CHLOE: Loop Transformation over Fully Homomorphic Encryption via Multi-Level Vectorization and Control-Path Reduction

Song Bian*, Zian Zhao*, Ruiyu Shen*, Zhou Zhang*, Ran Mao*, Dawei Li*, Yizhong Liu*,
Masaki Waga†, Kohei Suenaga†, Zhenyu Guan*, Jiafeng Hua‡, Yier Jin§, Jianwei Liu*✉
*School of Cyber Science and Technology, Beihang University,
Email: {sbian, zhaozian, 21377324, zhouzhang, maoran_44, lidawei, liuyizhong, guanzhenyu, liujianwei}@buaa.edu.cn
†Kyoto University, Email: {masakiwaga, ksuenaga}@gmail.com
‡Huawei Technology, Email: huajiafeng@huawei.com
§University of Science and Technology of China, Email: jinyier@gmail.com

***Abstract*—This work proposes a multi-level compiler framework to transform programs with loop structures to efficient algorithms over fully homomorphic encryption (FHE). We observe that, when loops operate over ciphertexts, it becomes extremely challenging to effectively interpret the control structures within the loop and construct operator cost models for the main body of the loop. Consequently, most existing compiler frameworks have inadequate support for programs involving non-trivial loops, undermining the expressiveness of programming over FHE. To achieve both efficient and general program execution over FHE, we propose CHLOE, a new compiler framework with multi-level control-flow analysis for the effective optimization of compound repetition control structures. We observe that loops over FHE can be classified into two categories depending on whether the loop condition is encrypted, namely, the transparent loops and the oblivious loops. For transparent loops, we can directly inspect the control structures and build operator cost models to apply FHE-specific loop segmentation and vectorization in a fine-grained manner. Meanwhile, for oblivious loops, we derive closed-form expressions and static analysis techniques to reduce the number of potential loop paths and conditional branches. In the experiment, we show that CHLOE can compile programs with complex loop structures into efficient executable codes over FHE, where the performance improvement ranges from $1.5\times$ to $54\times$ (up to $10^5\times$ for programs containing oblivious loops) when compared to programs produced by the-state-of-the-art FHE compilers.**

1. Introduction

Fully homomorphic encryption (FHE) is a class of cryptosystems that are capable to handling computing tasks over encrypted data without revealing the decryption key. Owing to the low communication complexity and high algorithm expressiveness, FHE presents to be useful in both composing compound secure multi-party computation (MPC) protocols [1–5], and establishing standalone privacy-preserving applications [2, 6–15].

In theory, FHE is generally applicable in the sense that any arbitrary program can be transformed to run over FHE primitives. However, due to the oblivious nature of data encryption, programs behave in a significantly different manner on ciphertexts than on plaintexts. For example, it is well known [8, 13, 16] that logic circuits are significantly more difficult to evaluate than arithmetic circuits over FHE ciphertexts, where the performance can easily differ by more than three orders of magnitude [13]. Furthermore, even for purely arithmetic-circuit programs, the performance is still heavily impacted by data encodings [2, 17], scheduling strategies [18, 19], and the encryption parameters [1, 3, 20].

```
#include<stdio.h>
int data_analysis(int data[512])
{
  int result = 0;
  for (uint16_t i = 0; i < 512; i++) {
    if (data[i] < 400)
        result += data[i];
  }
  return result;
}
```

Figure 1. An example of a manually-written data analysis program.

Motivated by the high design complexities of FHE algorithms, various types of FHE compilers have emerged to automatically translate programs on plaintext to those over FHE ciphertexts. In general, we see two distinct approaches towards FHE compiler designs: application-specific and general-purpose. For application-specific compilers, the design focus is on optimizing one particular set of FHE operators to perform well under a given circumstance. For example, a stream of FHE compilers [19, 21–27] focuses on how to optimize vectorized arithmetic over FHE ciphertexts, leveraging the single-instruction-multi-data (SIMD) capability of ring-based FHE schemes [28–30]. However, existing application-specific FHE compilers are mostly optimized for compiling arithmetic circuits only (i.e., circuits consisting of additions and multiplications), and are not good at handling non-arithmetic computations, e.g., comparisons, piece-wise functions, bit-wise operations, etc. To solve the usability issue, general-purpose FHE compilers [16, 31–33] are developed to translate (ideally) arbitrary plaintext programs to algorithms over FHE. Nonetheless, many existing general-purpose FHE compilers rely on logic synthesis tools to

transform the programs into Boolean-circuit representations, inducing significant performance overheads. To better balance the usability and efficiency, cross-scheme FHE compilers [16, 33, 34] are recently developed to make use of multiple FHE schemes in synthesizing a single program. For instance, by utilizing both the CKKS [30] and the TFHE [35] schemes, it is demonstrated [16, 33] that the generated programs can run significantly faster than the entirely circuit-based compiler [31].

Despite the considerable progress made in code transformation, most existing FHE compilers still remain less optimized for a crucial building block of structured programming: loop. A loop is one of the fundamental control structures, and it is well-known that programs often spend 90% of the time running 10% of the code in loops [36]. Additionally, loops are indispensable in implementing many non-algebraic expressions, such as transcendental functions, series, integrals, etc. As illustrated in Figure 1, a loop consists of two basic elements: a loop body and a loop condition. By definition, we see two fundamental challenges against effective optimizations of loops in FHE. First, the main body of the loop can contain arbitrary program segments, which makes it hard to optimize over FHE ciphertexts. For example, though some application-specific FHE compilers apply loop unrolling to vectorized SIMD arithmetic operations [24, 26], such compilers cannot successfully synthesize the program shown in Figure 1, due to the branching statement in the loop body. Second, in a homomorphic program (and secure multi-party algorithms in general [37]), we call the loop oblivious if the loop condition is private, and transparent otherwise. It is obvious that determining loop termination for oblivious loops can be extremely challenging (if not impossible). As a result, existing solutions either need to force a full loop unroll [31] or invoke periodic client-server interactions [15] to complete the loop, each of which incur expensive performance penalties. In short, a careful treatment of loops is essential in establishing a usable FHE compiler for the efficient synthesis of general-purpose programs.

1.1. Our Contributions

We propose CHLOE, a multi-level compiler framework that automatically synthesizes general-purpose programs into cross-scheme FHE algorithms. We observe that, similar to sequential programs, loops are also composed of arithmetic and non-arithmetic circuits. Hence, in CHLOE we first segment a complex loop into the respective arithmetic and non-arithmetic parts. Then, we apply fine-grained loop distribution [38] techniques to efficiently vectorize both arithmetic and non-arithmetic loops over SIMD-compatible FHE primitives such as CKKS [30] and BFV [28, 39]. Lastly, for oblivious loops, we derive closed-form expressions and propose new loop condition evaluation techniques to reduce the number of fully-unrolled control paths. The main contributions of this work are summarized as follows.

- **A Loop-Compatible FHE Compiler**: To the best of our knowledge, CHLOE is the first FHE compiler that can perform end-to-end transformations for programs containing non-trivial loops without human intervention. We observe that multiple levels of loop-specific optimizations involving both static and dynamic analyses are crucial in optimizing complex loops containing nested branching statements.
- **Mix-Circuit Vectorization**: We point out that properly vectorizing loops that contain both arithmetic and non-arithmetic circuit computations can be highly non-trivial. Specifically, the compiler needs to apply a series of segmentation, distribution, cost analysis and type alignment techniques to find the correct transformation pattern, such that we can fully leverage the SIMD capabilities of the FHE primitives.
- **Analyzing Oblivious Loops**: By carefully inspecting the control structures, we propose multiple optimization passes to boost the performance of oblivious loop evaluation. In particular, we derive close-form expressions to avoid fully unrolling oblivious loops, and propose new branching methods to optimize the performance of iteration selection when necessary.
- **Thorough Evaluation**: We rigorously study the performance and general applicability of CHLOE on various loop-containing program benchmarks. Specifically, we show that programs produced by CHLOE run $3.2\times$–$54\times$ faster on application programs containing transparent loops. Moreover, for oblivious loops, CHLOE can generate programs that run as much as $1.5\times$–$10^5\times$ faster than human-assisted existing solutions that require heavy full loop unrolling. An open-source implementation of CHLOE will be publicly available.

1.2. Related Works

Here, we first briefly summarize existing literature on MPC compilers in Section 1.2.1, and then give an overview on existing FHE compiler designs in Section 1.2.2. Our discussions pay particular attention to how loops are treated in the respective works.

1.2.1. MPC Compilers. Here, we focus on the class of MPC compilers whose primary goals are to enhance the usability and efficiency of MPC protocols [37, 46–57]. Similar to FHE compilers, we also classify existing MPC compilers into the application-specific ones and the general-purpose ones, and provide a brief summary as follows.

Application-Specific MPC Compilers: A line of MPC compilers target on how to implement specific privacy-preserving tasks efficiently [1, 3, 58, 59]. For instance, [58] proposes both basic MPC primitives and an end-to-end protocol compilation framework for privacy-preserving network analysis. Likewise, many application-specific MPC compilers, such as [1, 3, 59] optimizes privacy-preserving machine learning by automating the process of protocol selection and parameter instantiation. Overall, application-specific MPC compilers produce protocols that have near-optimized performance under a specific application context,

TABLE 1. SUMMARY OF RECENT (F)HE COMPILER DESIGNS

	Application-Specific FHE Compilers				General-Purpose FHE Compilers						
	[19, 40]	[23]	[24, 41]	[42, 43]	[22]	[34]	[31, 44]	[26, 27, 45]	[33]	[16]	**Ours**
FHE Scheme	CKKS	BFV	BFV/BGV*	BFV	BFV	BFV	TFHE	CKKS	CKKS+ TFHE	CKKS+ TFHE	BFV+TFHE +CKKS
Scheme Switch	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	◐**	●	●
Arith. SIMD	●	●	●	●	●	●	○	●	●	●	●
Cond. SIMD	○	○	○	○	◐†	○	○	◐$^\diamond$	◐$^\diamond$	○	●
Loop Type‡	N/A	N/A	T$\overline{\text{B}}$	T$\overline{\text{B}}$	T$\overline{\text{B}}$	N/A	TOB	TB$^\diamond$	T$\overline{\text{B}}$	TB	TOB
Loop Vec.	○	○	○	●	●	○	●	●	●	●	●
Loop Seg.	○	○	○	○	○	○	○	○	○	○	●
Loop Dist.	○	○	○	○	○	○	○	○	○	○	●

In the table, Arith. SIMD and Cond. SIMD refers to the capabilities of leverage arithmetic SIMD and conditional SIMD operators over RLWE ciphertexts. Loop vec., loop seg. and loop dist. denote the use of loop vectorization, loop segmentation and loop distribution, respectively.

* The backend for Marble [41] and HECO [24] can be either BFV or BGV, but not both.

** Scheme switching is a planned feature for Google HEIR [33].

† In theory, Ramparts [22] supports SIMD encoding for conditional statements by encoding Boolean values into the coefficients of a polynomial. However, such encoding requires assistances from the client and may not be generally applicable.

‡ T$\overline{\text{B}}$ depicts the support of transparent loop without conditional statements, and TB refers loops with nested conditional statements. TOB means that the compiler supports both transparent and oblivious loops with conditional statements.

$^\diamond$ As described in [26], the programmer can construct an approximate polynomial for processing conditional statement (with or without SIMD). However, such approximation needs to be done manually before the compilation process [26, 27, 45]

but the optimizations often do not apply to other protocol use cases.

General-Purpose MPC Compilers: To strengthen the general applicability of MPC, a plethora of general-purpose compilers are developed to compile arbitrary programs into MPC protocols [46–50, 52–54]. Given the large number of distinct MPC primitives, producing an optimal protocol for a given privacy-preserving task can be extremely hard [60, 61]. Thus, some MPC compilers propose to adopt domain-specific languages to reduce the design complexity and enhance security [51, 62–64]. Nonetheless, the learning costs of multiple domain-specific languages can be non-negligible, which motivated the development of MPC compilers that use native C or Java as the surface languages [49, 50, 52–54, 65, 66]. Even though general-purpose MPC compilers can significantly alleviate the usability issue, the frequently evolving nature of MPC primitives may quickly render such compilers obsolete. Moreover, while some very recent works [67, 68] start to explore how to leverage vectorized MPC primitives to accelerate protocol execution, most existing MPC compilers still lack sufficient optimization against non-trivial loops [37].

1.2.2. FHE Compilers. As illustrated in Table 1, a stream of works are proposed to design and implement compilers tailored for FHE [15, 16, 19, 21–25, 27, 31, 32, 34, 40, 41, 43, 44, 69, 70]. Similar to Section 1.2.1, we also group FHE compilers depending on whether they are application-specific or general-purpose, and compare the main compilation features as follows.

Application-Specific FHE Compilers: Most compilers of this class adopt the BFV [28] and the CKKS [30] types of FHE schemes, where arithmetic operators such as ciphertext additions and multiplications can act over batched ciphertexts. Application-specific FHE compilers [19, 21–24, 40, 43] can produce highly efficient programs for polynomial computations, such as linear transformations, polynomial-approximated functions, etc. However, as shown in Table 1, most application-specific FHE compilers do not support conditional statements well. In addition, albeit some recent work [43] studies how to efficiently vectorize loops, the design exploration is limited to simple loops that are composed of arithmetic operations only (i.e., vectorized homomorphic additions, multiplications, and rotations).

General-Purpose FHE Compilers: Based on the recent advances in FHE bootstrapping [35, 71–76], general-purpose FHE compilers become a viable solution for general program compilation. Within general-purpose FHE compilers, we identify three distinct approaches: polynomial-based, circuit-based, and hybrid. First, polynomial-based FHE compilers [26, 27, 45] convert a program into a sequence of polynomial evaluations, where each polynomial approximate (or evaluate exactly) one particular block of codes, where the approximated functionalities range from linear transformations, `IF-ELSE` statements to `FOR` loops. Nevertheless, it can be highly non-trivial to automate the process of polynomial approximation, and most polynomial-based FHE compilers rely on manual analysis for code block segmentation. Second, circuit-based FHE compilers [15, 31, 32, 44] employ a set of functionally complete Boolean gates to express general programs by circuits, i.e., sequences of Boolean gates. By leveraging electronic design automation tools [31, 32, 44], it is much easier to implement a general-purpose FHE compiler based on circuits than other approaches. Unfortunately, since FHE operators are not inherently Boolean, representing general programs as logic circuits result in degraded performance, especially for heavily-algebraic tasks (e.g., matrix-vector multiplication). Finally, hybrid compilers [16, 33, 69, 70] strive to achieve the best of both worlds. By incorporating both polynomial- and circuit-based representations, hybrid compilers first segment a given program into the arithmetic and logic parts, and then apply FHE-specific optimizations to

the respective code segments. Unfortunately, similar to other types of compilers [24, 43], most hybrid FHE compilers can only handle elementary loops, weakening the general applicability of such compiler frameworks.

2. Preliminaries and Background

In this work, we make use of three types of lattice-based FHE schemes: the BFV/BGV type [28, 29, 77], the CKKS [30] type, and the FHEW/TFHE type [35, 78–81]. In what follows, we describe the basic constructions and key operators for each of the FHE schemes in Section 2.1 and Section 2.2, respectively.

For notations, we use λ to denote the security parameter, p the plaintext modulus, q/Q the ciphertext moduli, where it generally holds that $Q > q$. $\mathbb{Z}_q$ denote the set of integers modulo q, and $\mathbb{N}$ is the set of natural numbers. For some lattice dimension N, we write $R_{N,Q} = \mathbb{Z}_Q[X]/(X^N + 1)$. We use tilde lower-case letters with square brackets such as $\tilde{a}[X]$ to refer to polynomials in the variable X, and bold lower-case letters such as $\mathbf{a}$ to depict vectors. $\mathbf{a}[i]$ notes the i-th element in the vector $\mathbf{a}$.

2.1. FHE Constructions

In this work, we use three types of ciphertexts: LWE, RLWE, RGSW, which are all constructed over the standard lattice hardness assumptions, i.e., the learning with errors (LWE) and ring learning with errors (RLWE) problems. Details for each type of ciphertext is explained as follows.

• **LWE Ciphertext**: First, for a plaintext message $m \in \mathbb{Z}$ and the secret key $\mathbf{s} \in \mathbb{Z}_q^n$, an LWE ciphertext is given by

$$\mathsf{LWE}_{\mathbf{s}}^{n,q}(m) = (b, \mathbf{a}) = (-\mathbf{a}^{\mathrm{T}}\mathbf{s} + \Delta m + e, \mathbf{a}), \quad (1)$$

where $\mathbf{a} \in \mathbb{Z}_q^n$ uniformly random, e is chosen from some distribution χ_{noise} with standard deviation $noise$, and Δ is a scaling factor.

• **RLWE Ciphertext**: Given an encoded plaintext polynomial $\tilde{m} \in R_{N,Q}$ and a secret key polynomial $\tilde{s} \in R_{N,Q}$, an RLWE ciphertext is defined as

$$\mathsf{RLWE}_{\tilde{s}}^{N,Q}(\tilde{m}) = (\tilde{b}, \tilde{a}) = (-\tilde{a} \cdot \tilde{s} + \tilde{m} + \tilde{e}, \tilde{a}). \quad (2)$$

Here, $\tilde{a} \in R_{N,Q}$ is a random polynomial, and $\tilde{e}$ is sampled from χ_{noise}. As discussed later, we merge the scaling factor for the RLWE ciphertext into the encoding process.

• **RGSW Ciphertext**: For a decomposition parameter l, and a message $m \in \mathbb{Z}_p$, an RGSW ciphertext under the secret key $\mathbf{s} \in \mathbb{Z}_q^n$ is given by $\mathsf{RGSW}_{\tilde{s}}^{N',Q'}(m) = (\mathbf{B}, \mathbf{A}) \in \mathbb{Z}_{Q'}^{2l\times 2}$. The concrete constructions of RGSW can be found in [79, 82]. Here, we can simply consider an RGSW ciphertext as a tuple of two $2l$-degree RLWE ciphertexts.

Plaintext Encodings for RLWE Ciphertexts: As mentioned, to be encrypted under RLWE, a message needs to be encoded into a plaintext polynomial $\tilde{m}$ in advance [83, 84]. To encode a vector of messages $\mathbf{m} \in \mathbb{Z}_p^N$ into a plaintext polynomial $\tilde{m} \in R_{N,Q}$, we have two main parameters: the embedding function σ and the scaling factor Δ. Based on the two parameters, the encoding process can be formulated as

$$\tilde{m} = \Delta\sigma(\mathbf{m}), \quad (3)$$

where $\Delta \in \mathbb{R}$ and σ can be the slot embedding [85] or the coefficient embedding [2, 7]. For example, for (a slightly modified version of) the CKKS scheme implemented in the SEAL [86] library, σ is instantiated to be the inverse number theoretic transform (INTT), and Δ is an arbitrary real number. In contrast, for BFV also realized in SEAL, σ is the identity function and $\Delta = \lfloor q/p \rceil$ [85].

2.2. FHE Operators

Here, we briefly overview the key FHE operators organized according to the ciphertext types of the associated operands.

2.2.1. Operators for LWE Ciphertexts. As described in Eq. (1), one LWE ciphertext can only encrypt a single integer message $m \in \mathbb{Z}$. Therefore, we have the following homomorphic operators permitted over LWE ciphertexts.

• $+$, $-$ and $\cdot$: We note that LWE ciphertexts have basic additive homomorphism. Hence, LWE ciphertexts are compatible with standard ciphertext addition, subtraction and constant multiplication.

• $\textsc{CMux}(\mathsf{RGSW}(\hat{s}), \mathsf{LWE}(m_0), \mathsf{LWE}(m_1))$: Given inputs $\mathsf{LWE}(m_0)$ and $\mathsf{LWE}(m_1)$, for a ciphertext control input $\mathsf{RGSW}(\hat{s})$ where $\hat{s} \in \{0, 1\}$, CMux homomorphically computes $\hat{s}\ ?\ \mathsf{LWE}(m_0)\ :\ \mathsf{LWE}(m_1)$, i.e., the function outputs $\mathsf{LWE}(m_0)$ if $\hat{s} = 1$ and $\mathsf{LWE}(m_1)$ if $\hat{s} = 0$ (without revealing $\hat{s}$).

• $\textsc{PBS}(\mathsf{LWE}(m), T(x))$: Given an LWE ciphertext $ct = \mathsf{LWE}_{\mathbf{s}}^{n,q}(m)$, a discrete function $T(x)$, PBS outputs $\mathsf{LWE}_{\mathbf{s}}^{n,q}(T(m))$ with a constant (i.e., input-independent) level of noise.

2.2.2. Operators for RLWE Ciphertexts. Different from LWE ciphertexts, an RLWE ciphertext can encrypt a polynomial encoding a vector of plaintext messages into a single ciphertext. Here, we summarize the list of key operators over RLWE ciphertexts.

• $+$, $-$ and $\cdot$: Similar to LWE, RLWE supports the execution of ciphertext addition, subtraction and constant multiplication operators. Furthermore, we can also apply ciphertext-ciphertext multiplication over RLWE ciphertexts, i.e., $\mathsf{RLWE}(m_0) \cdot \mathsf{RLWE}(m_1) = \mathsf{RLWE}(m_0 \cdot m_1)$.

• $\textsc{HomPoly}_{\tilde{p}}(\mathsf{RLWE}(\tilde{m}))$: For any polynomial $\tilde{p}[x]$, $\textsc{HomPoly}_{\tilde{p}}(\mathsf{RLWE}(\tilde{m}))$ represents the homomorphic evaluation of $\tilde{p}[x]$ over the input ciphertext, i.e., $\textsc{HomPoly}_{\tilde{p}}(\mathsf{RLWE}(\tilde{m})) = \mathsf{RLWE}(\tilde{p}[\tilde{m}])$.

• $\textsc{Rotation}(\mathsf{RLWE}(\tilde{m}), \hat{d})$: For a given ciphertext $\mathsf{RLWE}(\tilde{m})$, $\textsc{Rotation}(\mathsf{RLWE}(\tilde{m}, \hat{d})$ outputs a new ciphertext $\mathsf{RLWE}_{\mathrm{out}}(\tilde{m}[X^{\hat{d}}])$, i.e., the coefficient of the plaintext polynomial is rearranged by the parameter $\hat{d}$.

• $\textsc{ExtProduct}(\mathsf{RGSW}(\tilde{m}_0), \mathsf{RLWE}(\tilde{m}_1))$: Given two encoded plaintext $\tilde{m}_0$ and $\tilde{m}_1$, ExtProduct is another

way of carrying out homomorphic multiplication where $\text{EXTPRODUCT}(\mathsf{RGSW}(\tilde{m}_0), \mathsf{RLWE}(\tilde{m}_1)) = \mathsf{RLWE}(\tilde{m}_0 \cdot \tilde{m}_1)$. In general, EXTPRODUCT runs faster and generates significantly less noise compared to a straightforward homomorphic multiplication.

• SIMD-PBS($\mathsf{RLWE}(\tilde{m}), T(x)$): Recently proposed in [39], SIMD-PBS is basically a SIMD-version of the PBS operator, where the function T act over all elements of $\tilde{m}$ in a coefficient-wise manner, i.e., $\mathsf{RLWE}(T(\tilde{m})) = \text{SIMD-PBS}(\mathsf{RLWE}(\tilde{m}), T(x))$.

2.2.3. Conversion Operators. To support the evaluation of both logic and arithmetic functions, ciphertext format conversion operators are proposed to achieve the best of both worlds. Here, we mainly consider four conversion operators: EXTRACTLWE and PACKLWE that convert the ciphertext between the LWE format and the RLWE format [8, 35, 87, 88], as well as STOC and CTOS that convert the underlying plaintext format in an RLWE ciphertext between the slot encoding and the coefficient encoding [89, 90].

• EXTRACTLWE: EXTRACTLWE can extract an element of the encoded vector in an RLWE ciphertext ct ($\mathsf{ct} \in \mathsf{RLWE}^{N,Q}_{\tilde{s}}(\tilde{m})$) into an LWE ciphertext of the form

$$\text{EXTRACTLWE}(\mathsf{ct}, i) \rightarrow \mathsf{LWE}^{N,Q}_{\mathbf{s}}(\tilde{m}_i), i \in \{1, \cdots, N\}.$$

Here, the equation extracts the i-th plaintext message $\tilde{m}_i$ encrypted in the RLWE ciphertext ct.

• PACKLWE: PACKLWE can pack a set of LWE ciphertexts $\{\mathsf{ct}_i\}_{i \in \langle N \rangle}$, $\mathsf{ct}_i \in \mathsf{LWE}^{N,Q}_{\mathbf{s}}(\mathbf{m}[i])$ into a single RLWE ciphertext:

$$\text{PACKLWE}(\{\mathsf{ct}_i\}_{i \in \langle N \rangle}) \rightarrow \mathsf{RLWE}^{N,Q}_{\tilde{s}}(\tilde{m}),$$

where the set of LWE ciphertexts encrypt each element of the plaintext vector $\mathbf{m}$, and the output RLWE ciphertext encrypts $\tilde{m}$ such that $\tilde{m}_i = \mathbf{m}[i]$.

• STOC and CTOS: Let σ_s be the slot encoding and σ_c the coefficient encoding, we have that

$$\mathsf{RLWE}(\Delta\sigma_c(\mathbf{m})) = \text{STOC}\big(\mathsf{RLWE}(\Delta\sigma_s(\mathbf{m}))\big), \text{ and}$$
$$\mathsf{RLWE}(\Delta\sigma_s(\mathbf{m})) = \text{CTOS}\big(\mathsf{RLWE}(\Delta\sigma_c(\mathbf{m}))\big).$$

3. The CHLOE Compiler Framework

In this section, we first describe the challenges for compiling loops over FHE in Section 3.1. Then, we sketch the high-level overview of CHLOE and the set of proposed multi-level intermediate representations (IRs) in Section 3.2 and Section 3.3, respectively.

3.1. Problem Formulation and Key Observations

As mentioned, to compile general programs over FHE, the transformation and optimization of loops can be both crucial and challenging. On one hand, instructions outside loops in application programs tend to be one-time operations that are lightweight and deterministic. On the other hand, the evaluation of loops often involves repeated executions of complex computations with irregular branching structures. Therefore, the main objective of this work can be formulated as the following question. How can we architect a compiler infrastructure such that programs with non-trivial loop structures can be transformed into efficient algorithms over FHE? Here, we list the main obstacles against the design of such compiler and provide our key observations.

Challenge and Observation 1: The first challenge against loop transformation over FHE rises when the main loop body contains both conditional statements and arithmetic operations. As illustrated in Table 1, most existing FHE compilers, such as HECO [24] and Viaduct-HE [43], only support the unrolling and vectorization for loops containing only arithmetic operators, i.e., additions and multiplications. While some recent work [16] can compile loops containing arithmetic operations mixed with conditional statements, the transformed code segments cannot be optimized using the SIMD FHE operators, resulting in degraded performance of the generated programs. To tackle with such challenge, our first key observation is that the main loop body can be segmented into different code blocks (similar to the idea of loop distribution), such that the conditional and arithmetic operations can be separately unrolled and vectorized.

Challenge and Observation 2: The second difficulty is how to compile oblivious loops, i.e., loops with private loop conditions. As explained in Section 1.2, handling oblivious loops is hard for both FHE and general MPC compilers, where the compiler may not always be able to decide the exact exit point of the loop. Some existing work [15] rely on client-server interactions to periodically check the status of the loop condition, which can be heavy in both computation and communication. Therefore, our second observation is that many real-world loops are well-structured and can be optimized through static and dynamic loop analysis techniques.

3.2. CHLOE Workflow

Similar to recent FHE compiler constructions [16, 24, 26, 33], CHLOE is also developed on top of the MLIR framework [91]. To cope with the advanced looping structures and complex FHE schemes, however, we need to reformulate the designs of the MLIR layers to enable the proper mapping of FHE operators. The overall dialect architecture and compilation pipeline of CHLOE is sketched in Figure 2, and we provide a high-level explanation for each of the compilation stages as follows.

① ***Front-End Passes***: CHLOE accepts C-like plaintext code as surface language and uses Polygeist [92] as the front-end module to transform the high-level language into static single assignment (SSA) style IRs. Since Polygeist produces a program with high-level plaintext MLIR dialects, we need to map the operators and data types of the program produced by Polygeist to the unified fhe dialect. Within the fhe layer, the program is further segmented into arithmetic and non-arithmetic regions based on the method in [16]. Note that, at this stage, loops are treated as a black-box region and remain unchanged.

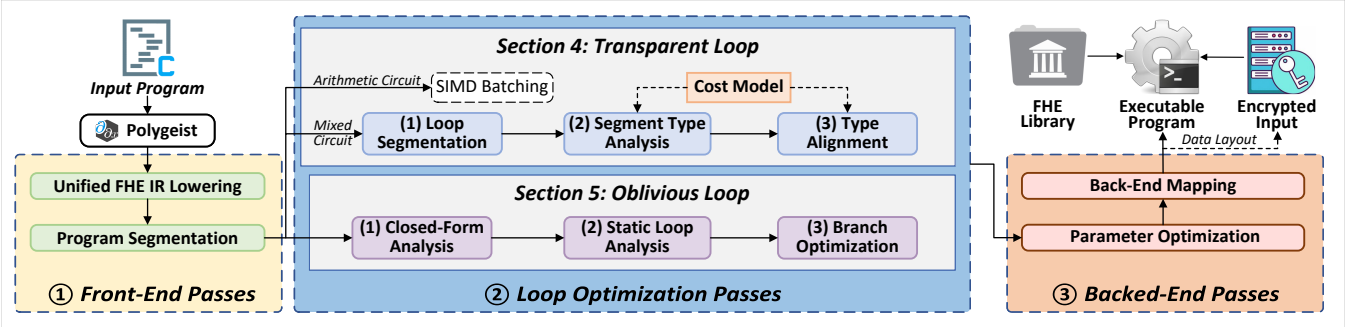


Figure 2. An overview on the compilation flow of CHLOE.

② ***Loop Optimization Passes***: In this stage, CHLOE establishes a compilation pipeline tailored on dealing with non-trivial loop structures. Here, the lowering passes are split into two routines: transparent loop lowering (Section 4) and oblivious loop lowering (Section 5). Transparent loops can be further classified based on the operations within the loop. Loops composed only of additions, subtractions, and multiplications (i.e., arithmetic operations) are defined as arithmetic-circuit loops. In contrast, loops that contain both arithmetic and non-arithmetic operations (such as conditional statements) are defined as mixed-circuit loops. Since transformations for arithmetic-circuit loops are relatively well studied [24, 43], we integrate the proposed optimization passes into CHLOE. Meanwhile, for the transformations of mixed-circuit loops, the core passes include loop segmentation, segment type analysis and cost-aware type alignment, which are further elaborated in Section 4. Lastly, for oblivious loops, CHLOE inspects the structures of the loops to apply branch-related optimizations and closed-form analysis, such that the high costs of full loop unrolling can be mitigated or avoided.

③ ***Back-End Passes and Code Generation***: After the loop-specific transformations, the program is ready to be lowered to an output binary with FHE-compatible data and control flows. In the back-end passes, higher level IRs are mapped into the lwe, rlwe dialects, where the exact encryption parameters for the respective ciphertexts can be properly determined. Finally, the program will be lowered to specific FHE operators implemented by a back-end FHE library via the MLIR EmitC toolchain.

3.3. Dialects in CHLOE

Here, we discuss the IR structures for CHLOE in detail. As shown in Figure 3, IRs in CHLOE form four main dialect layers: plaintext layer, unified encrypted layer, operational ciphertext layer and fundamental ciphertext layer. In each of the different dialect layers, we define layer-specific type aliases and operators, and the concrete constructions are explained below.

Plaintext Layer: In the front-end of CHLOE, an input program written in C is transformed into standard SSA-style dialect IRs, which we define as the plaintext representation

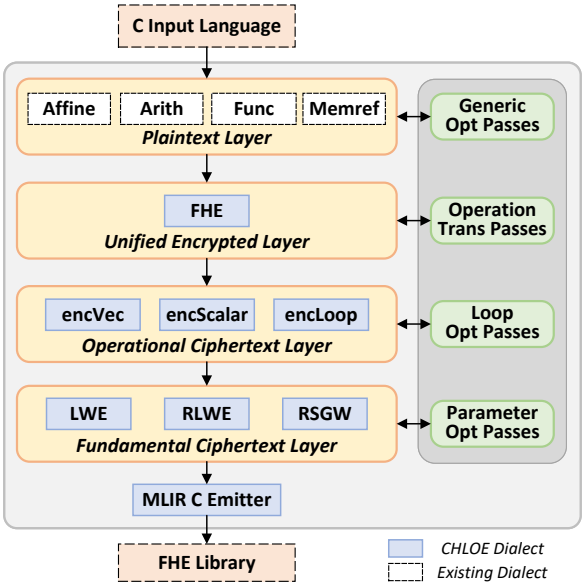


Figure 3. The dialect structure of CHLOE.

layer. In this layer, dialects such as Arith and Memref are adopted to express computation and data storage operations, while Affine and Func are used to represent the control flows and function structures of the input program. Optimization passes for generic programs, such as common sub-expression elimination and dead code elimination, are deployed based on the existing front-end compilation infrastructure. However, since CHLOE does not make use of domain-specific languages, there is no clear distinction between plaintext computation and encrypted data processing in the plaintext layer.

Unified Encrypted Layer: In this layer, the input program represented as IRs in the plaintext layer is transformed into the fhe IR. The primary goal of establishing a unified fhe dialect is to differentiate plaintext computations from ciphertext operations as much as possible, such that the amount of computation over FHE can be minimized. Thus, we define two basic data types in the fhe: `plain` and `cipher`. In CHLOE, all of the input variables and their subsequently derived variables are marked as `cipher`, while

other variables are considered as type `plain`. Based on such definition, we can optimize away complex computations over data marked as `plain` types, as these computations can be taken care of by standard (non-encrypted) compilation pipelines.

Operational Ciphertext Layer: We point out that a unified fhe dialect is inadequate for fine-grained optimizations, especially for non-scalar variables. For instance, an encrypted vector can be lowered to either an array of RLWE ciphertexts, a single RLWE ciphertext or an array of LWE ciphertexts, depending on the input vector length and subsequent computations. Therefore, we further granulate the fhe dialect into type-specific dialects such as encVec and encLoop. In encLoop dialect, we define operations suited for loop structures. For example, a new IF operation is defined to simplify optimizations for loop segmentation, dependency analysis, and further processing for loop conditions. In encVec dialect, we define data types including `EncVector` and `PlainVector`. We propose multiple optimization passes based on computation cost analysis to vectorized variables and configure the exact FHE attributes such as encoding method, encoding parameters, and data layouts assigned to the `EncVector` and `PlainVector` variables.

Fundamental Ciphertext Layer: After going through the optimizations made in the higher layers, we define low-level dialects that have strict one-to-one mappings with the application-program interfaces (APIs) of the underlying FHE implementation library. Since CHLOE uses three types of ciphertexts, we construct the LWE, RLWE, and RGSW dialects to represent different ciphertext formats and the associated computation and conversion operators introduced in Section 2 for final executable generation.

3.4. Threat Model

The threat model for CHLOE is similar to that of most existing FHE compilers [16, 24, 33, 45], where we have provable security under a semi-honest secure two-party computation model. In such scenario, both the client and the server are semi-honest adversaries against each other, where both parties do not actively manipulate the computation or compilation processes but only passively observes the ciphertexts (or plaintext results) in an attempt to gain knowledge of data that are private to the other party. Note that, we assume that all input data to the compiled program are owned by the client. Hence, the compilation process can be executed either by the client or by the server, since a semi-honest adversary will honestly execute the compilation procedures to produce the output programs over FHE.

4. Transparent Loop Transformation

As mentioned earlier, transforming transparent loops into FHE-friendly forms is a key compilation pipeline in CHLOE. Throughout this section, we use the code block shown in Figure 1 as an example input program, which a data analysis application commonly used in querying databases. The program starts by first identifying if the value of the data is less than 400 using an IF-ELSE statement. If the condition holds, the `data[i]` value loaded in that iteration is accumulated into the `result` variable. Therefore, the example program in Figure 1 demonstrates a typical loop that contains both arithmetic and non-arithmetic (IF statement) operations within its loop body, which most existing compilers cannot efficiently process.

4.1. Transformation Overview

As shown in Figure 2, the transformation of a mixed-circuit transparent loop contains three sub-procedures: (1) loop segmentation, (2) segment type analysis, and (3) type alignment. Here, we use the example program in Figure 1 to give an overview on the above three sub-procedures, where further details can be found in the respective sections.

Before processing the loop, note that we have already transformed the input program from the C-level description into the fhe dialect, as shown in Figure 4(a) for the example program in Figure 1. Here, due to the encryption of the condition variable, the IF statements over FHE are transformed into a sequence of instructions to execute all possible branches followed by a multiplexing of the conditioned result. After the IF transformation, operations inside the main loop body becomes a purely sequential structure, and we are ready to further process the loop.

(1) Loop Segmentation (Section 4.2): Here, we first analyze loop-carried dependencies in regions of non-arithmetic operations. Then, we identify regions in the main loop body that can be segmented, where each loop segment contains either arithmetic or non-arithmetic operations, but not both. For example, for the loop in Figure 4(a), Line 5–6 is a segment of non-arithmetic operations that can be completely separate from Line 7–12, which are two condition branches composed of purely arithmetic circuits. Therefore, by loop segmentation, we can distribute a complex loop into multiple simple loops, each containing only a single type of operation.

(2) Segment Type Analysis (Section 4.3): After segmentation, we analyze the operator type for each of the individual segments. Meanwhile, we establish cost models for each of the FHE operators in terms of their computation complexities to serve as a foundation for later optimization passes.

(3) Cost-Aware Type Alignment (Section 4.4): In this step, we perform type alignment, where concrete data types and encodings of the input and output ciphertexts are instantiated for each of the segmented loop pieces. The rewriting rules for data type alignment are supported by the cost model based on the dependency analysis in between different segments. Lastly, the respective scheme-switching and encoding-switching operators are inserted to bridge different ciphertext formats inside and outside the loop.

4.2. Loop Segmentation

To leverage the efficient SIMD packing capabilities of both the arithmetic and non-arithmetic FHE operators, we

```
func.func @data_analysis(%arg0: cipherVec<512>) ->
    cipher {
  %c400 = fhe.constant 400 : plain
  %c0 = fhe.constant 0 : plain
  %0 = affine.for %arg1 = 0 to 512 iter_args(%arg2 =
     %c0) -> (cipher) {
  %1 = fhe.load %arg0[%arg1] : cipherVec<512>
  %2 = fhe.cmp lt, %1, %c400 : cipher
  %3 = affine.if %2 -> (cipher) {
    %4 = fhe.add %arg2, %1 : cipher
    affine.yield %4 : cipher
  } else { affine.yield %arg2 : cipher }
  affine.yield %3 : cipher
  }
  return %0 : cipher
}
```

(a) The intermediate representation version of data analysis program.

```
LWECipher data_analysis(RLWECipher data)
{
  RLWECipher diff = RLWE_Sub(data, 400);
  // Batched Comparison
  vector<LWECipher> diffVec = SToCThenExtract(diff);
  RLWECipher index = SIMD_PBS(diffVec, LUT_SIGN);
  RLWECipher tmp = RLWE_MULT(data, index);
  RLWECipher rotTmp;
  // Result Accumulation
  for (int i = 1; i <= log2(512), i++) {
    rotTmp = RLWE_Rotate(tmp, 512/pow(2,i));
    tmp = RLWE_Add(tmp, rotTmp);
  }
  LWECipher result = Sample_Extract(tmp, 0);
  return result;
}
```

(b) The data analysis program after loop transformation.

Figure 4. Two programs calculating the sum of all data less than 400 out of a vector. (a) the input program of CHLOE represented in SSA IR format, which performs a comparison individually in each iteration and calculates aggregation if the condition is satisfied. (b) the output program of CHLOE, which transforms the comparisons into a SIMD functional bootstrapping and a batched multiplication to improve the efficiency of the program.

$$\text{SEGMENT}\frac{u \in \{\mathit{Inputs}, \mathit{Variables}\} \quad \text{OP\_TYPE} = \{\text{ARITH}, \text{NonARITH}\} \quad \{\mathit{useOp}_i\}_{i\in N} \leftarrow u.\textbf{\textit{getUseOp}} \quad \textbf{Type}(\mathit{useOp}_i) \,!= \textbf{Type}(u.op)}{\text{Initialize } \mathit{segOp} \leftarrow \textsc{SegmentOp} \quad \mathit{segOp}.\textbf{\textit{setSegID}}(\mathit{seg\_id}) \quad u.\textbf{\textit{replaceUse(i)}} \leftarrow \mathit{segOp} \quad \mathit{segOp}.\textbf{\textit{setUse}} \leftarrow \mathit{useOp}_i}$$

$$\text{GROUP}\frac{u \in \{Inputs, Variables\} \quad seg\_id \leftarrow \textbf{\textit{getSegID}}(u.ParentOp)}{u.op.\textbf{\textit{setSegID}} \leftarrow seg\_id}$$

Figure 5. The graph rewriting rules for operator grouping in CHLOE.

need to segment the main body of the loop into mutually-independent regions of purely arithmetic or non-arithmetic circuits, and then aggregate the results from each of the loop segments. We refer to this process as the loop segmentation stage, and the concrete steps are explained as follows.

(i) Branch Flattening: Since selective execution is not compatible with FHE computation, branch structures need to be flattened into sequential operators over ciphertexts. Hence, we traces the *Def-Use* chain of all variables appear in branches. If a variable is referenced outside the branch, we insert a multiplexing operation before the reference point to keep or clear the value stored in the variable depending on the encrypted IF condition.

(ii) Loop-Carried Dependency Analysis: The aim of loop segmentation is to separate loops of different circuit types, such that the SIMD capability of the low-level RLWE ciphertext can be effectively utilized. However, we observe that non-arithmetic operations with loop-carried dependencies can be difficult to batch. In contrast, arithmetic operations with loop-carried dependencies, such as array aggregation and inner product, are subject to SIMD optimization. Consequently, if any loop-carried variables are detected to act as operands in non-arithmetic operations, we stop trying to further segment and batch the loop. Otherwise, the loop is marked as SIMD-compatible for subsequent optimization passes.

(iii) Operator Grouping: In this step, we split the code block inside the main loop body into different segments. Based on the type of operations within the loop segment, we construct arithmetic and non-arithmetic loop segments. In other words, an arithmetic loop segment refers to the case where all operations contained in the segment are arithmetic operations (i.e., only additions, subtractions, multiplications and rotations). In Figure 5, we show the rewriting rules that group operations into different loop segments with *Def-Use* dependency analysis. Roughly speaking, when a variable is referenced multiple times by operators that have incompatible types with each other, we insert a segmentation operator at each of the operator-type-switching boundaries. We denote this class of variables as boundary variables.

Algorithm 1: Loop Splitting

Input : DAG of the program segment within a loop $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, total number of segments K
Output : Transformed DAG of the program $\mathcal{G} = (\mathcal{V}, \mathcal{E})$

▷ Code Segments Generation
for $u \in \mathcal{V}$ **do**
  $op = u.getOp$
  $Seg_{op.seg\_id}.append(op)$
end
▷ Boundary Variable Vectorization
for $Seg_k \in \cup\{Seg_i\}$ **do**
  Inputs, Outputs $\leftarrow$ ***IdentifyLoopIO***(Seg_k)
  for $u \in$ {Inputs, Outouts} **do**
    if u is a *Iterative Variable* **then**
      $u' \leftarrow$ ***AggregateIterations***(u)
      VecInputs.$append(u')$ // or VecOutputs
    else
      VecInputs.$append(u)$ // or VecOutputs
    end
  end
  $Loop_k \leftarrow$ ***GenLoop***$(Seg_k$, VecInputs, VecOutputs$)$
end
Return : $\{Loop_k\}_{k\in K}$

(iv) Loop Splitting: In (iii), we identify the segment boundaries and insert segmentation operators. Next, we distribute the segments of loop into different independent loops. The primary tasks here are dependency analysis and variable vectorization. In Algorithm 1, we show the concrete procedures for loop splitting. First, on Line 1-7, we reconstruct the loop according to the execution order of the instructions, ensuring that all variables needed in downstream loop segments are produced in the preceding segments. Second, on Line 8–18, boundary variables need to be transformed into vectorized variables since their definitions and usages are separated into different loop segments. We provide a concrete example to illustrate the transformation process in Section A.1.

4.3. Segment Type Analysis

After loop segmentation, long and complex loops are broken down into simple loops that are either arithmetic or non-arithmetic. Next, we proceed the optimization process by applying segment type analysis, a step that analyze each of the loop segments and establish their cost models. In particular, the most important optimization is if SIMD is to be applied to the loop segments, since vectorized evaluation does not always improve computational efficiency, especially the number of batched elements are small.

4.3.1. Vectorization Analysis. Here, we identify the vectorization techniques that can be applied to both arithmetic loop segments and non-arithmetic loop segments.

Non-Arithmetic Loop Segments: In this work, we utilize the PBS and SIMD-PBS to evaluate all non-arithmetic functions. Since PBS and SIMD-PBS virtually permit any non-arithmetic functions to be applied to a ciphertext, we can group consecutive non-arithmetic functions into a single PBS invocation function composition to minimize the number of bootstrapping operations, e.g.,

$$\text{PBS\_MERGE}\frac{\text{PBS}(\text{PBS}(f,\mathbf{u}),g),\mathbf{u}.Type \in \texttt{EncScalar}}{\text{PBS}(\mathbf{u}, f\circ g)}$$

Hence, a non-arithmetic loop segment has two transformation options. First, since there is no dependency within the loop, we can vectorize the entire loop segment with SIMD-PBS operations. Second, when the overall number of iterations is small, we can also completely unroll the loop and evaluate the statements within the loop iteration-by-iteration.

Arithmetic Loop Segment: Since only arithmetic operations are permitted in here, we can fully use the SIMD batching technique proposed in existing literature [16, 19, 24]. Therefore, we have three different options when transforming the loop. First, we can transform the loop iterations to SIMD-style computations on RLWE slot-encoding ciphertexts. Second, the loop can be converted into vectorized computations on coefficient-encoding ciphertexts. Third, as mentioned above, if the number of iterations is limited, we can fully unroll the loop directly.

Algorithm 2: Cost-Based Program Transformation

Input : DAG of the input program $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of code segments, and $\mathcal{E}$ is a set of dependencies between code segments.
Output : Transformed program DAG with minimum total computation cost.

1 $\mathcal{V}_{loop} \leftarrow$ loop segments in $\mathcal{V}$
2 **for** each vertex $v \in \mathcal{V}_{loop}$ **do**
3 **if** $\mathbf{Type}(v) ==$ Arithmetic **then**
4 $\mathbf{Transform}(v) \leftarrow$ {Unroll, Slot_SIMD, Coeff_SIMD}
5 **else if** $\mathbf{Type}(v) ==$ Non − Arithmetic **then**
6 $\mathbf{Transform}(v) \leftarrow$ {Unroll, PBS_SIMD}
7 **end**
8 **end**
9 $\mathbf{min\_cost} \leftarrow \infty$
10 $\mathcal{G}_{min} \leftarrow \{\}$
11 **for** each Transform Combination for all vertex v **do**
12 $\mathbf{current\_cost} \leftarrow 0$
13 **for** each vertex $v \in \mathcal{V}$ **do**
14 **if** $v \in \mathcal{V}_{loop}$ **then**
15 $\mathbf{current\_cost} \leftarrow \mathbf{current\_cost} + Cost(\mathbf{Transform}(v))$
16 **else**
17 $\mathbf{current\_cost} \leftarrow \mathbf{current\_cost} + Cost(v)$
18 **end**
19 **end**
20 **for** each edge $(u, v) \in \mathcal{E}$ **do**
21 $\mathbf{current\_cost} \leftarrow \mathbf{current\_cost} + Cost(\mathbf{Conversion}(u, v))$
22 $\mathcal{V}.append($Convertion_Operation$)$
23 **end**
24 **if** $\mathbf{current\_cost} < \mathbf{min\_cost}$ **then**
25 $\mathbf{min\_cost} \leftarrow \mathbf{current\_cost}$
26 $\mathcal{G}_{min} \leftarrow \mathcal{G}_{Current}$
27 **end**
28 **end**
Return : Transformed DAG with transforms $\mathcal{G}_{min}$

4.3.2. Cost Model Establishment. Given the large number of possible transformation options, it is crucial to evaluate each of the options based on a concrete set of cost models. In CHLOE, we establish the set of cost models for each of the FHE operators described in Section 2.2 based on the number of polynomial multiplication calls, since polynomial multiplication is the dominant cost for most FHE operators. Our analysis includes two classes of operators: namely, basic FHE operators and advanced FHE operators. Basic operators include plaintext-ciphertext multiplication, ciphertext-ciphertext multiplication, ciphertext rotation, and the external product operators. Advanced operators are those that can be built out of the basic operators, such as homomorphic linear transformation, homomorphic polynomial evaluation, programmable bootstrapping and SIMD programmable bootstrapping. Due to space constraint, the complete set of operators along with the cost analyses are placed in Section A.2.

4.4. Cost-Aware Type Alignment

Based on the cost models, we align ciphertext and operator types in a cost-efficient manner. Here, we formulate the program into a *Def-Use* dependency graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where each vertex is a code segment and edges represent

the dependencies between the segments. We note that any two neighboring vertices in G have different operation types due to the above loop segmentation passes. Therefore, when evaluating the overall computation cost of the program, we consider not only the costs of loop segments per se but also the costs of ciphertext type conversion and input/output encoding alignments. Hence, the cost model of the overall program can be formulated as:

$$Cost_{prog} = \sum_{V} Cost_{seg}(v_i) + \sum_{E} Cost_{conv}(E_{v_i \to v_j}). \quad (4)$$

To achieve efficient cost-based type alignment, we propose the algorithm illustrated in Algorithm 2 to find a graph path with the minimal cost. On Line 1-7 in Algorithm 2, we define the potential transformations based on the loop segment types as analyzed in Section 4.3. Then, on Line 11-24, we traverse all the possible transformation combinations across the graph. In this process, conversion operations between segments are inserted automatically for ciphertext type alignment based on the rewriting rules. In each traversal iteration, we assign different transformation types to the vertices (i.e., loop segments), and the overall cost is updated at the end of the iteration to identify if the current graph achieves the minimal evaluation cost. Lastly, we execute the program transformation according to $\mathcal{G}_{min}$, the segmentation graph with the minimal amount of computational cost. At this stage, all processes in the transparent loop transformation stage has finished, and the program is ready to be further lowered to fundamental ciphertext dialects for executable program generation.

5. Oblivious Loop Transformation

As noted in Section 3.1, oblivious loops are much harder to compile than transparent loops (Section 4) due to the obliviousness of the private loop conditions. By default, the encryption of the loop condition turns any loop into an infinite one. However, by carefully inspecting the different looping structures, we identify three main types of oblivious loops: closed-form oblivious loops, numeric oblivious loops, and general oblivious loops. First, as later explained in Section 5.1, we discover that a compiler can help to transform closed-form oblivious loops into single-line analytical expressions that do not involve iterative control structures, significantly improving the actual program performance. Second, by analyzing the data types of the loop variables, we can statically determine the termination condition for certain loops (termed numeric loops in Section 5.2). Lastly, general oblivious loops refer to loops that do not give any plaintext information on their termination conditions at compile time. For general oblivious loops, we need to add protocol-level primitives to help the program evaluator determine the exact loop termination status.

The main steps that CHLOE takes to handle oblivious loops are sketched in Figure 6. First, in (1) closed-form analysis (Section 5.1), we analyze and try to derive closed-form expressions for the input loop segment. If the loop cannot be expressed in closed form, we perform (2) static analysis (Section 5.2) to determine the exact loop type. Next, in step (3) branch optimization (Section 5.3), we carry out fine-grained control-path reduction to further optimize the performance of numeric oblivious loops. In what follows, we provide more detailed explanations for each of the transformation step.

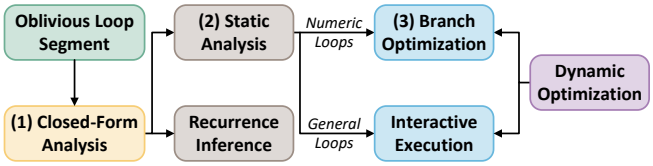


Figure 6. The overall transformation process for oblivious loops.

5.1. Closed-Form Analysis

In CHLOE, once a loop is identified to be oblivious, closed-form analysis is first applied to the loop before any other optimization techniques. Here, we take an α-order ($\alpha \in \mathbb{N}$) recurrence relation R on a sequence $f(\kappa)$ for $\kappa \in \mathbb{N}$ for example, where

$$f(\kappa + \alpha) = R(f(\kappa), \cdots, f(\kappa + \alpha), \kappa)). \quad (5)$$

It is easy to see that Eq. (5) is equivalent to a `FOR` loop that apply some function R on $f(\kappa)$ and κ in the $(\kappa + \alpha)$-th iteration step. When κ is a private variable (i.e., encrypted), such `FOR` loop can be treated as unsolvable by existing compilers [37]. In contrast, we observe that a number of recurrence relations, such as those belong to Gosper-summable and C-finite recurrences, have closed-form expression where the function $f(\kappa)$ can be directly solved without iteration. For instance, C-finite recurrence relations can be reduced to the general form of

$$f(\kappa) = \tilde{a}_0(\kappa)\Theta_0^{\kappa} + \cdots + \tilde{a}_{\psi-1}\Theta_{\psi-1}^{\kappa-1}, \quad (6)$$

where $\Theta_0, \cdots, \Theta_{\psi-1}$ are ψ distinct roots of the characteristic polynomial of $f(\kappa)$, and $\tilde{a}_i(\kappa)$'s are polynomials with degrees less than the multiplicity of Θ_i for $i = 0, \cdots, \psi-1$ [93, 94]. In particular, it is known that any linear recurrence with equality-checking conditional statements can be formulated as C-finite recurrence relations [94]. Therefore, by applying closed-form analysis, we are able to efficiently execute a number of oblivious loops that are practically useful (as exemplified in Section 6.2.2).

Remark: Note that, expressing a loop in its closed form can be quite non-trivial for programmers who are not familiar with FHE. For example, the closed form for a general Fibonacci program Fibonacci(a, b, n) is $F_n = (\frac{\sqrt{5}-1}{2\sqrt{5}}a + \frac{1}{\sqrt{5}}b)(\frac{1+\sqrt{5}}{2})^n + (\frac{\sqrt{5}+1}{2\sqrt{5}}a - \frac{1}{\sqrt{5}}b)(\frac{1-\sqrt{(5)}}{2})^n$. Since coding in such a manner can be highly counter-intuitive, we believe that compiler-assisted loop translation can be essential in enhancing the practical performance of general programs over FHE.

5.2. Static Loop Analysis

Static loop analysis is a typical tool in plaintext program synthesis to infer the worse-case number of iterations at compile time [95–97]. For oblivious loops, static analysis becomes crucial in inferring the structures and termination conditions of loops. For programs over FHE, we basically have two main types of oblivious loops: the numeric type and the general type, where the classification is based on whether the loop variable is an induction variable. If the loop variable is an induction variable, it is referred to as a numeric oblivious loop. For numeric oblivious loops, we can easily decide the maximum number of iterations based on the data type of the induction variable. In addition, it is easy to see that an induction variable with narrower domain results in less number of loop unrolls. Thus, CHLOE permits the definition of custom data types, such that the program designer can decide on the exact number of loop bounds to reduce performance overheads. Meanwhile, when the loop variable is not inductive, the loop is known as a general oblivious loops, and it may not be possible to decide how much loop unrolls are needed at compile time. In such case, client-server communications need to be inserted into the final protocol to guarantee program termination, e.g., [15].

5.3. Branch Optimization

After a numeric oblivious loop is fully unrolled, within each of the unrolled iterations, the encrypted execution result needs to be multiplied by the encrypted loop condition, such that the correct iteration step can be homomorphically selected. Consider the following piece of loop example.

```
for (uint8_t i = 0; i <= cipher(200); i++){
  if (a[i] < 10)
    result = a[i];
}
```

Since the loop condition `cipher`(200) is encrypted, we need to homomorphically identify the termination condition, i.e., $i == \texttt{cipher}(200)$, which translates to:

$$\sum_{i=0}^{2^8-1} \big(\texttt{cipher}(\tau_i) \cdot \texttt{cipher}(l_i)\big). \tag{7}$$

Here, we define a vector ℓ where, for each $l_i \in \ell$, $l_i = 1$ when $i = 200$ and $l_i = 0$ elsewhere. Meanwhile, τ_i is the value in the `result` variable after executing the main loop body i times. In other words, since we do not know when to stop, we have to fully unroll a loop originally of length 200 to a $2^8 = 256$ one, and homomorphically select the loop that iterated 200 times out of the 2^8 options.

To achieve such homomorphic selection, we basically need to translate Eq. (7) into concrete FHE operators and data types. Here, the main difficulty is how to efficiently generate ℓ homomorphically, since the product between $\tilde{\tau}$ encoding different values of `result` and $\tilde{l}$ encoding the elements in ℓ can be easily computed using a SIMD multiplication over RLWE ciphertexts (more detailed explanations are provided in the full manuscript). To compute ℓ, we have three different approaches: the SIMD-PBS-based, the polynomial-approximation-based, and the EXTPRODUCT-based. In short, let $\tilde{i}$ and $\tilde{200}$ be two polynomials whose coefficients are all fixed to the values of i and 200, respectively, for the SIMD-PBS approach, the evaluation of $\textsc{cipher}(\tilde{l})$ can be given by

$$\mathsf{RLWE}(\tilde{l}_i) = \text{SIMD-PBS}(\mathsf{RLWE}(\tilde{i}) - \mathsf{RLWE}(\tilde{200})), TZ), \tag{8}$$

where $TZ(x)$ is the zero-testing function, i.e., $TZ(x) = 1$ if and only if $x = 0$. In contrast, for the approximate polynomial case,

$$\mathsf{RLWE}(\tilde{l}_i) = \textsc{HomPoly}_{p_{TZ}}(\mathsf{RLWE}(\tilde{i}) - \mathsf{RLWE}(\tilde{200})), \tag{9}$$

where p_{TZ} is a polynomial that approximates the TZ function. Finally, when the encrypted loop condition is an input to the program, we can actually directly encrypt the condition 200 to be an RGSW ciphertext as $\mathsf{RGSW}(x^{-200})$. In such case, we can skip evaluating Eq. (7) altogether. Instead, we can encode each and every τ_j for $j = 0, \cdots, 2^8 - 1$ into the coefficients of a polynomial $\tilde{r}$ (i.e., $\tilde{r} = \sum_{j=0}^{2^8-1} \tau_j x^j$) and compute

$$\textsc{ExtractLWE}\big(\textsc{ExtProduct}(\tilde{r}, \mathsf{RGSW}(x^{-200})), 0\big), \tag{10}$$

and we get $\mathsf{LWE}(\tau_i)$. Essentially, we rotate the $\tilde{r}$ polynomial such that the fifth coefficient $\tau_{200}x^{200}$ becomes the constant term $\tau_{200}x^0$. We then perform $\mathsf{LWE}(\tau_{200}) = \textsc{ExtractLWE}(\mathsf{RLWE}(\tilde{r} \cdot x^{-200}), 0)$ to extract the constant term and get the output $\mathsf{LWE}(\tau_{200})$ for the loop. Since the evaluation of Eq. (10) only involves a single ciphertext multiplication, the EXTPRODUCT-based approach is significantly faster than the other two. However, the underlying assumption here is that the encrypted loop condition `cipher`(200) can be directly lowered into $\mathsf{RGSW}(x^{-200})$, which may not always be true in the general case.

TABLE 2. THE ENCRYPTION PARAMETERS SETS

Works	Scheme	Parameters
CHLOE	TFHE	$n = 1024,\ q = 2^{16} + 1$
	BFV	$N = 2^{15}, \log_2(Q) = 673, t = 2^{16} + 1$
HEIR [16]	TFHE	$n = 4096,\ \log_2(q) = 96$
	CKKS	$N = 2^{13}, \mathsf{Max}((\log_2(Q)) = 192$
PEGASUS [8]	TFHE	$n = 1024/4096,\ \log_2(q) = 45$
	CKKS	$N = 2^{16}, \mathsf{Max}(\log_2(Q)) = 599$

6. Evaluation

In this section, we evaluate the performance of programs compiled by CHLOE. We first compare CHLOE against the state-of-the-art general-purpose FHE compilers that support mixed-circuit compilation, such as Transpiler [31],

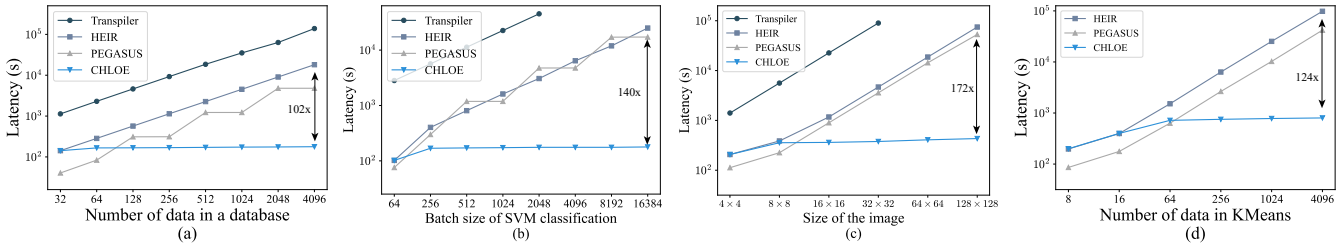


Figure 7. The results for the run time of compiled programs containing transparent loop programs in benchmarks of: (a) database querying (b) SVM classification, (c) iterative threshold algorithm for image segmentation, and (d) KMeans classification with two centroids.

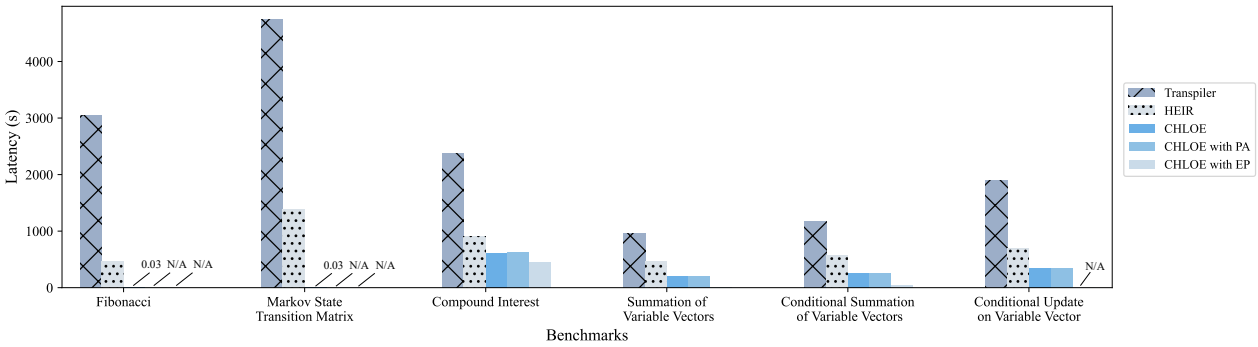


Figure 8. Execution time for different program benchmarks containing oblivious loops where the maximum number of iterations is set to be 64. PA refers to the polynomial-approximation-based branch optimization approach, and EP means the EXTPRODUCT-based approach.

HEIR [16], and programs manually crafted based on the PEGASUS [8] library in Section 6.2. Then, we compare CHLOE against approximation-polynomial based compilation approach based on the OpenFHE [98] framework in Section 6.3. Finally, in Section 6.4, we provide memory usage analysis for the generated programs to provide a more comprehensive view of CHLOE.

6.1. Evaluation Setup

We develop CHLOE based on the MLIR 19.0.0 framework and C++17. The final output program from the CHLOE compiler targets a unified FHE library built on top of SEAL [86], which is publicly available[1]. Our benchmark evaluations are conducted on a machine equipped with an Intel Xeon Gold 5318Y processor and 512GB of RAM, using a single thread for computation.

The FHE parameters instantiated in this work are summarized in Table 2, which are configured to achieve 128-bit security measured by [99]. Note that the RNS representation is employed for both BFV and CKKS ciphertexts, where the upper bound for each modulus is $\mathsf{Max}(\log(q_i)) = 60$.

6.2. Benchmarks

Here, we evaluate the efficiency of programs compiled by CHLOE using a set of benchmark programs that are specifically crafted to test the loop-related optimizations under both transparent and oblivious loop conditions.

1. https://github.com/heir-compiler/CHLOE. Note that the implementation does not contain the SIMD-PBS operator.

6.2.1. Evaluations for Transparent Loop Benchmarks.

When compiling programs with transparent loops, we demonstrate the efficiency of CHLOE on four different applications: database query, SVM classification, image segmentation and KMeans classification. We note that, since PEGASUS does not have compilation capability, we evaluate the benchmark performance using manually-crafted programs implemented over the library [100].

Due to the complexity of the KMeans circuit, which prevents the Transpiler from successfully compiling it for more than 8 data points, the experiment results of Transpiler KMeans classification are excluded from the evaluation.

Performance: As depicted in Figure 7, across all four benchmark evaluations, CHLOE achieves comparable or better execution time compared to Transpiler, PEGASUS and HEIR. Overall, CHLOE achieves a speedup of $1\times$–$102\times$ for database querying, $1\times$–$140\times$ for SVM classification, $1\times$–$172\times$ for image segmentation, and $1\times$–$172\times$ for KMeans classification. Here, we observe that, while the hand-tuned programs of PEGASUS run faster when the data size is small, CHLOE runs significantly faster as the data size increases. By amortizing the bootstrapping cost with SIMD-PBS, the increase in execution time of CHLOE is nearly independent of the vector size. Whereas, the latency of programs produced by other FHE compilers remain linear with respect to the vector size.

6.2.2. Evaluations for Oblivious Loop Benchmarks.

To evaluate the performance of oblivious loops on existing FHE compilers, we rewrite the input programs to transform the oblivious loops into transparent loops, inserting IF statement

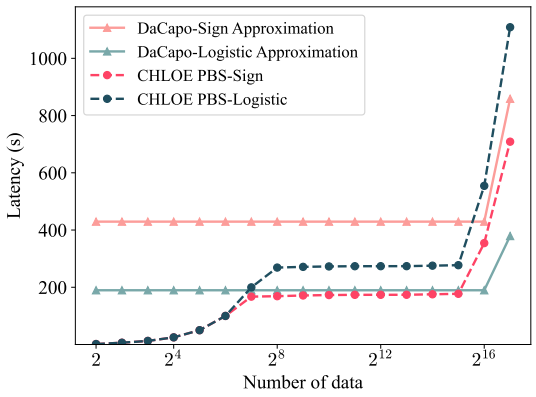


Figure 9. Execution time for evaluating SIGN and LOGISTIC functions with polynomial approximation used in DaCapo and cost-aware CHLOE PBS.

to test the equality between the induction variables and the loop conditions. Meanwhile, as explained in Section 5, CHLOE employs different branch optimization strategies to compile the oblivious loop depending on the induction variable. To show the performance trade-offs, we provide different versions of the same application in the experiment. For instance, when evaluating the program of conditional updates on a variable-length vector, CHLOE uses EXTPRODUCT-based approach to optimize the program when the loop condition is directly given in program input and uses SIMD-PBS-based approach to optimize the other version of the program.

Performance: As illustrated in Figure 8, we present the execution time for evaluating five different applications with various compilers. All loop condition types used in the benchmarks are restricted to uint6_t, translating to a maximum iteration number of $2^6 = 64$. Since we can derive closed-form expressions for the Fibonacci and the Markov state transition matrix benchmarks, such program benchmarks run extremely fast on CHLOE, where end-to-end program execution runs under 100 ms, translating to a speedup of $10^5\times$ compared to [16, 31]. For the other three benchmarks of compound interest, summation of variable-length vectors and conditional vector updates, we implement the three different types of loop branching as described in Section 5.3. Overall, CHLOE achieves of $1.4\times$–$2.2\times$ faster evaluation speed for compound interest and conditional vector update computations, and $6\times$–$50\times$ faster for the summation of a variable-length vector.

6.3. Comparison with Polynomial Approximation

In CHLOE, we primarily employ PBS and SIMD-PBS to evaluate non-arithmetic operations. However, another stream of compilers [27, 101, 102] focus on optimizing CKKS scheme over mixed-circuit programs, where non-arithmetic operations are carried out using polynomial approximations. To compare the efficiency of the CHLOE against CKKS-type FHE compilers [27], we investigate two common non-polynomial functions encountered in real-world applications, namely, the `Sign` function for homomorphic comparison and the `Logistic` function for privacy-preserving data analysis. Since the state-of-the-art CKKS-type compiler [27] is not open-sourced, we manually implement the program based on techniques in [27] using the OpenFHE library [98]. For fair comparison, we apply CKKS bootstrapping after evaluating the non-arithmetic functions to restore the multiplication levels.

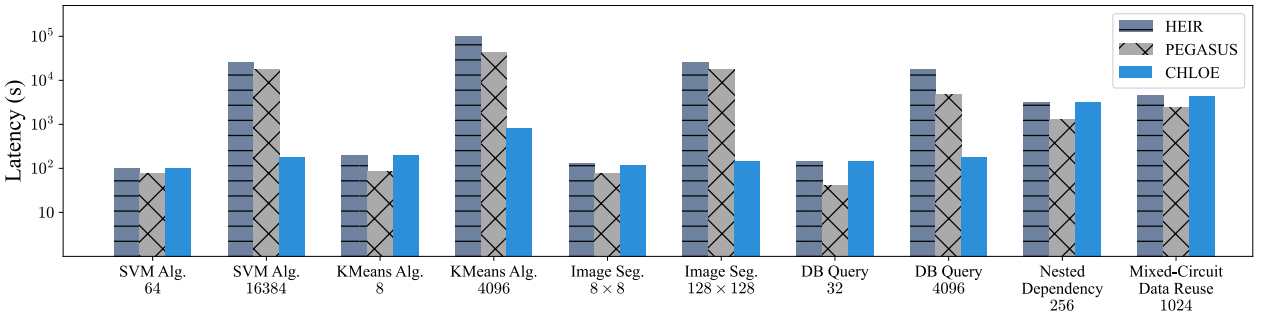
As the result presented in Figure 9, when data size is relatively small, CHLOE effectively uses the PBS from standard TFHE to avoid heavy HOMPOLY operations. As data size increases, CHLOE invokes SIMD-PBS for large data sizes since the overall cost is amortized over multiple data. We note that the efficiency of SIMD-PBS is related to the concrete function to evaluate. For instance, evaluating `Sign` function is more efficient than Logistic function over SIMD-PBS. In contrast, CKKS polynomial approximation is more efficient in evaluating continuous functions such as `Logistic`. Concretely, we generate a 103-order approximation polynomial for the `Logistic` function and a 3989-order one for the `Sign` function using the Chebyshev algorithm [103], both of which can achieve exactly 9-bit data precision. From Figure 9, we can see that, depending on the type of function and the size of the input data, both polynomial approximation and SIMD-PBS can outperform the other. Note that, since DaCapo treats CKKS bootstrapping as a single call of FHE bootstrapping operator, the performance differences shown in Figure 9 come from the fundamental differences of FHE operators, rather than compiler optimizations.

6.4. Discussions

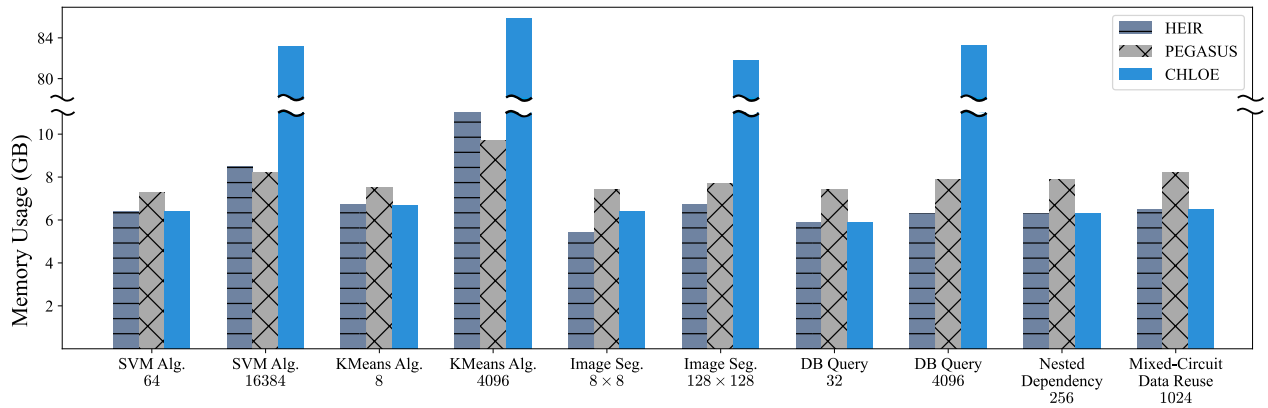
While CHLOE provides substantial performance gains for loop-containing FHE programs, we do identify the following shortcomings and limitations that come as the costs of our optimizations.

Memory Overheads: As illustrated in Figure 10, the significant performance gains from segmenting SIMD-compatible loops do result in worse memory consumption. The memory overheads primarily come from the SIMDPBS operator, and can be further reduced with certain operator optimizations (e.g., [89]).

Nested Dependency: We note that, when there are immediate data dependency between some of the arithmetic and the non-arithmetic statements in the loop, the loop segmentation cannot be performed. In such case, the loop is transformed into a long series of `IF`–`ELSE` statements, and evaluated in a similar way to the HEIR compiler. Hence, As shown in Figure 7 and Figure 10, both latency and memory consumption figures of CHLOE and HEIR come close when there are nested dependencies exist in the application programs (in this case a minimum index program). However, we do point out that, when a loop contains both inseparable (due to dependency) and separable statements, the compiler can group the inseparable statements to form a new loop that is separated from the separable statements, reducing the negative performance impacts of such control structures.



(a) Latency



(b) Memory consumption

Figure 10. Time-memory trade-off comparison across six benchmark programs. For SVM, KMeans, iterative threshold image segmentation and database querying, CHLOE consumes more memory due to the use of SIMD-PBS when the number of input data becomes large. By using SIMD-PBS, CHLOE provides approximately 100-fold reduction in latency. However, for programs with nested dependency and mixed-circuit data reuse, SIMD optimizations are not applicable. Hence, under these conditions, the performance of CHLOE is comparable to HEIR in both latency and memory consumption.

Mixed-Circuit Data Reuse: At its current state, CHLOE has a corner case limitation when a vector element is simultaneously used as by an arithmetic statement (e.g., $\mathsf{c[i]} = \mathsf{a[i]} * \mathsf{b[i]}$) and a non-arithmetic statement (e.g., $\mathsf{if(a[i]} > 0)$). This is more of an implementation limitation, and can be solved by inserting ciphertext conversion operators in between segmented loops.

7. Conclusion

We propose CHLOE, a loop-compatible compiler framework for the efficient transformation of general FHE programs. We introduce a set of new loop-related transformation passes into a multi-level intermediate representation system to effectively simplify and optimize complex loops with interleaved arithmetic and non-arithmetic computations. In the experiment, we show that programs generated by CHLOE can outperform those of existing FHE compilers by orders of magnitudes. Additionally, we demonstrate that automated cost-based program compilation can significantly improve the evaluation of loop-related FHE operations by up to $54\times$.

Acknowledgement

We thank the anonymous reviewers and shepherds for their helpful feedback. This work was supported by the National Key R&D Program of China (2023YFB3106200) and the National Natural Science Foundation of China (U21B2021, 61932014, 62472015). This work was also supported by Huawei Technologies Co., Ltd. Jianwei Liu is the corresponding author.

References

[1] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference service for neural networks," in *USENIX Security*, pp. 2505–2522, 2020.

[2] Z. Huang, W. Lu, C. Hong, and J. Ding, "Cheetah: Lean and fast secure two-party deep neural network inference," in *USENIX Security*, pp. 809–826, 2022.

[3] S. Bian, W. Jiang, Q. Lu, Y. Shi, and T. Sato, "NASS: Optimizing secure inference via neural architecture search," in *ECAI*, pp. 1746–1753, 2020.

[4] S. Bian, T. Wang, M. Hiromoto, Y. Shi, and T. Sato, "ENSEI: efficient secure inference via frequency-domain homomorphic convolution for privacy-preserving visual recognition," in *CVPR*, pp. 9400–9409, 2020.

[5] S. Eskandarian and M. Zaharia, "Oblidb: oblivious

query processing for secure databases," in *VLDB*, p. 169–183, 2019.

[6] L. Folkerts, C. Gouert, and N. G. Tsoutsos, "Redsec: Running encrypted discretized neural networks in seconds," in *NDSS*, 2023.

[7] S. Bian, D. Kundi, K. Hirozawa, W. Liu, and T. Sato, "APAS: application-specific accelerators for rlwe-based homomorphic linear transformations," *IEEE Trans. Inf. Forensics Secur.*, vol. 16, pp. 4663–4678, 2021.

[8] W. Lu, Z. Huang, C. Hong, Y. Ma, and H. Qu, "PEGASUS: bridging polynomial and non-polynomial evaluations in homomorphic encryption," in *IEEE S&P*, pp. 1057–1073, 2021.

[9] K. Cong, D. Das, J. Park, and H. V. L. Pereira, "Sortinghat: Efficient private decision tree evaluation via homomorphic encryption and transciphering," in *ACM CCS*, pp. 563–577, 2022.

[10] R. Akhavan Mahdavi, H. Ni, D. Linkov, and F. Kerschbaum, "Level up: Private non-interactive decision tree evaluation using levelled homomorphic encryption," in *ACM CCS*, pp. 2945–2958, 2023.

[11] Q. Lou and L. Jiang, "HEMET: A homomorphic-encryption-friendly privacy-preserving mobile neural network architecture," in *ICML*, pp. 7102–7110, 2021.

[12] X. Ren, L. Su, Z. Gu, S. Wang, F. Li, Y. Xie, S. Bian, C. Li, and F. Zhang, "HEDA: multi-attribute unbounded aggregation over homomorphically encrypted database," *VLDB*, pp. 601–614, 2022.

[13] S. Bian, Z. Zhang, H. Pan, R. Mao, Z. Zhao, Y. Jin, and Z. Guan, "HE^{3}DB: An efficient and elastic encrypted database via arithmetic-and-logic fully homomorphic encryption," in *ACM CCS*, pp. 2930–2944, 2023.

[14] Z. Zhang, S. Bian, Z. Zhao, R. Mao, H. Zhou, J. Hua, Y. Jin, and Z. Guan, "ArcEDB: an arbitrary-precision encrypted database via (amortized) modular homomorphic encryption," in *ACM CCS*, 2024.

[15] K. Matsuoka, R. Banno, N. Matsumoto, T. Sato, and S. Bian, "Virtual secure platform: A five-stage pipeline processor over TFHE," in *USENIX Security*, pp. 4007–4024, 2021.

[16] S. Bian, Z. Zhao, Z. Zhang, R. Mao, K. Suenaga, Y. Jin, Z. Guan, and J. Liu, "HEIR: A unified representation for cross-scheme compilation of fully homomorphic computation," in *NDSS*, 2024.

[17] C. Juvekar, V. Vaikuntanathan, and A. P. Chandrakasan, "GAZELLE: A low latency framework for secure neural network inference," in *USENIX Security*, pp. 1651–1669, 2018.

[18] A. Kim, Y. Polyakov, and V. Zucca, "Revisiting homomorphic encryption schemes for finite fields," in *ASIACRYPT*, pp. 608–639, 2021.

[19] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "EVA: an encrypted vector arithmetic language and compiler for efficient homomorphic computation," in *PLDI*, pp. 546–561, 2020.

[20] Q. Lou, S. Bian, and L. Jiang, "Autoprivacy: Automated layer-wise parameter selection for secure neural network inference," *NeurIPS*, vol. 33, pp. 8638–8647, 2020.

[21] E. Crockett, C. Peikert, and C. Sharp, "ALCHEMY: A language and compiler for homomorphic encryption made easy," in *ACM CCS*, pp. 1020–1037, 2018.

[22] D. Archer, J. Trilla, J. Dagit, A. Malozemoff, Y. Polyakov, K. Rohloff, and G. Ryan, "RAMPARTS: A programmer-friendly system for building homomorphic encryption applications," in *WAHC*, pp. 57–68, 2019.

[23] M. Cowan, D. Dangwal, A. Alaghi, C. Trippel, V. T. Lee, and B. Reagen, "Porcupine: a synthesizing compiler for vectorized homomorphic encryption," in *PLDI*, pp. 375–389, 2021.

[24] A. Viand, P. Jattke, M. Haller, and A. Hithnawi, "HECO: fully homomorphic encryption compiler," in *USENIX Security*, pp. 4715–4732, 2023.

[25] Y. Lee, S. Heo, S. Cheon, S. Jeong, C. Kim, E. Kim, D. Lee, and H. Kim, "HECATE: performance-aware scale optimization for homomorphic encryption compiler," in *CGO*, pp. 193–204, 2022.

[26] Y. Lee, D. Kim, D. Lee, and H. Kim, "Elasm: Error-latency-aware scale management for fully homomorphic encryption," in *USENIX Security*, pp. 1–15, 2023.

[27] S. Cheon, Y. Lee, D. Kim, J. M. Lee, S. Jung, T. Kim, D. Lee, and H. Kim, "Dacapo: Automatic bootstrapping management for efficient fully homomorphic encryption," in *USENIX Security*, 2024.

[28] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR ePrint*, 2012.

[29] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *ITCS*, pp. 309–325, 2012.

[30] J. H. Cheon, A. Kim, M. Kim, and Y. S. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *ASIACRYPT*, pp. 409–437, 2017.

[31] S. Gorantala, R. Springer, and S. P. , et. al., "A general purpose transpiler for fully homomorphic encryption," *IACR ePrint*, 2021.

[32] C. Gouert and N. G. Tsoutsos, "Romeo: Conversion and evaluation of HDL designs in the encrypted domain," in *DAC*, pp. 1–6, 2020.

[33] Google, "HEIR: Homomorphic Encryption Intermediate Representation." https://github.com/google/heir, 2024.

[34] E. Chielle, O. Mazonka, H. Gamil, and M. Maniatakos, "Accelerating fully homomorphic encryption by bridging modular and bit-level arithmetic," in *ICCAD*, 2022.

[35] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: fast fully homomorphic encryption over the torus," *J. Cryptol.*, vol. 33, no. 1, pp. 34–91, 2020.

[36] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[37] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic, "SoK: General purpose compilers for secure multi-party computation," in *IEEE S&P*, pp. 1220–1237, 2019.

[38] K. Kennedy and K. S. McKinley, "Maximizing loop parallelism and improving data locality via loop fusion and distribution," in *International Workshop on Languages and Compilers for Parallel Computing*, pp. 301–320, Springer, 1993.

[39] Z. Liu and Y. Wang, "Amortized functional bootstrapping in less than 7 ms, with o˜(1) polynomial multiplications," in *ASIACRYPT*, pp. 101–132, 2023.

[40] S. Chowdhary, W. Dai, K. Laine, and O. Saarikivi, "EVA improved: Compiler and extension library for CKKS," in *WAHC*, pp. 43–55, 2021.

[41] A. Viand and H. Shafagh, "Marble: Making fully homomorphic encryption accessible to all," in *WAHC*, pp. 49–60, 2018.

[42] R. Malik, K. Sheth, and M. Kulkarni, "Coyote: A compiler for vectorizing encrypted arithmetic circuits," in *ASPLOS*, pp. 118–133, 2023.

[43] R. Recto and A. C. Myers, "A compiler from array programs to vectorized homomorphic encryption," *CoRR*, 2023.

[44] Z. Guan, R. Mao, Q. Zhang, Z. Zhang, Z. Zhao, and S. Bian, "Autohog: Automating homomorphic gate design for large-scale logic circuit evaluation," *IEEE TCAD*, 2024.

[45] S. Park, W. Song, S. Nam, H. Kim, J. Shin, and J. Lee, "Heaan. mlir: An optimizing compiler for fast ring-based homomorphic encryption," *PLDI*, vol. 7, no. PLDI, pp. 196–220, 2023.

[46] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay - secure two-party computation system," in *USENIX Security*, pp. 287–302, 2004.

[47] A. Ben-David, N. Nisan, and B. Pinkas, "FairplayMP: a system for secure multi-party computation," in *ACM CCS*, pp. 257–266, 2008.

[48] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen, "Asynchronous multiparty computation: Theory and implementation," in *PKC*, pp. 160–179, 2009.

[49] Y. Zhang, A. Steele, and M. Blanton, "PICCO: a general-purpose compiler for private distributed computation," in *ACM CCS*, pp. 813–826, 2013.

[50] M. Franz, A. Holzer, S. Katzenbeisser, C. Schallhart, and H. Veith, "CBMC-GC: An ansi c compiler for secure two-party computations," in *CC*, pp. 244–249, 2014.

[51] A. Rastogi, M. A. Hammer, and M. Hicks, "Wysteria: A programming language for generic, mixed-mode multiparty computations," in *IEEE S&P*, pp. 655–670, 2014.

[52] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "ObliVM: A programming framework for secure computation," in *IEEE S&P*, pp. 359–376, 2015.

[53] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor, "Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation," in *IEEE EuroS&P*, pp. 112–127, 2016.

[54] X. Wang, A. J. Malozemoff, and J. Katz, "EMP-toolkit: Efficient MultiParty computation toolkit." https://github.com/emp-toolkit, 2016.

[55] D. Demmler, S. Katzenbeisser, T. Schneider, T. Schuster, and C. Weinert, "Improved circuit compilation for hybrid mpc via compiler intermediate representation," *IACR ePrint*, 2021.

[56] T. Araki, A. Barak, J. Furukawa, M. Keller, Y. Lindell, K. Ohara, and H. Tsuchida, "Generalizing the spdz compiler for other protocols," in *ACM CCS*, pp. 880–895, 2018.

[57] D. Demmler, T. Schneider, and M. Zohner, "ABY-A framework for efficient mixed-protocol secure two-party computation.," in *NDSS*, 2015.

[58] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos, "SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics," in *USENIX Security*, pp. 223–240, 2010.

[59] P. Mohassel and P. Rindal, "ABY3: A mixed protocol framework for machine learning," in *ACM CCS*, pp. 35–52, 2018.

[60] F. Kerschbaum, T. Schneider, and A. Schröpfer, "Automatic protocol selection in secure two-party computations," in *ACNS*, pp. 566–584, 2014.

[61] E. Pattuk, M. Kantarcioglu, H. Ulusoy, and B. Malin, "CheapSMC: A framework to minimize secure multiparty computation cost in the cloud," in *DBSec*, pp. 285–294, 2016.

[62] M. Keller, "MP-SPDZ: A versatile framework for multi-party computation," in *ACM CCS*, pp. 1575–1590, 2020.

[63] L. Braun, D. Demmler, T. Schneider, and O. Tkachenko, "Motion–a framework for mixed-protocol multi-party computation," *ACM TOPS*, vol. 25, no. 2, pp. 1–35, 2022.

[64] Y. Bao, K. Sundararajah, R. Malik, Q. Ye, C. Wagner, N. Jaber, F. Wang, M. H. Ameri, D. Lu, A. Seto, B. Delaware, R. Samanta, A. Kate, C. Garman, J. Blocki, P.-D. Letourneau, B. Meister, J. Springer, T. Rompf, and M. Kulkarni, "HACCLE: Metaprogramming for secure multi-party computation," in *GPCE*, p. 130–143, 2021.

[65] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, and T. Schneider, "HyCC: Compilation of hybrid protocols for practical secure computation," in *ACM CCS*, pp. 847–861, 2018.

[66] T. Heldmann, T. Schneider, O. Tkachenko, C. Weinert, and H. Yalame, "LLVM-Based circuit compilation for practical secure computation," in *ACNS*, pp. 99–121, 2021.

[67] B. Levy, M. Ishaq, B. Sherman, L. Kennard, A. Milanova, and V. Zikas, "COMBINE: compilation and Backend-INdependent vEctorization for multi-party computation," in *ACM CCS*, pp. 2531–2545, 2023.

[68] L. Braun, M. Huppert, N. Khayata, T. Schneider, and O. Tkachenko, "FUSE - flexible file format and intermediate representation for secure multi-party computation," in *ACM ASIACCS*, pp. 649–663, 2023.

[69] E. Chielle, O. Mazonka, N. G. Tsoutsos, and M. Maniatakos, "E^3: A framework for compiling C++ pro-

grams with encrypted operands," *IACR ePrint*, 2018.
[70] S. Carpov, P. Dubrulle, and R. Sirdey, "Armadillo: A compilation chain for privacy preserving applications," in *SCC@ASIACCS*, pp. 13–19, 2015.
[71] I. Chillotti, M. Joye, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks," in *CSCML*, pp. 1–19, 2021.
[72] J. Bossuat, C. Mouchet, J. R. Troncoso-Pastoriza, and J. Hubaux, "Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys," in *EUROCRYPT*, pp. 587–617, 2021.
[73] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption," in *EUROCRYPT*, Springer, 2018.
[74] Y. Lee, J. Lee, Y. Kim, Y. Kim, J. No, and H. Kang, "High-precision bootstrapping for approximate homomorphic encryption by error variance minimization," in *EUROCRYPT*, pp. 551–580, Springer, 2022.
[75] D. Lee, S. Min, and Y. Song, "Functional bootstrapping for fv-style cryptosystems," *IACR ePrint*, 2024.
[76] Y. Bae, J. H. Cheon, J. Kim, and D. Stehlé, "Bootstrapping bits withnbsp;ckks," in *EUROCRYPT*, p. 94–123, 2024.
[77] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *CRYPTO*, pp. 868–886, 2012.
[78] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *STOC*, pp. 84–93, 2005.
[79] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *CRYPTO*, 2013.
[80] L. Ducas and D. Micciancio, "FHEW: bootstrapping homomorphic encryption in less than a second," in *EUROCRYPT*, pp. 617–640, 2015.
[81] J. Alperin-Sheriff and C. Peikert, "Faster bootstrapping with polynomial error," in *CRYPTO*, pp. 297–314, Springer, 2014.
[82] A. Khedr, G. Gulak, and V. Vaikuntanathan, "Shield: scalable homomorphic implementation of encrypted data-classifiers," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2848–2858, 2015.
[83] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," in *CRYPTO*, pp. 850–867, 2012.
[84] C. Gentry, S. Halevi, and N. P. Smart, "Fully homomorphic encryption with polylog overhead," in *EUROCRYPT*, pp. 465–482, 2012.
[85] N. P. Smart and F. Vercauteren, "Fully homomorphic SIMD operations," *Des. Codes Cryptogr.*, vol. 71, no. 1, pp. 57–81, 2014.
[86] "Microsoft SEAL (release 3.7)." https://github.com/Microsoft/SEAL, Sept. 2021. Microsoft Research.
[87] D. Micciancio and J. Sorrell, "Ring packing and amortized FHEW bootstrapping," in *ICALP*, pp. 100:1–100:14, 2018.
[88] H. Chen, W. Dai, M. Kim, and Y. Song, "Efficient homomorphic conversion between (ring) LWE ciphertexts," in *ACNS*, pp. 460–479, 2021.
[89] K. Han, M. Hhan, and J. H. Cheon, "Improved homomorphic discrete fourier transforms and FHE bootstrapping," *IEEE Access*, vol. 7, pp. 57361–57370, 2019.
[90] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption," in *EUROCRYPT*, pp. 360–384, 2018.
[91] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. A. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: scaling compiler infrastructure for domain specific computation," in *CGO*, pp. 2–14, 2021.
[92] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, "Polygeist: Raising C to polyhedral MLIR," in *PACT*, pp. 45–59, 2021.
[93] A. Humenberger, D. Amrollahi, N. Bjørner, and L. Kovács, "Algebra-based reasoning for loop synthesis," *Formal Aspects of Computing*, vol. 34, no. 1, pp. 1–31, 2022.
[94] C. Wang and F. Lin, "Solving conditional linear recurrences for program verification: The periodic case," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 28–55, 2023.
[95] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution," in *RTSS*, pp. 57–66, IEEE, 2006.
[96] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel, "A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models," in *CGO*, pp. 136–146, IEEE, 2009.
[97] T. Sewell, F. Kam, and G. Heiser, "Complete, high-assurance determination of loop bounds and infeasible paths for wcet analysis," in *RTAS*, pp. 1–11, IEEE, 2016.
[98] A. Badawi, J. Bates, F. Bergamaschi, *et al.*, "OpenFHE: Open-source fully homomorphic encryption library," in *WAHC*, pp. 53–63, 2022.
[99] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," *J. Math. Cryptol.*, vol. 9, no. 3, pp. 169–203, 2015.
[100] Alibaba, "OpenPEGASUS." https://github.com/Alibaba-Gemini-Lab/OpenPEGASUS, 2021.
[101] J. Lee, E. Lee, Y. Lee, Y. Kim, and J. No, "High-precision bootstrapping of RNS-CKKS homomorphic encryption using optimal minimax polynomial approximation and inverse sine function," in *EUROCRYPT*, pp. 618–647, 2021.
[102] W. Ao and V. N. Boddeti, "AutoFHE: automated adaption of cnns for efficient evaluation over FHE," in *USENIX Security*, pp. 1–18, 2024.
[103] H. Chen, I. Chillotti, and Y. Song, "Improved bootstrapping for approximate homomorphic encryption," in *EUROCRYPT*, pp. 34–54, 2019.

Appendix Table A1. INSTRUCTIONS AND DATA TYPES SUPPORTED IN SURFACE LANGUAGE

Instructions	Data Type	Description
+, +=	FP16	Addition
-, -=	FP16	Substraction
*, *=	FP16	Multiplication
=	FP16	Assignment
==	FP8	Equality comparison
>, ≥	FP8	Greater than operator
<, ≤	FP8	Less than operator
a[i]	FP16	Member access from an array
if else	FP8	If Else statement
for	FP8/FP16	For Loop statement
Function Call	FP8	Instructions defined by function call

```
1  func.func @data_analysis(%arg0: cipherVec<512xi16>)
                                        -> cipher {
2    %c400 = fhe.constant 400 : plain
3    %c0 = fhe.constant 0 : plain
4    %0 = fhe.constant 0 : cipherVec<512xi16>
5    affine.for %arg1 = 0 to 512 {
6    %1 = fhe.load %arg0[%arg1] : cipherVec<512xi16>
7    %2 = fhe.cmp lt, %1, %c400 : cipher
8    fhe.store %2, %0[%arg1] : cipherVec<512xi16>
9    }
10   %3 = affine.for %arg2 = 0 to 512
11             iter_args(%arg3 = %c0) -> (cipher) {
12   %4 = fhe.load %arg0[%arg2] : cipherVec<512xi16>
13   %5 = fhe.load %0[%arg2] : cipherVec<512xi16>
14   %6 = fhe.mult %5, %4 : cipher
15   %7 = fhe.add %arg3, %6 : cipher
16   affine.yield %7 : cipher
17   }
18   return %3 : cipher
19 }
```

Appendix Figure A1. The data analysis program after loop segmentation passes.

Appendix A. Supplementary Materials

A.1. Example Program after Loop Segmentation

In Figure A1, we present the example data analysis program after applying loop segmentation passes. Compared to the original program, the loop structure is segmented into two independent loops. The first loop on Lines 5-9 performs the non-arithmetic operations for comparison (Lines 5-6 in Figure 4(a)). The second loop on Lines 10-17 performs arithmetic operations for data aggregation (Lines 7-10 in Figure 4(a)) and inherits the loop-carried variable `%c0` from the original loop. To address the boundary variable vectorization in the loop splitting step, a vector variable `%0` with `cipherVec<512xi16>` type is created to store the results of `%2` from different iterations. Subsequently, all the uses of `%2` in the second loop are replaced by `%0`, with an additional VECTORLOAD operator introduced on Line 13.

A.2. FHE Operator Cost Analysis

The cost models for each of the FHE operators are summarized as follows. We established the cost models based on complexity analysis in terms of the number of degree-N polynomial multiplications required, where N is the lattice dimension for the RLWE ciphertexts.

- Plaintext-Ciphertext Multiplication: The computation cost between a plaintext polynomial and an RLWE ciphertext $\mathsf{RLWE}_{\tilde{s}}^{N,\prod_{i=1}^{k} q_i}(\tilde{m})$ in RNS representation with k moduli is $L_{PM} = 2k$.

- Ciphertext Multiplication: For multiplications between RLWE ciphertexts with k moduli, there exists $4k$ polynomial multiplications. In the relinearization phase, the computation between the linearization key and multiplied ciphertext requires $2k$ polynomial multiplications. Therefore, the computation cost of ciphertext multiplication is $L_{CM} = 6k$.

- Ciphertext Rotation: In this operation, we omit the automorphism part and only consider the cost of key switching. Considering a key-switching key with a decomposition parameter ℓ, the cost of rotating a ciphertext with k moduli is $L_{Rot} = 2k\ell$.

- External Product: In this operation, multiplication is carried out between an RGSW ciphertext with a decomposition parameter ℓ and an RLWE ciphertext with k moduli. The process of an external product performs 2ℓ plaintext-ciphertext multiplications in parallel. Therefore, the computation cost of EXTPRODUCT is $L_{EP} = 4k\ell$.

Based on the computation cost model of the above low-level homomorphic operations, the cost of numerous high-level homomorphic operations are given as follows.

- Linear Transform: By using the Giant-Step Baby-Step (BSGS) technique, linear transform can be evaluated on an RLWE ciphertext and a plain matrix $\mathbf{A} \in Z_p^{R\times N}$ where $R < N$ is power-of-2. The cost can be represented as $L_{LT} = (2\sqrt{R} + \log(N/R))L_{Rot} + \sqrt{R}L_{PM}$. We note that encoding conversion operations CTOS and STOC are equivalent to a linear transform with $\mathbf{A} \in Z_p^{N\times N}$.

- PBS: For an LWE ciphertext $\mathsf{LWE}_{\mathbf{s}}^{N,q}(m)$, PBS can be evaluated through computing external product and accumulation iteratively. Thus, the cost of PBS is $L_{PBS} = 2NL_{EP}$.

- SIMD-PBS: This operation can evaluate PBS on $R = 2^r$ LWE ciphertexts simultaneously. The process of SIMD-PBS is composed of a linear transform and a HOMPOLY operation. Therefore, the cost of SIMD-PBS is $L_{SPBS} = (2\sqrt{R}+\log(N/R))L_{Rot}+\sqrt{R}L_{PM}+\sqrt{2t}L_{CM}$.

- PackLWE: In this operation, $R = 2^r$ LWE ciphertexts can be packed into an RLWE ciphertext either in slot encoding or coefficient encoding. For slot encoding packing, the procedure and computation cost are identical to SIMD-PBS. For coefficient encoding [88], the cost of repacking operation is $L_{Pack} = (R - 1 + log(N/R))L_{Rot}$.

A.3. Branch Optimization for Oblivious Loops

Here, we discuss how an oblivious loop can be evaluated through a full loop unroll. Consider the same simple loop segment as discussed in Section 5.3

```
for (uint8_t i = 0; i < cipher(200); i++){
  if (a[i] < 10)
    result = a[i]
}
```

The idea of a full loop unroll is that, regardless of the exact value of the encrypted condition `cipher`(200), the loop will always terminate within 256 iterations since the value of the `uint8-t`-type variable i can only take 256 possibilities (i.e., $i = 2^8$). Consequently, we can transform the above loop into the evaluation of the following equation:

$$\sum_{i=0}^{256} \texttt{result}_i \cdot (\texttt{i == cipher}(200))$$

where $\texttt{result}_i$ is the value of `result` at the i-th loop iteration. In other words, to generate an output variable from a fully unrolled loop, we need to take an inner product between a sequnce of particular variable (`result` in this case) at each iteration step and the sequence of the loop condition `i == cipher`(200) at each iteration step. Since the condition `i == cipher`(200) only holds true when $i = 200$, we have that

$$\ell = [0 \ \cdots \ 0 \ 1 \ \cdots \ 0], \text{ and} \tag{A1}$$

$$\tau = [\texttt{result}_0 \ \texttt{result}_1 \ \cdots \ \texttt{result}_{255}], \tag{A2}$$

and we can see that $\ell \circ \tau$ (where $\circ$ is the Hadamard product) can homomorphically clear all but 200-th element in τ. Therefore, we can encode both ℓ and τ as the coefficients of two polynomials, $\tilde{l}$ and $\tilde{\tau}$, respectively, which can be encrypted as ciphertexts or produced by homomorphic computations. Then, a homomorphic Hadamard produce can be used to clear the encrypted $\tilde{\tau}$ using $\tilde{l}$.

Note that, in Section 5.3, we introduced three ways of generating $\tilde{l}$. Since Eq. (8) and Eq. (9) involve the evaluations of ciphertext bootstrapping, it appears that such approaches are less performant than the solution in Eq. (10). Nonetheless, we note that Eq. (10) requires the condition variable to be encrypted in a special format, namely, $\mathsf{RGSW}(x^{-200})$, which may not be practical if the conditional variable is generated on-the-fly instead of given as inputs to the program. While it is possible to convert both LWE and RLWE ciphertexts to RGSW ciphertext with exponent encoding, such conversion involves non-trivial use of circuit bootstrapping [71], and is not as efficient as Eq. (8) and Eq. (9).