_{\text{relinearize}}\) and \(n_{\text{rescale}}\), respectively. Thus, \(n^* = m_1 x_1 n_2 \pm m_2 x_2 n_1 \pm m_1 n_2 \pm n_{\text{relinearize}}\) and \(n^{**} = n^*/R + n_{\text{rescale}}\) 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 \((\|n_1\|^2 + \|n_2\|^2)^{1/2}\) and \(\|n_{\text{relinearize}}\|\) is \(8\sqrt{\pi}\sigma N + 8\sqrt{\pi}N + \sqrt{3N}\) that depends only on \(N\). In Figure 1, the noise after rescaling becomes \(n^{**} = n^*/R + n_{\text{rescale}}\) where \(n^*\) is the noise of input ciphertext.](image)

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 \(\|n_{\text{rescale}}\| = 8\sqrt{\pi}\sigma N + \sqrt{3N}\) that depends only on \(N\). In Figure 1, the noise after rescaling becomes \(n^{**} = n^*/R + n_{\text{rescale}}\) 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 \(\|n_{\text{rotate}}\| = 8\sqrt{\pi}\sigma N + 8\sqrt{\pi}N + \sqrt{3N}\), 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 relinearization operation on each ciphertext multiplication. Thus, a ciphertext multiplication in effect is considered to introduce additional noise \(n_{\text{relinearize}}\), as depicted in Figure 1.

The computation error \(\epsilon\) 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 \(\epsilon\) is then determined by the
The encryption parameters and a scale management scheme, this work proposes to use a new find-grained noise-aware constraint (Equation 8) based on the noise $n$ introduced in §4.

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 $\text{rotate}(0.1x)$ example. Consider Figure 3a (input scale = $10^5$) 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 $\pm 0.1$ 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 $m_1 x_1 n_2 \pm m_2 y_1 n_1 \pm n_1 y_2 \pm n_{\text{relinearize}}$ (See Figure 1). Here, $\|n_2\|$ and $\|n_{\text{relinearize}}\|$ are 0 because $1/10$ is plaintext. Thus, the noise becomes 100 ($= 10^2 \cdot 10^1 / 10$) and the error is 0.01 ($= 100 / 10^9$). Finally, rotate introduces an additional noise $n_{\text{rotate}} = 100$, and the final noise and error become 140 ($= \sqrt{100^2 + 100^2}$) and 0.014 ($= 140 / 10^9$).

Figure 3b (input scale = $10^5$) 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 ($= 10^5 \cdot 10^1 / 10$, and the error is 0.001 ($= 1000 / 10^9$). In this case, EVA applies to rescale that divides the noise by scale and adds $\|n_{\text{rescale}}\| = 10$. After rescaling, the scale is $10^2$, the noise is 10 ($\approx \sqrt{(1000 / 10^2)^2 + 10^2}$) and the error is 0.01 ($= 10^9 / 10^9$). Finally, rotate increases the noise to 100 ($\approx \sqrt{100^2 + 10^2}$) after adding $\|n_{\text{rotate}}\| = 100$. The final error becomes 0.1 ($= 100 / 10^9$), 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

\begin{align*}
0 < l & \quad (1) \quad L \leq l \quad (2) \\
|v_l| \leq R_l & \quad (3) \quad R_l \leq f_{\text{sec}}(k) \quad (4) \\
l_1 = l_2 & \quad (5) \quad m_1 = m_2 \quad (6) \\
m_w \leq m & \quad (7) \quad m_{\text{op}} = SNR \cdot n_{\text{op}} \leq m \quad (8)
\end{align*}

Figure 2: RNS-CKKS Constraints. $m$ 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 $m = 10^2$, 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 $l$ 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 $\text{rotate}(0.1x)$, 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 $\text{rotate}(0.1x)$ example. Consider Figure 3a (input scale = $10^5$) 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 $\pm 0.1$ 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 $m_1 x_1 n_2 \pm m_2 y_1 n_1 \pm n_1 y_2 \pm n_{\text{relinearize}}$ (See Figure 1). Here, $\|n_2\|$ and $\|n_{\text{relinearize}}\|$ are 0 because $1/10$ is plaintext. Thus, the noise becomes 100 ($= 10^2 \cdot 10^1 / 10$) and the error is 0.01 ($= 100 / 10^9$). Finally, rotate introduces an additional noise $n_{\text{rotate}} = 100$, and the final noise and error become 140 ($= \sqrt{100^2 + 100^2}$) and 0.014 ($= 140 / 10^9$).

Figure 3b (input scale = $10^5$) 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 ($= 10^3 \cdot 10^1 / 10$, and the error is 0.001 ($= 1000 / 10^9$). In this case, EVA applies to rescale that divides the noise by scale and adds $\|n_{\text{rescale}}\| = 10$. After rescaling, the scale is $10^2$, the noise is 10 ($\approx \sqrt{(1000 / 10^2)^2 + 10^2}$) and the error is 0.01 ($= 10^9 / 10^9$). Finally, rotate increases the noise to 100 ($\approx \sqrt{100^2 + 10^2}$) after adding $\|n_{\text{rotate}}\| = 100$. The final error becomes 0.1 ($= 100 / 10^9$), 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...
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.

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 $\varepsilon$ 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 $R = 10^2$; 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
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 $m_{op}$ is no longer a fixed value but 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 $Q = R^k$, 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 re-scaling 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

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

Performance 

Cost Function 

\( \text{cost}(35, 0.01) = 315.66 \)

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 \( \alpha(P^*|P) = \min(1, \text{cost}(P)/\text{cost}(P^*)) \). Hence, the cost function affects the error and latency of an optimized program. ELASM allows a user to design the cost function (e.g., \( \text{cost}(P) = T \cdot E \) 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 \( \text{rotate}(0.1)^2 \) 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 \((y^2, \text{rotate}(y^2))\).

In simple, the edge representation based on the group id is \((0, 1), (0, 2), (1, 2), (2, 3), \) and \((3, 4)\). 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 \( \text{rotate}(y^2) \) (i.e., before \( \text{rotate} \)).

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 group generation optimization, latency estimation, and backend code implementation of Hecate [45].

---

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.
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 $\varepsilon$. * implies an estimated value. The addition of errors means root-sum-square ($\|v_1 \pm \varepsilon_1\| = (\|v_1\|^2 + \|\varepsilon_1\|^2)^{1/2}$).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Est. Value</th>
<th>Estimated Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>mulcc$(v_1, v_2)$</td>
<td>$x_1 x_2$</td>
<td>$\varepsilon_1^x x_2^2 \pm \varepsilon_2^x x_1^2 \pm n_{\text{relinearize}} / m$</td>
</tr>
<tr>
<td>mulcp$(v_1, v_2)$</td>
<td>$x_1^x x_2$</td>
<td>$\varepsilon_1^x \pm \varepsilon_2^x$</td>
</tr>
<tr>
<td>addcc$(v_1, v_2)$</td>
<td>1</td>
<td>$\varepsilon_1^y \pm \varepsilon_2^y$</td>
</tr>
<tr>
<td>addcp$(v_1, v_2)$</td>
<td>1</td>
<td>$\varepsilon_1^y$</td>
</tr>
<tr>
<td>rotate$(v)$</td>
<td>$x^y$</td>
<td>$\varepsilon^y \pm n_{\text{rotate}} / m$</td>
</tr>
<tr>
<td>upscale$(v)$</td>
<td>$x^y$</td>
<td>$\varepsilon^y \pm n_{\text{rescale}} / (m/R)$</td>
</tr>
<tr>
<td>downscale$(v)$</td>
<td>$x^y$</td>
<td>$\varepsilon^y \pm n_{\text{rescale}} / m_w$</td>
</tr>
</tbody>
</table>

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 $\text{rotate}(y^2)$, 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 $\varepsilon$, the resulting error after multiplication becomes $\varepsilon_1 x_1 \pm \varepsilon_2 x_1 \pm \varepsilon_1 x_2 \pm \varepsilon_2 x_2 + n_{\text{relinearize}} / m$, 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 $\text{rotate}(0.1 \times x^2)$, 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 $F$ 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 $(\text{re})$, plain $(\text{pl})$, or cipher $(\text{ci})$. The $\text{re}$ type represents a raw message that is not encrypted. The $\text{pl}$ type represents an integer vector that contains the encoded data for a raw message. The $\text{ci}$ type represents a ciphertext with scale $m$, and depth $d$ properties. The depth $d$ represents the number of $\text{rescale}$, $\text{downscale}$ and $\text{modswitch}$ 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 $\text{rescale}$, $\text{downscale}$ and $\text{modswitch}$ 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 $\text{rotate} (h, i)$ 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 $\text{rescale}$, $\text{modswitch}$, $\text{upscale}$, and $\text{downscale}$. The $\text{rescale} (h)$ operation decreases the scale by the predefined rescaling factor $R$, and increases the depth by one. The $\text{modswitch} (h)$ operation does not alter the scale but increases the depth by one. The $\text{upscale} (h, m)$ operation, syntactic sugar for multiplying one with an arbitrary scale, increases the scale of $h$ by $m$ but does not change the level. The $\text{downscale} (h)$ operation, syntactic sugar for applying $\text{upscale}$ and $\text{rescale}$ in sequence, reduces the scale to a waterline. Formal operational semantics of the ELASM IR can be found in Appendix A.3.

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 and Rescale for ciphertext multiplication, Rescale for rotate, and Rescale for rescale and downscale. The $\text{mrescale}$ waterline is easily satisfied as a ciphertext multiplication increases the scale by itself. On the other hand, to satisfy the SNR-based noise-aware watermark constraint in Equation 8, the waterlines $m_{\text{rotation}}$ and $m_{\text{rescale}}$ should be defined as $n_{\text{rotation}} \cdot \text{SNR}$ and $n_{\text{rescale}} \cdot \text{SNR}$ for a given SNR. From Table 1, $n_{\text{rotation}}$ is $8\sqrt{2}/3 \cdot \sqrt[N]{N}$ and $n_{\text{rescale}}$ is $8\sqrt{2}/3 \cdot \sqrt[N]{N} + \sqrt{3N}$. Because $n_{\text{rotation}}$ 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 (mrotation) 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 mrescale, because the scale of all input data for a program is set to mrescale.

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 × 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 ldec ∼ U[0,2] and scale increment minc = ReLU(X) for X ∼ 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 √\# of candidate positions. We use √T · (60 + log₂ 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 log₂ 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^{30}. 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×) and 4.2 bits (18.1×) 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
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 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
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. 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.
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

![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.](image-url)
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


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


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


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 **MulC**, 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 mrescale**, because the scale of all input data for a program is set to **mrescale**. 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 **mrotation**, which is computed by **nrotation 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.

\[
\begin{align*}
\langle v := h, s \rangle & \rightarrow s[v \mapsto \mathcal{H}[h][s]] \\
\langle S_1, s \rangle & \rightarrow s' \\
\langle S_1; S_2, s \rangle & \rightarrow \langle S_2, s' \rangle \\
\langle S, s \rangle & \rightarrow s' \quad \mathcal{H}[h][s'] = \pi \\
\langle S; h, s \rangle & \rightarrow \text{halt}(\pi)
\end{align*}
\]

**Operand space**: A constant \( c \in \text{Vector} \) represents the constant vector of floating point numbers and a variable \( x \in \text{Var} \) represents a variable name which can store an operand of...
Table 3: The operational semantics of HE operations. Case represents the abbreviation of each operation. Semantics describes how the HE semantics function \( H \) maps an HE expression \( h \) and a state \( s \) to operand space \( O \). Condition restricts the application of the semantics function \( H \) to satisfy the interface of an HE library.

<table>
<thead>
<tr>
<th>Case</th>
<th>Semantics</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>const</td>
<td>( H\mathcal{c}[s] = \mathcal{R}^k[c] )</td>
<td>( c \in \text{Vector} )</td>
</tr>
<tr>
<td>var</td>
<td>( H\mathcal{c}[s] = o )</td>
<td>( x \in \text{Var}, xx = o \in O )</td>
</tr>
<tr>
<td>encode</td>
<td>( H\mathcal{e}[\text{encode}(h,m)]s = (\mathcal{P}[v], m, L) )</td>
<td>( H\mathcal{h}[s] = \mathcal{R}^k[v], m \geq 1 )</td>
</tr>
<tr>
<td>encrypt</td>
<td>( H\mathcal{e}[\text{encrypt}(h)]s = (\mathcal{C}[v + n\text{rescale}], m, l) )</td>
<td>( H\mathcal{h}[s] = (\mathcal{P}[v], m, 1) )</td>
</tr>
<tr>
<td>negate</td>
<td>( H\mathcal{e}[-h]s = (\mathcal{C}[l], m, 1) )</td>
<td>( H\mathcal{h}[s] = (\mathcal{C}[v], m, 1) )</td>
</tr>
<tr>
<td>addecp</td>
<td>( H\mathcal{e}[h_1 + h_2]s = (\mathcal{C}[v_1 + v_2], m, l) )</td>
<td>( H\mathcal{h}[s] = (\mathcal{P}[v], m, 1) )</td>
</tr>
<tr>
<td>addec</td>
<td>( H\mathcal{e}[h_1 + h_2]s = (\mathcal{C}[v_1 + v_2], m, l) )</td>
<td>( H\mathcal{h}[s] = (\mathcal{P}[v], m, 1) )</td>
</tr>
<tr>
<td>mulc</td>
<td>( H\mathcal{e}[h_1 \times h_2]s = (\mathcal{C}[v_1v_2], m, l) )</td>
<td>( H\mathcal{h}[s] = (\mathcal{P}[v], m, 1) )</td>
</tr>
<tr>
<td>mullc</td>
<td>( H\mathcal{e}[h_1 \times h_2]s = (\mathcal{C}[v_1v_2], m, l) )</td>
<td>( H\mathcal{h}[s] = (\mathcal{C}[v], m, 1) )</td>
</tr>
<tr>
<td>rotate</td>
<td>( H\mathcal{e}[\text{rotate}(h,l)]s = (\mathcal{C}[v + n\text{rotate}], m, l) )</td>
<td>( H\mathcal{h}[s] = (\mathcal{C}[v], m, 1) )</td>
</tr>
<tr>
<td>upscale</td>
<td>( H\mathcal{e}[\text{upscale}(h,m)]s = (\mathcal{C}[v + n\text{rescale}], m, l - 1) )</td>
<td>( H\mathcal{h}[s] = (\mathcal{C}[v], m, 1), m/R \leq m' \leq m )</td>
</tr>
<tr>
<td>downs &lt; &lt;scale &gt; &gt;</td>
<td>( H\mathcal{e}[\text{downscale}(h,m)]s = (\mathcal{C}[v + n\text{rescale}], m, l - 1) )</td>
<td>( H\mathcal{h}[s] = (\mathcal{C}[v], m, 1), m/R \leq m' \leq m )</td>
</tr>
<tr>
<td>modswitch</td>
<td>( H\mathcal{e}[\text{modswitch}(h)]s = (\mathcal{P}[v], m, l - 1) )</td>
<td>( H\mathcal{h}[s] = (\mathcal{P}[v], m, 1) )</td>
</tr>
<tr>
<td>upscale</td>
<td>( H\mathcal{e}[\text{upscale}(h,m)]s = (\mathcal{C}[v + n\text{rescale}], m, l - 1) )</td>
<td>( H\mathcal{h}[s] = (\mathcal{C}[v], m, 1) )</td>
</tr>
</tbody>
</table>

HE operations. An operand \( o \) is a element of operand space \( O \) which is defined by \( O : \mathbb{R}^k \cup \mathcal{P} \times \mathbb{R} \times \mathbb{Z}^+ \cup \mathcal{C} \times \mathbb{R} \times \mathbb{Z}^+ \) which is defined by \( \mathbb{R}^k \) is a real values where \( m \) 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 \( s : \text{Var} \rightarrow O \) and HE semantics function \( H : h \rightarrow s : O \). \( \mathcal{R}^k[c] \) embeds vector representations \( c \) to a \( k \)-dimensional real-valued vector. Moreover, \( \mathcal{P}[v] \) and \( \mathcal{C}[v] \) 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. Further, an addition operation preserves 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 upscale operations manipulate the encryption parameter of an operand. A rescale operation reduces the scale of the operand by rescaling value \( \text{Rand} \) 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.