

Orbit: Optimizing Rescale and Bootstrap Placement with Integer Linear Programming Techniques for Secure Inference

Zikai Zhou*† William Seo*‡ Edward Chen‡ Alex Ozdemir§ Fraser Brown‡ Wenting Zheng‡

†*Tsinghua University* ‡*Carnegie Mellon University* §*Max Planck Institute for Security and Privacy*

zhouzk22@mails.tsinghua.edu.cn, {wys, ejchen, fraserb, wenting}@cmu.edu, alex.ozdemir@mpi-sp.org

Abstract

Fully Homomorphic Encryption (FHE) allows computation on encrypted data without decrypting it. In theory, FHE makes privacy-preserving machine learning possible. In practice, however, it remains impractically slow for real workloads. A major source of slowdown is bootstrap operations; in CKKS, a popular FHE scheme for tensor workloads, the slowdown is compounded by scale management and rescale operations.

FHE compilers for machine learning inference aim to make bootstrap placement and scale management efficient and easy by compiling high-level tensor programs into optimized CKKS computations. Unfortunately, existing approaches miss crucial optimization opportunities because they overlook a key property of CKKS programs: bootstrap and rescale placement are fundamentally coupled through the level budget. In this paper, we present Orbit, an FHE compiler that jointly optimizes bootstrap and rescale placement through a novel Integer Linear Programming (ILP) formulation that reasons about *both* ciphertext level and scale constraints. To make this formulation tractable for structured tensor workloads, particularly convolutional neural networks, we introduce three techniques that reduce ILP complexity while preserving optimality. Across five workloads and multiple cryptographic parameter configurations, Orbit achieves a geometric mean speedup of 19% over DaCapo, 73% over Orion, and 52% over ReSBM, keeps compilation under 6 minutes, and retains model accuracy within 0.3% of plaintext execution.

1 Introduction

Fully Homomorphic Encryption (FHE) allows computation directly on encrypted data. This capability makes it a natural fit for domains where sensitive data must be processed by untrusted cloud infrastructure, including finance, healthcare, and—most prominently—privacy-preserving machine learning. Data remains encrypted throughout the entire cloud-based machine learning computation, with original data and intermediate results only accessible to the data owner via a secret key.

Among the various FHE schemes, CKKS [10] has emerged as the front runner for privacy-preserving machine learning workloads [8, 19, 22, 27, 32, 36, 38, 39, 47]. This is because it supports approximate computation over encrypted fixed-point values with SIMD-style parallelism, which is necessary for secure inference in cloud-hosted ML services. Unfortunately, it is not sufficient: while CKKS supports ML computations, optimizing these computations remains difficult. Each ciphertext carries two key attributes: *level*, which tracks the remaining multiplication budget, and *scale*, which tracks numerical precision. Multiplications increase scale, rescale operations consume levels to control scale growth, and expensive bootstrap operations refresh depleted levels. Efficient encrypted inference therefore depends on determining where to place bootstrap, rescale, and related ciphertext-management operations throughout the computation graph.

We call this optimization problem the *bootstrap placement problem*: given a computational graph and cryptographic parameters, determine where to place bootstrap and ciphertext management operations to minimize execution time while ensuring that no operation exhausts the available level budget or encounters scale mismatches. This problem is challenging because *bootstrap decisions are interdependent*—placing a bootstrap at one location affects the level budget and scale state for all subsequent operations, creating a complex optimization landscape where local decisions have global consequences.

Traditional approaches [30, 47] rely on manual bootstrap placement for specific benchmarks, but this process is tedious, error-prone, and brittle. Manually placing bootstraps can take *days* for a single benchmark and requires repeating the entire manual analysis for different parameter settings or even minor program changes. *Automated approaches* make bootstrap placement easier on the programmer: they compile high-level programs to low-level FHE implementations, automatically managing level and scale along the way. Current FHE compilers fall into two broad classes, each with fundamental limita-

*Equal contribution.

tions. One class [17,28] restricts bootstrap placement to predefined program points such as layer boundaries, which enables fast compilation but misses potential optimization opportunities and ignores bootstrap-rescale interactions. The other class [11, 35] identifies "choke points" where bootstraps are likely needed, but relies on depth- or region-based heuristics that optimize rescale and bootstrap placement sequentially. In reality, these operations are deeply coupled: bootstrap operations restore levels but affect the available budget for rescales, while rescale placement consumes levels and influences bootstrap frequency. Our key insight is that *efficient placement for encrypted tensor workloads requires the compiler to jointly reason about bootstrap placement, rescale placement, and the regular structure of the underlying computation graph.*

In this paper, we introduce Orbit, a compiler optimization framework for structured encrypted tensor programs, with a focus on CNN workloads. Our approach formulates the problem as an optimization problem encoded as an Integer Linear Program (ILP) [21] that jointly reasons about ciphertext level and scale. This formulation gives Orbit fine-grained control over ciphertext-management decisions, allowing it to consider placements that are difficult for coarse or sequential heuristics to find.

A naive global ILP, however, does not scale to end-to-end encrypted inference graphs. To overcome the scalability limitations of ILP, we introduce several new techniques that *exploit structural properties of tensor workloads*. First, we observe that these workloads contain repetitive computational patterns—such as the multiply-and-accumulate structure in convolutions and matrix multiplication—that create redundancies in the ILP formulation. We develop an algorithm that compresses such patterns into single ILP nodes, which dramatically reduces the size of the ILP problem without affecting solution optimality.

Even after compression, large programs require further decomposition to maintain tractable ILP solving times. We develop an approach that decomposes the computation graph into single-input single-output (SISO) regions, independently optimizes each partition, and combines the local ILP solutions into a global one. Our key insight is that by partitioning along SISO boundaries, we can limit the number of cross-partition variables and avoid additional ciphertext management operations. This lets Orbit efficiently "stitch" together locally optimal solutions into a global placement. These techniques are most effective on structured tensor workloads with repeated subgraphs and clean SISO boundaries, as in convolutional neural network inference; we discuss generality and limitations in Section 8.

We implement and evaluate Orbit on five convolutional neural networks across diverse FHE parameter configurations. Our techniques yield compilation times under 6 minutes for all benchmarks. Through our novel ILP formulation that jointly optimizes bootstrap placement and rescale operations, Orbit achieves geometric mean speedups of 7-19% over DaCapo, 2-73% over Orion (on configurations where Orion compiles), and up to 52% over ReSBM. The key to these improvements is our ILP formulation, which reduces rescale operations by up to 5.4× while achieving competitive or lower bootstrap counts (e.g., 31-34% reduction compared to Orion) and maintaining model accuracy within 0.3% of plaintext execution.

Our core contributions are as follows:

- We introduce a **level-scale aware ILP formulation** that jointly optimizes bootstrap and rescale placement, addressing the limitation of prior work which treats these deeply-coupled operations independently.
- We develop a **compression algorithm** that identifies and merges repetitive computational patterns, dramatically reducing the number of ILP variables while provably preserving solution optimality.
- We introduce partitioning and **Quadratic Behavior Profiles (QBPs)** for structured SISO-oriented tensor graphs. QBPs are a caching mechanism that precomputes local solutions for each partition, allowing Orbit to efficiently "stitch" local solutions into a global solution.
- We present **iterative partition merging with boundary scale optimization** that achieves practical QBP table sizes while maintaining near-optimal solution quality through dynamic scale selection.
- Our open-source implementation is available at https://github.com/cmu-cryptosystems/Orbit and our archived artifact can be found at https://zenodo.org/records/20278435.

2 Background

In this section, we provide background on the CKKS fully homomorphic encryption (FHE) scheme—including its encoding, level and scale management, and associated constraints. We also provide a primer on integer linear programming (ILP), which is the core technique we use to solve the bootstrap placement problem.

2.1 Fully Homomorphic Encryption (FHE)

The most popular and efficient FHE schemes today are lattice-based schemes like BGV/BFV [6, 7, 18], and CKKS [10]. In this paper, we focus on CKKS [10], which supports encrypted computing on floating-point numbers and is widely used for approximate computing workloads like machine learning inference. Table 1 details our notation used to describe CKKS attributes.

Computational Operations CKKS supports three fundamental homomorphic operations that enable computation on encrypted data: addition (`add`), multiplication (`mul`), and rotation (`rot`). Since CKKS efficiently batches a vector of plaintext values into polynomials, additions and multiplications are single instruction, multiple data (SIMD) operations that operate element-wise across all slots. Ciphertext rotation is the only intra-ciphertext movement operation available in CKKS. Any ciphertext permutation must be implemented using combinations of rotations and masking with 0-1 bitmasks.

CKKS Encoding and Encryption CKKS plaintexts and ciphertexts are represented as polynomials in the ring $R_Q = \mathbb{Z}_Q[X]/(X^N+1)$. The ring dimension N (typically 2^{14} to 2^{17}) determines both the polynomial degree and the number of SIMD slots ($n = N/2$) available for batching. Each polynomial coefficient is an integer modulo Q, which is decomposed via the Residue Number System (RNS) into L smaller primes: $Q = q_1 \times q_2 \times \cdots \times q_L$. To encrypt a plaintext vector $\vec{x}$ of real/complex numbers, we first encode it into a plaintext polynomial **m**, then encrypt this polynomial into a ciphertext **ct**.

CKKS ciphertexts have two key attributes for managing precision and noise: **Scale** (s) is a scaling factor that enables encoding real numbers as integer polynomial coefficients while preserving precision. A minimum waterline scale S_{min} is required throughout computation to prevent precision loss from accumulated rounding errors. In our notation, s is represented as a logarithmic scaling factor, i.e., $scale = 2^{40}$ is represented as $s = 40$. **Level** (ℓ) tracks the number of remaining RNS moduli, indicating the multiplication budget. Fresh ciphertexts start at the maximum level L; multiplication operations consume levels until bootstrap restores them to L_{bts}, where $L_{bts} < L$ due to the cost of bootstrapping itself. The maximum scale capacity between bootstraps is $S_{max} = S_f^{L_{bts}}$, where S_f is the rescaling factor (typically $S_f = q_i$ for each prime modulus).

Notation	Definition	Used in Orbit
N	Ring Dimension	2^{15}, 2^{17}
Q	Ciphertext modulus	2^{1488}, 2^{1692}
L	Maximum Levels	27, 31
L_{bts}	Effective levels for computation	12, 16
S_f	Rescaling factor	2^{51}
S_{min}	Minimum (waterline) scale	2^{40}, 2^{51}
S_{max}	Maximum scale	$2 \cdot S_{min} + S_f$

Table 1: CKKS Ciphertext Management Parameters.

Ciphertext Management Operations CKKS provides four operations to manage level and scale (see Table 2): `rescale` divides scale by S_f and reduces level by 1; `modswitch` reduces level while preserving scale; `upscale` increases scale via multiplication while preserving level; and `bootstrap` resets level to L_{bts} and scale to S_f. Bootstrap is needed when level becomes too low to support further computation.

Operation	Constraints	Level	Scale
Computational Operations			
Addition	$\ell_i = \ell_j, s_i = s_j$	$\ell_k = \ell_i$	$s_k = s_i$
Multiplication	$\ell_i = \ell_j, \ell_i \geq 1$	$\ell_k = \ell_i$	$s_k = s_i \times s_j$
Rotation	$\ell_i > 1$	$\ell_k = \ell_i$	$s_k = s_i$
Ciphertext Management Operations			
Rescale	$\ell_i > 1, S_k \geq S_{min}$	$\ell_k = \ell_i - 1$	$s_k = s_i / S_f$
ModSwitch	$\ell_i > 1$	$\ell_k = \ell_i - 1$	$s_k = s_i$
Upscale	$\ell_i \geq 1$	$\ell_k = \ell_i$	$s_k > s_i$
Bootstrap	$\ell_i \geq 1$	$\ell_k = L_{bts}$	$s_k = S_f$

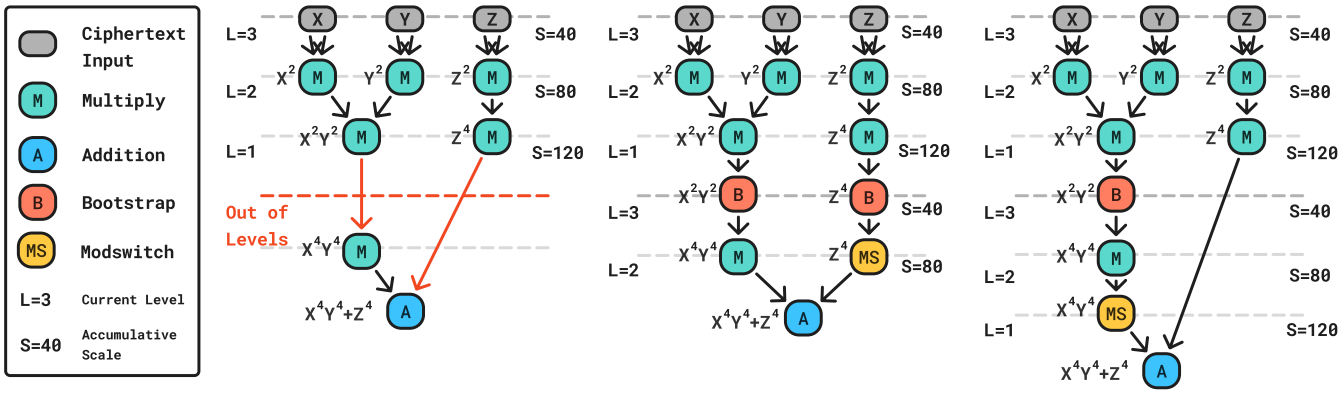
Table 2: Level and Scale constraints and effects on CKKS operations. ℓ and s denote the ciphertext level and scale attributes. Subscripts i and j denote operand ciphertexts, and k denotes the resulting ciphertext.

Cost of CKKS operations The cost of computational operations depends on the level. Operations at higher levels are more expensive because they involve more RNS moduli. Bootstraps dominate the overall cost, being 2-3 orders of magnitude more expensive than any other ciphertext operation. However, minimizing bootstrap count alone does not guarantee optimal performance. For example, inserting an additional bootstrap to reduce ciphertext levels can decrease the cost of subsequent operations enough to improve overall execution time. We detail our cost-modeling approach in Appendix B.

2.2 Integer linear programming (ILP)

Integer linear programming (ILP) is an optimization framework for linear objectives and constraints, where decision variables take integer values. An ILP problem consists of: *decision variables*, the unknown values to be determined, representing specific levels assigned to ciphertexts or binary decisions on whether to insert bootstrap operations; *constraints*, a set of linear inequalities or equalities that define the feasible region, enforcing the "rules" of the system such as ensuring an operation's level does not exceed the available level budget; and an *objective function*, a linear expression to be maximized or minimized, such as the overall latency.

Modern solvers such as Gurobi [21], CPLEX [26], and SCIP [23] leverage advanced techniques—including branch-and-bound, cutting planes, and presolving—to efficiently solve large-scale practical instances. ILP has been successfully applied to diverse domains including scheduling [41], resource allocation [14], network design [40], and compiler optimization [45].



(a) Polynomial Approximation **(b)** Bootstrap Placement with depth heuristics **(c)** Bootstrap Placement with Orbit

Figure 1: Motivating example: bootstrap placement for polynomial approximation $f(x,y,z) = x^4y^4 + z^4$ with L_{bts}=3, S_{min}=2^{40}, S_f=2^{40}. (a) Naive execution exhausts the level budget. (b) Prior systems (DaCapo, ReSBM) require two bootstraps using depth-based heuristics. (c) Orbit discovers an optimal solution using only one bootstrap through fine-grained ILP optimization.

3 Overview

Orbit is an FHE compiler framework designed to provide fine-grained, cost-aware placement for ciphertext management operations. In this section, we first present a motivating example that highlights the limitations of prior works, then show Orbit's end-to-end compiler optimization pipeline.

3.1 Motivation: Bootstrap Placement is Hard

Given an FHE program and a set of cryptographic parameters, the bootstrap placement problem asks: *where should we insert bootstrap and other ciphertext management operations to produce a valid, efficient FHE program?* A valid placement must satisfy the constraints defined in Table 2.

To illustrate these challenges, consider a simple polynomial approximation function $f(x,y,z) = x^4y^4 + z^4$, shown in Figure 1. In this example, we configure the cryptographic parameters as follows: $L_{bts} = 3$, $S_{min} = 2^{40}$, $S_f = 2^{40}$, and $S_{max} = 2^{120}$. By setting the waterline scale and scale factor to identical values (2^{40}), we can simplify the bootstrap placement problem by assuming a rescale after each multiplication. This configuration keeps all operations at scale 2^{40}, thus we only need to consider changes in levels.

Without proper ciphertext management operations, this program would produce undecipherable results (see Figure 1a). First, the level budget is exhausted after computing x^2y^2 and z^4, and further computation without bootstrap operations risks corrupting the plaintext data. Second, x^4y^4 and z^4 are computed in parallel but end up at different levels, resulting in a level misalignment that prevents the addition $x^4y^4 + z^4$. To resolve this problem, someone—either the program author or the compiler—must place bootstrap operations to prevent level exhaustion and validate that all computations have their levels aligned. A naive approach places bootstrap operations immediately once the level budget is exhausted, while proactively using modswitch operations to align levels for subsequent operations. This approach comes at the cost of additional bootstrap operations, a steep performance overhead.

Prior works—DaCapo [11], Orion [17], and ReSBM [35]—either compile this example suboptimally or cannot compile it at all. To reduce the search space, both DaCapo and ReSBM use heuristics that partition vertices along their multiplicative-depth (as denoted by the dashed lines). Instead of assessing each of the ten vertices, this multiplicative-depth partitioning heuristic reduces the bootstrap candidate set to the four possible multiplication levels, thereby reducing the search space. In this example, both systems insert bootstraps after x^2y^2 and z^4 (Figure 1b). Unfortunately, this solution comes at the cost of performance as it uses two bootstraps—later we will show that only one is required. Orion cannot generate a valid placement for this example as the polynomial exceeds the multiplicative depth budget, and Orion is not designed to insert bootstraps within a layer.

Orbit models the input program as an ILP formulation, which optimizes bootstrap placement with fine-grained precision. This approach lets Orbit consider the entire input computation IR as a global optimization problem rather than relying on predefined boundaries to place bootstraps. Figure 1c shows how Orbit discovers an optimal solution that only uses a *single* bootstrap operation: by performing an early `modswitch` after x^2y^2, the system satisfies the level constraints for both x^4y^4 and z^4 while minimizing computational overhead.

3.2 Orbit's pipeline

Orbit is an FHE compilation framework that aims to produce efficient FHE programs by optimizing rescale and bootstrap placement (Figure 2). Orbit builds on top of DaCapo and thus inherits its frontend and intermediate representation (IR). Or-

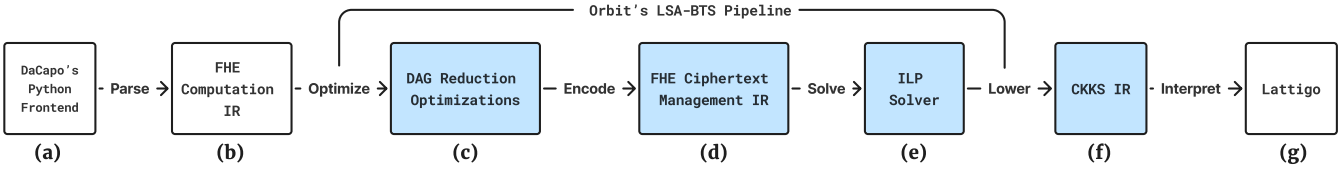


Figure 2: Orbit's compilation pipeline. Blue marks our system contributions.

bit uses DaCapo to parse programs written in a Python tensor DSL (Figure 2a), which supports tensor operators (e.g., convolutions) and polynomial approximations (e.g., ReLU) for implementing end-to-end machine learning inference models. This yields an FHE Computation IR (Figure 2b), a directed acyclic graph (DAG) of *solely* computational operations (`add`, `mul`, `rot`); this representation excludes ciphertext management operations (`rescale`, `modswitch`, `bootstrap`). Orbit's optimizations operate on the FHE Computation IR, which makes it easy to integrate as an optimization pass in other FHE compiler frameworks (e.g., HEIR [4] and Rotom [9]).

To find an efficient placement, Orbit encodes the FHE Computation IR into an FHE Ciphertext Management IR (Figure 2d). This IR represents a novel ILP formulation that jointly optimizes the level and scale of each computational operation. The ILP formulation introduces integer decision variables for each operation's level and scale, along with constraints that enforce FHE semantics (detailed in Table 2). The objective function minimizes execution time using a profiled cost model that captures the level-dependent performance characteristics of the FHE operations (Appendix B). Orbit uses Gurobi [21] to solve this optimization problem, and the result dictates the locations to insert ciphertext management operations, e.g., rescale operations (to manage scale capacity) and bootstrap operations (to refresh levels) while minimizing the overall estimated performance cost.

Directly encoding the entire input DAG is prohibitively expensive, since ILP solving time grows exponentially with the number of constraints. For example, the naive ILP encoding of the entire AlexNet benchmark, with $\approx 10^5$ total constraints, did not finish solving in over 24 hours. To improve ILP scalability, Orbit introduces three novel DAG reduction optimizations and heuristics (Figure 2c); these optimization passes are Orbit's key contributions (see Section 5). While these heuristics marginally affect search optimality, they come with significant compile time improvements, e.g., AlexNet-SiLU, which previously did not finish solving in 24 hours, finished in under a minute.

After solving for an efficient placement, Orbit lowers the modified FHE Computation IR, with all ciphertext management operations inserted, into a CKKS IR (Figure 2e). The CKKS IR serves as a backend-agnostic abstraction layer, allowing Orbit to target different FHE libraries (e.g., SEAL, Lattigo, OpenFHE) [1, 5, 42]. This design makes Orbit's optimizations portable across the FHE ecosystem. In our implementation, Orbit interprets the CKKS IR on Lattigo [1].

4 Orbit's ILP Formulation

This section presents our approach to the bootstrap placement problem. Prior works [11, 17, 35] solve this problem in two sequential phases: first placing rescale operations to manage scale constraints, then inserting bootstrap operations based on the resulting level consumption. In contrast, Orbit jointly solves for both rescale and bootstrap placement through a novel ILP formulation. Our key insight is that rescale and bootstrap placement are fundamentally intertwined. Rescales consume levels and trigger the need for bootstraps, while bootstraps reset levels and alter the optimal rescale strategy. This mutual dependency requires joint optimization rather than sequential decision-making. We formalize this unified view as the *level-scale-aware bootstrap placement* (LSA-BTS) problem, which simultaneously optimizes both rescale and bootstrap placements.

We begin by defining the FHE Computation IR and how Orbit lowers it into the FHE Ciphertext Management IR, introduce our ILP formulation, and conclude by explaining how the FHE Ciphertext Management IR is lowered into the CKKS IR. Appendix C provides a detailed summary of our assumptions that help reduce the ILP complexity.

4.1 Input: FHE Computation IR

The LSA-BTS pass operates on Orbit's FHE Computation IR, which represents a program as a DAG $G = (V, E)$. Vertices V represent computational operations. A directed edge e from vertex u to vertex v (i.e., $e = (u, v) \in E$) represents a data dependency between two vertices. Figure 3a illustrates a simple FHE Computation IR, G, with a single edge between two vertices. Note that G initially contains *no* ciphertext management operations (e.g., `bootstrap`) and *thus does not represent a valid FHE execution*. Next, we discuss how Orbit transforms G to a G' that includes such operations.

4.2 Lowering FHE Computation IR to FHE Ciphertext Management IR

As shown in Figure 3, Orbit lowers a program represented in FHE Computation IR (G) to one represented in FHE Ciphertext Management IR (G'). The goal of the G' is to model the

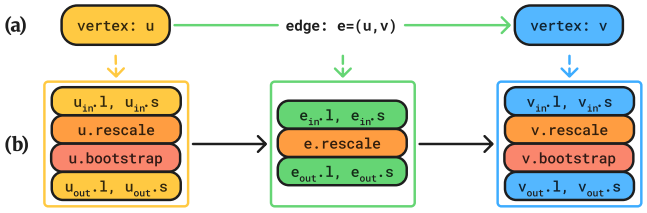


Figure 3: Lowering FHE Computation IR in **(a)** to FHE Ciphertext Management IR in **(b)**.

ILP formulation by (1) defining level and scale constraints and (2) creating candidate locations to insert ciphertext management operations in G.

In DAG $G' = (V', E')$, each vertex $v \in V$ is expanded into a vertex $v' \in V'$ that tracks computational and ciphertext management operations. As shown in Figure 3b, each v' maintains input/output level and scale attributes, i.e., (v'_{in}.l, v'_{in}.s) and (v'_{out}.l, v'_{out}.s), an integer variable v'.rescale indicating the number of rescale operations applied, and a binary variable v'.bootstrap indicating whether bootstrap was applied on v'. Similarly, each edge $e' \in E'$ maintains input/output level and scale attributes, i.e., (e'_{in}.l, e'_{in}.s) and (e'_{out}.l, e'_{out}.s), and an integer variable for rescales (e'.rescale). Edges require this rescale attribute because rescales can be selectively placed on specific data flow paths, allowing for more selective rescale placement. This DAG structure models the level and scale values that *could possibly* be assigned to each vertex and edge and all candidate locations in which rescales and bootstraps could be placed. It does not imply that Orbit actually places a bootstrap or rescale in every possible location.

4.3 Lowering Ciphertext Management IR into ILP Constraints

Orbit lowers G' into ILP constraints that model the FHE program. There are four classes of constraints: bound, computational, ciphertext management, and vertex-edge constraints. A full table of the constraints are detailed in Appendix A.

Bound constraints ensure their level and scale attributes are within valid bounds. For all vertices and edges in V' and E', the levels must be within the range $[1, L_{bts}]$, and all scales must be within the range $[S_{min}, S_{max}]$.

Computational constraints define level and scale constraints for `add`, `mul`, and `rot` operations. For each of these operations in G', Orbit generates operator-specific constraints. For example, consider a vertex v with two incoming edges $e0$ and $e1$ in G'. If v's operator is `add`, Orbit generates constraints that match both levels and scales across all operands, i.e., (v_{in}.l $= e0_{\text{out}}$.l $= e1_{\text{out}}$.l) and (v_{in}.s $= e0_{\text{out}}$.s $= e1_{\text{out}}$.s). On the other hand, if v's operator is `mul`, Orbit generates the constraints that match the operand levels but sums the operand scales[1], i.e., (v_{in}.l $= e0_{\text{out}}$.l $= e1_{\text{out}}$.l) and (v_{in}.s $= e0_{\text{out}}$.s $+ e1_{\text{out}}$.s).

Ciphertext management constraints update a vertex's or edge's level and scale with respect to `rescale` and `bootstrap` operations. Each rescale operation decrements the level by 1 and divides the scale by S_f. Since the ILP solver may place multiple rescale operations on a single rescale vertex, we account for their cumulative effect. Given either a vertex or edge x with k rescale operations placed, the constraints are: (x_{in}.l $\geq x_{\text{out}}$.l $+ k$) and (x_{in}.s $\leq x_{\text{out}}$.s $+ S_f \cdot k$).

Each `bootstrap` operation refreshes a vertex's level back to L_{bts} and scale to S_f. Since the ILP formulation contains conditional constraints based on whether a bootstrap is placed on vertex v, we apply the big-M method [20] to convert them to linear constraints: (v_{in}.s $\leq S_f \cdot v_{\text{in}}$.l $+ M_1 \cdot (1 - v$.bootstrap$)$); (v_{out}.l $\geq 2 - M_2 \cdot (1 - v$.bootstrap$)$); ($v_{\text{out}}$.s $\geq S_f - M_3 \cdot (1 - v$.bootstrap$)$), where M_1, M_2, M_3 are sufficiently large constants to relax the constraints when v.bootstrap $= 0$.[2]

Vertex-edge constraints ensure consistency between vertex outputs and edge inputs. For all edges $e = (u, v) \in E'$, we have ($e_{in}.l = u_{\text{out}}$.l) and ($e_{in}.s = u_{\text{out}}$.s).

4.4 Objective Function

The optimization objective is to minimize the total compute cost of all operations. We construct a cost model by profiling homomorphic operations at different levels (see Appendix B). The cost includes the operation cost at the input level, the bootstrap cost if applied, and the rescale cost proportional to the number of rescales. The objective function is as follows:

$$\begin{aligned}\min \quad & \sum_{v' \in V'} \Big[\text{cost}(v'.\text{op}, v_{in}.\text{l}) \\ & \quad + v'.\text{bootstrap} \cdot \text{cost}(\texttt{bootstrap}) \\ & \quad + v'.\text{rescale} \cdot \text{cost}(\texttt{rescale}) \Big] \\ & + \sum_{e' \in E'} e'.\text{rescale} \cdot \text{cost}(\texttt{rescale})\end{aligned}$$

4.5 Lowering ILP Solutions to CKKS IR

The ILP solver determines optimal level and scale pairs for each vertex and edge in G' and where to insert bootstrap and rescale operations. To generate a valid, executable CKKS IR from these decisions, Orbit needs to determine the specific sequence of ciphertext management operations needed to achieve each level and scale transition. Since `modswitch` and `upscale` operations are inexpensive (together accounting for less than 0.1% of total execution time), Orbit does not directly model their costs in the ILP formulation, and instead inserts them as needed to satisfy intermediate constraints. We defer the details to Appendix G.

[1] Note that s is represented in log-scale.

[2] In our implementation, we use Gurobi's *General Constraint Indicator* to encode these constraints efficiently, rather than relying on the *big-M* method.

5 Improving ILP Solver Scalability

The previous section presents our ILP formulation for rescale and bootstrap placement. Unfortunately, naively applying the ILP formulation to an entire FHE Computation IR is computationally expensive, as ILP solving time scales exponentially with program size.

In the following sections, we introduce three techniques that scale the ILP formulation to large, real-world benchmarks with little impact on search quality. These techniques apply to any input FHE Computation IR and aim to minimize the size and complexity of each ILP solver call. The first technique is DAG Compression, which compresses parallelizable operations to reduce the number of ILP constraints while provably preserving optimality. The second technique is partitioning, which divides the circuit into smaller sub-DAGs that can individually be solved within seconds. However, combining optimal local solutions from partitioning can yield a suboptimal global solution. To address this, we introduce the Quadratic Behavior Profile (QBP), a caching mechanism that enables efficient merging of local solutions into a global solution. In some cases, this approach provably finds the optimal solution. However, the number of entries in the QBP tables scales poorly with the level budget and scale range, which limits scalability. Our last technique mitigates this through iterative merging, which heuristically selects a set of high-performing scale choices and avoids additional cross-partition ciphertext management operations.

5.1 DAG Compression

Many tensor operations, such as convolution, contain repetitive and parallelizable operations. Since these structures are so predictable, we can compress the parallel operations into a single operation. This compressed form requires far fewer constraints to encode in our ILP formulation.

Orbit's compression algorithm operates by grouping vertices into equivalence classes based on their local topologies. To compute these classes, we employ *partition refinement* as our underlying algorithmic framework. Within this framework, we define *n-depth similarity* as the specific refinement criterion: intuitively, two vertices are considered *n-depth similar* if they are *identical* in operations when you examine their local neighborhood up to depth n in both directions (parents and children). We formalize this criterion as follows:

Definition 5.1 (n-Depth Similarity / Refinement Criterion)**.** For a non-negative integer n, v_A and v_B are n-depth similar ($v_A \approx_n v_B$) as follows:
Base Case: If $v_A = v_B$, then $v_A \approx_n v_B$ for any $n \geq 0$.
Recursive Case: Otherwise, $v_A \approx_n v_B$ if and only if all of the following hold:
• *Same operation:* $v_A.\mathsf{op} = v_B.\mathsf{op}$;
• *Same parent structure:* both $v_A.\mathsf{parents}$ and $v_B.\mathsf{parents}$ are non-empty and have the same sequence of operations up to n vertices from v, with a bijection f from $v_A.\mathsf{parents}$ to $v_B.\mathsf{parents}$ such that $p \approx_n f(p)$ for every p in $v_A.\mathsf{parents}$;
• *Same child structure* (when $n > 0$): both $v_A.\mathsf{children}$ and $v_B.\mathsf{children}$ are non-empty and have the same sequence of operations up to $n-1$-depth vertices from v, with a bijection g from $v_A.\mathsf{children}$ to $v_B.\mathsf{children}$ such that $c \approx_{n-1} g(c)$ for every c in $v_A.\mathsf{children}$.

This recursive definition naturally leads to a standard partition refinement algorithm. We initialize the graph by grouping vertices by their operation types, and then iteratively refine these equivalence classes using the n-depth similarity criterion as n increments ($n = 0, 1, 2, \ldots$). This refinement process terminates when the partition converges and no further splitting occurs.

This partition refinement establishes a *localized* equivalence relation rather than strict global graph isomorphism. Because a vertex's ILP constraints depend solely on the multiset of its neighborhood states, vertices within the same final *localized* equivalence class are guaranteed to generate isomorphic constraints. Consequently, their optimal bootstrap placements are identical. This allows Orbit to safely compress each equivalence class into a single representative vertex without sacrificing global search optimality.

In the worst case, this procedure takes $O(|V|)$ iterations; however, in practice, we find the highly parallel and structured nature of the FHE Computation IR leads to fast convergence—within 10 seconds in our experiments (see compile times in Table 3).

To support compression in our ILP formulation, we introduce a new vertex attribute $v.\mathsf{cpr}$ (or $e.\mathsf{cpr}$ for edges), which records how many *n-depth similar* vertices and edges have been compressed together. The default value $cpr = 1$, since vertices and edges begin uncompressed in the FHE Computation IR. This compression factor is used in the objective function to scale the cost of each compressed vertex or edge, accounting for the fact that a single representative vertex/edge now represents multiple identical operations. The updated objective function including compression is

$$\min \quad \sum_{v \in V'} \left[\mathbf{v.cpr} \cdot \mathrm{cost}(v.\mathsf{op}, v_{\text{in}}.\mathsf{l}) + \cdots \right] + \sum_{e \in E'} \mathbf{e.cpr} \cdot e.\mathsf{rescale} \cdot \mathrm{cost}(\texttt{rescale})$$

Theorem 5.1 (Compression Preserves Search Optimality)**.** Let $G = (V, E)$ be a FHE Computation IR and $\tilde{G} = (\tilde{V}, \tilde{E})$ be its compressed version obtained by merging equivalent vertices under partition refinement (Definition 5.1). Given an optimal solution $\mathrm{sol}(\tilde{G})$ on the compressed graph, a solution $\mathrm{sol}(G)$ is obtained by assigning all vertices in each equivalence class the same decision as their merged representative in $\tilde{G}$. This expanded solution is optimal for G.

Proof Sketch. The proof relies on two foundations within our partition refinement framework. First, vertices with n-depth

similarity act as roots for *semi-equivalent* sub-DAGs. Their isomorphic local topologies allow us to safely substitute partial placements. Second, a *reconstruction mapping* guarantees that any valid compressed placement sol$\left(\tilde{G}\right)$ expands into a full placement sol(G) while strictly preserving topological validity and total operational cost (σ). This mapping dispatches the theorem's validity claim and bridges the cost relations between the compressed and original domains.

To prove optimality, we proceed by contradiction. Suppose a strictly better, potentially asymmetric placement sol$(G)^*$ exists for the original graph ($\sigma(\text{sol}(G)^*) < \sigma(\text{sol}(G))$). Our goal is to construct a strictly better placement for the compressed graph $\tilde{G}$, contradicting the optimality of sol$\left(\tilde{G}\right)$. We first "symmetrize" sol$(G)^*$: if vertices within the same equivalence class receive different assignments, we induce their corresponding semi-equivalent sub-DAGs, select the minimum-cost placement, and overwrite the others. This yields a globally valid, *symmetric* placement sol$(G)^{**}$ with monotonically non-increasing cost ($\sigma(\text{sol}(G)^{**}) \leq \sigma(\text{sol}(G)^*)$). By our second foundation, this symmetric sol$(G)^{**}$ compresses back into a valid solution for $\tilde{G}$ with a strictly lower cost than sol$\left(\tilde{G}\right)$, contradicting its assumed optimality.

We defer the formal proof to Appendix E.

5.2 Partitioning and Behavior Profiles

While Orbit's DAG compression aims to reduce the number of constraints for linear tensor operators, compression alone fails to address all of the ILP scalability challenges. For example, the compressed DAG for AlexNet-ReLU has ≈4,300 vertices, which would take days to solve. To improve solve times, we design a way to partition a large DAG into smaller sub-DAGs, independently solve each sub-DAG, and then combine the local solution into a global solution. The key challenges are 1) determining where to partition the DAG to achieve efficient ILP solve times and 2) avoiding expensive cross-partition ciphertext management when merging sub-DAG solutions.

We address both challenges by partitioning along single-input single-output (SISO) boundaries, a natural choice for two reasons. First, SISO boundaries provide functional modularity—both polynomial activation functions and compressed linear regions naturally segment along these boundaries. Solving a SISO partition lets Orbit reuse the local solutions across structurally identical sub-DAGs, saving on ILP solving time. Second, partitioning at SISO boundaries minimizes the number of boundary configurations that must be reconciled. Each SISO boundary has exactly one input and one output, so the space of possible level and scale assignments at the boundary is $O(L_{bts}^2 S^2)$. On the other hand, partitioning at boundaries with k input and output nodes would instead require enumerating $O(L_{bts}^k S^k)$ configurations, which grows exponentially in k.

After partitioning, we can run the ILP solver multiple times to enumerate *all* potential per-partition solutions, one for each combination of input level/scale and output level/scale. To cache all solutions, we introduce the Quadratic Behavior Profile (QBP), a lookup table that stores all of the optimal LSA-BTS placements for a single SISO partition. Entries in the QBP table are indexed by (input level, input scale, output level, output scale). The total number of entries is quadratic in the number of possible level and scale combinations. Formally, we define the QBP as follows:

Definition 5.2 (Quadratic Behavior Profile)**.** Let $D = (V, E)$ be a SISO partition with one input vertex v_{in} and one output vertex v_{out}. Let S be the difference in the maximum and minimum scale, i.e., $S_{max} - S_{min} + 1$. The QBP of D is a table QBP_D containing up to $\mathrm{O}(L_{bts}^2 S^2)$ entries, where $\mathsf{QBP}_D(l_{\text{in}}, s_{\text{in}}, l_{\text{out}}, s_{\text{out}})$ gives the optimal ILP solution for D when $v_{\text{in}}.\mathsf{l}{=}l_{\text{in}}$, $v_{\text{in}}.\mathsf{s}{=}s_{\text{in}}$, $v_{\text{out}}.\mathsf{l}{=}l_{\text{out}}$, $v_{\text{out}}.\mathsf{s}{=}s_{\text{out}}$.

Orbit merges the per-partition QBP solutions into a single global solution using dynamic programming. Suppose G is composed of m sequential SISO sub-DAGs $D_1, \cdots, D_m$. We define $\text{dp}(i, l_{\text{out}}, s_{\text{out}})$ as the minimum cost for the first i sub-DAGs with the i-th sub-DAG's output level l_{out} and output scale s_{out}. The base case initializes all possible starting states to zero cost: for all valid levels $1 \leq l \leq L_{bts}$ and scales $S_{min} \leq s \leq S_{max}$, we set $\text{dp}(0, l, s) = 0$. The recurrence relation combines sub-DAG costs optimally: $\text{dp}(i, l_{\text{out}}, s_{\text{out}})$ equals the minimum over all possible input levels l_{in} and scales s_{in} of the sum $\text{dp}(i-1, l_{\text{in}}, s_{\text{in}}) + \mathsf{QBP}_{D_i}(l_{\text{in}}, s_{\text{in}}, l_{\text{out}}, s_{\text{out}})$. The global solution is obtained by taking the minimum of $\text{dp}(m, l_{\text{out}}, s_{\text{out}})$ over all possible output levels l_{out} and scales s_{out}.

Theorem 5.2 (Complete QBPs Preserve Search Optimality)**.** Let G be a DAG decomposed into sequential SISO sub-DAGs $D_1, \ldots, D_m$, and let QBP_{D_i} denote a *complete* QBP table for each sub-DAG D_i. Optimal solutions obtained via dynamic programming over the complete QBP tables correspond to optimal solutions for the original DAG G.

Proof Sketch. The proof relies on the properties of composing and decomposing placements across SISO boundaries. *Placement composition* merges valid SISO sub-placements into a globally valid placement with strictly additive operational cost (σ). Conversely, *placement decomposition* splits any valid global placement at a SISO boundary into valid sub-placements while perfectly preserving this cost additivity.

We proceed by induction on the number of partitions. Assume the dynamic programming (DP) correctly computes the optimal cost for the first $i-1$ partitions. Suppose, for contradiction, a strictly better global placement sol$(G_i)^*$ exists for the first i partitions ($\sigma(\text{sol}(G_i)^*) < \text{dp}(i, l_{\text{out}}, s_{\text{out}})$). Decomposing sol$(G_i)^*$ at the i-th boundary yields two valid sub-placements meeting at a specific boundary state $(l_{\text{in}}^*, s_{\text{in}}^*)$. By the inductive hypothesis and QBP optimality, their respective costs are lower-bounded by $\text{dp}(i-1, l_{\text{in}}^*, s_{\text{in}}^*)$ and the corresponding QBP entry $\mathsf{QBP}_{D_i}(l_{\text{in}}^*, s_{\text{in}}^*, l_{\text{out}}, s_{\text{out}})$. Summing these bounds via cost additivity demonstrates that the DP

could achieve a cost $\leq \sigma(\text{sol}(G_i)^*)$ simply by selecting this specific state. Since the DP recurrence explicitly minimizes over *all* valid boundary states, it mathematically guarantees a final cost $\text{dp}(i, l_{\text{out}}, s_{\text{out}}) \leq \sigma(\text{sol}(G_i)^*)$, directly contradicting the assumed optimality of $\text{sol}(G_i)^*$.

We defer the formal proof to Appendix F.

In practice, solving for complete QBPs is tractable when $S_f = S_{min}$, a setting used by some FHE applications and systems [17, 35]. Since every rescale operation brings the scale down to S_{min}, there is effectively only one scale value throughout the computation. This reduces the QBP table from $O(L_{bts}^2 S^2)$ entries to just $O(L_{bts}^2)$ entries, since $S = 1$.

In this setting, all operations are restricted to S_{min}, which means we cannot exploit the fact that some operations could safely run at lower scales and consume fewer levels. As a result, this setting precludes some optimizations.

5.2.1 Bypass Edges

Bypass edges (e.g., residual or skip connections) are long-range dependencies that span multiple tensor operations. This challenge appears in our ResNet benchmark, where residual connections span multiple layers, creating dependencies that cross many SISO boundaries. Without explicit handling for bypass edges, our partitioning algorithm would result in large partitions that negatively impact ILP solving time.

To address this, we extend our partitioning technique to first isolate bypass paths to break up oversized partitions, as shown in Figure 4. By isolating bypass edges, Orbit creates forking points (1-in-2-out) and merging points (2-in-1-out) along the partition boundaries, denoted by the purple partitions. For these partitions, we solve for all possible solutions along their partition boundaries, which requires $O(L_{bts}^3 S^3)$ ILP instances. This enumeration remains practical compared to solving a single large monolithic ILP for a large partition (see Table 5).

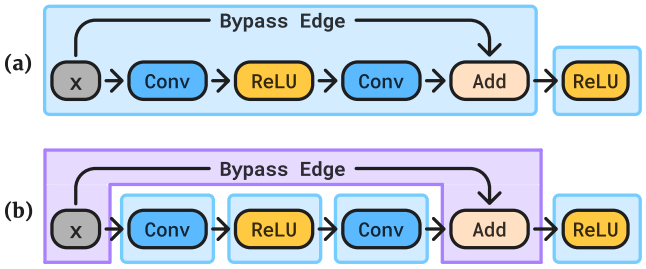


Figure 4: **(a)** SISO partitioning without bypass handling creates large partitions. **(b)** Explicit bypass handling isolates skip connections, enabling more partitioning along the main path.

5.3 Handling Cross-partition Scale Alignment

While partitioning and QBPs enable efficient compilation by reducing the size of the ILP instances, the scale dimension remains a practical bottleneck. Even with modest scale ranges, the $O(L_{bts}^2 S^2)$ boundary configurations can become prohibitive. To ensure practical compile times, we devise an iterative technique to minimize the scale range that must be explored across partition boundaries.

Existing FHE systems configure the relationship between the rescaling factor (S_f) and waterline scale (S_{min}) in different ways. Some systems [17, 35] restrict $S_f = S_{min}$, forcing all operations to operate at the waterline scale and simplifying scale management at the cost of performance. More flexible systems [11] place no restrictions on the relationship between S_f and S_{min}, creating a harder scale management problem but enabling better performance through finer-grained rescale placement. Orbit focuses on the latter case, introducing an iterative approach to efficiently explore the scale space when S_f and S_{min} are decoupled.

Relaxing the $S_f = S_{min}$ constraint introduces two key challenges. First, the QBP table size explodes: with typical parameters ($L_{bts} = 16$, $S = 92$), tracking all input/output level and scale combinations requires $2,166,784$ entries for a SISO partition. Second, scale alignment across partition boundaries is no longer automatic. Different boundary scale choices affect both partition compatibility and performance, where mismatched scales require extra rescale or bootstrap operations. Together, these challenges make QBP construction and partition stitching more complex than the $S_f = S_{min}$ case.

To address these challenges, we propose an approach that *iteratively* solves each partition and dynamically finds efficient and compatible boundary scales. Our solution exploits a key observation: *lower output scales* preserve "headroom" for subsequent partitions by providing more flexibility in their input scale choices, thereby conserving levels and delaying when to insert additional bootstraps. For example, if P_i outputs at scale S_{min} rather than $2 \cdot S_{min}$, the subsequent partition P_{i+1} can execute more operations before its scale budget is exhausted and a rescale is required. For each input/output level pair, multiple optimal solutions exist along a Pareto frontier trading off local partition cost against output scale. By accepting slightly higher local costs to achieve lower output scales, we can improve global performance. We use this insight by modifying our objective function to incorporate boundary scale awareness: $f' = f + \alpha \cdot V \cdot \text{cost}(\textit{rescale}) \cdot \textit{out}_{\textit{scale}}$, where α is a hyperparameter controlling a scale penalty and V is number of vertices in the partition. Empirically, we find that setting $\alpha = 0.2$ provides an effective balance between local optimality and boundary scale minimization.

To find a global solution among all local partitions, we solve each partition and their QBP table iteratively in topological order. For input partitions, we initialize input scales to the minimum valid scales (either S_{min} or a compatible scale for addition) that satisfy the circuit's input constraints. For subsequent partitions P_i, we use the output scales from the preceding partition P_{i-1} as fixed input scales for P_i. This allows us to explore only the relevant portion of the scale

space for each partition, selecting the output scale for each input level that balances local cost with downstream flexibility. The resulting QBP tables are compact since we cache only the single best output scale discovered for each level pair during this iterative process. This approach dramatically reduces table size from the naive $O(L_{bts}^2 S^2) = 2.1M$ entries to just $O(L_{bts}^2) = 256$ entries per partition, while ensuring scale compatibility across partition boundaries.

6 Implementation

We implement Orbit in approximately 2,700 lines of Python and 2,200 lines of Golang code. The Python component handles MLIR parsing, DAG reduction optimizations, ILP formulation via the Gurobi optimizer [21], and code generation, while the Golang component implements the cost modeling and runtime execution in Lattigo. Orbit extends DaCapo's frontend interface [11] and parses input programs into a MLIR-based FHE Computation IR. Our implementation consists of three main stages: (1) ILP formulation to determine bootstrap and rescale placements (see Section 4), (2) DAG reduction optimizations to reduce ILP problem size (see Section 5), and (3) a Lattigo interpreter that executes the FHE program following the optimized placement.

7 Evaluation

This section presents our evaluation of Orbit. First we describe our evaluation setup, followed by detailed microbenchmarks to evaluate our ILP scalability techniques. We evaluate Orbit against three state-of-the-art FHE systems that support bootstrap placement: DaCapo [11], Orion [17], and ReSBM [35]. To perform a controlled experiment, we keep the cryptographic parameters consistent on all benchmarks and run all systems using our Lattigo backend. Note that Lattigo does not support a variable bootstrap implementation, which uses cheaper bootstraps to a desired target level rather than always bootstrapping to the maximum level. Thus, we compare against ReSBM using an estimated cost model.

We exclude HALO [12] and HEILP [46] from our baseline comparison. HALO targets inter-iteration bootstrap placement across dynamic for-loops and is orthogonal to Orbit, which optimizes over a global static computation graph; the two are complementary rather than competing, and Orbit could in principle replace HALO's intra-loop solver. HEILP shares an ILP foundation but is infeasible on our benchmarks: its dense formulation requires $O(L^2)$ variables per vertex, causing state explosion on large graphs, whereas Orbit requires only 7–10 variables per vertex and incorporates DAG reduction techniques that HEILP lacks.

7.1 Evaluation Setup

All benchmarks were compiled on AWS EC2 *m8i.8xlarge* instances (32 cores, 128 GB RAM) [3] and evaluated on *m7g.16xlarge* instances (64 cores, 256 GB RAM). We evaluate Orbit on 5 convolutional neural networks used to evaluate prior FHE compilers: ResNet-20 [30], SqueezeNet [25], AlexNet [29], MobileNet [24], and VGG16 [43]. These benchmarks are implemented in DaCapo's frontend and lowered into Orbit's FHE Computation IR, which serves as the standardized input representation for all evaluated bootstrap placement algorithms. To enable fair comparison with prior work, we re-implemented the bootstrap placement algorithms from Orion [17] and ReSBM [35] to operate on this common IR. This is because Orion's implementation compiles their custom Python benchmarks directly to Lattigo without an exposed IR, and ReSBM's algorithm is not publicly available in their released compiler implementation.

We evaluate two variants of each benchmark, distinguished by their activation and pooling functions. The two activation functions used are SiLU (S) and ReLU (R). SiLU is approximated using a single Chebyshev polynomial; ReLU is approximated using a larger composite polynomial consisting of three Chebyshev polynomials [31]. Only AlexNet, SqueezeNet, and VGG16 support pooling functions, where the ReLU variants use max-pooling and the SiLU variants use average-pooling.

7.2 Micro-benchmarks

To evaluate our heuristic-based techniques for improving ILP scalability, we provide microbenchmarks to evaluate our DAG reduction techniques: compression, partitioning, QBP reuse, and bypass handling.

Compression & Partitioning We conduct an ablation study using a 4-layer ResNet-20 with SiLU activations to compare our compression and partitioning techniques. Each layer in the benchmark contains a mix of convolution operations (parallel linear layers) and activation functions (polynomial approximations). We compare four configurations: (1) baseline ILP with no optimizations, (2) compression only, (3) partitioning only, and (4) both partitioning and compression applied together.

As shown in Table 3, the baseline ILP requires over 10 minutes to solve the ILP formulation and scales poorly to larger benchmarks. Compression alone achieves a compilation time of 205 seconds by reducing the number of vertices, and thus ILP variables, by 2.6× in the linear operations. Applying partitioning alone reduces the compilation time down to 108 seconds by breaking the problem into 27 independent subproblems of manageable size. The combined approach

[3] Orbit is highly memory efficient, requiring under 5 GB in total for compilation. The high-memory instances for compilation were selected to leverage high core counts for parallelizing QBP solving.

Config	# Parts	Max	Avg	Comp	Exec
No Opts	1	4822	4822	652.78 s	829.07 s
Compression	1	1840	1840	205.35 s	830.92 s
Partitioning	27	1091	180	108.28 s	851.46 s
Both	18	179	103	**33.12 s**	838.26 s

Table 3: Impact of compression and partitioning on 4-layers of ResNet-20-S. **Max/Avg** represents the max/average partition size. **Comp/Exec** represents the compile/execution time.

achieves the best performance with a solve time of 33 seconds (highlighted in green), representing a 19.7× speedup over our no-optimization baseline. Importantly, all configurations produce placements with nearly identical end-to-end latency (within 1% of each other), confirming that our scalability optimizations preserve solution quality while improving solver efficiency.

Cross-benchmark QBP Reuse Table 4 evaluates the impact of QBP-reuse across sequential compilations. The compilation order is AlexNet, SqueezeNet, VGG16, ResNet-20, starting with the ReLU variant followed by the SiLU variant and $N = 2^{15}$. This order was determined alphabetically aside from ResNet-20, which has many partitions that cannot be reused in other benchmarks.

QBP reuse provides compilation speedups ranging from 1% to 73%, with a geometric mean of 23%. The overall speedup of compiling the entire benchmark suite is 26%. AlexNet shows minimal speedup because it is compiled first with no prior QBPs available to reuse, and the 3-5 second overhead of loading and saving QBPs offsets any gains from reusing QBPs. These results demonstrate that caching QBPs across multiple compilation runs can greatly reduce the ILP solving time for polynomial approximation functions.

Benchmark	w/o Reuse	w/ Reuse	Speedup	Reuse %
AlexNet-R	342.7	345.2	0.99×	52.8
AlexNet-S	75.0	74.6	1.01×	42.9
SqNet-R	345.8	200.4	1.73×	67.2
SqNet-S	74.9	68.9	1.09×	58.3
VGG16-R	180.1	126.2	1.43×	74.0
VGG16-S	87.3	58.9	1.48×	77.5
ResNet-20-R	55.3	41.7	1.33×	82.0
ResNet-20-S	49.7	46.1	1.08×	82.2

Table 4: Impact of QBP reuse on compilation time (s). Reuse% includes both intra-benchmark and cross-benchmark QBP reuse.

Bypass Handling To evaluate our bypass handling technique, we conduct an ablation study on ResNet-20, which contains numerous skip connections spanning convolution and activation layers. We compare baseline partitioning (which keeps entire residual blocks within single SISO partitions) against Orbit's bypass handling (which separates bypass edges and applies a specialized 2-in-1-out ILP formulation for merging partitions). Table 5 shows that bypass handling decomposes the network into smaller partitions (average 95 vs. 176 vertices), reducing total compilation time by $5.2-8.6\times$ while incurring negligible runtime overhead (under 0.25%). These results demonstrate that explicitly modeling bypass edges is essential for handling modern neural architectures with complex control flow patterns.

Config	Bypass	# Parts	Avg Size	Comp	Exec
R, $N = 2^{15}$	N	21	165	285.97 s	687.98 s
	Y	39	89	**55.33 s**	688.79 s
S, $N = 2^{15}$	N	21	183	166.15 s	431.98 s
	Y	39	98	**49.74 s**	430.97 s
R, $N = 2^{17}$	N	21	169	482.67 s	3147.07 s
	Y	39	91	**56.36 s**	3141.18 s
S, $N = 2^{17}$	N	21	187	351.95 s	1868.76 s
	Y	39	100	**54.33 s**	1871.19 s

Table 5: Impact of bypass handling on partitions

7.3 Macro-benchmarks

Across 5 convolutional neural networks, we compare Orbit against DaCapo, Orion, and ReSBM using multiple CKKS parameter settings. The base configuration is ($N = 2^{17}$, $S_{min} = 40$, $L_{bts} = 16$); the default used in DaCapo. Like prior FHE compilers, Orbit takes the cryptographic parameters, including the available level budget L_{bts}, as input. In practice, developers can estimate an initial level budget using simple static analysis, such as the multiplicative depth of the layers in a machine learning inference program, and iteratively refine the parameters based on security and performance requirements. From the base configuration, we test 2 additional changes while keeping the rest of the parameters the same: reducing the number of slots ($N = 2^{15}$) and using a smaller level budget ($L_{bts} = 12$)[4]. These additional configurations evaluate Orbit's robustness to parameter variation. Our evaluation demonstrates Orbit consistently completes compilation within minutes across varying L_{bts} settings, whereas DaCapo experiences significant compile-time bottlenecks and Orion fails on several constrained-budget configurations.

[4]The $N = 2^{15}$ configuration does not meet the 128-bit security standards and is not intended for deployment.

7.3.1 Comparison to DaCapo

Compilation: Table 6 compares compilation times between Orbit and DaCapo. Orbit compiles all benchmarks across all configurations in under 6 minutes, with most completing in under 1 minute. This shows that Orbit's approach is scalable to end-to-end benchmarks. When reducing the ring dimension to $N = 2^{15}$, DaCapo's compilation time dramatically increases from minutes to hours on various benchmarks. Intuitively, this slowdown occurs because DaCapo's bootstrap placement algorithm depends on their liveness analysis, which computes the working set size of ciphertexts at each program point. Decreasing the ring dimension increases the ciphertext working set size at each program point, which in turn increases their analysis complexity. For example, in SqueezeNet-R, the number of bootstrap candidates grew from 175 to 5,722, increasing compile time by 435×. For both AlexNet benchmarks, DaCapo fails to compile within 24 hours.

Two additional outliers occur when compiling the benchmarks. First, both Orbit and DaCapo fail to compile MobileNet at $N = 2^{15}$ due to a bug in DaCapo's frontend. At a high-level, the frontend does not correctly determine which input values are marked as ciphertexts and fails on an internal check. Second, DaCapo's bootstrap placement algorithm times out on AlexNet-S when using a smaller level budget ($L_{bts} = 12$). This is attributed to an increase in the number of bootstrap candidates, similar to the effects of lowering the ring dimension.

Evaluation: Figure 5 presents the evaluation time comparison to DaCapo. Each benchmark is presented with its operation cost breakdown; each test is averaged over 3 runs. The bars represent bootstrap cost (blue), rotation cost (orange), rescale cost (green), and all other operation costs (red).

Across the base configuration, Orbit improves on DaCapo by a geometric mean of 7%. Orbit's performance improvements stem from its joint bootstrap and rescale optimization approach, which enables fine-grained placement decisions that reduce overall latency. There are two notable benchmarks to highlight: SqueezeNet-S and AlexNet-R. On SqueezeNet-S, Orbit improves on DaCapo by 8% while using *one additional bootstrap*. Even though bootstraps are a very expensive operation, Orbit finds a placement that trades off this additional bootstrap for lower computation cost (as denoted by the smaller orange bar) by running rotations at lower levels. Orbit performs the best on AlexNet-R, improving by 14%. Orbit reduces the number of bootstraps from 71 to 62 and rescales from 12,437 to 2,304. This 5.4× reduction in rescales demonstrates how joint optimization of bootstrap and rescale placement enables more efficient scale management than the depth-based heuristics approach used by DaCapo.

Across Different Configurations: On a smaller level budget ($L_{bts} = 12$), Orbit improves on DaCapo by a geometric mean of 19%. With this parameter configuration, the SiLU activation function does not fit nicely between two bootstrap operations. DaCapo's depth-based bootstrap placement approach fails to find efficient bootstrap candidates on irregular graphs. This bootstrap reduction is exemplified in AlexNet-R (116 →100), MobileNet-S (118→67), ResNet-S (85→47), and VGG16-S (70→43), where the number of bootstraps is significantly lower in Orbit's placement.

On a smaller ring dimension ($N = 2^{15}$), Orbit improves on DaCapo by a geometric mean of 9%. The performance improvements are comparable to the base configuration because changing the ring dimension does not affect rescale or bootstrap placement. Rather, it primarily affects the ciphertext working set size and hence compilation time. While Orbit's compression and partitioning heuristics maintain fast compilation times, DaCapo times out on AlexNet due to the increased complexity of its ciphertext liveness analysis.

Benchmark	Base		N=2^{15}		L_{bts}=12	
	D	**O**	**D**	**O**	**D**	**O**
AlexNet-S	356	54	t-o	75	t-o	78
ResNet-20-S	22	54	49	50	335	47
SqueezeNet-S	52	141	27952	75	1912	134
VGG-16-S	191	64	31132	87	11242	63
MobileNet-S	248	65	err	err	16494	72
AlexNet-R	874	251	t-o	343	755	232
ResNet-20-R	22	56	47	55	19	44
SqueezeNet-R	85	200	36992	346	86	177
VGG-16-R	231	107	48904	180	230	87
MobileNet-R	240	67	err	err	231	50

Table 6: Compilation time in seconds separated by activation function. **D** stands for DaCapo, and **O** stands for Orbit. t-o indicates the benchmark timed-out after 24 hours, and err indicates compilation errors.

7.3.2 Comparison to Orion

Compilation: We evaluate Orbit against Orion across multiple cryptographic parameter configurations. Recall that Orion restricts $S_f = S_{min}$, which simplifies the bootstrap placement problem and enables fast compilation times when successful. Table 7 shows that when Orion successfully compiles, it often achieves compilation times within seconds. However, Orion's simplifying assumptions lead to frequent compilation failure when a partition's multiplication depth exceeds the level budget L_{bts}. This particularly affects benchmarks using max-pooling operations (e.g., AlexNet-R, SqueezeNet, and VGG16-R), which require deeper computation than average-pooling alternatives. Under constrained parameter settings, Orion's compilation failures become more prevalent, while Orbit successfully compiles under all configurations.

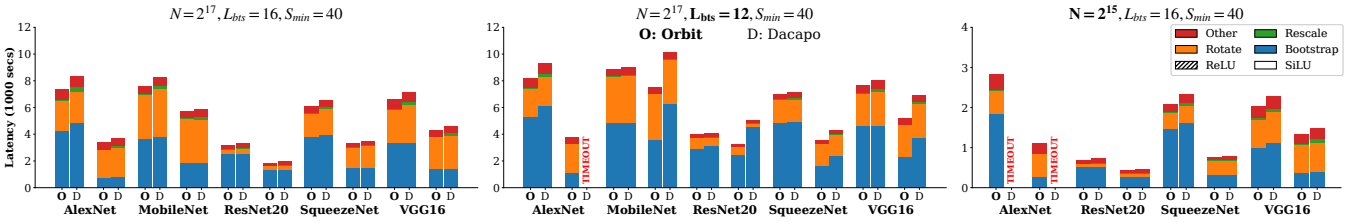


Figure 5: Evaluation comparison between Orbit and DaCapo.

System	AlexNet R	AlexNet S	MobileNet R	MobileNet S	ResNet-20 R	ResNet-20 S	SqueezeNet R	SqueezeNet S	VGG16 R	VGG16 S
Same Waterline Scale: $S_{min}=51$										
Orbit	93	29	47	47	34	32	102	74	63	41
Orion	err	3	3	3	1	1	err	1	err	3
Smaller Level Budget: $L_{bts}=12$										
Orbit	232	78	50	72	44	47	177	134	87	63
Orion	err	err	2	err	1	err	err	err	err	err
Smaller Slots: $N=2^{15}$										
Orbit	343	75	-	-	55	50	346	75	180	87
Orion	err	8	-	-	1	1	err	err	err	7

Table 7: Compilation time (seconds) comparison across multiple configurations. R=ReLU, S=SiLU activation. err indicates compilation error.

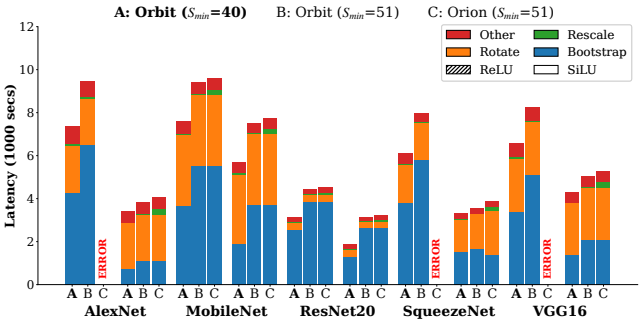


Figure 6: Evaluation comparison between Orbit and Orion.

Evaluation: To compare against Orion, we compare the performance of Orbit on the base configuration setting where $S_{min}=40$, to that of both Orbit and Orion on $S_f=S_{min}=51$. Figure 6 presents our evaluation results. When $S_f=S_{min}=51$, Orbit's ILP-based approach enables it to optimize over entire DAG structures, resulting in better rescale placements and providing 2-9% speedups in evaluation time. Under the base configuration, Orbit finds better bootstrap placements, providing 19-73% speedups and a 31–34% reduction in the number of bootstrap operations compared to Orion. This improvement is due to Orbit's lower waterline scale, which enables the system to save levels and reduce the number of bootstraps required. For instance, Orbit's fine-grained approach can place rescale operations after *multiple* multiplications (whereas Orion requires a rescale after every multiplication), thereby amortizing level consumption across operations.

7.3.3 Comparison to ReSBM

Benchmark	ReSBM R	ReSBM S	Orbit R	Orbit S	Speedup R	Speedup S
Compilation Time (seconds)						
AlexNet	37.9	14.3	223.6	42.8	0.17×	0.33×
ResNet-20	25.2	24.3	52.9	44.3	0.48×	0.55×
SqueezeNet	40.5	16.7	205.0	171.4	0.20×	0.10×
VGG16	36.4	24.2	91.3	48.0	0.40×	0.50×
MobileNet	39.2	37.7	61.7	54.3	0.64×	0.69×
Estimated Execution Time (seconds)						
AlexNet	8565.7	3421.7	5572.2	2966.2	1.54×	1.15×
ResNet-20	3577.4	3400.9	2172.0	1344.6	1.65×	2.53×
SqueezeNet	6714.9	3124.4	4389.3	2486.7	1.53×	1.26×
VGG16	7285.2	4522.1	4981.6	3500.9	1.46×	1.29×
MobileNet	7660.1	6933.6	5820.6	4621.3	1.32×	1.50×

Table 8: Compilation and execution comparison with ReSBM.

ReSBM [35] introduces variable bootstrapping, which allows bootstrapping to arbitrary target levels rather than a fixed L_{bts}, potentially reducing overhead by avoiding unnecessary level restoration. Since Lattigo does not support variable bootstrapping, we implement an estimated cost model by empirically measuring bootstrap costs at different target levels and incorporate this model into Orbit's ILP formulation.

Table 8 compares Orbit against ReSBM on the base configuration and achieves speedups ranging from $1.15-2.53\times$. Although Orbit does not perform variable bootstrapping, it estimates much of the same benefit through a post-placement analysis. After the ILP produces a bootstrap placement, Orbit computes the estimated program latency by summing per-bootstrap costs based on their individually assigned target levels. This approach allows Orbit to estimate the performance benefits of variable bootstrapping without compromising the solvability of the ILP.

Orbit's performance advantage comes from two primary limitations in ReSBM's heuristics. First, ReSBM relies on a greedy rescaling strategy (inserting a rescale immediately when the scale exceeds the rescaling factor), which often results in suboptimal level consumption compared to Orbit's global planning. Second, ReSBM partitions the DAG into

sequential regions based on multiplication depth, enforcing strict uniformity of levels and scales within each region. This structural rigidity proves inefficient for complex sub-DAGs (e.g., activation functions) where edges frequently cross multiple regions. In such cases, ReSBM incurs additional overhead and potentially unnecessary rescales and bootstraps to align levels and scales at region boundaries. For example, on ResNet-20-S, the maximum rescaling depth of ReSBM's compiled program is 325, while the maximum rescaling depth of Orbit's compiled program is 251. This demonstrates that ReSBM's program consumes 29% more levels than Orbit's due to greedy rescaling strategy and region alignment.

7.3.4 End-to-end Accuracy

System	AlexNet	ResNet	SqueezeNet	VGG	MobileNet
Pytorch-S	85.4	92.6	87.5	93.0	91.4
Orbit-S	85.5	92.6	87.5	92.9	91.3
Pytorch-R	88.1	90.6	89.4	93.8	90.0
Orbit-R	87.8	90.7	89.6	93.9	90.0

Table 9: Accuracy (%) on 1000 CIFAR-10 images.

To validate that Orbit's optimizations preserve model accuracy, we evaluate all networks on 1,000 randomly sampled CIFAR-10 test images and compare against plaintext PyTorch baselines. Table 9 shows that Orbit achieves accuracy within 0.3% of PyTorch across all benchmarks, with most models matching exactly. For all benchmarks, we run 1,000 plaintext inferences and verify with 100 encrypted inferences using the $N = 2^{17}$ configuration. The maximum difference between encrypted and plaintext execution is 10^{-4}, demonstrating that Orbit's level-scale aware optimizations introduce negligible numerical error. These results confirm that our bootstrap placement and rescale strategies maintain the accuracy of the original models while delivering performance improvements.

8 Limitations and Future Work

In this section, we discuss Orbit's limitations and potential future work to address them.

Orbit is optimized for *structured* ML tensor workloads, such as CNN inference, rather than arbitrary FHE programs. The effectiveness of its DAG compression and partitioning techniques relies on the computational graph exhibiting regular parallel structures, repeated tensor subgraphs, and clean SISO partition boundaries. Less-structured programs featuring irregular control flow, overlapping skip connections, or complex multi-input/multi-output paths may not be effectively compressed or cleanly partitioned, resulting in high compilation time. Additionally, we employ partitioning heuristics and iterative merging to maintain practical compilation times, but this approach trades some optimality for scalability. While our evaluation demonstrates that these heuristics work well in practice—achieving compilation times under 6 minutes for all benchmarks—they do not guarantee global optimality.

Orbit currently uses a fixed level budget (L_{bts}) for bootstrapping, while recent systems like ReSBM support *variable bootstrapping* to arbitrary target levels. Although our estimated cost model for variable bootstrapping shows promising results (see Section 7.3.3), evaluating with an actual FHE library implementation can demonstrate more concrete results. Future work aims to extend Orbit's ILP formulation to support variable bootstrapping.

Finally, Orbit models FHE operations at the granularity of individual additions, multiplications, rotations, and rescales. However, *functional bootstrapping* [3, 13] can evaluate arbitrary LUTs during the bootstrap operation itself, which can accelerate computationally intensive operations like the large polynomial activation functions. While functional bootstrapping traditionally required scheme switching between CKKS and TFHE with associated conversion costs, recent advances support functional bootstrapping directly within CKKS [3]. Incorporating functional bootstrapping into Orbit's optimization framework is a promising direction for future work.

9 Related Work

The challenge of automatic ciphertext management in FHE has motivated several generations of compiler research, each addressing different aspects of the optimization problem. We discuss related work in three areas: early ILP-based approaches, compilers focusing on rescale placement, and recent systems targeting bootstrap placement.

ILP-based Frameworks Paindavoine and Vialla [37] is the first work to explore ILP-based bootstrap placement, demonstrating performance improvements over greedy placements. Their formulation minimizes the number of bootstraps needed to satisfy multiplicative depth constraints across all computation paths. HEILP [46] is a recent work that introduced an ILP-based scale management, representing an important step toward principled rescale optimization, but its formulation is complicated and does not scale well to large neural networks.

Compilers for Rescale Placement Early compilers for FHE focused primarily on automatic rescale placement. EVA [15] was the first compiler to introduce waterline rescaling, which maintains ciphertext scales above a minimum threshold by proactively inserting rescale operations, but restricts rescaling to multiplication depth boundaries. HECATE [34] extended this specialized scale management operations and global optimizations, achieving better performance over EVA, but incurs longer compile-time overhead. ELASM [33] explored a fine-grained rescale placement by introducing scale-to-noise ratios and noise-aware waterlines.

Compilers for Bootstrap Placement The latest generation of FHE compilers [2, 12, 17, 28, 35] focus on solving automatic bootstrap placement and are the closest work to Orbit. All of these works rely on some depth-based heuristics in order to make the bootstrap placement problem more tractable. DaCapo's [11] approach uses HECATE's scale management to find candidate bootstrap placement points, then relies on depth-based heuristics that can be suboptimal for complex models. ReSBM [35] is one of the first works to try and solve bootstrap placement for variable bootstrapping.

Another class of FHE compilers primarily target layout assignment [9,16,44]. HEIR [4] provides MLIR-based compiler infrastructure for FHE with modular dialect abstractions enabling interoperability across FHE libraries. Our techniques are complementary to these systems, and our MLIR interface should make it easier to integrate into these systems.

10 Conclusion

Orbit is an FHE compiler that jointly optimizes bootstrap and rescale placement through a novel integer linear programming formulation. To overcome ILP scalability challenges, Orbit uses DAG reduction techniques to reduce the ILP problem complexity. Orbit's modular design facilitates integration with existing FHE compiler infrastructure, enabling broader adoption of our optimizations across the FHE ecosystem.

11 Acknowledgements

We thank the anonymous USENIX Security reviewers and our shepherd for their thoughtful comments and guidance. We are also grateful to Aprille Xi for her help with the Lattigo lowering and cost modeling passes. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1745016, NSF Grant CNS-2238671, and a grant from CMU CyLab. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Ethical Considerations

Orbit is a compiler that optimizes bootstrap and rescale placement for FHE programs. While our work focuses on technical compiler optimizations, we recognize that improvements to FHE technology have potential dual-use implications that warrant careful consideration.

Potential Negative Use Cases. More efficient FHE bootstrap placement could enable malicious actors to perform computations on stolen encrypted data with reduced overhead, potentially making such attacks more practical. Enhanced performance might allow adversaries to operate encrypted data processing pipelines that evade detection or regulatory oversight. Additionally, improved FHE efficiency could create asymmetries where powerful entities can afford to deploy these optimizations while smaller organizations cannot, potentially concentrating privacy-preserving capabilities among well-resourced actors.

Weighing Benefits Against Harms. The benefits of efficient bootstrap placement are substantial and widespread. By reducing the performance overhead of FHE, particularly the expensive bootstrap operations that currently dominate execution time, Orbit makes privacy-preserving computation more practical for sensitive applications. This includes secure medical data analysis across institutions, privacy-preserving machine learning inference for sensitive domains, confidential financial computations, and secure cloud computing services. Importantly, by automating bootstrap placement through principled compiler techniques, Orbit democratizes access to efficient FHE implementations, reducing the expertise barrier that currently limits adoption to a small number of cryptography experts who can hand-tune bootstrap placements over days or weeks.

Our Assessment and Future Considerations. We conclude that publishing Orbit serves the greater good. The optimization techniques we present represent standard compiler optimization methodologies applied to the FHE domain. These techniques do not create fundamentally new cryptographic capabilities; rather, they automate and systematize optimization strategies that expert cryptographers have been developing manually for specific applications. Withholding this research would not prevent adversaries from developing similar optimizations independently, while it would significantly hinder legitimate applications where current bootstrap placement challenges remain a major barrier to adoption. The positive applications have clear societal benefits: enabling privacy-preserving healthcare research, protecting user privacy in machine learning services, and facilitating secure outsourced computation. We publish our work through rigorous academic peer review, provide sufficient technical detail for reproducibility and validation, and actively engage with the FHE community to responsibly integrate these optimizations into widely-used open-source infrastructure. We believe that advancing compiler technology for FHE through automated, principled optimization is essential for realizing the promise of privacy-preserving computation, and that responsible publication best serves these societal interests.

Open Science

We fully support the principles of the Open Science Policy. We open source all research artifacts produced for this project under the BSD-3 License, including the complete source code of Orbit and comprehensive documentation for reproducing all evaluation results reported in this paper. We encourage the scientific community to freely ac-

cess, review, validate, and build upon our work. Our open-source implementation is available at https://github.com/cmu-cryptosystems/Orbit and our archived artifact can be found at https://zenodo.org/records/20278435.

References

[1] Lattigo v6. Online: https://github.com/tuneinsight/lattigo, August 2024. EPFL-LDS, Tune Insight SA.

[2] Ehud Aharoni, Allon Adir, Moran Baruch, Nir Drucker, Gilad Ezov, Ariel Farkash, Lev Greenberg, Ramy Masalha, Guy Moshkowich, Dov Murik, et al. Helayers: A tile tensors framework for large neural networks on encrypted data. arXiv preprint arXiv:2011.01805, 2020.

[3] Andreea Alexandru, Andrey Kim, and Yuriy Polyakov. General functional bootstrapping using ckks. In Annual International Cryptology Conference, pages 304–337. Springer, 2025.

[4] Asra Ali, Jaeho Choi, Bryant Gipson, Shruthi Gorantala, Jeremy Kun, Wouter Legiest, Lawrence Lim, Alexander Viand, Meron Zerihun Demissie, and Hongren Zheng. Heir: A universal compiler for homomorphic encryption. arXiv preprint arXiv:2508.11095, 2025.

[5] Ahmad Al Badawi, Andreea Alexandru, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Carlo Pascoe, Yuriy Polyakov, Ian Quah, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. OpenFHE: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2022/915, 2022. https://eprint.iacr.org/2022/915. URL: https://eprint.iacr.org/2022/915.

[6] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In Annual cryptology conference, pages 868–886. Springer, 2012.

[7] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT), 6(3):1–36, 2014.

[8] Alon Brutzkus, Ran Gilad-Bachrach, and Oren Elisha. Low latency privacy preserving inference. In International Conference on Machine Learning, pages 812–821. PMLR, 2019.

[9] Edward Chen, Fraser Brown, and Wenting Zheng. Bridging usability and performance: A tensor compiler for autovectorizing homomorphic encryption. Cryptology ePrint Archive, 2025.

[10] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In International conference on the theory and application of cryptology and information security, pages 409–437. Springer, 2017.

[11] Seonyoung Cheon, Yongwoo Lee, Dongkwan Kim, Ju Min Lee, Sunchul Jung, Taekyung Kim, Dongyoon Lee, and Hanjun Kim. {DaCapo}: Automatic bootstrapping management for efficient fully homomorphic encryption. In 33rd USENIX Security Symposium (USENIX Security 24), pages 6993–7010, 2024.

[12] Seonyoung Cheon, Yongwoo Lee, Hoyun Youm, Dongkwan Kim, Sungwoo Yun, Kunmo Jeong, Dongyoon Lee, and Hanjun Kim. 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, pages 572–585, 2025.

[13] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption over the torus. Journal of Cryptology, 33(1):34–91, 2020.

[14] NM Mosharaf Kabir Chowdhury, Muntasir Raihan Rahman, and Raouf Boutaba. Virtual network embedding with coordinated node and link mapping. In IEEE INFOCOM 2009, pages 783–791. IEEE, 2009.

[15] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. Eva: An encrypted vector arithmetic language and compiler for efficient homomorphic computation. In Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation, pages 546–561, 2020.

[16] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Chet: an optimizing compiler for fully-homomorphic neural-network inferencing. In Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation, pages 142–156, 2019.

[17] Austin Ebel, Karthik Garimella, and Brandon Reagen. Orion: A fully homomorphic encryption framework for deep learning. In Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, pages 734–749, 2025.

[18] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, 2012.

[19] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. 2016.

[20] Fred Glover. Improved linear integer programming formulations of nonlinear integer problems. Management science, 22(4):455–460, 1975.

[21] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2026. URL: https://www.gurobi.com.

[22] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. Logistic regression on homomorphic encrypted data at scale. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 33, pages 9466–9471, 2019.

[23] Christopher Hojny, Mathieu Besançon, Ksenia Bestuzheva, Sander Borst, Antonia Chmiela, João Dionísio, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, Adrian Göß, Alexander Hoen, Rolf van der Hulst, Dominik Kamp, Thorsten Koch, Kevin Kofler, Jurgen Lentz, Stephen J. Maher, Gioni Mexi, Erik Mühmer, Marc E. Pfetsch, Sebastian Pokutta, Felipe Serrano, Yuji Shinano, Mark Turner, Stefan Vigerske, Matthias Walter, Dieter Weninger, and Liding Xu. The SCIP Optimization Suite 10.0. Technical report, Optimization Online, November 2025. URL: https://optimization-online.org/2025/11/the-scip-optimization-suite-10-0/.

[24] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861, 2017.

[25] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. arXiv preprint arXiv:1602.07360, 2016.

[26] IBM. Ibm ilog cplex optimization studio, 2026. URL: https://www.ibm.com/products/ilog-cplex-optimization-studio.

[27] Jae Hyung Ju, Jaiyoung Park, Jongmin Kim, Minsik Kang, Donghwan Kim, Jung Hee Cheon, and Jung Ho Ahn. Neujeans: Private neural network inference with joint optimization of convolution and fhe bootstrapping. pages 4361–4375, 2024.

[28] Aleksandar Krastev, Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. A tensor compiler with automatic data packing for simple and efficient fully homomorphic encryption. Proceedings of the ACM on Programming Languages, 8(PLDI):126–150, 2024.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. Communications of the ACM, 60(6):84–90, 2017.

[30] Eunsang Lee, Joon-Woo Lee, Junghyun Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and Woosuk Choi. Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions. In International Conference on Machine Learning, pages 12403–12422. PMLR, 2022.

[31] Eunsang Lee, Joon-Woo Lee, Jong-Seon No, and Young-Sik Kim. Minimax approximation of sign function by composite polynomial for homomorphic comparison. IEEE Transactions on Dependable and Secure Computing, 19(6):3711–3727, 2021.

[32] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, et al. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. volume 10, pages 30039–30054, 2022.

[33] Yongwoo Lee, Seonyoung Cheon, Dongkwan Kim, Dongyoon Lee, and Hanjun Kim. {ELASM}:{Error-Latency-Aware} scale management for fully homomorphic encryption. In 32nd USENIX Security Symposium (USENIX Security 23), pages 4697–4714, 2023.

[34] Yongwoo Lee, Seonyeong Heo, Seonyoung Cheon, Shinnung Jeong, Changsu Kim, Eunkyung Kim, Dongyoon Lee, and Hanjun Kim. Hecate: performance-aware scale optimization for homomorphic encryption compiler. In 2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pages 193–204. IEEE, 2022.

[35] Yan Liu, Jianxin Lai, Long Li, Tianxiang Sui, Linjie Xiao, Peng Yuan, Xiaojing Zhang, Qing Zhu, Wenguang Chen, and Jingling Xue. Resbm: Region-based scale and minimal-level bootstrapping management for fhe via min-cut. In Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, pages 924–939, 2025.

[36] Jungho Moon, Dongwoo Yoo, Xiaoqian Jiang, and Miran Kim. Thor: Secure transformer inference with ho-

momorphic encryption. Association for Computing Machinery, 2025. doi:10.1145/3719027.3765150.

[37] Marie Paindavoine and Bastien Vialla. Minimizing the number of bootstrappings in fully homomorphic encryption. In International Conference on Selected Areas in Cryptography, pages 25–43. Springer, 2015.

[38] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. Bolt: Privacy-preserving, accurate and efficient inference for transformers. pages 4753–4771. IEEE, 2024. doi:10.1109/SP54263.2024.00262.

[39] Dongjin Park, Eunsang Lee, and Joon-Woo Lee. Powerformer: Efficient privacy-preserving transformer with batch rectifier-power max function and optimized homomorphic attention. Cryptology ePrint Archive, Paper 2024/1921, 2024. URL: https://eprint.iacr.org/2024/1921.

[40] Rajiv Ramaswami and Kumar N Sivarajan. Design of logical topologies for wavelength-routed optical networks. IEEE Journal on Selected areas in communications, 14(5):840–851, 2002.

[41] Eike Schweissguth, Peter Danielis, Dirk Timmermann, Helge Parzyjegla, and Gero Mühl. Ilp-based joint routing and scheduling for time-triggered networks. In Proceedings of the 25th International Conference on Real-Time Networks and Systems, pages 8–17, 2017.

[42] Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL, January 2023. Microsoft Research, Redmond, WA.

[43] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556, 2014.

[44] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. {HECO}: Fully homomorphic encryption compiler. In 32nd USENIX Security Symposium (USENIX Security 23), pages 4715–4732, 2023.

[45] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. Acm sigplan notices, 35(5):121–133, 2000.

[46] Weidong Yang, Shuya Ji, Jianfei Jiang, Naifeng Jing, Qin Wang, Zhigang Mao, and Weiguang Sheng. Heilp: An ilp-based scale management method for homomorphic encryption compiler. In 2025 Design, Automation & Test in Europe Conference (DATE), pages 1–6. IEEE, 2025.

[47] Jiawen Zhang, Xinpeng Yang, Lipeng He, Kejia Chen, Wen-jie Lu, Yinghao Wang, Xiaoyang Hou, Jian Liu, Kui Ren, and Xiaohu Yang. Secure transformer inference made non-interactive. Internet Society, 2025.

A ILP Formulation

Figure 7 in the Appendix details the complete ILP formulation used by Orbit.

B Cost Modeling

To guide the ILP optimization, we profile the actual execution time of each homomorphic operation using the Lattigo library. Our cost model generates representative ciphertexts at each level and measures the average latency and its standard deviation across multiple trials for each operation type: ciphertext-ciphertext addition/multiplication, ciphertext-plaintext addition/multiplication, rotation, upscaling, rescaling, modulus switching, and bootstrapping. For each operation, we collect 100 timing samples (10 for the more expensive bootstrap operation) and compute the mean latency, as shown by the points in Figure 8. To maintain linearity in our ILP formulation, we apply linear regression to the measured timings. This allows us to express operation costs as linear functions of the level decision variables: $\text{cost}_{op} = a \cdot \text{level} + b$, where a and b are the fitted coefficients. Without this linear approximation, the cost model would introduce non-linear terms into the ILP objective function. The linear fits achieve high accuracy ($R^2 > 0.97$ for all operation types) while keeping the ILP tractable.

To validate our cost model, Figure 9 presents our estimated latencies against end-to-end execution times across 312 benchmark configurations, achieving an $R^2 = 0.9975$. These configurations combine 3 cryptographic parameter settings ($N = 2^{15}/2^{17}$ ring dimension, $L_{bts} = 12/16$ level budget, $S_{min} = 40/51$ waterline scale) across 10 macro-benchmarks and 3 layout assignment systems (Orbit, DaCapo, Orion), taking the average of 3 runs per configuration.

C ILP Modeling Assumptions

To maintain the tractability of the ILP optimization problem while preserving solution quality, we adopt six specific modeling assumptions. These constraints reduce the search space complexity with negligible impact on the optimality of the final placement.

1. **Vertex-Restricted Bootstrapping:** Bootstrapping operations are restricted to vertices. If a vertex v is bootstrapped, the refresh applies to all of its outgoing edges. This restriction is non-limiting because placing a bootstrap on an edge (u, v) is functionally equivalent to bootstrapping at vertex u (refreshing all outputs). In cases where downstream neighbors require different input levels, Orbit inserts inexpensive `modswitch` operations on specific edges to satisfy those local constraints.

Variables

For all vertices $v \in V$:

- $v_{\text{in}}.\mathsf{l}, v_{\text{out}}.\mathsf{l} \in [1, L_{bts}]$ — input/output level
- $v_{\text{in}}.\mathsf{s}, v_{\text{out}}.\mathsf{s} \in [S_{min}, S_{max}]$ — input/output scale
- $v.\mathsf{bootstrap} \in \{0,1\}$ — bootstrap indicator
- $v.\mathsf{rescale} \in [0, L_{bts}-1]$ — rescale count

For all edges $e = (u,v) \in E$:

- $e_{\text{in}}.\mathsf{l}, e_{\text{out}}.\mathsf{l} \in [1, L_{bts}]$ — input/output level
- $e_{\text{in}}.\mathsf{s}, e_{\text{out}}.\mathsf{s} \in [S_{min}, S_{max}]$ — input/output scale
- $e.\mathsf{rescale} \in [0, L_{bts}-1]$ — rescale count

Constraints

- **Lattigo Scale Limits:** $\forall x \in V \cup E$
  - $x_{\text{in}}.\mathsf{s} \le S_f \cdot (x_{\text{in}}.\mathsf{l}+1) - 7$
  - $x_{\text{out}}.\mathsf{s} \le S_f \cdot (x_{\text{out}}.\mathsf{l}+1) - 7$
- **Computational:** $\forall v \in V$
  - If $v.\mathsf{op} = \texttt{mul}$ with input edges $e0, e1$:
    $e0_{\text{out}}.\mathsf{l} = e1_{\text{out}}.\mathsf{l} = v_{\text{in}}.\mathsf{l}, \quad e0_{\text{out}}.\mathsf{s} + e1_{\text{out}}.\mathsf{s} = v_{\text{in}}.\mathsf{s}$
  - If $v.\mathsf{op} \neq \texttt{mul}$; $\forall$ input edges e:
    $e_{\text{out}}.\mathsf{l} = v_{\text{in}}.\mathsf{l}, \quad e_{\text{out}}.\mathsf{s} = v_{\text{in}}.\mathsf{s}$
- **Vertex-Edge:** $\forall e = (u,v) \in E$
  - $e_{\text{in}}.\mathsf{l} = u_{\text{out}}.\mathsf{l}, \quad e_{\text{in}}.\mathsf{s} = u_{\text{out}}.\mathsf{s}$
- **Bootstrap:** $\forall v$ with $v.\mathsf{bootstrap} = 1$
  - $v_{\text{in}}.\mathsf{s} \le S_f \cdot v_{\text{in}}.\mathsf{l}$
  - $v_{\text{out}}.\mathsf{l} \ge 2$
  - $v_{\text{out}}.\mathsf{s} \ge S_f$
- **Rescale:**
  - $\forall v$ with $v.\mathsf{bootstrap} = 0$:
    $v_{\text{in}}.\mathsf{l} \ge v_{\text{out}}.\mathsf{l} + v.\mathsf{rescale}, \quad v_{\text{in}}.\mathsf{s} \le v_{\text{out}}.\mathsf{s} + S_f \cdot v.\mathsf{rescale}$
  - $\forall e \in E$:
    $e_{\text{in}}.\mathsf{l} \ge e_{\text{out}}.\mathsf{l} + e.\mathsf{rescale}, \quad e_{\text{in}}.\mathsf{s} \le e_{\text{out}}.\mathsf{s} + S_f \cdot e.\mathsf{rescale}$

Objective Function

$$\begin{aligned}\min \quad & \sum_{v \in V} v.\mathsf{cpr} \cdot \Big[\text{cost}(v.\mathsf{op}, v_{\text{in}}.\mathsf{l}) && \text{operation cost} \\ & \quad + v.\mathsf{bootstrap} \cdot \text{cost}(\texttt{bootstrap}) && \text{bootstrap cost} \\ & \quad + v.\mathsf{rescale} \cdot \text{cost}(\texttt{rescale}) \Big] && \text{vertex rescale cost} \\ & + \sum_{e \in E} e.\mathsf{cpr} \cdot e.\mathsf{rescale} \cdot \text{cost}(\texttt{rescale}) && \text{edge rescale cost}\end{aligned}$$

Figure 7: ILP formulation for Orbit.

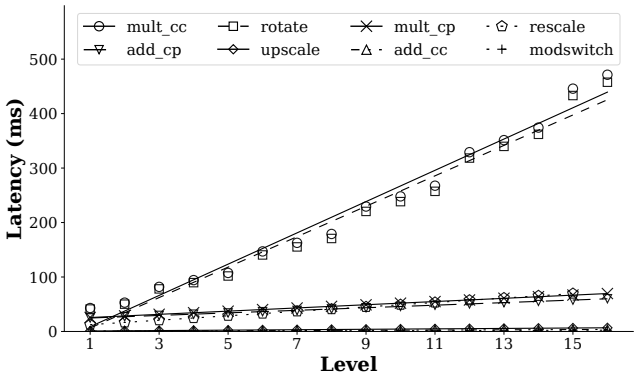


Figure 8: Linear cost model of CKKS operations.

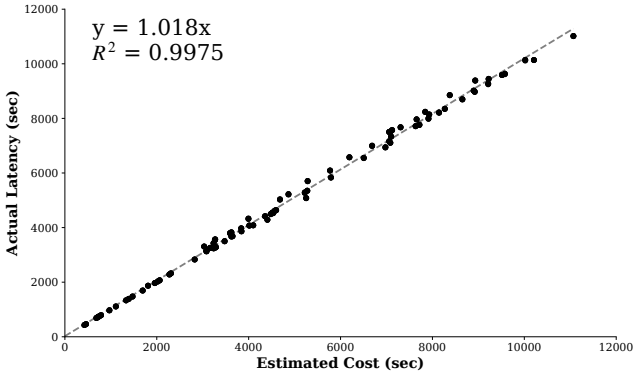


Figure 9: Comparison of estimated versus actual latency. The x-axis shows Orbit's estimates derived from profiled data; the y-axis shows the actual latency measurements.

2. **Bootstrap Dominance:** If a bootstrap is scheduled on a vertex v, the cost of any co-located `rescale` operations on v is excluded from the objective function. Given the strict hierarchy $\text{cost}(\texttt{bootstrap}) \gg \text{cost}(\texttt{rescale})$, the bootstrap cost dominates, rendering the rescale cost negligible in this context.

3. **Scale Upper Bound:** We set the upper bound for scale variables to be $S_{max} = 2 \cdot S_{min} + S_f$. This bound is derived from the worst-case arithmetic growth: a multiplication of two ciphertexts at minimum scale S_{min} yields a result at $2 \cdot S_{min}$. We allocate an additional buffer of S_f to accommodate potential scale management flexibility before a rescale is strictly required. This upper bound effectively prunes the search space of invalid states, which aligns with the approach used in DaCapo.

4. **Negligible Maintenance Costs:** The latency costs of `modswitch` and `upscale` operations are excluded from the objective function. Our evaluation indicates these operations collectively account for $< 0.1\%$ of total execution time, making their inclusion unnecessary for cost minimization.

5. **Linearized Computational Costs:** To enforce linearity in the objective function, we model the cost of arithmetic operations ($\texttt{op} \in \{\texttt{add}, \texttt{mul}, \texttt{rot}\}$) as linear functions of the level variable: $\text{cost}(\texttt{op}, l) = a \cdot l + b$, where a, b are the fitted coefficients. As detailed in Appendix B, this linear regression fits empirical measurements with high accuracy ($R^2 > 0.97$).

6. **Constant Rescale Cost:** We model the cost of `rescale` as a constant fixed at its maximum value: $\text{cost}(\texttt{rescale}) = \text{cost}(\texttt{rescale}, L_{bts} - 1)$. In practice, the target level for rescale operations is typically fixed by the ciphertext, so there is little flexibility to select lower-cost rescale operations without potentially wasting valuable levels. As a result, optimizing only the *count* of rescale operations—rather than their precise cost—has negligible impact on solution quality, but ensures the objective function remains linear and the ILP can be solved efficiently.

D Partitioning Algorithm

Algorithm 1 presents our partitioning algorithm, which tracks vertices by their depth from an input vertex. We group vertices into partitions of at least $N_{\min}$, indicated by the function argument in line 11. To prevent excessive fragmentation (i.e., creating many single-vertex or trivially small partitions), we heuristically set $N_{\min} = 100$. This threshold is user-configurable to balance compilation time and solution quality.

Algorithm 1 DAG Partitioning Algorithm

Require: FHE Computation IR $G = (V, E)$, min partition size $N_{\min}$
Ensure: Set of partitions $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$
1: $depth \leftarrow \text{COMPUTEMAXDEPTH}(G)$
2: $child_depth \leftarrow \text{COMPUTECHILDDEPTH}(G, depth)$
3: $candidates \leftarrow \emptyset$, $live \leftarrow \emptyset$
4: **for** $d = 0$ to $\max(depth)$ **do**
5: $\quad live \leftarrow live \setminus \{v \mid child_depth(v) \leq d\}$
6: $\quad live \leftarrow live \cup \{v \mid depth(v) = d\}$
7: $\quad$ **if** $|live| = 1$ **then**
8: $\quad\quad candidates \leftarrow candidates \cup \{d\}$
9: $\quad$ **end if**
10: **end for**
11: $\mathcal{P} \leftarrow \text{CREATEPARTITIONS}(candidates, depth, N_{\min})$
12: **return** $\mathcal{P}$

E Correctness of the DAG Compression

In this section, we provide the formal proofs for the correctness of our DAG compression technique.

E.1 Preliminaries

Let $G = (V, E)$ denote a computational DAG (FHE Computation IR). We denote a specific ciphertext management schedule (placement) for G as $\text{sol}(G)$. A placement $\text{sol}(G)$ is *valid* if it satisfies all ILP constraints defined in Figure 7. The cost of a placement, denoted $\sigma(\text{sol}(G))$, is the sum of the costs of its constituent operations:

$$\begin{aligned}\sigma(\text{sol}(G)) = \sum_{v \in V} v.\mathsf{cpr} \cdot \Big[&\text{cost}(v.\mathsf{op}, v_{\text{in}}.\mathsf{l}) \\ &+ v.\mathsf{bootstrap} \cdot \text{cost}(\texttt{bootstrap}) \\ &+ v.\mathsf{rescale} \cdot \text{cost}(\texttt{rescale}) \Big] \\ &+ \sum_{e \in E} e.\mathsf{cpr} \cdot e.\mathsf{rescale} \cdot \text{cost}(\texttt{rescale})\end{aligned}$$

We formalize the local neighborhood attributes and the definition of a valid sub-DAG.

Definition E.1 (Vertex Neighborhoods)**.** Given a DAG $G = (V, E)$ and a vertex $v \in V$, we define the parent and child sets as:

- $v.\mathsf{parents} = \{p \in V \mid (p, v) \in E\}$
- $v.\mathsf{children} = \{c \in V \mid (v, c) \in E\}$

Definition E.2 (Sub-DAG)**.** Let $G = (V, E)$ be a DAG. A tuple $D_A = (V_A, E_A)$ is a *sub-DAG* of G if:

- $V_A \subseteq V$ and $E_A \subseteq E$.
- D_A is fully connected.

- E_A contains only edges connecting vertices within V_A (i.e., $E_A \subseteq V_A \times V_A$).

E.2 Equivalence of Sub-DAGs

We now define the structural condition required for two sub-DAGs to be considered compressible. We term this *semi-equivalence*.

Definition E.3 (Sub-DAG Semi-equivalence)**.** Let $G = (V, E)$ be a DAG, and let $D_A = (V_A, E_A)$ and $D_B = (V_B, E_B)$ be *sub-DAGs* of G. D_A and D_B are *semi-equivalent*, if there exists a bijection $f : V_A \leftrightarrow V_B$ such that for each $v_A \in V_A$, all the conditions below are true:

- **Operation Match:** $f(v_A).\mathsf{op} = v_A.\mathsf{op}$.
- **Parent Preservation:** $\forall p_A \in v_A.\mathsf{parents}, \exists p_B \in f(v_A).\mathsf{parents}$ such that $f(p_A) = p_B$.
- **Child Preservation:** $\forall c_A \in v_A.\mathsf{children}, \exists c_B \in f(c_A).\mathsf{children}$ such that $f(c_A) = c_B$.

Note that if D_A and D_B overlap (i.e., $V_A \cap V_B \neq \emptyset$), the identity mapping $f(v) = v$ satisfies these conditions for the overlapping region.

The core of our correctness argument relies on showing that the n-depth similarity metric (Definition 5.1) is sufficient to guarantee the existence of such semi-equivalent sub-DAGs. We prove this constructively.

Theorem E.1 (Similarity Implies Semi-equivalence)**.** Let $G = (V, E)$ be a DAG with maximum depth Δ. If two distinct vertices $v_A, v_B \in V$ satisfy $v_A \approx_\Delta v_B$, then there exist two semi-equivalent sub-DAGs D_A, D_B containing v_A and v_B respectively, with an bijection f such that $f(v_A) = v_B$.

Proof. We prove existence by construction. We define a procedure to construct $D_A = (V_A, E_A)$, $D_B = (V_B, E_B)$, and the bijection f.
Initialization: Set $V_A = \{v_A\}, V_B = \{v_B\}, f(v_A) = v_B$.
Phase 1: Upward Expansion (Ancestry) We perform a breadth-first search (BFS) starting from v_A traversing edges in reverse (towards parents). We maintain the invariant that for all visited $u \in V_A$, $u \approx_\Delta f(u)$.

1. Extract u_A from the frontier. Let $u_B = f(u_A)$. It holds that $u_A \approx_\Delta u_B$.
2. For each incoming edge $e_A = (p_A, u_A)$:
   (a) By the definition of $\approx_\Delta$, there exists a corresponding parent $p_B \in u_B.\mathsf{parents}$ such that $p_A \approx_\Delta p_B$.
   (b) Add p_A to V_A, p_B to V_B, and set $f(p_A) = p_B$.
   (c) Add edges e_A to E_A and $e_B = (p_B, u_B)$ to E_B.
   (d) If p_A is unvisited, and $p_A \neq p_B$, add p_A to frontier.

We repeat this process until the frontier is empty. This ends Phase 1. At the end of Phase 1, the roots nodes of V_A and V_B will be identical. These roots represent the point in the computation where it splits into semi-equivalent sub-DAGs. Additionally, since $p_A \approx_\Delta p_B$ holds at every step, the invariant is preserved.
Phase 2: Downward Expansion (Descendants) We perform a forward BFS starting from the roots identified in Phase 1. We maintain the invariant that for all visited $u \in V_A$, $u \approx_k f(u)$ for some $k \geq 0$.

1. Extract u_A from the frontier. Let $u_B = f(u_A)$. It holds that $u_A \approx_k u_B$ for some positive k.
2. For each outgoing edge $e_A = (u_A, c_A)$:
   (a) By definition of $\approx_k$ (where $k > 0$), there exists a child $c_B \in u_B.\mathsf{children}$ such that $c_A \approx_{k-1} c_B$.
   (b) Add c_A to V_A, c_B to V_B, and set $f(c_A) = c_B$.
   (c) Add edges e_A to E_A and $e_B = (u_B, c_B)$ to E_B.
   (d) If c_A is unvisited, and $c_A \neq c_B$, add c_A to frontier.

We repeat this process until the frontier is empty. This ends Phase 2. At the end of Phase 2, the the bijection $f : V_A \leftrightarrow V_B$ maintains the property that for each $v_A \in V$, $v_A \approx_k f(v_A)$ for some non-negative k. Return D_A and D_B.
Correctness: The resulting graphs D_A and D_B are sub-DAGs by construction, as the BFS process ensures that both D_A and D_B are fully connected and only involve vertices from their respective vertex sets V_A and V_B.

To show that D_A and D_B are semi-equivalent, we check that the bijection f established during traversal satisfies all conditions in Definition E.3. The first condition (operation match) is satisfied from the fact that for all $v_A \in V_A$, $v_A \approx_k f(v_A)$ for some non-negative k. The latter two conditions (parent/child preservation) are satisfied, because for every edge $e_A = (p_A, c_A)$ added to E_A during the BFS searches, there is an edge $e_B = (p_B, c_B)$ added to E_B such that $f(p_A) = p_B$ and $f(c_A) = c_B$.

Additionally, the steps in the BFS searches ensure that edges are branched out v_A and v_B in an equivalent manner when constructing D_A and D_B. This ensures v_A and v_B are corresponding vertices on the two *sub-DAGs*. Thus, D_A and D_B are semi-equivalent sub-DAGs with $f(v_A) = v_B$. □

E.3 Compression and Decompression of Placements

To formalize the relationship between the original DAG G and the compressed DAG $\tilde{G}$, we define a mapping function $g : V \to \tilde{V}$ based on the equivalence classes formed by Δ-depth similarity. Specifically, $u \approx_\Delta v \iff g(u) = g(v)$.

Definition E.4 (DAG Compression)**.** Let $G = (V, E)$ be a DAG where all base compression factors are unity ($\forall x \in$

$V \cup E, x.\mathsf{cpr} = 1$). The compressed DAG $\tilde{G} = \mathsf{reduce}(G)$ is defined as $\tilde{G} = (\tilde{V}, \tilde{E})$, where:

- **Vertices:** $\tilde{V}$ is the image of V under g.
- **Edges:** $\tilde{E}$ is the set of edges $(\tilde{u}, \tilde{v})$ such that there exists at least one edge $(u, v) \in E$ with $g(u) = \tilde{u}$ and $g(v) = \tilde{v}$.
- **Attributes:**
  - For $\tilde{v} \in \tilde{V}$: $\tilde{v}.\mathsf{op} = v.\mathsf{op}$ (where $g(v) = \tilde{v}$). This is well-defined as all equivalent vertices share the same operator.
  - The compression factor is the size of the pre-image:

$$\tilde{v}.\mathsf{cpr} = |\{v \in V \mid g(v) = \tilde{v}\}|$$

  - For $\tilde{e} = (\tilde{u}, \tilde{v}) \in \tilde{E}$:

$$\tilde{e}.\mathsf{cpr} = |\{(u, v) \in E \mid g(u) = \tilde{u} \wedge g(v) = \tilde{v}\}|$$

The following definition formalizes how a placement for the compressed DAG can be systematically expanded to yield a placement for the original DAG.

Definition E.5 (Placement Reconstruction)**.** Let $\mathrm{sol}\left(\tilde{G}\right)$ be a placement for the compressed DAG $\tilde{G}$. The reconstructed placement for the original DAG, denoted $\mathrm{sol}(G) = \mathsf{reconstruct}\left(\mathrm{sol}\left(\tilde{G}\right), G\right)$, is defined by expanding the compressed attributes:

- For every vertex $v \in V$: Assign the attributes of $g(v)$ from $\mathrm{sol}\left(\tilde{G}\right)$, and set $v.\mathsf{cpr} = 1$.
- For every edge $e = (u, v) \in E$: Assign the attributes of $\tilde{e} = (g(u), g(v))$ from $\mathrm{sol}\left(\tilde{G}\right)$, and set $e.\mathsf{cpr} = 1$.

With the reconstruction process defined, we must verify that the resulting placement on the original DAG is valid. A specific topological edge case arises in the compressed DAG $\tilde{G}$: it is possible for a multiplication vertex $\tilde{v}$ to have only a *single* incoming edge $\tilde{e} = (\tilde{u}, \tilde{v})$ if both operands in the original graph come from the same equivalence class. In this specific case, the ILP formulation treats $\tilde{u}$ as supplying both operands, and the constraints become:

$$\tilde{v}_{\text{in}}.\mathsf{l} = \tilde{e}_{\text{out}}.\mathsf{l}, \quad \tilde{v}_{\text{in}}.\mathsf{s} = 2 \cdot \tilde{e}_{\text{out}}.\mathsf{s}$$

We demonstrate that validity is preserved.

Lemma E.1 (Decompression Validity)**.** If $\mathrm{sol}\left(\tilde{G}\right)$ is a *valid* placement for the compressed DAG $\tilde{G}$, then $\mathrm{sol}(G) = \mathsf{reconstruct}\left(\mathrm{sol}\left(\tilde{G}\right), G\right)$ is a *valid* placement for the original DAG G.

Proof. We verify that $\mathrm{sol}(G)$ satisfies all ILP constraints defined in Figure 7.

1. **Variable Bounds:** For any $v \in V$, the reconstructed attributes are direct copies: $v_{\text{in}}.\mathsf{l}, \mathsf{s} = \tilde{v}_{\text{in}}.\mathsf{l}, \mathsf{s}$. Since $\tilde{v}$ satisfies global bounds and Lattigo scale limits in $\mathrm{sol}\left(\tilde{G}\right)$, v satisfies them in $\mathrm{sol}(G)$. The same logic applies to edges.

2. **Vertex-Edge:** Consider edge $e = (u, v) \in E$ mapping to $\tilde{e} = (\tilde{u}, \tilde{v})$. By reconstruction, $e_{\text{in}}.\mathsf{l}, \mathsf{s} = \tilde{e}_{\text{in}}.\mathsf{l}, \mathsf{s}$ and $u_{\text{out}}.\mathsf{l}, \mathsf{s} = \tilde{u}_{\text{out}}.\mathsf{l}, \mathsf{s}$. Since $\mathrm{sol}\left(\tilde{G}\right)$ is valid, $\tilde{e}_{\text{in}}.\mathsf{l}, \mathsf{s} = \tilde{u}_{\text{out}}.\mathsf{l}, \mathsf{s}$. Transitively, $e_{\text{in}}.\mathsf{l}, \mathsf{s} = u_{\text{out}}.\mathsf{l}, \mathsf{s}$. Thus, topology constraints hold.

3. **Computational:** For level compatibility, for any $v \in V$ with input edge e, levels are inherited directly from $\tilde{v}$ and $\tilde{e}$. Since $\tilde{v}_{\text{in}}.\mathsf{l} = \tilde{e}_{\text{out}}.\mathsf{l}$ holds in $\mathrm{sol}\left(\tilde{G}\right)$, $v_{\text{in}}.\mathsf{l} = e_{\text{out}}.\mathsf{l}$ holds in $\mathrm{sol}(G)$. Similarly, the scale compatibility constraints for non-multiplication operations hold.

   For scale constraints for multiplication, let v be a multiplication node with inputs $e0 = (u_0, v)$ and $e1 = (u_1, v)$. **Case A:** $g(u_0) \neq g(u_1)$. $\tilde{v}$ has two distinct inputs $\tilde{e0}, \tilde{e1}$. The constraint $\tilde{v}_{\text{in}}.\mathsf{s} = \tilde{e0}_{\text{out}}.\mathsf{s} + \tilde{e1}_{\text{out}}.\mathsf{s}$ maps directly to v. **Case B:** $g(u_0) = g(u_1)$. $\tilde{v}$ has a single input $\tilde{e}$. The reconstruction assigns $e0_{\text{out}}.\mathsf{s} = e1_{\text{out}}.\mathsf{s} = \tilde{e}_{\text{out}}.\mathsf{s}$. The original constraint requires $v_{\text{in}}.\mathsf{s} = e0_{\text{out}}.\mathsf{s} + e1_{\text{out}}.\mathsf{s}$. Substituting the reconstructed values: $v_{\text{in}}.\mathsf{s} = 2 \cdot \tilde{e}_{\text{out}}.\mathsf{s}$. This matches the special compressed constraint for single-input multiplication. Thus, validity holds.

4. **Rescale and Bootstrap:** These are local constraints on vertices/edges. Since attributes (rescale counts, bootstrap indicators) are copied identically from valid counterparts in $\mathrm{sol}\left(\tilde{G}\right)$, the constraints are satisfied.

□

Having established the validity of the reconstructed placement, we next demonstrate that the reconstruction process preserves the total cost exactly.

Lemma E.2 (Decompression Cost)**.** For any placement $\mathrm{sol}\left(\tilde{G}\right)$, the reconstructed placement $\mathrm{sol}(G) = \mathsf{reconstruct}\left(\mathrm{sol}\left(\tilde{G}\right), G\right)$ satisfies $\sigma(\mathrm{sol}(G)) = \sigma\left(\mathrm{sol}\left(\tilde{G}\right)\right)$.

Proof. Let $\mathrm{cost}(x)$ denote the operational cost of a vertex or edge x (excluding the compression factor weight). That is,

$$\sigma(\mathrm{sol}(G)) = \sum_{v \in V} v.\mathsf{cpr} \cdot \mathrm{cost}(v) + \sum_{e \in E} e.\mathsf{cpr} \cdot \mathrm{cost}(e)$$

By definition of reconstruction, for any $v \in V$, $\mathrm{cost}(v) = \mathrm{cost}(g(v))$, and for any $e \in E$, $\mathrm{cost}(e) = \mathrm{cost}(g(e))$.

We expand the cost function of the compressed graph:

$$\begin{aligned}\sigma\left(\mathrm{sol}\left(\tilde{G}\right)\right) &= \sum_{\tilde{v}\in\tilde{V}} \tilde{v}.\mathsf{cpr}\cdot\mathrm{cost}(\tilde{v}) \\ &\quad + \sum_{\tilde{e}\in\tilde{E}} \tilde{e}.\mathsf{cpr}\cdot\mathrm{cost}(\tilde{e}) \\ &= \sum_{\tilde{v}\in\tilde{V}} |\{v \mid g(v)=\tilde{v}\}|\cdot\mathrm{cost}(\tilde{v}) \\ &\quad + \sum_{\tilde{e}\in\tilde{E}} |\{(u,v) \mid (g(u),g(v))=\tilde{e}\}|\cdot\mathrm{cost}(\tilde{e})\end{aligned}$$

We can rewrite these sums by iterating over the pre-images in the original graph G:

$$\begin{aligned}\sigma\left(\mathrm{sol}\left(\tilde{G}\right)\right) &= \sum_{\tilde{v}\in\tilde{V}}\sum_{v\in g^{-1}(\tilde{v})} \mathrm{cost}(g(v)) + \sum_{\tilde{e}\in\tilde{E}}\sum_{e\in g^{-1}(\tilde{e})} \mathrm{cost}(g(e)) \\ &= \sum_{v\in V}\mathrm{cost}(v) + \sum_{e\in E}\mathrm{cost}(e) \\ &= \sigma(\mathrm{sol}(G))\end{aligned}$$

Thus, the total cost is exactly preserved. □

E.4 Proof of Theorem 5.1

We first re-state the theorem using the formalism of placement reconstruction.

Theorem E.2 (Compression Preserves Search Optimality)**.** Let $G=(V,E)$ be a DAG and $\tilde{G}=\mathsf{reduce}(G)$ be its compressed form. If $\mathrm{sol}\left(\tilde{G}\right)$ is an valid and optimal placement for $\tilde{G}$, then the reconstructed placement $\mathrm{sol}(G) = \mathsf{reconstruct}\left(\mathrm{sol}\left(\tilde{G}\right), G\right)$ is valid and optimal for G.

Proof. **Validity:** The validity of $\mathrm{sol}(G)$ is established by Lemma E.1.

Optimality: We prove this by contradiction. Assume there exists a placement $\mathrm{sol}(G)^*$ for the original DAG G that is strictly better than the reconstructed placement $\mathrm{sol}(G)$. That is, $\sigma(\mathrm{sol}(G)^*) < \sigma(\mathrm{sol}(G))$. Without loss of generality, let $\mathrm{sol}(G)^*$ be the *optimal* placement for G.

Our strategy is to construct a *symmetric* placement $\mathrm{sol}(G)^{**}$ from $\mathrm{sol}(G)^*$ such that:

1. $\mathrm{sol}(G)^{**}$ is valid and optimal ($\sigma(\mathrm{sol}(G)^{**}) \le \sigma(\mathrm{sol}(G)^*)$).
2. All equivalent vertices and edges in $\mathrm{sol}(G)^{**}$ share identical attributes.
3. $\mathrm{sol}(G)^{**}$ can be compressed into a valid solution $\mathrm{sol}\left(\tilde{G}\right)^*$ for $\tilde{G}$ with $\sigma(\mathrm{sol}\left(\tilde{G}\right)^*) < \sigma(\mathrm{sol}\left(\tilde{G}\right))$, violating the optimality of $\mathrm{sol}\left(\tilde{G}\right)$.

Step 1: Symmetrizing Vertices

Consider any pair of vertices $v_A, v_B \in V$ such that $g(v_A) = g(v_B)$ (i.e., they are in the same equivalence class) but they are assigned different attributes in $\mathrm{sol}(G)^*$.

By the definition of Δ-level similarity ($g(v_A)=g(v_B)$), we invoke Theorem E.1 to identify two semi-equivalent sub-DAGs D_A and D_B that v_A, v_B are corresponded. Let $\mathrm{sol}(D_A)$ and $\mathrm{sol}(D_B)$ denote their respective partial placements in $\mathrm{sol}(G)^*$.

- **Case 1 ($\sigma(\mathrm{sol}(D_A)) < \sigma(\mathrm{sol}(D_B))$):** We construct a new placement by replacing the attributes of D_B with the attributes of D_A (mapped via the isomorphism $f: V_A \to V_B$). Since D_A, D_B are semi-equivalent, the internal validity constraints are identical. The new cost is strictly lower, contradicting the assumption that $\mathrm{sol}(G)^*$ is optimal. Thus, this case is impossible for an optimal solution.
- **Case 2 ($\sigma(\mathrm{sol}(D_B)) < \sigma(\mathrm{sol}(D_A))$):** Symmetric to Case 1; impossible.
- **Case 3 ($\sigma(\mathrm{sol}(D_A)) = \sigma(\mathrm{sol}(D_B))$):** The costs are identical, but attributes differ. We arbitrarily standardize by overwriting the attributes of D_B with those of D_A. This operation preserves total cost and validity while reducing the number of asymmetric pairs.

By iteratively applying this standardization to all equivalence classes, we obtain a placement where all $v \in g^{-1}(\tilde{v})$ share identical attributes. This process terminates in $O(|V|^2)$ steps.

Step 2: Symmetrizing Edges

Similarly, consider edges $e_A=(u_A,v_A)$ and $e_B=(u_B,v_B)$ such that $g(u_A)=g(u_B)$ and $g(v_A)=g(v_B)$. If their attributes differ, we again invoke the semi-equivalence property. Since $u_A \approx_\Delta u_B$, the bijection f constructed in Theorem E.1 can ensure $f(u_A)=u_B$, effectively mapping the edge e_A to e_B. Using the same logic as Step 1, if $\mathrm{cost}(D_A) \neq \mathrm{cost}(D_B)$, we could swap the sub-DAG configurations to lower the total cost (contradicting optimality). If costs are equal, we overwrite D_B's attributes with D_A's to enforce symmetry. This process terminates in $O(|E|^2)$ steps.

Step 3: Compression and Contradiction

After Steps 1 and 2, we possess a placement $\mathrm{sol}(G)^{**}$ that is:

1. **Optimal:** $\sigma(\mathrm{sol}(G)^{**}) = \sigma(\mathrm{sol}(G)^*)$.
2. **Symmetric:** For all $x,y \in V \cup E$, $g(x)=g(y) \implies \mathrm{attr}(x)=\mathrm{attr}(y)$.[5]

[5] For edge $e=(u,v)\in E$, we denote $g(e)$ as $(g(u),g(v))$.

We can now unambiguously define a placement $\text{sol}\left(\tilde{G}\right)^*$ for the compressed graph $\tilde{G}$. For every $\tilde{x} \in \tilde{V} \cup \tilde{E}$, we assign the attributes of its pre-image from $\text{sol}(G)^{**}$. This is well-defined due to the symmetry of $\text{sol}(G)^{**}$.

We observe two properties of $\text{sol}\left(\tilde{G}\right)^*$:

1. **Validity:** It satisfies all compressed constraints, as they are mere aggregations of the satisfied constraints in $\text{sol}(G)^{**}$.

2. **Cost:** By Lemma E.2, $\sigma(\text{sol}\left(\tilde{G}\right)^*) = \sigma(\text{sol}(G)^{**})$.

Combining our inequalities:

$$\begin{aligned}\sigma(\text{sol}\left(\tilde{G}\right)^*) = \sigma(\text{sol}(G)^{**}) &\leq \sigma(\text{sol}(G)^*) \\ &< \sigma(\text{sol}(G)) = \sigma(\text{sol}\left(\tilde{G}\right))\end{aligned}$$

This implies $\sigma(\text{sol}\left(\tilde{G}\right)^*) < \sigma(\text{sol}\left(\tilde{G}\right))$, which contradicts the initial assumption that $\text{sol}\left(\tilde{G}\right)$ was the optimal placement for the compressed graph. Therefore, the reconstructed placement $\text{sol}(G)$ must be optimal. □

F Correctness of the DAG Partitioning

In this section, we provide a formal proof of Theorem 5.2. We begin by defining the algebraic properties of DAG composition and decomposition, showing that the cost function and validity constraints are preserved across SISO boundaries.

We adopt the notation for DAGs (G), placements ($\text{sol}(G)$), and costs ($\sigma(\text{sol}(G))$) established in Appendix E.

For SISO sub-DAGs, we introduce the concept of a *Virtual Input Node*. The input vertex v_{in} of a SISO partition acts solely as a data source; therefore, we assign it the operator $v_{in}.\mathsf{op} = \perp$ (identity operation), with $\text{cost}(\perp, l) = 0$ for all levels $1 \leq l \leq L_{bts}$.

F.1 Composability of Placements

We first define the composition of two compatible SISO sub-DAGs into a larger structure.

Definition F.1 (DAG Composition)**.** Let $D_1 = (V_1, E_1)$ and $D_2 = (V_2, E_2)$ be two SISO DAGs. Let $u \in V_1$ be the output vertex of D_1, and $w \in V_2$ be the input vertex of D_2. The pair (D_1, D_2) is *composable* if the compression factors match at the boundary: $u.\mathsf{cpr} = w.\mathsf{cpr}$.

The composition $G = D_1 \oplus D_2$ is defined as $G = (\hat{V}, \hat{E})$, where:

- $\hat{V} = V_1 \cup V_2$.
- $\hat{E} = E_1 \cup E_2 \cup \{e^{bond} = (u, w)\}$.
- The boundary edge e^{bond} inherits $e^{bond}.\mathsf{cpr} = 0$.

Definition F.2 (Placement Composition)**.** Let $\text{sol}(D_1)$ and $\text{sol}(D_2)$ be valid placements for composable DAGs D_1 and D_2. The pair $(\text{sol}(D_1), \text{sol}(D_2))$ is *composable* if the boundary states align: $(u_{\text{out}}.\mathsf{l}, u_{\text{out}}.\mathsf{s}) = (w_{\text{in}}.\mathsf{l}, w_{\text{in}}.\mathsf{s})$.

The composed placement $\text{sol}(G) = \text{sol}(D_1) \oplus \text{sol}(D_2)$ is constructed by taking the union of all attribute assignments. For the boundary edge $e^{bond} = (u, w)$, we assign:

- $e_{\text{in}}.\mathsf{l}, \mathsf{s} = u_{\text{out}}.\mathsf{l}, \mathsf{s}$ and $e_{\text{out}}.\mathsf{l}, \mathsf{s} = w_{\text{in}}.\mathsf{l}, \mathsf{s}$.
- $e.\mathsf{rescale} = 0$.

Lemma F.1 (Validity Preservation)**.** If $\text{sol}(D_1)$ and $\text{sol}(D_2)$ are *valid* composable placements, then $\text{sol}(G) = \text{sol}(D_1) \oplus \text{sol}(D_2)$ is a *valid* placement for $G = D_1 \oplus D_2$.

Proof. Constraints for internal vertices and edges in D_1 and D_2 are satisfied by hypothesis. We inspect the boundary edge $e^{bond} = (u, w)$:

1. **Variable bounds:** By definition, ${e^{bond}}_{\text{in}}.\mathsf{l}, \mathsf{s} = u_{\text{out}}.\mathsf{l}, \mathsf{s}$ and ${e^{bond}}_{\text{out}}.\mathsf{l}, \mathsf{s} = w_{\text{in}}.\mathsf{l}, \mathsf{s}$. Since the variable bound and Lattigo scale limit constraints hold for u and w, they hold for e^{bond}.

2. **Computational:** Since $w.\mathsf{op} = \perp$, and ${e^{bond}}_{\text{out}}.\mathsf{l}, \mathsf{s} = w_{\text{in}}.\mathsf{l}, \mathsf{s}$, the computational constraint for w is satisfied.

3. **Vertex-Edge:** By definition, ${e^{bond}}_{\text{in}}.\mathsf{l}, \mathsf{s} = u_{\text{out}}.\mathsf{l}, \mathsf{s}$, so the vertex-edge consistency holds for e^{bond}.

4. **Rescale:** Since $e^{bond}.\mathsf{rescale} = 0$ and levels/scales are equal across the edge, the rescale constraint for e^{bond} is trivially satisfied.

Thus, $\text{sol}(G)$ satisfies all global ILP constraints. □

Lemma F.2 (Cost Additivity)**.** If $\text{sol}(D_1)$ and $\text{sol}(D_2)$ are composable placements, then $\sigma(\text{sol}(D_1) \oplus \text{sol}(D_2)) = \sigma(\text{sol}(D_1)) + \sigma(\text{sol}(D_2))$.

Proof. The vertex sets are disjoint, so vertex costs sum linearly. The edge sets are disjoint except for e^{bond}. Since $e^{bond}.\mathsf{cpr} = 0$, it contributes zero cost. Thus, the total cost is strictly additive. □

F.2 Decomposability of Placements

First, we define the structural condition that allows a DAG to be split.

Definition F.3 (DAG Separability)**.** A DAG $D = (V, E)$ is *separable* into SISO sub-DAGs (D_1, D_2) at boundary vertex $u \in V$ if D can be formed by taking the composition $D_1 \oplus D_2$ and contracting the boundary:

1. Remove the virtual input node w of D_2.
2. Remove the bond edge $e^{bond} = (u, w)$.

3. Rewire all edges starting from w to start from u.

Formally, $V = V_1 \cup (V_2 \setminus \{w\})$ and $E = E_1 \cup \{(u,z) \mid (w,z) \in E_2\} \cup \{(x,y) \mid (x,y) \in E_2, x \neq w\}$.

We now define the inverse operation: extracting sub-solutions from a global DAG.

Definition F.4 (Placement Decomposition)**.** Let D be separable into (D_1, D_2) at u. Given a valid placement $\mathrm{sol}(D)$, the decomposition $\mathrm{split}(\mathrm{sol}(D), D_1, D_2) = (\mathrm{sol}(D_1), \mathrm{sol}(D_2))$ is defined as:

1. Extracting $\mathrm{sol}(D_1)$: Includes all vertices $v \in V_1$ and edges $e \in E_1$ with attributes from $\mathrm{sol}(D)$.

2. Extracting $\mathrm{sol}(D_2)$: Includes all vertices $v \in V_2 \setminus \{w\}$ with attributes from $\mathrm{sol}(D)$. To reconstruct the valid SISO structure of D_2, we restore the virtual input components:

- **Virtual Node w:** Added with attributes inherited from u: $w.\mathsf{cpr} = u.\mathsf{cpr}, w_{\text{in}}.\mathsf{l}, \mathsf{s} = w_{\text{out}}.\mathsf{l}, \mathsf{s} = u_{\text{out}}.\mathsf{l}, \mathsf{s}$. The operator is set to $w.\mathsf{op} = \perp$. The rescaling and bootstrapping attributes are set to zero, i.e. $w.\mathsf{rescale} = 0$, $w.\mathsf{bootstrap} = 0$.

- **Rewired Edges:** For every edge (u,z) in $\mathrm{sol}(D)$ where $z \in V_2$, we create the edge (w,z) in $\mathrm{sol}(D_2)$ carrying the exact attributes of (u,z).

We now verify that this decomposition preserves validity, which is the inverse of composition validity in Lemma F.1.

Lemma F.3 (Decomposition Validity)**.** If $\mathrm{sol}(D)$ is a *valid* placement, then the induced placements $\mathrm{sol}(D_1)$ and $\mathrm{sol}(D_2)$ are *valid*.

Proof. **Validity of $\mathrm{sol}(D_1)$:** Since $\mathrm{sol}(D_1)$ strictly inherits vertices and edges from $\mathrm{sol}(D)$, and $\mathrm{sol}(D)$ is valid, all internal constraints for $\mathrm{sol}(D_1)$ are satisfied.

Validity of $\mathrm{sol}(D_2)$: The attributes of all internal vertices and edges ($z \in V_2 \setminus \{w\}$) are inherited from $\mathrm{sol}(D)$ and thus valid. We need only verify the constraints related to the reconstructed virtual node w and the rewired edges $(w,z) \in E_2$.

- **Variable Bounds:** The level and scale attributes of w are explicitly inherited from u (i.e., $w_{\text{in}}.\mathsf{l}, \mathsf{s} = u_{\text{out}}.\mathsf{l}, \mathsf{s}$). Since u satisfies all variable bounds and Lattigo scale limits in the valid solution $\mathrm{sol}(D)$, these bounds necessarily hold for w.

- **Vertex-Edge Constraints:** Consider any rewired edge $e' = (w,z) \in E_2$. By construction, its input level is $e_{\text{in}}.\mathsf{l} = (u,z)_{\text{in}}.\mathsf{l}$. Since the original edge (u,z) was valid in $\mathrm{sol}(D)$, its input matched the output of u: $(u,z)_{\text{in}}.\mathsf{l} = u_{\text{out}}.\mathsf{l}$. By our reconstruction of w, $u_{\text{out}}.\mathsf{l} = w_{\text{out}}.\mathsf{l}$. Therefore, $e_{\text{in}}.\mathsf{l} = w_{\text{out}}.\mathsf{l}$, satisfying the vertex-edge constraint for w. The same logic applies to scale attributes.

- **Rescale Constraints (Vertex):** We explicitly set $w.\mathsf{bootstrap} = 0$ and $w.\mathsf{rescale} = 0$. Since we assigned $w_{\text{in}}.\mathsf{l}, \mathsf{s} = w_{\text{out}}.\mathsf{l}, \mathsf{s}$, the standard rescale invariant holds with equality. Thus, w satisfies rescale constraints.

- **Rescale Constraints (Edge):** The rewired edge (w,z) inherits all attributes (including rescale counts) from the original edge (u,z). Since (u,z) satisfied rescale constraints in $\mathrm{sol}(D)$, (w,z) satisfies them in $\mathrm{sol}(D_2)$.

Since all component constraints hold, $\mathrm{sol}(D_2)$ is valid. □

Next, we show that the decomposition preserves the total cost, which is the inverse of composition cost in Lemma F.2.

Lemma F.4 (Decomposition Cost)**.** Suppose $D = (V,E)$ is a DAG separable into (D_1, D_2) at boundary u. Let $(\mathrm{sol}(D_1), \mathrm{sol}(D_2)) = \mathrm{split}(\mathrm{sol}(D))$. Then $\sigma(\mathrm{sol}(D)) = \sigma(\mathrm{sol}(D_1)) + \sigma(\mathrm{sol}(D_2))$.

Proof. To prove this equality, we verify the cost contribution of every structural difference between the real DAG $\mathrm{sol}(D)$ and the decomposed pair $(\mathrm{sol}(D_1), \mathrm{sol}(D_2))$. Specifically, we address the virtual bond edge, the virtual input vertex, and the rewired edges.

1. **Bond Edge Cost:** In the composition view (which relates D to $D_1 \oplus D_2$), there exists a logical bond edge (u,w). However, its compression factor is $(u,w).\mathsf{cpr} = 0$. Thus, its exclusion from $\mathrm{sol}(D)$ and $\mathrm{sol}(D_1)/\mathrm{sol}(D_2)$ does not affect the cost equality.

2. **Virtual Vertex Cost:** The decomposition adds the virtual input vertex w to $\mathrm{sol}(D_2)$. Since $w.\mathsf{op} = \perp$ and we enforce $w.\mathsf{rescale} = w.\mathsf{bootstrap} = 0$, the computational cost of this additional vertex is 0.

3. **Rewired Edge Cost:** The edges $E_{cross} = \{(u,z)\}$ in $\mathrm{sol}(D)$ are replaced by $E_{virtual} = \{(w,z)\}$ in $\mathrm{sol}(D_2)$. Since the attributes of (w,z) are explicitly copied from (u,z), they share the exact same compression factors and rescale counts. Therefore, $\sum \mathrm{cost}(E_{cross}) = \sum \mathrm{cost}(E_{virtual})$.

Since all other vertices and edges in D_1 and D_2 are identical to those in D, the total cost is conserved: $\sigma(\mathrm{sol}(D)) = \sigma(\mathrm{sol}(D_1)) + \sigma(\mathrm{sol}(D_2))$. □

F.3 Proof of Theorem 5.2

Proof. We proceed by induction on the number of partitions m. Let $\mathrm{dp}(i, l_{\text{out}}, s_{\text{out}})$ denote the minimum cost computed by the algorithm for the first i partitions ending in state $(l_{\text{out}}, s_{\text{out}})$.

Base Case ($i = 1$): $\mathrm{dp}(1, l_{\text{out}}, s_{\text{out}}) = \min_{l_{\text{in}}, s_{\text{in}}} \mathsf{QBP}_{D_1}(l_{\text{in}}, s_{\text{in}}, l_{\text{out}}, s_{\text{out}})$. By definition, the QBP stores the optimal cost for D_1.

Inductive Step:

Assume $\forall l_{\text{in}}, s_{\text{in}}$, $\text{dp}(i-1, l_{\text{in}}, s_{\text{in}})$ is optimal. We show $\forall l_{\text{out}}, s_{\text{out}}$, $\text{dp}(i, l_{\text{out}}, s_{\text{out}})$ is optimal. We denote G_i as the first i partitions of G.

Suppose for contradiction that there exists a global placement $\text{sol}(G_i)^*$ for the first i partitions ending at $(l_{\text{out}}, s_{\text{out}})$ with $\sigma(\text{sol}(G_i)^*) < \text{dp}(i, l_{\text{out}}, s_{\text{out}})$.

Our partitioning algorithm ensures that G_i is separable into (G_{i-1}, D_i) at boundary vertex u. Apply the decomposition $\text{split}(\text{sol}(G_i)^*) = (\text{sol}(G_{i-1})^*, \text{sol}(D_i)^*)$. Let the boundary state at vertex u (output of $i-1$) be $(l^*_{\text{in}}, s^*_{\text{in}})$.

By Lemma F.3, both $\text{sol}(G_{i-1})^*$ and $\text{sol}(D_i)^*$ are valid placements. By Lemma F.4, their costs sum to that of the global solution: $\sigma(\text{sol}(G_i)^*) = \sigma(\text{sol}(G_{i-1})^*) + \sigma(\text{sol}(D_i)^*)$. From the inductive hypothesis, $\text{dp}(i-1, l^*_{\text{in}}, s^*_{\text{in}}) \le \sigma(\text{sol}(G_{i-1})^*)$. From QBP optimality, $\text{QBP}_{D_i}(l^*_{\text{in}}, s^*_{\text{in}}, l_{\text{out}}, s_{\text{out}}) \le \sigma(\text{sol}(D_i)^*)$.

Summing these:

$$\begin{aligned}&\text{dp}(i-1, l^*_{\text{in}}, s^*_{\text{in}}) + \text{QBP}_{D_i}(l^*_{\text{in}}, s^*_{\text{in}}, l_{\text{out}}, s_{\text{out}})\\ \le&\sigma(\text{sol}(G_{i-1})^*) + \sigma(\text{sol}(D_i)^*)\\ =&\sigma(\text{sol}(G_i)^*)\end{aligned}$$

The DP recurrence minimizes over all boundary states:

$$\text{dp}(i, l_{\text{out}}, s_{\text{out}}) \le \text{dp}(i-1, l^*_{\text{in}}, s^*_{\text{in}}) + \text{QBP}_{D_i}(\dots) \le \sigma(\text{sol}(G_i)^*)$$

Thus $\text{dp}(i, l_{\text{out}}, s_{\text{out}}) \le \sigma(\text{sol}(G_i)^*)$, contradicting the assumption that $\text{sol}(G_i)^*$ is strictly better. □

G Lowering ILP Solutions to CKKS IR

In this section, we formalize the procedure for lowering a valid ILP solution into an executable CKKS IR. Given an input ciphertext state $(l_{\text{in}}, s_{\text{in}})$ and a target output state $(l_{\text{out}}, s_{\text{out}})$, determine the optimal sequence of ciphertext management operations to transform $(l_{\text{in}}, s_{\text{in}}) \to (l_{\text{out}}, s_{\text{out}})$ such that the total cost is minimized.

Our approach is grounded in two key observations derived from our cost model. First, operation costs are monotonic with respect to level; performing an operation at a lower level is strictly cheaper than at a higher level (i.e., $\forall l < l', \text{cost}(\texttt{op}, l) \le \text{cost}(\texttt{op}, l')$), as lower levels involve fewer RNS moduli. Second, there exists a strict hierarchy in operation costs: $\text{cost}(\texttt{bootstrap}) \gg \text{cost}(\texttt{rescale}) \gg \text{cost}(\texttt{upscale}) \gg \text{cost}(\texttt{modswitch})$. Thus, we adopt a *lexicographical optimization strategy* by minimizing the cost of bootstraps, followed hierarchically by rescales, upscales, and modulus switches.

G.1 Rescale Inference

We first address transitions that do not require bootstrapping.

Lemma G.1 (Minimum Rescale Count)**.** Assume a transition where $l_{\text{in}} \ge l_{\text{out}}$ and the scale invariant holds: $s_{\text{in}} - S_f \cdot l_{\text{in}} \le s_{\text{out}} - S_f \cdot l_{\text{out}}$. Any valid transition without bootstrapping requires at least $n = \lceil (s_{\text{in}} - s_{\text{out}})/S_f \rceil$ rescale operations. A sequence of exactly n rescales is sufficient to realize this transition.

Proof. **Necessity:** Suppose, for the sake of contradiction, that the transition uses $m < n$ rescale operations. Since `upscale` and `modswitch` cannot decrease the scale, the final scale would be at least $s_{\text{in}} - m \cdot S_f$. Given $m \le n-1$, this results in a final scale strictly greater than s_{out}, rendering the transition to $(l_{\text{out}}, s_{\text{out}})$ impossible without bootstrapping.

Sufficiency: We construct a valid transition using exactly n rescales as follows:

1. Apply $l_{\text{in}} - l_{\text{out}} - n$ `modswitch` operations (if $l_{\text{in}} - l_{\text{out}} - n > 0$), resulting in state $(l_{\text{out}} + n, s_{\text{in}})$.
2. Apply n `rescale` operations, resulting in state $(l_{\text{out}}, s_{\text{in}} - n \cdot S_f)$.
3. Since $s_{\text{in}} - n \cdot S_f \le s_{\text{out}}$, apply one `upscale` operation (if needed) to increase the scale by the residual amount $s_{\text{out}} - (s_{\text{in}} - n \cdot S_f)$, reaching the target $(l_{\text{out}}, s_{\text{out}})$.

□

This construction aligns with our lexicographical strategy: it utilizes the minimum theoretical number of rescales (n), and performs them at the lowest possible levels ($l_{\text{out}}, \dots, l_{\text{out}} + n - 1$) to minimize computational cost.

G.2 Bootstrap Inference

Next, we determine the conditions necessitating a bootstrap operation.

Lemma G.2 (Bootstrap Necessity)**.** A bootstrap operation is required for the transition $(l_{\text{in}}, s_{\text{in}}) \to (l_{\text{out}}, s_{\text{out}})$ if and only if $l_{\text{in}} < l_{\text{out}}$ or $s_{\text{in}} - S_f \cdot l_{\text{in}} > s_{\text{out}} - S_f \cdot l_{\text{out}}$.

Proof. **Sufficiency ($\Rightarrow$):** By Lemma G.1, if the conditions for a bootstrap-free transition are met, a bootstrap is not required.

Necessity ($\Leftarrow$): We observe the invariant $I(l, s) = s - S_f \cdot l$. The `rescale` operation preserves I, while `modswitch` and `upscale` can only increase I. Thus, without bootstrapping, I is non-decreasing. If $s_{\text{in}} - S_f \cdot l_{\text{in}} > s_{\text{out}} - S_f \cdot l_{\text{out}}$, the target state has a strictly lower invariant value, implying the transition is impossible without bootstrapping. Similarly, without bootstrapping, levels are non-increasing; thus $l_{\text{in}} < l_{\text{out}}$ also necessitates bootstrapping. □

Transition with Bootstrapping: If a bootstrap is required, we construct the transition as follows:

1. Transform $(l_{\text{in}}, s_{\text{in}}) \to (1, S_f)$ using the non-bootstrapping method (valid since $1 \le l_{\text{in}}$).
2. Perform `bootstrap` with target level l_{out}, resulting in (l_{out}, S_f).

3. Transform $(l_{\text{out}}, S_f) \to (l_{\text{out}}, s_{\text{out}})$ using at most one `upscale`.

This construction guarantees minimal bootstrapping cost by invoking at most one bootstrap operation only when strictly necessary, and by selecting the lowest permissible target level for bootstrapping. This approach is consistent with our lexicographical optimization strategy, which prioritizes minimizing the number and cost of bootstraps above all other operations.

G.3 Consistency with ILP Formulation

We now show that the constraints imposed by our ILP formulation guarantee the existence of valid lowering sequences.

Lemma G.3. If the ILP solution satisfies $l_{\text{in}} \geq l_{\text{out}} + n$ and $s_{\text{in}} \leq s_{\text{out}} + S_f \cdot n$ for some integer $n \geq 0$, then the transition is feasible without bootstrapping.

Proof. The condition $l_{\text{in}} \geq l_{\text{out}} + n$ implies $l_{\text{in}} \geq l_{\text{out}}$ (since $n \geq 0$). Multiplying the first inequality by $-S_f$ and adding it to the second yields $s_{\text{in}} - S_f \cdot l_{\text{in}} \leq s_{\text{out}} - S_f \cdot l_{\text{out}}$. Thus, the preconditions of Lemma G.1 are met. □

Lemma G.4. If the ILP solution satisfies $s_{\text{in}} \leq S_f \cdot l_{\text{in}}$ and $s_{\text{out}} \geq S_f$, then the transition is feasible with exactly one bootstrap.

Proof. The condition $s_{\text{in}} \leq S_f \cdot l_{\text{in}}$ ensures the transition $(l_{\text{in}}, s_{\text{in}}) \to (1, S_f)$ is valid (as $s_{\text{in}} - S_f \cdot l_{\text{in}} \leq S_f - S_f \cdot 1 = 0$). The condition $s_{\text{out}} \geq S_f$ ensures the transition $(l_{\text{out}}, S_f) \to (l_{\text{out}}, s_{\text{out}})$ is valid via upscaling. Combined with the bootstrap operation, the full sequence is valid. □

Theorem G.1. Given a FHE Ciphertext Management IR satisfying the ILP constraints in Figure 7, there exists a constructive mapping to a valid CKKS IR.

Proof. We construct the CKKS IR by generating transitions for every vertex $v \in V$ (mapping $v_{\text{in}} \to v_{\text{out}}$) and every edge $e \in E$ (mapping $e_{\text{in}} \to e_{\text{out}}$).

Vertices ($v \in V$): If the ILP selects the rescale path, the constraints enforce the conditions of Lemma G.3, guaranteeing a valid rescale-based transition. Otherwise, $v.\texttt{bootstrap} = 1$, the constraints enforce $v_{\text{in}}.\texttt{s} \leq S_f \cdot v_{\text{in}}.\texttt{l}$ and $v_{\text{out}}.\texttt{s} \geq S_f$. By Lemma G.4, a valid bootstrap-based transition exists.

Edges ($e \in E$): The ILP rescaling constraints directly ensure the conditions of Lemma G.3, allowing a valid transition from e_{in} to e_{out} without bootstrapping.

Interconnections: The vertex-edge consistency constraints ensure $(u_{\text{out}}.\texttt{l}, u_{\text{out}}.\texttt{s}) = (e_{\text{in}}.\texttt{l}, e_{\text{in}}.\texttt{s})$ for all $e = (u, v)$, requiring no additional operations. Similarly, computational constraints ensure compatibility between edges and vertices.

In conclusion, valid transition sequences exist for all components, yielding a globally valid CKKS IR. □



USENIX Security '26 Artifact Appendix: Orbit: Optimizing Rescale and Bootstrap Placement with Integer Linear Programming Techniques for Secure Inference

Zikai Zhou*† William Seo*‡ Edward Chen‡ Alex Ozdemir§ Fraser Brown‡ Wenting Zheng‡
†Tsinghua University ‡Carnegie Mellon University §Max Planck Institute for Security and Privacy
zhouzk22@mails.tsinghua.edu.cn, {wys, ejchen, fraserb, wenting}@cmu.edu, alex.ozdemir@mpi-sp.org

A Artifact Appendix

A.1 Abstract

This artifact contains Orbit, an FHE compiler that jointly optimizes bootstrap and rescale placement for CKKS secure-inference workloads through a level-scale-aware integer linear programming (ILP) formulation. It includes the Orbit implementation (about 3,500 lines of Python and 2,500 of Go), MLIR inputs for five CNNs (ResNet-20, SqueezeNet, AlexNet, MobileNet, VGG16) in SiLU and ReLU variants, profiled Lattigo cost models, and scripts that reproduce the paper's experiments. The artifact packages Orbit itself; the baseline placement algorithms (DaCapo [3], Orion [4], ReSBM [6]) are not included, so it reproduces Orbit's own results and compares them against the baseline numbers reported in the paper. Reviewers can compile every benchmark and check Orbit's compilation times (Tables 6–7), estimate or execute the compiled programs to check Orbit's latencies and operation counts (Figures 5–6, Table 8), and verify accuracy within 0.3% of plaintext PyTorch (Table 9). All experiments target CKKS in Lattigo on a standard Linux machine.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None. All experiments are purely computational, run in user space, and use only synthetic inputs and public CIFAR-10 samples.

A.2.2 How to access

An archived version matching the evaluated paper is on Zenodo (`https://zenodo.org/records/20278435`); ongoing development is on GitHub (`https://github.com/cmu-cryptosystems/Orbit`).

*Equal contribution.

A.2.3 Hardware dependencies

Orbit needs only a commodity CPU and under 5 GB of RAM to compile. To reproduce the paper we recommend a high core count (QBP solving is parallelized) and, for executing the larger benchmarks, up to 256 GB of RAM; the paper compiled on AWS `m8i.8xlarge` (32 cores) and executed on `m7g.16xlarge` (64 cores, 256 GB). Allow about 20 GB of disk for compilation and one benchmark's execution data; the plaintext constants for *all* benchmarks total order 50 GB.

A.2.4 Software dependencies

Orbit was tested with Python 3 (the Docker image pins Python 3.11 on Debian, with Go 1.24 for the backend). Python dependencies (`gurobipy`, `joblib`, `matplotlib`, `more-itertools`, `networkx`, `numpy`) are in `requirements.txt` and installable with `pip`. The `gurobipy` package ships a size-limited license sufficient for every standard (partitioned) benchmark; only the `-nopart` and `-nocomp` ablations exceed its cap and need a full (e.g., free academic) Gurobi license. The DaCapo frontend (generates plaintext inputs) and Lattigo backend (executes compiled MLIR) are needed only for execution; **reviewers checking only estimated results need neither.** See `frontend/README.md` and `backend/README.md`, or build both inside the container with `scripts/setup_dependencies.sh`.

A.2.5 Benchmarks

Five CNNs used by prior FHE compilers—ResNet-20, SqueezeNet, AlexNet, MobileNet, VGG16—each in a SiLU variant (single-Chebyshev SiLU, average-pooling) and a ReLU variant (three-Chebyshev ReLU, max-pooling; pooling only in AlexNet, SqueezeNet, VGG16). They are MLIR files in `mlirs_input/`, lowered by DaCapo into Orbit's FHE Computation IR (the paper's shared input representation). Accuracy is validated against plaintext PyTorch on CIFAR-10.

A.3 Set-up

A.3.1 Installation

Install the Python dependencies:

```
git clone \
  https://github.com/cmu-cryptosystems/Orbit
cd Orbit && pip install -r requirements.txt
```

To execute (not only estimate) programs, also install the frontend and backend (their READMEs), or use the Docker image below—which needs no manual Python or Gurobi setup.

A.3.2 Basic Test

Run `pytest`, then compile one benchmark end to end:

```
python3 scripts/optimizer/orbit/run_orbit.py \
  --model ResNet --act SiLU \
  --n 64 --Lm 16 --Sw 40
```

Output lands under `mlirs_output/`: the `.txt` log reports `Orbit Compilation time` and the estimated `Final tdag latency`; the `.mlir` is the compiled program.

A.3.3 Reproducing the experiments with Docker

Build the image once and drive it with `scripts/reproduce.sh`; the image bundles the dependencies and the size-limited Gurobi license, so the cost-model experiments (E1, E3) need no further setup:

```
docker build --build-arg USER_ID=$(id -u) \
    --build-arg GROUP_ID=$(id -g) -t orbit:dev .
docker run --rm orbit:dev   # tests+smoke
```

The experiments below run `reproduce.sh` targets through this image; mounting the repository with `-v $PWD:/opt/orbit` keeps generated files (compiled MLIRs and logs) on the host. The `-nopart`/`-nocomp` CompPart ablations additionally need a full Gurobi license, added with `-v $HOME/gurobi.lic:/opt/gurobi/gurobi.lic:ro` and `-e GRB_LICENSE_FILE=/opt/gurobi/gurobi.lic`. To execute (E2, E4), first build the backend and frontend and generate inputs once, then use the `execute` target:

```
D="docker run --rm -e HOME=/tmp \
   -v $PWD:/opt/orbit"
$D orbit:dev \
    ./scripts/setup_dependencies.sh backend
$D orbit:dev \
    ./scripts/setup_dependencies.sh frontend
$D -w /opt/orbit/frontend/dacapo orbit:dev \
    bash -c 'source .venv/bin/activate && \
    ../dacapo_patch/gen_all_mlirs.sh && \
    python3 examples/tests/gen_input_data.py 10'
$D orbit:dev ./scripts/reproduce.sh execute \
    --model ResNet --act SiLU \
    --n 16 --Lm 16 --Sw 40 --run 0
```

Decrypted output lands in `mlirs_execute/`, comparable to the PyTorch reference from the data-generation step (E4).

A.4 Evaluation workflow

A.4.1 Major Claims

The baselines (DaCapo, Orion, ReSBM) are **not** part of this artifact; each claim is reproduced for Orbit, and the paper computes the comparisons against the reported baseline numbers.

(C1): Orbit compiles every benchmark and configuration in under 6 minutes; the reproduced times are the Orbit columns of Tables 6–7. Verified by E1.

(C2): Orbit's joint bootstrap-and-rescale ILP lowers execution cost: its compiled programs use up to 5.4× fewer rescales and 31–34% fewer bootstraps than Orion (Figures 5–6, Table 8), yielding the paper's speedups of 7–19% over DaCapo, 2–73% over Orion, and up to 52% over ReSBM. Verified by E2.

(C3): Orbit's DAG-reduction techniques (compression, partitioning, QBP reuse, bypass handling) cut ILP solving time while keeping end-to-end latency within 1% (Tables 3–5). Verified by E3.

(C4): Orbit stays within 0.3% of plaintext PyTorch accuracy across all benchmarks (Table 9). Verified by E4.

A.4.2 Experiments

Each suite below is shown as its native script `auto_scripts/<name>.sh`; equivalently, run it through the Docker image as `reproduce.sh <t>`—short for `docker run -rm -v $PWD:/opt/orbit orbit:dev ./scripts/reproduce.sh <t>`—whose targets mirror the script names (e.g., `compile_orbit_eval_base.sh` ≡ `compile-base`). Execution (E2, E4) additionally needs `-e HOME=/tmp` and the one-time backend/frontend build from Set-up. Below, the instructions use the commands for compilation and execution in Docker.

(E1): Compilation time [15 human-minutes + several compute-hours]. Compile the four configuration suites (base n=64k, plus 16k, L_{bts}=12, and S_{min}=51):

```
reproduce.sh compile-base
reproduce.sh compile-16k
reproduce.sh compile-lm12
reproduce.sh compile-sw51
```

Each `.txt` log under `mlirs_output/` reports `Orbit Compilation time`: on the recommended multi-core machine every benchmark compiles in under 6 minutes, matching the Orbit columns of Tables 6–7. Compilation parallelism follows `ORBIT_THREADS` (default 32; set it to the machine's core count).

(E2): Runtime performance [10 human-minutes + ∼1-hour setup + minutes–hours per benchmark]. Install the frontend and backend (or use Docker) to execute, or skip for estimates only. Compile as in E1, then run each program through the `execute` target (the command shown in Set-up) on Lattigo; `compile-sim-vari` selects the variable-bootstrapping cost model for the ReSBM comparison. Compare Orbit's latencies and the rescale/bootstrap counts printed in each `.txt` log against the Orbit data in Figures 5–6 and Table 8. Building the frontend/backend and generating inputs is a one-time ∼1-hour step; plaintext evaluation then takes seconds to minutes (∼10 s for AlexNet-SiLU at 16k) and encrypted evaluation minutes to hours (∼25 min for AlexNet-SiLU at 16k; the 64k benchmarks run for hours and need ∼256 GB RAM).

(E3): DAG-reduction microbenchmarks [10 human-minutes + 1–2 compute-hours]. Run the ablation suites; `-nocomp`, `-nopart`, `-nobypass`, and `-qbp` on `run_orbit.py` toggle the individual techniques:

```
reproduce.sh compile-micro-comppart
reproduce.sh compile-micro-bypass
reproduce.sh compile-micro-reqbp
```

Compare compilation times, partition counts, and latencies against Table 3 (compression/partitioning), Table 4 (QBP reuse), and Table 5 (bypass); end-to-end latency stays within 1%. Each ablation compiles in seconds to ∼30 minutes. The `-nopart`/`-nocomp` variants in `compile-micro-comppart` are the slowest and exceed the bundled Gurobi license [5], so they require a full license (mounted as in Set-up); the other ablations run on the bundled license.

(E4): End-to-end accuracy [10 human-minutes + ∼1-hour setup + hours per benchmark]. Install the frontend and backend so CIFAR-10 inputs and PyTorch references are generated into `input_data/` (a one-time ∼1-hour build and data-generation step). Execute the compiled benchmarks at $n = 2^{17}$ on encrypted inputs and compare with the PyTorch outputs (Table 9): accuracy is within 0.3%, most models matching exactly (max difference order 10^{-4}). [1]

A.5 Notes on Reusability

Orbit is a modular pass over a backend-agnostic CKKS IR, so it retargets to other FHE libraries (e.g., SEAL, OpenFHE) via a new backend lowering and integrates into frameworks such as HEIR [1] and Rotom [2] through its MLIR interface. Compile arbitrary MLIR with `optimizer.py`; the core components—the ILP formulation (`ilp_core.py`), SISO partitioning and bypass handling (`siso_partition.py`), iterative solving and merging (`iterative_partition.py`), QBP management (`qbp_manager.py`), and compression (`auto_compression.py`)—are organized for extension, and new backend/hardware cost models drop into `cost_models/` as profiled JSON.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2026/.

[1] The accuracy for $n = 2^{15}$ varies from PyTorch execution due to bugs in Dacapo frontend. As stated in the paper, this configuration is not intended for deployment.

References

[1] Asra Ali, Jaeho Choi, Bryant Gipson, Shruthi Gorantala, Jeremy Kun, Wouter Legiest, Lawrence Lim, Alexander Viand, Meron Zerihun Demissie, and Hongren Zheng. Heir: A universal compiler for homomorphic encryption. *arXiv preprint arXiv:2508.11095*, 2025.

[2] Edward Chen, Fraser Brown, and Wenting Zheng. Bridging usability and performance: A tensor compiler for autovectorizing homomorphic encryption. *Cryptology ePrint Archive*, 2025.

[3] Seonyoung Cheon, Yongwoo Lee, Dongkwan Kim, Ju Min Lee, Sunchul Jung, Taekyung Kim, Dongyoon Lee, and Hanjun Kim. {DaCapo}: Automatic bootstrapping management for efficient fully homomorphic encryption. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6993–7010, 2024.

[4] Austin Ebel, Karthik Garimella, and Brandon Reagen. Orion: A fully homomorphic encryption framework for deep learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 734–749, 2025.

[5] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2026.

[6] Yan Liu, Jianxin Lai, Long Li, Tianxiang Sui, Linjie Xiao, Peng Yuan, Xiaojing Zhang, Qing Zhu, Wenguang Chen, and Jingling Xue. Resbm: Region-based scale and minimal-level bootstrapping management for fhe via min-cut. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 924–939, 2025.