HALO: Loop-aware Bootstrapping Management for Fully Homomorphic Encryption

Seonyoung Cheon
Yonsei University
Seoul, Republic of Korea
seonyoung@yonsei.ac.kr

Yongwoo Lee
Yonsei University
Seoul, Republic of Korea
dragonrain96@yonsei.ac.kr

Hoyun Youm
Yonsei University
Seoul, Republic of Korea
hoyun@yonsei.ac.kr

Dongkwan Kim
Yonsei University
Seoul, Republic of Korea
dongkwan@yonsei.ac.kr

Sungwoo Yun
Yonsei University
Seoul, Republic of Korea
sungwoo@yonsei.ac.kr

Kunmo Jeong
Yonsei University
Seoul, Republic of Korea
kunmo@yonsei.ac.kr

Dongyoon Lee
Stony Brook University
New York, USA
dongyoon@cs.stonybrook.edu

Hanjun Kim
Yonsei University
Seoul, Republic of Korea
hanjun@yonsei.ac.kr

Abstract

Thanks to the computation ability on encrypted data, fully homomorphic encryption (FHE) is an attractive solution for privacy-preserving computation. Despite its advantages, FHE suffers from limited applicability in small programs because repeated FHE multiplications deplete the level of a ciphertext, which is finite. Bootstrapping reinitializes the level, thus allowing support for larger programs. However, its high computational overhead and the risk of level underflow require sophisticated bootstrapping placement, thereby increasing the programming burden. Although a recently proposed compiler automatizes the bootstrapping placement, its applicability is still limited due to lack of loop support.

This work proposes the first loop-aware bootstrapping management compiler, called HALO, which optimizes bootstrapping placement in an FHE program with a loop. To correctly support bootstrapping-enabled loops, HALO matches encryption status and levels between live-in and loop-carried ciphertexts in the loops. To reduce the bootstrapping overheads, HALO decreases the number of bootstrapping within a loop body by packing the loop-carried variables to a single ciphertext, reduces wasted levels in a short loop body by unrolling the loop, and optimizes the bootstrapping latency by adjusting the target level of bootstrapping as needed. For seven machine learning programs with flat and nested loops, HALO shows 27% performance speedup compared to the state-of-the-art compiler that places bootstrapping operations on fully unrolled loops. In addition, HALO reduces the compilation time and code size by geometric means of 209.12× and 11.0× compared to the compiler, respectively.

ASPLOS '25, March 30–April 3, 2025, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0698-1/25/03.
https://doi.org/10.1145/3669940.3707275

CCS Concepts: • **Software and its engineering** → **Compilers**; **Software usability**; • **Security and privacy** → **Privacy-preserving protocols**.

Keywords: Fully homomorphic encryption, CKKS, Bootstrapping, Loop optimization, Compiler, Privacy-preserve machine learning

ACM Reference Format:
Seonyoung Cheon, Yongwoo Lee, Hoyun Youm, Dongkwan Kim, Sungwoo Yun, Kunmo Jeong, Dongyoon Lee, and Hanjun Kim. 2025. HALO: Loop-aware Bootstrapping Management for Fully Homomorphic Encryption. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3669940.3707275

1 Introduction

Fully homomorphic encryption (FHE) [28] offers an attractive solution for a privacy-preserving application. FHE enables computations on encrypted data, thus opening up secure cloud computing and privacy-preserving data analytics in privacy sensitive fields such as finance and healthcare. Among various FHE schemes [10–12, 15, 25, 28], RNS-CKKS [11] stands out for its support of fixed-point arithmetic and single instruction multiple data (SIMD) computations, enlarging its applicability to applications requiring high computational throughput, such as machine learning and data analytics on encrypted data.

Despite its promise, implementing an FHE application poses significant challenges due to the complex encryption parameter management. In RNS-CKKS, a ciphertext consists of multiple vectors, called residue polynomial, the number of which is called "level". Repeated multiplications require level management operations that consume a level of a ciphertext, eventually depleting the level entirely. *Bootstrapping* is an FHE operation that recovers the consumed level back to the initial level. Since bootstrapping incurs the longest runtime among RNS-CKKS operations, manually managing bootstrapping operations [40, 42] imposes a considerable burden on programmers, and often results in poor performance.

Recently proposed RNS-CKKS compilers [5, 13, 22, 23, 44, 46, 49, 59] can alleviate the burden on FHE programmers by automatizing the encryption parameter management. These compilers optimize FHE applications by analyzing and manipulating ciphertext levels that reflect multiplication depths. However, their applicability has been limited to small applications because most of these compilers, except DaCapo [13], do not support bootstrapping operations. Thus, they cannot support applications beyond their maximum multiplication depths constrained by the initial ciphertext level. Although DaCapo automatizes bootstrapping management, it can only support applications with simple control flows, such as no loops or loops with a static iteration count after full unrolling [13, 59]. Currently no FHE compilers including DaCapo supports applications that contain loops with dynamic iteration counts such as regression algorithms which repeatedly execute a sequence of operations without a predetermined number of iterations.

Supporting an FHE application with loops is challenging due to constraints on encryption parameters of "loop-carried variables" that depend on their values from previous iterations. Additionally, the high overhead of bootstrapping can be significantly amplified within a loop. FHE operations have constraints on their argument ciphertext levels. Since statically placed FHE operations are repeatedly applied to the loop-carried ciphertexts while multiplications in each iteration dynamically change their levels, the levels of the loop-carried ciphertexts should be identical across iterations. Moreover, since the FHE operations including bootstrapping in a loop are repeated as many as loop iteration counts, placing bootstrapping at optimal places is crucial to achieve efficient performance.

This work proposes HALO, a loop-aware automatic bootstrapping management compiler for RNS-CKKS that places and optimizes bootstrapping operations reflecting loop structures. HALO matches the encryption status such as `plain` and `cipher`, and the levels of loop-carried ciphertexts across iterations via loop peeling and bootstrapping placement. This transforms the loop into a *type-matched loop*, ensuring the encryption status and the levels of all loop-carried variables are consistent across iterations. Furthermore, HALO performs three bootstrapping-aware optimizations: 1) decreasing the number of bootstrapping within the loop body by packing the loop-carried variables to a single ciphertext, 2) reducing wasted levels in a short loop body by unrolling the loop reflecting consumed ciphertext levels, and 3) optimizing the bootstrapping latency by adjusting the target level of bootstrapping as needed.

Table 1. FHE parameters and their corresponding values used in the evaluation of HALO.

Parameter	Definition	Value
N	Polynomial modulus degree	2^{17}
Q	Coefficient modulus	2^{1479}
R_f	Rescaling factor	2^{51}
L	Maximum level after bootstrapping	16

HALO is built on top of MLIR [39] with a Python frontend. We evaluated seven machine learning programs that embed flat and nested loops with various iteration counts. The evaluation results show that HALO achieves 27% performance speedup compared to DaCapo [13] that places bootstrapping operations on fully unrolled loops. Especially, the bootstrapping-aware loop optimizations of HALO lead to 2.39× geometric average performance improvement compared to a baseline type-matched loop. In addition, HALO improves the compilation time by geometric means of 209.12× and reduces the code size by up to 11.0× compared to DaCapo.

This work describes the following contributions:

- The first FHE compiler that supports bootstrapping operations in loops.
- A new type-matching process for loop-carried ciphertexts that enables bootstrapping in a loop.
- New bootstrapping-aware loop optimization schemes such as *loop-carried variable packing*, *level-aware unrolling* and *target level tuning*.

2 Background and Motivation

This section explains the RNS-CKKS [11] encoding and encryption processes and relevant operations. RNS-CKKS offers fast approximate computations on encrypted data. This scheme provides SIMD computation (vectorized parallel computation) and fixed-point arithmetic operations. It is particularly well-suited for machine learning applications requiring operations on real or complex numbers.

2.1 RNS-CKKS Scheme

CKKS [12] encodes complex (real) numbers as scaled integer values by multiplying *scale* (*e.g.,* 1.23 is encoded as 123 with scale 100). The encoded values are mapped onto the coefficients of a N-degree cyclotomic polynomial [9], where N is referred to as *polynomial modulus degree*. A higher degree N increases the complexity of the underlying hardness of Ring Learning with Errors (RLWE) [48]. The coefficients of the polynomial are bounded by a large number, known as the *coefficient modulus* Q. This single polynomial form is denoted as plaintext μ before encryption. CKKS encrypts the plaintext μ into a ciphertext, which consists of a pair of

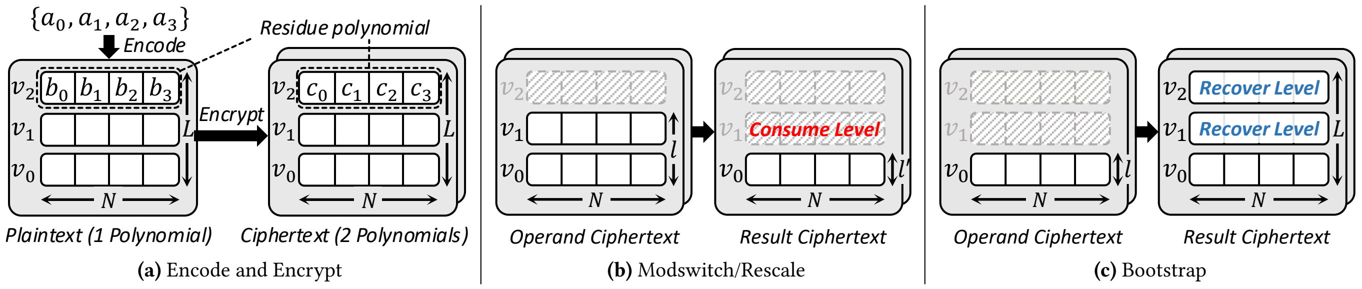


Figure 1. High-level abstraction of a ciphertext and FHE operations that change the ciphertext level, `modswitch/rescale`, and `bootstrap`. In this example figure, the ciphertext parameters are set to N=4, and L_{max}=L=3.

Table 2. Latency of FHE operations at different levels (μs). The results were measured using the GPU-accelerated version of the HEaaN library.

Operation	Operand Level (l)				Result Level (l')
	1	5	10	15	
`multcc`	758	1146	1974	2528	l
`rescale`	126	288	516	731	$l-1$
`modswitch`	15	46	77	107	$l-1$

polynomials $(-A \cdot s + \mu + e, A)$, where A and e are random noises and s is the secret key.

RNS-CKKS [11] is a CKKS variant that applies the residue number system (RNS) to CKKS, making FHE computation more efficient. RNS-CKKS builds the coefficient modulus Q as a product of smaller prime moduli ($Q = \prod q_i$), where each q_i is a prime number. The RNS basis consists of a set of moduli $\{q_0, ..., q_{L-1}\}$, with the maximum number of moduli denoted as the level L. A *residue polynomial* v is defined as a polynomial whose coefficients are represented in residue form with respect to one prime modulus q_i. The *level* l of a ciphertext denotes the number of residue polynomials in the ciphertext. RNS-CKKS operations, such as arithmetic operations, can be performed independently on each residue.

Figure 1a illustrates the plaintext and ciphertext structures with RNS-CKKS parameters. $\{a_0, a_1, a_2, a_3\}$ are encoded to a plaintext before encryption. The encoding process first maps the value vector to the polynomial (coefficient representation) by applying fast fourier transform (FFT). The polynomial coefficients are scaled to integers, $\{a_0', a_1', a_2', a_3'\}$, by multiplying the parameter called scale. Then, encoding process generates the residue polynomial (*e.g.*, $\{b_0, b_1, b_2, b_3\} \equiv \{a_0', a_1', a_2', a_3'\}$ *mod* q_2) by calculating the residue of each coefficient for the corresponding modulus.

2.2 Arithmetic and Rotation Operations

Addition and **Multiplication** operations add/multiply either two ciphertexts (`addcc` and `multcc`, respectively) or one ciphertext and one plaintext (`addcp` and `multcp`). In the case

Table 3. The `bootstrap` operation latency (μs) is in proportion to target level.

Operation	Target Level (l)				
	4	7	10	13	16
`bootstrap`	294928	339302	384637	423781	463171

of addition, the operands are required to have the same scale and level, and the resulting scale and level remain the same as the operands. Multiplication requires only the levels of the operands to be the same. The resulting level remains consistent with the operand level, while the scale increases to the product of the operand scales.

Rotation shifts an encrypted-value vector circularly by a given offset. In the case Figure 1a, rotating the data within the vector by an offset of 2 shifts the data from $\{c_0, c_1, c_2, c_3\}$ to $\{c_2, c_3, c_0, c_1\}$. `rotate` does not manipulate the scale or level of a ciphertext.

2.3 Level Management Operations

Managing the level of ciphertexts is crucial for overall performance and correctness in the RNS-CKKS scheme.

Modswitch/Rescale *consumes the level* of an operand ciphertext from l to $l-1$ by dividing the coefficient modulus Q by rescaling factor S_f, which means dropping the modulus q_l from Q. Figure 1b illustrates the abstraction of a ciphertext after applying `modswitch` and `rescale`. After `modswitch` and `rescale`, the number of residue polynomials to be processed is reduced, thus the latency of succeeding (arithmetic/rotation) operations decreases. Table 2 presents the latency of operations at different levels. RNS-CKKS operations perform faster as the operand level decreases. EVA [22] optimizes performance by using this property, applying `modswitch` early, reducing the levels of overall ciphertexts. If the accumulated scale exceeds the coefficient modulus Q after multiplications, the result will be corrupted. `rescale` prevents scale overflow using *rescaling factor* S_f (*i.e.*, scale/S_f). In addition, to meet the arithmetic operation constraints, `modswitch/rescale` can be used to adjust the operand levels. Compilers like Hecate and others [44–46] propose `rescale` placement strategies to enhance overall performance, such as hoisting `rescale` operations to minimize the level of costly tasks (e.g., multiplication, rotation) and reducing the input ciphertext levels to decrease the levels of subsequent operations.

Bootstrapping *recovers the level* up to the maximum of L as shown in Figure 1c, and it is the most expensive operation in RNS-CKKS. If a program includes a large number of multiplications, ciphertexts will eventually exhaust all their levels, preventing further operations. Therefore, despite its significant time cost, bootstrapping is essential for continued accurate multiplications. DaCapo [13] automatically inserts the `bootstrap` operations, thus enabling a long multiplication chain without increasing programmers' burden. Since the `bootstrap` operation is extremely heavy, DaCapo supports additional `bootstrap` placement optimization. When compared with the latency of `modswitch`, `bootstrap` is over 4,400 times slower. The latency of bootstrapping decreases as the target level (resulting level) gets lower, as shown in Table 3. To enhance the overall performance, DaCapo places the `bootstrap` while balancing overhead between the `bootstrap` the others with higher ciphertext levels.

2.4 Lack of Loop Support in Existing FHE Compilers

Unfortunately, existing FHE compilers [13, 44–46, 59] do not support loops with dynamic iteration counts due to the challenges which will be discussed in the next section. Instead, they support loops with static iterations by simply unrolling the entire loops and tracing the data flow to apply optimizations, which significantly degrades programmability. Furthermore, supporting only static iterations also presents difficulties in scaling to large programs. Whenever the number of iterations changes, the code should be recompiled, which can be a huge burden. Furthermore, compile time tends to increase significantly as the number of iterations grows.

3 Challenges

To address this limitation, HALO aims to provide a loop-aware automatic bootstrapping management compiler for RNS-CKKS. It specifically focuses on supporting loops with static and/or dynamic iteration counts, but without ciphertext-dependent control flows. Handling control flows based on ciphertext comparisons remains an open challenge, as seen in existing CKKS compilers [13, 44–46, 59], since this requires knowledge of the encrypted values.

Enabling automatic bootstrapping management for loops presents two main challenges. Each iteration of a loop executes identical code segments. To meet the constraints on FHE operations with the identical loop body across different iterations, the compiler (or programmers) should consider two kinds of types, *encryption status and level.* The encryption

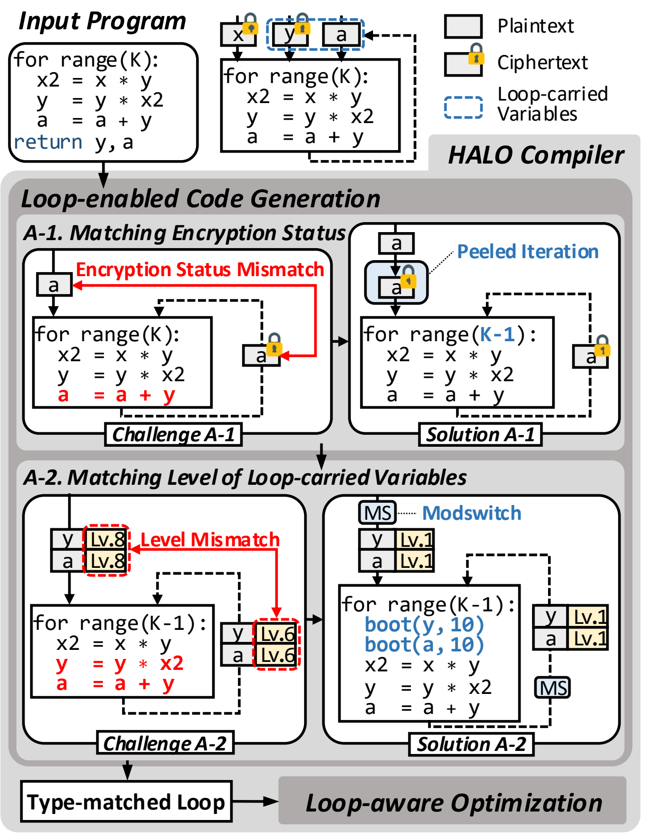


Figure 2. Design of Type-matched code generation of HALO. Dashed arrows represent loop-carried dependencies. The initial levels of `x` and `y` are set to 10.

status consists of `plain` and `cipher`, and the level is represented as an integer. In addition, maintaining an identical loop body restricts the placement of bootstrappings, which is crucial due to the long latency of bootstrapping. We will discuss the type mismatch problems in Section 3.1 and bootstrapping management problem in Section 3.2.

3.1 Type Mismatch on Loop-carried Variables

The challenges illustrated in Figure 2 describe the two cases where type mismatches can occur.

Challenge A-1: The encryption status may mismatch on loop-carried variables. In RNS-CKKS, arithmetic operations involving `plain` and `cipher` (*e.g.,* `multcp`, `addcp`) always result in `cipher`. This can lead to an encryption status mismatch. If an input variable type is initially `plain` but updated as `cipher` due to its use in arithmetic operations, it will be `cipher` in subsequent iterations.

For instance, Figure 2: Challenge A-1 illustrates the occurrence of a loop-carried dependency type mismatch in the variable `a`. In the first iteration, `a` enters the loop as `plain` (represented without a lock). After the first iteration, `a` changes to `cipher` due to addition with a `cipher` operand y (represented with a lock). Thus, a type mismatch occurs between the initial `plain` a and the subsequent `cipher` a at the end of the first iteration.

Challenge A-2: The level type may mismatch on loop-carried variables. Successive multiplications within a loop often necessitate using level management operations such as `rescale` and `modswitch`, which lower ciphertext levels. Consequently, there may arise a level mismatch among loop-carried variables. For example, Figure 2: Challenge A-2 demonstrates the problem of the level type mismatch. Initially, y and a, both at level 8, are the inputs to the loop. After two multiplications within the loop, the level of y is reduced to 6 (*e.g.,* by `rescale`), and the level of a is adjusted to 6 (*e.g.,* by `modswitch`) for addition with y. Subsequently, both y and a, now at level 6, become the inputs for the next iteration.

3.2 Bootstrap-Induced Performance Overhead

In order to keep an identical loop body, the compiler should match the level type of loop-carried variables by inserting bootstrap operations, the most expensive operation, for every iteration. This behavior often degrade overall performance through excessive bootstrapping.

Challenge B-1: Bootstrapping all loop-carried ciphertexts introduces significant performance overhead. Bootstrapping loop-carried ciphertexts is necessary to address the level type mismatch problem when supporting a loop with dynamic interaction count (where unrolling is not feasible). However, bootstrapping them all introduces a significant performance overhead. In the scenario shown in Figure 3: Challenge B-1, the loop has two loop-carried variables, y and a, resulting in two `bootstrap` per iteration. For more complex scenarios, such as Multivariate regression as seen in our benchmark, which may involve nine loop-carried variables, bootstrapping all of them would lead to substantial overhead.

Challenge B-2: A loop may not fully utilize the recovered level. The levels recovered through bootstrapping may not be fully utilized within the loop, especially when the loop body is too short. For example, in the scenario demonstrated in Figure 3: Challenge B-2, the loop body consumes only 4 levels per iteration, leaving 5 levels underutilized. The loop then discards (via `modswitch`) these unused levels and moves on to the next iteration. A more efficient loop structure could allow for better utilization of these excess levels.

Challenge B-3: Recovering the level to its maximum via `bootstrap` may incur unnecessary overhead. As shown in Table 3, the overhead of `bootstrap` increases for higher target levels. When a loop fails to utilize all levels, bootstrapping loop-carried ciphertexts to their maximum levels results in unnecessary overhead. However, DaCapo [13], the state-of-the-art bootstrapping placement compiler, generates code with `bootstrap` that blindly elevates the level to a predetermined maximum target level. The scenario in Figure 2: Challenge B-3 shows bootstrapping applied to the loop-carried variable t to the maximum level unnecessarily.

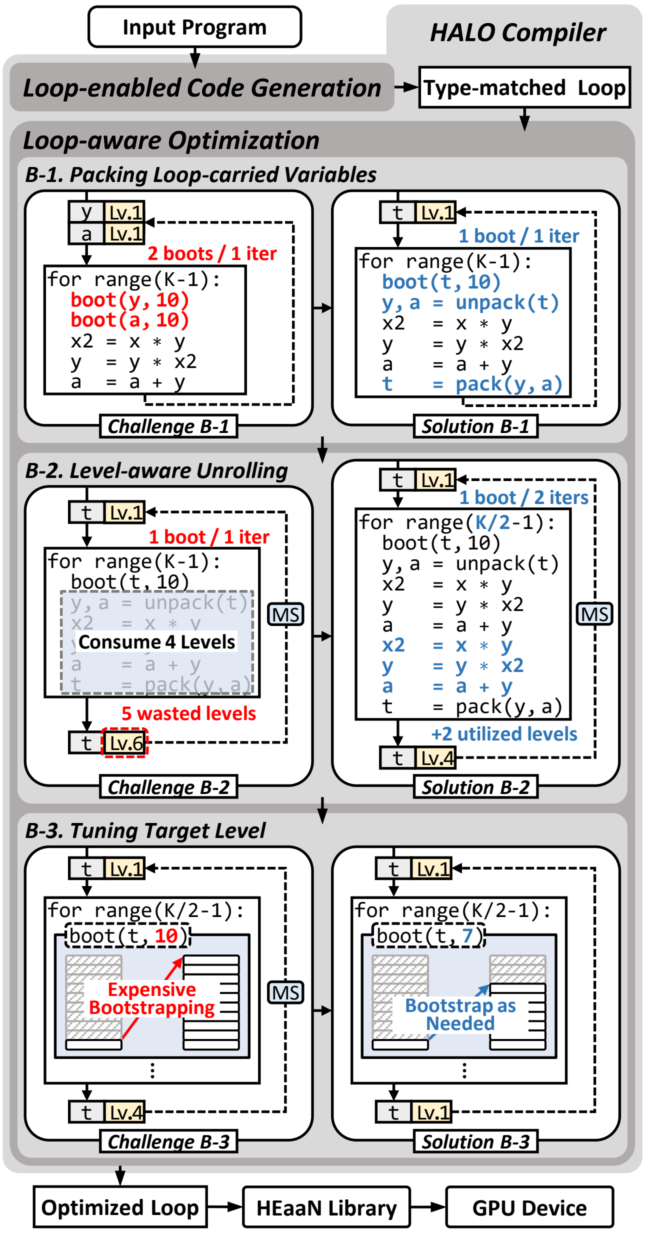


Figure 3. Design of Loop-aware optimization of HALO.

Although bootstrapping restores t to the maximum level 10, the short computation (6 levels consumed) within the loop body does not utilize all the levels. Consequently, the loop executes a `modswitch` on the unused levels before advancing to the next iteration. This inefficiency results in wasted computational resources.

4 HALO Design

In this section, we discuss the design for type-matched loop code generation in Section 4.1, propose optimization methods to improve performance in Section 4.2, and introduce the overall design of the HALO compiler in Section 4.3.

4.1 Type-matched Code Generation

HALO presents two solutions to resolve the type mismatch problems mentioned in Section 3.1.

Solution A-1: Peeling the first iteration of a loop outside the loop. The mismatch in encryption status occurs solely during the initial iteration. Once loop-carried variables transition from `plain` to `cipher`, ciphertexts are never reverted to plaintext without decryption. Therefore, peeling the first iteration can effectively address the encryption status problem.

Figure 2: Solution A-1 displays the program after loop peeling. As the first iteration of the loop is peeled, the number of iterations is reduced from `K` to `K-1`. `a` is transformed from `plain` to `cipher` at the peeled iteration, and stays as `cipher` in the following iterations.

Solution A-2: Bootstrapping all the loop-carried variables at the beginning of each iteration. Regardless of consumed levels of ciphertexts, bootstrapping reinitializes their level. Therefore, inserting the `bootstrap` operation at the beginning of the loop iteration can resolve the level type mismatch problem. Figure 2: Solution A-2 depicts the level matching process of the input and output of the loop body. Before entering the loop body, HALO reduces each level of `y` and `a` to the minimum level using `modswitch` for live-in and loop-carried variables. Then, HALO recovers the levels of the loop inputs to 10 by bootstrapping all loop inputs at the beginning of the loop.

After type matching process, the loop is transformed to a *type-matched loop* (repeatable). However, the type-matched loops are inefficient. For efficient loop execution, bootstrap-based loop restructuring is required.

4.2 Bootstrap-based Loop Restructuring

In loop-aware bootstrapping management, HALO newly proposes three solutions to mitigate the performance degradation caused by bootstrapping.

Solution B-1: Packing loop-carried ciphertexts into a single ciphertext. To reduce the number of the `bootstrap` operations, instead of individually bootstrapping loop-carried ciphertexts, HALO packs them into a single ciphertext and bootstraps it. Since RNS-CKKS encrypts vectors as a single ciphertext, it is possible to pack multiple ciphertexts into one unless its dimension does not exceed the slot number ($\frac{N}{2}$), where N is the polynomial modulus degree. For example, in Figure 3: Challenge B-1, the loop has two loop-carried ciphertexts, `y` and `a`, requiring two bootstrapping at each iteration. As illustrated in Figure 3: Solution B-1, HALO packs `y` and `a` into a single ciphertext (`t`), bootstraps `t`, and unpacks the bootstrapped `t` into the original ciphertexts (`y` and `a`). Consequently, `pack` reduces the number of `bootstrap` from twice per iteration to once per iteration, and also reduces its overheads despite the overheads from `pack` and `unpack` that are much cheaper than `bootstrap`.

Solution B-2: Unrolling the loops reflecting the required level. Loop unrolling reduces its iteration count, thus reducing the number of `bootstrap`. If computation in a loop iteration does not fully consume ciphertext levels, HALO unrolls the loop to fully utilize the levels recovered from `bootstrap`. In Figure 3: Challenge B-2, the loop body including packing and unpacking consumes only four of ten levels. HALO unrolls the loop by two, consuming eight levels per iteration as in Figure 3: Solution B-2. The number of `bootstrap` is reduced to once per two iterations, thereby enhancing the performance.

Solution B-3: Tuning the bootstrapping target level. Since the latency of `bootstrap` decreases as the target level gets lower, as shown in Table 3, recovering the exact required level can reduce the bootstrapping overhead. To adjust the target level, HALO calculates the required levels within the loop body. By following the dataflow chain on a ciphertext, HALO sets the appropriate bootstrapping target level. For instance, in Figure 3: Challenge B-3, the loop body requires 7 levels, and in Figure 3: Solution B-3, HALO changes the bootstrapping target level from 10 to 7. This can prove to be non-trivial considering the fact that the difference in tuning the target level from 10 to 7 amounts to 45,335 microseconds from Table 3, which is comparable to reducing the overhead of about 60 `multcc` operations.

4.3 Overall Design

Figure 2 and Figure 3 together illustrate the overall design of HALO. In detail, HALO takes a program written in Python-based domain-specific language as input and converts it into traced code, which consists of RNS-CKKS operations and the structured *For* operation like structured control-flow dialect (`scf` dialect) in MLIR. The operation identifies loop-carried variables, iteration counts (either constant or variable), and the number of elements (for packing) in the ciphertext explicitly.

The *Loop-enabled Code Generation* module of HALO transforms the traced code into the type-matched loop that is repeatable. The Loop-enabled Code Generation module consists of two type matching processes. First, the module peels the loop to ensure that the encryption status of loop-carried variables are consistent between the first and subsequent iterations. Second, the module bootstraps the loop-carried variables to match the level of inputs and outputs in the loop body. If the loop body or outside of the loop has a long multiplication chain, HALO inserts additional bootstrapping operations at required places.

To optimize the performance of the type-matched loop, the *Loop-aware Optimization* module restructures the loop via three optimizations. First, the module packs loop-carried ciphertexts into a single ciphertext to reduce the number of `bootstrap`. Second, the module unrolls the loop to minimize wasted levels, thus reducing the number of `bootstrap` per iteration. Third, the module tunes the target level of

Algorithm 1: Loop-enabled Code Generation

```
   Input: forOp: Structured For operation (Type-mismatched)
1  Function MatchTypeForLoop (forOp):
2      // Peel the first loop if type mismatch
3      if any_of(forOp.getInitArgs()) is plain then
4          peelFirstIteration(forOp)
5      // Reduce the level of loop inputs and yielded results
6      for input ∈ forOp.getInitArgs() do
7          modswitch (input)
8      end
9      for output ∈ forOp.getBody().getTerminator() do
10         modswitch (output)
11     end
12     // Bootstrap the loop-carried variables
13     for arg ∈ forOp.getBody().getArguments() do
14         bootstrapped ← bootstrap (arg)
15         arg.replaceUsesWith(bootstrapped)
16     end
17 end
```

bootstrap only to the required level, thus reducing the bootstrapping latency.

Through the loop-enabled code generation and the loop-aware optimization, HALO generates an executable, loop-supporting, optimized code. The FHE operations in the generated code are mapped to the GPU-accelerated APIs in the HEaaN library [32].

5 Loop-enabled Code Generation

This section explains how the loop-enabled code generation algorithm (Algorithm 1) peels a loop for encryption type matching (Section 5.1), and aligns levels of ciphertexts (Section 5.2). Section 5.3 describes the method used when additional bootstrapping is required within the loop body.

5.1 Peeling the First Iteration

Line 3–4 in Algorithm 1 shows the peeling process to match the encryption status of loop-carried variables. HALO applies peeling when an input of the loop body has `plain` type. Since arithmetic operations such as `multcp`, `addcp`, `multcc`, and `addcc` always yield a ciphertext, results do not revert to plaintext during program execution.

When a plaintext enters as the input to the loop and the output remains `plain`, it indicates that no FHE operation within the loop updates the plaintext. This means there is no actual dependency between loop-carried variables, or dead code elimination has removed the operations. In other words, if the output is plaintext in the first iteration, the encryption status will never change in subsequent iterations. Therefore, an encryption status mismatch can occur only in the first iteration. Peeling the first iteration ensures that the encryption status of loop-carried variables remain consistent for both loop input and output.

5.2 Inserting Level Management Operations

Line 6–16 in Algorithm 1 describes the level-matching process for loop-carried variables. The outputs of the loop body are the inputs of the next iteration, as loop-carried variables. To match the levels of the outputs with the inputs, HALO inserts `modswitch` before the loop for the inputs, and at the end of the loop for outputs (line 6–11). This ensures that the levels of the loop inputs and outputs are matched to the minimum, creating a level-matched loop. Once the levels of the loop inputs and outputs (loop-carried variables) are matched, HALO bootstraps the loop-carried variables to maximize their levels immediately upon entering the loop (line 13–16). This approach, which bootstraps loop-carried variables at the beginning of the loop body, provides sufficient levels that allow a long computation chain in the loop body.

5.3 Additional Bootstrapping Placement

When the loop body contains many multiplications, additional bootstrapping may be necessary. To handle this case, HALO applies the automatic bootstrapping management proposed in DaCapo [13] to the loop body.

DaCapo fully unrolls loops and automatically places bootstrapping operations. The DaCapo model applies liveness analysis at each program point to identify candidate points that can minimize the amount of bootstrapping operations. Considering all candidates at each program point would significantly increase compilation time, DaCapo filters the candidate bootstrapping insertion points and selects a few candidates. Then, the compiler uses an algorithm that employs dynamic programming to find the best plan within the selected candidates.

When focusing solely on the loop body, if there is no inner loop, the loop can be simplified to a sequence of instructions (equivalent to fully unrolled), allowing the application of DaCapo on a small scope. Then, although using the same approach, HALO can identify bootstrapping targets more quickly. For nested loops, HALO treats the inner loop as a black box, considering the loop itself as a simple operation. Afterward, HALO recursively applies the bootstrapping placement of DaCapo to the inner loop body.

6 Loop-aware Optimization

This section discusses the implementation details of restructuring loops to optimize performance in HALO. Section 6.1 explains the method for packing loop-carried ciphertexts. Section 6.2 discusses how to determine the unroll factor in optimization. Finally, Section 6.3 describes how to set the target level for bootstrapping.

6.1 Packing Loop-carried Variables

The proposed packing method aims to reduce the performance overhead caused by bootstrapping all loop-carried ciphertexts. Packing requires information on the number of elements (size) in the value vector, which programmers specify as the parameter. When a value vector has a size (num_e) no greater than 2^{N-1} (elements in a ciphertext vector), HALO enlarges the vector by repeating its elements until the size reaches 2^{N-1}. Then, the vector is encrypted. Thus, it can be said that the ciphertext is filled with redundant data.

To `pack` the ciphertexts, the valid (non-redundant) data in a ciphertext is extracted using `multcp` with a plaintext that consists of zeroes and ones. For m number of loop-carried variables, HALO initially generates m zero-filled plaintexts. In general, to extract the valid portion of the i-th ciphertext ($0 \leq i < m$), HALO selectively replaces the zeros in the i-th plaintext with ones, specifically, from the $i \times num_e$-th element to the $(i+1) \times num_e$-th element. Then, the i-th ciphertext and the i-th plaintext are multiplied using `multcp` to create a masked ciphertext. As a result, m number of masked ciphertexts are created and combined to a single ciphertext using `addcc`.

`unpack` is inserted after bootstrapping the packed ciphertext. `unpack` also requires `multcp` and `addcc`, similar to packing, but also involves `rotate`. To unpack, HALO generates the same plaintexts used in `pack` consisting of zeros and ones, acting as masks. Multiplying the packed ciphertext with each masking plaintext extracts the i-th ciphertext data from the packed ciphertext. As the extracted ciphertext is filled with zeros for the unmasked portion, the ciphertext should be returned to its original state. This can be achieved through rotation and addition. By using `rotate`, the ciphertext can be cyclically shifted, and then by applying `addcc` to the prior ciphertext and the rotated ciphertext, the empty elements can be filled. Through iterative rotation and addition of the ciphertext, the ciphertext can be restored to its original form. Although the packing method can induce overhead, especially for applications with many loop-carried ciphertexts, the overhead can be justified by improved latency through reduced bootstrapping.

6.2 Level-aware Loop Unrolling

HALO applies the loop unrolling optimization to reduce wasted levels in the loop body and to decrease the number of `bootstrap` per iteration. The first step is to measure the maximum multiplicative depth of the loop body. In the input program shown in Figure 2, HALO calculates the multiplicative depth by following the definition-use chain, starting with the loop-carried variables `y` and `a`. The multiplicative depth of `x2`, which uses `x` and `y` in a multiplication operation, is 1. The depth of `y` becomes 2 as `x2` is the operand of the multiplication. The depth of `a` becomes 2 due to its addition with `y`, which has a depth of 2. Therefore, in this case, the multiplicative depth of the loop will be 2. If loop-carried ciphertexts have different depths, HALO chooses the maximum depth ($depth_{max}$).

The second step is to decide on the unrolling factor. The $depth_{limit}$ is the maximum level usable in the loop body without additional bootstrapping. In our case, $depth_{limit}$ can be set to the maximum level after bootstrapping, L. However, When the `pack` and `unpack` operations are included, the `multcp` from the packing process need to be accounted for, be subtracting 2 ($depth_{limit}$=L-2). Then, once $depth_{limit}$ is fixed, HALO can determine the unrolling factor as $\lfloor depth_{limit}/depth_{max} \rfloor$. HALO does not perform loop unrolling if the unroll factor is 0 or 1. After the unrolling factor is fixed, the loop body is unrolled according to the unrolling factor, reducing the number of iterations. This approach allows efficient use of the levels in the loop body.

6.3 Bootstrapping Target Level Tuning

To set the target level for `bootstrap` operations, HALO tracks the levels wasted in the program. The existing scale management works [22, 44–46] hoist `modswitch` as early as possible, enabling the entire program to operate at lower levels, which is beneficial because lower-level homomorphic operations are faster.

However, the presence of `modswitch` after bootstrapping indicates that levels are not being fully utilized in the subsequent operations. Therefore, HALO traverses the uses of `bootstrap` (definition), subtracting the minimum $downFactor$ (gap between result level and operand level) of the users' `modswitch` from the maximum bootstrapping level ($L - downFactor$) indicating the amount of level used after bootstrapping. Tuning the target level of bootstrapping to $L - downFactor$ reduces the latency of bootstrapping.

7 Evaluation

We implement HALO compiler on top of the MLIR [39] framework, the hardware-side interfaces using the HEaaN library [32] for a GPU backend. The experiments were conducted on an Intel® Core™ i7-12700 CPU and an NVIDIA GeForce RTX A6000 GPU with 48GB memory.

We compare the performance of programs generated by five bootstrapping management compilers as follows:

- **DaCapo**[13] applies the state-of-the-art automatic bootstrapping management after fully unrolling the loops.
- **Type-matched** uses the loop structure that only matches types without any optimization.
- **Packing** applies only packing optimization to the Type-matched loop.
- **Packing+Unrolling** applies additional unrolling optimization to Packing.
- **HALO** applies all optimizations including packing, unrolling, and level tuning to Type-matched loop.

HALO is evaluated on the seven benchmarks represented in Table 4, all of which share the common characteristic of iteratively computing values in loops.

- **Linear Regression (Linear)** is a statistical method that models the linear relationship between variables by fitting a linear equation to observed data.

Table 4. Characteristics of the benchmarks used in HALO. Max and Min RMSE represent each benchmark's maximum and minimum root mean square error (RMSE) value, calculated by comparing the encrypted and non-encrypted results.

Benchmark	Loop Depth	# of Loop-carried Vars.	Approx. Functions	RMSE Max	RMSE Min
Linear	1	2	-	4.14E-6	4.01E-6
Polynomial	1	3	-	5.59E-6	4.91E-6
Multivariate	1	9	-	3.43E-5	1.88E-5
Logistic	1	1	sigmoid	1.52E-4	9.37E-6
K-means	1	2	sign	9.13E-3	5.40E-3
SVM	1	3	sign	4.41E-3	1.59E-4
PCA	2	1, 1	sqrt	1.16E-4	2.16E-6

- **Polynomial Regression (Polynomial)** models a polynomial relationship between the dependent and independent variables, allowing for complex patterns.
- **Multivariate Regression (Multivariate)** captures relationships among sets of independent variables and their response values.
- **Logistic Regression (Logistic)** is a statistical model for predicting the probability of a binary outcome with one or more predictor variables.
- **K-means** is an unsupervised clustering algorithm that divides a dataset into K groups by minimizing the within-cluster sum of squares.
- **Support Vector Machine (SVM)** finds the optimal hyperplane that maximizes the margin between different classes for classification and regression tasks.
- **Principal Component Analysis (PCA)** is a dimensionality reduction technique that simplifies a dataset into a set of orthogonal components.

The seven benchmarks consist of six flat loops (depth 1) and one nested loop (depth 2). The regression benchmarks (Linear, Polynomial, Multivariate, Logistic) exhibit an RMSE of up to 10^{-4}, while the other benchmarks (K-means, SVM, PCA) show larger errors, up to 10^{-3}, due to the approximation of non-linear functions. We implemented non-linear functions based on the algorithm suggested in [41]. Specifically, we approximate the sign function with a minimax composite polynomial using degrees {15, 15, 27} (multiplicative depth of 13) and the sigmoid function utilizing a 96th-order single polynomial (multiplicative depth of 7). On the other hand, the square root (sqrt) function iteratively approximates the sqrt value of the input. Hence, the sqrt function introduces an inner loop within the loop of PCA.

The Linear, Polynomial, and Multivariate regression benchmarks use 4096 randomly generated inputs for each independent variable. For the SVM and K-means benchmarks, randomly generated clusters are used as inputs. The PCA benchmark operates on the iris dataset [26], while the logistic regression benchmark employs the breast cancer dataset [62].

Table 5. Bootstrapping count of benchmarks with varying compilers. The number of bootstrapping operations are based on 40 iterations.

Benchmark	DaCapo	Type-matched	Packing	Packing+ Unrolling	HALO
Linear	18	78	39	20	20
Polynomial	39	117	39	20	20
Multivariate	81	351	39	20	20
Logistic	40	79	79	79	79
K-means	160	200	200	200	200
SVM	273	240	160	160	160

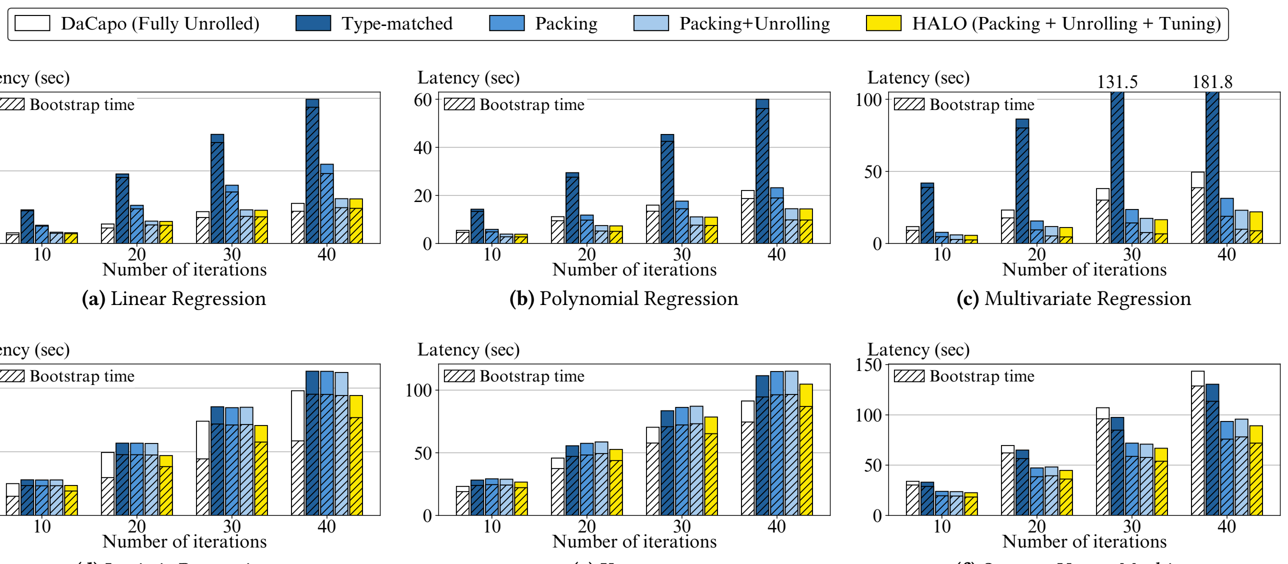
HALO can support deep learning applications such as ResNet like DaCapo [13]. However, since these applications typically involve loops with fixed loop counts that DaCapo supports by fully unrolling them, they may not effectively demonstrate HALO's strengths. Their performance gains are similar to those seen with DaCapo, and this work does not include them in the evaluation.

Table 1 provides the notation, definition, and values of FHE parameters used in the evaluation. The polynomial modulus degree N is 2^{17}, allowing a ciphertext to contain 65536 elements, which half of N. The coefficient modulus is composed of primes with the same scaling factor R_f of 51 bits, and its value is equal to 2^{1479}.

7.1 Latency Comparison

This section analyzes the end-to-end latency of programs generated by five compilers. Figure 4 shows the latency of the code generated by each compiler. HALO overall performs better than DaCapo with one exception in Figure 4e (K-means), which shows a performance overhead of up to 12.4%. DaCapo performs better than HALO because it optimizes the fully unrolled code, which is not required to execute the same code for every iteration, allowing the compiler to adapt the bootstrapping placement to the exact level of consumption. While this approach may work well for some cases like K-means (at the cost of full unrolling), for other applications, DaCapo often results in suboptimal performance (even after full unrolling) due to the challenges in determining bootstrapping points in a scalable manner.

On average, HALO achieves a performance improvement of 27% compared to DaCapo. DaCapo employs fully unrolled code, suffering from an excessive number of bootstrapping candidate points. To manage this, DaCapo filters these points by the number of required bootstrapping operations at that point (the same as the live-in variable at that point) and selects bootstrapping insertion points from the filtered list. However, this heuristic-based filtering approach can miss better solutions, leading to slower performance. Figure 4c (Multivariate) shows the most significant performance improvements, enhancing by up to 2.18×.



(a) Linear Regression **(b)** Polynomial Regression **(c)** Multivariate Regression

(d) Logistic Regression **(e)** K-means **(f)** Support Vector Machine

Figure 4. End-to-end latency comparison of six flat loop benchmarks with five compilers. The part with diagonal lines indicates the time taken during bootstrap in end-to-end processing.

To compare the effects of the three optimizations, we analyze the bootstrapping count from Table 5. First, Packing in HALO is effective when there are two or more loop-carried variables. Comparing the bootstrapping count of Type-matched and Packing shows a reduction in bootstrapping numbers when the benchmark has more than two loop-carried variables. However, in the case of K-means, packing has no effect due to insufficient $depth_{limit}$ within the loop body. As the loop body requires more levels, one additional bootstrapping is inserted, resulting in the same bootstrapping count with a small overhead. The target level tuning can mitigate this additional bootstrapping overhead. Second, the unrolling optimization is particularly effective for regression benchmarks that involve short computations within a loop body. In the case of Linear, Polynomial, and Multivariate, comparing the bootstrapping count of Packing and Packing + Unrolling indicates that bootstrapping has been effectively reduced in these three benchmarks. Third, target level tuning is effective for benchmarks with long computations that require additional bootstrapping within the loop body. Comparing the bootstrapping count of Packing + Unrolling with HALO shows no difference, but Figure 4 reveals a reduction in bootstrapping latency due to target level tuning. The benchmarks (Logistic, K-means, SVM) with long multiplicative depths include multiple bootstrappings within the loop body. This case can result in unnecessary level increases, especially when bootstrapping occurs near the end of the loop body. Target level tuning alone achieved up to a 19% performance improvement in Logistic.

Table 6. Compile time (*sec*) of each benchmark. The numbers below DaCapo (fully unrolled) represent the number of iterations used in the benchmark.

Benchmark	**DaCapo [13]**				**HALO**	**Improvement**
	10	20	30	40		(for iter 40)
Linear	0.13	0.34	0.60	0.94	0.041	23.09x
Polynomial	0.21	0.49	0.91	1.40	0.067	20.79x
Multivariate	13.6	24.6	37.6	54.1	0.301	179.6x
Logistic	5.51	19.9	45.7	80.1	0.360	222.7x
K-means	9.80	38.0	85.3	152	0.270	562.6x
SVM	95.0	314	675	1172	0.151	7743x

7.2 Compilation Time

This section evaluates the compilation time of DaCapo and HALO for six flat loop benchmarks with varying iterations. Table 6 presents the compilation time details for each benchmark in seconds. DaCapo fully unrolls loops for analysis and exhibits a compilation time that increases significantly as the number of iterations rises, scaling up to 15.51× from 10 to 40 iterations for K-means. Due to the lack of loop support for dynamic iterations, DaCapo has to handle an increasingly large amount of code as iterations increase, resulting in a dramatic increase in compile time. The quadratic growth of the compilation time of K-means is due to the exploration-based analysis employed in DaCapo. In contrast, HALO is loop-aware and supports dynamic iterations, leading to a constant compilation time regardless of the number of iterations. HALO achieves a geometric mean reduction in compilation time of 209.12× compared to DaCapo. In the most critical

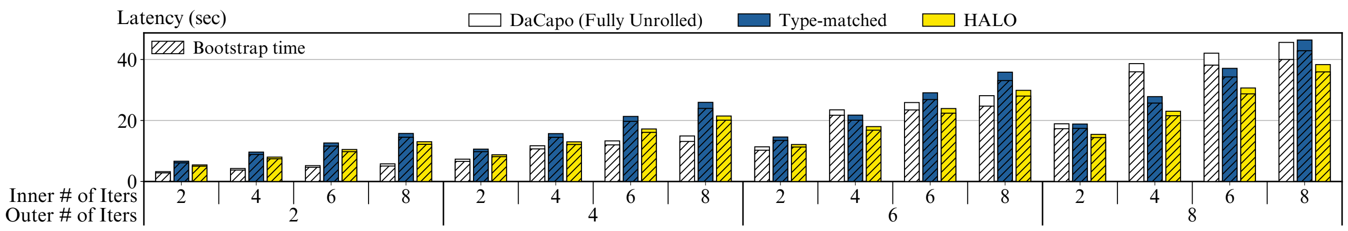


Figure 5. The latency of Principle Component Analysis (PCA) based on the number of loops of inner and outer of iterations.

Table 7. Code size (*KB*) of each benchmark. The numbers below DaCapo (fully unrolled) represent the number of iterations used in the benchmark.

Benchmark	**DaCapo [13]**				**HALO**	**Improvement**
	10	20	30	40		(for iter 40)
Linear	8.37	16.4	24.5	32.3	5.946	5.438x
Polynomial	13.2	25.6	37.8	50.4	9.290	5.429x
Multivariate	36.2	70.9	106	140	32.08	4.376x
Logistic	67.8	131	195	259	17.20	15.04x
K-means	122	242	362	483	15.49	31.17x
SVM	118	235	352	470	16.09	29.19x

case, HALO outperforms the compilation time of DaCapo by 7743×, showcasing the efficiency and scalability of HALO, particularly in scenarios requiring multiple iterations.

7.3 Code Size Comparison

This section evaluates the code size of DaCapo and HALO after the compilation of six flat loop benchmarks with varying iterations. Table 7 summarizes the code size details of each benchmark in KiloBytes (*KB*). The code size includes the constant sizes. Since DaCapo does not support dynamic iterations, the generated code sizes increase up to 3.98× as the number of iterations increases from 10 to 40 for the SVM benchmark. The results show that the code size increases linearly with iterations, which can lead to substantial problems in large programs, as the growing code base can inefficiently consume more memory and storage of the system. On the other hand, HALO supports dynamic iteration, generating constant code sizes regardless of the number of iterations. The improvement of HALO compared to DaCapo is a geometric mean of 11.0× at iteration 40, proving a significant enhancement compared to the state-of-the-art and demonstrating the impact of loop support on FHE applications.

7.4 Case Study: PCA Nested-Loop

We perform a case study on the PCA benchmark, which includes a nested loop, to demonstrate that HALO can be applied to the complex loop structure. PCA consists of a nested loop with a depth of 2, with one loop-carried variable in both the inner and outer loops. Each loop has a long multiplicative

Table 8. Bootstrapping count of PCA benchmark. The numbers below DaCapo (fully unrolled), Type-matched and HALO represent the number of iterations of outer and inner loops in PCA benchmark.

Outer # of Iters	2		4		6		8	
Inner # of Iters	2	8	2	8	2	8	2	8
DaCapo	5	10	13	27	21	52	36	85
Type-matched	12	30	20	50	28	70	36	90
HALO	12	30	20	50	28	70	36	90

depth, so unrolling does not take an effect. Therefore, HALO only performs the target level tuning optimization.

Figure 5 compares the latency of the compiled program based on the number of inner and outer iterations. Because Type-matched and HALO generate the same loop body for all configurations, the latencies of them are proportional to the iteration counts. However, DaCapo does not show consistent results. Although DaCapo fully unrolls the loop for each configuration and gets more optimization chances, DaCapo cannot find the better bootstrapping management for large iteration counts (*e.g.,* (outer iteration, inner iteration) = (8,4)). One notable result is that HALO shows iteration-proportional performance for (8,2) and (8,4), but DaCapo shows significant overhead in (8,4). Because a fully unrolled code exposes an excessive number of bootstrapping candidates, DaCapo filters the candidates and then selects the most efficient solution among them. The filtering of the optimization space in DaCapo filters out the better solution and shows worse performance in cases like (8,4). On the contrary, HALO optimizes the repeated code segment (loop body), not the full trace of the program, so the optimization of HALO can be scaled well with the large iteration count.

Table 8 shows the bootstrapping count corresponding to the number of iterations in the inner and outer loops. When both the inner and outer loop have 8 iterations, HALO reduces the code size by 13.66 × and speeds up the compilation time by 146.75 × compared to DaCapo.

8 Related Work

FHE Compilers: To hide the complexity and increase the efficiency of FHE programs, researches have been done on dedicated libraries [1, 3, 16, 21, 24, 27, 32, 33, 53, 57, 64, 65].

The libraries provide algorithmic optimizations [2, 8, 31, 35, 43, 47, 54] for various FHE operations, enhancing performance. Most of these libraries offer low-level APIs, which give developers the flexibility to optimize FHE programs aggressively but at the cost of significant effort. To address the programming challenges, specialized compilers [4–7, 14, 17–20, 22, 23, 29, 38, 44–46, 49, 55, 58, 59, 61, 63] have been developed, providing higher-level abstractions and optimizations that simplify the development and optimization of FHE programs. Extensive surveys [30, 50, 60] have been conducted in this domain, organizing and detailing the diversities of the libraries and compilers.

EVA [22], Hecate [46], ELASM [44], and Lee *et al.* [45] are compilers that provide scale management, a concept newly introduced in the CKKS scheme. EVA [22] introduces a new FHE language and its optimizing compiler, utilizing a rule-based approach to scale management. Hecate [46] employs an exploration-based approach to identify the best-known scale management plan, achieving speedup. ELASM [44] builds on exploration-based approach by adding noise considerations, resulting in programs with the same latency but lower error. Lee *et al.* [45] adopts a more analytical approach, finding scale management plans similar to Hecate's but in much shorter times. However, Hecate and ELASM are limited to static loops in their implementation. All loops are unrolled before optimizations, which is distinctive from this work.

Porcupine [19], HECO [59], and Coyote [49] focus on efficient data layout to accelerate FHE programs. Porcupine [19] can automatically generate vectorized FHE kernels using comprehensive program synthesis. HECO [59], an end-to-end design for FHE compilers, offers SIMD batching, delivering results similar to Porcupine but at much faster speeds. Coyote [49] optimizes the use of rotate operations to effectively utilize the vector format of ciphertexts. Unlike prior works, which focus on efficient arithmetic computation, HALO's packing method targets loop-carried variables that are candidates of bootstrapping operations, applying target-level bootstrapping to a packed ciphertext. This approach effectively reduces the latency of costly bootstrapping operations, removing the need to bootstrap ciphertexts individually. In addition, Porcupine, HECO, and Coyote are limited to static loops, which they process by unrolling during preprocessing, thus lacking support for dynamic loops.

Automatic Boostrap Management: Bootstrapping is essential for making homomorphic schemes fully homomorphic, allowing unlimited computation on encrypted ciphertexts. However, the bootstrapping operation is very costly, so careful placement and minimizing the number of bootstraps are crucial for the efficiency of FHE programs. Paindavoine and Vialla [52] demonstrates that determining the minimal number of bootstraps and their corresponding locations in the program is an NP-complete problem.

DaCapo [13] is the first compiler to tackle the bootstrapping problem by automatically inserting bootstrap operations at candidate live-out ciphertexts. However, HALO differentiates itself from DaCapo by providing loop support and adaptively using bootstrappings to efficiently match the target level of loop-carried variables with minimal latency. In addition, HALO offers optimizations that unroll loops and fine-tune bootstrap target levels for further speedup.

Privacy Preserving Machine Learning: Research has been conducted on making privacy-preserving machine learning services more usable through novel approaches. Recent studies [34, 36, 37, 40, 42, 51, 56] have focused on Convolutional Neural Network implementations. Some of these studies [36, 51, 56] utilize two-party computation techniques to handle non-linear functions, thereby reducing the overhead of FHE computation. However, when two parties are involved in FHE computation, a trade-off between security and latency emerges, where privacy benefits are often sacrificed to deliver quick results to users.

9 Conclusion

In conclusion, this work introduces HALO, the first loop-aware bootstrapping management compiler designed to optimize bootstrapping placement in FHE programs with loops. HALO ensures correct loop-enabled code generation by matching encryption status and levels between input and output variables of the loop body and reduces bootstrapping overheads through ciphertext packing, loop unrolling, and target level tuning. HALO achieves a 27% performance speedup over the current state-of-the-art, reduces compilation time by 209.12×, and code sizes by 11.0× in geometric mean, demonstrating the contribution of this work.

Acknowledgements

We thank the anonymous reviewers and Dr. Deeksha Dangwal for their insightful comments and shepherding this paper. We also thank the CoreLab members for their support and feedback during this work. This work is supported by No. RS-2024-00358765, No. RS-2020-II201361, No. RS-2022-II220050, No. RS-2023-00277060, and No. RS-2024-00395134 funded by the Ministry of Science and ICT. This work is also supported by Samsung Electronics. Additionally, this work is in part supported by the National Science Foundation (NSF) under grants #2414504, #2214980, #2153747, and #2135157. (Corresponding author: Hanjun Kim)

References

[1] "Lattigo v4," Online: https://github.com/tuneinsight/lattigo, Aug. 2022, ePFL-LDS, Tune Insight SA.

[2] R. Agrawal, J. H. Ahn, F. Bergamaschi, R. Cammarota, J. H. Cheon, F. D. M. de Souza, H. Gong, M. Kang, D. Kim, J. Kim, H. de Lassus, J. H. Park, M. Steiner, and W. Wang, "High-precision rns-ckks on fixed but smaller word-size architectures: theory and application," in *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic*

Cryptography, ser. WAHC '23. Association for Computing Machinery, 2023.

[3] A. Al Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee *et al.*, "Openfhe: Open-source fully homomorphic encryption library," in *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2022, pp. 53–63.

[4] D. W. Archer, J. M. Calderón Trilla, J. Dagit, A. Malozemoff, Y. Polyakov, K. Rohloff, and G. Ryan, "Ramparts: A programmer-friendly system for building homomorphic encryption applications," in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. WAHC'19. Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3338469.3358945

[5] 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."

[6] F. Boemer, A. Costache, R. Cammarota, and C. Wierzynski, "ngraph-he2: A high-throughput framework for neural network inference on encrypted data," in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2019, pp. 45–56.

[7] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski, "ngraph-he: a graph compiler for deep learning on homomorphically encrypted data," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, 2019, pp. 3–13.

[8] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, "Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2021, pp. 587–617.

[9] Z. Brakerski, C. Gentry, and S. Halevi, "Packed Ciphertexts in LWE-Based Homomorphic Encryption," in *Public-Key Cryptography - PKC 2013*, K. Kurosawa and G. Hanaoka, Eds. Springer, 2013.

[10] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ser. ITCS '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 309–325. [Online]. Available: https://doi.org/10.1145/2090236.2090262

[11] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full rns variant of approximate homomorphic encryption," in *Selected Areas in Cryptography – SAC 2018*, C. Cid and M. J. Jacobson Jr., Eds. Springer International Publishing, 2018.

[12] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*. Springer, 2017, pp. 409–437.

[13] 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 *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 6993–7010.

[14] E. Chielle, O. Mazonka, H. Gamil, N. G. Tsoutsos, and M. Maniatakos, "E3: A framework for compiling c++ programs with encrypted operands," Cryptology ePrint Archive, Report 2018/1013, 2018, https://ia.cr/2018/1013.

[15] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: Fast fully homomorphic encryption over the torus," *Journal of Cryptology*, no. 1, pp. 34–91, 2020, https://doi.org/10.1007/s00145-019-09319-x.

[16] ——, "TFHE: Fast fully homomorphic encryption library," August 2016, https://tfhe.github.io/tfhe/.

[17] S. Chowdhary, W. Dai, K. Laine, and O. Saarikivi, "Eva improved: Compiler and extension library for ckks," in *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. WAHC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3474366.3486929

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

[19] M. Cowan, D. Dangwal, A. Alaghi, C. Trippel, V. T. Lee, and B. Reagen, "Porcupine: A synthesizing compiler for vectorized homomorphic encryption," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 375–389.

[20] E. Crockett, C. Peikert, and C. Sharp, "Alchemy: A language and compiler for homomorphic encryption made easy," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1020–1037.

[21] B. Dai, Weiand Sunar, "cuhe: A homomorphic encryption accelerator library," in *Cryptography and Information Security in the Balkans*, E. Pasalic and L. R. Knudsen, Eds. Cham: Springer International Publishing, 2016, pp. 169–186.

[22] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "EVA: An encrypted vector arithmetic language and compiler for efficient homomorphic computation," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2020. [Online]. Available: https://doi.org/10.1145/3385412.3386023

[23] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "CHET: An Optimizing Compiler for Fully-homomorphic Neural-network Inferencing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2019. [Online]. Available: http://doi.acm.org/10.1145/3314221.3314628

[24] L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2015.

[25] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Report 2012/144, 2012, https://eprint.iacr.org/2012/144.

[26] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of eugenics*, 1936.

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

[28] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.

[29] S. Gorantala, R. Springer, S. Purser-Haskell, W. Lam, R. Wilson, A. Ali, E. P. Astor, I. Zukerman, S. Ruth, C. Dibak *et al.*, "A general purpose transpiler for fully homomorphic encryption," *arXiv preprint arXiv:2106.07893*, 2021.

[30] C. Gouert, D. Mouris, and N. Tsoutsos, "Sok: New insights into fully homomorphic encryption libraries via standardized benchmarks," *Proceedings on privacy enhancing technologies*, 2023.

[31] K. Han and D. Ki, "Better bootstrapping for approximate homomorphic encryption," in *Cryptographers' Track at the RSA Conference*. Springer, 2020, pp. 364–390.

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

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

[34] Z. Huang, W.-j. Lu, C. Hong, and J. Ding, "Cheetah: Lean and fast secure {Two-Party} deep neural network inference," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 809–826.

[35] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 114–148, 2021.

[36] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "{GAZELLE}: A low latency framework for secure neural network inference," in

27th {USENIX} Security Symposium ({USENIX} Security 18), 2018, pp. 1651–1669.

[37] D. Kim, J. Park, J. Kim, S. Kim, and J. H. Ahn, "Hyphen: A hybrid packing method and optimizations for homomorphic encryption-based neural networks," *arXiv preprint arXiv:2302.02407*, 2023.

[38] D. Kim, Y. Lee, S. Cheon, H. Choi, J. Lee, H. Youm, D. Lee, and H. Kim, "Privacy set: Privacy-authority-aware compiler for homomorphic encryption on edge-cloud system," *IEEE Internet of Things Journal*, vol. 11, no. 21, pp. 35 167–35 184, 2024.

[39] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 2–14.

[40] E. Lee, J.-W. Lee, J. Lee, Y.-S. Kim, Y. Kim, J.-S. No, and W. Choi, "Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions," in *International Conference on Machine Learning*. PMLR, 2022, pp. 12 403–12 422.

[41] E. Lee, J.-W. Lee, J.-S. No, and Y.-S. Kim, "Minimax approximation of sign function by composite polynomial for homomorphic comparison," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 6, pp. 3711–3727, 2021.

[42] J.-W. Lee, H. Kang, Y. Lee, W. Choi, J. Eom, M. Deryabin, E. Lee, J. Lee, D. Yoo, Y.-S. Kim *et al.*, "Privacy-preserving machine learning with fully homomorphic encryption for deep neural network," *IEEE Access*, 2022.

[43] J.-W. Lee, E. Lee, Y.-S. Kim, and J.-S. No, "Rotation key reduction for client-server systems of deep neural network on fully homomorphic encryption," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2023, pp. 36–68.

[44] Y. Lee, S. Cheon, D. Kim, D. Lee, and H. Kim, "{ELASM}:{Error-Latency-Aware} scale management for fully homomorphic encryption," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4697–4714.

[45] ——, "Performance-aware scale analysis with reserve for homomorphic encryption," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2024, pp. 302–317.

[46] Y. Lee, S. Heo, S. Cheon, S. Jeong, C. Kim, E. Kim, D. Lee, and H. Kim, "HECATE: Performance-Aware Scale Optimization for Homomoprhic Encryption Compiler," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2022.

[47] Y. Lee, J. Seo, Y. Nam, J. Chae, and J. H. Cheon, "Heaan-stat: A privacy-preserving statistical analysis toolkit for large-scale numerical, ordinal, and categorical data," *IEEE Transactions on Dependable and Secure Computing*, 2024.

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

[49] R. Malik, K. Sheth, and M. Kulkarni, "Coyote: A compiler for vectorizing encrypted arithmetic circuits," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 118–133.

[50] C. Marcolla, V. Sucasas, M. Manzano, R. Bassoli, F. H. P. Fitzek, and N. Aaraj, "Survey on fully homomorphic encryption, theory, and applications," *Proceedings of the IEEE*, 2022.

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

[52] M. Paindavoine and B. Vialla, "Minimizing the number of bootstrappings in fully homomorphic encryption," in *Selected Areas in Cryptography–SAC 2015: 22nd International Conference, Sackville, NB, Canada, August 12–14, 2015, Revised Selected Papers 22*. Springer, 2016, pp. 25–43.

[53] "PALISADE Lattice Cryptography Library," https://palisade-crypto.org/, Oct. 2020.

[54] J. Park, D. Kim, J. Kim, S. Kim, W. Jung, J. H. Cheon, and J. H. Ahn, "Toward practical privacy-preserving convolutional neural networks exploiting fully homomorphic encryption," 2023.

[55] S. Park, W. Song, S. Nam, H. Kim, J. Shin, and J. Lee, "HEaaN.MLIR: An optimizing compiler for fast ring-based homomorphic encryption," no. PLDI, 2023.

[56] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow2: Practical 2-party secure inference," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 325–342.

[57] "Microsoft SEAL (release 4.1)," https://github.com/Microsoft/SEAL, Jan. 2023, microsoft Research, Redmond, WA.

[58] T. van Elsloo, G. Patrini, and H. Ivey-Law, "Sealion: a framework for neural network inference on encrypted data," 2019.

[59] A. Viand, P. Jattke, M. Haller, and A. Hithnawi, "HECO: Fully homomorphic encryption compiler," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[60] A. Viand, P. Jattke, and A. Hithnawi, "Sok: Fully homomorphic encryption compilers," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1092–1108.

[61] A. Viand and H. Shafagh, "Marble: Making fully homomorphic encryption accessible to all," in *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. WAHC '18. Association for Computing Machinery, 2018.

[62] M. O. S. N. Wolberg, William and W. Street, "Breast Cancer Wisconsin (Diagnostic)," UCI Machine Learning Repository, 1995, DOI: https://doi.org/10.24432/C5DW2B.

[63] Zama, "Concrete: TFHE Compiler that converts python programs into FHE equivalent," 2022, https://github.com/zama-ai/concrete.

[64] ——, "TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data," 2022, https://github.com/zama-ai/tfhe-rs.

[65] A. Şah Özcan and E. Savaş, "HEonGPU: a GPU-based fully homomorphic encryption library 1.0," Cryptology ePrint Archive, Paper 2024/1543, 2024. [Online]. Available: https://eprint.iacr.org/2024/1543