Performance-aware Scale Analysis with Reserve for Homomorphic Encryption

Yongwoo Lee
Yonsei University
Seoul, Republic of Korea

Seonyoung Cheon
Yonsei University
Seoul, Republic of Korea

Dongkwan Kim
Yonsei University
Seoul, Republic of Korea

Dongyoon Lee
Stony Brook University
New York, USA

Hanjun Kim
Yonsei University
Seoul, Republic of Korea

Abstract

Thanks to the computation ability on encrypted data and the efficient fixed-point execution, the RNS-CKKS fully homomorphic encryption (FHE) scheme is a promising solution for privacy-preserving machine learning services. However, writing an efficient RNS-CKKS program is challenging due to its manual scale management requirement. Each ciphertext has a scale value with its maximum scale capacity. Since each RNS-CKKS multiplication increases the scale, programmers should properly rescale a ciphertext by reducing the scale and capacity together. Existing compilers reduce the programming burden by automatically analyzing and managing the scales of ciphertexts, but they either conservatively rescale ciphertexts and thus give up further optimization opportunities, or require time-consuming scale management space exploration.

This work proposes a new performance-aware static scale analysis for an RNS-CKKS program, which generates an efficient scale management plan without expensive space exploration. This work analyzes the scale budget, called "reserve", of each ciphertext in a backward manner from the end of a program and redistributes the budgets to the ciphertexts, thus enabling performance-aware scale management. This work also designs a new type system for the proposed scale analysis and ensures the correctness of the analysis result. This work achieves 41.8% performance improvement over EVA that uses conservative static scale analysis. It also shows similar performance improvement to exploration-based Hecate yet with 15526× faster scale management time.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; **Compilers**; **Software performance**; • **Security and privacy** → **Privacy-preserving protocols**.

Keywords: Homomorphic encryption, CKKS, scale management, static analysis, reserve, compiler, privacy-preserve machine learning

ACM Reference Format:
Yongwoo Lee, Seonyoung Cheon, Dongkwan Kim, Dongyoon Lee, and Hanjun Kim. 2024. Performance-aware Scale Analysis with Reserve for Homomorphic Encryption. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3617232.3624870

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA*, https://doi.org/10.1145/3617232.3624870.

1 Introduction

Fully homomorphic encryption (FHE) [2] allows an arbitrary arithmetic circuit to process an encrypted input and produce its encrypted result. Once decrypted, the result is the same as the one computed without encryption. The property of FHE opens a new opportunity for privacy-preserving machine learning services in privacy-sensitive fields such as insurance, finance, and healthcare [3, 26, 39, 40, 48, 51]. Among various FHE schemes [7–9, 12–14, 18–20, 27, 29–33], RNS-CKKS [13] supports efficient fixed-point arithmetic and SIMD vectorization, making it well suited for privacy-preserving machine learning applications [54]. Thus, many recent FHE compilers [24, 25, 45] and libraries [28, 38, 52] support RNS-CKKS.

However, RNS-CKKS imposes a new programming burden when designing a correct and fast privacy-preserving machine learning application. RNS-CKKS stores a fixed-point number as an integer by multiplying a scale parameter and requires programmers to manually manage the *scale* m of a ciphertext while respecting several other constraints on ciphertexts and arithmetic operations. Each ciphertext has the maximum integer size, called coefficient modulus $Q = R^l$, derived from encryption parameters *rescaling factor* R and *level* l. In this setting, each multiplication increases the scale of its result and consumes the coefficient modulus. Because the depleted coefficient modulus (also known as scale overflow) harms the correctness, programmers should rescale a ciphertext, using `rescale` operation, after some multiplications by dividing a ciphertext by the rescaling factor, reducing its scale, level, and coefficient modulus: $m' = m/R$, $l' = l - 1$,

and $Q' = Q/R$. Despite its increased programming burden, RNS-CKKS allows users to manipulate ciphertext parameters reflecting trade-off between computation noises and latency, unlike the other schemes such as BGV and BFV.

RNS-CKKS compilers such as EVA [24] and Hecate [45] reduce the programming burden by automatically managing the scales of ciphertexts, but they either fail to fully optimize the performance or require time-consuming scale management exploration. From the beginning to the end of a program, they analyze the scale of a ciphertext and insert scale management operations to satisfy the RNS-CKKS constraints. EVA aims to minimize the input coefficient modulus Q by inserting `rescale` operations if its rescaled scale remains larger than the pre-defined minimal scale, called "waterline". In RNS-CKKS, higher levels of operand ciphertexts incur more latency, so designing level-aware scale management is crucial for performance. Since the level and scale of ciphertexts are tightly coupled, its forward scale analysis is not suitable to analyze the levels of intermediate ciphertexts while inserting `rescale` operations. On the other hand, Hecate achieves better performance than EVA by iteratively exploring many different scale management plans. However, its exploration-based approach cannot scale to large applications because of its long compilation time. For instance, LeNet-5 [42] requires 14763 iterations, leading to 483 seconds of compilation time.

This work proposes *reserve analysis*, a new performance-aware backward static scale analysis for an RNS-CKKS program. This work defines a new scale management term, *reserve r*, as the coefficient modulus over the current scale: $r = R^l/m$, representing an available scale budget of a ciphertext. A key property of reserve is that it is invariant over `rescale`, hence simplifying the reserve analysis. Then, this work formalizes the semantics of reserve and designs the *reserve type system* to correctly manage the reserves and efficiently analyze the latency of RNS-CKKS operations. The reserve analysis performs backward analysis, inferring the operand reserves from the result reserve for each operation, and from the end to the beginning of a program. Then it allocates reserves while prioritizing heavy operations so that it can aggressively reduce the level of those heavy operations. For a given reserve allocation, the rescale placement algorithm statically analyzes the cost of different `rescale` placements and finds the optimal `rescale` placement.

This work implements the proposed reserve type system and analysis on the top of the MLIR [41] compiler framework. This work evaluates the compiler with eight machine learning and deep learning applications in terms of performance (runtime latency) and compilation time, compared to existing RNS-CKKS compilers. This work achieves 41.8% performance (latency) improvement compared to EVA [24] that uses forward static scale analysis. This work also shows similar performance improvement as exploration-based Hecate [45], yet reduces the scale management time by 15526× and the total compilation time by 24.44×.

This work makes the following contributions:

- A new scale management term called *reserve* and its type system that decouples the scale analysis from scale management operations;
- A new static scale analysis called *reserve analysis* that analyzes reserves of each ciphertext in a backward manner, thus enabling improved scale management;
- A new *rescale placement* algorithm finding an efficient position for a `rescale` operation; and
- A new FHE compiler that implements exploration-free performance-aware scale management.

2 Background on RNS-CKKS

This paper focuses on the RNS variant of CKKS (RNS-CKKS) [13] known to be best suited for ML workloads [54], thanks to its efficient support for (approximate) real numbers with fixed-point arithmetic and SIMD vectorization. Other FHE schemes such as BGV/BFV [8, 27] and GSW [34] compute naturally on integers and boolean, and thus require sophisticated, inefficient encoding procedures for real numbers [21].

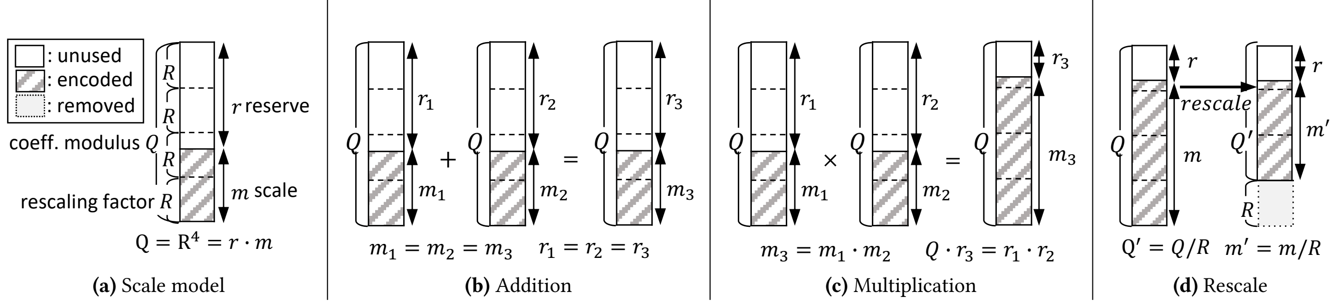
2.1 RNS-CKKS Plaintexts and Ciphertexts

RNS-CKKS exploits the properties of integer polynomial rings for its plaintext and ciphertext spaces. To this end, RNS-CKKS *encodes* a vector of complex (including real) values $x \in \mathbb{C}^{N/2}$ into a polynomial plaintext with integer coefficients $p(X) \in \mathbb{Z}[X]/(X^N + 1)$, where N denotes the degree of a polynomial degree modulus. Then, based on the Ring Learning with Errors (R-LWE) [46], RNS-CKKS *encrypts* a plaintext $p(X)$ to a pair of polynomials ciphertext $c = (c_0(X), c_1(X)) \in (\mathbb{Z}_Q[X]/(X^N + 1))^2$, where the coefficients of the polynomials are bounded by the ciphertext coefficient modulus Q.

The encoding process determines the *scale m* of a plaintext and a ciphertext. To construct a polynomial with integer coefficients, RNS-CKKS multiplies a scale to the real-value data x and embeds a rounded integer $v = \lfloor m \cdot x \rceil$. For instance, $v = 123$ (approximation) is used for $x = 1.234$ at scale $m = 100$. In turn, the scale determines the range of an integer value embedded in a plaintext and a ciphertext.

The encryption process controls the maximum *level l* of a ciphertext. Because the range of the coefficients bounds the range of the embedded integer value, the coefficient modulus Q should be larger than the embedded integer values: *i.e.,* $\lfloor m \cdot x \rceil < Q$. Otherwise, a ciphertext is not recoverable. In RNS-CKKS, the coefficient modulus Q is a multiple of small moduli, also referred to as a rescaling factor R, and the level l of a ciphertext represents the number of rescaling factors therein: *i.e.,* $Q \approx R^l$ (assuming uniform rescaling factors).

One major difference between BGV/FV and RNS-CKKS is how to manage the encoded value and the noise. In BGV/FV,



(a) Scale model (b) Addition (c) Multiplication (d) Rescale

Figure 1. The scale model for RNS-CKKS operations. The ciphertext has a coefficient modulus $Q = R^4$ where R is the rescaling factor. m and r represent the scale and reserve of a ciphertext, respectively.

the range of the encoded value is fixed and only the noise grows during the program execution. On the other hand, in RNS-CKKS, the range of both the encoded value and noise varies according to scale and does not separately manage the noise from the encoded value. Unlike BGV/FV which produces the precise value, RNS-CKKS allows an inherent error from the noise. Hence, the parameter selection in BGV/FV aims to prevent noise overflow, but the goal of parameter selection in RNS-CKKS is to reduce the error, which is parameterized by scale, unlike the fixed noise of the operation.

The scale model in Figure 1a depicts the relations between scale m, level l, and coefficient modulus Q. Conceptually, the coefficient modulus defines the range of the encrypted variable, and the scale represents the actual magnitude of the encrypted data. The multiple rescaling factors are incorporated to support a large coefficient modulus.

This paper newly introduces *reserve* r, which represents the unused bits in the coefficients of a polynomial, *i.e.,* a reserve allocated to ensure that the scale m of a ciphertext remains less than Q in the succeeding operations. Table 1 summarizes the RNS-CKKS parameters and their relations.

Table 1. RNS-CKKS parameters and relations.

Param.	Description
	Encryption Key Parameters (Fixed)
N	Polynomial modulus degree.
Q_{max}	Coefficient modulus of the encryption key.
L	Level of Q_{max}.
R	Rescaling factor. *i.e.,* $Q_{max} \approx R^L$
	Ciphertext Parameters (Varying)
Q	Coefficient modulus of a ciphertext. *i.e.,* $Q < Q_{max}$
m	Scale of an encoded plain/ciphertext.
l	Level of a plain/ciphertext. *i.e.,* $l < L$ and $Q \approx R^l$
r	Reserve of a plain/ciphertext. *i.e.,* $r = Q/m$
μ	Relative scale. *i.e.,* $\mu = \log_R m$
ρ	Relative reserve. *i.e.,* $\rho = \log_R r$ and $\rho = l - \mu$.
	Compiler Parameters (Fixed)
W	Waterline. *i.e.,* $W \leq m$
ω	Relative waterline. *i.e.,* $\omega = \log_R W$
x_{max}	The maximum encoded value. *i.e.,* $m \cdot x_{max} < Q$
	Notations
$\lceil x \rceil$	Ceiling function *e.g.,* $\lceil 0.5 \rceil = 1$
$\{x\}$	Fractional part function. Defined as $\{x\} = x + 1 - \lceil x \rceil$. *e.g.,* $\{1\} = 1$.

2.2 RNS-CKKS Operations and Constraints

RNS-CKKS operations consist of arithmetic operations and scale management operations, whose latency depends on the level of operand ciphertexts. Table 2 summarizes the operations and their constraints.

Some arithmetic operations require their operands to obey operation-specific constraints. For addition (Figure 1b), the scale and level of the two operands should be the same. The result scale and level remain the same as the operands. For multiplication (Figure 1c), only the level of the two operands should coincide and the resulting level remains the same as the operands. After multiplication, the scale of the result increases to the product of the two operand scales. On the other hand, unary operations such as negation and rotation do not have scale/level constraints and the resulting scale and level remain the same as the operand's. Note that `rotate` operation rotates the position of each element in an encrypted vector.

The scale management operations do not affect the encrypted values in a ciphertext, but change the level and scale of a ciphertext. The scale management operations help users to avoid *scale overflow* (exceeding Q) due to multiplications and satisfy the above arithmetic operation constraints. First, `rescale` (Figure 1d) divides the embedded integer value of a ciphertext by the rescaling factor R and removes R from the coefficient modulus Q, hence dividing the scale m by R and decreasing the level l by 1: *i.e.,* $m' = m/R$, $l' = l - 1$, and $Q' = Q/R$. Second, `modswitch` only removes R from the coefficient modulus Q, decreasing the level by 1 but not changing the scale: *i.e.,* $m' = m$, $l' = l - 1$, and $Q' = Q/R$. Finally, `upscale` multiplies a multiplicative identity with a

Table 2. RNS-CKKS operations and constraints.

Op.	Description
Arithemetic Operations (affect encoded values)	
×	Multiplication. The ciphertext parameters of $e_1 \times e_2$ is $m = m_1 \cdot m_2$ and $l = l_1 = l_2$ where the scale and level of e_i is m_i and l_i, respectively.
+	Addition. $m = m_1 = m_2$ and $l = l_1 = l_2$.
−	Negation. $m = m_1$ and $l = l_1$.
`rotate`	Rotation. $m = m_1$ and $l = l_1$.
Scale Management Operations (not affect encoded values)	
`rescale`	Rescaling operation. $m = m_1/R$ and $l = l_1 - 1$
`modswitch`	Modulus switch operation. $m = m_1$ and $l = l_1 - 1$
`upscale`	Upscaling operation. $m = m_1 \cdot m_{up}$ and $l = l_1$

given scale, increasing the scale by the given scale. This work assumes that the programmer writes computations using arithmetic operations, and an FHE compiler inserts scale management operations as in EVA [24] and HECATE [45].

The result scale of RNS-CKKS operations should be large enough to mitigate the noise introduced by the operation. In particular, the ciphertext multiplication (more precisely, `relinearize` after multiplication), `rotate`, and `rescale` operations add a scale-independent noise. Because the magnitude of the noise is fixed, a larger (result) scale reduces the relative magnitude of the noise. Hence, existing scale management schemes allow a programmer to set the lower bound of a scale, called *waterline* [24], and FHE compilers insert `rescale` when the scale after `rescale` is larger than the waterline instead of right after the multiplication.

To meet the RNS-CKKS operation constraints, the user needs to insert scale management operation properly. Suppose that the user inserts `rescale` right after × whose result scale is larger than $W \cdot R$ to prevent scale overflow, as in EVA [24]. The reduced level of the multiplication result introduces the level mismatch for × and +, so the user needs to insert `modswitch` to adjust the level of the other operations. Furthermore, + suffers from the scale mismatch, so the user inserts `upscale` to match the scale of the operands of +.

3 Motivation

This section describes existing scale management approaches in the state-of-the-art RNS-CKKS compilers EVA [24] and Hecate [45], and then discusses their three limitations, motivating new solutions.

3.1 Forward Static Scale Analysis

EVA [24] introduces a forward static scale analysis that analyzes the scale from the begin to the end of a program and inserts a `rescale` operation if the resulting scale becomes larger than the global *waterline*, given by programmers as

Table 3. Latency of RNS-CKKS operations for level 1 to 5 (μs). The other parameters are $N = 2^{15}$ and $R = 2^{60}$.

Op	Level				
	1	2	3	4	5
`modswitch (plain)`	29	43	57	71	86
`modswitch (cipher)`	48	86	156	208	286
`cipher + plain`	50	98	153	209	269
`cipher + cipher`	85	204	250	339	421
`cipher × plain`	211	421	642	853	1120
`rescale (cipher)`	1926	3119	4525	5706	6901
`rotate (cipher)`	3828	7966	13584	20933	28832
`cipher × cipher`	4363	9172	15658	23517	33974

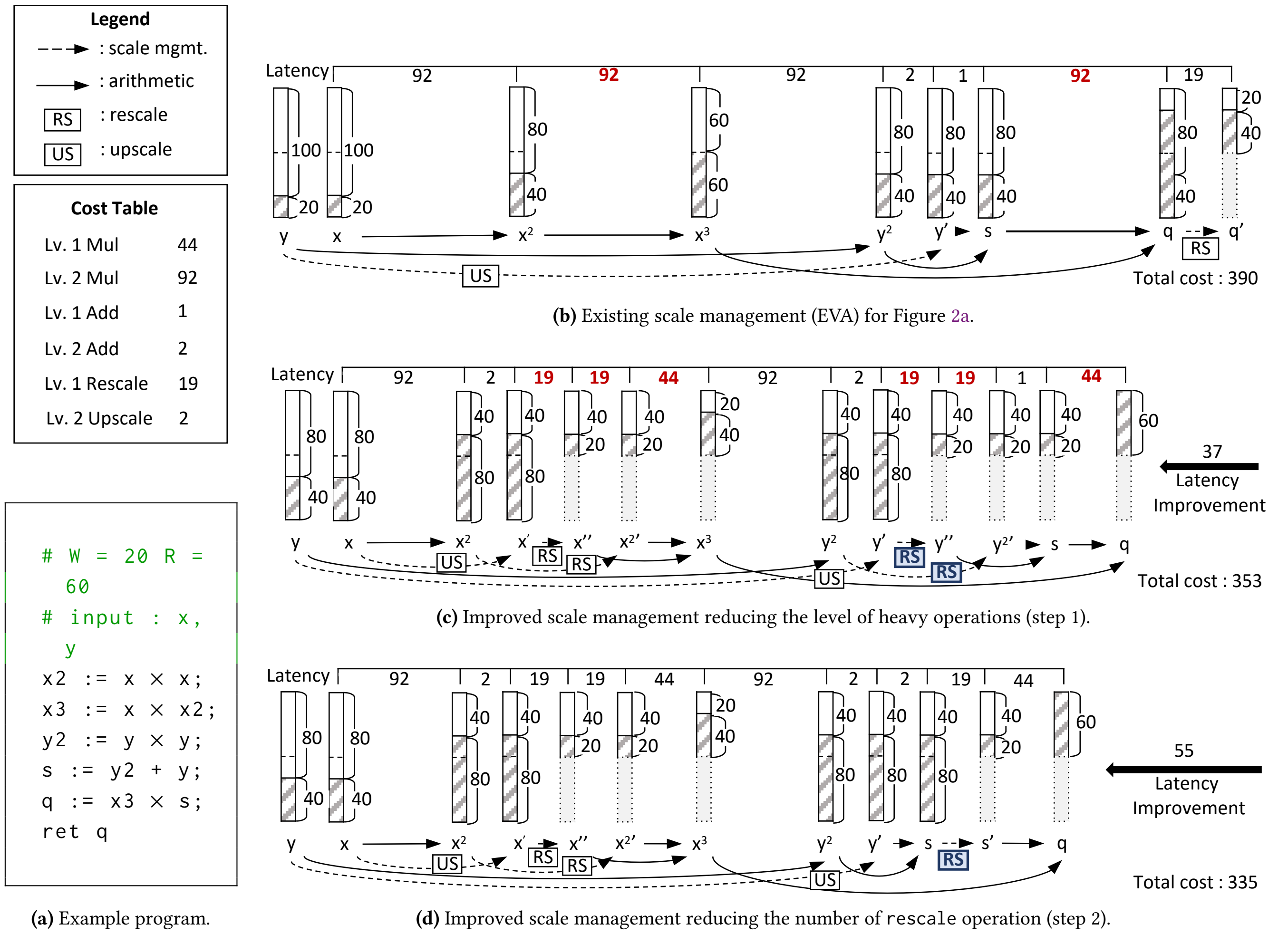
the scale of an input cipertext. This way, EVA intends to minimize the accumulated scale ($m \cdot R^l$) of the program result, which in turn determines the level of the input ciphertexts.

Consider an example program $x^3 \cdot (y^2 + y)$ in Figure 2a where input scale or waterline $W = 20$ (more precisely, 2^{20}) and recaling factor $R = 60$. EVA performs scale analysis and adds scale management operations as shown in Figure 2b. Starting from the input x and y with scale $m = 20$ (waterline W), each multiplication increases the result's scale. EVA inserts `rescale` when its result scale is larger than the waterline. Since the scale of x^2 is not large enough for `rescale`, EVA does not insert `rescale` between x^2 and x^3, and the level and latency of x^2 and x^3 are the same in Figure 2b.

An `upscale` operation is added to increase y's scale from 20 to 40 so that the two operands of the addition $s = y^2 + y$ have the same scale 40. A `rescale` operation is inserted after the last multiplication as it can reduce the scale of q from 100 to 40 yet it remains larger than waterline $W = 20$. By rescaling q, EVA can reduce the ciphertext size, saving storage and network costs. Using the forward analysis, EVA can find that the level of input ciphertexts should be $l \geq 2$ (to avoid scale overflow and afford one `rescale`), and determines that the minimal safe-to-use coefficient modulus $Q = R^2 = 120$.

However, EVA's static analysis does not find the optimal level for each intermediate ciphertext, leading to sub-optimal performance (latency). In RNS-CKKS, the level l determines the size of a ciphertext (Figure 1a) and thus the latency of an RNS-CKKS operation. The lower level, the lower latency. Table 3 shows the results of our latency experiments on different RNS-CKKS operations at different operand levels. As `cipher × cipher` and `rotate` are common in deep learning and machine learning applications, it is crucial for an RNS-CKKS compiler to manage levels of heavy operation.

The new scale management solution in Figure 2c, enabled by this work (which will be discussed later), applies `rescale` operations to x and y aggressively and early and allows many operations to be performed with lower-level operands (level 1 vs. 2 in Figure 2b), reducing end-to-end latency. The early `rescale` indeed increased the accumulated scales. However,



(a) Example program.

(b) Existing scale management (EVA) for Figure 2a.

(c) Improved scale management reducing the level of heavy operations (step 1).

(d) Improved scale management reducing the number of `rescale` operation (step 2).

Figure 2. Scale management plan and execution time for the example program (Figure 2a) that calculates $x^3 \cdot (y^2 + y)$ of EVA (Figure 2b) and this work (Figures 2c and 2d) for the given waterline 20. The numeric numbers for rescaling factor, waterline, scale, and reserve are given in the log base 2.

it does not harm the latency because the increased accumulated scale fully utilizes the remaining scale (reserve) of the result, and more importantly it does not increase the level. Figure 2b has an under-utilized scale 20 in the last ciphertext q', whereas Figure 2c fully utilizes all 60 in q and the maximum level still remains to be 2.

The problem is that it is fundamentally hard for EVA's forward static analysis to perform such level-aware and thus performance-aware scale management. The forward analysis is oblivious to the succeeding operations when it inserts a scale management operation, and thus it cannot optimize the level of an operation without affecting other operations. What is needed is a new construct that enables analyzing a program in a backward direction with a scale budget. This paper introduces the new *reserve* concept that represents the required scale budget from succeeding operations and *reserve analysis* that analyzes the reserve in a backward direction.

3.2 Tightly Coupled Scale Management and Analysis

Unlike arithmetic operations that are given by a program, the number of scale management operations varies across different scale management plans. As shown in Table 3, `rescale` operation has the highest latency among three scale management operations: `rescale` > `upscale` > `modswitch`. The latency of `upscale` is identical to that of `cipher` × `plain` or `cipher` + `plain` depending on implementation. This implies that if all remain equal, reducing the number of `rescale` operations is an important factor for RNS-CKKS program optimization. Yet, existing compilers do not separate the placement of `rescale` operations from scale allocation, missing the optimization opportunity.

Compared to Figure 2c that uses four `rescale` operations, Figure 2d uses three and achieves lower latency. Two solutions differ for the addition $s = y^2 + y$ in that the former

performs `rescale` on two operands before addition, and the latter applies one `rescale` on the result after addition. To enable the optimization, one should be able to determine the position of a `rescale` operation, independent from the scale allocation. For instance, for the addition result s, scale $m = 20$ and level $l = 1$ in Figure 2c; and scale $m = 80$ and level $l = 2$ in Figure 2d. This case motivates a new construct that remains *invariant* to `rescale` operations, decoupling scale analysis from placing scale management operations. The proposed *reserve* concept is invariant over `rescale` operations and this work proposes a new `rescale` placement method, decoupled from the reserve analysis.

3.3 Exploration-based Scale Management

Hecate [45] proposes an alternative iterative exploration-based scale management. On each iteration, Hecate creates multiple scale management plans, each of which adds a scale management operation at a different (random) program point, and generates candidate programs that obey RNS-CKKS constraints by adding additional scale management operations as needed. Hecate selects the one that is statically estimated to have the least latency, and then Hecate iteratively explores the scale management space using the hill-climbing method. With a large number of iterations, Hecate shows that it can achieve less end-to-end runtime latency than EVA. However, as expected, the exploration dramatically increases its compilation time. For instance, Hecate takes 483 seconds of compilation time for the deep-learning application LeNet-5 (§8). Things would get worse for larger networks. Furthermore, scale management time is important because a faster scale management scheme opens a new optimization chance such as bootstrap insertion and data layout selection, which repeatedly invokes the scale management. This work aims to achieve a similar performance improvement without extensive scale management space exploration.

3.4 Problem Statement

Scale management is a process that *inserts the scale management operation and generates a program that satisfies the RNS-CKKS constraints without changing program semantics.* The goal of the optimizing compiler is not just finding a legal scale management but finding an optimal scale management. The optimal scale management problem is defined as:

Problem 1 (Optimal scale management problem)**.** Find a set of ciphertext parameters and additional scale management operations that minimize the program latency from a given program.

This work splits the goal of optimal scale management problem into two-step sub-problems. The first step sub-problem is *minimizing the latency of arithmetic operations* without considering the overheads of scale management operations,

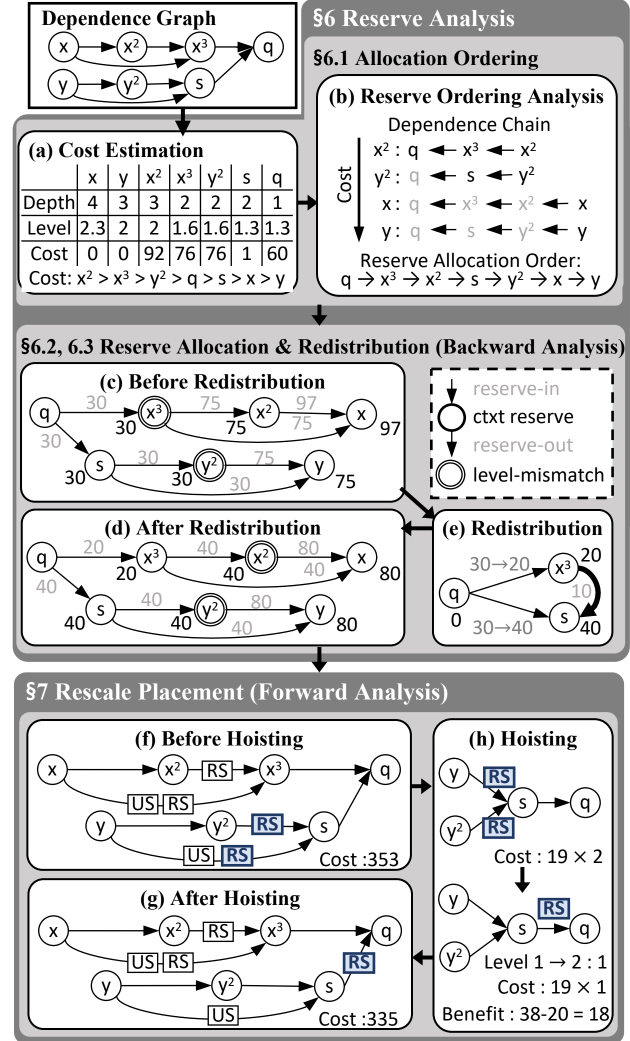


Figure 3. Overview of the reserve analysis and rescale placement, using the same example program as in Figure 2a.

and the second step sub-problem is *minimizing the total latency without changing the reserve of the arithmetic operations.* The two-step problem-solving can simplify the original problem by liberating the compiler from tracing the overhead of scale management operations in the first step and from assigning reserves for each ciphertext in the second step.

Problem 1.1. Find a set of ciphertext parameters that minimize the latency of arithmetic operations without considering the overheads of scale management operations.

Problem 1.2. Find a set of ciphertext parameters and additional scale management operations that minimize the total latency for given reserves of the arithmetic operations.

4 Overview

This work newly introduces a new scale management term, called *reserve*, which represents an available scale budget that will be consumed by succeeding RNS-CKKS operations. Figure 1 illustrates reserve r as the unused bits in the coefficients of a polynomial, and its multiplied result with scale m is equal to its coefficient modulus $Q = r \cdot m$. For addition, reflecting the scale property ($m_1 = m_2 = m_3$), the reserves of the two operands and the result are the same ($r_1 = r_2 = r_3$). For multiplication, since the result's scale is the product of the two operand scales ($m_3 = m_1 \cdot m_2$), and $m = Q/r$ by definition, the reserves have the relation $Q \cdot r_3 = r_1 \cdot r_2$. For `rescale`, unlike the scale that is changed by $1/R$, the reserve remains unchanged, rendering the proposed reserve analysis decoupled from the `rescale` operation placement.

This work newly presents a *reserve type system* (§5) to correctly manage the reserves and efficiently analyze the level (latency) of RNS-CKKS operations. Based on the reserve type system, this work further proposes a new performance-aware static scale analysis for an RNS-CKKS program, named *reserve analysis* (§6), as a solution of Problem 1.1. The reserve type system allows reserve analysis to keep track of the reserve of each ciphertext and its minimum required level to meet the RNS-CKKS (including waterline) constraints.

Figure 3 illustrates the proposed reserve analysis that consists of allocation ordering, reserve allocation, and reserve redistribution, using the same example program as in Figure 2a (whose details will follow). The reserve analysis is a function-level analysis that operates on a function and processes all the operations in the function. The allocation ordering (§6.1) estimates the cost of each operation, analyzes its dependence chain to the return values, and determines the order of the reserve analysis prioritizing heavier (larger latency) operations. The reserve allocation (§6.2) assigns the reserve of each operation based on the reserve type system. Since a higher ciphertext level incurs longer latency for an RNS-CKKS operation, to minimize levels and reserves, the reserve allocation analyzes ciphertext reserves from the end of the program starting with the minimal output reserve in a backward way. The reserve redistribution (§6.3) reassigns reserves prioritizing heavy operation chains to further reduce the levels of heavy operations.

Lastly, as a solution of Problem 1.2, this work proposes a new *rescale placement* algorithm (§7) that places scale management operations for better performance. Starting from default locations defined by the reserve allocation result, the algorithm hoists scale management operations to new locations based on cost analysis.

5 Reserve Type System

To simplify the reserve analysis (§6) and the rescale placement algorithm (§7), this work proposes a new reserve type system. Since a reserve represents a scale budget required for the succeeding operations, reducing a reserve leads to level reduction and in turn latency improvement. Figures 4 and 5 describe the syntax of an RNS-CKKS program intermediate representation (IR) and the typing rules of the proposed reserve type system.

$$
\begin{aligned}
&Prg ::= \overline{F} \\
&F ::= \texttt{func}\ \mathit{fid}\,(\,\overline{v : T}\,)\ \{s; \mathit{ret}\ \overline{e}\} \\
&s ::= v := e \mid s; s \\
&e ::= c \mid v \mid e + e \mid e \times e \mid -e \mid \texttt{rotate}\,(e, i) \\
&T ::= \texttt{real} \mid \texttt{cipher}\,(r) \mid \overline{T} \rightarrow \overline{T}
\end{aligned}
$$

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

Figure 4. The simplified syntax for an RNS-CKKS program. The syntax is similar to that of [45] except for plaintext type and scale management operations. $\overline{A}$ means a list of A.

In the remaining part of this paper, we use log base R terms of relative waterline ($\omega = \log_R W$), relative scale ($\mu = \log_R m$), and relative reserve ($\rho = \log_R r$) to simplify a formula. In addition, $\lceil x \rceil$ is a ceiling function, and $\{x\} = x + 1 - \lceil x \rceil$ is a fractional part function. Note that $\{1\} = 1$, not 0.

5.1 Rationale

One major reason to introduce the reserve type is to precisely detect the chance of level reduction during reserve analysis. As the scale of any ciphertext should be larger than waterline: $m = Q/r > W$, the level of the ciphertext needs to satisfy the condition $Q = R^l \geq W \cdot r$ (*i.e.,* $l \geq \omega + \rho$) for reserve r and waterline W. The minimal level l satisfying the inequality, $l = \lceil \omega + \rho \rceil$, is called *principal level.* As multiplication changes the reserve of a ciphertext, its result principal level can be different from its operand principal level. If so, the multiplication is called a *level-mismatch* operation, implying that `rescale` is necessary for its result. The crux of this formulation is that only with its reserve ρ (and a given parameter ω), the reserve analysis can infer the principal level and the necessity of `rescale`, allowing it to identify level reduction opportunities.

Another major reason to introduce the reserve type system is to liberate the reserve analysis from scale management operation placement. Since a reserve is a scale budget for the succeeding operations, and the `upscale` operation can reduce the reserve as needed, a reserve can represent any reserve that is larger than itself. Introducing the subtyping rule (Equation Sub) that a larger reserve is a subtype of a smaller one. The reserve analysis can safely omit the scale management operation because the type system allows implicit conversion between this type mismatch, separating reserve analysis from scale management operation placement.

$\boxed{\Gamma \vdash e : T}$ Under context Γ, e has type T. $\boxed{\Gamma \vdash s : \Gamma'}$ Under context Γ, s produces context Γ'.

$$\frac{\Gamma \vdash e : \texttt{cipher}(\rho) \quad \rho' \leq \rho}{\Gamma \vdash e : \texttt{cipher}(\rho')} \text{ (Sub)} \qquad \frac{\Gamma, \overline{v : T} \vdash s : \Gamma' \quad \Gamma' \vdash e : T'}{\Gamma \vdash \texttt{func}\ \mathit{fid}\ (\overline{v : T})\ \{s; \mathit{ret}\ \overline{e}\} : \overline{T} \rightarrow \overline{T'}} \text{ (Func)} \qquad \frac{}{\Gamma \vdash c : \texttt{real}} \text{ (Const)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{cipher}(\rho) \quad \Gamma \vdash e_2 : \texttt{real}}{\Gamma \vdash e_1 + e_2 : \texttt{cipher}(\rho)} \text{ (PAdd)} \qquad \frac{\Gamma \vdash e : \texttt{cipher}(\rho)}{\Gamma \vdash \texttt{rotate}(e, i) : \texttt{cipher}(\rho)} \text{ (Rot)} \qquad \frac{\Gamma \vdash e : T}{\Gamma \vdash -e : T} \text{ (Neg)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{cipher}(\rho) \quad \Gamma \vdash e_2 : \texttt{cipher}(\rho)}{\Gamma \vdash e_1 + e_2 : \texttt{cipher}(\rho)} \text{ (Add)} \qquad \frac{\Gamma \vdash e_1 : \texttt{cipher}(\rho + \omega) \quad \Gamma \vdash e_2 : \texttt{real}}{\Gamma \vdash e_1 \times e_2 : \texttt{cipher}(\rho)} \text{ (PMul)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{cipher}(\rho_1) \quad \Gamma \vdash e_2 : \texttt{cipher}(\rho_2) \quad l = \lceil \rho_1 + \omega \rceil = \lceil \rho_2 + \omega \rceil \quad \rho_1 + \rho_2 = \rho + l}{\Gamma \vdash e_1 \times e_2 : \texttt{cipher}(\rho)} \text{ (Mul)}$$

Figure 5. Typing rules of reserve type system. W means the minimal scale required by `rescale` operation.

5.2 Typing Rules

Figure 5 presents the typing rules of the reserve type system. In this section, we mainly discuss the subtyping rule (Equation Sub) and the ciphertext multiplication rule (Equation Mul). Note that unary and addition operations simply have the same type for operands and results.

Subtyping: The reserve type system defines the subtyping rule, Equation Sub, instead of introducing scale management operations, thus hiding scale management operations from the type system. For a given `cipher` type with reserve r, the subtyping rule accepts another `cipher` type with a lower reserve r', which can be generated by any sequence of scale management operations: `upscale`, `rescale`, and `modswitch`. Because `upscale` can reduce reserve as needed, any reserve $r' \leq r$ (*i.e.,* $\rho' \leq \rho$) is a subtype. The reserve is invariant for `rescale`, and `modswitch` is a combination of `upscale` and `rescale`.

Multiplication: The typing rule for multiplication (Equation Mul) incorporates the level and waterline constraints. From $m_1 \cdot m_2 = (R^l/r_1) \cdot (R^l/r_2) = (R^l/r) = m$, the equality $r_1 \cdot r_2 = r \cdot R^l$ (*i.e.,* $\rho_1 + \rho_2 = \rho + l$) holds for $e_1 \times e_2$, where e_1, e_2, $e_1 \times e_2$ has reserve r_1, r_2, r respectively, and the principal level l of operands commonly. The principal level $l = \lceil \rho_1 + \omega \rceil = \lceil \rho_2 + \omega \rceil$ comes from the waterline constraint described in §5.1. Another multiplication rule (Equation PMul) is the specialization for the ciphertext-plaintext multiplication, assuming that the plaintext is encoded with waterline W (*i.e.,* $\rho_2 = l - \omega$).

These typing rules are used for the backward analysis to distribute reserves. Details about how the backward analysis exploits the typing rules are described in §6.2.

6 Reserve Analysis

The goal of the reserve analysis is to minimize the principal levels to reduce the latency of each operation. Unlike existing forward analysis which uses a fixed input scale and derives the output accumulated scale for each operation, the backward reserve analysis uses a fixed output reserve and analyzes the input reserve for each operation. Hence, minimizing the reserve of an operation directly affects the level of the operation and allows aggressive level reduction, contrary to the forward analysis which does not control the level of each operation.

The main idea of the reserve analysis is that the analysis can aggressively reduce the level of a heavy operation (*e.g.,* `rotate`) by deferring level-mismatch to the earlier point of the program. The analysis prioritizes heavy (high latency) operations with allocation ordering to increase level reduction impacts (§6.1), allocates the reserve of ciphertexts based on the allocation order (§6.2), and defers the level-mismatch with reserve redistribution (§6.3) during reserve allocation.

6.1 Allocation Ordering

Allocation ordering gives a processing order for the reserve analysis algorithm to prioritize heavy operations. The insight behind the allocation ordering is that the level of a heavier operation affects the total latency more than the lighter operations, and the reserve allocation should give more optimization chances such as deferring level-mismatch to the heavier operations. Furthermore, the allocation ordering scheme groups the operations that have the same arithmetic structure with the same multiplicative depth as one operation, thus removing duplicated analysis and reducing the analysis time.

To prioritize a heavy operation, the algorithm first estimates the latency of each operation. The level and operation type determine the latency of the operation, but the algorithm does not know the level before reserve allocation. Hence, the algorithm estimates the level from the multiplicative depth and waterline as $1 + \mathit{depth} \cdot \omega$ which is the lower bound of the level, assuming the minimal level increase (ω) for each multiplication. The multiplicative depth means the maximum number of multiplications on the paths from the operation to the return value (starting from 1, not 0). The estimator interpolates the estimated latency from the latency table as the estimated level is often not an integer.

For example, derived from the same program in Figure 2a, x^3 in Figure 3a has the multiplicative depth of 2 because q is the result of a multiplication. Based on the multiplicative depth, the level is estimated as $1 + 2 \times 1/3$ (from $\omega = 20/60$). Then, the cost is computed by $44 \times 1/3 + 92 \times 2/3 = 76$ by interpolating the costs 44 and 92 of ciphertext multiplication for levels 1 and 2, respectively.

Based on the latency estimation, the ordering prioritizes the succeeding operations of a heavy operation because the reserve of succeeding operations needs to be allocated prior to the heavy operation. This work prioritizes the succeeding operations based on the dependency chain with the largest multiplicative depth from the heavy operation to the return value. If two different chains have the same depth, lower-depth operation in the chains is prioritized. For operations with the same operation and chain depths, the ordering uses the chain from the next heaviest operation as the tie-breaker.

Figure 3b shows how to order the operations. The algorithm tracks the longest dependency chain (q, x^3, x^2) for the heaviest operation x^2, and then prioritizes the lower-depth operation ($q > x^3 > x^2$). The next heaviest operation, x^3 is omitted because its dependency chain is a subset of x^2's, then the algorithm finds the dependency chain (q, s, y^2) of the next operation y^2, and so on.

6.2 Reserve Allocation

Given the allocation order from earlier step, reserve allocation infers and allocates the reserve of each ciphertext. For each ciphertext, the allocation algorithm selects the maximum value among incoming reserve values (reserve-ins) from its uses, which is the least common supertype for the reserve-ins.

For example, Figure 3c illustrates how reserves are allocated along the reversed (backward) dependency graph. The gray number represents reserve-ins/outs and the black number stands for the result's reserve selected. The initial case for q (starting from reserve 0) will be explained later. x in Figure 3c has two reserve-ins 97 and 75, and the algorithm sets the reserve of x as 97. Then, the reserve allocation infers its out-going reserve (reserve-out) using the typing rules in Figure 5. For all the operations except multiplication, the operand reserve is the same as the result reserve, so the reserve allocation sets the reserve-out as the reserve.

For multiplication whose operand reserves differ from the result reserve, the reserve allocation algorithm should compute the operand reserves from the result reserve. For plaintext multiplication, the operand reserve is $\rho_{op} = \rho + \omega$ for a given ciphertext reserve ρ. If the operand reserve does not satisfy the waterline condition $l \geq \rho_{op} + \omega = \rho + 2\omega$, the level of the operand is different from the result, the multiplication is marked as a level-mismatch operation.

For ciphertext multiplication, the allocation algorithm should infer the operand reserves ρ_1 and ρ_2 with the result reserve ρ and the operand level l. Note that the operand level l is determined by the operand reserves, not the given result reserve. Thus, the allocation algorithm should infer the operand level first. From $\rho_1 + \rho_2 = \rho + l$ at Equation Mul, $\lceil \rho + l + 2\omega \rceil = \lceil \rho_1 + \rho_2 + 2\omega \rceil \leq \lceil \rho_1 + \omega \rceil + \lceil \rho_2 + \omega \rceil = 2l$. Since l is integer, $\lceil \rho + l + 2\omega \rceil = \lceil \rho + 2\omega \rceil + l \leq 2l$, so $\lceil \rho + 2\omega \rceil \leq l$. Since $l = \lceil \rho_1 + \omega \rceil = \lceil \rho_1 + \omega \rceil$, the allocation algorithm can always satisfy the equality condition, $l = \lceil \rho + 2\omega \rceil$, by controlling reserve allocation on ρ_1 and ρ_2. Then, the reserve allocation equally distributes the reserves to each operand:

$$\rho_1 = \rho_2 = (l + \rho)/2 \;\; where \;\; l = \lceil \rho + 2\omega \rceil \tag{1}$$

If the operand level $\lceil \rho + 2\omega \rceil$ is different from the result level $\lceil \rho + \omega \rceil$, the multiplication is a level-mismatch operation.

For example, q in Figure 3c has the reserve of 0 ($\rho = 0/60$). Since $l = \lceil \rho + 2\omega \rceil = \lceil 40/60 \rceil = 1$ where the waterline is 20 ($\omega = 20/60$), $\rho_1 = \rho_2 = (\rho + l)/2 = 30/60$, and its reserve-out becomes 30. According to the allocation order, the allocation algorithm continues to infer reserves of x^3 with its reserve-in of 30. Here, $l_{x^3} = \lceil \rho_{x^3} + 2\omega \rceil = \lceil 30/60 + 40/60 \rceil = 2$, so x^3 becomes level-mismatch.

6.3 Reserve Redistribution

The reserve redistribution algorithm removes unnecessary level increases caused by the reserve allocation that equally distributes the reserves into operands. Since the level increases at the level-mismatch operations, the reserve redistribution algorithm first finds the level-mismatch operations in which result and operand levels are different ($\lceil \rho + 2\omega \rceil \neq \lceil \rho + \omega \rceil$). If so, since $\{\rho + 2\omega\}$ is the overflowed part from the ceiling, by decreasing ρ by $\{\rho + 2\omega\}$, the reserve redistribution algorithm can avoid the level-mismatch. Because the ciphertext reserve should be the maximum reserve among its reserve-ins, all of the reserve-ins should be equal to or smaller than $\rho - \{\rho + 2\omega\}$.

To remove the evitable level-mismatch, the algorithm iterates the users of the level-mismatched multiplication. If the user cannot redistribute the reserve (*e.g.,* addition), the algorithm recursively finds the proper redistribution target until the required reserve reduction is achieved. If the user is the ciphertext multiplication, the reserve can be redistributed from the other operand (*i.e.,* redistribution target) to the level-mismatched multiplication. If the redistribution target is prioritized over the level-mismatched multiplication, the reserve-out to the target only can increase until the ciphertext reserve of the target ($\rho_{redist} = \rho_{target} - \rho_{target,user}$). Otherwise, the reserve-out to the target can increase freely. Note that the redistribution should not change the principal level of each operand.

Figure 3e shows the example of reserve redistribution where the redistribution target has lower priority than the level-mismatched multiplication. x^3 has a higher priority than the y^2, so redistribution can redistribute reserve until the principal level is not changed. Then, the maximum amount of redistribution is 10 because the maximum reserve

for the given level is 40. The required amount of redistribution is 10 ($\{\rho + 2\omega\} = \{30/60 + 2 \cdot 20/60\} = 10/60$), so the redistribution is succeeded. Figure 3d show the reserve allocation result after redistribution.

6.4 Optimality of Reserve Analysis

The reserve analysis allocates the reserve of each operation one by one. For each allocation step, the reserve analysis minimizes the arithmetic operation latency for the given reserve allocation results of the previous steps at each reserve allocation step, guaranteeing the *local optimal* solution of Problem 1.1.

Assumption 1. The results of the previous allocation steps are locally optimal. The reserve analysis has allocated reserves for the operation with higher priority than the current arithmetic operation in the previous steps, and cannot increase the allocated reserves at the current step. The reserve analysis can freely change only the reserves of the operations with lower priority than the current arithmetic operation.

Theorem 1 (Local-optimality of Reserve Analysis)**.** *The reserve analysis produces a local-optimal solution to the scale management problem for arithmetic operations. In other words, at each reserve allocation step, the reserve analysis produces an optimal solution for the given reserve allocation results of the previous steps.*

Proof. Showing that there does not exist a better reserve allocation A^* that has a lower arithmetic latency than the reserve analysis result A for the given results of the previous steps in a certain reserve allocation step for an operation op.

(Step 1) Let's assume that an reserve allocation A^* has a lower latency than A. Since A^* is faster than A, the level of the op in A^* is lower than A, meaning that level-mismatch occurs only in A. In other words, the allocated reserve ρ^* of the op in A^* is smaller than the allocated reserve ρ in A, and ρ^* is small enough not to suffer from level-mismatch.

(Step 2) If the op is level-mismatched, the reserve redistribution algorithm finds its redistribution target, and redistributes the allocated reserve ρ of op to the target. Since the target has a higher priority and its redistributable reserve (defined by $\rho_{redist} = \rho_{target} - \rho_{target,user}$) is not large enough to avoid level-mismatch, A cannot reduce ρ, and level-mismatch occurs.

(Step 3) On the other hand, A^* redistributes the reserve of op to the target, reducing the reserve from ρ to ρ^*. Since A^* avoids the level-mismatch, the redistributed reserve is larger than the redistributable reserve ($\rho - \rho^* > \rho_{redist}$), changing the reserve of the target which has a higher priority as the given condition.

By contradiction, A^* cannot exist, and A is the optimal solution for the reserve allocation step. □

To find a better solution, the compiler needs to test multiple allocation orders because the local optimal solution preserves the reserve allocation results of the previous steps, and the allocation order affects the quality of the solution. To find the global optimal solution, the compiler should extend the reserve redistribution algorithm to redistribute the reserve from the higher priority operation to the lower priority operation, removing Assumption 1. However, either extension increases the compilation cost a lot like the exploration-based scale management scheme, so this work only finds the local optimal solution, whose performance is competitive to the exploration-based scheme in practice.

7 Rescale Placement

The rescale placement algorithm consists of two steps such as scale management operation insertion and rescale hoisting.

The scale management operation insertion step translates the reserve-typed program into RNS-CKKS compliant program where scale management operations are well placed satisfying the RNS-CKKS constraints. On one hand, the placement algorithm inserts `rescale` to the result of level mismatched multiplications such as x^2 and y^2 in Figure 3f whose result principal level is different from the operand principal level. On the other hand, the algorithm inserts `upscale` and `rescale` between the reserve-in and the ciphertext if their principal levels and reserves are mismatched like x and y in Figure 3f. Thus, the scale management operation insertion can resolve the reserve and level mismatch from subtyping.

The rescale hoisting step moves a `rescale` operation to a later position if profitable. Since the reserve analysis finds level-mismatched operations at the earliest places, the hoisted position is always behind the current position, simplifying the hoisting analysis. First, the hoisting algorithm estimates the benefit of dynamic programming. There exist three cost sources such as level increases, the hoisted source `rescale`, and the destination `rescale`. In Figure 3h, the hoist increases the level of s from 1 to 2, thus causing the cost of 1. Since the two `rescale` operations are hoisted to one `rescale`, the source cost is 38 (19×2) and the destination is 19 (19×1). From the estimated costs, the algorithm computes the benefit as 18 (38 - 1 - 19).

Here, even if the hoisted `rescale` is not beneficial, the hoisting algorithm keeps the destination `rescale` as a further hoisting candidate. If the hoisting algorithm finds a beneficial hoisting opportunity for the candidate whose benefit is larger than the current costs, the hoisting algorithm hoists the `rescale` operations. Then, the algorithm transforms the program. If the use of the source `rescale` is one, the `rescale` operation can be removed.

The proposed algorithm may not produce the globally optimal solution for Problem 1.2. The proposed algorithm cannot find all of the beneficial hoisting candidates, because the algorithm only reflects the benefit from the `rescale` removal when the use of the source `rescale` is one, without considering `rescale` operations with multiple uses.

Table 4. Compile time of EVA, Hecate, and this work. (Speedup over Hecate)

Benchmarks	# Ops	# Iters	Compile Time (ms)				Scale Management Time (ms)			
			EVA	Hecate	This work	Speedup	EVA	Hecate	This work	Speedup
SF	60	553	97.31	319.4	94.11	3.39x	1.641	215.4	0.1405	1533x
HCD	110	736	111.5	494.1	106.8	4.63x	3.190	395.3	0.2151	1838x
LR	123	2675	106.2	4441	109.2	40.66x	2.946	4386	0.2497	17562x
MR	550	3326	215.4	8879	216.0	41.06x	5.323	8688	0.3451	25177x
PR	183	5959	129.0	15768	130.7	120.01x	4.839	15708	0.4031	38965x
MLP	462	677	233.7	2074	232.5	8.92x	3.903	1829	0.2324	7868x
Lenet-5	8895	14763	6802	482.7E3	6805	70.92x	91.50	476.1E3	4.7528	100169x
Lenet-C	9845	13208	7333	469.3E3	7330	64.03x	99.93	462.3E3	5.2385	88253x

8 Evaluation

This work implements the proposed reserve analysis, reserve type system, and code transformations on the top of MLIR compiler framework [41] and uses Microsoft SEAL [52] (Release 3.6.1) for the RNS-CKKS backend library. This evaluation makes comparison with the state-of-the-art EVA [24] and Hecate [45] in terms of compilation time and performance (runtime latency).

This work implements DSL on python that is similar to Hecate DSL. The frontend transforms the python code into the MLIR dialect for this work. The entire compiler framework is open-sourced at the github repository [37]. For the benchmarks, we implemented and tested the eight machine learning and deep learning applications: Sobel Filter (SF), Harris Corner Detection (HCD), Linear Regression (LR), Multivariate Regression (MR), Polynomial Regression (PR), Multilayer Perceptron (MLP), Lenet-5 on MNIST dataset [42] (Lenet-5), and Lenet-5 with CIFAR-10 dataset (Lenet-C). The benchmark sets are the same as those used in Hecate except for the newly added Lenet-CIFAR (Lenet-C).

The image processing benchmarks SF and HCD use 4096 pixels of 64×64 images. The regression benchmarks use 16384 randomly generated inputs for each variable. They perform a training workload that computes the corresponding function. The deep learning benchmarks use random input from MNIST and CIFAR-10 datasets, performing an inference workload. This work uses the gradient descent algorithm for the regression benchmarks with two epochs. The benchmarks assume a packed ciphertext with 16384 slots.

This evaluation runs experiments on Intel(R) Core(TM) i7-8700 @ 3.20GHz with 64GB RAM. For all EVA, Hecate, and this work, 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.

8.1 Compilation Time

Table 4 shows the compile time of EVA, Hecate, and this work. *# op* represents the number of operations in the program and represents the size of the program. *# iter* reports the number of explored plans in Hecate, representing the complexity of the program. For example, MLP consists of 462 operations which are 4x larger than LR, but the iteration is 4x smaller. MLP includes two matrix multiplication and two square operations with a single input which does not introduce the arithmetic between different multiplicative depths. However, LR includes the subtraction between different multiplicative depths and two different inputs, hence the compiler needs to examine more scale management plans than MLP. The most complex and heavy benchmark is Lenet-5 and Lenet-C. The difference between them is that Lenet-C requires a little bit smaller iteration but consists of more operations.

Compile time includes I/O time and the processing time for scale management and the other optimizations including common subexpression elimination and dead code elimination. Without iterative exploration, this work achieves 24.44× speedup over Hecate on average. The compilation time of this work mainly comes from the other processing and I/O, not from scale management time. The scale management time contributes only to 0.14% of the total compilation time.

The most notable benchmarks are Lenet-5 and Lenet-C benchmarks that Hecate takes very long scale management time. The structure of LeNet is Conv - x^2 - AvgPool - Conv -x^2 - AvgPool - FC - x^2 - FC - x^2 - FC which has 11 multiplicative depths. Furthermore, the rotation and additions are applied in Conv, Avg, FC layers. In the evaluation, Hecate reports more than 40 candidate places to insert additional scale management operations, and the brute force search requires more than 2^{40} iterations.

The comparison on the scale management time achieves 15526× speedup over Hecate on average. The speedup of scale management time mainly comes from the removal of exploration. The theoretical speedup should be on par with the *#iter*, but this work achieves further speedup. The ratio of actual speedup over the theoretical speedup is *5.744x* on average. The additional speedup comes from the difference between the single iteration of Hecate and this work. Because Hecate includes multiple optimization passes like CSE and DCE in exploration to precisely reflect the explored performance, the single iteration is much heavier than this

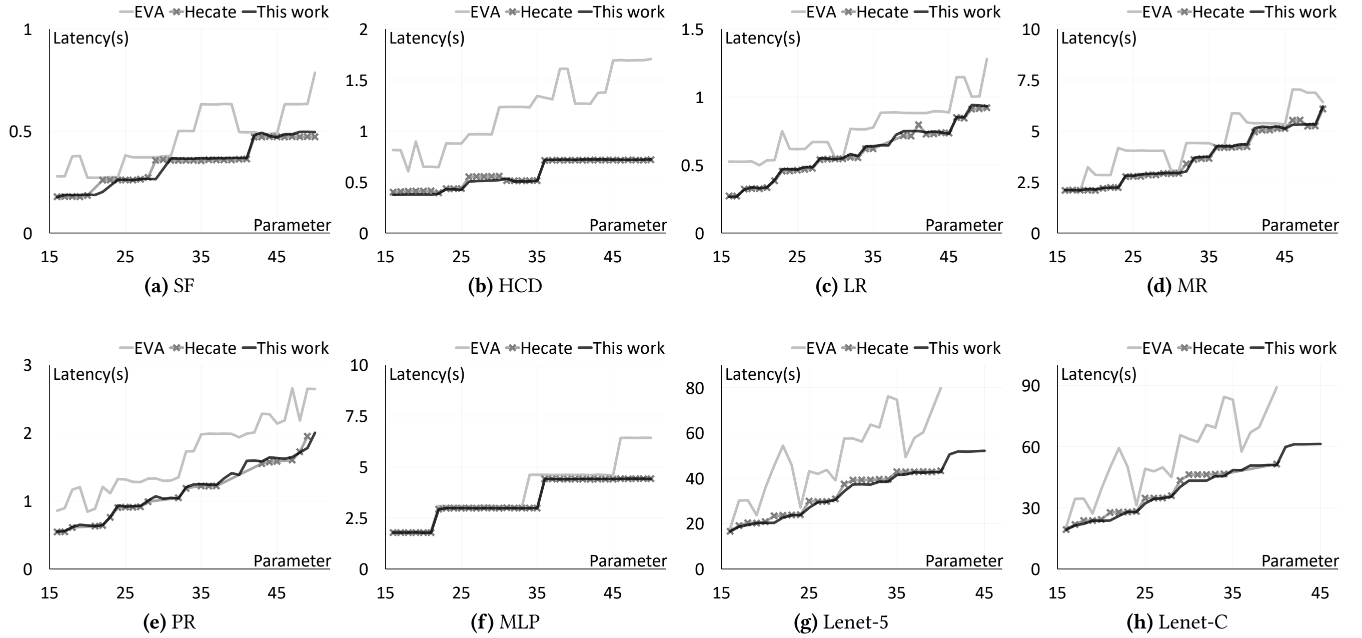


Figure 6. Latency comparison of EVA, Hecate, and this work for a set of waterline parameter (15-50).

work which does not include additional optimizations in scale management. Furthermore, this work achieves a 15x speedup over EVA, mostly coming from scale management units, similar to Hecate, that reduces analysis space.

8.2 Performance

Figure 6 shows the latency of the program compiled by EVA, Hecate, and this work. The latency of EVA shows the limitation of forward analysis that the scale management scheme suffers to control the level and latency. Hecate shows the effect of performance-aware optimization (at the cost of expensive exploration). This work shows almost the same latency with Hecate for the same parameter and 41.8% performance improvement compared to EVA.

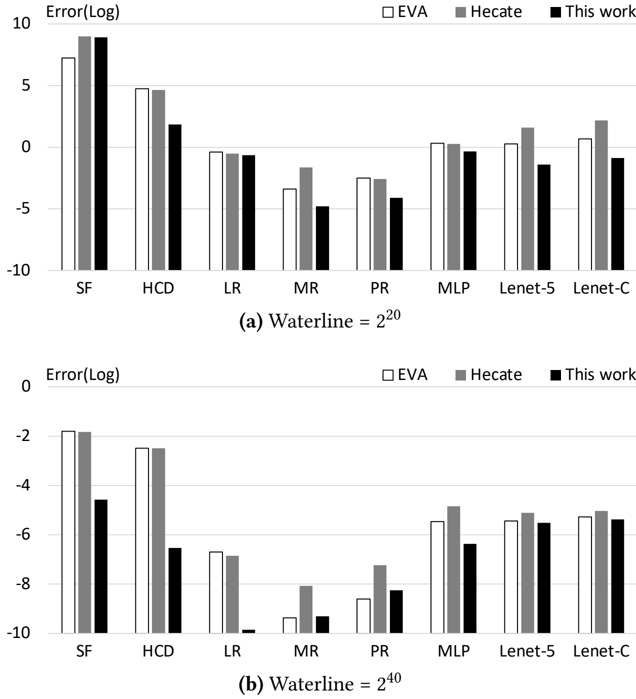
In general, this work achieves similar performance to Hecate for most parameters. This work achieves better performance (max 8.7%) on a few parameters in SF, HCD, Lenet-5, and Lenet-C. Hecate uses a hill-climbing method that converges to local optima not globally optimal. Thus, it could not find a better scale management plan. On the other hand, this work shows some slowdown (max 6.5%) on a few parameters in LR, PR, and MR. Further investigation reveals that the difference comes from the latency of `rescale` operations. The rescale placement algorithm was not able to hoist `rescale` operations to optimal places for some multi-use cases.

We also compare the error of the program compiled by EVA, Hecate, and this work on two waterlines in Figure 7. Hecate extensively uses downscale to minimize the scale of each ciphertext. However, minimizing the scale may increase the error because the noise introduced by RNS-CKKS operations remains invariant to the ciphertext scale and the error is noise over scale. In contrast, this work reflects the cascading effect with reserve, thus not unnecessarily minimizing the scale if it does not improve the performance. Hence, this work gives more chances to increase the scale of each ciphertext and reduces the errors in general without sacrificing the performance thanks to the performance-aware reserve analysis. Thus, this work unexpectedly shows a better error for the parameter in general.

8.3 Performance Improvement Breakdown

Figure 8 shows the breakdown of the proposed algorithms for two waterlines. BA is the baseline implementation of reserve-based backward analysis that does not include scale redistribution (§6.3) and rescale placement (§7). RA is the reserve allocation implementation that includes scale redistribution, but not rescale placement. On average, RA and this work achieve 9.1% and 11.6% speedups over BA for $W = 20$. Speedup was 7.4% and 19.6% for $W = 40$, respectively.

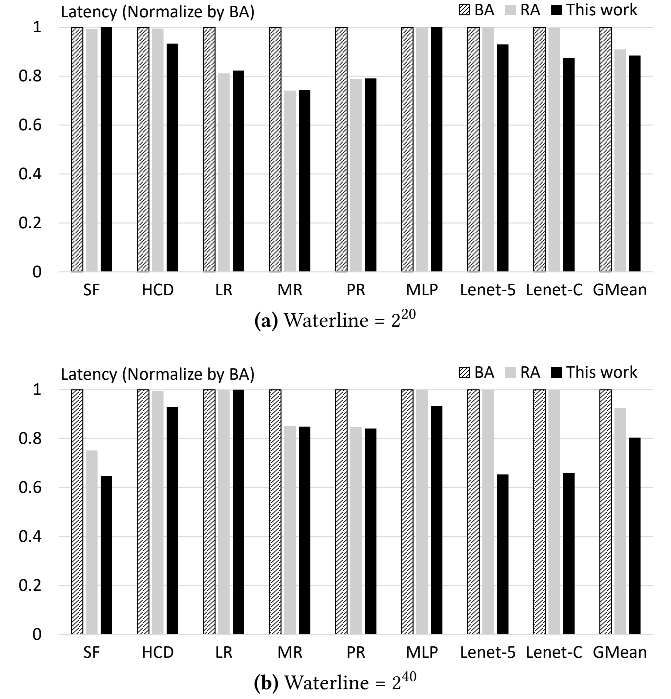
Comparing Figure 8a and Figure 8b, the speedup from each algorithm depends on the type of benchmark. For example, RA does not show speedup over BA on MLP, Lenet-5, and Lenet-C. The speedup of RA over BA is attributed to scale redistribution, which is expected to take effect when there are multiplications between two different ciphertexts. The majority of ciphertext multiplications in deep learning benchmarks are indeed squaring the same ciphertexts. Hence, scale redistribution has little impact on them.



(a) Waterline = 2^{20}

(b) Waterline = 2^{40}

Figure 7. Error comparison of EVA, Hecate, and this work for two different waterlines (m_w)



(a) Waterline = 2^{20}

(b) Waterline = 2^{40}

Figure 8. Performance comparison of Backward Analysis without the reserve redistribution and rescale placement (BA), Reserve Allocation without rescale placement (RA), and this work.

On the other hand, this work does not show speedup over RA on the regression benchmarks LR, MR, and PR. The speedup over RA stems from rescale placement, which targets addition between two different ciphertexts. More precisely, the ciphertext summation can be classified into two classes. The first class is the internal summation that adds up the data stored within a ciphertext, requiring rotation. The other external summation sums up the data stored in different ciphertexts. Rescale placement does not make a visible impact on the internal summation case, which is common in regression benchmarks. Hence, no speedup is noticed.

9 Related Work

There are many fully homomorphic encryption libraries, including HElib [38], PALISADE [50], SEAL [52], and HEaaN [36]. All of these libraries have implementations of specific HE schemes, and provide a low level FHE operations for programmers to implement application. In [54], they evaluates the existing FHE compilers and tools to show performance and usability aspects on various applications. The existing FHE compilers and languages hide the complex details of FHE schemes behind a high-level language and automatically choose an appropriate encryption parameter for a given application. Furthermore, the existing FHE compilers propose various optimization techniques for FHE applications.

9.1 General-purpose HE compilers

Previous works [4, 10, 15, 17, 22–24, 43, 53, 55] proposes new programming languages or the implementation of general-purpose HE applications for existing programming languages.

The programming language and compiler for non-CKKS schemes are proposed by several works. For GSW variants that use a boolean circuit program, Cingulata [10, 17], E^3 [15], Marble [55] and Google's transpiler [35] target programming language and runtime supports to run general purpose C++ program over encrypted data and Lobster [43] uses program synthesis to optimize the Cigulata program. For BGV/BFV schemes that support integer encoding with encryption parameter and SIMD processing with packed ciphertext, RAMPARTS [4] targets a system for optimizing arithmetic computation circuit, supporting direct evaluation of a general-purpose Julia functions over encrypted data on PALISADE [50] library. ALCHEMY [23] supports parameter optimization for the Haskell front-end. The Porcupine compiler [22] and HECO [53] proposes data layout optimization that optimizes ciphertext packing for vectorized HE kernels. Coyote [47] proposes a new FHE-aware vectorization for general purpose (even non-regular) applications by co-optimizing data layout and vector packing. While the compilers only target non-CKKS schemes [8, 16, 27] or lack

consideration for RNS-CKKS scale management, this work proposes performance-aware scale management for CKKS-schemes.

Since RNS-CKKS requires users to match scale and level of each operands, scale manipulation needs a huge programming efforts. So, Encrypted Vector Arithmetic (EVA) [24] and Hecate [45] introduces a new language for FHE computation, designed to be an intermediate representation for other domain-specific languages to ease the burden of FHE parameter optimization. EVA provides arithmetic operations on fixed-width vectors and facilitates encrypted SIMD computations. EVA includes an optimizing compiler that provides a static scale management scheme. Hecate [45] introduces a new scale management space exploration with proactive rescaling algorithm. Unlike EVA optimizes the level, Hecate directly optimizes latency with performance estimation. ELASM [44] improves Hecate to reflect the error of the program.

This work adopts advantages only of EVA and Hecate in terms of scale management: a performance-aware and exploration-free scale management. This work provides better scale management than EVA in terms of performance and Hecate in terms of compilation time with a new performance-aware backward static scale management that consists of a new reserve concept and reserve allocation and rescale placement algorithms.

9.2 Domain-specific HE compilers

Other works [5, 6, 11, 25] target specific domains like DNN inference. CHET [25], nGraph-HE [5, 6] and AHEC [11] are optimizing compilers that enable privacy-preserving deep learning. CHET targets layout selection for packed ciphertext, transforming an input tensor computation circuit into a sequence of FHE operations using domain-specific information. On the other hand, nGraph-HE and AHEC target to support various frontend and backend to enhance usability. nGraph-HE, which extends Intel's nGraph [49], implements the HE backend for nGraph to use the neural network models with TensorFlow [1]. Furthermore, nGraph-HE proposes a CKKS-specific optimization called *lazy rescaling* that places `rescale` operations only after linear layers. AHEC supports nGraph and TensorFlow as frontend and also supports multiple hardware backends with GPU-based HE libraries. All of these compilers support various HE-specific optimizations to improve the performance of the HE applications.

This work can be applied to existing domain-specific compilers, so the existing compiler can get benefit from the lower compilation time and faster performance of the proposed performance-aware static scale management to further improve the performance of the target application.

10 Conclusion

This work proposes a new performance-aware static scale analysis for efficient RNS-CKKS scale management, called *reserve analysis*. With the newly proposed *reserve* term and its type system, this work can decouple the reserve analysis from scale management operations. With the *reserve allocation* algorithm, this work can statically optimize scales and levels for each ciphertext. Finally, with the *rescale placement* algorithm, this work can find an efficient position for a `rescale` operation. Compared to the exploration-based scale management, this work achieves similar performance improvement (41.8% speedup over the existing conservative static analysis) with 15526× faster scale management time.

The proposed light-weight scale management scheme makes a wide range of optimization schemes practical such as data layout selection and bootstrap insertion although the proposed algorithm may not find the global optimal solution. The proposed algorithms are heuristic, which find a satisfiable solution with a small compilation time instead of finding the globally optimal solution with a huge compilation time. Since many homomorphic optimizations repeatedly require scale management to verify their correctness and efficiency, a fast and effective scale management scheme is crucial for optimizations. We will leave the application of the scale management schemes to homomorphic encryption with global-level scale optimization as a future work.

11 Acknowledgements

We thank the anonymous reviewers and Prof. Sara Achour for their valuable feedback and shepherding this paper. We also thank the CoreLab members for their support and feedback during this work. This work is partly supported by IITP-2020-0-01847, IITP-2020-0-01361, IITP-2021-0-00853, IITP-2022-0-00050, and No. RS-2023-00277060 funded by the Ministry of Science and ICT, South Korea. This work is in part supported by the National Science Foundation under Grant No. CCF-2153747 and CNS-2135157. This work is also partly supported by Samsung Electronics, LX Semicon and CryptoLab. (Corresponding author: Hanjun Kim)

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)*, pages 265–283, 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] 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.

[4] 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*, WAHC'19, pages 57–68. ACM, 2019.

[5] 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*, WAHC'19, pages 45–56. ACM, 2019.

[6] 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*, CF '19, pages 3–13. ACM, 2019.

[7] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Annual Cryptology Conference*, pages 868–886. Springer, 2012.

[8] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 309–325, New York, NY, USA, 2012. Association for Computing Machinery.

[9] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on Computing*, 43(2):831–871, 2014.

[10] 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*, pages 13–19, 2015.

[11] 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.

[12] 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*, pages 315–335. Springer, 2013.

[13] 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*, pages 347–368, Cham, 2018. Springer International Publishing.

[14] 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*, pages 409–437, Cham, 2017. Springer International Publishing.

[15] 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.

[16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020. https://doi.org/10.1007/s00145-019-09319-x.

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

[18] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Scale-invariant fully homomorphic encryption over the integers. In *International Workshop on Public Key Cryptography*, pages 311–328. Springer, 2014.

[19] 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, pages 487–504, 08 2011.

[20] 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*, pages 446–464, 04 2012.

[21] Anamaria Costache, Nigel P Smart, Srinivas Vivek, and Adrian Waller. Fixed-point arithmetic in she schemes. In *Selected Areas in Cryptography–SAC 2016: 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers 23*, pages 401–422. Springer, 2017.

[22] 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*, pages 375–389. Association for Computing Machinery, 2021.

[23] 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*, CCS '18. Association for Computing Machinery, 2018.

[24] 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*, PLDI 2020, pages 546–561. Association for Computing Machinery, 2020.

[25] 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*, pages 142–156, 2019.

[26] 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.

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

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

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

[30] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. Association for Computing Machinery.

[31] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. volume 6632, pages 129–148, 05 2011.

[32] Craig Gentry, Shai Halevi, and Nigel P Smart. Better bootstrapping in fully homomorphic encryption. In *International Workshop on Public Key Cryptography*, pages 1–16. Springer, 2012.

[33] 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*, pages 465–482. Springer, 2012.

[34] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Annual Cryptology Conference*, pages 75–92. Springer, 2013.

[35] Shruthi Gorantala, Rob Springer, Sean Purser-Haskell, William Lam, Royce Wilson, Asra Ali, Eric P Astor, Itai Zukerman, Sam Ruth,

Christoph Dibak, et al. A general purpose transpiler for fully homomorphic encryption. *arXiv preprint arXiv:2106.07893*, 2021.

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

[37] Hecate HE Compiler. https://github.com/corelab-src/elasm, 2023.

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

[39] Övünç Kocabaş and Tolga Soyata. Medical data analytics in the cloud using homomorphic encryption. In *E-Health and Telemedicine: Concepts, Methodologies, Tools, and Applications*, pages 751–768. IGI Global, 2016.

[40] 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)*, pages 443–446, 2013.

[41] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[42] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[43] 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*, PLDI 2020, pages 503–518, New York, NY, USA, 2020. Association for Computing Machinery.

[44] Yongwoo Lee, Seonyoung Cheon, Dongkwan Kim, Dongyoon Lee, and Hanjun Kim. ELASM: Error-latency-aware scale management for fully homomorphic encryption. In *32nd USENIX Security Symposium (USENIX Security 23)*, Aneheim, CA, August 2023. USENIX Association.

[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 Berlin Heidelberg.

[47] Raghav Malik, Kabir Sheth, and Milind Kulkarni. Coyote: A compiler for vectorizing encrypted arithmetic circuits. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 118–133, New York, NY, USA, 2023. Association for Computing Machinery.

[48] 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.*, 2019:1113, 2019.

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

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

[51] 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.

[52] Microsoft SEAL (Release 3.6.1). https://github.com/microsoft/SEAL, 2020.

[53] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. Heco: Automatic code optimizations for efficient fully homomorphic encryption. In *32nd USENIX Security Symposium (USENIX Security 23)*, Aneheim, CA, August 2023. USENIX Association.

[54] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. Sok: Fully homomorphic encryption compilers. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 1092–1108. IEEE Computer Society, may 2021.

[55] 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*, WAHC '18. Association for Computing Machinery, 2018.