# Error-Latency-Aware Scale Management Compiler for Fully Homomorphic Encryption

**Yongwoo Lee**

**The Graduate School**
**Yonsei University**
**Department of Electrical and Electronic Engineering**

# Error-Latency-Aware Scale Management Compiler for Fully Homomorphic Encryption

**A Dissertation Submitted**
**to the Deparment of Electrical and Electronic Engineering**
**and the Graduate School of Yonsei University**
**in partial fulfillment of the**
**requirements for the degree of**
**Doctor of Philosophy in Electrical and Electronic Engineering**

**Yongwoo Lee**

**June 2024**

**This certifies that the Dissertation of Yongwoo Lee is approved.**

| | |
|---|---|
| Thesis Supervisor | Prof. Hanjun Kim |
| Thesis Committee Member | Prof. Won Woo Ro |
| Thesis Committee Member | Prof. William Jinho Song |
| Thesis Committee Member | Prof. Youngsok Kim |
| Thesis Committee Member | Prof. Dongyoon Lee |

**The Graduate School**
**Yonsei University**
**June 2024**

## ACKNOWLEDGEMENTS

이 학위논문을 완성할 수 있도록 도와주신 모든 분들께 깊은 감사의 인사를 드립니다.

먼저, 이 논문을 지도해 주신 김한준 교수님께 진심으로 감사드립니다. 포항공대 재학 중이였던 2017년 연구실 인턴부터 지금까지 교수님의 뛰어난 지도력과 끊임없는 격려는 저에게 큰 힘이 되었습니다. 연구 과정에서 항상 좋은 개선 방향을 제시해 주시고, 저의 아이디어를 존중하며 발전시킬 수 있도록 도와주신 교수님의 헌신에 깊이 감사드립니다. 또한, 학위과정 동안 많은 도움을 주시고 논문 심사에 참여해주신 이동윤 교수님께도 특별히 감사드립니다. 2020년부터 함께 동형암호 연구를 시작하여 지금까지 연구와 관련하여 많은 조언을 주시고 항상 열과 성을 다해 도와주셨기 때문에 지금까지 연구를 이어올 수 있었습니다. 논문 심사를 맡아주신 노원우 교수님, 송진호 교수님, 김영석 교수님께도 깊은 감사의 말씀을 드립니다. 귀중한 시간을 내어 제 논문을 꼼꼼히 검토해 주시고, 소중한 피드백과 조언을 해주신 덕분에 논문의 질을 한층 더 높일 수 있었습니다. 교수님들의 지원과 조언에 진심으로 감사드립니다.

컴파일러 최적화 연구실의 모든 동료들께도 감사의 마음을 전합니다. 연구실 동료들의 따뜻한 응원과 협력 덕분에 연구 과정에서 많은 어려움을 극복할 수 있었습니다. 그동안 연구실에서 함께했던 허선영 교수님, 이경민 박사님, 김봉준 박사님, 김창수 박사님, 송승빈 박사님, 조성준, 정신녕, 이재호, 김근우, 최희림, 윤성우, 박현준, 이주민, 권현호, 이찬, 정건모, 정해은, 남주현 선생님에게 감사드립니다. 특별히 동형암호 연구를 함께한 천선영, 김동관, 염호윤에게 깊은 감사의 말을 전합니다. 가족들에게도 깊은 감사를 드립니다. 항상 변함없는 사랑과 지지로 저를 응원해 주신 부모님께 특별히 감사드립니다. 또한, 힘든 시기마다 곁에서 힘이 되어 준 동생에게도 고마움을 전합니다. 여러분들의 응원과 격려가 저에게 큰 힘이 되었습니다. 친구들에게 감사의 인사를 전합니다. 연구 외의 시간에도 함께 웃고 즐기며 스트레스를 풀 수 있었던 시간들이 저에게 큰 위로가 되었습니다. 일상을 함께하고 서로 이해하며 사람다운 삶을 살게 해주는 소중한 사람과 대학원 생활 중 여유를 가지게 해준 이재원, 이인혁에게 특별히 감사의 말을 전합니다. 여러분들과 함께한 시간들은 제 인생에서 소중한 추억으로 남을 것입니다.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

# ABSTRACT


## Error-Latency-Aware Scale Management Compiler for Fully Homomorphic Encryption

Owing to its capabilities for fixed-point arithmetic and SIMD-like vectorization, among fully homomorphic encryption (FHE) schemes that enable computations on encrypted data, RNS-CKKS stands out as a popular choice for privacy-preserving machine learning services. While previous efforts automate scale management essential for RNS-CKKS's fixed-point arithmetic, they show limited performance improvement and accuracy gain. This limitation restricts the ability of users to investigate and optimize the trade-off between error margins and latency.

This dissertation encompasses three pivotal studies that collectively advance the domain of fully homomorphic encryption (FHE), particularly the RNS-CKKS scheme, to bolster privacy-preserving machine learning services. The first study introduces HECATE, an innovative FHE compiler framework that optimizes ciphertext scales by leveraging a novel type system and a rescaling operation termed "downscale". HECATE analyzes various scale management plans for their expected performance impact, enabling optimal rescaling points throughout FHE applications.

The second study delves into the ELASM scheme, which proposes an error- and latency-aware scale management for RNS-CKKS, addressing the limitations of previous works that overlook the output error's impact. By actively managing the ciphertext scale, ELASM minimizes the error-latency cost function, introducing a new scale-to-noise ratio (SNR) parameter and noise-aware waterlines for enhanced error-latency trade-offs. This approach

demonstrates superior performance on machine and deep learning benchmarks compared to existing solutions.

The third study proposes a performance-aware static scale analysis for RNS-CKKS programs, aimed at overcoming the challenges of manual scale management and the inefficiencies of existing compilers. Through backward analysis of the scale "reserve" of each ciphertext and a novel type system, this method redistributes scale budgets for performance-aware management.

Together, these studies present a comprehensive approach to optimizing FHE applications through advanced compiler frameworks, scale management schemes, and performance analysis techniques. They not only demonstrate the feasibility of efficient, privacy-preserving applications but also open new avenues for further research in optimizing encrypted computation, resulting in a 41.8% performance improvement over conservative static analysis approaches and significantly faster scale management times compared to exploration-based methods.

Keywords: Homomorphic encryption, RNS-CKKS, scale management, compiler, error-latency-awareness

# Chapter 1

# INTRODUCTION

Fully homomorphic encryption (FHE) [1] is a breakthrough in cryptography that allows for performing any computational function on encrypted data. The computational function on encrypted data produces an encrypted result that the decryption of the result matches the outcome of the same computations performed on the original data. This remarkable capability supports secure data processing on external platforms like cloud services, crucial for privacy-sensitive applications in areas such as healthcare, finance, and insurance [2, 3, 4, 5, 6, 7] where confidentiality and adherence to stringent regulations are vital. Companies like Microsoft and IBM have already demonstrated FHE's potential through real-world applications in the fields like bioinformatics and financial services.

The RNS-CKKS [8] scheme stands out among numerous FHE schemes [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21] for its suitability for machine learning (ML) tasks [22], offering unique advantages tailored for these applications. It supports fixed-point arithmetic by scaling decimal numbers to integers, allowing precise calculations involving decimals without losing accuracy. Additionally, it features SIMD-like vectorization, or batching, which processes multiple data points at once within a single encrypted ciphertext, boosting computational efficiency and speed—benefits that are particularly valuable in ML scenarios that involve large datasets.

RNS-CKKS has become a popular choice for developing FHE libraries and compilers that cater to privacy-preserving ML services, thanks to libraries like SEAL, HElib, and

HEAAN [23, 24, 25], which provide tailored functionalities to facilitate the deployment of FHE applications. FHE compilers such as EVA [26] and CHET [27] are designed to optimize the use of RNS-CKKS, simplifying the creation of secure ML applications. These tools are essential in translating the theoretical advantages of FHE into practical, operational solutions that enable secure, insightful data analysis without compromising data privacy.

Creating an ML application that is secure, accurate, and fast using the RNS-CKKS scheme involves overcoming significant challenges due to the complex handling required for ciphertext scales. In the fixed-point arithmetic of RNS-CKKS, multiplication operations increase the scale of ciphertexts, which can lead to overflow and inaccuracies. Additionally, certain operations introduce noise that does not depend on the scale, necessitating meticulous balance to prevent underflow and large error margins, thus preserving data integrity.

Understanding the interplay of scale management with error and latency is critical. Accumulated errors from RNS-CKKS operations can degrade the quality of service (QoS), reducing the accuracy of ML predictions. This underscores the importance of precise scale management to maintain or enhance QoS. Furthermore, latency varies with the scaling level of operands, adding layers of complexity to scale management. Strategic rescaling across different stages can markedly influence program performance and introduce extra noise, presenting additional challenges in scale management.

### 1.1 Fully Homomorphic Encryption Application and Compiler

Figure 1.1 illustrates the operational model for applications employing fully homomorphic encryption (FHE). Within this model, an automated compiler adept at managing scale is responsible for both transforming a standard program into its FHE counterpart and for selecting appropriate encryption parameters. These parameters are then utilized to create both a private key and a public key. In the context of an FHE application, the client uses the

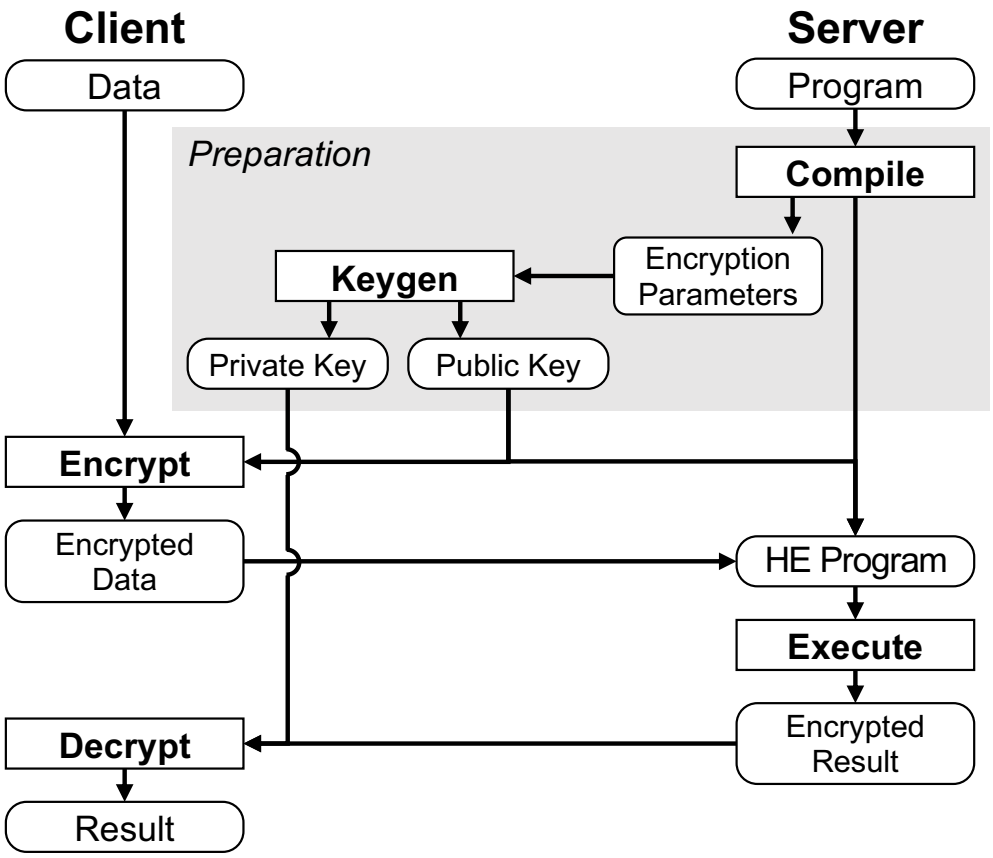



Figure 1.1: FHE application service model.

public key to encrypt sensitive data, which is subsequently sent to the server. The server, in turn, processes this encrypted data using the FHE program compiled by the compiler and sends back the outcome of the computation in an encrypted format. Upon receiving this encrypted computational result, the client can retrieve the original, unencrypted result by using the private key to decrypt the data. This process underscores the seamless integration of encryption into the application's workflow, ensuring data privacy and security throughout the computational exchange.

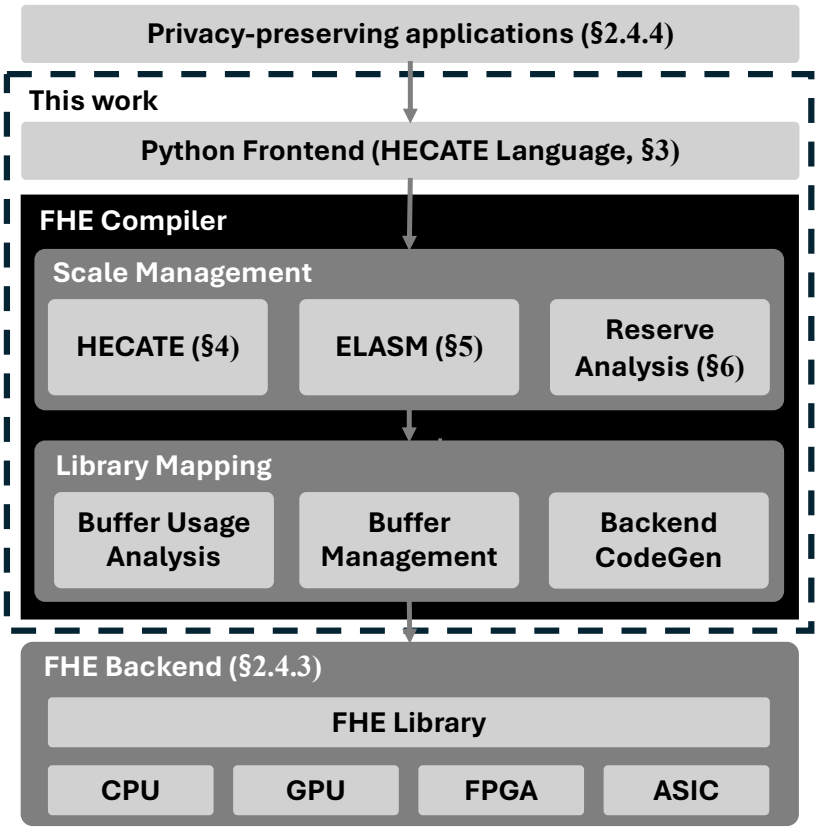



Figure 1.2: FHE Compiler Design.

The advance of fully homomorphic encryption (FHE) compilers [26, 27] has marked a significant milestone in the domain of secure computing, offering developers streamlined tools that automate the intricate process of managing ciphertext scales. These compilers, emerging from recent proposals in the literature, undertake the task of meticulously tracking the progression of scale within cryptographic computations. They are designed to automatically introduce rescale operations whenever the scale of a ciphertext exceeds a predefined threshold, known as the waterline. This approach ensures that the rescaling levels of FHE operands are consistently aligned, thereby maintaining the computational integrity of FHE applications.

Figure 1.2 depicts the role of a scale management compiler for fully homomorphic encryption. The privacy-preserving application (§2.4.4) and FHE library and accelerators (§2.4.3) are not in the scope of this dissertation. The proposed scale management compiler accepts the privacy-preserving applications written in HECATE language (§3) and then the compiler performs scale management as proposed in this dissertation (§4,5,6). The proposed scale management shares the same compiler implementation that includes the scale management as a compiler pass. The compiler provides the library mapping layer that performs buffer usage analysis, buffer management, and backend code generation. The library mapping layer supports optimizations to reduce the ciphertext memory usage and code generations that can be adapted to FHE libraries. The runtime that uses FHE libraries can run the generated code.

## 1.2 Performance-aware Scale Optimization

Recent advancements in fully homomorphic encryption (FHE) compilers [26, 27] have aimed to alleviate the complexity of programming by automating the management of ciphertext scales. These compilers monitor scale progression throughout computations, strategically inserting rescale operations whenever the scale exceeds a predefined minimum threshold, known as the waterline. They also ensure that the rescaling levels of FHE operands are appropriately matched. Despite these efforts, such compilers have not achieved optimal efficiency in FHE application performance. The primary oversight lies in the waterline-based rescaling approach, which neglects to account for the performance implications of rescale decisions. Specifically, in the RNS-CKKS scheme, the latency associated with any given operation is influenced by the rescaling levels of its operands—a factor that current scale management practices overlook, thus missing out on crucial performance optimization possibilities.

This research introduces HECATE [28], an advanced compiler framework for fully homomorphic encryption (FHE) that is designed to finely tune the scales of ciphertexts by taking into account their rescaling levels and the potential impact on performance. HECATE embarks on this task with a multi-pronged approach aimed at optimizing scale management in a way that has not been previously achieved. Initially, HECATE brings to the fore a novel parameter-switching operation named 'downscale'. This operation distinctively rescales a ciphertext even when its scale is below the combined value of the rescaling factor and a predefined threshold, known as the waterline, thereby facilitating a more proactive approach to rescaling.

Further deepening its analysis, HECATE scrutinizes the ciphertexts along with their corresponding FHE operations, organizing them into scale management units. This organization is based on clustering together ciphertexts that share identical scales and rescaling levels, thereby streamlining the scale management process. Subsequently, HECATE embarks on constructing an array of scale management strategies, rigorously estimating the performance of each to identify the most effective scale management plan. This process is not merely about optimizing for efficiency; it also carefully considers the optimal timing and application of rescaling to enhance the overall performance and accuracy of the FHE applications.

Moreover, acknowledging the critical importance of adhering to the scale and rescaling level constraints that FHE operations impose, HECATE integrates a newly developed type system. This system is ingeniously designed to verify the compatibility of scales and rescaling levels of FHE operands, ensuring that the scale management adheres to the necessary computational constraints and maintains the integrity and security of the encrypted data. Through these innovations, HECATE represents a significant leap forward in the field of FHE, offering a robust framework that enhances the scalability, performance, and practicality of privacy-preserving computations.

### 1.3 Error-Latency-Aware Scale Management

In addition to ensuring correctness in their developments, programmers must also weigh the implications of scale management on both error accumulation and operational latency. The accumulation of errors from each operation within the RNS-CKKS framework can significantly impact the Quality of Service (QoS) for an application, with a notable example being the degradation of prediction accuracy in machine learning (ML) applications as the magnitude of error increases. Therefore, developers must minimize these resultant errors to maintain or enhance QoS. Complicating matters further, the latency of RNS-CKKS operations varies based on the rescaling levels of operands, making manual scale management a challenging task due to its indirect yet significant effects on both program latency and the level of noise introduced by subsequent operations.

In response to these challenges, automated scale management proposals such as EVA and Hecate [26, 28] have been developed. Yet, these approaches exhibit two fundamental shortcomings that inhibit users from achieving more efficient error-latency optimizations. Firstly, neither scheme sufficiently addresses the impact of scale management on the resulting errors. EVA's strategy [26] involves adding rescale operations only when the post-rescaling scale surpasses a pre-determined minimum threshold, known as the waterline. This threshold is rigidly set at the maximum scale of the input ciphertexts, disregarding potential impacts on error and latency. Meanwhile, Hecate [28] introduces a proactive scale reduction operation called downscale, which adjusts the ciphertext scale to a predetermined waterline and investigates various scale management strategies focusing solely on latency improvements, neglecting error considerations in the process.

Secondly, both approaches utilize a rudimentary, noise-unaware waterline to mitigate scale underflow risks, failing to account for the variance in noise levels introduced by different

RNS-CKKS operations. This oversight can result in a waterline that is either excessively high for operations introducing minimal noise, or insufficiently low for those generating significant noise. Although adopting a conservative waterline approach may prevent scale underflow, it risks compromising the delicate balance between latency and error, potentially leading to suboptimal trade-offs.

This research introduces ELASM [29], a pioneering approach aimed at enhancing the exploration of trade-offs between error and latency for users. Initially, ELASM brings forth a groundbreaking error prediction model tailored for the RNS-CKKS scheme. This model is adept at anticipating discrepancies between outcomes obtained from computations performed on plaintext and those executed within the FHE framework. It achieves this by accounting for the distinct noise introduced by each RNS-CKKS operation, based on the operation type and the rescaling level of its operands. The model then predicts the resulting error, taking into consideration the noise associated with each operation, the scale of the ciphertext, and the cumulative effect as the data progresses through the computational flow.

Continuing, the paper unveils the Error-Latency-Aware Scale Management (ELASM) strategy, designed to pinpoint the most effective scale management method that aligns with a user-defined cost function that considers both error and latency. ELASM starts by creating an array of scale management scenarios through varied integrations of scale management operations. It proceeds to assess the resulting error via the newly developed error prediction model and calculates latency by aggregating the delays incurred by each RNS-CKKS operation. Utilizing these assessments, ELASM employs the user-defined criteria to compute the cost, selecting the strategy that minimizes this cost. Uniquely, by factoring in both error and latency into its cost function, ELASM is poised to recommend adjustments to the ciphertext's scale when deemed advantageous.

Lastly, this study proposes the introduction of a novel scale-to-noise ratio (SNR) metric, complemented by detailed, noise-aware thresholds for various RNS-CKKS operations. The SNR parameter ensures that the scale m of ciphertext and the noise n produced by an operation must satisfy or surpass the SNR value (i.e., m/n ≥ SNR), drawing a parallel with the signal-to-noise ratio utilized in traditional signal processing. Leveraging this SNR, ELASM delineates specific thresholds for different operations known for their noise generation, such as rescale and rotate, thereby enabling a more favorable balance between error and latency.

### 1.4 Performance-aware Static Scale Analysis

Compilers for the RNS-CKKS encryption scheme, such as EVA and Hecate, significantly alleviate the programming workload by automating the management of ciphertext scales. However, they either do not achieve optimal performance enhancement or necessitate labor-intensive exploration of scale management strategies. Throughout a program's execution, these compilers assess the scale of ciphertexts and integrate scale management operations to adhere to RNS-CKKS's specific requirements. EVA's strategy focuses on reducing the input coefficient modulus Q, incorporating rescale operations when the scale after rescaling exceeds a predetermined minimum scale known as the "waterline". Given that higher operand ciphertext levels translate to increased latency in RNS-CKKS, incorporating level-sensitive scale management becomes essential for enhancing performance. However, EVA's methodology of forward scale analysis struggles to accurately assess the levels of intermediate ciphertexts due to the intricate relationship between level and scale, making it less effective in inserting rescale operations judiciously.

Conversely, Hecate outperforms EVA by conducting an iterative review of various scale management strategies, aiming for a more refined performance optimization. Despite its superior performance outcomes, Hecate's method of iterative exploration faces scalability

challenges with larger applications due to protracted compilation times. For example, compiling the LeNet-5 architecture involves 14,763 iterations, resulting in a compilation duration of 483 seconds. This extensive compilation time underscores the limitations of Hecate's exploration-based approach in efficiently scaling to more complex applications.

In this study [30], we introduce a novel concept of reserve analysis, a performance-oriented backward static scale analysis tailored for RNS-CKKS programs. This analysis introduces the concept of "reserve", defined as the quotient of the coefficient modulus divided by the current scale of a ciphertext. This reserve represents the available scale "budget" for a ciphertext, providing a measure of how much scaling capacity remains before the ciphertext scale reaches its operational limit. A critical attribute of the reserve is its consistency across rescaling operations, which markedly streamlines the process of reserve analysis by maintaining this invariance.

Expanding upon this foundation, the study meticulously formalizes the semantics of reserve, paving the way for the development of a bespoke reserve type system. This system is meticulously crafted to manage reserves accurately and to conduct an efficient latency analysis of operations within the RNS-CKKS framework. Utilizing a backward analytical approach, this reserve analysis methodically deduces the operand reserves based on the reserves of the operation results, tracing from the conclusion of the program to its commencement. This reverse traversal facilitates a strategic allocation of reserves, giving precedence to more computationally intensive operations. Such prioritization allows for a more aggressive reduction in the scaling level of these "heavy" operations, optimizing overall program performance.

Upon establishing a specific reserve allocation, the study delves into a static analysis of the placement of rescale operations within the program. This rescale placement algorithm meticulously evaluates the cost implications of various rescale placement strategies, aiming

to discern the most cost-effective arrangement. By comparing the potential impacts of different placements on program performance, the algorithm seeks to identify the optimal rescale placement. This comprehensive analysis not only enhances the understanding of scale management within RNS-CKKS programs but also contributes significantly to the optimization of latency and computational efficiency, marking a significant advancement in the domain of encrypted computation.

### 1.5 Dissertation Organization

The structure of this dissertation is organized as follows:

Chapter 2 provides an overview of the RNS-CKKS scheme, focusing on ciphertext operations and existing scale management approaches. It also reviews other relevant works in fully homomorphic encryption compilers.

Chapter 3 introduces the HECATE language and its type systems, which formalize the semantics of FHE programs. It sets the foundation for formally describing the proposed scale management schemes.

Chapter 4 details HECATE, a compiler that enhances performance through optimized scale management. It discusses the implementation of scale management units, a scale management space explorer, and scale management code generation.

Chapter 5 introduces ELASM, which incorporates a new compilation parameter, SNR, to achieve error-latency-aware scale management. It explores the concepts of noise-aware waterlines and error-optimizing mechanisms in scale management.

Chapter 6 describes a performance-aware static scale analysis that uses a new concept, the reserve, to accurately analyze and influence the effects of scale management. It explains how this analysis contributes to effective scale management code generation.

Chapter 7 concludes the dissertation, summarizing the contributions and discussing potential avenues for future work.

# CHAPTER 2

# BACKGROUND

Ever since Gentry introduced the world to the concept of fully homomorphic encryption (FHE) in 2009 [9, 10], a variety of FHE methods have been developed, including BGV/BFV [16, 19], CKKS [8, 15], and GSW [31]. This paper focuses RNS-CKKS method, which has proven particularly effective for machine learning (ML) tasks. This effectiveness comes from RNS-CKKS's ability to handle fixed-point numbers (numbers with decimal points) and to perform operations on many data points simultaneously, a feature known as vectorization.

While BGV/BFV also allows for working with multiple data points at once, they're better suited for calculations with whole numbers rather than decimals. On the other hand, GSW stands out for its rapid bootstrapping capabilities—a process that refreshes encrypted data to prevent it from becoming too noisy and unreadable over many calculations. However, unlike RNS-CKKS and BGV/BFV, GSW does not have built-in support for vectorization, making it less ideal for tasks that benefit from processing many data points in parallel.

## 2.1 RNS-CKKS Encoding and Encryption

The RNS-CKKS encryption method takes advantage of the special properties found in rings of integer polynomials for storing and manipulating data in both its unencrypted (plaintext) and encrypted (ciphertext) forms. Essentially, this method transforms a set of complex numbers (which can include real numbers) into a polynomial equation with whole number coefficients.

This polynomial equation lives in a space defined by the equation $X^N + 1$, where $N$ is related to the polynomial's complexity and is known as its degree.

Formally, the set of complex numbers $x \in \mathbb{C}^{N/2}$ turned into a polynomial $p(X) \in \mathbb{Z}[X]/(X^N + 1)$. $N$ denotes the degree of polynomial and the number of complex numbers is exactly half of the degree. $\mathbb{Z}[X]/(X^N + 1)$ means that the coefficient of the polynomial is in a set of integers ($\mathbb{Z}$) and the polynomial is in a quotient ring defined by $X^N + 1$ (*i.e.,* $p(X) \equiv p(X) + (X^N + 1)$).

A crucial step in this process is scaling, which adjusts the data to fit within the polynomial's integer coefficients. This is done by multiplying the original data values by a scaling factor ($m$) and rounding to get a whole number. For example, to encode the value 1.234 for encryption, it could be scaled by 100 to become the integer 123 before embedding in the polynomial. This scaling factor also sets the boundaries for the data's size within the polynomial.

To secure the data, RNS-CKKS uses a technique called Ring Learning with Errors (R-LWE), turning the plaintext polynomial into a ciphertext made up of two polynomial parts. These parts are contained within a specific range, controlled by a parameter known as the ciphertext coefficient modulus ($Q$). In other words, The coefficient of a polynomial should be the element of a quotient ring defined by $Q$ (*i.e.,* $(a \in \mathbb{Z}_{\mathbb{Q}}[X]/(X^N + 1))$). The ciphertext for a given plaintext ($p$) is given by $(a \cdot s + p + e, a)$ for a random mask polynomial $a$, secret key polynomial $s$, and an error polynomial $e$.

Encrypting the data involves defining a maximum 'level' ($l$) for the ciphertext, which ensures the data's size does not exceed the limits set by $Q$. If $Q$ is not large enough to contain the scaled data (*i.e.,* $Q < \lfloor m \cdot x \rceil$), the encryption cannot be correctly undone. RNS-CKKS selects $Q$ as a product of smaller values, known as rescaling factors ($R$), with the level ($l$) indicating how many of these factors are used, essentially controlling the data's precision and

Table 2.1: RNS-CKKS parameters and relations adapted from [30].

| Param. | Description | Relation |
|---|---|---|
| | Encryption Key Parameters (Static) | |
| $N$ | Polynomial modulus degree. | |
| $Q_{max}$ | Maximum coefficient modulus. | |
| $L$ | Level of $Q_{max}$. | |
| $R$ | Rescaling factor. | $Q_{max} \approx R^L$ |
| | Ciphertext Parameters (Dynamic) | |
| $Q$ | Coefficient modulus of a ciphertext. | $Q < Q_{max}$ |
| $m$ | Scale of an encoded plain/ciphertext. | |
| $n$ | Noise of a ciphertext. | |
| $\epsilon$ | Error of a ciphertext. | $\epsilon = n/m$ |
| $l$ | Level of a plain/ciphertext. | $l < L$ and $Q \approx R^l$ |
| $d$ | Rescaling level of a plain/ciphertext. | $l + d = L$ |
| $r$ | Reserve of a plain/ciphertext. | $r = Q/m$ |
| $\mu$ | Log-scale encoding scale. | $\mu = \log_R m$ |
| $\rho$ | Log-scale reserve. | $\rho = \log_R r = l - \mu$. |
| | Compiler Parameters (Static) | |
| $W$ | Waterline. | $W \leq m$ |
| $\omega$ | Log-scale waterline. | $\omega = \log_R W$ |
| $x_{max}$ | Upper bound of encoded value. | $m \cdot x_{max} < Q$ |
| | Notations | |
| $\lceil x \rceil$ | Ceiling function. *e.g.,* $\lceil 0.5 \rceil = 1$ | |
| $\{x\}$ | Fractional part function. *e.g.,* $\{1\} = 1$ | $\{x\} = x + 1 - \lceil x \rceil$ . |

security level. The rescaling factor is predefined as usual, so the level is selected to satisfy the constraint $Q \approx R^l \geq \lfloor m \cdot x \rceil$.

Table 1 summarizes the RNS-CKKS parameters and their interrelationships, providing a reference to understand the various components and their roles within the encryption scheme.

The scale model in Figure 2.1a illustrates the relationships between the scale ($m$), level ($l$), and coefficient modulus ($Q$) in the RNS-CKKS scheme. The coefficient modulus defines the upper bound of the range for the encrypted values, while the scale represents the actual

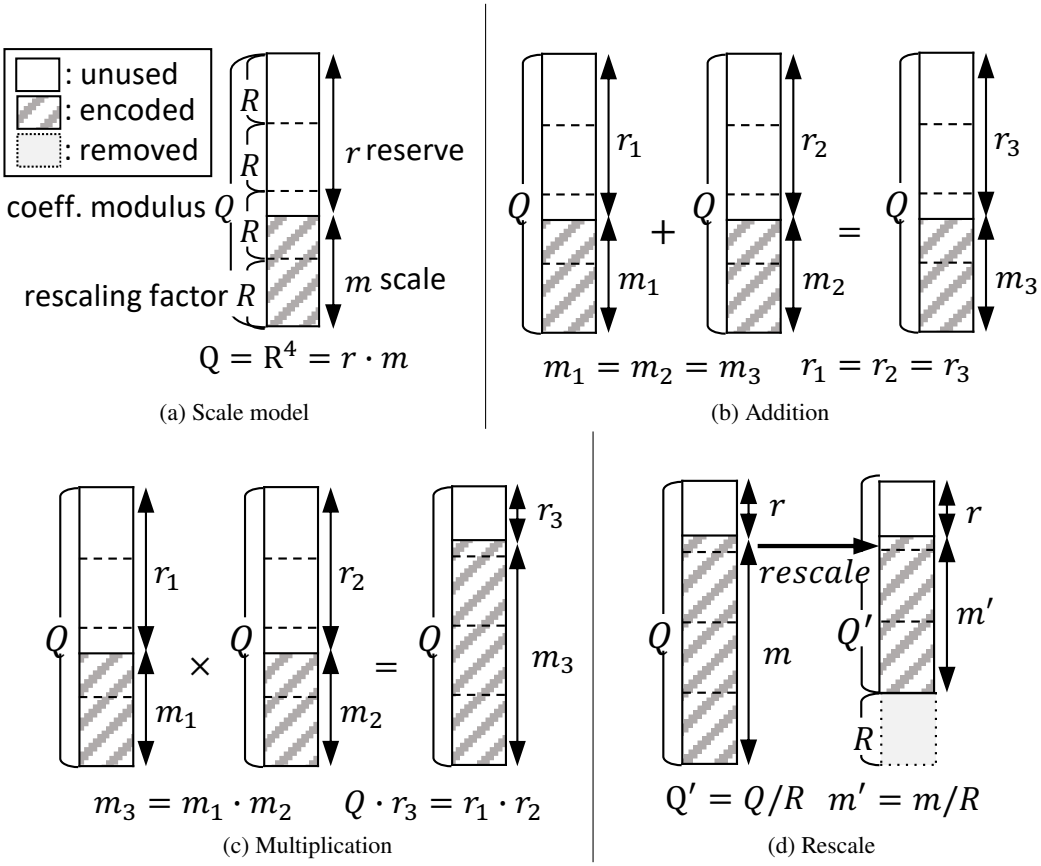



Figure 2.1: The scale model for RNS-CKKS operations reproduced from [30]. The ciphertext has a coefficient modulus $Q = R^4$ where R is the rescaling factor. $m$ and $r$ represent the scale and reserve of a ciphertext, respectively.

magnitude of the encrypted data within that range. Multiple rescaling factors are utilized to support a larger coefficient modulus.

This paper introduces the concept of reserve ($r$), which denotes the unused portion of the coefficient modulus. The reserve is allocated to ensure that the scale ($m$) of a ciphertext remains smaller than the coefficient modulus ($Q$) during subsequent operations (*i.e.,* $Q = r \cdot l$), serving as a safety margin to prevent overflow.

Table 2.2: RNS-CKKS operations and constraints adapted from [30]

| Op. | Description |
|---|---|
| | Arithemetic Operations (affect encoded values) |
| $\times$ | Multiply ciphertext and cipher/plaintext. The 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. |
| $+$ | Add ciphertext and cipher/plaintext. $m = m_1 = m_2$ and $l = l_1 = l_2$. |
| $-$ | Negate ciphertext. $m = m_1$ and $l = l_1$. |
| `rotate` | Change (Rotates) the position of encoded value in ciphertext. $m = m_1$ and $l = l_1$. |
| | Scale Management Operations (not affect encoded values) |
| `rescale` | Remove a single small modulus of coefficient modulus, reducing the scale of ciphertext. $m = m_1/R$ and $l = l_1 - 1$ |
| `modswitch` | Remove a single small modulus of coefficient modulus. $m = m_1$ and $l = l_1 - 1$ |
| `upscale` | Increase the scale of ciphertext. $m = m_1 \cdot m_{up}$ and $l = l_1$ |

### 2.2 RNS-CKKS Operations and Conditions

In the RNS-CKKS scheme, operations are divided into two main categories: arithmetic and scale management, each with its own latency that hinges on the complexity level of the encrypted data involved as listed in Table 2.3. Table 2.2 lays out these operations along with their specific requirements.

For arithmetic operations, certain rules must be followed. For example, when adding two numbers (as shown in Figure 2.1b), both the scale (the precision of the numbers) and the level (a measure of complexity) of the operands need to be identical. The outcome of the addition retains the same scale and level as the inputs. In multiplication (Figure 2.1c), however, it's only the level of the operands that need to match, and while the level of the result stays the same, the scale doubles because it's the product of the scales of the two operands. Unary operations like negation and vector rotation don't impose restrictions on scale or level – the results simply mirror the scale and level of the input. Notably, the `rotate` operation shuffles the positions of elements in an encrypted sequence.

Scale management operations, meanwhile, play a crucial role in maintaining the integrity of encrypted data without altering its value. They adjust both the scale and level of the ciphertext to prevent issues like scale overflow (when multiplication operations cause the scale to exceed a critical threshold) and to meet the requirements for arithmetic operations. The `rescale` operation (Figure 2.1d), for instance, reduces the scale by dividing the integer values embedded in the ciphertext by a factor, simultaneously lowering the level by one. The `modswitch` operation also decreases the level by one but doesn't change the scale. Conversely, `upscale` increases the scale according to a specified factor. These management steps ensure that operations comply with the RNS-CKKS system's constraints, and it's the job of FHE compilers like EVA and HECATE to integrate these steps appropriately within computations.

RNS-CKKS is built on the RLWE principle [32], which works by adding a small random noise during the encryption and the other operations. Alongside the initial noise introduced during encryption, three specific RNS-CKKS operations—`rescale`, `rotate`, and `relinearize` —add extra noise to the final ciphertext, as shown in Table 2.3.

The computation error $\epsilon$ is the difference between the results of plain and FHE computations. Notably, the RNS-CKKS operations add noise regardless of the scale. The error $\epsilon$ is determined by the ratio of the noise to the scale; for instance, a noise $n = 10$ results in an error of 0.01 for a scale $m = 10^3$ and an error of 0.001 if $m = 10^4$. This relationship shows that for a particular level of noise, maintaining a minimum scale is crucial to limiting the maximum error in an FHE operation.

Correctly inserting scale management operations is key to adhering to RNS-CKKS's operational rules. For instance, placing a `rescale` immediately after a multiplication that results in a scale larger than the noise of `rescale` and rescaling factor prevents overflow. However, this can cause issues like level mismatch in subsequent operations, necessitating the use of `modswitch` to align levels and `upscale` to synchronize scales for addition operations.

Table 2.3: Time complexity [27] and noise [33] of RNS-CKKS operations adapted from [29]. The multiplication between ciphertexts implicitly performs `relinearize` after multiplication (so ciphertext-ciphertext multiplication adds noise). $N$: polynomial modulus, $l$: level, $\sigma$: standard deviation of encryption error.

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

This strategy ensures that all operations within the RNS-CKKS framework smoothly integrate, maintaining both the integrity and precision of the encrypted data.

### 2.3 RNS-CKKS Scale Management Compiler

To lighten the workload for programmers, several FHE compilers [26, 27, 34, 35] for RNS-CKKS have been developed. One of the newest among these is the Encrypted Vector Arithmetic (EVA) compiler [26], which introduces two innovative concepts: waterline rescaling and rescale chain. These features aim to automate the management of scale and the selection of parameters effectively.

Scale management involves inserting specific operations to ensure a program adheres to the RNS-CKKS framework, all while maintaining the original meaning of the program. The objective of an optimizing compiler goes beyond merely incorporating acceptable scale management practices; it strives to identify the most efficient scale management solution.

Firstly, EVA automates the placement of `rescale` operations to ensure the scale of a ciphertext remains below the coefficient modulus $Q$, yet it triggers these `rescale` operations only if the resulting scale is still above a predefined threshold called the waterline. Specifically, EVA sets the waterline at the maximum scale of the input ciphertexts, aiming to maintain the scales of all intermediate and resulting ciphertexts above this threshold.

Secondly, EVA introduces the idea of a rescale chain to meet the requirement that operands of binary operations be at the same level. This rescale chain maps out a sequence of `rescale` and `modswitch` operations needed from the initial (root) ciphertext to the target ciphertext. As these operations can increase the rescaling level of a ciphertext, the number of operations in a rescale chain determines the rescaling level of that ciphertext. EVA monitors these levels through the rescale chains to ensure they meet this level consistency requirement.

Finally, EVA sets the coefficient modulus based on the longest rescale chain. Based on the rescaling level which means the number of `rescale` and `modswitch` from the initial ciphertext to the target and the scale of the target ciphertext, the compiler calculates the consumption of the scale capacity. In detail, the scale consumption (accumulated scale) of the ciphertext is $m \cdot R^d$ where $d$ denotes the rescaling level. The maximum accumulated scale of the ciphertexts is selected to the coefficient modulus to prevent the scale overflow.

The concrete example of scale management in EVA will be revisited in the following sections to motivate each work and compare the EVA with the proposed ideas.

### 2.4 Other Related Work

Numerous fully homomorphic encryption (FHE) libraries such as HElib [24], PALISADE [36], SEAL [23], and HEaaN [37] are available, each supporting specific HE schemes and offering low-level FHE operations for application implementation. A study in [22] evaluates these FHE compilers and tools to assess their performance and usability across different applications. These compilers simplify FHE by concealing its complex details behind a high-level language and automatically selecting suitable encryption parameters for applications.

### 2.4.1 General-purpose HE compilers

Previous works [26, 38, 39, 40, 41, 42, 43, 44, 45, 46] proposes new programming languages or the implementation of general-purpose HE applications for existing programming languages.

Several works propose new programming languages or adaptations of general-purpose HE applications for existing languages [26, 38, 39, 40, 41, 42, 43, 44, 45, 46]. Notably, compilers for non-CKKS schemes like GSW and BGV/BFV have been developed. Tools for C++ program like Cingulata [39, 40], $E^3$ [44], Marble [43], Google's transpiler [47] target general-purpose programming, while RAMPARTS [38] and ALCHEMY [45] focus on optimizing arithmetic computations and parameter settings, respectively. The Porcupine compiler [46] and HECO [42] enhance data layout for vectorized HE kernels, with Coyote [48] introducing FHE-aware vectorization for broader application scopes. Despite these advancements, there is a gap in addressing RNS-CKKS scale management, which this work aims to fill by proposing error-latency-aware scale management specifically for CKKS schemes. DaCapo [49] proposes a new automatic bootstrapping placement on the top of HECATE [28]. This improvement enlarges the computational capability of HECATE.

### 2.4.2 Domain-specific HE compilers

Specific domains such as DNN inference have also seen dedicated HE compiler developments, including CHET [27], nGraph-HE [34, 35], and AHEC [50], which enable privacy-preserving deep learning. These compilers, like CHET [27], optimize the data layout for encrypted data, whereas nGraph-HE [34, 35] and AHEC [50] extend support for various frontends and backends to enhance usability and performance. Notably, nGraph-HE introduces 'lazy rescaling,' an optimization for CKKS that minimizes rescaling operations. This work has potential applications in existing domain-specific compilers, enhancing their performance

through the proposed static scale management strategies, thereby benefiting a wider range of applications.

### 2.4.3 RNS-CKKS Algorithm and Acceleration

Several works propose RNS-CKKS algorithm improvements [51, 52, 53, 54, 55] and accelerations [56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66]. Han and Ki [51] propose a new RNS decomposition method that reduces the time complexity of rotation and ciphertext multiplication, and also propose a new bootstrapping method that employs double angular formula. Lee et al. [52] improves the bootstrapping operation by employing the inverse sine function and a new approximation algorithm. Bossuat et al. [53] proposes a double-hoisting algorithm for matrix multiplication, reducing the number of compute-intensive NTT operations during rotation. On the other hand, Bossuat et al. [54] propose a new sparse secret encapsulation that switches the secret key during bootstrapping, allowing better security by employing dense secret. Lee et al. [55] improves the rotation key generation to dynamically generated by a single root key, reducing the size of key set. This algorithmic improvement of RNS-CKKS can improve the FHE libraries and its application is orthogonal to the scale management.

Recent works that accelerate RNS-CKKS on GPU [56], FPGA [57, 62, 63, 65], and ASIC [58, 59, 60, 61, 64, 66] also improve the performance a lot. Jung et al. [56] proposes a first GPU implementation of FHE algorithms. They analyzes the bottleneck of the algorithm and then propose a memory-centric optimization over GPU. Kim et al. [57] proposes an accelerator on FPGA for NTT operation that takes the majority of computation in FHE. HEAX [63] also proposes NTT acceleration on FPGA, utilizing the multiple levels of parallelism from residue-polynomial-level to ciphertext-level. FAB [62] improves the FHE acceleration on FPGA to allow the larger parameters that are compatible with bootstrapping.

Poseidon [65] further improves the NTT implementation and also the automorphism operation in rotation on FPGA implementation.

On the other hand, ASIC implementations tailor the datapath suitable to FHE operations. F1 [61] proposes a fully-pipelined functional unit for automorphism and NTT operation. It is designed as a wide-vector processor with specialized units, reducing data movement—which is its main bottleneck—through a managed memory hierarchy and a compiler that optimizes data reuse and scheduling. BTS [59] and CraterLake [60] improves F1 by reorganizing the functional units and allow a large parameter that enables bootstrapping. ARK [58] proposes a new functional unit that generates a twiddling factor for NTT and a part of evaluation keys in runtime (on chip), so it reduces the memory pressure a lot from the other works. SHARP [66] proposes to use a low rescaling factor parameter and small hardware word size that allows efficient hardware design and shows that the smaller scale parameter can achieve enough accuracy for a real-world problem. MAD [64] proposes a memory-aware design technique that allows to use of a smaller on-chip scratchpad and cache, making the FHE hardware design more viable. All of these hardware accelerations are orthogonal to the scale management and code generation can be retargeted for the proposed accelerators.

### 2.4.4 Privacy-preserving Machine Learning

The Gazelle framework introduced a convolution algorithm tailored for homomorphic encryption, as outlined in [67]. Subsequent studies, including [68, 69, 70, 71, 72, 73], have applied this algorithm in Convolutional Neural Network (CNN) implementations. These approaches often employ multi-party computation to handle non-linear functions within secure computations, sacrificing some benefits of homomorphic encryption due to increased communication overhead. [74] was the first to implement the standard ResNet-20 using the RNS-CKKS fully homomorphic encryption scheme with bootstrapping, though the

extensive use of bootstrapping and convolution operations led to diminished performance. To enhance application efficiency, [75] developed an optimized deep CNN model that reduces bootstrapping times by integrating multiplexed packing and parallel convolution techniques. HyPHEN [76] introduces an advanced convolutional neural network (CNN) construction for fully homomorphic encryption (FHE), enhancing private inference by incorporating novel convolution algorithms, data packing methods, and square activation. Despite these advancements, current privacy-preserving machine learning implementations remain manually optimized and lack automatic scale management. Programmers must also manually manage bootstrapping for each modification in waterline or model structure, increasing workload and potentially leading to less efficient target programs. Scale management compiler can compile the proposed PPML applications to automate the daunting scale management task.

# CHAPTER 3

# HECATE LANGUAGE AND TYPE SYSTEMS

This chapter introduces the HECATE language along with its operational semantics and type systems, which are used to depict FHE programs and formalize the workings and restrictions of FHE operations. This chapter aims to establish a structured framework that will help in formally describing the optimization techniques discussed in subsequent chapters.

## 3.1 HECATE Language

Figure 3.1 presents the formal syntax of the HECATE language, which is generated and utilized by the scale management compiler during the optimization process. The syntax highlighted in the gray box outlines the scale management operations that are not present in the original input program but are inserted by the scale management algorithm.

An HECATE program, $Prg$, consists of a series of functions $F$, which can be invoked by an external driver. Each function is structured as a sequence of statements $S$, ended by a return expression. Within the body of a function, HECATE strictly defines a series of assignment statements, $v := e$. It is important to note that an RNS-CKKS program excludes conditional or looping constructs like if-else statements and for-loops.

Each function argument $v$ in HECATE must be assigned a scale type $T$ or $R$. This type $T$ can be a real vector (`real`), plain (`plain`), or cipher (`cipher`). The `real` type denotes raw, unencrypted data. The `cipher` type indicates a ciphertext, characterized by a scale $m$ and a depth $d$ or a level $l$. Note that HECATE language can be used with two different types namely

$$
\begin{array}{rcl}
Prg & ::= & \overline{F} \\
F & ::= & \texttt{func}\ \mathit{fid}\ (\overline{v : T|R})\ \{s; \overline{h}\} \\
S & ::= & \epsilon \mid v := h \mid S; S \\
h & ::= & c \mid v \mid h + h \mid h \times h \mid -h \mid \texttt{rotate}(h, i) \mid \texttt{rescale}(h) \\
 & & \mid \texttt{modswitch}(h) \mid \texttt{upscale}(h,\ m) \mid \texttt{downscale}(h) \\
T & ::= & \texttt{real} \mid \texttt{cipher}\,(m, d) \mid \texttt{plain}\,(m, d) \mid \overline{T} \to \overline{T} \\
R & ::= & \texttt{real} \mid \texttt{cipher}\,(r) \mid \overline{R} \to \overline{R}
\end{array}
$$

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

Figure 3.1: The formal syntax of the HECATE language adapted from [29, 30]. The syntax with a gray box shows the scale management operations which is not used by a programmer. $\overline{A}$ means a list of $A$. T and R means scale and reserve type, respectively.

scale type $T$ and reserve type $R$. With scale type $T$ and reserve type $R$, the program would use rescaling level (depth) $d$ and ciphertext level $l$.

Depth $d$ counts the number of scale-adjusting operations—rescale, downscale, and modswitch—applied to a ciphertext. Essentially, depth $d$ is a different approach to conceptualizing the ciphertext level $l$, which reduces from an initial level $L$ with each `rescale`, `downscale`, and `modswitch` operation, where $d = L - l$. Since the initial level $L$ is determined after scale management and isn't known until compilation, HECATE uses depth $d$, which increases from 0, as opposed to level $l$, which decreases from $L$.

Expressions in FHE, denoted as $h$, could be constants, variables, binary operations (addition and multiplication), negation, or rotation. Subtraction is executed using negation, while division is handled by multiplying by the inverse of the divisor. The rotate operation, `rotate`$(h, i)$, specifically involves shifting vectored data within a ciphertext $h$ by an offset $i$.

HECATE language simplifies the programmer's task by abstracting away the need to manually code low-level scale management operations like `rescale`, `modswitch`, `upscale`, and `downscale`. The `rescale`$(h)$ operation decreases the scale by a predefined rescaling factor R and increments the depth by one. `modswitch`$(h)$ does not affect the scale but increases the

depth by one. The `upscale`$(h, i)$ operation, effectively multiplying by a unit with an arbitrary scale, raises the scale of $h$ by $i$ but does not alter the depth. The `downscale`$(h)$ operation, which combines `upscale` and `rescale`, adjusts the scale down to a specified waterline. The detailed operational semantics of the HECATE language are available in §3.3.

### 3.2 Scale Type Systems

This section describes scale type system of HECATE language without explaining reserve type system. Because reserve type system requires an understanding of the concept of reserve, the details of reserve type system will be explained in §6.3.

Figure 3.2 outlines the typing rules of the HECATE language. This type system is specifically crafted to meet and uphold the constraints of the RNS-CKKS encryption scheme, as detailed in Table 2.2, including the Signal-to-Noise Ratio (SNR)-based, noise-aware waterlines outlined in §5.2. The type soundness of the HECATE language ensures that a program that is correctly typed will not breach any of the stipulated RNS-CKKS constraints. A provisional proof of this is provided in §3.4.

The scale type system incorporates the operation-wise minimal scale constraints effectively. Crucially, three specific rules—Equation $\text{Mul}_{CC}$ for ciphertext multiplication, Equation Rot for rotation, and Equation RS for rescaling—mandate certain minimum scales or waterlines: $m_{relinearize}$ for ciphertext multiplication, $m_{rotation}$ for rotation, and $m_{rescale}$ for rescaling.

In general, waterline constraints from EVA and HECATE assume $m_{relinearize} = m_{rescale} = m_{rotation} = W$. On the other hand, ELASM assumes fine-grained operation-wise minimal scale constraints. The $m_{relinearize}$ waterline is naturally met as the multiplication of ciphertexts inherently boosts the scale. However, meeting the fine-grained, noise-aware waterline constraints requires defining the waterlines $m_{rotation}$ and $m_{rescale}$, proportional to $n_{rotation}$ and $n_{rescale}$ respectively. The waterlines are computed based on the given constants

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

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash v := e : \Gamma, v : T}\ \text{(Asn)} \qquad \frac{\Gamma \vdash s : \Gamma' \quad \Gamma' \vdash s' : \Gamma''}{\Gamma \vdash s; s' : \Gamma''}\ \text{(Stm)}$$

$$\frac{\Gamma, \overline{v : T} \vdash s : \Gamma' \quad \overline{T} \in \{\texttt{real}, \texttt{cipher}(m, 0)\} \quad \overline{\Gamma' \vdash e : U}}{\Gamma \vdash \texttt{func}\ \mathit{fid}\ (\overline{v : T})\ \{s; \overline{e}\} : \overline{T} \to \overline{U}}\ \text{(Fun)}$$

$$\frac{}{\Gamma \vdash c : \texttt{real}}\ \text{(Const)} \qquad \frac{\Gamma \vdash h : \texttt{real}}{\Gamma \vdash -h : \texttt{real}}\ \text{(Neg}_R\text{)}$$

$$\frac{\Gamma \vdash h : \texttt{cipher}(m, d)}{\Gamma \vdash -h : \texttt{cipher}(m, d)}\ \text{(Neg}_C\text{)} \qquad \frac{\Gamma \vdash h_1 : \texttt{real} \quad \Gamma \vdash h_2 : \texttt{real}}{\Gamma \vdash h_1 \oplus h_2 : \texttt{real}}\ \text{(Bin}_R\text{)}$$

$$\frac{\Gamma \vdash h_1 : \texttt{cipher}(m, d) \quad \Gamma \vdash h_2 : \texttt{scale}(m, d)}{\Gamma \vdash h_1 + h_2 : \texttt{cipher}(m, d)}\ \text{(Add)}$$

$$\frac{\Gamma \vdash h_1 : \texttt{cipher}(m, d) \quad \Gamma \vdash h_2 : \texttt{plain}(m', d)}{\Gamma \vdash h_1 \times h_2 : \texttt{cipher}(mm', d)}\ \text{(Mul}_{CP}\text{)}$$

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

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

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

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

$$\frac{\Gamma \vdash h : \texttt{real}}{\Gamma \vdash \texttt{upscale}(h, m) : \texttt{plain}(m, 0)}\ \text{(US}_R\text{)} \qquad \frac{\Gamma \vdash h : \texttt{cipher}(m, d) \quad m' \geq m}{\Gamma \vdash \texttt{upscale}(h, m') : \texttt{cipher}(m', d)}\ \text{(US}_C\text{)}$$

Figure 3.2: Typing rules of the HECATE language with scale type reproduced from [29]. $m_{rescale}$ means the minimal scale required by a rescale operation and $m_{rotation}$ means the minimal scale required by a rotate operation. `scale` includes `cipher` and `plain` types, and $\oplus$ includes $+$, $\times$.

(called SNR) and the noises associated with the `rotate` and `rescale` operations. Due to the noise level $n_{rotate}$ being influenced by the ciphertext level—which is not predetermined prior to scale management— ELASM accounts for this by using a worst-case scenario estimate.

### 3.3 FHE Operational Semantics

The runtime systems of HECATE use a set of comprehensive rules to define the big-step operational semantics, which guide the execution of homomorphic encryption (HE) operations.

The HE semantics function $\mathcal{H}$ captures the meaning of HE expressions $h$, as detailed in Figure 3.1.

$$\frac{}{\langle v := h, s\rangle \mapsto s[v \mapsto \mathcal{H}[\![h]\!]s]} \tag{3.1}$$

This equation shows how an assignment operation updates the state $s$ by assigning the result of the HE expression $h$ evaluated under $s$ to the variable $v$.

$$\frac{\langle S_1, s\rangle \mapsto s'}{\langle S_1; S_2, s\rangle \mapsto \langle S_2, s'\rangle} \tag{3.2}$$

Here, if the execution of statement $S_1$ in state $s$ leads to a new state $s'$, then the execution continues with the next statement $S_2$ in the new state $s'$.

$$\frac{\langle S, s\rangle \mapsto s' \quad \mathcal{H}[\![\overline{h}]\!]s' = \overline{o}}{\langle S; \overline{h}, s\rangle \mapsto halt(\overline{o})} \tag{3.3}$$

This final rule specifies that if the execution of a sequence of statements $S$ results in a state $s'$, and evaluating the expressions $\overline{h}$ in this state yields outputs $\overline{o}$, then the program halts and outputs $\overline{o}$.

**Operand space:** A constant $c \in$ Vector stands for a constant vector of floating-point numbers, while a variable $x \in$ Var represents a variable name that can hold operands for homomorphic encryption (HE) operations. An operand $o$ is an element of the operand space

Table 3.1: The operational semantics of HE operations reproduced from [29]. Case represents the abbreviation of each operation. Semantics describes how the HE semantics function $\mathcal{H}$ maps an HE expression $h$ and a state $s$ to operand space $\mathcal{O}$. Condition restricts the application of the semantics function $\mathcal{H}$ to satisfy the interface of an HE library.

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

$\mathcal{O}$, which includes real-valued vectors, plaintexts and ciphertexts, each paired with a scale and a level. Specifically, $\mathcal{O}$ is defined as $\mathcal{O} : \mathbb{R}^k \cup \mathcal{P} \times \mathbb{R} \times \mathbb{Z}^+ \cup \mathcal{C} \times \mathbb{R} \times \mathbb{Z}^+$. Here, $\mathbb{R}^n$ refers to vectors of real values where $n$ indicates the number of slots in a packed ciphertext. $\mathcal{P}$ and $\mathcal{C}$ denote the plaintext and ciphertext spaces as defined by the encryption parameters, respectively. Both plaintexts and ciphertexts require a scale $m \in \mathbb{R}$ and a level $l \in \mathbb{Z}^+$, as explained in §2.1. The outcome of any HE operation is also an element of this operand space $\mathcal{O}$.

**HE semantics function:** Table 3.1 outlines the semantics of HE operations. A homomorphic expression ($h$) symbolizes the HE operations. To explain the semantics of HE operations, we establish a state function $s \in \mathcal{S} : \text{Var} \to \mathcal{O}$ and an HE semantics function $\mathcal{H} : h \to \mathcal{S} \to \mathcal{O}$. The function $\mathbb{R}^k[\![c]\!]$ embeds vector representations $c$ into a $k$-dimensional real-valued vector. Additionally, $\mathcal{P}[\![v]\!]$ and $\mathcal{C}[\![v]\!]$ embed a real vector $v$ into a plaintext and ciphertext, respectively.

Although encoding and encrypting operations are foundational for utilizing other HE operations, these operations are typically not visible in the program as encryption should occur prior to program execution and encoding is managed within the `upscale` operation. The scale and level of a resultant ciphertext depend on the operands involved and the operation conducted. Operations like negate and rotate preserve the scale and level of the ciphertext operand, and addition maintains the scale and level of operands, requiring that operands share the same scale and level for the result to remain consistent. For multiplication, the resulting scale equals the product of the operand scales.

`rescale`, `modswitch`, and `downscale` operations adjust the encryption parameters of an operand. A `rescale` operation diminishes the scale of the operand by a rescaling value and decreases the level. A `modswitch` operation simply reduces the level of an operand without altering its scale. A `downscale` operation decreases the scale by an arbitrary amount.

An `upscale` operation functions differently for a ciphertext and a constant vector. For a ciphertext, `upscale` modifies the scale by multiplying it by an identity value encoded with a specified scale. For a constant vector, the `upscale` operation encodes the vector to a plaintext value.

### 3.4 Type Soundness

The type soundness of the HECATE language ensures that a program that is correctly typed can be executed without breaking the rules of homomorphic encryption (HE) operations, as detailed in Table 3.1. This document outlines the principles of type soundness for HECATE language and introduces a concise overview of the proof.

**Definition 1** (Type abstraction)**.** Suppose $\Gamma$ is a typing context and $s$ is a runtime state. Then, $\Gamma$ abstracts $s$ at an initial level $L$, noted as $\Gamma \approx_L s$ for a specified $L$, if the following condition is met:

$$\Gamma \vdash e : \begin{cases} \texttt{real} & \text{if } \mathcal{H}[\![e]\!]s = \mathbb{R}^n[\![v]\!] \\ \texttt{plain}(m, L-l) & \text{if } \mathcal{H}[\![e]\!]s = (\mathcal{P}[\![v]\!], m, l) \\ \texttt{cipher}(m, L-l) & \text{if } \mathcal{H}[\![e]\!]s = (\mathcal{C}[\![v]\!], m, l) \end{cases} \tag{3.4}$$

This definition delineates how static types abstract over runtime properties, ensuring that the type system reflects these runtime characteristics. To adhere to certain constraints, $L$ must be chosen with care, particularly to ensure that $m \times R^d \times |x_i| \leq R^L$, where $x_i = v_i/m$ must hold true, and all values on the left side of this inequality are known at compile time except for $x_i$. The maximum value of $x_i$ must be specified by the programmer to avoid overflow.

**Theorem 2** (Progress)**.** *Assuming any $\Gamma$, $s$, and $L$ where $\Gamma \approx_L s$ is valid, if $\Gamma \vdash S; \overline{e} : T$, then the system will either progress to a new state $\langle S'; \overline{e}, s' \rangle$ or halt with output $\overline{o}$.*

To demonstrate theorem 2, it's necessary to focus solely on the expression assignment case (Equation 3.1), as other statements (Equation 3.2) and function body (Equation 3.3)

behaviors are covered by theorem statement itself. Each expression can be evaluated by $\mathcal{H}$ following the typing rules illustrated in Figure 3.2.

**Theorem 3** (Type Preservation)**.** *Assume any* $\Gamma$*,* $s$*, and* $L$ *where* $\Gamma \approx_L s$ *holds. If the system transitions from* $\langle S_1; S_2; \overline{e}, s\rangle$ *to* $\langle S_2; \overline{e}, s'\rangle$ *and* $\Gamma \vdash S_1 : \Gamma'$*, then* $\Gamma' \approx_L s'$*.*

Theorem 3 can be proven similarly to the theorem 2, utilizing case analysis on the typing rules.

# CHAPTER 4

# PERFORMANCE-AWARE SCALE OPTIMIZATION

This chapter introduces HECATE [28], a sophisticated compiler framework for fully homomorphic encryption (FHE) tailored to precisely adjust the scales of ciphertexts considering their rescaling levels and potential performance impacts. HECATE adopts a comprehensive strategy to refine scale management in ways not previously accomplished. Initially, HECATE introduces a new parameter-switching operation called `downscale`. Uniquely, this operation allows for the rescaling of a ciphertext even if its scale falls below the sum of the rescaling factor and a set threshold, known as the waterline, thus enabling a more proactive rescaling approach.

HECATE further enhances its approach by closely examining the ciphertexts and their related FHE operations, grouping them into scale management units based on shared scales and rescaling levels. This method simplifies the scale management process. Following this, HECATE develops a variety of scale management strategies, carefully evaluating each to determine the most efficient scale management plan. This step goes beyond mere efficiency optimization; it strategically considers the placement of rescaling operations to improve the overall performance and precision of FHE applications.

Additionally, recognizing the importance of adhering to the scale and rescaling level constraints imposed by FHE operations, HECATE incorporates a newly crafted type system, explained in §3. This system is cleverly designed to check that the scales and rescaling levels of FHE operands are compatible, ensuring that scale management complies with essential

computational requirements and upholds the integrity and security of the encrypted data. Through these advancements, HECATE marks a significant advancement in the FHE field, providing a robust framework that elevates the scalability, performance, and practicality of privacy-preserving computations.

### 4.1 Necessity of Performance-aware Scale Optimization

The existing scale management compiler, EVA [26], manages scale to meet the three constraints described below.

- (C1) The scale of a ciphertext must be smaller than the coefficient modulus $Q$;
- (C2) The scale must be greater than the waterline to prevent message corruption; and
- (C3) The level of operands for multiplications and additions must match. (The addition must match the scale of operands additionally.)

EVA's scale management approach has three main drawbacks that prevent it from achieving the best possible performance for fully homomorphic encryption (FHE) programs.

The first limitation is that EVA's rescaling method is reactive and only uses a fixed scaling factor. In EVA, a rescale operation is only added when the resulting scale after rescaling is still greater than a predefined value called the waterline. This means that EVA only reduces the scale after a multiplication operation has already increased it. Additionally, EVA's rescale operation can only decrease the scale by a fixed amount, known as the scaling factor. As a result, EVA is limited to inserting the rescale operation only when the scale exceeds the product of the scaling factor and the waterline. This approach misses out on chances to optimize the scale proactively. Later on, we will introduce a new rescaling operation that can change the scale by any desired amount, allowing for more effective scale management.

The second limitation is that EVA analyzes the scale and level separately. To satisfy both the scale-related constraints (C1 and C2) and the level-related constraint (C3), EVA uses a

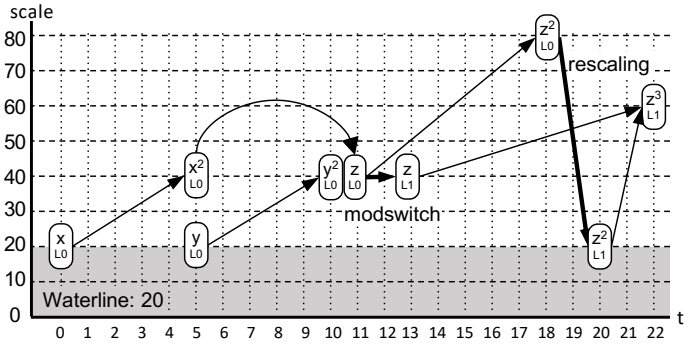
two-step process. First, it performs waterline rescaling to add rescale operations based on the scale. Then, using the code that now includes the rescale operations, EVA adds modswitch operations where necessary to handle the level constraint. In simpler terms, EVA first focuses on getting the scale right and then separately adjusts the level to meet the level requirement. We will demonstrate later that looking at both the scale and level together creates new opportunities for better scale management, allowing an FHE compiler to find more optimal solutions.

The third limitation is that EVA's scale management doesn't take performance into account. EVA assumes that the fixed-factor rescale operation is the only way to manage the scale, and it believes that a lower scale always leads to better performance. Because of this, EVA puts all its effort into lowering the scale. However, we will show that when more flexible rescaling options are available, a lower scale doesn't necessarily mean better performance. In fact, the cost of an FHE operation goes down as the rescaling level goes up. For example, multiplication at level 1 is 4 times faster than at level 0. To optimize performance, an FHE compiler should think about alternative scale management strategies that might result in a higher scale but allow more of the computations to happen at higher levels, which are faster.

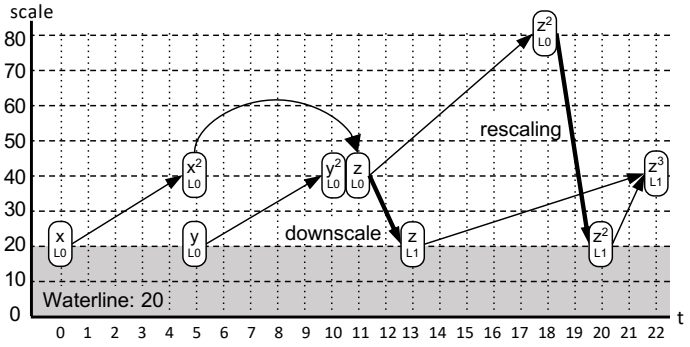
### 4.2 Overview of Performance-aware Scale Optimization

This work introduces HECATE, a new compiler for fully homomorphic encryption (FHE) tailored for RNS-CKKS applications, focusing on advanced scale management strategies. HECATE enhances performance-aware scale management through several innovative approaches:

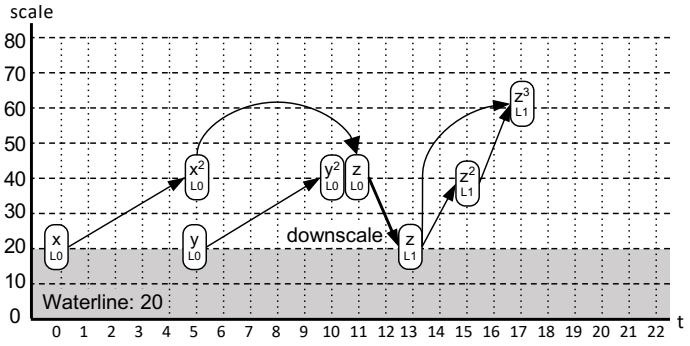
(1) It introduces a new scale management operator called `downscale` along with a proactive rescaling scheme. This operator allows for the adjustment of a ciphertext's scale to a specific target, known as the waterline, thereby enabling more deliberate control over scaling




(a) Existing HE compiler (EVA)




(b) Proactive Rescaling (PARS)




(c) Scale management space explorer (SMSE)

Figure 4.1: Comparison of how EVA and HECATE manage scale for a sample program that computes $(x^2 + y^2)^3$, an operation involved in calculating the root mean square, reproduced from [28]. The `rescale` operation decreases the scale by $2^{60}$ and raises the rescaling level by one. The `modswitch` operation simply raises the level, and the `downscale` operation lowers the scale to the waterline and also increases its level.

operations. (2) It develops a new type system that simultaneously analyzes both the scale and rescaling level of ciphertexts, facilitating more integrated and efficient scale management. (3) It establishes a method for exploring different scale management strategies, known as Scale Management Space Exploration (SMSE), which includes performance estimation to identify the most effective approaches.

**Solution 1: Proactive Flexible-Factor Scale Management.** HECATE introduces the `downscale` operator, which uniquely adjusts the scale of a ciphertext by a specific amount, rather than incrementally. This operation is detailed in the semantics shown in Table 3.1, where it reduces the ciphertext's scale precisely to the waterline $W$ and increments the rescaling level by one. This proactive approach allows HECATE to adjust the scale before operations like multiplication, which contrasts with EVA's reactive method which scales down only after multiplication has increased the scale excessively. For example, as shown in Figure 4.1b, HECATE can apply `downscale` to adjust the scale of $z$ to the waterline before multiplication, resulting in a final scale for $z^3$ that is significantly lower than what is achieved with EVA's approach.

**Solution 2: A New Type System for Scale and Level Management.** HECATE proposes a novel type system that considers the scale and level of ciphertexts together, enabling more strategic scale management. This integrated approach differs from EVA's, which typically handles scale adjustments first and then addresses any resulting level discrepancies. This holistic view allows HECATE to explore various scale management possibilities more effectively.

For instance, in the computation of $(x^2 + y^2)^3$, Figure 4.1c illustrates another strategy where `downscale` is applied even before the initial multiplication of $z*z*z$, where $z = (x^2+y^2)$. This and other scale management strategies demonstrated in Figure 4.1 show that different uses of `rescale`, `modswitch`, and `downscale` can lead to varied cumulative scales and

potentially impact performance. The type system aids HECATE in safely navigating the scale management landscape while adhering to RNS-CKKS constraints.

**Solution 3: Scale Management Space Exploration (SMSE) with Performance Estimation.** HECATE also innovatively explores the scale management landscape using a hill-climbing method to optimize performance. It performs static analysis to determine scale management units—clusters of operations where scale and level are managed collectively, minimizing the need for mid-sequence adjustments. HECATE iteratively constructs scale management plans, adjusting one parameter at a time from the previous plan, and integrates scale management operators into FHE codes that satisfy the constraints (C1-C3). It then evaluates the performance of each plan, selecting the most effective for further refinement.

Figure 4.1c displays the scale management plan predicted to perform optimally. Notably, although plan (c) has a higher cumulative scale than plan (b), the early use of `downscale` permits subsequent multiplications to occur at a consistent level, enhancing overall performance.

**Overview.** Figure 4.2 illustrates the design of the HECATE framework. HECATE is built on top of MLIR to ensure it can be extended to support a variety of frontends (such as ONNX-MLIR, NumPy) and backends (beyond SEAL) in future developments. HECATE offers a Python frontend, making it simpler to write FHE programs. The HECATE intermediate representation (IR) generator then converts programs written in this Python frontend into HECATE language, as discussed in §3. Following this, HECATE organizes the program into scale management units as outlined in §4.3.

To navigate the scale management space, the planner (§4.4.1) creates a series of new plans based on the most effective plan from the previous cycle. With these plans, HECATE then produces accurate and efficient FHE codes that adhere to RNS-CKKS constraints (§4.5). Additionally, the performance estimator (§4.4.2) evaluates the cost of these codes to inform the next iteration of planning. For the final output, HECATE includes a SEAL dialect, which

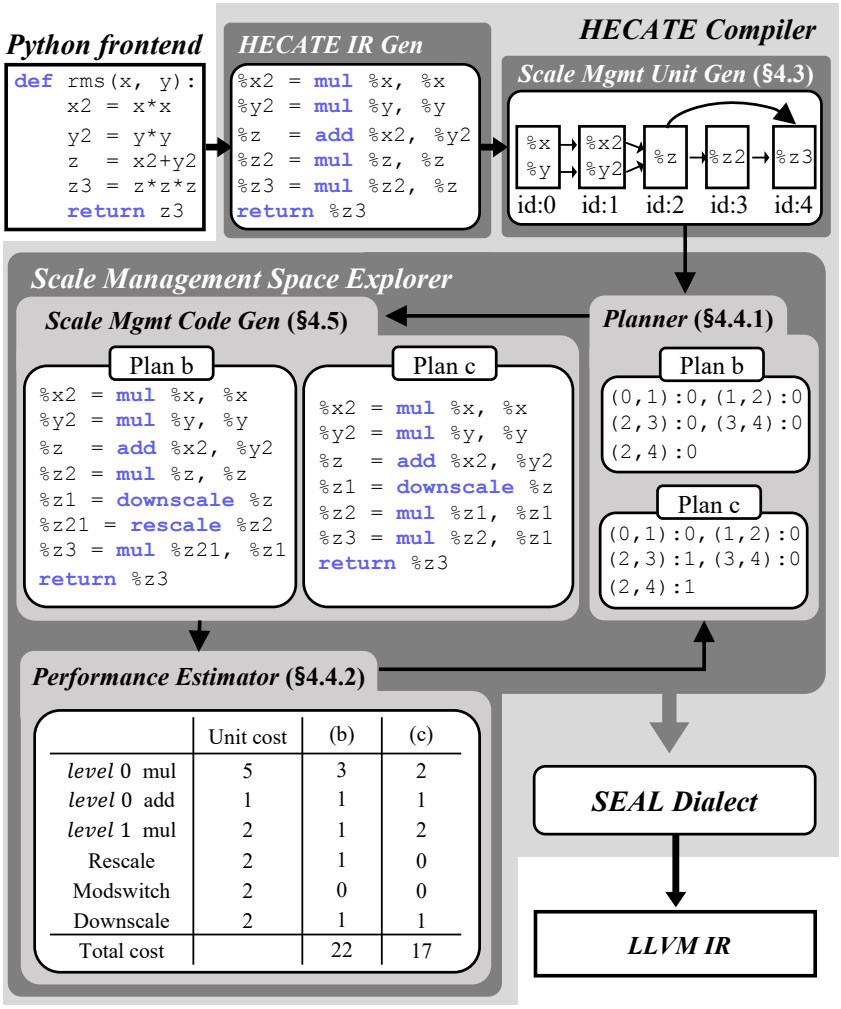



Figure 4.2: Design of the HECATE compiler framework reproduced from [28]. The example code uses the program in Figure 4.1. Plan (b) and (c) are the same with Figures 4.1b and 4.1c.

serves as the backend for the FHE program and enhances memory efficiency by analyzing data usage.

### 4.3 Scale Management Unit Generation

For scale management space exploration, HECATE examines a HECATE language program and creates scale management units where the scale and level of the data can be managed together, simplifying the search space. algorithm 1 details HECATE's three-phase algorithm for analyzing scale management units using a custom *Group* data structure. The *Group.insert(res)* function adds and returns a new set which can be accessed via "*res*." *Group[v]* locates a set associated with a value *v*. *Group.merge(A, B)* combines sets *A* and *B* but retains the key of *A*. *Group.split(A, v)* separates and returns *v* from *A*.

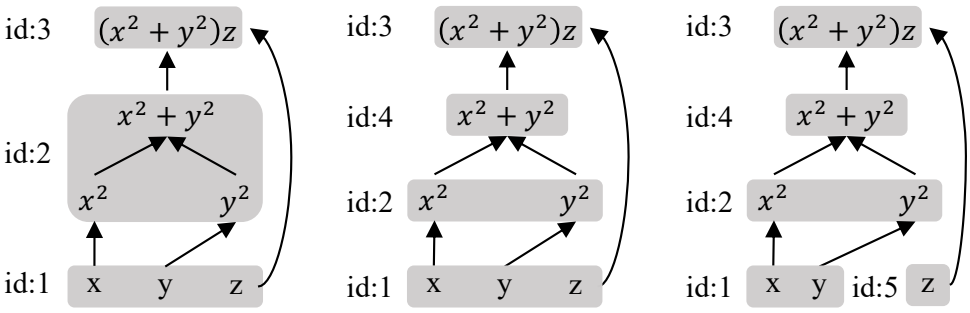
The initial phase, known as the definition-aware merge step (Line 3-13), conducts a forward program analysis, grouping values that share the same scale and level into the same unit. For instance, a plaintext addition ($+_p$) does not modify the scale and level of a ciphertext. Similarly, a ciphertext addition ($+_c$) maintains the existing scale and level, provided the operands are identical in these aspects. In such cases (Line 6-7), HECATE places the resulting plaintext/ciphertext and its operands in the the same unit.

Conversely, for scenarios like a ciphertext addition where the operands have differing scales/levels necessitating a scale management operation, or a ciphertext multiplication that alters the resulting ciphertext's scale, HECATE sets up a new scale management unit unless an existing case matches the sthe ame operator and operand group combination (Line 8-12).

For example, as depicted in Figure 4.3a, the definition-aware merge step groups ($x^2$, $y^2$, and $x^2 + y^2$) with the the same scale/level into a single unit. The subsequent operation $(x^2 + y^2)z$ causes a scale increase, necessitating a separate unit.

**Algorithm 1:** Scale management unit analysis [28]

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




(a) Definition-aware merge (b) Operation-aware split (c) User-aware split

Figure 4.3: Scale management unit analysis example for $(x^2 + y^2)z$, reproduced from [28].

The second step (Line 14-23) further divides the scale management units created in the first step into smaller groups based on the type of operation (refer to Line 18, which uses $op$ as a key). This phase specifically aims to separate operations involving multiplication from those that do not. The logic behind this separation is that multiplication operations generally result in a scale significantly larger than $W^2$, providing an opportunity for proactive scale management. Since the scale of the multiplication operands is larger than $W$, the resulting scale of these operations will consistently exceed $W \times W$.

Using the same example shown in Figure 4.3a, where the first two operations are multiplications ($\times$) and the third is an addition ($+$), the operation-aware split step divides the unit containing ($x^2$, $y^2$, and $x^2 + y^2$) into two separate units: one for ($x^2$, $y^2$) and another for ($x^2 + y^2$), as illustrated in Figure 4.3b.

The final user-aware split step (Line 24-39) further segments the scale management units. This stage conducts a backward analysis to determine how each ciphertext is "used" in the program and assigns ciphertexts that are utilized differently to distinct units. This analysis helps to tailor scale management strategies more effectively based on the specific application of each ciphertext.

In the ongoing example from Figure 4.3b, the variables $x$ and $y$ are used in the operations $x^2$ and $y^2$ respectively, while $z$ is used in the operation $(x^2 + y^2)z$. The third step splits these into separate units: one containing $(x, y)$ and another for $(z)$. Figure 4.3c displays the final arrangement of scale management units.

### 4.4 Scale Management Space Explorer

As shown Fig. 3, HECATE's Scale Management Space Explorer (SMSE) consists of three components: scale management planner (§4.4.1), code generator, and performance estimator (§4.4.2). The detailed algorithm of the code generator will be further discussed in §4.5

### 4.4.1 Scale Management Planner

The planner takes as input scale management units (see §4.3) and the best plan from the previous iteration, then generates a set of new scale management plans. Each plan assigns a degree of optimization to connections between scale management units, representing the number of additional scale management operations such as `rescale`, `downscale`, and `modswitch`.

The planner employs the steepest ascent hill-climbing method to create new scale management plans. Starting with the best plan from the last iteration, the planner creates new plans by increasing the optimization degree by one at different points. For example, if a program is divided into three scale management units—0, 1, and 2—with edges (0,1) and (1,2), and the best existing plan is {(0, 1): 1, (1, 2): 0}, the new plans might be {(0, 1): 2, (1, 2): 0} and {(0, 1): 1, (1, 2): 1}. Essentially, the planner tries to add more scale management operations than in the previous iteration.

Each plan specifies (a) where to place scale management operations and (b) how many operations to place at each location. Based on the scale of an operand, the planner determines (c) which type of scale management operation to apply. If the scale is greater than the waterline after `rescale`, the planner opts for `rescale`. If `rescale` is not suitable but the scale can still be reduced, `downscale` is applied. Otherwise, `modswitch` is chosen. This strategy relies on the operand's scale information and must adhere to RNS-CKKS constraints, so the planner uses the proactive rescaling algorithm (PARS, algorithm 2), detailed in the subsequent section, to ensure the generated FHE codes are correct.

After the code generation step, the latency of a generated code is estimated by a performance estimator. The estimated latency is used to the "steepest" ascent plan which means the plan with minimal latency among the candidate plans generated in the the same

iteration. The plan with minimal latency will be used to generate the new plans in the next iteration.

### 4.4.2 Performance Estimator

The performance estimator statically calculates the expected execution time of a HECATE language program. Running all candidate programs dynamically would be prohibitively expensive. The latency of an FHE operation in RNS-CKKS depends on the level of the operands and the polynomial modulus $N$, with the time complexity being linear or quadratic to the level of the operands, and linear or log-linear to $N$, varying by operation type. Using this insight, we profile the execution time of each FHE operation at different levels and $N$, as shown in Figure 4.2. With this profiled per-level latency data for each FHE operation, the performance estimator can predict the total execution time of a HECATE program, leveraging the level information readily provided by HECATE's type system.

## 4.5 Code Generation: Proactive Rescaling

The scale management code generation of HECATE, called proactive rescaling (PARS), is done by term rewriting method based on rewriting rules presented in Figure 4.4. This section depicts how the concrete code generation algorithm applies the rewriting rules.

Algorithm 2 outlines how HECATE integrates scale management operations to efficiently manage the cumulative scale of each ciphertext. HECATE aims to minimize the overall scale of the operands involved in each operation, thereby reducing the resultant cumulative scale. The algorithm encompasses five phases: (a) encoding, (b) `rescale` analysis, (c) level matching, (d) scale matching, and (e) `downscale` analysis. Additionally, HECATE incorporates an early `modswitch` optimization technique similar to what is used in EVA, positioning `modswitch` earlier in the process.

**Algorithm 2:** Proactive rescaling algorithm (PARS) [28]

**Parameter :** $R$: Rescale Value
**Parameter :** $W$: Waterline Value
**Input:** *Op*: The operation type of HE operation.
**Input:** *arg0, arg1*: Argument ciphertext of an HE operation.
**Input:** *res*: Result ciphertext of an HE operation.

**Function** *PARS (op, arg0, arg1, res) :*
    // (a) Encode
    **if** *op* $\in \{+, \times\}$ **then**
        // Without Loss of Generality
        **if** *arg0.type* = `real` **then**
            *arg0.type* $\leftarrow$ `plain` *(arg0,* $W$*)*
    // (b) Rescale Analysis
    // Without Loss of Generality
    **if** *arg0.scale* $> W \cdot R$ **then**
        *arg0* $\leftarrow$ `rescale` *(arg0)*
    // (c) Level Match
    // Without Loss of Generality
    **if** *op* $\in \{+, \times\} \wedge$ *arg0.level* $<$ *arg1.level* **then**
        **if** *arg0.scale* $= W$ **then**
            *arg0* $\leftarrow$ `modswitch` *(arg0)*
        **else if** *arg0.scale* $> W$ **then**
            *arg0* $\leftarrow$ `downscale` *(arg0)*
    // (d) Scale Match
    // Without Loss of Generality
    **if** *op* $\in \{+\} \wedge$ *arg0.scale* $<$ *arg1.scale* **then**
        *arg0* $\leftarrow$ `upscale` *(arg0, arg1.scale)*
    // (e) Downscale Analysis
    **if** *op* $\in \{\times\} \wedge$ *arg0.scale* * *arg1.scale* $> W^2 \cdot R$ **then**
        *arg0* $\leftarrow$ `downscale` *(arg0)*
        *arg1* $\leftarrow$ `downscale` *(arg1)*
**end**

$$\frac{\Gamma \vdash h : \texttt{scale}(m,d) \quad \Gamma \vdash h' : \texttt{scale}(m',d) \quad m \cdot m' < W^2 \cdot R}{h \times h' \xrightarrow{rewrite} \texttt{downscale}(h) \times \texttt{downscale}(h')} \quad \text{(DScale)}$$

$$\frac{\Gamma \vdash h : \texttt{scale}(m,d) \quad m \geq W \cdot R}{h \xrightarrow{rewrite} \texttt{rescale}(h)} \quad \text{(Rescale)}$$

$$\frac{\Gamma \vdash e : \texttt{scale}(m,d) \quad \Gamma \vdash e' : \texttt{scale}(m',d') \quad m > W \quad d < d'}{e \oplus e' \xrightarrow{rewrite} \texttt{downscale}(e) \oplus e'} \quad \text{(DMatch)}$$

$$\frac{\Gamma \vdash h : \texttt{cipher}(m,d) \quad \Gamma \vdash h' : \texttt{real}}{h + h' \xrightarrow{rewrite} h + \texttt{upscale}(h', m)} \quad \text{(EncodeAdd)}$$

$$\frac{\Gamma \vdash e : \texttt{scale}(m,d) \quad \Gamma \vdash e' : \texttt{scale}(m',d') \quad m = W \quad d < d'}{e \oplus e' \xrightarrow{rewrite} \texttt{modswitch}(e) \oplus e'} \quad \text{(LMatch)}$$

$$\frac{\Gamma \vdash h : \texttt{cipher}(m,d) \quad \Gamma \vdash h' : \texttt{real}}{h \times h' \xrightarrow{rewrite} h \times \texttt{upscale}(h', W)} \quad \text{(EncodeMul)}$$

$$\frac{\Gamma \vdash h : \texttt{scale}(m,d) \quad \Gamma \vdash h' : \texttt{scale}(m',d) \quad m < m'}{h + h' \xrightarrow{rewrite} \texttt{upscale}(h, m'/m) + h'} \quad \text{(SMatch)}$$

Figure 4.4: Rewriting rules for proactive rescaling. `scale` includes `cipher` and `plain` type.

**(a) Encode.** This initial step sets the scale for operands of the `real` type within binary expressions. If an operand is `real`-type, HECATE converts it to `plain`-type with a scale set at the waterline $W$.

**(b) Rescale analysis.** This phase adjusts the scale of operands if the resultant scale remains above the waterline. Specifically, the `rescale` operation reduces the scale from $m$ to $m/R$, given the rescale factor $R$. Thus, if $m > W \times R$, HECATE applies `rescale` to lower the operand's scale in binary expressions, ensuring the minimized scale still respects the waterline constraint.

**(c) Level match.** This step ensures that both operands in binary expressions have the same level, as required. If the scale of the operand with the smaller level equals the waterline, HECATE employs `modswitch` to elevate the operand's rescaling level, mimicking EVA's approach. If the scale is also above the rescale factor $R$, and if the smaller level's scale is

below the waterline, HECATE opts for `downscale` to decrease the scale while raising the rescaling level.

**(d) Scale match.** This phase addresses the scale requirements for addition operations, ensuring that both operands share the same scale. Here, HECATE applies an `upscale` operation to the operand with the lesser scale to equalize them.

**(e) Downscale analysis.** The final step determines whether applying `downscale` to both operands of a multiplication could be advantageous. After the preceding steps, suppose two operands are of types $\texttt{scale}(m, d)$ and $\texttt{scale}(m', d)$. One approach is to perform the multiplication first and then apply `downscale`, resulting in a type $(mm'/R, d+1)$. Alternatively, applying `downscale` to both operands before multiplication could adjust their types to $(W, d+1)$ each, and post-multiplication, the resultant type would be $(W^2, d+1)$. HECATE chooses to apply `downscale` if $mm'/R > W^2$, optimizing the operation's outcome.

## 4.6 Evaluation of Performance-aware Scale Optimization

### 4.6.1 Experimental Setup

For performance evaluation, this study compares HECATE with the state-of-the-art EVA [26] using various benchmarks. HECATE incorporates proactive rescaling (PARS, §4.5) and scale management space exploration (SMSE, §4.4). This research also assesses the individual benefits of these components. Overall, this study explores four different scale management strategies:

- **EVA** [26] implements fixed-factor scale management and the *waterline rescaling* algorithm, both of which are reconfigured within the HECATE framework.

- **PARS** employs the proactive rescaling scheme but does not include scale management space exploration.

- **SMSE** focuses on scale management space exploration but does not incorporate proactive rescaling. Instead, it adopts EVA's waterline rescaling algorithm.

- **HECATE** [28] integrates both proactive rescaling and scale management space exploration for comprehensive scale management.

For benchmarking, the following six applications were implemented and tested, using the same benchmark set as EVA and CHET, except for SqueezeNet, and including an additional test on MLP:

- **Sobel Filter (SF)** is a traditional edge detection algorithm that computes changes in an image across vertical and horizontal directions using $3 \times 3$ image gradient filters.

- **Harris Corner Detection (HCD)** identifies corner points in an image by calculating pixel differences within a window.

- **Linear Regression (LR)** fits a linear equation to model the relationship between a dependent variable and one or more independent variables.

- **Polynomial Regression (PR)** models nonlinear relationships using an $n$-th degree polynomial equation, specifically employing a quadratic equation for this study.

- **Multi-layer Perceptron (MLP)** is a feed-forward neural network designed for image classification, utilizing layers sized $784\times100$ and $100\times10$ with square activation functions.

Table 4.1: RMS Error of the programs reproduced from [28]

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

- **LeNet-5** [77] is a convolutional neural network tailored for image classification, modified from the original network to change the output channel of the second fully connected layer to 64 and using a square function for activation.

Experiments were conducted using Microsoft SEAL (Release 3.5.9) on an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz with 6 physical cores and 64GB RAM, maintaining a 128-bit security level for all tests. Benchmarking assumed a packed ciphertext with $2^{14}$ slots. The image processing benchmarks processed 4096 pixels from $64 \times 64$ images, regression benchmarks utilized 16384 randomly generated inputs for each variable, and the deep learning benchmarks used a random input from the MNIST dataset. The regression benchmarks employed the gradient descent algorithm for 2 and 3 epochs.

### 4.6.2 Performance Evaluation

Figure 4.5 displays the lowest latency achieved for each benchmark while adhering to a maximum error limit of $2^{-8}$. For all the evaluated schemes, this study tested 36 different waterline settings and identified the optimal one in terms of latency that still maintained the error within the set bounds. The precise root-mean-square error of the compiled programs

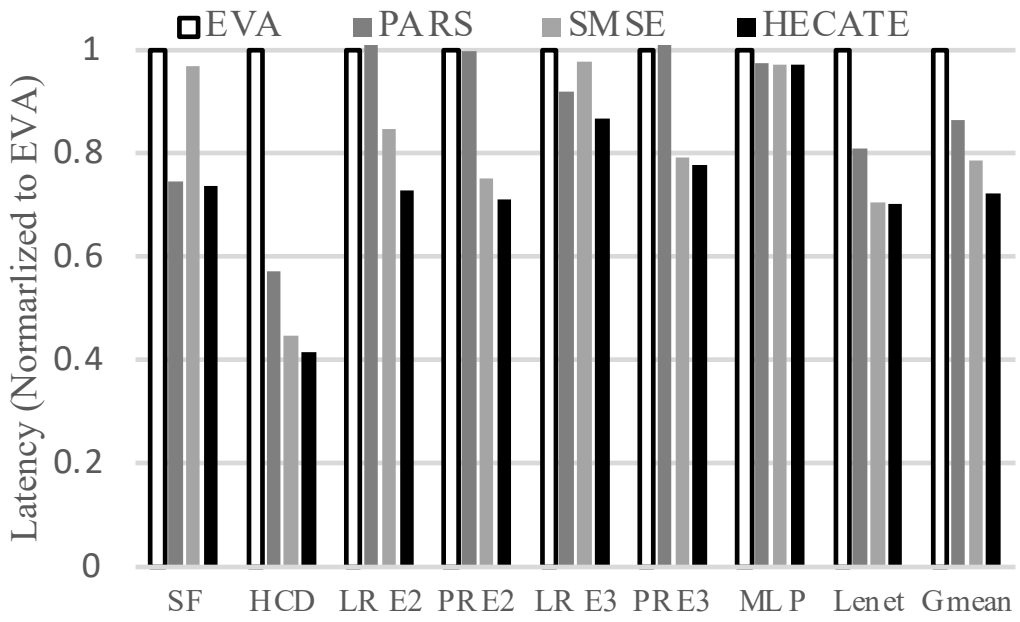



Figure 4.5: Performance of 8 benchmarks with different scale management schemes: Sobel Filter, Harris Corner Detection, Linear Regression and Polynomial Regression, Multi-layer Perceptron, LeNet-5, reproduced from [28]. This work executes regression benchmarks with two and three epochs (denoted as E2 and E3).

is detailed in Table 4.1. It's important to note that a smaller error does not necessarily indicate better exploration; it might simply mean that no better optimization that compromises precision for improved performance was found.

The evaluation results demonstrate that HECATE significantly boosts the performance of various homomorphic encryption (HE) applications through the use of PARS and SMSE. On average, performance gains from PARS and SMSE are 13.38% and 21.36%, respectively, with HECATE achieving an overall average performance improvement of 27.85%.

**Speedup of PARS.** While PARS generally results in a smaller cumulative scale, which sets the initial level of the program, it does not always lead to speed improvements over EVA. For benchmarks like SF, HCD, and LeNet, PARS enhances performance by reducing the

cumulative scales and thus the initial levels of the ciphertexts. However, for MLP, LR E2 and E3, and PR E2 and E3, PARS does not improve performance because the reduction in cumulative scale does not significantly lower the necessary level, leaving the initial levels of the ciphertexts unchanged. On average, PARS shows a 13.38% speed improvement over EVA.

**Speedup of SMSE.** SMSE consistently shows improvements over EVA. This is because SMSE discards any optimization plan that results in estimated latencies slower than the previous plan, ensuring that any change leads to a performance gain if the estimations are accurate. As detailed in §4.6.4, the estimations are sufficiently precise, explaining the 21.35% average speedup achieved by SMSE.

However, SMSE's impact is minimal for benchmarks like SF, MLP, and LR E3. Similar outcomes in other schemes for MLP and LR E3 suggest that EVA may already optimize well for these applications. For SF, the limited number of scale management unit (SMU) edges constrains the scope for scale management. Thus, the effectiveness of SMSE largely depends on how it integrates with code generation strategies such as PARS.

**Speedup of HECATE.** HECATE outperforms other optimization approaches. For benchmarks such as SF, HCD, and LR E3, HECATE significantly enhances performance by simultaneously utilizing SMSE and PARS. The results for LR E2, PR E2, and PR E3 illustrate how the approach to code generation influences HECATE's speedup. While using PARS alone does not enhance performance for these benchmarks, HECATE achieves better latency than SMSE with waterline rescaling because the code generation strategy affects the effectiveness of the optimization plan. The performance of LeNet highlights the influence of SMSE. Not surprisingly, SMSE can effectively position the `downscale` operation and may target the same optimization points as PARS. Consequently, even with waterline rescaling, SMSE is capable of discovering a scale management plan comparable to HECATE. Overall, HECATE delivers the highest average performance speedup, at 27.85%.

Table 4.2: Search space reduction reproduced from [28]. (uses: # of uses, SMU: # of scale management units, epoch: # of exploration iterations, plan: # of explored spaces)

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

HECATE secures significant performance improvements through compiler optimizations alone, without the need for additional hardware or changes to algorithms. These optimizations can synergistically enhance the performance of modern hardware acceleration approaches like HEAX [78], further boosting their efficiency.

### 4.6.3 Search Space Reduction

This section illustrates how generating scale management units (SMUs) can significantly narrow down the search space for optimization. Without SMUs, every ciphertext uses within a program would require individual optimization, vastly increasing the number of potential plans. This expansion in the number of plans is exacerbated not only by the number of connections (edges) in the program but also by each iteration (epoch) of the hill-climbing algorithm used for optimization.

To assess how effectively SMU generation reduces the exploration space of the Scale Management Space Exploration (SMSE), this study implemented a basic exploration strategy.

This naive approach utilizes the same hill-climbing algorithm but applies it directly to the use-definition (use-def) edges within the program, without forming any SMUs.

Table 4.2 presents the comparative results between the naive scheme and HECATE. For benchmarks like Sobel Filter (SF) and Multi-layer Perceptron (MLP), utilizing SMUs does not decrease the number of epochs, indicating that the optimized plans affect only a single-use SMU edge. However, these benchmarks benefit from having fewer units to manage. In regression benchmarks such as Linear Regression (LR) E2, LR E3, Polynomial Regression (PR) E2, and PR E3, the generation of SMUs by HECATE slightly reduces the number of epochs required. These benchmarks feature several parallel operations that can utilize the same scheduling, resulting in a modest reduction in epochs.

For benchmarks like LeNet and Harris Corner Detection (HCD), the reduction in epochs is significant due to the high number of parallel operations these programs exhibit. The LeNet results demonstrate that SMUs scale well with program size, showing that the compilation time is influenced by the number of iterations during SMSE. Thanks to the search space reduction (Table 4.2), the longest compilation time for HECATE is only 340 seconds, compared to the 649 hours required by the naive scheme.

#### 4.6.4 Performance Estimation

Performance estimation significantly influences the effectiveness of Scale Management Space Exploration (SMSE). As demonstrated in Figure 4.6, the estimated latencies for various configurations closely match the actual latencies of the compiled programs. The geometric average of the relative error is 1.3%, with the maximum error reaching 4.8%. These findings confirm that the straightforward estimation method outlined in §4.4.2 is adequate for SMSE.

The consistency in latency estimation is attributed to the inherent characteristics of homomorphic encryption (HE) operations. These operations typically involve executing

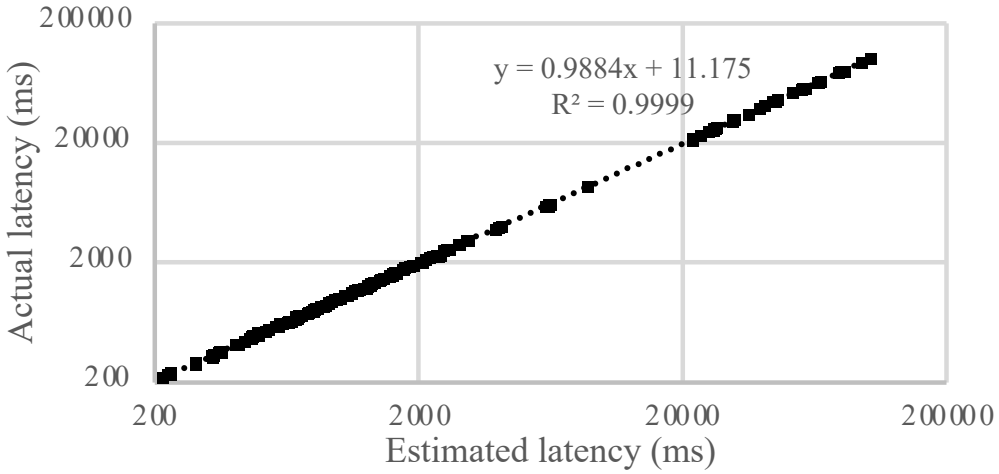



Figure 4.6: Comparison between estimated and actual latencies reproduced from [28]. The comparison plots the data of 1152 different settings that uses 36 different waterlines for 8 benchmarks and 4 optimization schemes. The maximum relative error is 4.8%

lengthy and regular computational loops, which minimize the variance in latency per iteration through statistical effects. This attribute of HE operations underpins why the estimation method can so accurately predict latency.

### 4.7 Summary

This work introduces the HECATE compiler framework, which innovatively enhances performance-aware scale management for fully homomorphic encryption (FHE) applications. The core of HECATE is its robust type system tailored to FHE, complemented by advanced scale management operations that include a newly developed `downscale` operation. This operation is part of a broader strategy encompassing a proactive rescaling algorithm and a comprehensive scale management space explorer, which together seek to optimize computational efficiency dynamically.

The proactive rescaling algorithm within HECATE intelligently adjusts the scale of ciphertexts prior to performing operations, significantly reducing unnecessary computational overhead and enhancing the overall execution speed. Furthermore, the scale management space explorer systematically evaluates various scale management strategies, enabling the compiler to select the most efficient pathway through potential scale configurations.

By integrating these elements, HECATE not only advances the state-of-the-art in compiler design for FHE but also demonstrates a notable performance improvement. Specifically, HECATE achieves a 27.38% speedup over existing methods, showcasing its effectiveness in optimizing scale management for enhanced computational efficiency in FHE operations. This combination of a new type system, innovative scale management operations, and strategic exploratory algorithms positions HECATE as a pivotal development in the field of cryptographic computation.

# CHAPTER 5

# ERROR-LATENCY-AWARE SCALE MANAGEMENT

This chapter introduces ELASM [29], a new error-latency-aware scale management compiler that introduces three innovative solutions that enhance the ability to balance error and latency in RNS-CKKS computations. First, it presents the first error estimation model tailored for RNS-CKKS, designed to accurately predict the discrepancy between results computed on plaintext and those computed on fully homomorphic encrypted data. Since each RNS-CKKS operation introduces a specific noise based on the operation type and rescaling level of its operands, this model projects the resultant error by analyzing the noise introduced at each step, factoring in the scale of the ciphertext and the cumulative impact along the data flow.

Second, this study introduces a new scheme called *Error-Latency-Aware Scale Management (ELASM)*, which strategically searches for the optimal scale management plan that minimizes a cost function defined by error and latency parameters. ELASM starts by generating various scale management plans, each incorporating different scale management operations. It then uses the previously mentioned error estimation model to evaluate the potential error and accumulates the latencies of each RNS-CKKS operation to assess overall performance. With these evaluations, ELASM calculates the cost function and selects the plan that offers the lowest cost. Notably, unlike previous methods, ELASM may opt to *increase* the scale of a ciphertext if the anticipated reduction in cost justifies such an action.

Third, the work proposes a new parameter called the *scale-to-noise ratio* (SNR) and introduces precise, noise-aware waterlines for various RNS-CKKS operations. The SNR

parameter allows users to set a threshold ensuring that the scale $m$ of the ciphertext relative to the noise $n$ introduced by any RNS-CKKS operation remains above a specified value ($m/n \geq SNR$), similar to a traditional signal-to-noise ratio. With this SNR in place, ELASM can assign specific waterlines for different operations that introduce noise, such as `rescale` and `rotate`, thereby facilitating improved trade-offs between error and latency. The proposed techniques are implemented as a new optimization pass on the top of HECATE compiler.

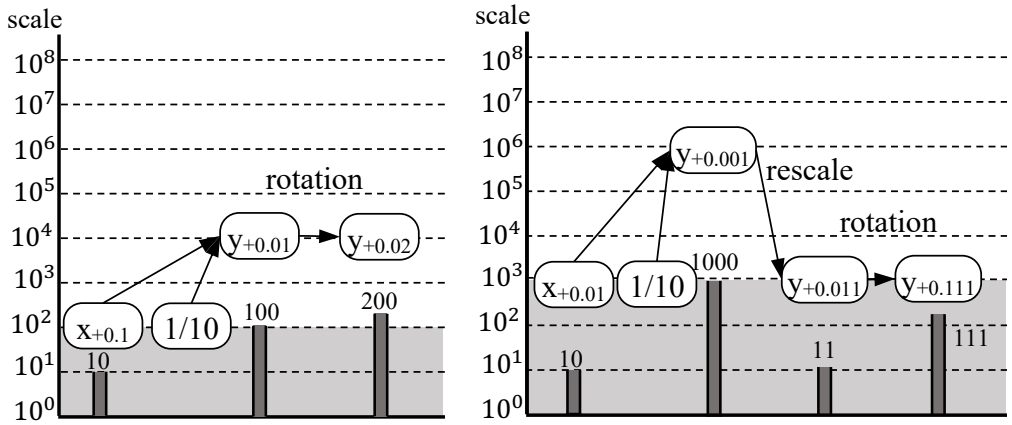
### 5.1 Necessity of Error-Latency-Aware Scale Management

This section evaluates existing scale management schemes and their limitations, underlining the need for a new error-aware scale management approach. The scale management strategy employed in EVA focuses on maintaining a minimal scale by tracking scale growth and implementing the `rescale` operation when the scale post-rescaling remains above a coarse-grained waterline, $W$. The waterline is typically set as the maximum scale of input ciphertexts, with users having the ability to adjust the input scale to modify the waterline accordingly. HECATE introduces an enhanced rescaling method named `downscale`, which allows for reducing the scale by arbitrary amounts, yet it adheres to the same waterline constraint.

Figures 5.1a and 5.1b demonstrate how EVA (and HECATE) function when computing `rotate`$(0.1x)$, a part of convolution, with varying input scales (waterlines) $10^2$ and $10^3$, and a rescaling factor $R = 10^3$. For a waterline of $10^2$ (Figure 5.1a), EVA does not add `rescale` as the scale after rescaling is smaller than the waterline: i.e., $10^4/10^3 < 100$. Conversely, for a waterline of $10^3$ (Figure 5.1b), EVA incorporates `rescale` to lower the scale from $10^6$ to $10^3$ between multiplication and rotation.

The limitations of current scale management schemes are outlined as follows:

**Limitation 1: Current solutions lack error control during scale management.** Increasing the waterline does not necessarily result in reduced error. For example, using the `rotate`$(0.1x)$




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

Figure 5.1: EVA scale management for handling parts of a convolution process that compute `rotate`$(0.1x)$, with various waterline settings, reproduced from [29]. In the diagrams, a gray bar and a subscript illustrate the accumulated noise and the corresponding error in a ciphertext. A `rescale` operation within this context reduces both the scale and the encrypted value by a factor of 1000. Additionally, the noise contributions from the `rescale` and `rotate` operations are quantified as 10 and 100, respectively.

scenario in Figure 5.1a (input scale = $10^2$), where a bar graph denotes noise and a subscript indicates error, consider an initial noise of 10. With an input scale of $10^2$, the error for the initial ciphertext $x$ is 0.1 (see the subscript $+0.1$ next to $x$). Multiplication $y = x \cdot 1/10$ raises $y$'s scale to $10^4$ ($= 10^2 \cdot 10^2$). The resultant noise after multiplication is calculated as $m_1x_1n_2 + m_2x_2n_1 + n_1n_2 + n_{relinearize}$, where $n_2$ and $n_{relinearize}$ are 0 since $1/10$ is plaintext. Thus, the noise becomes 100 ($= 10^2 \cdot 10 \cdot 1/10$) and the error is 0.01 ($= 100/10^4$). The `rotate` operation then adds an additional noise of $n_{rotate} = 100$, bringing the total noise and error to 200 and 0.02 ($= 200/10^4$), respectively.

Figure 5.1b (input scale = $10^3$) illustrates that a higher input scale can paradoxically lead to a higher error. After multiplication, the scale of $y$ soars to $10^6$ ($= 10^3 \cdot 10^3$), the noise increases to 1000 ($= 10^3 \cdot 10 \cdot 1/10$), and the error is 0.001 ($= 1000/10^6$). EVA then applies

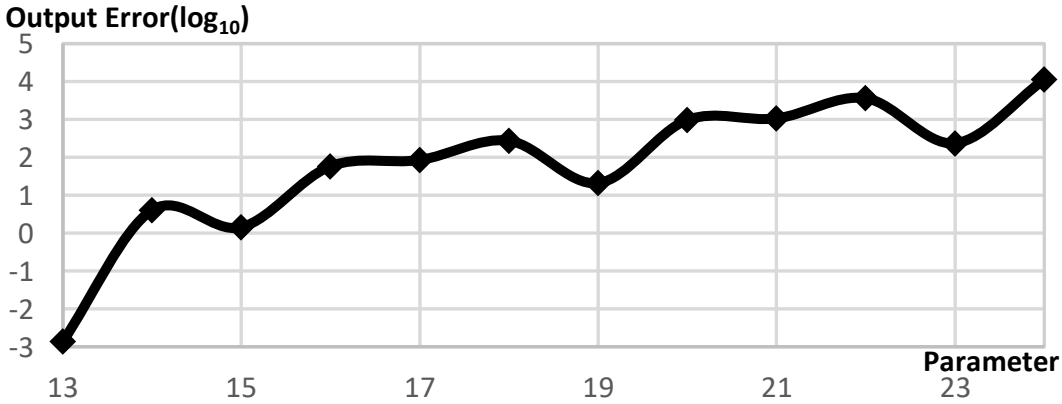



Figure 5.2: The input scale parameter of EVA leads to an arbitrary variation in the output error in LeNet-5, reproduced from [29].

`rescale`, which reduces the noise by the scale and adds $n_{rescale} = 10$. Post-rescaling, the scale is $10^3$, the noise is 11 ($= 1000/10^3 + 10$), and the error is 0.011 ($= 11/10^3$). `rotate` subsequently boosts the noise to 111 after adding $n_{rotate} = 100$. The final error rises to 0.111 ($= 111/10^3$), which is greater than in Figure 5.1a.

**Limitation 2: Ignoring noise variations and relying on a fixed, noise-oblivious waterline.** As highlighted in Table 2.3, different RNS-CKKS operations inject varying amounts of noise into the resulting ciphertext. For example, the `rotate` operation introduces a consistent noise of $n_{rotate} = 100$ in both scenarios depicted in Figure 5.1, thereby increasing the resultant error. Despite this, EVA does not specifically address these noise increments, even though increasing the scale could mitigate the noise's impact on error. Moreover, using a single fixed waterline can unnecessarily limit the scale of operations that minimally affect the resulting error, thereby restricting opportunities for better latency-error trade-offs.

**Importance of Managing Error.** Excessive output error can adversely affect the quality of service (QoS), even in machine learning applications that are somewhat tolerant to errors. Figure 5.3 illustrates the impact of varying output errors on the prediction accuracy (QoS)

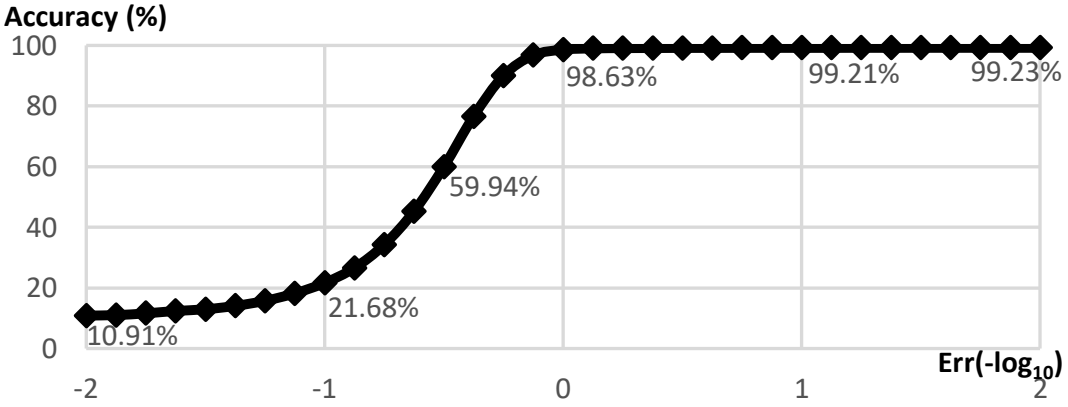



Figure 5.3: Inference accuracy of LeNet-5 for different errors reproduced from [29].

of LeNet-5 when tested on the MNIST dataset. Significant accuracy declines occur with large errors ($-\log\epsilon \leq -1$). Conversely, for smaller errors ($-\log\epsilon \geq -1$), the accuracy improves gradually as the output error diminishes, presenting opportunities to explore various error-latency trade-offs.

The absence of error-aware scale management and precise waterline settings means that existing solutions cannot ensure specific output errors, potentially resulting in RNS-CKKS programs with unpredictable errors. For instance, Figure 5.2 demonstrates that EVA's compilation parameters (i.e., waterline) do not effectively regulate the output error, leading to random variations in error levels. This variability complicates the process of balancing latency and error.

### 5.2 Overview of Error-Latency-Aware Scale Management

This work introduces two major advancements: (1) a new error-latency-aware scale management system (ELASM, §5.2.1) that provides refined control over the scale of ciphertexts; (2) an innovative fine-grained noise-aware waterline management strategy that incorporates a new

error-proportional compile parameter named SNR (§5.2.2). Together, these developments expand the scale management options available, facilitating better error and latency trade-offs.

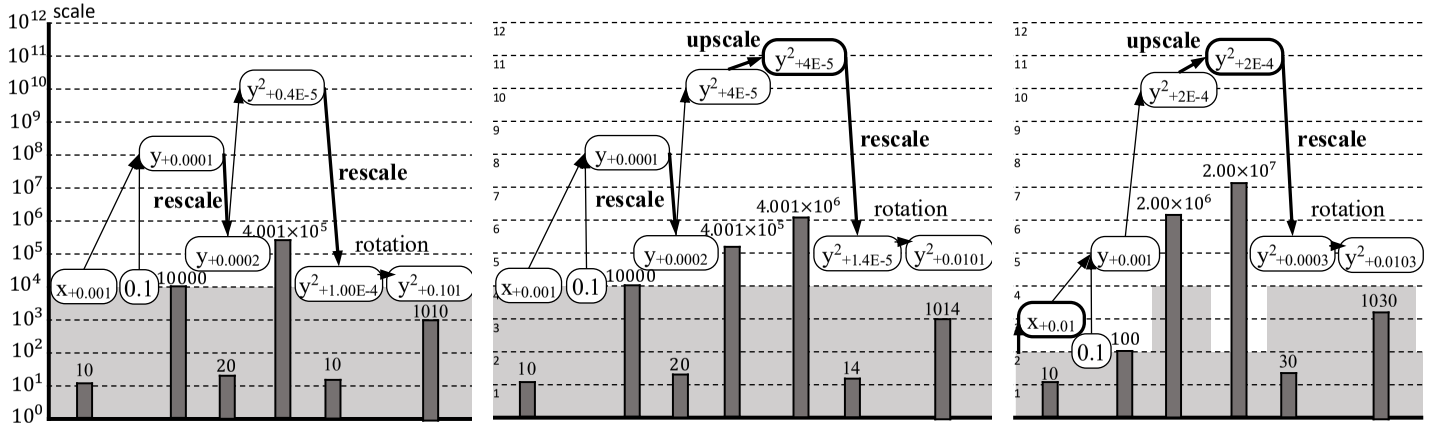
Additionally, this work integrates these concepts into the ELASM compiler, complete with a new noise-aware FHE language and type system (§5.2.3). This integration enhances the compiler's ability to manage and optimize fully homomorphic encryption operations effectively.

### 5.2.1 Error-Latency-Aware Scale Management

The Error-Latency-Aware Scale Management (ELASM) approach is built on the crucial insight that increasing the scale of a ciphertext can significantly reduce the impact of noise on the error rate, as the error ($\epsilon$) is inversely proportional to the scale ($m$), calculated as $\epsilon = n/m$. Unlike previous approaches that utilize the `upscale` operation only to align the scales for addition operations, ELASM also employs `upscale` as a strategic tool for reducing error.

Consider a new scenario illustrated in Figure 5.4 where the computation involves `rotate`$(0.1x)^2$. Assume the initial scale (or waterline) is $10^4$, the rescaling factor $R = 10^3$, and the noise contributions from `rescale`, `rotate`, and `relinearize` (ciphertext multiplication) operations are 10, 1000, and 1000, respectively. Figure 5.4a shows that traditional methods result in an error of 0.101.

In contrast, Figure 5.4b demonstrates how actively increasing the scale can improve the output error to 0.01014. By applying `upscale` to $y^2$, the scheme effectively minimizes the noise impact from the `rotate` operation. Notably, Figure 5.4b implements the same number of `rescale` operations (2) as Figure 5.4a, maintaining the same rescaling level. Since latency is largely influenced by the level ($l$)—as detailed in Table 2.3—this suggests that both approaches would exhibit similar performance times. Therefore, Figure 5.4b presents a more favorable error-latency balance.




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

Figure 5.4: Comparison between the scale management approaches of EVA and ELASM for a program that calculates `rotate`$(0.1x)^2$, reproduced from [29]. In the illustrations, a gray bar (e.g., 10 on the first bar) and a subscript (e.g., +0.001 next to the first $x$) indicate the accumulated noise and error in a ciphertext, respectively. The rescaling factor, $R$, is set at $10^3$. The noise contributions from `rescale`, `rotate`, and `relinearize` (ciphertext multiplication) operations are 10, 1000, and 1000, respectively. In scenarios (a) and (b), the waterline is consistently set at $W = 10^4$. In scenario (c), the waterline is adjusted to be noise-aware, calculated as $m_{op} = SNR \times n_{op}$ with the SNR (Scale-to-Noise Ratio) preset at 10.

Building on this principle, ELASM systematically explores various scale management strategies by positioning `upscale` operations at different points and adjusting scaling factors accordingly. Specifically, ELASM uses Markov Chain Monte-Carlo (MCMC) [79] sampling to iteratively refine the plan and optimize a custom latency-error cost function. Users can tailor a specific cost function for ELASM, factoring in both estimated latency and error. To streamline the exploration process and reduce overhead, ELASM estimates the error and latency for each proposed scale management plan rather than executing the generated code and measuring these metrics directly, which can be time-consuming.

### 5.2.2 SNR: Fine-grained Noise-aware Waterline

This work introduces a new parameter called the *scale-to-noise ratio* (SNR) and establishes fine-grained, noise-aware waterlines for scale management. SNR is analogous to the traditional signal-to-noise ratio used in signal processing. It allows users to specify a minimum ratio between the scale $m$ of a ciphertext and the noise $n$ introduced by an RNS-CKKS operation: $SNR \leq m/n$.

With the SNR parameter, the waterline $m_w$ is no longer a static value set by the input scale. Instead, it dynamically adjusts according to the noise levels: $m_{op} = SNR \times n_{op} \leq m$.

Figures 5.4a and 5.4b demonstrate the drawbacks of using a fixed waterline that is oblivious to varying noise levels, typically set by the maximum input scales as in traditional methods like EVA. In such systems, the resulting error tends to be heavily influenced by operations that introduce significant noise, such as `rotate`. According to Table 2.3, three RNS-CKKS operations introduce different amounts of noise, while others might not. A single, fixed waterline can inappropriately constrain the scales of operations that generate minimal or no noise, thus having little effect on the overall error.

Conversely, Figure 5.4c shows how a noise-aware, flexible waterline can expand the scale management options, enhancing performance without compromising on error rates. For instance, with an SNR of 10, the noise from encryption and `rescale` operations is 10, while the noise from `rotate` and `relinearize` (ciphertext multiplication) is 1000. Consequently, the waterline for `rotate` and ciphertext multiplication is set at $10 \times 1000 = 10000$, and for all other operations at $10 \times 10 = 100$. The waterline for plaintext values, which is not influenced by noise, is set at the lowest level among these values. Unlike in ciphertext multiplications, the waterline for `rotate` also affects its operands because the scales of the operands and the resulting ciphertexts remain unchanged.

Figure 5.4c illustrates that this adaptable waterline approach does not unnecessarily limit the scale of ciphertexts such as $x$, $0.1$, and $y$, which minimally impact the scale. Consequently, only one `rescale` operation is necessary in this scenario, as opposed to the two needed in the other examples (a) and (b). Fewer `rescale` operations imply the possibility of using a smaller coefficient modulus $Q = R^L$, which can lead to reduced latency (refer to Table 2.3). When compared to Figure 5.4b, Figure 5.4c provides a superior latency-error trade-off, achieving lower latency with a comparably minimal error (0.0101 vs. 0.0103).

### 5.2.3 ELASM Compiler Design

This work introduces the ELASM compiler, designed to support the ELASM framework and a fine-grained noise-aware waterline through its dedicated language and type system. This type system ensures that the scale and rescaling level information is embedded within each ciphertext type, and that each RNS-CKKS operation adheres to its specified SNR-based waterlines.

Figure 5.5 outlines the architecture of the ELASM compiler. The scale management plan sampler (§5.3.1) evaluates various optimization plan candidates, each proposing a

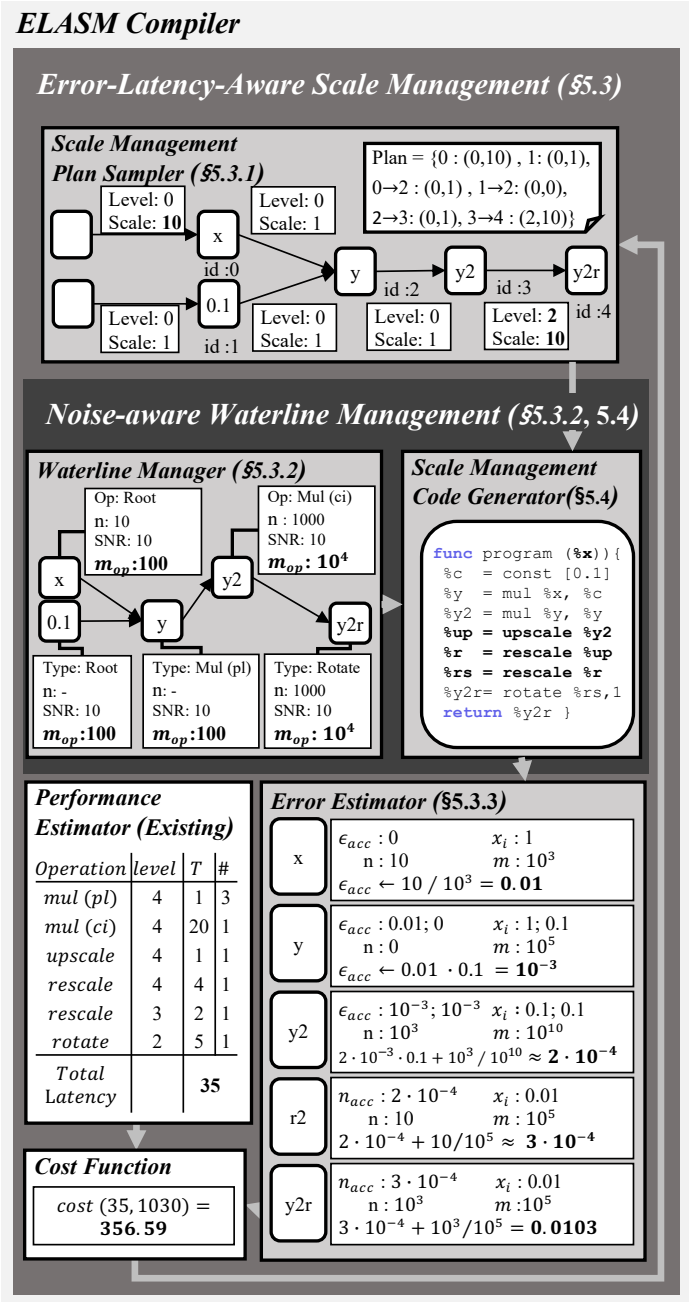



Figure 5.5: Design of the ELASM compiler reproduced from [29]. The example scale management plan and code are the same as Figure 5.4c. $m_{op}$ means a fine-grained waterline for each operation.

specific combination of level reduction (e.g., using `rescale`) and scale increase (e.g., using `upscale`). Utilizing the SNR parameter, ELASM computes noise-aware waterlines for each FHE operation as an integral part of the type system (§3.2).

For each proposed scale management plan, the scale management code generator (§5.4.2) integrates necessary scale management operations (e.g., `rescale`, `modswitch`, `upscale`) to meet the RNS-CKKS constraints, including noise-aware waterlines. It then produces a valid RNS-CKKS program. For every program generated, ELASM evaluates its error (§5.3.3) and latency, calculates a user-defined error-latency cost function, and iteratively inputs this data back into the scale management plan sampler.

Furthermore, ELASM incorporates techniques from HECATE, such as scale management group generation, latency estimation, and backend code implementation, enhancing its efficiency and performance in managing scale and error-latency trade-offs.

## 5.3 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 (§5.3.1), a noise-aware waterline management and code generation (§5.3.2), and an error estimation that enables fast iteration of the exploration (§5.3.3).

### 5.3.1 Sampling of Scale Management Space

ELASM utilizes the Metropolis-Hastings algorithm, a popular Markov-Chain Monte-Carlo (MCMC) sampling method [79], to iteratively seek improved error-latency trade-offs based on the specified SNR parameter. The process begins with an initial plan $P$ and generates a new proposal $P^*$ from it. The algorithm then evaluates whether to accept this new plan based

on the probability $\alpha(P^*|P) = \min(1, \frac{cost(P)}{cost(P^*)})$. If the new plan is accepted, it becomes the basis for the next proposal; otherwise, the next proposal is derived from the original plan $P$. The decision influences the error and latency in the optimized program, allowing users to customize the cost function (e.g., $cost(P) = T \cdot E$ for latency $T$ and error $E$ of plan $P$).

In ELASM, each sample represents a specific scale management plan that specifies where to place scale management operations and the degree of scale increase or level decrease. For example, in the $\texttt{rotate}(0.1x)^2$ program depicted in Figure 5.5, possible locations for scale management operations include transitions between values ($x$, $y$), ($0.1$, $y$), ($y$, $y^2$), and ($y^2$, $\texttt{rotate}(y^2)$). The initial transition ($null$, $x$) is also considered for scale management to simulate adjustments in the scale of an encryption operation. Therefore, a scale management plan in ELASM can be represented as a mapping from an edge (as a key) to a pair indicating the level and scale change (as a value). For instance, the plan shown in Figure 5.5 $\{(null, x) : (0, 10); (null, 0.1) : (0, 1); (x, y) : (0, 1); (0.1, y) : (0, 1); (y, y^2) : (0, 1); (y^2, \texttt{rotate}(y^2)) : (2, 10)\}$ indicates a plan that reduces the level by 2 and increases the scale by 10 between $y^2$ and $\texttt{rotate}(y^2)$ (i.e., before `rotate`). To propose new plans, ELASM randomly selects target positions and adjusts the level and scale accordingly.

### 5.3.2 Noise-aware Waterline Management

Merely inserting scale management operations based on a plan does not ensure the creation of a valid program that adheres to all RNS-CKKS constraints highlighted in Table 3.1. The ELASM compiler employs a type system, which is detailed in §5.4.1, to enforce these RNS-CKKS constraints, including the innovative SNR-based noise-aware waterlines. Additionally, this work establishes a set of rewriting rules (§5.4.2) that facilitate the generation of a program compliant with RNS-CKKS constraints.

Table 5.1: Value and error estimation for each FHE operation reproduced from [29]. The suffix 'c' in operation names indicates a ciphertext, while 'p' signifies a plaintext; for example, $mulcp$ represents a multiplication between a ciphertext and a plaintext. Operations not listed here maintain their error and value constants. Each operation processes an encoded integer $v = mx + n = m(x + \epsilon)$, where $m$ is the scale, $x$ is the value, $n$ is the noise, and $\epsilon$ is the error. An asterisk (*) denotes an estimation.

| Operation | Est. Value | Estimated Error |
|---|---|---|
| mulcc($v_1$,$v_2$) | $x_1^* x_2^*$ | $\epsilon_1^* x_2^* + \epsilon_2^* x_1^* + \epsilon_1^* \epsilon_2^* + n_{relinearize}/m$ |
| mulcp($v_1$,$v_2$) | $x_1^* x_2$ | $\epsilon_1^* x_2$ |
| addcc($v_1$,$v_2$) | 1 | $\epsilon_1^* + \epsilon_2^*$ |
| addcp($v_1$,$v_2$) | 1 | $\epsilon_1^*$ |
| `rotate` ($v$) | $x^*$ | $\epsilon^* + n_{rotate}/m$ |
| `rescale` ($v$) | $x^*$ | $\epsilon^* + n_{rescale}/(m/R)$ |
| `downscale` ($v$) | $x^*$ | $\epsilon^* + n_{rescale}/m_w$ |
| `upscale` ($v$) | $x^*$ | $\epsilon^*$ |

For instance, in Figure 5.5 where the SNR is set to 10, ELASM calculates specific waterlines for each ciphertext depending on the type of operation involved. For operations like $y^2$ and `rotate`$(y^2)$, the waterlines are calculated as $10 \times 10^3 = 10^4$. For other operations, the waterlines are set at $10 \times 10 = 100$. Subsequently, ELASM incorporates scale management operations such as `upscale` and `rescale` into the program. The integration of these operations ensures that the program conforms to all RNS-CKKS constraints, validated by the robustness of the ELASM's type system as discussed in §3.4.

### 5.3.3 Error Estimation

ELASM estimates the error and latency of candidate programs statically, rather than dynamically executing and measuring these metrics, which can be prohibitively expensive. For latency, ELASM simply sums up the expected time for each RNS-CKKS operation, as identified in Table 2.3, where the time complexity depends on the polynomial modulus $N$ and the level $l$, values that can be determined once a program is specified.

Estimating the exact amount of error, however, poses more of a challenge, especially since calculating the error from ciphertext multiplication involves unencrypted values, which are not accessible. As depicted in Table 3.1, when two ciphertexts $v_1 = m_1(x_1 + \epsilon_1)$ and $v_2 = m_2(x_2 + \epsilon_2)$ are multiplied, the resulting error includes terms dependent on the values $x_1$ and $x_2$, such as $\epsilon_1 x_2 + \epsilon_2 x_1 + \epsilon_1 \epsilon_2 + n_{relinearize}/m$.

To tackle this issue, ELASM adopts a straightforward value estimation approach that, while not capable of determining the absolute error, is adequate for comparing different candidate programs. Table 5.1 lists ELASM's formulas for estimating value and error for each FHE operation. These estimations are then propagated through the data flow of the operations. Notably, for most operations, estimating error does not necessitate an estimated value. However, for operations like $mulcc$, the error estimation does rely on an estimated value.

Given the difficulties in precisely estimating values, this work assumes that maintaining a consistent estimated value at the same program points across different candidate programs suffices for evaluating error differences. Consequently, ELASM sets the estimated value of an addition to 1—the multiplicative identity—which simplifies error estimation for operations like $mulcc$ and $mulcp$. This approach aims to accurately reflect the effects of multiplications, while accepting less precision for additions. By resetting values to 1, ELASM effectively conducts a piece-wise analysis of FHE programs, focusing on enabling comparative assessments rather than precise value estimations.

We will later demonstrate that this simplified error estimation closely aligns with the actual resultant errors, as discussed in §5.5.2.

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

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

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

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

Figure 5.6: Typing rules that uses fine-grained waterline reproduced from [29]. The minimal scale for a `rescale` operation is $m_{rescale}$ and the minimal scale for a `rotate` is $m_{rotation}$.

## 5.4 Code Generation

This section outlines the code generation process in ELASM, focusing on the scale type system that upholds the RNS-CKKS constraints, including the advanced noise-aware waterline (§5.4.1). It also discusses the ELASM code generation approach, which utilizes rewriting rules to ensure compliance with these constraints (§5.4.2).

For a detailed exploration of the formal operational semantics and the robustness of the type system, refer to Sections 3.3 and 3.4.

### 5.4.1 Type System of ELASM

Figure 5.6 displays a part of the typing rules that incorporate the waterline constraints. The comprehensive set of typing rules is available in §3.2. This type system is crafted to uphold the RNS-CKKS constraints detailed in Table 3.1, including the SNR-based noise-aware waterlines outlined in §5.2.2. The type soundness of HECATE language ensures that a well-typed program adheres to these RNS-CKKS constraints. A brief proof of this is provided in §3.4.

The ELASM type system specifically enforces the SNR constraint. Notably, four rules, such as Equations Mul$_{CC}$, Rot, DS, and RS, mandate minimum scales (waterlines): $m_{relinearize}$ for ciphertext multiplication, $m_{rotation}$ for rotation operations, and $m_{rescale}$ for both rescale and downscale operations. The waterline $m_{relinearize}$ is typically satisfied naturally since ciphertext multiplication inherently increases the scale.

However, to meet the SNR-based noise-aware waterline requirement, the waterlines $m_{rotation}$ and $m_{rescale}$ must be defined as $n_{rotate} \times SNR$ and $n_{rescale} \times SNR$ respectively, for a given $SNR$. From Table 2.3, $n_{rotate}$ is calculated as $\frac{8\sqrt{3}}{3}\sigma lN + \frac{8\sqrt{2}}{3}N + \sqrt{3N}$ and $n_{rescale}$ is $\frac{8\sqrt{2}}{3}N + \sqrt{3N}$. Since $n_{rotate}$ is influenced by the ciphertext level, which is unknown prior to rescaling, ELASM uses the highest plausible value to ensure compliance.

### 5.4.2 ELASM Rewriting Rules

Given a program that might not be legally executable without necessary scale management operations, ELASM generates a well-typed program that complies with all RNS-CKKS constraints for a specified SNR parameter. To achieve this, ELASM utilizes a series of code rewriting rules that introduce scale management operations such as `downscale`, `upscale`, and `rescale`.

These rewriting rules (see Figure 5.7) modify expressions to satisfy the conditions required by the typing rules in Figure 5.6. For example, Equation DScale inserts `downscale` for both operands when it proves more efficient than performing multiplication followed by rescale. Equation DMatch and Equation LMatch add `downscale` and `modswitch`, respectively, to align the levels of operands in binary operations, thereby meeting the requirements of typing rules for addition and multiplication (Equations Add to Mul$_{CC}$). Equation SMatch uses `upscale` to equalize the scales of operands in binary operations, fulfilling the typing rule Equation Add.

$$\frac{\Gamma \vdash h : \texttt{scale}(m, d) \quad \Gamma \vdash h' : \texttt{scale}(m', d) \quad m \cdot m' < {m_{rescale}}^2 \cdot R}{h \times h' \xrightarrow{rewrite} \texttt{downscale}(h) \times \texttt{downscale}(h')} \text{(DScale)}$$

$$\frac{\Gamma \vdash h : \texttt{scale}(m, d) \quad m \geq R m_{rescale}}{h \xrightarrow{rewrite} \texttt{rescale}(h)} \text{(Rescale)}$$

$$\frac{\Gamma \vdash e : \texttt{scale}(m, d) \quad \Gamma \vdash e' : \texttt{scale}(m', d') \quad m > m_{rescale} \quad d < d'}{e \oplus e' \xrightarrow{rewrite} \texttt{downscale}(e) \oplus e'} \text{(DMatch)}$$

$$\frac{\Gamma \vdash h : \texttt{cipher}(m, d) \quad \Gamma \vdash h' : \texttt{real}}{h + h' \xrightarrow{rewrite} h + \texttt{upscale}(h', m)} \text{(EncodeAdd)}$$

$$\frac{\Gamma \vdash e : \texttt{scale}(m, d) \quad \Gamma \vdash e' : \texttt{scale}(m', d') \quad m = m_{rescale} \quad d < d'}{e \oplus e' \xrightarrow{rewrite} \texttt{modswitch}(e) \oplus e'} \text{(LMatch)}$$

$$\frac{\Gamma \vdash h : \texttt{cipher}(m, d) \quad \Gamma \vdash h' : \texttt{real}}{h \times h' \xrightarrow{rewrite} h \times \texttt{upscale}(h', m_{rescale})} \text{(EncodeMul)}$$

$$\frac{\Gamma \vdash h : \texttt{scale}(m, d) \quad \Gamma \vdash h' : \texttt{scale}(m', d) \quad m < m'}{h + h' \xrightarrow{rewrite} \texttt{upscale}(h, m'/m) + h'} \text{(SMatch)}$$

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

Figure 5.7: Rewriting rules for ELASM reproduced from [29]. $m_{rescale}$ means the minimal scale required by a rescale operation, and $m_{rotation}$ means the minimal scale required by a rotate operation. `scale` includes `cipher` and `plain` type.

Additionally, Equation EncodeAdd employs `upscale` to convert a `real` type operand to a `plain` type, setting its scale to match that of the other operand as required by Equation Add, which does not permit `real` type operands. Meanwhile, Equation EncodeMul elasmlies `upscale` for casting in multiplication scenarios, aligning the encoding scale with the waterline of `rescale`, $m_{rescale}$, as all input data for the program are set to this scale.

A new rule, Equation URot, specifically for rotation operations, elasmlies `upscale` if the operand's scale falls below the noise-aware waterline for rotation ($m_{rotation}$), calculated as $n_{rotate} \times SNR$. This ensures compliance with the typing rule Equation RS in Figure 5.6.

Once the scale management code generation is complete, and the optimal program code is chosen, it is translated into LLVM IR, which then invokes functions from the FHE library. Microsoft SEAL [23], which implements the RNS-CKKS scheme, is used as the backend for this process.

### 5.5 Evaluation of Error-Latency-Aware Scale Management

This study compares ELASM with leading FHE compilers such as EVA [26] and Hecate [28] to assess the performance improvements brought by the proposed error-latency aware scale management and noise-aware waterlines. For the evaluation, seven machine learning and deep learning applications are implemented and tested. These benchmarks include the same set used in Hecate, with the addition of multivariate regression (MR) for epoch 2 and 3, and all except SqueezeNet from EVA's benchmarks. The evaluations run on the same setting with HECATE as described in §4.6.1

ELASM processes 12000 scale management plan samples across 12 parallel threads, estimating the level decrement as $l_{dec} \sim U_{[0,2]}$ and the scale increment as $m_{inc} = ReLU(X)$ for $X \sim U_{[-10,10]}$, where $U_{[a,b]}$ represents a uniform distribution over the interval [a,b]. The number of newly sampled positions for a new plan is calculated as $\sqrt{\text{\# of candidate positions}}$.

The cost function used in ELASM is $\sqrt{T} \cdot (60 + \log E)$, balancing the influence of the compilation parameters on latency $T$ (quadratic) and the resultant error $E$ (inversely exponential). Here, the square root is applied to $T$ and a logarithmic transformation to $E$, after which they are multiplied. An addition of 60 to $\log E$ ensures the value remains positive.

#### 5.5.1 Pareto Curve of Error-Latency Trade-off

Figure 5.8 displays Pareto-optimal error and latency trade-off options for all benchmarks. This study adjusts the compilation parameter (the waterline in EVA and Hecate, and the

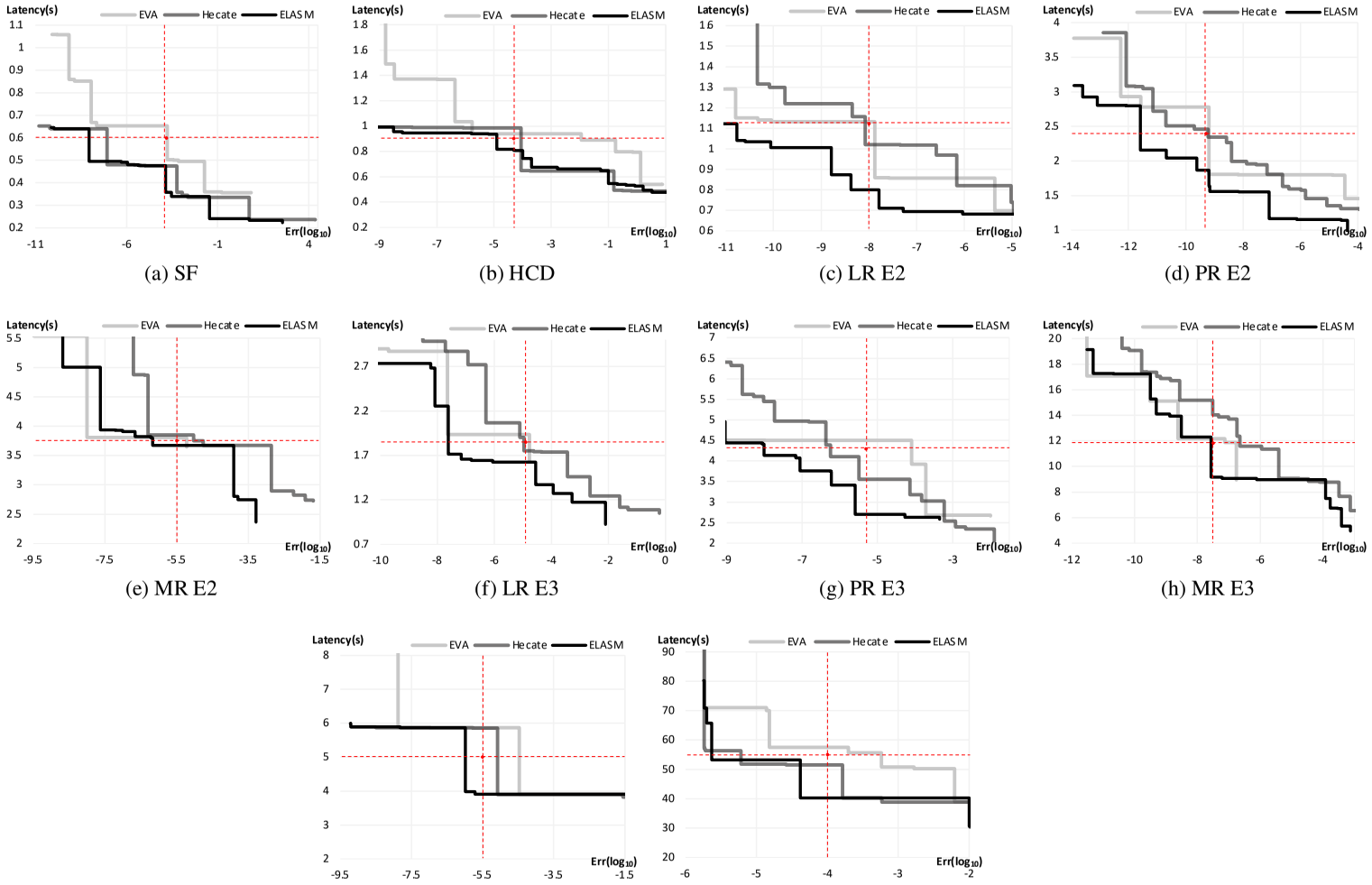
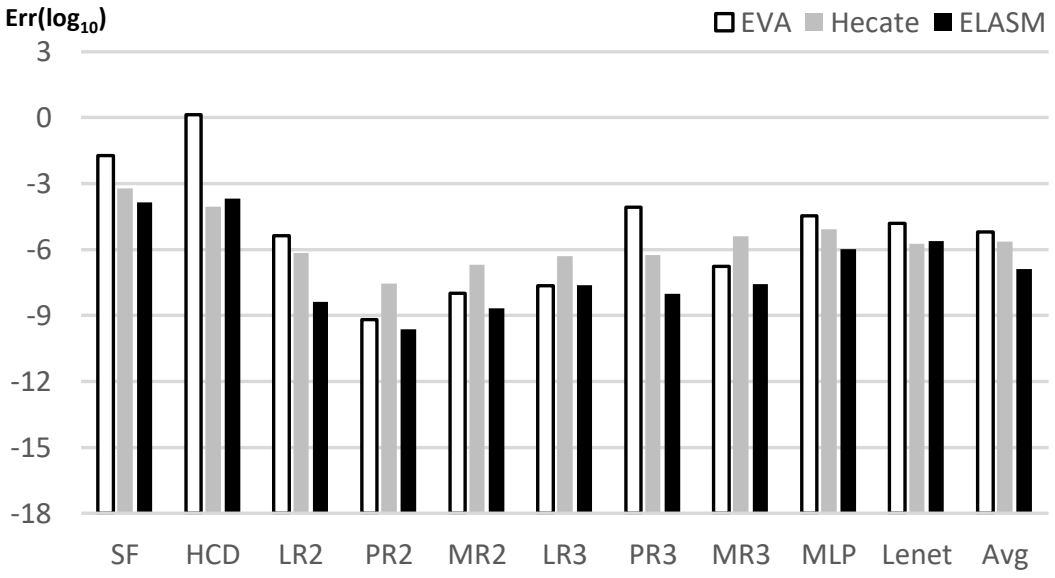
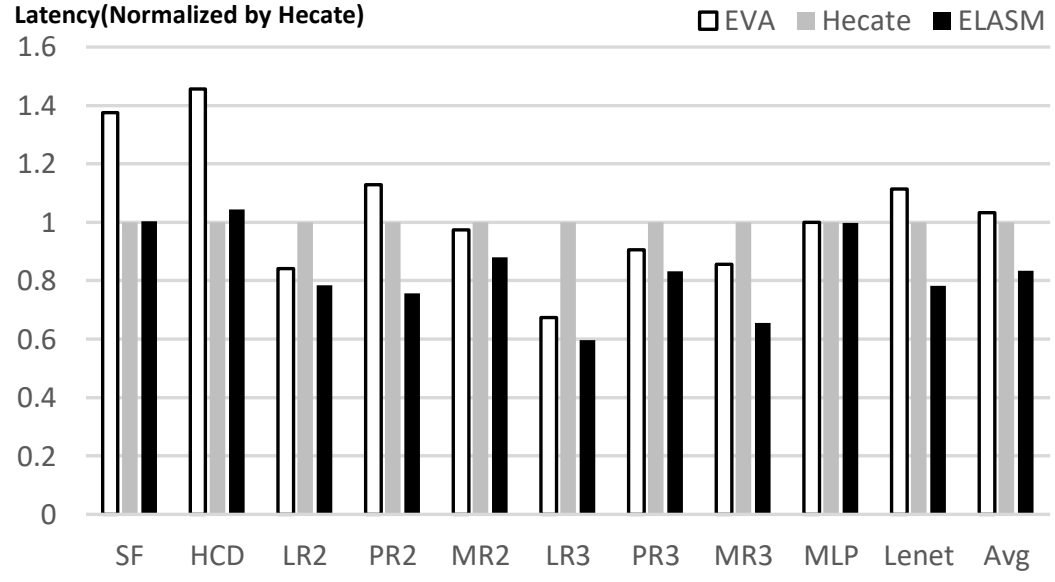



Figure 5.8: Pareto-frontier plots of error-latency trade-offs reproduced from [29]. E2 and E3 stands for 2 and 3 epoch of the gradient descent algorithm of regression benchmarks.




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




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

Figure 5.9: Error and latency of the benchmarks for a given constraint, plotted on Figure 5.8, reproduced from [29].

waterline of `rescale` derived from $SNR$ in ELASM) from $2^{15}$ to $2^{50}$. Benchmarks such as SF, MR E2 (epoch 2), MR E3 (epoch 3), and MLP demonstrate that their Pareto curves shift left, indicating reduced error for a given latency. For LR E2, LR E3, PR E2, and PR E3, the curves shift left and downward, showing simultaneous improvements in both latency and error. For other benchmarks like HCD and Lenet, the curve shapes differ too much for direct comparison, yet generally, they show enhanced performance and reduced error.

To quantify the Pareto curve improvement for the error-latency trade-off, the best error and latency for a specified constraint are marked with red lines in Figure 5.8. These constraint points represent the average error and latency values among the results explored.

Figure 5.9 compares error and latency among EVA, Hecate, and ELASM. ELASM focuses on enhancing both error and latency, evidenced by observed improvements in these metrics. Table 4.1 indicates that for the same latency across each application, on average, ELASM exhibits smaller errors than EVA and Hecate by 312.8× and 31.2×, respectively. Differing from previous methods, ELASM actively adjusts the scale of ciphertext to better manage error. Moreover, Figure 5.9b reveals that at comparable error levels, ELASM achieves faster performance than EVA and Hecate by 26.7% and 21.3%, respectively. Thanks to its broader scale management search space facilitated by the innovative noise-aware waterlines, ELASM effectively discovers more efficient scale management plans with reduced latency compared to Hecate.

### 5.5.2 Error Estimation

This study assesses the accuracy of the error estimation method used in ELASM. As outlined in §5.3.3, ELASM is not designed to calculate the exact magnitude of errors for a specific program. Rather, its error estimation method aims to provide proportional estimations that are useful for computing and comparing the cost functions of different scale management

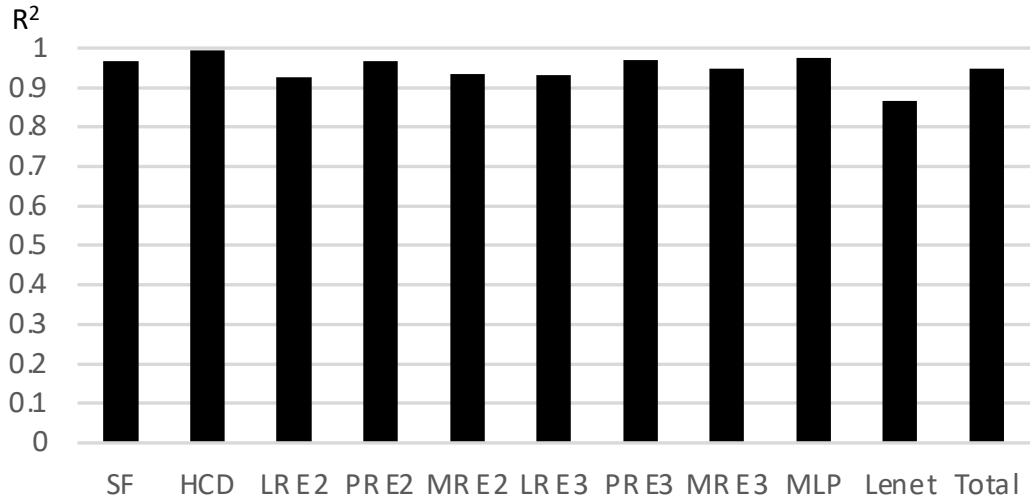



Figure 5.10: Coefficient of determination ($R^2$) from linear fitting of estimated error and measured error reproduced from [29].

plans. The crucial measure of accuracy for this error estimation is how closely the estimated errors correlate proportionally with the actual measured errors.

Figure 5.10 displays the $R^2$ value of the linear relationship between the estimated errors and the actual measured errors. ELASM's method of estimating errors achieves an average $R^2$ value of 0.948, suggesting that the error estimation is sufficiently proportional for evaluating different scale management plans. The lowest $R^2$ value, observed with Lenet at 0.884, indicates that the simple assumptions used for value estimations lack precision. This inaccuracy becomes more pronounced through multiple computational steps involving multiplications, rotations, and additions, which can amplify the discrepancies between assumed and actual values.

### 5.5.3 Error-proportionality of SNR parameter

This section assesses how closely the compiler parameters used in EVA, Hecate, and ELASM relate to the resulting error. A compiler parameter that proportionally influences the error

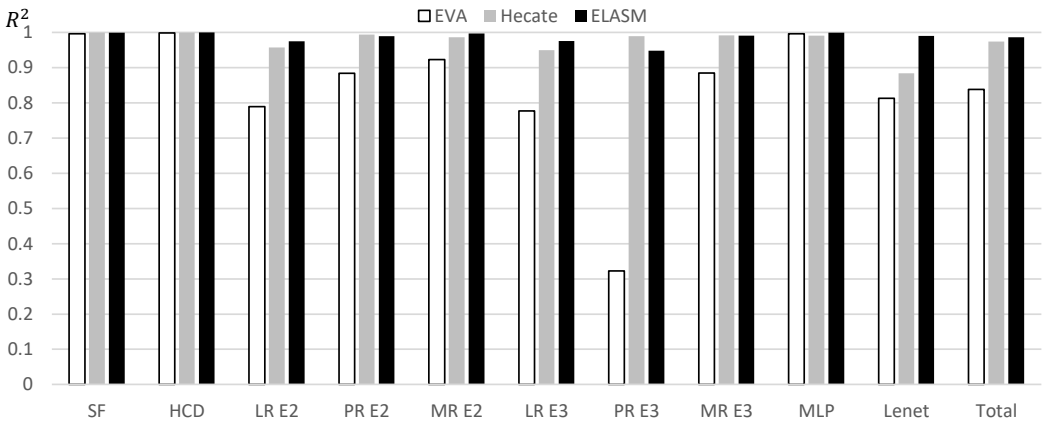



Figure 5.11: Coefficient of determination ($R^2$) from linear fitting of parameter and error reproduced from [29]

enables users to effectively explore various error-latency trade-offs. By adjusting a single parameter while observing corresponding changes along the Pareto curve, users can predict how alterations to the parameter will impact the error-latency balance. This approach eliminates the need to test every possible parameter setting.

Figure 5.11 displays the $R^2$ values from linear regressions between the compiler parameters and the output errors for each application. This indicates how the error correlates with the compiler parameter. Since EVA, Hecate, and ELASM utilize parameters that function differently, we align these parameters on the same x-axis based on their maximum allowable operation-wise error.

Overall, ELASM demonstrates superior error proportionality compared to EVA, which seems ineffective at controlling errors. EVA exhibits very low proportionality, especially in PR E3, due to its suboptimal scale management and the deep multiplications required in PR E3, which necessitate large coefficient moduli at some parameter settings, rendering them impractical.

Interestingly, for PR E3, EVA registers a slightly higher $R^2$ value than ELASM. It's important to note that ELASM is designed to optimize a cost function that includes both latency and error. Consequently, ELASM may select a scale management strategy where the error is higher if it results in a lower overall error-latency cost, causing some deviations in the results.

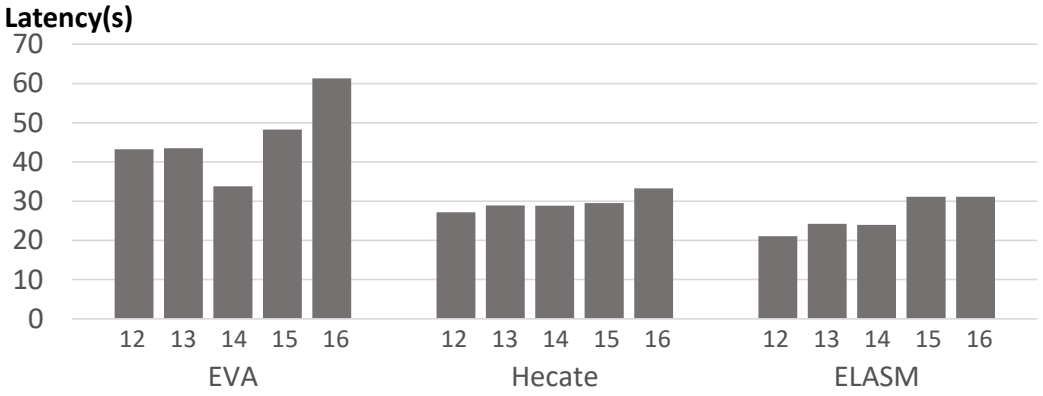
Nevertheless, Figure 5.11 illustrates that ELASM consistently achieves high $R^2$ values across all tested applications, with the lowest $R^2$ value being 0.948 and an average $R^2$ of 0.986.

### 5.5.4 Case Study: End-to-end DNN Application

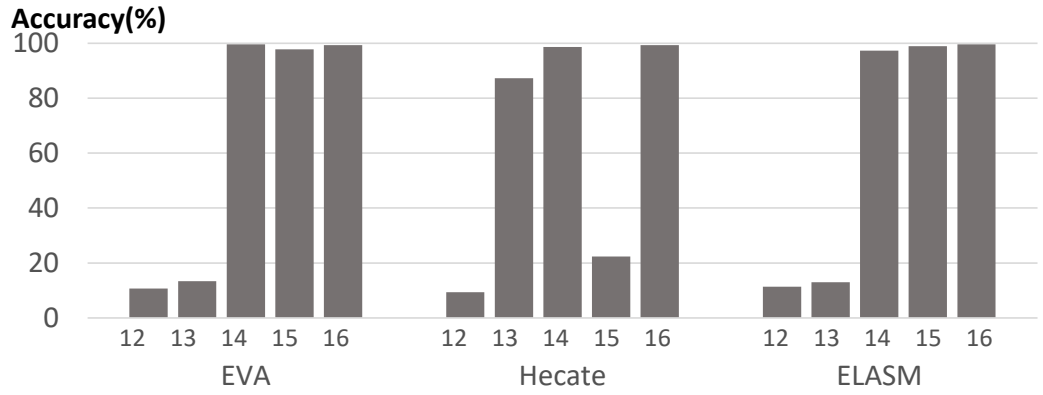
We developed a prototype for end-to-end image classification using LeNet-5. In this scenario, a client encrypts an image from the MNIST evaluation dataset and sends it to a server. The server processes the encrypted image using LeNet-5 and returns it to the client, who then decrypts it to verify its correctness. Both the client and server are connected via a 1Gbps network and have identical configurations for the evaluation setup.

Figure 5.12a displays the end-to-end latency results. EVA does not prioritize latency during its scale management, resulting in no consistent correlation between its parameters and latency. In contrast, Hecate and ELASM incorporate latency considerations into their scale management, demonstrating a monotonic relationship between their parameters and latency.

Figure 5.12b illustrates the inference accuracy for EVA, Hecate, and ELASM. This figure indicates a significant accuracy improvement at a certain threshold, as detailed in Figure 5.3. Notably, Hecate, which focuses on minimizing latency often at the expense of accuracy, shows unpredictable accuracy drops at a waterline of $2^{15}$. In comparison, ELASM incorporates




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




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

Figure 5.12: A case study of using LeNet-5 for end-to-end inference reproduced from [29]. This includes measuring the latency, which accounts for both network time and the time taken for encryption and decryption processes. Accuracy measurements are conducted using a subset of the MNIST evaluation dataset..

error estimates into its scale management and maintains a consistent relationship between parameters and accuracy.

Overall, both EVA and Hecate fall short in facilitating an efficient exploration between accuracy and latency. ELASM, however, demonstrates a consistent improvement in both metrics, underscoring the effectiveness of its error-proportional parameter strategy. Additionally, for comparable error levels, ELASM achieves the lowest end-to-end latency, allowing users to efficiently navigate the most advantageous accuracy-latency trade-offs.

### 5.6 Summary

This work introduces a novel approach to scale management in RNS-CKKS fully homomorphic encryption, termed Error- and Latency-Aware Scale Management (ELASM). ELASM is designed to meticulously explore a variety of scale management plans, efficiently evaluating the output error and latency associated with each. Through a methodical iterative process, ELASM identifies the most effective plan that minimizes both error and latency, marking a significant advancement in scale management techniques.

Central to ELASM is the innovative scale-to-noise ratio (SNR), a new error-proportional parameter that enhances the precision of scale adjustments based on noise levels. This feature, alongside the introduction of fine-grained noise-aware waterlines, significantly expands the scope of scale management exploration, allowing for more nuanced adjustments tailored to specific encryption needs.

The ELASM framework has been fully integrated into the ELASM compiler, which also features a noise-aware ELASM type system specially developed to support this advanced scale management approach. To demonstrate the effectiveness of ELASM, this work conducts extensive evaluations using ten diverse machine learning and deep learning benchmarks. ELASM provides 21.3% and 31.2$\times$ better latency and error for a given error and latency,

respectively. The results highlight ELASM's superior ability to optimize latency and error trade-offs compared to current leading RNS-CKKS compilers such as EVA and HECATE. These enhancements make ELASM a cutting-edge tool in the domain of homomorphic encryption, setting new standards for performance and accuracy in secure computation.

# CHAPTER 6

# PERFORMANCE-AWARE STATIC SCALE ANALYSIS

This research introduces a novel approach [30] termed *reserve analysis* for performance-aware backward static scale analysis in RNS-CKKS programs. The concept of *reserve* $r$ is defined as the quotient of the coefficient modulus $R^l$ by the current scale $m$, indicating the remaining scale capacity available in a ciphertext. A crucial feature of the reserve is its consistency across `rescale` operations, which simplifies the analysis process.

The semantics of reserve are formalized, and a *reserve type system* is established to effectively manage reserves and assess the latency implications for RNS-CKKS operations. Reserve analysis utilizes a backward approach, starting from the results and working towards the inputs to deduce the reserve requirements for each operand in an operation.

This approach prioritizes operations that are computationally intensive by strategically allocating reserves to potentially lower their operational levels aggressively. Following this allocation, a rescale placement algorithm evaluates the cost-effectiveness of various `rescale` positions within the program, aiming to identify the most efficient placement strategy. This method ensures an optimized balance between computational efficiency and resource utilization in encrypted computations. The proposed techniques are implemented as a new optimization pass on the top of HECATE compiler.

Table 6.1: Latency of RNS-CKKS operations for level 1 to 8 ($\mu s$) adapted from [30]. The other parameters are $N = 2^{15}$ and $R = 2^{60}$.

| Op | Level | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| `modswitch` (`plain`) | 29 | 43 | 57 | 71 | 86 | 100 | 114 | 128 |
| `modswitch` (`cipher`) | 48 | 86 | 156 | 208 | 286 | 315 | 391 | 457 |
| `cipher` + `plain` | 50 | 98 | 153 | 209 | 269 | 335 | 409 | 472 |
| `cipher` + `cipher` | 85 | 204 | 250 | 339 | 421 | 531 | 615 | 723 |
| `cipher` $\times$ `plain` | 211 | 421 | 642 | 853 | 1120 | 1260 | 1509 | 1726 |
| `rescale` (`cipher`) | 1926 | 3119 | 4525 | 5706 | 6901 | 8198 | 9570 | 10781 |
| `rotate` (`cipher`) | 3828 | 7966 | 13584 | 20933 | 28832 | 40137 | 51080 | 64134 |
| `cipher` $\times$ `cipher` | 4363 | 9172 | 15658 | 23517 | 33974 | 43235 | 56611 | 68785 |

## 6.1 Necessity of Performance-aware Static Scale Analysis

This section explores the current scale management techniques used in leading RNS-CKKS compilers like EVA [26] and HECATE, highlighting three main limitations and setting the stage for new, innovative solutions.

### 6.1.1 Forward Static Scale Analysis

EVA [26] utilizes a forward static scale analysis that progresses from the start to the end of a program, adding a `rescale` operation whenever the resultant scale exceeds a predefined global *waterline*. This waterline, set by the programmer, represents the scale of an input ciphertext and aims to minimize the accumulated scale of the program's result, which in turn influences the level of the input ciphertexts.

For instance, consider a sample program $x^3 \cdot (y^2 + y)$ with an input scale or waterline $W = 20$ (effectively $2^{20}$) and a rescaling factor $R = 60$. EVA conducts a scale analysis and implements scale management operations as detailed in Figure 6.1b. Starting with input variables $x$ and $y$ at scale $m = 20$, each multiplication escalates the resultant scale. EVA adds `rescale` when the resultant scale surpasses the waterline. For example, the scale of $x^2$ does

not warrant a `rescale`, so no operation is inserted between $x^2$ and $x^3$, maintaining the same level and latency for both.

An `upscale` operation boosts $y$'s scale from $20$ to $40$ to equalize the scales of the two operands in the addition $s = y^2 + y$. A `rescale` is applied after the final multiplication, reducing $q$'s scale from $100$ to $40$, yet it remains above the waterline $W = 20$. By rescaling $q$, EVA minimizes the ciphertext size, which reduces storage and network costs. This forward analysis determines that the level of input ciphertexts must be at least $l \geq 2$ to prevent scale overflow and accommodate one `rescale`, setting the minimal safe coefficient modulus $Q = R^2 = 120$.

Nonetheless, EVA's method doesn't optimally manage levels for each intermediate ciphertext, leading to potential performance inefficiencies. In RNS-CKKS, the level $l$ affects a ciphertext's size (Figure 2.1a) and thus the operation latency. Lower levels correspond to reduced latencies. Table 6.1 outlines our latency measurements for various RNS-CKKS operations at different levels. Managing levels of operations, especially heavy ones like `cipher` $\times$ `cipher` and `rotate`, which are prevalent in ML applications, is crucial.

The improved scale management approach shown in Figure 6.1c, developed in this work, applies `rescale` operations to $x$ and $y$ early, enabling many operations at lower levels (level 1 vs. 2 in Figure 6.1b), which decreases overall latency. Early `rescale` operations increase the accumulated scales but do not impact latency, as they fully utilize the remaining scale (reserve) of the result without increasing the level. Figure 6.1b shows an under-utilized scale of $20$ in the final ciphertext $q'$, whereas Figure 6.1c fully utilizes all $60$ in $q$ while maintaining the maximum level at 2.

EVA's forward static analysis struggles with level-aware, performance-focused scale management because it does not consider subsequent operations when inserting scale management operations, hindering optimization. What is needed is a mechanism for analyzing

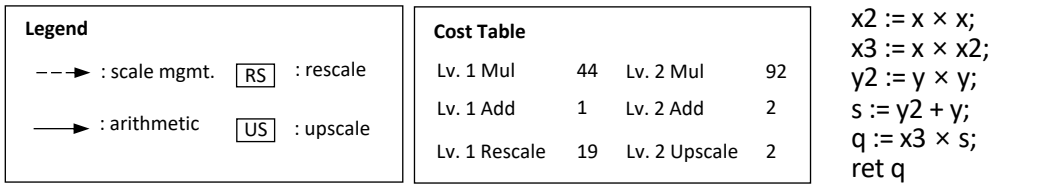
a program backward with a defined scale budget. This paper introduces a novel *reserve* concept representing the required scale budget from subsequent operations and a *reserve analysis* that examines reserves in a backward direction.

### 6.1.2 Tightly Coupled Scale Management and Analysis

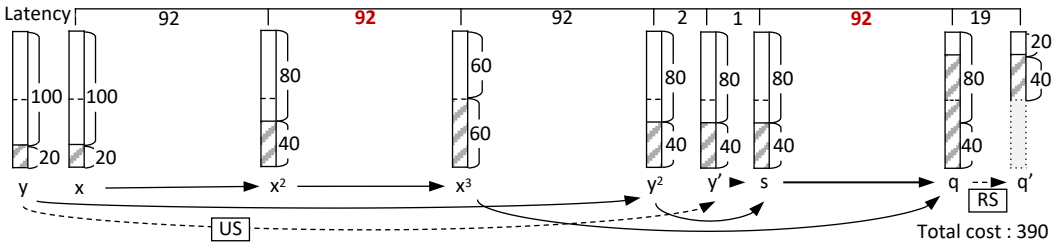
Unlike the fixed arithmetic operations specified in a program, the quantity of scale management operations can vary significantly across different management strategies. As demonstrated in Table 6.1, among the three scale management operations—`rescale`, `upscale`, and `modswitch` —`rescale` exhibits the highest latency. The latency for `upscale` is comparable to that of `cipher` $\times$ `plain` or `cipher` $+$ `plain`, depending on how it is implemented. This observation suggests that minimizing the number of `rescale` operations is crucial for optimizing RNS-CKKS programs. However, existing compilers often do not distinguish between the placement of `rescale` operations and scale allocation, thereby overlooking potential optimizations.

In contrast to the scenario depicted in Figure 6.1c, which utilizes four `rescale` operations, Figure 6.1d implements three and achieves reduced latency. The key difference lies in the handling of the addition $s = y^2 + y$: the former approach applies `rescale` to both operands before addition, while the latter performs a single `rescale` on the result after the addition. To facilitate this kind of optimization, it is essential to independently determine the placement of `rescale` operations apart from scale allocation. For example, for the addition result $s$, Figure 6.1c uses a scale $m = 20$ and level $l = 1$, whereas Figure 6.1d uses a scale $m = 80$ and level $l = 2$.

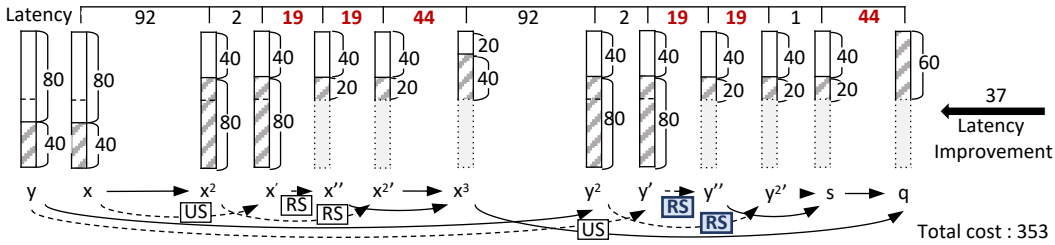
This scenario underscores the need for a new construct that remains *invariant* to `rescale` operations, effectively separating scale analysis from the placement of scale management operations. The newly proposed *reserve* concept maintains this invariance across `rescale`



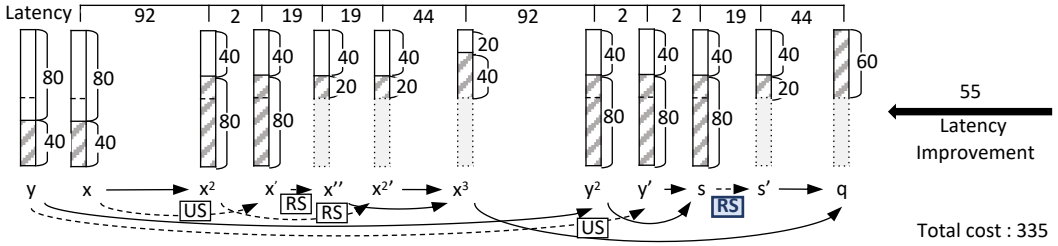

(a) Example program.




(b) Existing scale management (EVA) for Figure 6.1a.




(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 6.1: Execution time and scale management plan for the example program (Figure 6.1a) that computes $x^3 \cdot (y^2 + y)$ in EVA (Figure 6.1b) and this work (Figures 6.1c and 6.1d) for the given rescaling factor 60 and waterline 20, reproduced from [30]. The cost is the latency at Table 6.1 with the unit of $100\mu$s The numeric numbers for rescaling factor, waterline, scale, and reserve are given in the log base 2.

operations, leading to the introduction of a novel `rescale` placement method that is decoupled from reserve analysis.

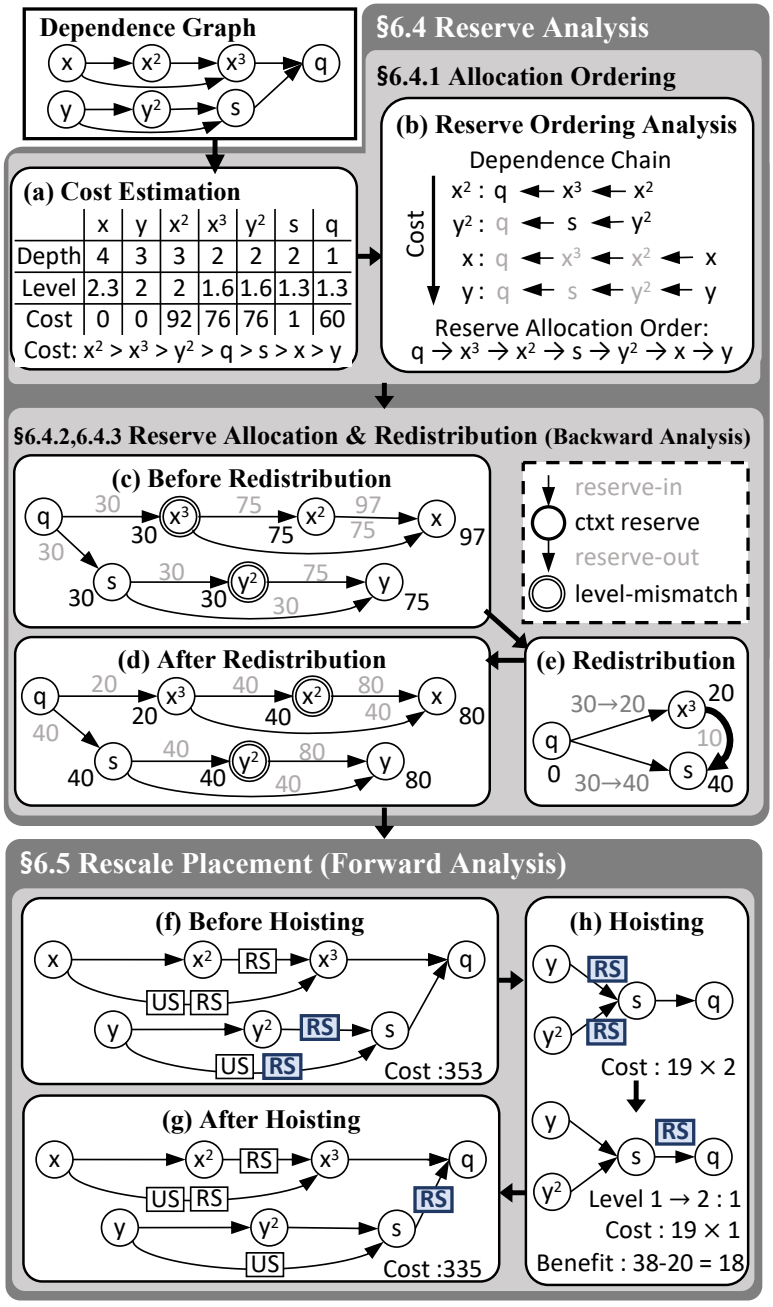
### 6.1.3 Exploration-based Scale Management

HECATE (and ELASM) introduces a method of iterative exploration for scale management. In each exploration cycle, it generates several scale management strategies. Each strategy introduces a scale management operation at a randomly selected point within the program. These strategies are then used to create candidate programs that comply with the RNS-CKKS framework by incorporating necessary scale management operations.

HECATE evaluates each candidate program to identify the one with the lowest estimated latency and continues to refine its scale management approach through a method known as hill-climbing. This process is repeated across many iterations. As a result, HECATE can often achieve shorter overall runtime latencies compared to EVA.

However, this iterative exploration process significantly lengthens the time it takes to compile programs. For example, compiling the deep-learning application LeNet-5 with HECATE takes about 483 seconds. The compilation time can become even longer for larger neural networks. This extended scale management time is critical because quicker scale management methods can lead to new optimization opportunities, such as the insertion of bootstrapping and the selection of data layouts, which frequently require re-running scale management.

This work seeks to deliver comparable improvements in performance without the need for extensive exploration of scale management possibilities.




| | x | y | $x^2$ | $x^3$ | $y^2$ | s | q |
|---|---|---|---|---|---|---|---|
| Depth | 4 | 3 | 3 | 2 | 2 | 2 | 1 |
| Level | 2.3 | 2 | 2 | 1.6 | 1.6 | 1.3 | 1.3 |
| Cost | 0 | 0 | 92 | 76 | 76 | 1 | 60 |

Cost: $x^2 > x^3 > y^2 > q > s > x > y$

Figure 6.2: Overview of the rescale placement and reserve analysis, using the same example of Figure 6.1a, adapted from [30].

## 6.2 Overview of Performance-aware Static Scale Analysis

This work introduces a new concept in scale management called *reserve*, defined as the available scale budget for future RNS-CKKS operations. Illustrated in Figure 2.1, the reserve $r$ is seen as the unused capacity in the coefficients of a polynomial, where the product of the reserve and the scale $m$ equals the coefficient modulus $Q = r \cdot m$. During addition, given that the scales are equal ($m_1 = m_2 = m_3$), the reserves for both operands and the result remain constant ($r_1 = r_2 = r_3$). In multiplication, where the resultant scale is the product of the operand scales ($m_3 = m_1 \cdot m_2$), and considering $m = Q/r$, the relationship among the reserves becomes $Q \cdot r_3 = r_1 \cdot r_2$. For `rescale` operations, though the scale changes by $1/R$, the reserve remains consistent, thus allowing reserve analysis to be independent from `rescale` placement.

To manage these reserves effectively and analyze operation latency efficiently, this work introduces the *reserve type system* (§6.3). This system supports the newly proposed *reserve analysis* (§6.4), a performance-aware static scale analysis for RNS-CKKS programs. The reserve type system tracks each ciphertext's reserve and the minimum required level to satisfy RNS-CKKS constraints, including waterline constraints.

Figure 6.2 demonstrates the reserve analysis process, which includes allocation ordering, reserve allocation, and reserve redistribution, illustrated using the same example program as in Figure 6.1a. The reserve analysis is performed at the function level, analyzing all operations within a function. During allocation ordering (§6.4.1), the analysis assesses each operation's cost, examines its dependency chain to return values, and prioritizes heavier (higher latency) operations. The reserve allocation (§6.4.2) assigns reserves based on the reserve type system from the end of the program to the start, using the minimal output reserve as a starting point. This backward approach helps in minimizing the levels and reserves required for each

$\boxed{\Gamma \vdash e : T}$ Under context $\Gamma$, $e$ has type $T$.

$$\frac{\Gamma \vdash e : \texttt{cipher}(\rho) \quad \rho' \leq \rho}{\Gamma \vdash e : \texttt{cipher}(\rho')} \text{ (Sub)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{cipher}(\rho) \quad \Gamma \vdash e_2 : \texttt{real}}{\Gamma \vdash e_1 + e_2 : \texttt{cipher}(\rho)} \text{ (PAdd)}$$

$$\frac{}{\Gamma \vdash c : \texttt{real}} \text{ (Const)}$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash -e : T} \text{ (Neg)}$$

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

$\boxed{\Gamma \vdash s : \Gamma'}$ Under context $\Gamma$, $s$ produces context $\Gamma'$.

$$\frac{\Gamma,\ \overline{v : T} \vdash s : \Gamma' \quad \Gamma' \vdash e : T'}{\Gamma \vdash \texttt{func}\ \mathit{fid}\ (\ \overline{v : T}\ )\ \{s; ret\ \overline{e}\} : \overline{T} \to \overline{T'}} \text{ (Func)}$$

$$\frac{\Gamma \vdash e : \texttt{cipher}(\rho)}{\Gamma \vdash \texttt{rotate}(e, i) : \texttt{cipher}(\rho)} \text{ (Rot)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{cipher}(\rho) \quad \Gamma \vdash e_2 : \texttt{cipher}(\rho)}{\Gamma \vdash e_1 + e_2 : \texttt{cipher}(\rho)} \text{ (Add)}$$

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

Figure 6.3: Typing rules of reserve type system reproduced from [30]. $W$ means the minimal scale required by `rescale` operation.

operation. Finally, the reserve redistribution (§6.4.3) reallocates reserves to prioritize chains of heavy operations, further reducing their operational levels.

Finally, this work introduces a new *rescale placement* algorithm (§6.5) that optimizes the placement of scale management operations to enhance performance. Initially using default positions determined by the outcomes of reserve allocation, the algorithm then adjusts these locations by lifting scale management operations to more cost-effective points based on a thorough cost analysis.

### 6.3 Reserve Type System

To streamline the processes of reserve analysis (§6.4) and the rescale placement algorithm (§6.5), this work introduces a novel reserve type system. The concept of a reserve, which represents the remaining scale budget needed for future RNS-CKKS operations, facilitates level reduction and consequently improves latency. Figure 6.3 illustrate the typing rules of this new reserve type system.

In subsequent discussions, this work utilizes logarithmic terms relative to the rescaling factor $R$: log-scale waterline ($\omega = \log_R W$), log-scale scale ($\mu = \log_R m$), and log-scale reserve ($\rho = \log_R r$) to simplify expressions. Additionally, $\lceil x \rceil$ represents the ceiling function, and $\{x\} = x + 1 - \lceil x \rceil$ denotes the fractional part function, where $\{1\} = 1$, not 0.

### 6.3.1 Rationale

A primary motivation for introducing the reserve type is to accurately pinpoint opportunities for level reduction during reserve analysis. Since the scale of any ciphertext must be greater than the waterline: $m = Q/r > W$, the level of the ciphertext must fulfill the condition $Q = R^l \geq W \cdot r$ (*i.e.,* $l \geq \omega + \rho$) for any given reserve $r$ and waterline $W$. The minimal level $l$ that satisfies this inequality, $l = \lceil \omega + \rho \rceil$, is termed the *principal level*. If a multiplication results in a different principal level from its operands, it is considered a *level-mismatch* operation, necessitating a `rescale` for its outcome. This formulation allows the reserve analysis to infer the principal level and the need for `rescale` solely from its reserve $\rho$ (and a given parameter $\omega$), thus identifying potential for level reduction.

Another significant reason for adopting the reserve type system is to decouple reserve analysis from the placement of scale management operations. Since a reserve represents a scale budget for subsequent operations and an `upscale` operation can adjust the reserve as necessary, any larger reserve can represent a smaller one. The introduction of a subtyping rule (Equation Sub) asserts that a larger reserve is a subtype of a smaller one, allowing the reserve analysis to bypass explicit scale management operations due to type system-permitted implicit conversions between types.

### 6.3.2 Typing Rules

Figure 6.3 displays the typing rules for the reserve type system. Here, this work focuses on discussing the subtyping rule (Equation Sub) and the rule for ciphertext multiplication (Equation Mul). Unary and addition operations maintain consistent types between operands and results.

**Subtyping:** The reserve type system establishes a subtyping rule, Equation Sub, which abstracts away scale management operations within the type system framework. A `cipher` type with a certain reserve $r$ accepts another `cipher` type with a smaller reserve $r'$, achievable by any sequence of scale management operations: `upscale`, `rescale`, and `modswitch`. As `upscale` can decrease the reserve as required, any reserve $r' \leq r$ (*i.e.,* $\rho' \leq \rho$) qualifies as a subtype. Reserves remain unchanged by `rescale`, and `modswitch` combines `upscale` and `rescale`.

**Multiplication:** The multiplication typing rule (Equation Mul) integrates both level and waterline constraints. For operands $e_1$ and $e_2$ with reserves $r_1$ and $r_2$, and result $e_1 \times e_2$ having reserve $r$, the condition $r_1 \cdot r_2 = r \cdot R^l$ (*i.e.,* $\rho_1 + \rho_2 = \rho + l$) must hold, with $l$ being the common principal level derived from the waterline constraint ($l = \lceil \rho_1 + \omega \rceil = \lceil \rho_2 + \omega \rceil$). Another multiplication rule (Equation PMul) addresses ciphertext-plaintext multiplication, presuming the plaintext is encoded at the waterline W (*i.e.,* $\rho_2 = l - \omega$).

These typing rules facilitate the backward distribution of reserves, and the detailed application of these rules in backward analysis is explained in §6.4.2.

## 6.4 Reserve Analysis

The objective of reserve analysis is to minimize the principal levels to decrease the latency of each operation. This approach differs from traditional forward analysis, which starts with a

fixed input scale and calculates the output scale for each operation. In contrast, backward reserve analysis begins with a determined output reserve and assesses the necessary input reserve for each operation. This method allows for direct impact on an operation's level by minimizing its reserve, enabling more aggressive level reduction than forward analysis, which does not adjust individual operation levels.

The core strategy of reserve analysis is to aggressively lower the level of operations that are latency-intensive (e.g., `rotate`) by shifting the level mismatch earlier in the program's sequence. This process involves prioritizing operations with high latency through allocation ordering to enhance the effect of level reduction (§6.4.1). It then assigns reserves to ciphertexts according to this order (§6.4.2) and adjusts the timing of level mismatches through reserve redistribution (§6.4.3) during the reserve allocation phase.

### 6.4.1 Allocation Ordering

Allocation ordering sets the sequence for the reserve analysis, emphasizing the processing of operations that significantly affect total latency. This approach recognizes that the level of a high-latency operation has a greater impact on overall program performance, and thus, the reserve allocation should prioritize these operations, especially in managing level mismatches. This method uses scale management unit that groups operations with similar arithmetic structure and identical multiplicative depths to streamline analysis and reduce redundancy, thereby speeding up the process.

To determine which operations to prioritize, the algorithm first calculates the estimated latency for each operation. Although the operation type and its level typically determine latency, the exact level isn't known until after reserve allocation. Consequently, the algorithm uses an initial estimate based on the multiplicative depth and a predefined waterline, calculated

as $1 + \text{depth} \times \omega$, representing the minimal level increase for each multiplication, where $\omega$ is derived from the ratio of the waterline to the rescaling factor ($\omega = \log_R W$).

For instance, considering the operation $x^3$ from the same example in Figure 6.1a, it has a multiplicative depth of 2. Assuming $\omega = 1/3$ (from $20/60$), the level is estimated as $1 + 2 \times 1/3$. The latency cost, interpolated from the latency table, would then be calculated as $44 \times 1/3 + 92 \times 2/3 = 76$, considering the costs at levels 1 and 2.

After estimating the latencies, the ordering algorithm prioritizes operations based on their dependencies, particularly emphasizing those that follow a heavy operation, as their reserves need allocation before that of the heavy operation. This prioritization follows the longest dependency chain leading from the operation to the program's output. In case of equal depths in different chains, operations lower in the chain are prioritized. For operations of the same depth and type, the algorithm uses the dependency chain of the next heaviest operation as a tie-breaker.

Figure 6.2b illustrates this ordering. The algorithm identifies the longest dependency chain for the heaviest operation $x^2$, prioritizing the operations as $q > x^3 > x^2$. As $x^3$'s dependency chain is encompassed within $x^2$'s, it's skipped, and the algorithm proceeds to sequence the operations linked to the next heaviest operation, $y^2$, and continues in this manner.

### 6.4.2 Reserve Allocation

Following the allocation order determined previously, the reserve allocation process assigns a reserve value to each ciphertext in the program. It accomplishes this by selecting the highest incoming reserve value (reserve-in) from its dependencies, effectively choosing the least restrictive reserve that satisfies all incoming conditions.

For instance, Figure 6.2c visualizes how reserves are assigned in a reversed (backward) dependency graph. In the diagram, the gray numbers indicate incoming and outgoing reserves, while the black numbers represent the finally assigned reserve for each result. Initially, the reserve for $q$ is set to 0, and the process for this starting point will be detailed later. As depicted, $x$ receives two incoming reserve values, 97 and 75, from which the algorithm selects the higher value, 97, to ensure it satisfies the most restrictive incoming condition.

Subsequently, the reserve allocation uses the typing rules outlined in Figure 6.3 to infer the outgoing reserve (reserve-out). For operations other than multiplication, the operand reserves match the result reserve, allowing the algorithm to straightforwardly set the reserve-out equal to the assigned reserve.

For operations involving multiplication, where the operand reserves often differ from the result reserve, the algorithm calculates operand reserves based on the result reserve. Specifically, for plaintext multiplication, it computes the operand reserve as $\rho_{op} = \rho + \omega$ for a given ciphertext reserve $\rho$. If this operand reserve fails to meet the waterline condition $l \geq \rho_{op} + \omega = \rho + 2\omega$, indicating a level mismatch, the operation is flagged accordingly.

For ciphertext multiplication, the algorithm determines operand reserves $\rho_1$ and $\rho_2$ based on the result reserve $\rho$ and the operand level $l$, which itself depends on the operand reserves. The equation $\rho_1 + \rho_2 = \rho + l$ from Equation Mul guides this determination. The algorithm first finds the level $l$ from $\lceil \rho + l + 2\omega \rceil = \lceil \rho_1 + \rho_2 + 2\omega \rceil \leq \lceil \rho_1 + \omega \rceil + \lceil \rho_2 + \omega \rceil = 2l$. By ensuring $l = \lceil \rho + 2\omega \rceil$, the algorithm achieves this by evenly distributing the reserves to each operand:

$$\rho_1 = \rho_2 = \frac{(l + \rho)}{2} \quad \text{where} \quad l = \lceil \rho + 2\omega \rceil$$

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

Taking $q$ from Figure 6.2c as an example, with a reserve of 0 ($\rho = 0/60$) and a waterline of 20 ($\omega = 20/60$), this work finds $l = \lceil \rho + 2\omega \rceil = \lceil 40/60 \rceil = 1$. Consequently, $\rho_1 = \rho_2 = \frac{(\rho+l)}{2} = 30/60$, leading to an outgoing reserve of 30. The algorithm then continues this process for $x^3$, calculating its reserve-in at 30, and determining $l_{x^3} = \lceil \rho_{x^3} + 2\omega \rceil = \lceil 30/60 + 40/60 \rceil = 2$. This calculation flags $x^3$ as having a level mismatch due to the differences in the calculated levels.

### 6.4.3 Reserve Redistribution

The reserve redistribution algorithm aims to eliminate unnecessary level increments introduced during the reserve allocation phase, which equally divides reserves among operation operands. Level mismatches occur when the levels of results and operands differ, identified as $\lceil \rho + 2\omega \rceil \neq \lceil \rho + \omega \rceil$. To resolve this, the algorithm reduces the reserve $\rho$ by the fractional overflow $\{\rho + 2\omega\}$, effectively preventing the level mismatch.

The adjusted reserve for a ciphertext must be the highest among its incoming reserve values, meaning all reserves feeding into it must be less than or equal to $\rho - \{\rho + 2\omega\}$.

To correct avoidable level mismatches, the redistribution algorithm inspects each operation influenced by a level-mismatched multiplication. If an operation, such as addition, cannot redistribute its reserve, the algorithm recursively searches for a feasible redistribution target until it achieves the necessary reserve reduction. For ciphertext multiplication operations, the reserve can be adjusted across operands, where one operand compensates for the mismatch of the other. Redistribution prioritizes operations based on their computational weight; if the target operation (where reserves might be increased) is prioritized higher than the mismatched operation, the reserve increment at the target is constrained to ensure no reserve exceeds its calculated upper limit ($\rho_{redist} = \rho_{target} - \rho_{target,user}$). However, if the redistribution target

has lower priority, reserve increases are less restricted, provided they do not alter the principal level of the operands.

Figure 6.2e shows a scenario where the redistribution target is of lower priority than the level-mismatched multiplication. In this case, $x^3$ has higher priority than $y^2$, allowing for reserve redistribution up to the point where it does not change the principal level. The maximum permissible reserve redistribution in this example is 10, which aligns with the required redistribution amount of 10 ($\{\rho + 2\omega\} = \{30/60 + 2 \cdot 20/60\} = 10/60$), hence the redistribution is successful.

Figure 6.2d displays the outcomes of the reserve allocations following the redistribution adjustments, showing how the operations' reserves are optimized to prevent unnecessary level increments while maintaining the operational dependencies and constraints of the RNS-CKKS scheme.

### 6.5 Code Generation: Rescale Placement

The rescale placement algorithm includes two main steps: inserting scale management operations and hoisting rescale operations.

In the first step, the algorithm translates a reserve-typed program into one that complies with RNS-CKKS constraints, by appropriately placing scale management operations. It inserts a `rescale` operation for results of level-mismatched multiplications like $x^2$ and $y^2$ in Figure 6.2f, where the result's principal level differs from the operand's principal level. Additionally, it adds `upscale` and `rescale` operations to adjust the scales and levels between the reserve-in and the ciphertexts, such as for $x$ and $y$, when their principal levels and reserves do not match. This step resolves discrepancies in reserve and level due to subtyping mismatches.

The second step, rescale hoisting, involves moving a `rescale` operation to a later position if it leads to lower overall latency or cost. The reserve analysis initially places level-mismatched operations at the earliest possible point in the program. Therefore, any rescale hoisting moves the operation further down the program flow. The hoisting process begins by evaluating potential benefits using dynamic programming, considering three cost factors: the increased levels from hoisting, the original location of the `rescale` operation, and its new location. For instance, in Figure 6.2h, hoisting the rescale operation increases the level of $s$ from 1 to 2, adding a cost of 1. The cost from merging two `rescale` operations into one results in a savings of 19, leading to a net benefit of 18.

Even if a particular hoisting does not immediately yield benefits, the algorithm retains the destination `rescale` as a potential candidate for further hoisting. If a more advantageous hoisting opportunity arises—one that offsets the current costs—the algorithm will execute the hoisting. Subsequently, the program is transformed; if the original `rescale` is used only once, it can be removed.

## 6.6 Evaluation of Performance-aware Static Scale Analysis

This research implements the proposed reserve analysis and reserve type system, alongside code transformations, within the MLIR compiler framework [80]. For the RNS-CKKS backend, this study employs Microsoft SEAL [23] (Release 3.6.1). The evaluation includes a comparison with leading frameworks such as EVA [26] and HECATE, focusing on both compilation time and runtime latency performance. For the benchmarks, this research uses the same sets as those employed by HECATE, with the addition of a new benchmark, Lenet-CIFAR (Lenet-C). The evaluations are conducted under the same experimental settings as described in §4.6.1.

### 6.6.1 Compilation Time

Table 6.2 displays the compilation times for EVA, HECATE, and this work. The column labeled *# op* indicates the number of operations in each program, reflecting its size. The *# iter* column shows the number of scale management plans explored in HECATE, representing the program's complexity. For instance, the MLP program has 462 operations, which is four times more than LR, but its iteration count is four times smaller. This is because MLP involves simpler operations like two matrix multiplications and two squarings on a single input, avoiding operations across different multiplicative depths. In contrast, LR involves subtractions across different depths and two distinct inputs, necessitating more extensive exploration of scale management plans. The most complex benchmarks are Lenet-5 and Lenet-C, with Lenet-C needing slightly fewer iterations but involving more operations.

Compilation time encompasses I/O operations and processing time for scale management and other optimizations like common subexpression elimination and dead code elimination. Without iterative exploration, this work achieves an average speedup over HECATE. Most of this work's compilation time derives from processing and I/O, with scale management time accounting for only 0.14% of the total.

Particularly noteworthy are the Lenet-5 and Lenet-C benchmarks, where HECATE takes a considerable amount of time for scale management. The structure of LeNet, which includes Convolution, $x^2$, Average Pooling, and Fully Connected layers, results in 11 multiplicative depths, and operations like rotation and addition occur in Conv, Avg, and FC layers. In our tests, HECATE identifies over 40 potential places for inserting scale management operations, requiring brute-force searches exceeding $2^{40}$ iterations.

Regarding scale management time, this work achieves an average speedup over HECATE, mainly due to eliminating exploration. The theoretical speedup aligns with the *# iter*, yet

Table 6.2: Compile time of EVA, HECATE, and this work, reproduced from [30]. (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 |

this work surpasses it. The ratio of actual to theoretical speedup averages at about 5.744x, thanks to the differences in iteration weight between HECATE, which incorporates multiple optimization passes in its exploration phase to accurately gauge performance, and this work, which excludes additional optimizations in scale management. Moreover, this work outperforms EVA by a factor of 15, largely due to using scale management units similar to HECATE, which effectively reduces the analysis space.

### 6.6.2 Performance

Figure 6.4 illustrates the latency of programs compiled by EVA, HECATE, and this work. The latency data from EVA demonstrates the limitations of its forward analysis, where the scale management scheme struggles to control level and latency effectively. HECATE illustrates the benefits of performance-aware optimization, although it incurs high exploration costs. This work matches the latency performance of HECATE for the same parameters and shows an average performance improvement over EVA.

Generally, this work parallels HECATE in performance across most parameters. It even achieves up to an 8.7% better performance on certain parameters in benchmarks such as SF, HCD, Lenet-5, and Lenet-C. HECATE relies on a hill-climbing method that often converges to local optima rather than the global optimum, potentially missing more effective scale management plans. Conversely, this work sometimes shows a slowdown of up to 6.5% on certain parameters in LR, PR, and MR benchmarks. This discrepancy arises from the latency associated with `rescale` operations, where the rescale placement algorithm fails to optimally hoist `rescale` operations in scenarios involving multiple uses.

The comparison of program errors compiled by EVA, HECATE, and this work for two different waterlines is presented in Figure 6.5. HECATE frequently uses `downscale` to minimize the scale of each ciphertext. However, reducing the scale can inadvertently

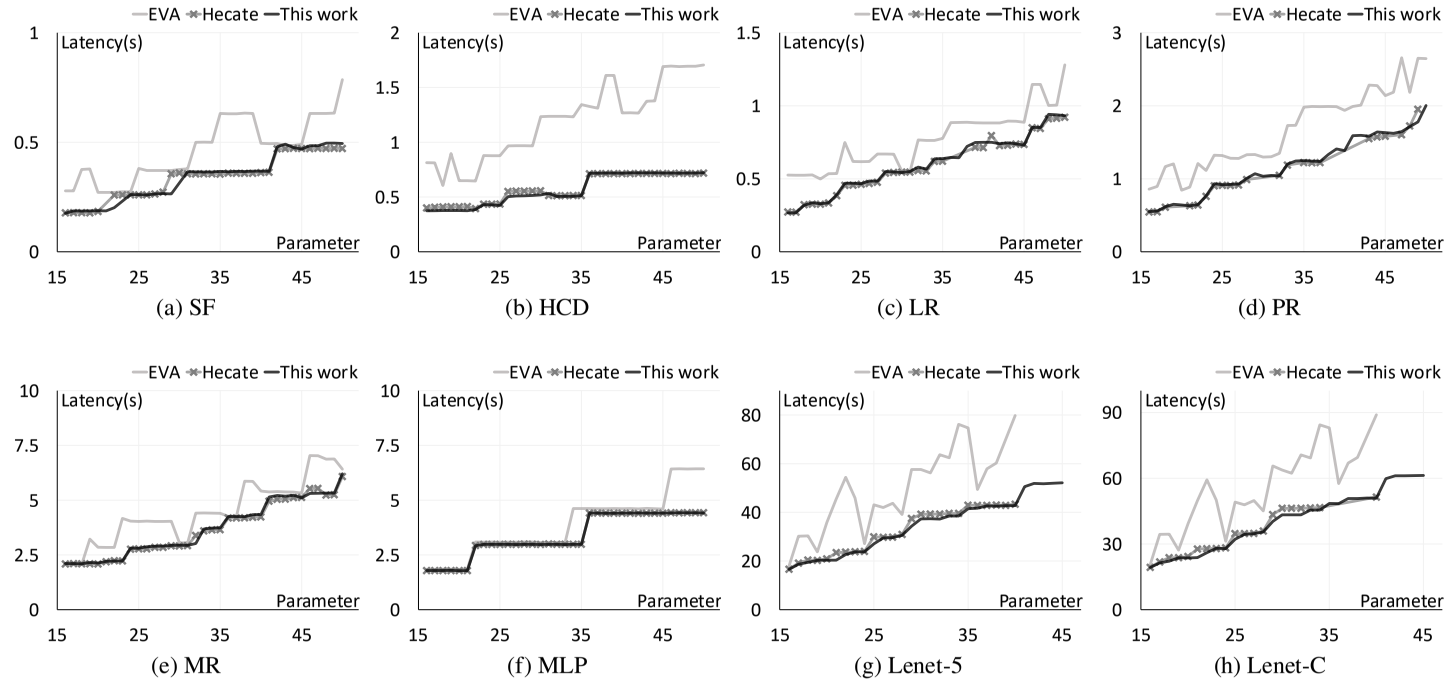
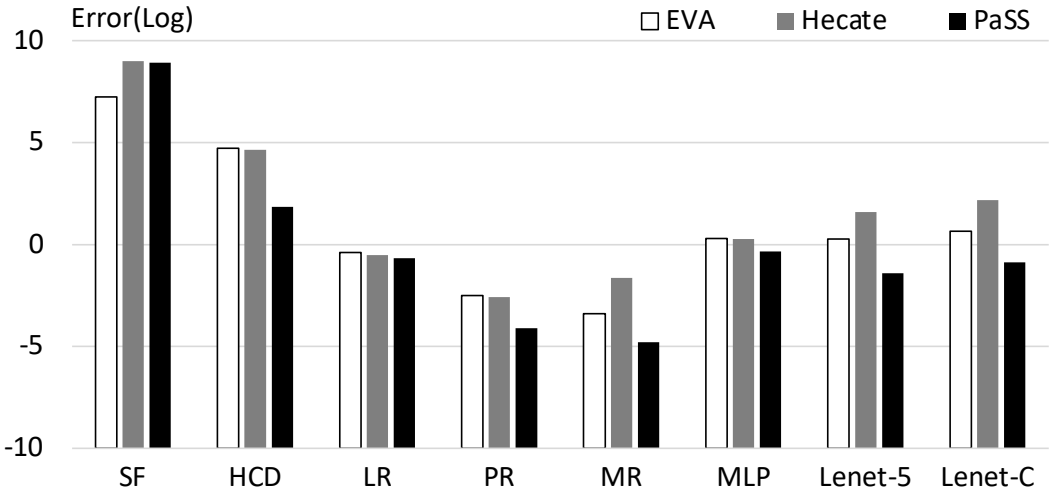
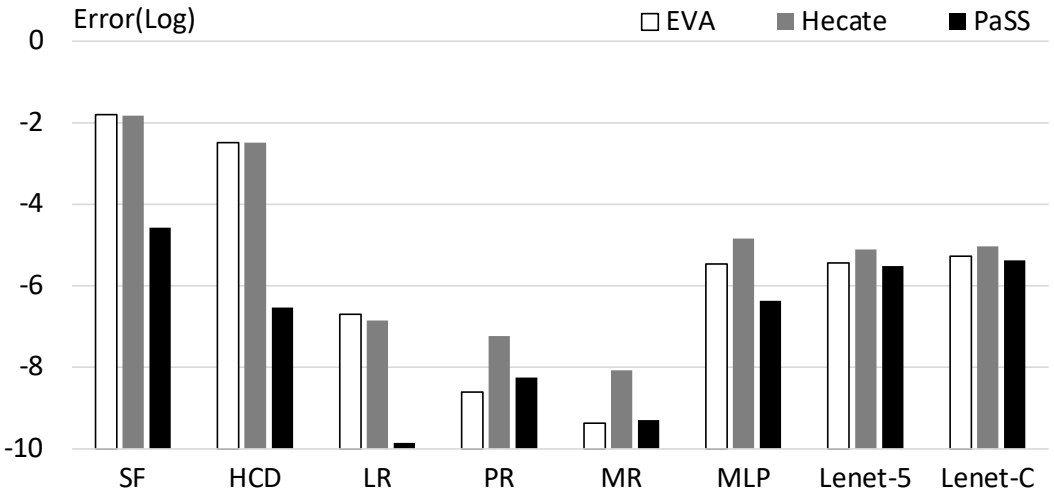



Figure 6.4: Latency comparison of EVA, HECATE, and this work for a set of waterline parameter (15-50), reproduced from [30].




(a) Waterline = $2^{20}$




(b) Waterline = $2^{40}$

Figure 6.5: Error comparison of EVA, HECATE, and this work for two different waterlines ($W$), reproduced from [30].

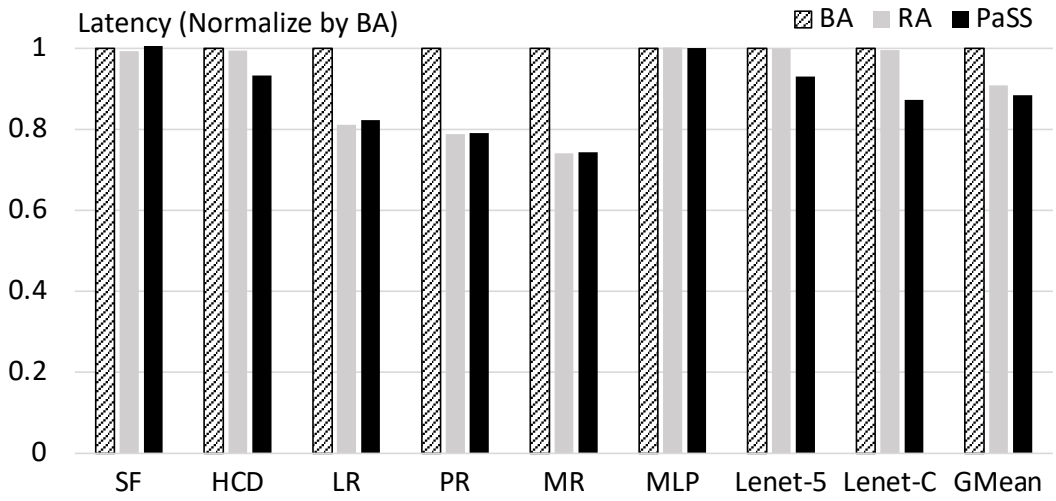
increase the error, as the noise introduced by RNS-CKKS operations is invariant relative to the ciphertext scale, and error is defined as noise over scale. In contrast, this work considers the cascading effect with its reserve analysis, avoiding unnecessary scale reductions unless they enhance performance. This approach provides more opportunities to increase the scale of each ciphertext, generally reducing errors without compromising performance. Thus, surprisingly, this work tends to produce better error rates for the parameters overall.

### 6.6.3 Performance Improvement Breakdown

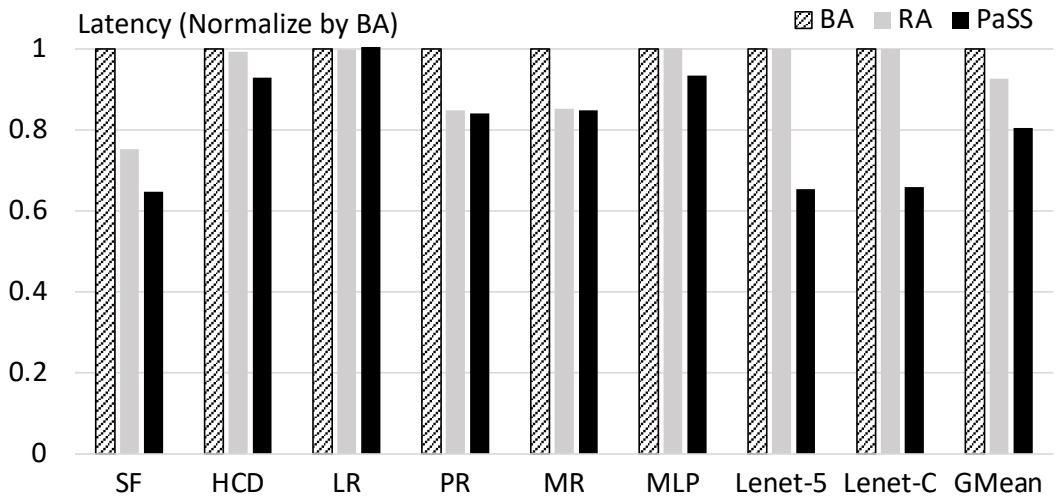
Figure 6.6 displays the performance breakdown of the proposed algorithms for two different waterlines. BA represents the basic implementation of reserve-based backward analysis without scale redistribution (§6.4.3) or rescale placement (§6.5). RA includes scale redistribution but excludes rescale placement. On average, RA and this work improve performance by 9.1% and 11.6% over BA for $W = 20$, and by 7.4% and 19.6% for $W = 40$, respectively.

The performance gains from each algorithm vary depending on the benchmark type, as shown in Figure 6.6a and Figure 6.6b. For instance, RA shows no speedup over BA in MLP, Lenet-5, and Lenet-C. The benefits of RA mainly come from scale redistribution, which is impactful in scenarios involving multiplications between different ciphertexts. However, in deep learning benchmarks, many of the multiplications are squarings of the same ciphertext, which limits the effectiveness of scale redistribution.

Conversely, this work does not exhibit noticeable speedups over RA in the regression benchmarks LR, MR, and PR. The additional performance improvements in this work come from rescale placement, which is more effective in scenarios involving additions between different ciphertexts. Specifically, ciphertext summation is divided into two types: internal summation, which aggregates data within a single ciphertext and often requires rotation, and external summation, which combines data from different ciphertexts. The rescale placement




(a) Waterline = $2^{20}$




(b) Waterline = $2^{40}$

Figure 6.6: Breakdown comparison between backward analysis excepting the reserve redistribution and rescale placement (BA), reserve allocation excepting rescale placement (RA), and this work, reproduced from [30].

strategy tends to be less effective in internal summation scenarios, common in regression benchmarks, hence the lack of observed speedup in these cases.

### 6.7 Summary

This work introduces a new performance-aware static scale analysis for RNS-CKKS, called "reserve analysis." It utilizes the innovative concept of a "reserve" and an accompanying type system, allowing for separation of the reserve analysis from scale management operations. With the "reserve allocation" algorithm, this approach statically optimizes scales and levels for each ciphertext, enhancing efficiency. Additionally, the "rescale placement" algorithm strategically determines the most effective positions for rescale operations, maximizing performance.

In comparison to traditional exploration-based scale management, this method achieves similar performance enhancements (41.8% speedup over conventional conservative static analysis) and offers 15526$\times$ faster scale management time. This streamlined scale management approach facilitates a broad spectrum of optimizations, such as data layout selection and bootstrap insertion. Although the proposed algorithms are heuristic—prioritizing satisfactory solutions with minimal compilation time over exhaustive searches for globally optimal solutions—they set the stage for significant improvements in homomorphic encryption optimizations. Future work will explore applying these scale management schemes to global-level scale optimization in homomorphic encryption, aiming to validate their efficacy and optimize their implementation further.

# CHAPTER 7

# CONCLUSION

This dissertation consolidates developments in scale management for fully homomorphic encryption (FHE) across three pioneering works, each introducing innovative approaches to optimize computational efficiency and accuracy within RNS-CKKS encryption systems.

## 7.1 Contributions

The first research details the HECATE compiler framework, which introduces performance-aware scale management facilitated by a robust type system and innovative scale management operations, including a newly developed `downscale` operation. HECATE's core innovation lies in its proactive rescaling algorithm and an extensive scale management space explorer that together reduce computational overhead and dynamically optimize execution pathways. This approach not only advances the compiler design for FHE but also achieves a remarkable 27.38% speedup over existing methods, significantly enhancing the computational efficiency of FHE operations.

The second research introduces the Error-Latency-Aware Scale Management (ELASM), an advanced method that refines scale management by focusing on minimizing error and latency through an iterative exploration of scale management plans. At the heart of ELASM is the scale-to-noise ratio (SNR), a novel error-proportional parameter that, along with fine-grained noise-aware waterlines, broadens the exploration space for scale management. Implemented in the ELASM compiler, this framework demonstrates superior performance

on benchmarks against leading compilers like EVA and HECATE, setting new benchmarks in the precision and efficiency of homomorphic encryption. ELASM provides 21.3% and 31.2× better latency and error for a given error and latency over HECATE, respectively.

The third research presents a new approach called "reserve analysis," a performance-aware static scale analysis that utilizes a newly introduced concept of "reserve" and a corresponding type system to separate reserve analysis from scale management operations. This method contrasts with traditional exploration-based scale management by offering similar performance improvements and faster scale management time. The reserve analysis promotes a range of further optimizations, marking a significant stride toward efficient and practical homomorphic encryption practices. This method achieves similar performance enhancements (41.8% speedup over conventional conservative static analysis) and offers 15526× faster scale management time.

### 7.2 Future Work

FHE compiler can perform several different optimizations to improve the performance of the FHE application. In the front-end of scale management, scale management can incorporate range analysis and automatic bootstrapping. By using the exact range analysis, the error estimation can be improved and thus generate more efficient code. On the other hand, to support a larger application like ResNet, automatic bootstrapping management is required. Bootstrapping management and scale management are tightly coupled since scale management determines the amount of allowed computation without bootstrapping and bootstrapping management determines the point of reinitialization of a ciphertext. Although the proposed scale management can easily support bootstrapping, bootstrapping management is not integrated. Further research on the integration of bootstrapping management and the proposed scale management can greatly increase the applicability.

For the scale management, optimal scale management remains an open issue. Although the proposed scale management schemes in this work achieve the best error-latency trade-off and fast compilation, the optimal solution is not known for the scale management problem, preventing the exact evaluation of the quality of the solution. Because finding the optimal scale management solution in large FHE program is not viable due to the long compilation time, the optimal solution can evaluate the quality of the proposed schemes and give the insight about the better scale management scheme.

In the back-end of scale management, a fine-grained (residue-polynomial-level) code generation allows further optimization, but it is not addressed well. With fine-grained code generation, compiler can further optimize the program by hoisting the heavy internal computation like NTT, especially removing the overhead of scale management operations. By doing so, scale management schemes can reflect the NTT hoisting for the rescale to reduce the scale management overhead. Nonetheless, the scheduling and buffer allocation for the fine-grained FHE computation code is also an open problem.

### 7.3 Summary

This dissertation consists of three new scale management schemes that provide distinct insights. Together, these researches present a comprehensive advancement in FHE scale management, each contributing unique insights and tools that pave the way for future research in the field. This dissertation not only demonstrates the individual merits of each approach but also highlights the potential for their integration into a cohesive scale management strategy that could further revolutionize the landscape of secure cryptographic computation.

# REFERENCES


[1] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A Survey on Homomorphic Encryption Schemes: Theory and Implementation," *ACM Comput. Surv.*, vol. 51, no. 4, Jul. 2018.

[2] O. Kocabas, T. Soyata, J.-P. Couderc, M. Aktas, J. Xia, and M. Huang, "Assessment of cloud-based health monitoring using Homomorphic Encryption," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013.

[3] O. Masters *et al.*, "Towards a Homomorphic Machine Learning Big Data Pipeline for the Financial Services Sector," in *Real World Crypto*, 2020.

[4] D. Archer *et al.*, "Applications of homomorphic encryption," *HomomorphicEncryption. org, Redmond WA, Tech. Rep.*, 2017.

[5] Ö. Kocabaş and T. Soyata, "Medical data analytics in the cloud using homomorphic encryption," in *E-Health and Telemedicine: Concepts, Methodologies, Tools, and Applications*, IGI Global, 2016.

[6] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Manual for Using Homomorphic Encryption for Bioinformatics," Tech. Rep. MSR-TR-2015-87, Nov. 2015.

[7] J. 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/, Jul. 2020.

[8] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full rns variant of approximate homomorphic encryption," in *Selected Areas in Cryptography – SAC 2018*, C. Cid and M. J. Jacobson Jr., Eds., Cham: Springer, 2018, ISBN: 978-3-030-10970-7.

[9] C. Gentry, "A Fully Homomorphic Encryption Scheme," Ph.D. dissertation, Stanford, CA, USA, 2009, ISBN: 9781109444506.

[10] C. Gentry, "Fully Homomorphic Encryption Using Ideal Lattices," in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, Bethesda, MD, USA: ACM, 2009.

[11] C. Gentry and S. Halevi, "Implementing Gentry's Fully-Homomorphic Encryption Scheme," vol. 6632, May 2011, ISBN: 978-3-642-20464-7.

[12] J.-S. Coron, A. Mandal, D. Naccache, and M. Tibouchi, "Fully Homomorphic Encryption over the Integers with Shorter Public Keys," in *Advances in Cryptology – CRYPTO 2011*, vol. 6841, Aug. 2011, ISBN: 978-3-642-22791-2.

[13] J.-S. Coron, D. Naccache, and M. Tibouchi, "Public Key Compression and Modulus Switching for Fully Homomorphic Encryption over the Integers," in *Advances in Cryptology – EUROCRYPT 2012*, Apr. 2012.

[14] J. H. Cheon *et al.*, "Batch fully homomorphic encryption over the integers," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2013.

[15] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology – ASIACRYPT 2017*, T. Takagi and T. Peyrin, Eds., Cham: Springer, 2017, ISBN: 978-3-319-70694-8.

[16] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) Fully Homomorphic Encryption without Bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, Cambridge, Massachusetts: ACM, 2012.

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

[18] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," *SIAM Journal on Computing*, vol. 43, no. 2, 2014.

[19] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical GapSVP," in *Annual Cryptology Conference*, Springer, 2012.

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

[21] C. Gentry, S. Halevi, and N. P. Smart, "Better bootstrapping in fully homomorphic encryption," in *International Workshop on Public Key Cryptography*, Springer, 2012.

[22] A. Viand, P. Jattke, and A. Hithnawi, "SoK: Fully Homomorphic Encryption Compilers," in *2021 2021 IEEE Symposium on Security and Privacy (SP)*, IEEE Computer Society, May 2021.

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

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

[25] *FullRNS-HEAAN*, `https://github.com/KyoohyungHan/FullRNS-HEAAN`, 2018.

[26] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, London, UK: ACM, 2020, ISBN: 978-1-4503-7613-6.

[27] R. Dathathri *et al.*, "CHET: An Optimizing Compiler for Fully-homomorphic Neural-network Inferencing," in *Proceedings of the 40th ACM SIGPLAN Conference on*

*Programming Language Design and Implementation*, (Phoenix, AZ, USA), New York, NY, USA: ACM, 2019, ISBN: 978-1-4503-6712-7.

[28] Y. Lee *et al.*, "HECATE: Performance-Aware Scale Optimization for Homomoprhic Encryption Compiler," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2022.

[29] Y. Lee, S. Cheon, D. Kim, D. Lee, and H. Kim, "ELASM: Error-Latency-Aware Scale Management for Fully Homomorphic Encryption," in *Proceedings of the 32nd USENIX Conference on Security Symposium*, ser. SEC '23, Anaheim, CA, USA: USENIX Association, 2023, ISBN: 978-1-939133-37-3.

[30] Y. Lee, S. Cheon, D. Kim, D. Lee, and H. Kim, "Performance-aware Scale Analysis with Reserve for Homomorphic Encryption," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '24, La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 302–317, ISBN: 9798400703720.

[31] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Annual Cryptology Conference*, Springer, 2013.

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

[33] A. Kim, A. Papadimitriou, and Y. Polyakov, "Approximate homomorphic encryption with reduced approximation error," in *Cryptographers' Track at the RSA Conference*, Springer, 2022, pp. 120–144.

[34] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski, "nGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, (Alghero, Italy), ser. CF '19, New York, NY, USA: ACM, 2019, pp. 3–13, ISBN: 978-1-4503-6685-4.

[35] F. Boemer, A. Costache, R. Cammarota, and C. Wierzynski, "nGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data," in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, (London, United Kingdom), ser. WAHC'19, New York, NY, USA: ACM, 2019, pp. 45–56, ISBN: 978-1-4503-6829-2.

[36] *PALISADE Lattice Cryptography Library*, `https://palisade-crypto.org/`, Oct. 2020.

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

[38] D. W. Archer *et al.*, "RAMPARTS: A Programmer-Friendly System for Building Homomorphic Encryption Applications," in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, (London, United

Kingdom), ser. WAHC'19, New York, NY, USA: ACM, 2019, pp. 57–68, ISBN: 978-1-4503-6829-2.

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

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

[41] D. Lee, W. Lee, H. Oh, and K. Yi, "Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Term Rewriting," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, London, UK: Association for Computing Machinery, 2020, pp. 503–518, ISBN: 9781450376136.

[42] A. Viand, P. Jattke, M. Haller, and A. Hithnawi, "HECO: Automatic Code Optimizations for Efficient Fully Homomorphic Encryption," in *32nd USENIX Security Symposium (USENIX Security 23)*, Aneheim, CA: USENIX Association, Aug. 2023.

[43] A. Viand and H. Shafagh, "Marble: Making Fully Homomorphic Encryption Accessible to All," in *Proceedings of the 6th Workshop on Encrypted Computing; Applied Homomorphic Cryptography*, ser. WAHC '18, Association for Computing Machinery, 2018.

[44] E. Chielle, O. Mazonka, H. Gamil, N. G. Tsoutsos, and M. Maniatakos, *E3: A Framework for Compiling C++ Programs with Encrypted Operands*, Cryptology ePrint Archive, Report 2018/1013, `https://ia.cr/2018/1013`, 2018.

[45] E. Crockett, C. Peikert, and C. Sharp, "ALCHEMY: A Language and Compiler for Homomorphic Encryption Made EasY," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, Association for Computing Machinery, 2018.

[46] M. Cowan, D. Dangwal, A. Alaghi, C. Trippel, V. T. Lee, and B. Reagen, "Porcupine: A Synthesizing Compiler for Vectorized Homomorphic Encryption," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, Association for Computing Machinery, 2021, pp. 375–389.

[47] S. Gorantala *et al.*, "A general purpose transpiler for fully homomorphic encryption," *arXiv preprint arXiv:2106.07893*, 2021.

[48] R. Malik, K. Sheth, and M. 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*, ser. ASPLOS 2023, Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 118–133, ISBN: 9781450399180.

[49] S. Cheon *et al.*, "DaCapo: Automatic bootstrapping management for efficient fully homomorphic encryption," in *33rd USENIX Security Symposium (USENIX Security 24)*.

[50] H. Chen, R. Cammarota, F. Valencia, F. Regazzoni, and F. Koushanfar, "AHEC: End-to-end Compiler Framework for Privacy-preserving Machine Learning Acceleration," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020.

[51] K. Han and D. Ki, "Better bootstrapping for approximate homomorphic encryption," in *Topics in Cryptology – CT-RSA 2020*, S. Jarecki, Ed., Cham: Springer International Publishing, 2020, pp. 364–390, ISBN: 978-3-030-40186-3.

[52] J.-W. Lee, E. Lee, Y. Lee, Y.-S. Kim, and J.-S. No, "High-precision bootstrapping of rns-ckks homomorphic encryption using optimal minimax polynomial approximation and inverse sine function," in *Advances in Cryptology – EUROCRYPT 2021*, A. Canteaut and F.-X. Standaert, Eds., Cham: Springer International Publishing, 2021, pp. 618–647, ISBN: 978-3-030-77870-5.

[53] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, "Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys," in *Advances in Cryptology – EUROCRYPT 2021*, A. Canteaut and F.-X. Standaert, Eds., Cham: Springer International Publishing, 2021, pp. 587–617, ISBN: 978-3-030-77870-5.

[54] J.-P. Bossuat, J. Troncoso-Pastoriza, and J.-P. Hubaux, "Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret

encapsulation," in *Applied Cryptography and Network Security*, G. Ateniese and D. Venturi, Eds., Cham: Springer International Publishing, 2022, pp. 521–541, ISBN: 978-3-031-09234-3.

[55] J.-W. Lee, E. Lee, Y.-S. Kim, and J.-S. No, "Rotation Key Reduction for Client-Server Systems of Deep Neural Network on Fully Homomorphic Encryption," in *Advances in Cryptology – ASIACRYPT 2023*, J. Guo and R. Steinfeld, Eds., vol. 14443, Singapore: Springer Nature Singapore, 2023, pp. 36–68, ISBN: 978-981-9987-35-1 978-981-9987-36-8.

[56] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 114–148, Aug. 11, 2021.

[57] S. Kim, K. Lee, W. Cho, Y. Nam, J. H. Cheon, and R. A. Rutenbar, "Hardware Architecture of a Number Theoretic Transform for a Bootstrappable RNS-based Homomorphic Encryption Scheme," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2020, pp. 56–64.

[58] J. Kim *et al.*, "ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse," presented at the 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO), IEEE Computer Society, Oct. 1, 2022, pp. 1237–1254, ISBN: 978-1-66546-272-3.

[59] S. Kim *et al.*, "BTS: An Accelerator for Bootstrappable Fully Homomorphic Encryption," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22, New York, NY, USA: Association for Computing Machinery, June 11, 2022, pp. 711–725, ISBN: 978-1-4503-8610-4.

[60] N. Samardzic *et al.*, "CraterLake: A Hardware Accelerator for Efficient Unbounded Computation on Encrypted Data," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22, New York, NY, USA: Association for Computing Machinery, June 11, 2022, pp. 173–187, ISBN: 978-1-4503-8610-4.

[61] N. Samardzic *et al.*, "F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21, New York, NY, USA: Association for Computing Machinery, October 17, 2021, pp. 238–252, ISBN: 978-1-4503-8557-2.

[62] R. Agrawal *et al.*, "FAB: An FPGA-based Accelerator for Bootstrappable Fully Homomorphic Encryption," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 882–895.

[63] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An Architecture for Computing on Encrypted Data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20, New York, NY, USA: Association for Computing Machinery, March 13, 2020, pp. 1295–1309, ISBN: 978-1-4503-7102-5.

[64] R. Agrawal, L. De Castro, C. Juvekar, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "MAD: Memory-Aware Design Techniques for Accelerating Fully Homomorphic Encryption," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23, New York, NY, USA: Association for Computing Machinery, December 8, 2023, pp. 685–697, ISBN: 9798400703294.

[65] Y. Yang, H. Zhang, S. Fan, H. Lu, M. Zhang, and X. Li, "Poseidon: Practical Homomorphic Encryption Accelerator," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2023, pp. 870–881.

[66] J. Kim, S. Kim, J. Choi, J. Park, D. Kim, and J. H. Ahn, "SHARP: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.

[67] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "GAZELLE: A low latency framework for secure neural network inference," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1651–1669.

[68] Z. Huang, W.-j. Lu, C. Hong, and J. Ding, "Cheetah: Lean and Fast Secure {Two-Party} Deep Neural Network Inference," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 809–826.

[69] D. Rathee *et al.*, "CrypTFlow2: Practical 2-party secure inference," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 325–342.

[70] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference system for neural networks," in *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice*, 2020, pp. 27–30.

[71] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow: Secure tensorflow inference," in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 336–353.

[72] A. Wood, K. Najarian, and D. Kahrobaei, "Homomorphic encryption for machine learning in medicine and bioinformatics," *ACM Computing Surveys (CSUR)*, vol. 53, no. 4, pp. 1–35, 2020.

[73] A. Al Badawi *et al.*, "Towards the alexnet moment for homomorphic encryption: Hcnn, the first homomorphic cnn on encrypted data with gpus," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 3, pp. 1330–1343, 2020.

[74] J.-W. Lee *et al.*, "Privacy-preserving machine learning with fully homomorphic encryption for deep neural network," *IEEE Access*, vol. 10, pp. 30 039–30 054, 2022.

[75] E. Lee *et al.*, "Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions," in *International Conference on Machine Learning*, PMLR, 2022, pp. 12 403–12 422.

[76] D. Kim, J. Park, J. Kim, S. Kim, and J. H. Ahn, "HyPHEN: A Hybrid Packing Method and Its Optimizations for Homomorphic Encryption-based Neural Networks," *IEEE Access*, 2023.

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

[78] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An Architecture for Computing on Encrypted Data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Lausanne Switzerland: ACM, Mar. 9, 2020, pp. 1295–1309, ISBN: 978-1-4503-7102-5.

[79] W. K. Hastings, "Monte Carlo Sampling Methods Using Markov Chains and Their Applications," *Biometrika*, vol. 57, no. 1, Apr. 1, 1970.

[80] C. Lattner *et al.*, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, IEEE, 2021, pp. 2–14.

# 국 문 초 록

## 동형암호를 위한 오차-성능 반영 스케일 관리 컴파일러


고정 소수점 산술 및 SIMD와 유사한 벡터화 기능 덕분에 RNS-CKKS는 암호화된 데이터에 대한 계산을 가능하게 하는 완전 동형 암호화(FHE) 기법 중에서도 프라이버시 보존 머신러닝 서비스를 위한 인기 있는 선택지로 자리매김했다. 이전 연구들은 RNS-CKKS의 고정 소수점 산술에 필수적인 스케일 관리 작업을 자동화하는 데 진전을 이루었지만, 제한된 성능향상과 정확도 향상을 보였다. 이러한 제한은 사용자가 오류 범위와 지연 시간 사이의 최적의 균형을 탐색하고 최적화할 수 있는 능력을 제한했다.

이 학위 논문은 프라이버시 보존 머신러닝 서비스를 강화하기 위해 특히 RNS-CKKS 기법을 중심으로 완전 동형 암호화(FHE) 분야를 진전시키는 세 가지 핵심 연구를 포함한다. 첫 번째 연구에서는 새로운 타입 시스템과 "다운스케일"이라는 새로운 리스케일링 작업을 활용하여 암호문 스케일을 최적화하는 혁신적인 FHE 컴파일러 프레임워크인 HECATE를 소개했다. HECATE는 다양한 스케일 관리 계획을 분석하여 기대되는 성능 영향을 평가하고, FHE 애플리케이션 전반에 걸쳐 최적의 리스케일링 지점을 찾아내어 이전 접근법보다 27%의 속도 향상을 달성했다.

두 번째 연구에서는 RNS-CKKS에 대한 오류 및 지연 시간을 고려한 스케일 관리인 ELASM 기법을 제안했다. 이는 이전 작업들이 출력 오류의 영향을 간과한 한계를 해결했다. 암호문 스케일을 적극적으로 관리함으로써 ELASM은 오류-지연 비용 함수를 최소화하고, 새로운 스케일-노이즈 비율(SNR) 매개변수를 도입하며, 오류-지연 트레이드오프를 향상시키기 위한 노이즈 인식 수위선을 소개했다. 이 접근법은 기존 솔루션들에 비해 머신러닝 및 딥러닝 벤치마크에서 우수한 성능을 보여주었다.

세 번째 연구에서는 수동 스케일 관리의 어려움과 기존 컴파일러의 비효율성을 극복하기 위해 RNS-CKKS 프로그램을 위한 성능 인식 정적 스케일 분석을 제안했다. 각 암호문의 스케일

"여유분"을 프로그램의 끝에서 역방향으로 분석하고 새로운 타입 시스템을 설계하여 스케일 예산을 재분배함으로써, 성능 인식 스케일 관리를 가능하게 했다.

이 연구들은 발전된 컴파일러 프레임워크, 스케일러 관리 기법, 성능 분석 기술을 통해 FHE 애플리케이션을 최적화하는 포괄적인 접근 방식을 제시한다. 이들은 효율적인 프라이버시 보존 머신러닝 서비스의 가능성을 입증할 뿐만 아니라 암호화된 계산 최적화를 위한 추가 연구의 새로운 길을 열어주었고, 보수적인 정적 스케일 분석 접근법보다 41.8%의 성능 향상을 달성하고, 탐색 기반 방법보다 크게 개선된 스케일 관리 시간을 보여주었다.

핵심되는 말: 완전동형암호, RNS-CKKS, 스케일 관리 기법, 컴파일러, 오차-성능 반영