Orion: A Fully Homomorphic Encryption Framework for Deep Learning

Austin Ebel*
New York University
Brooklyn, NY, USA
abe5240@nyu.edu

Karthik Garimella*
New York University
Brooklyn, NY, USA
kg2383@nyu.edu

Brandon Reagen
New York University
Brooklyn, NY, USA
bjr5@nyu.edu

Abstract

Fully Homomorphic Encryption (FHE) has the potential to substantially improve privacy and security by enabling computation directly on encrypted data. This is especially true with deep learning, as today, many popular user services are powered by neural networks in the cloud. One of the major challenges facing wide-scale deployment of FHE-secured neural inference is effectively mapping these networks to FHE primitives. FHE poses many programming challenges including packing large vectors, automatically managing noise via bootstrapping, and translating arbitrary and general-purpose programs to the limited instruction set provided by FHE. These challenges make building large FHE neural networks intractable using the tools available today.

In this paper we address these challenges with **Orion**, a fully-automated framework for private neural inference in FHE. Orion accepts deep neural networks written in PyTorch and translates them into efficient FHE programs. We achieve this by proposing a novel single-shot multiplexed packing strategy for arbitrary convolutions and through a new, efficient technique to automate bootstrap placement. We evaluate Orion on common benchmarks used by the FHE deep learning community and outperform state-of-the-art by 2.38× on ResNet-20, the largest network they report. Orion extends naturally to larger networks. We demonstrate this by evaluating ResNet-50 on ImageNet and present the first high-resolution homomorphic object detection experiments using a YOLO-v1 model with 139 million parameters. Finally, we open-source our framework Orion at the following repository: https://github.com/baahl-nyu/orion.

1 Introduction

Fully Homomorphic Encryption (FHE) is a powerful encryption scheme that allows for computation to be performed directly on encrypted data without ever needing to decrypt. FHE provides a solution to the privacy concerns of outsourced cloud computation by allowing personal and sensitive data to be first encrypted under FHE before being processed securely in the cloud. In this manner, both the original data and any intermediate results remain encrypted and can only be observed in the clear when the data owner decrypts it locally using their secret key. FHE has broad

implications for areas such as finance or health and other situations involving sensitive data. Many of these services are now driven by deep learning, specifically neural networks.

Despite this, the wide-scale deployment of FHE-enabled private deep learning remains limited because of 1) the significant computational costs of FHE and 2) the programming challenges of translating standard, unencrypted neural networks into FHE programs. While hardware accelerators have addressed this performance gap [23, 26, 40, 41, 55, 58, 61] and FHE compilers have made progress in translating workloads into FHE programs [18, 20, 21, 42, 51, 64], it remains a challenge to run FHE neural inference on large datasets in a high-level deep learning framework (e.g., ImageNet inference in PyTorch). This paper focuses on efficiently and *automatically* mapping neural networks, as implemented in modern deep learning libraries, to FHE programs. Multiple compounding factors make effectively mapping neural networks to FHE challenging.

Data packing: FHE encrypts large vectors that vary in length from 2^{14} to 2^{17}, with recent work favoring the larger lengths [16, 41, 61]. Scalar data must be packed into the *slots* of these vectors and FHE only allows SIMD addition, SIMD multiplication, and cyclic rotation upon these encrypted vectors. Individual encrypted elements cannot be indexed, and any function applied to a vector is applied to all its elements. Therefore, it is crucial to optimize data packing and layout in FHE vectors to efficiently make use of the available slots and leverage this SIMD property.

Bootstrap placement: Each encrypted vector can only execute a finite number of computations before decryption fails. To prevent this, bootstrapping can be used to reset this computational budget but comes at an extremely high latency cost [10]. Effectively inserting bootstrap operations into an FHE program is challenging; bootstrapping at any point in an FHE circuit directly affects the latency and resource requirements of subsequent operations, including future bootstraps. Therefore, an efficient solution to bootstrap placement requires a holistic understanding of the entire FHE program.

Programmability: FHE is notoriously hard to program. Every program must be translated into a series of SIMD additions, multiplications, and cyclic rotations on very large vectors. This translation is especially challenging for deep learning because 1) convolutional and fully-connected layers

---

must be implemented using just these three FHE operations and 2) element-wise nonlinear activation functions must be approximated with high-degree polynomials. Furthermore, most existing FHE libraries are not designed with machine learning in mind and are cumbersome for building large FHE programs. Finally, several auxiliary FHE operations (e.g. rescaling, bootstrapping, level adjustment) must be manually inserted into FHE programs making it hard for researchers and practitioners to immediately make use of FHE.

To address these challenges, we develop and open-source **Orion**: a fully-automated framework for private neural inference that advances the state-of-the-art in all three areas. For data packing, we introduce a novel packing strategy for convolutions that we call *single-shot multiplexing*. This approach directly improves upon the multiplexed packing technique of Lee et al. [48] in three ways: (1) it halves the multiplicative depth of every convolution from two to one, (2) it supports convolutions with arbitrary parameters (e.g., stride, padding, groups, dilation), and (3) it reduces (expensive) ciphertext rotations in modern neural networks by up to 6.41×.

We then propose a fully-automated and efficient solution for placing bootstrap operations within a neural network. Our approach uses a linear time topological sort to determine the optimal placement of bootstraps. For example, it takes Orion just 6.67 (49.4) seconds to place bootstraps in ResNet-20 (ResNet-110) and requires no user input. Our algorithm scales linearly with network depth and results in a similar number of bootstraps to Dacapo [16] while also being up to 255× faster. Notably, our method also places just 351 bootstrap operations in a ResNet-50 evaluated on ImageNet, whereas 8480 bootstraps are placed in HeLayers [6]. For ResNet-20, Fhelipe [42] performs 58 bootstraps, whereas we have only 37.

Finally, we design Orion to be accessible to machine learning researchers and practitioners who are unfamiliar with FHE. Orion inherits directly from PyTorch and extends the functionality of its most popular CNN layers. For instance, users can train Orion networks with existing PyTorch training scripts and directly load the weights of pre-trained models with `torchvision`. Orion's interoperability with PyTorch allows us to easily verify that our FHE outputs match the outputs of running inference directly in PyTorch.

We evaluate Orion using a set of neural networks and datasets and report significant improvements over state-of-the-art [16, 42, 48]. To the best of our knowledge, we are the first to make the following contributions:

1. We develop and open-source Orion: a fully-automated framework for running FHE neural network inference directly in PyTorch. Using Orion on the standard ResNet-20 FHE benchmark, we achieve a speedup over state-of-the-art (Fhelipe [42]) by 2.38×. We also run ResNet-50 on ImageNet with no code changes, which most prior work does not readily support.

2. We implement the *single-shot multiplexed* packing strategy in Orion and support arbitrary convolutions while only having a multiplicative depth of one.

3. We implement our automatic bootstrap placement algorithm in Orion, which requires no user input, and matches state-of-the-art in terms of number of bootstraps while performing up to 255× faster.

4. We are the first to perform high-resolution object detection using a YOLO-v1 model with 139 million parameters on an image of size $448 \times 448 \times 3$ [56].

2 CKKS Background

In this section, we describe the CKKS homomorphic encryption scheme that is used in Orion. At a high level, CKKS encrypts large vectors of complex (or real) numbers and enables three operations on these vectors: element-wise addition, element-wise multiplication, and cyclic rotation [14]. These properties make CKKS a natural choice for applications that operate on real-valued vectors such as deep learning. Below, we detail the fundamental datatypes and operations in CKKS.

2.1 Datatypes

The three datatypes in CKKS are *cleartexts*, *plaintexts*, and *ciphertexts*. A cleartext is an unencrypted vector of complex (or real) numbers. The process of encoding converts a cleartext into a plaintext, which is still unencrypted and is an element of the polynomial ring $\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^N + 1)$. Here, N is a power of two and each coefficient is an integer modulo Q. In this way, a plaintext is a polynomial up to degree $N - 1$ and each coefficient is in $\mathbb{Z}_Q$.

A plaintext can then be encrypted into a ciphertext which consists of two polynomials (i.e., an element in $\mathcal{R}_Q \times \mathcal{R}_Q$). To maintain 128-bit security and enable deep arithmetic computations in CKKS, it is common for N to be between 2^{14} to 2^{17} and Q to be on the order of hundreds to thousands of bits [3]. In practice, these constraints mean plaintexts and ciphertexts are KBs to MBs in size. CKKS performs polynomial operations on both plaintexts and ciphertexts that correspond to SIMD addition, SIMD multiplication, and cyclic rotations on the underlying cleartext vectors.

2.2 Encoding: Cleartext → Plaintext

Encoding converts a cleartext vector $\mathbf{m} \in \mathbb{C}^{N/2}$ into a plaintext polynomial $[\mathbf{m}] \in \mathcal{R}_Q$ by 1) applying an inverse Fast Fourier Transform (iFFT) on the cleartext values, 2) multiplying each output by a scaling factor Δ, and 3) rounding each element to the nearest integer. In this manner, one can *pack* $n = N/2$ complex values into a single plaintext. If the underlying data consists only of real numbers (such as our use case), it is possible to pack $n = N$ real values into a single plaintext. Decoding converts a plaintext polynomial back into a cleartext vector by performing the FFT on the polynomial coefficients and dividing each value by the scaling factor Δ. Other encoding and decoding schemes for CKKS exist,

but do not preserve the SIMD property on the underlying cleartext values (see Section IV.A of [37]).

2.3 Encryption: Plaintext → Ciphertext

A plaintext $[\mathbf{m}] \in \mathcal{R}_Q$ is encrypted into a ciphertext $[[\mathbf{m}]] \in \mathcal{R}_Q \times \mathcal{R}_Q$ by adding noise $[\mathbf{e}] \in \mathcal{R}_Q$ to the plaintext and encrypting via either the secret or public key. Decryption can only occur via the secret key. Both rounding during encoding and the addition of noise during encryption introduce small errors, and so CKKS is an approximate homomorphic encryption scheme. However, these approximations tend to be tolerable for deep learning.

2.4 RNS-CKKS

Each coefficient of a CKKS polynomial can be up to thousands of bits wide (e.g. $\log_2 Q \approx 1500$), making operations on these large coefficients compute intensive. Therefore, polynomials are decomposed into $L+1$ *residual* polynomials, each with small coefficients that fit within a machine-sized word. Now, the coefficients of a CKKS polynomial can be viewed as matrix of size $(L + 1) \times N$ where each row of the matrix is an N-degree residual polynomial with small coefficients.

This process occurs via the Residual Number System (RNS) in which the large modulus Q is decomposed into $L+1$ smaller moduli $Q = \prod_{i=0}^{L} q_i$. Using RNS, a coefficient $x \in \mathbb{Z}_Q$ can be represented using $L + 1$ limbs as $x := (x_0, \ldots, x_L)$ where $x_i = x \bmod q_i$. Modular addition (multiplication) between two large elements in $\mathbb{Z}_Q$ is equivalent to performing modular addition (multiplication) on each limb independently [28].

The integer L is known as the maximum multiplicative level of the polynomial, and as we will see, homomorphic multiplications *consume* levels. At any given time, a ciphertext can be at level ℓ, such that $0 \leq \ell \leq L$, and a ciphertext at level ℓ consists of $\ell + 1$ residual polynomials. When a ciphertext reaches level $\ell = 0$, it has depleted its multiplicative budget. A *bootstrap* procedure can be used to increase its level up to L_{eff} to perform further multiplications.

2.5 CKKS Operations

We now describe the primitive CKKS operations: addition, multiplication, rotation, and bootstrapping. CKKS performs modular arithmetic on plaintext and ciphertext polynomials to implement these operations. For fast polynomial multiplication, the Number Theoretic Transformation (NTT) is used to convert the polynomials from the coefficient to the evaluation representation to reduce the running time from $O(N^2)$ to $O(N \log N)$. For this reason, it is common to keep all plaintexts and ciphertexts in the evaluation representation unless otherwise needed. For this section, we will assume we have two cleartext vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$ that we encode (as plaintexts $[\mathbf{a}]$ and $[\mathbf{b}]$) both with a scaling factor Δ and level ℓ, and further encrypt (as ciphertexts $[[\mathbf{a}]]$ and $[[\mathbf{b}]]$).

2.5.1 Addition. CKKS supports point-wise addition between one plaintext and one ciphertext (PAdd $([\mathbf{a}], [[\mathbf{b}]])$) or between two ciphertexts ((HAdd $([[\mathbf{a}]], [[\mathbf{b}]])$). Each operand must be at the same number of levels and have the same scaling factor. In both PAdd and HAdd, the output is a ciphertext $[[\mathbf{c}]]_{\text{add}}$ with the same level and scale as the input operands. This ciphertext corresponds to the SIMD addition of the underlying cleartexts: $\text{decode}(\text{decrypt}([[\mathbf{c}]]_{\text{add}})) \approx \mathbf{a} + \mathbf{b}$.

2.5.2 Multiplication. Similar to addition, CKKS supports point-wise multiplication between one plaintext and one ciphertext (PMult $([\mathbf{a}], [[\mathbf{b}]])$) or between two ciphertexts (HMult $([[\mathbf{a}]], [[\mathbf{b}]])$). Each operand must be at the same number of levels, but not necessarily the same scaling factor. Both PMult and HMult result in a ciphertext $[[\mathbf{c}]]_{\text{mult}}$ with a scaling factor Δ^2. To prevent exponential growth of the scaling factor, CKKS supports a rescaling procedure that divides the scaling factor by the last prime limb, q_ℓ, and by choosing each $q_i \approx \Delta$, the scaling factor remains roughly consistent throughout the computation. Rescaling also removes the last prime limb from each polynomial within $[[\mathbf{c}]]_{\text{mult}}$, effectively reducing the level of the output ciphertext by one.

In addition to rescaling, HMult requires an expensive key-switching operation (that involves auxiliary *evaluation keys*) to ensure correct decryption. Key-switching itself is a computationally expensive procedure involving many NTTs and RNS basis conversions [36]. After rescaling (and key-switching for HMult), the output is a ciphertext $[[\mathbf{c}]]_{\text{mult}}$ at level $\ell - 1$ and a scaling factor of $\Delta^2 / q_\ell \approx \Delta$. This ciphertext corresponds to the SIMD multiplication of the underlying cleartexts: $\text{decode}(\text{decrypt}([[\mathbf{c}]]_{\text{mult}})) \approx \mathbf{a} \cdot \mathbf{b}$.

2.5.3 Rotation. Besides addition and multiplication, CKKS also supports the cyclic rotation (HRot$_k$ $([[\mathbf{a}]])$) of the underlying cleartext slots by an amount $0 < k < n$. Similar to HMult, HRot also requires a key-switching step using *rotation keys* so that decryption of the rotated ciphertext is correct. The resulting ciphertext $[[\mathbf{c}]]_{\text{rot}}$ has the same level and scale as the input ciphertext and corresponds to the underlying cleartext vector cyclically rotated "up" by k slots: $\text{decode}(\text{decrypt}([[\mathbf{c}]]_{\text{rot}})) \approx (a_k, a_{k+1}, \ldots, a_{k-2}, a_{k-1})$.

2.5.4 Bootstrapping. Finally, for a scheme to be fully homomorphic, it must include a way of increasing the number of remaining levels; the bootstrap operation provides this functionality. Bootstrapping is a computationally demanding procedure that increases levels but also consumes a fixed number (L_{boot}) of levels in the process. A ciphertext that begins at level L can only reach an effective level, $L_{\text{eff}} = L - L_{\text{boot}}$ after bootstrapping. A typical L_{boot} is 15 levels [1, 10].

3 FHE Matrix-Vector Products

In this section, we introduce how efficient homomorphic matrix-vector products are performed within Orion. We focus on the *diagonal method* of encoding matrices, and its state-of-the-art optimizations [10] which add both the baby-step giant-step (BSGS) and double-hoisting algorithms. We use these optimizations in *every* linear transformation in Orion including all linear and convolutional layers of neural

networks and the matrix-vector products within bootstrapping. Similar to prior work, and in alignment with our threat model, we assume that the matrix is a cleartext that can be pre-processed before being encoded, and that the vector is a ciphertext.

3.1 Diagonal Encoding Method

The diagonal method [30] is a straightforward technique for performing homomorphic matrix-vector products. This method works by first extracting the *generalized diagonals* of a matrix $\mathbf{M}$, defined as $\text{diag}_k = \mathbf{M}_{[0,k]}, \mathbf{M}_{[1,k+1]}, \ldots, \mathbf{M}_{[w-1,k+w-1]}$, where w is the matrix width and the second index is taken modulo w. Each diagonal diag_k is then multiplied by the input ciphertext, rotated up by k slots, to produce each partial product. Summing all partial products gives the final result. Figure 1a illustrates the diagonal method for a 6×6 matrix. Notably, this method requires n ciphertext rotations to perform any $n \times n$ matrix-vector product. Thus, its runtime complexity can be seen as $O(n)$.
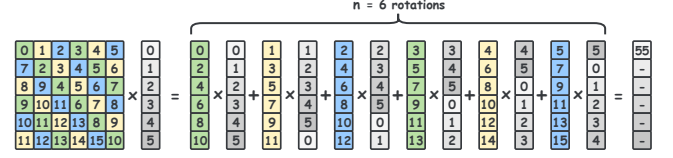
3.2 Baby-Step Giant-Step Optimization

The baby-step giant-step (BSGS) algorithm reduces the runtime complexity of homomorphic matrix-vector products to $O(\sqrt{n})$ by leveraging the fact that matrix diagonals can be cheaply rotated before being encoded. More specifically, this algorithm decomposes a matrix-vector product into n_1 groups of n_2 diagonals. Each group shares a common (baby-step) ciphertext rotation, and each diagonal within a group is rotated by a different multiple of n_1. The standard diagonal method is then applied across groups to produce n_2 partial products, each offset by a multiple of n_1. Thus, n_2 (giant-step) ciphertext rotations are required to realign partial products before they are summed to produce the desired result. Equation (1) defines the BSGS algorithm, where $\widetilde{\text{diag}}_k = \text{Rot}_{-j \cdot n_1}(\text{diag}_k)$, the k'th generalized diagonal of an $n \times n$ matrix, cyclically rotated down by $j \cdot n_1$ slots. Figure 1b also visualizes the BSGS algorithm when $n_1 = 3$, $n_2 = 2$, and $n_1 n_2 = 6$, where diagonals are shaded by group. Notably, one can choose any n_1, n_2 such that $n_1 n_2 = n$, however the number of ciphertext rotations is minimized when $n_1 = n_2 = \sqrt{n}$.
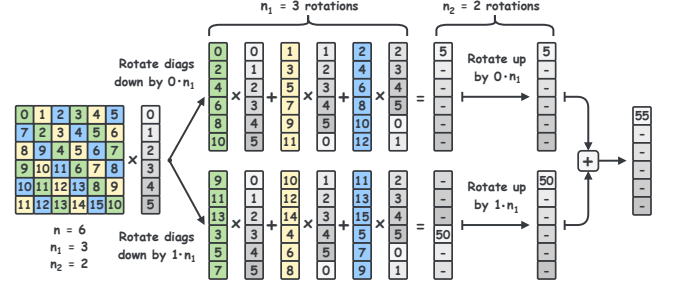
$$\text{ct}_{\text{out}} = \sum_{j=0}^{n_2-1} \text{HRot}_{j \cdot n_1} \left\{ \sum_{i=0}^{n_1-1} \text{PMult}\left(\text{HRot}_i(\text{ct}_{\text{in}}), \widetilde{\text{diag}}_{j \cdot n_1 + i}\right) \right\} \quad (1)$$

3.3 Hoisting Optimizations

At a high level, *hoisting* is a cryptographic technique that amortizes the most expensive aspects of the key-switch procedure *across* many ciphertext rotations. This optimization is only possible when rotating the *same* ciphertext by different amounts and can therefore only be applied to baby-step rotations in the BSGS algorithm. Bossuat et al. [10] separately extend this *single-hoisting* technique to giant-steps and aptly name its implementation *double-hoisting*. In Orion, every matrix-vector product uses the double-hoisting BSGS

(a) The diagonal method [30]. $n = 6$ ciphertext rotations are required, one per non-zero diagonal (including the trivial rotation by 0).

(b) Extending the diagonal method with BSGS [30]. Note that only $n_1 + n_2 = 5$ rotations are required and that $n_1 n_2 = n = 6$.

Figure 1. Visualizing how the BSGS algorithm [30] reduces the number of ciphertext rotations in matrix-vector products.

algorithm (see Algorithm 6 in [10]). We refer readers to [10, 23, 35] for a more detailed explanation of hoisting.

Takeaway: The double-hoisting BSGS method has two fundamental benefits: (1) every matrix-vector product consumes just one multiplicative level, and (2) its performance savings *increase* with increasing matrix size since BSGS decreases the number of ciphertext rotations from $O(n)$ to $O(\sqrt{n})$.
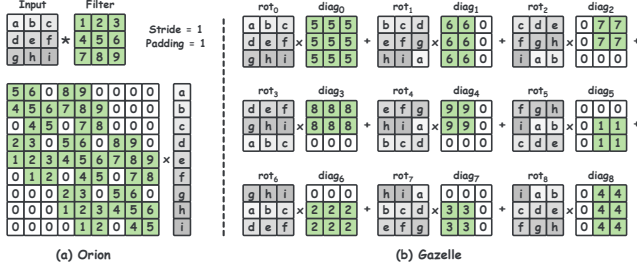
4 Efficient Convolutions in Orion

In this section, we show how Orion efficiently expresses *all* convolutions as matrix-vector products to leverage the savings from BSGS and hoisting discussed in Section 3. To achieve this, we introduce a new packing technique called *single-shot multiplexing*. This technique utilizes a modified Toeplitz formulation of convolutions and is highly effective: it supports arbitrary parameters and halves the level consumption of the multiplexed packing approach of Lee et al. [48] while also significantly reducing rotation counts.

Notation: A convolution operation processes a three dimensional input image of size $h_i \times w_i \times c_i$, representing the height, width, and number of channels of the input image, respectively. Then, c_o sets of trainable filters of size $f_h \times f_w \times c_i$ are *convolved* with the input image to produce an output image of size $h_o \times w_o \times c_o$, with parameters such as stride (s) and padding (p) controlling the output's dimensions.

Since CKKS operates on one-dimensional vectors of complex (or real) numbers, it is common to first flatten a three-dimensional input into a row-major (raster-scanned) vector of length $h_i w_i c_i$. For simplicity, we assume that the number

Figure 2. An example showing how we transform the packed SISO method from Gazelle (b) into its analogous Toeplitz matrix (a) to leverage the BSGS and hoisting optimizations.
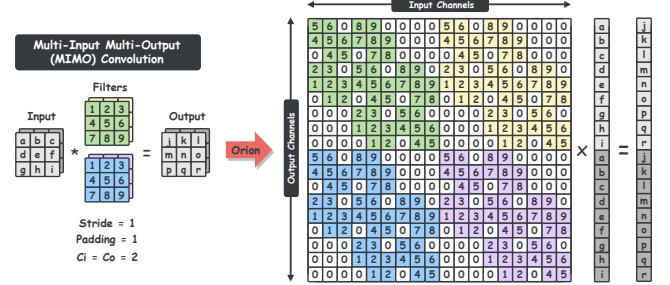

Figure 3. The Toeplitz formulation for SISO convolutions from Figure 2 generalizes to MIMO convolutions; filters for separate input (output) channels are placed across the columns (rows) of the Toeplitz matrix.

of slots, n, matches this length so that all data fits perfectly into a single ciphertext. In Section 4.3, we relax this assumption to handle inputs that can be larger than n.

4.1 SISO Convolutions in Orion

We begin with single-input, single-output (SISO) convolutions, where $c_i = c_o = s = 1$, and restrict our attention to same-style convolutions, where the input and output spatial dimensions are equal. An illustrative example is shown in Figure 2. Gazelle [35] first introduced a packed SISO method to homomorphically evaluate this convolution, and subsequent works [16, 42, 48] have continued to follow this blueprint (Figure 2b). Here, the input is flattened, packed into a ciphertext, and cyclically rotated $f_h f_w$ times. Each rotated ciphertext is then multiplied by a *punctured* plaintext, which encodes n copies of a unique filter weight and is selectively zeroed to ensure only the pixels that interact with the filter weight are processed. Finally, the partial products from each multiplication are summed to produce the final output.

Orion: Our key observation is that prior works' packed SISO method is equivalent to the diagonal method: ciphertexts are rotated and multiplied with pre-processed plaintexts. We show this by working backwards from Figure 2b to derive its analogous matrix-vector product. In doing so, we arrive at the *Toeplitz* formulation of a convolution, shown in Figure 2a. In this representation, the filter expands into a weight matrix of size $h_o w_o \times h_i w_i$, where each row corresponds to one filter multiplication. For example, the first row of the matrix in Figure 2a performs a dot product between the filter weights {5,6,8,9} and the input pixels {a,b,d,e} which corresponds to the filter being placed at the top left corner of the input with padding $p = 1$. Subsequent rows of the Toeplitz matrix are generated as the kernel slides across the input image. Performing the diagonal encoding method on the matrix in Figure 2a produces the *exact* operations seen in the packed SISO method in Figure 2b.

The fundamental benefit of this observation is that we can build a Toeplitz matrix for *any* arbitrary convolution and apply the double-hoisting BSGS algorithm when evaluating its matrix-vector product. To the best of our knowledge slytHErin [34] is the only prior work which proposes

a similar strategy. With this, Orion reduces the number of ciphertext rotations for any SISO convolution from $O(f)$ to $O(\sqrt{f})$, where f is the total number of filter elements.

4.2 MIMO Convolutions in Orion

The Toeplitz formulation extends naturally to both multiple input and multiple output (MIMO) convolutions. Figure 3 shows a MIMO convolution with $c_i = c_o = 2$, along with its analogous Toeplitz matrix. This matrix is constructed in much the same way as the SISO case. Each row represents one filter multiplication, and we can apply the diagonal method to this matrix to produce a raster-scanned output ciphertext in a single multiplicative level. Once again, this formulation lets us apply double-hoisting BSGS to significantly reduce the number of required ciphertext rotations.
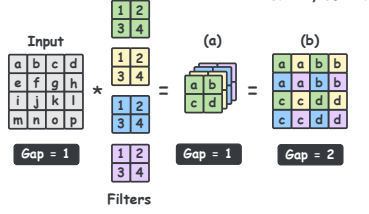
4.3 Single-Shot Multiplexed Convolutions

The Toeplitz approach works well for same-style convolutions because each consecutive row in the matrix shifts the filter by one position over the input. This keeps the same filter element aligned within each matrix diagonal and therefore ensures that the number of expensive ciphertext rotations is *independent* of the input's spatial dimensions.
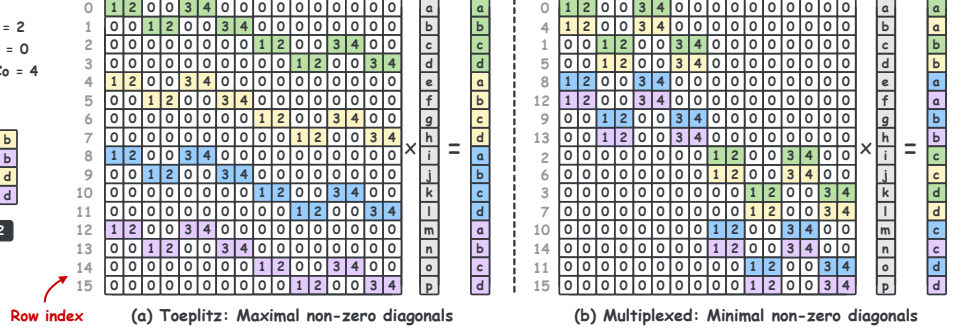
However, this convenient property breaks down when considering strided convolutions. Figure 4a illustrates why this occurs using a convolution with a stride of $s = 2$. In this case, each successive row in the matrix in Figure 4a shifts the kernel by a multiple of s, creating exactly $c_i h_i w_i$ non-zero diagonals in the process. Thus, while this method is straightforward, its dependence on the input's spatial dimensions causes it to scale poorly to larger convolutions.

We solve this problem by recognizing that the rows of any Toeplitz matrix can be permuted without changing its underlying computation. This lets us relax the invariant that our ciphertext must represent the image as a flattened tensor. Our approach builds on the multiplexed parallel convolutions introduced by Lee et al. [48], which interleaves channels within a ciphertext to efficiently handle strided convolutions. Notably, their method consumes two multiplicative levels: one level to first perform a non-strided convolution, and a

Figure 4. Strided convolutions produce Toeplitz matrices with many sparse non-zero diagonals (a). Orion automatically converts these to densely packed multiplexed convolutions to reduce the number of expensive ciphertext rotations (b).

second level to mask and collect only the correct elements (see Figure 5 of [48]). On the other hand, our single-shot multiplexed packing strategy consumes just one level by fusing this mask-and-collect step directly into the weight matrix, which can be pre-processed.

Figure 4b shows our method for converting the standard Toeplitz matrix in Figure 4a to our single-shot multiplexed solution. Our method produces a densely packed output ciphertext, parameterized by a gap, g. Subsequent non-strided convolutions maintain this gap, while strided convolutions increase it by a factor of s. This reformulation lets us leverage both BSGS and double-hoisting and doing so reduces the number of ciphertext rotations in ResNet-20 [31] from 1457 to 836. Table 2 of our results further compares these two packing strategies in larger neural networks.

Multi-ciphertext: When an input image does not fit into a single ciphertext (i.e. the number of pixels is greater than the number of slots), the image is split across multiple ciphertexts into contiguous slots where the last ciphertext may only be partially filled with data. In this case, we perform a blocked matrix-vector product with blocks of size $slots \times slots$. While we evaluate Orion in a single-threaded setting, each block performs independent work and is well-suited for parallel execution across multiple threads or cores.

5 Automatic Bootstrap Placement

In this section, we discuss the problem of bootstrap placement in private neural inference. We begin by highlighting its barrier to entry to motivate the need for automated solutions. Then, we describe the challenges that arise in automation. We close with a general and efficient solution for modern neural networks based on topological sorting.

5.1 Problem Overview

Recall from Section 2 that the goal of CKKS bootstrapping is to enable further homomorphic multiplications by increasing a ciphertext's level from $\ell = 0$ to $\ell = L_{\text{eff}} < L$. Determining the optimal location of bootstrap operations within a
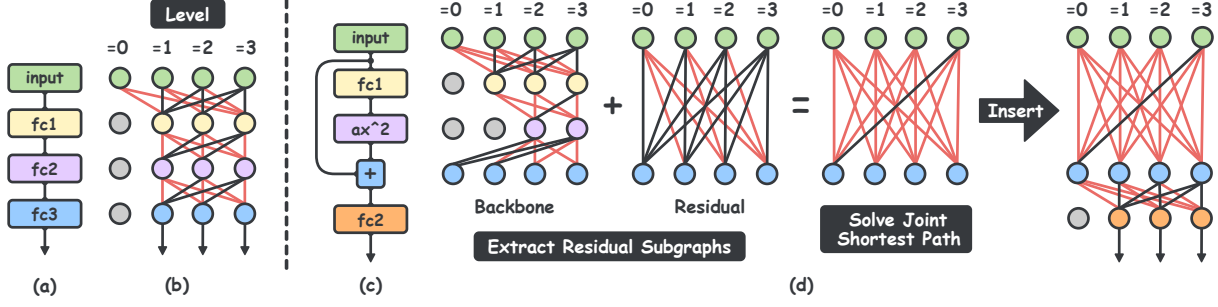
network is perhaps the most challenging aspect of FHE inference for three reasons. First, placing a bootstrap at any point in the network directly affects the levels of all subsequent operations, including future bootstrap locations. Second, naïve strategies that delay bootstrapping until absolutely necessary often result in more bootstraps being placed, particularly in networks with residual connections (see Figure 10 of Fhelipe [42]). And finally, bootstrap runtime grows superlinearly with L_{eff}. Thus, only minimizing the number of bootstrap operations often counter-intuitively increases network latency.

In Orion, our goal is to automatically determine the locations of bootstrap operations to minimize network latency. In doing so, we also determine the levels of all linear layers and activation functions. Collectively, we refer to our decisions as a *level management policy*.

Our constraints: To efficiently realize this goal, we impose three constraints. First, we restrict the placement of bootstrap operations between network layers. This greatly simplifies our analysis without significantly impacting its quality, since optimal policies inherently avoid bootstrapping within layers, where circuits are the widest. Second, we neglect the cost of activation functions and only model the latencies of evaluating linear layers and bootstraps. Indeed, 92.6% of our ResNet-20 latency comes from these two sources. Finally, we exclude networks with overlapping skip connections from our analysis, such as DenseNets [33].

5.2 Orion's Level Management Policy

The core idea: To introduce our level management policy, consider the simple three-layer fully connected network in Figure 5a without intermediate activation functions. The linear layers (`fc1-fc3`) each consume one level, and we set the effective level (L_{eff}) to 3. By restricting the placement of bootstrap operations to between network layers, we make it possible to efficiently construct a directed acyclic graph (DAG) that enumerates all possible network states. We refer to these graphs as *level digraphs*, and the level digraph for the network in Figure 5a is shown in Figure 5b. Each row of nodes in Figure 5b represents one linear layer. Nodes within

Figure 5. A feed-forward neural network with no skip connections (a) and its associated level digraph when $L_{\text{eff}} = 3$ (b). Edges between nodes connect network layers and red edges correspond to bootstrap operations. This skip-less network does not require a bootstrap operation if the input ciphertext begins at $\ell = 3$. Networks with skip connections (c) require us to first solve the sub-problem along the residual path before holistically solving bootstrap placement for the entire network. This network requires at least one bootstrap.

a row are each assigned a valid level and weighted by the latency of performing their corresponding linear layer at that level. A valid level ℓ satisfies $d \leq \ell \leq L_{\text{eff}}$, where d is the layer's multiplicative depth. Edges between nodes are weighted by the latency of bootstrapping up to L_{eff} if it is required. In Figure 5b, we highlight the edges that require bootstrap operations in red.

Note that even when a bootstrap occurs, the subsequent layer can still be performed at $\ell < L_{\text{eff}}$. While this choice may on the surface seem wasteful, the additional degree of freedom often leads to more efficient level management policies, especially in more realistic networks. The *optimal* level management policy in Figure 5b contains the operations along the shortest path from any input node ($\text{input}_{0 \leq \ell \leq 3}$) to any output node ($\text{fc3}_{1 \leq \ell \leq 3}$) in the level digraph, minimizing the latency (with respect to our heuristics) of an inference.

A general strategy: One problem with the approach in Figure 5b is that it does not support networks with multiple paths from input to output. Consider the network in Figure 5c with a residual connection and an activation function. Residual connections are a staple of modern neural network design, however complicate the choice of level management policy. A viable solution to Figure 5c might include solving two independent level digraphs, one per path through the network. However, this approach can lead to clashing choices for the levels of nodes common to both paths.

Instead, we first extract the sub-network around any residual connection and treat it as its own distinct problem. Then, we construct two separate level digraphs: one for the backbone network and one for the residual connection. Afterwards, we collapse these separate digraphs into one *aggregate* level digraph by solving a joint shortest path problem between every pair of input and output nodes. For instance, in Figure 5d, the edge in the aggregate level digraph from $\{\text{input}\}_{\ell=0}$ to $\{+\}_{\ell=0}$ is weighted by the sum of shortest paths between those same nodes in *each* individual digraph. Finally, we insert this aggregate level digraph back into the larger network, allowing it to be solved similarly to Figure 5b.

Implementation details: We estimate the latencies of both the linear layers and bootstrap operations with an analytical model. Then, Orion automatically inserts bootstrap operations within the network, abstracting these complexities away from the end-user. Orion's automatic level management and bootstrap placement takes only 6.67 (49.4) seconds for ResNet-20 (ResNet-110) and scalability is further explored in Table 4 of our results.

6 The Orion Framework

We now present the Orion framework and API, which includes automated support for the optimizations and techniques presented above (i.e., bootstrap placement and packing strategies). In addition, Orion automatically handles several subtle, yet difficult, challenges that arise when building large-scale FHE programs: range estimation for high-precision bootstrapping and Chebyshev polynomial evaluation, error-free scaling factor management, and system support for very large data structures.

The Orion API: FHE neural network applications are notoriously difficult to program, and significant research efforts [18, 20, 21, 42, 51, 64] have focused on reducing this barrier to entry. Yet, despite this progress, no high-level, high-performance API exists, which practitioners have come to expect from modern tools like JAX or Pytorch [11, 54]. Orion provides this interface. We implement custom Python bindings to the Lattigo [1] FHE library that directly interact with PyTorch tensors. Then, we use these bindings to build Orion modules that inherit and extend the functionality of their corresponding PyTorch modules. For example, Listing 1 shows a ResNet block in Orion that closely mirrors the equivalent PyTorch implementation. Orion facilitates PyTorch's ease of use while achieving state-of-the-art FHE neural network performance. While we chose PyTorch, it is straightforward to add support for other modern deep learning libraries.

Range estimation: Both high-precision bootstrapping and high-degree (Chebyshev) polynomial activation functions

Listing 1: ResNet block instantiation in Orion.

```python
import torch.nn as nn
import orion.nn as on

class BasicBlock(on.Module):
  def __init__(self, Ci, Co, stride=1):
    super().__init__()
    self.conv1 = on.Conv2d(Ci, Co, 3, stride, 1)
    self.bn1   = on.BatchNorm2d(Co)
    self.act1  = on.ReLU(degrees=[15,15,27])

    self.conv2 = on.Conv2d(Co, Co, 3, 1, 1)
    self.bn2   = on.BatchNorm2d(Co)
    self.act2  = on.SiLU(degree=63)

    self.add = on.Add()
    self.shortcut = nn.Sequential()
    if stride != 1:
      self.shortcut = nn.Sequential(
        on.Conv2d(Ci, Co, 1, stride, 0),
        on.BatchNorm2d(Co))

  def forward(self, x):
    out = self.act1(self.bn1(self.conv1(x)))
    out = self.bn2(self.conv2(out))
    out = self.add(out, self.shortcut(x))
    return self.act2(x)
```

require the inputs to be in the range $[-1, 1]$. Thus, when building large FHE programs, practitioners must insert specific scale-down operations to ensure this property holds for all calls to bootstrapping and Chebyshev evaluations. This step is especially necessary for neural networks, which generate intermediate activation values outside of $[-1, 1]$. Prior work chooses *per network* scaling factors by manually inspecting and tracking the largest intermediate values seen [37, 48]. Orion handles this process automatically through net.fit(), which accepts the entire training dataset as input, calculates *per layer* scaling factors, and inserts scale-down multiplications directly into the computational graph.

Scale management: Recall from Section 2.2 that encoding a cleartext into a CKKS plaintext involves multiplication by a scaling factor, Δ. Properly managing this scale factor for large FHE programs is challenging. To highlight this, consider performing a homomorphic multiplication (either PMult or HMult) where both operands are at level ℓ and have a scaling factor of Δ. The resulting ciphertext has a scaling factor of Δ^2 that is then rescaled to $\Delta^2/q_\ell \approx \Delta$ where q_ℓ is the last prime limb. Thus, rescaling also introduces approximation errors into the resulting ciphertext which grows with circuit depth [36].

In Orion, we propose a new technique to automatically handle scale management that we call *errorless neural network evaluation*. At a high level, we maintain the invariant that the scaling factor of any ciphertext between network layers must be precisely Δ. This technique extends the errorless, depth-optimal polynomial evaluation [10] to full neural network inferences. As an example of our technique, assume

we are performing a convolutional in Orion. We leverage the fact that our compilation phase has already pre-determined the level at which to perform this convolution (we will call this level j). We choose to then encode its weights with scale factor q_j (rather than Δ), the last RNS modulus at level j. Performing this convolution with a ciphertext at scale Δ results in an output with scale $\Delta \cdot q_j$. Rescaling will divide the ciphertext by the last prime modulus, q_j, thus precisely resetting this output ciphertext scale back to Δ.

Handling large data structures: Large datasets and networks require hundreds of gigabytes of rotation keys and matrix diagonals. Orion provides support to store these large data structures to disk in HDF5 format [27]. Then, the correct data is loaded dynamically during inference to minimize the size of our transient data.

7 Methodology

CPU setup: We evaluate Orion on a C4 GCP instance with an Intel Xeon Platinum 8581C processor clocked at 2.3 GHz and 512 GB of RAM. We target Lattigo v5.0.2 [1] in our evaluation, an open-source, single-threaded FHE library that supports both double-hoisted matrix-vector products and errorless, depth-optimal polynomial evaluation. Future work includes supporting multi-threaded FHE backends (e.g., OpenFHE [5]) and GPU implementations (e.g., HEAAN [15]). For a fair comparison, we rerun Lee et al. [48] and Fhelipe [42] on the exact same C4 GCP instance, which are also single-threaded.

Activation functions: As shown in Table 1, we use the x^2 activation function for MNIST networks and either ReLU or its smoother variant, SiLU [24] for all other datasets and networks. We approximate ReLU via a minimax composite polynomial [48]. Here, a high degree polynomial is decomposed into a composition of several lower degree polynomials to reduce the number of homomorphic multiplications. Following prior work, our composition consists of three polynomials of degree 15, 15, and 27. For SiLU, we use a degree-127 Chebyshev polynomial, obtained using a similar minimax approach. The multiplicative depth of ReLU is 14 (13 for $\text{sign}(x)$ and 1 for its multiplication with x), whereas the depth of SiLU is just 7. Later, we will show how the choice of activation function introduces a trade-off in latency and test accuracy.

Benchmarks: Our evaluation consists of four datasets (image sizes): MNIST ($28{\times}28{\times}1$) [46], CIFAR-10 ($32{\times}32{\times}3$) [43], Tiny ImageNet ($64{\times}64{\times}3$) [45], and ImageNet ($224{\times}224{\times}3$) [22]. We evaluate three neural networks for MNIST from prior work: a 3-layer MLP from SecureML [52], a 3-layer CNN from LoLA CryptoNets [12], and the largest LeNet-5 model from CHET [21] and EVA [20]. For CIFAR-10, we benchmark AlexNet [44], VGG-16 [60], and ResNet-20 [31], each with ReLU and SiLU activation functions. For Tiny ImageNet, we implement MobileNet-v1 [32] and ResNet-18, and for ImageNet, we evaluate ResNet-34 and ResNet-50. For all networks, we replace max pooling with average pooling.

Validation: We validate each benchmark by comparing Orion's FHE outputs against the equivalent cleartext outputs from PyTorch. For MNIST, we perform this validation across all 10,000 test images. For CIFAR-10 (Tiny ImageNet), we instead randomly sample 1,000 (100) test images. And for ImageNet, we perform just a single encrypted inference for comparison due to resource constraints.

Alongside accuracy and latency, we also report the mean *precision* (in bits) of the output, which is defined as $-\log_2(\epsilon)$, where ϵ is the mean absolute difference between the outputs of Orion and PyTorch. To compare with prior work, we also report the number of ciphertext rotations and number of bootstrap operations for each network. The links to code blocks for each network and their respective parameter sets can be found in Table 1.

8 Evaluation

In this section, we evaluate Orion, quantify our improvements over prior work, and demonstrate the effectiveness of our approach. We begin by presenting our evaluations across all networks and datasets in Table 1 that highlight the benefits of our single-shot multiplexing strategy as well as our automatic bootstrap placement algorithm. Next, we analyze the efficiency of our automatic bootstrap placement algorithm as we increase network depth, and finally we close with an object detection and localization case study. To the best of our knowledge, this is the first high-resolution ($448 \times 448 \times 3$) object detection using a deep neural network in FHE.

8.1 MNIST

For MNIST, we use the conjugate invariant [38] ring type in CKKS to set the number of slots equal to the ring degree, as opposed to the traditional $n = N/2$ used when bootstrapping is required. Notably, since our single-shot multiplexed convolutions consume only one level, the depth of networks such as LoLA are also roughly halved when compared to prior work. For instance, the LoLA implementation in Fhelipe [42] (PLDI '24) has a multiplicative depth of 10, whereas in Orion its depth is just 5. This enables us to reduce the ring degree from the typical $N = 2^{14}$ to $N = 2^{13}$ while remaining 128-bit secure. As a result, we improve upon the results of Fhelipe by nearly 83×, reducing end-to-end latencies from 19.0 seconds to just 0.23 seconds. With the same parameter set as Fhelipe, we achieve a mean latency of 0.97 seconds (19× reduction). Similarly, our LeNet-5 latency of 2.93 seconds is roughly 44× faster than the single-threaded results from EVA [20].

8.2 CIFAR-10

Packing comparisons: Table 2 more concretely compares our single-shot multiplexing strategy against the multiplexed approach from Lee et al. [48] using the CIFAR-10 networks from Table 1 alongside ResNet-110. Notably, our improvement over prior work *increases* with model complexity. This improvement occurs for two reasons. First, the benefits of

BSGS increase with filter size since, from Section 4, the complexity of homomorphic convolutions decreases from $O(f)$ to $O(\sqrt{f})$, with f the number of filter elements. Second, for small networks such as ResNet-20, we rely on Gazelle's hybrid method to diagonalize matrices that are often much smaller than $n \times n$, where n is the number of slots. Doing so maintains the property that convolutions only consume one level, however it induces sparser plaintext diagonals. For larger, multi-ciphertext networks that Orion primarily targets, the hybrid method is not needed, and plaintext diagonals are packed as densely as possible. These two reasons are also why AlexNet, despite having 86× the number of parameters as ResNet-20, only has 1.76× more rotations.

Performance comparisons: Table 3 further highlights our sources of improvement over the prior work of Fhelipe [42] when run on the same GCP instance. Notably, despite having only 1.71× fewer ciphertext rotations, our convolutional runtime is 11.2× faster which occurs for two reasons. First, roughly half of all ciphertext rotations in Orion are *hoisted*. Recall from Section 3 that hoisting amortizes the expensive aspects of the key-switch procedure *across* many ciphertext rotations and is only possible when using the diagonal encoding. Second, Orion's compilation phase automatically generates and stores all rotation keys and encoded matrix diagonals. On the other hand, Fhelipe, generates all encoded plaintexts on-the-fly *during* each convolution. The former is a better strategy, even if data must be stored to disk, since CKKS encoding involves both the iFFT and NTT.

Choice of activation function: Finally, we explore the trade-off in latency and accuracy by using different activation functions. In more detail, SiLU consumes half the levels of ReLU, which reduces the total multiplicative depth of the circuit. In turn, this reduction in multiplicative depth means less bootstraps must be performed during inference. We find that using SiLU decreases the cleartext accuracy on average by 2.1% but results in a 1.77× average speedup. This trade-off is straightforward to further explore given Orion's native support for low-degree polynomial activation functions with `on.Activation()`. For larger experiments (e.g. Tiny ImageNet and ImageNet), we opt to train our models with SiLU to reduce multiplicative depth and decrease FHE runtime.

8.3 Tiny ImageNet

We now present the results of our Tiny ImageNet experiments on MobileNet-v1 and ResNet-18. Prior work does not run inference using Tiny ImageNet; we report our runs in Table 1. Despite both networks having fewer parameters than VGG-16 and AlexNet, the number of ciphertext rotations increases substantially. This occurs because the number of ciphertext rotations is more closely tied to the network's FLOPS (number of floating point operations), than its parameter count. Since the input image size has increased four-fold from ($32 \times 32 \times 3$) to ($64 \times 64 \times 3$), the size of our matrix-vector products has also increased by the same amount.

	Model	Params (M)	FLOPS (M)	Code	Set	# Rots	Act.	Depth	# Boots	Clear Acc.	FHE Acc.	Prec. (b)	Time (s)
MNIST	MLP	0.12	0.12	🔗	🔗	70	x^2	5	0	98.02%	98.03%	4.60	0.29
	LoLA	0.10	0.13	🔗	🔗	73	x^2	5	0	98.63%	98.62%	4.81	0.23
	LeNet	1.66	4.30	🔗	🔗	282	x^2	7	0	99.31%	99.31%	10.4	2.93
CIFAR-10	AlexNet	23.3	188	🔗	🔗	1470	ReLU	109	15	92.83%	92.80%	4.27	337.2
							SiLU	60	7	89.42%	89.30%	7.19	190.3
	VGG-16	14.7	314	🔗	🔗	1771	ReLU	227	28	94.53%	94.50%	5.10	588.6
							SiLU	137	14	92.27%	93.60%	9.72	397.4
	ResNet-20	0.27	41.2	🔗	🔗	836	ReLU	287	37	93.21%	93.40%	4.84	618.2
							SiLU	154	19	92.61%	91.70%	13.6	301.4
Tiny	MobileNet	3.25	47.4	🔗	🔗	2508	SiLU	218	42	56.31%	62.00%	8.94	892.4
	ResNet-18	11.3	2260	🔗	🔗	10838	SiLU	138	61	60.57%	57.00%	8.56	1447
IMNet	ResNet-34	21.8	3670	🔗	🔗	48108	SiLU	267	146	73.66%	N/A	8.59	14338
	ResNet-50	25.6	4110	🔗	🔗	143217	SiLU	395	351	76.22%	N/A	8.90	32324

Table 1. We evaluate Orion on a series of networks and datasets ranging from MLP on MNIST to ResNet-50 on ImageNet. We provide the code, parameter set (Set), rotation amount, number of bootstraps as well as both cleartext and FHE accuracy.

Work	ResNet-20	ResNet-110	VGG-16	AlexNet
Lee et al. [48]	1382	7622	9214	9422
Orion (us)	836	4676	1771	1470
Improvement	1.65×	1.64×	5.20×	6.41×

Table 2. A comparison of ciphertext rotation counts in CIFAR-10 networks between Lee et al. [48] and Orion.

Work	# Rots.	# Boots.	Convs. (s)	Latency (s)
Fhelipe [42]	1428	58	334.5	1468
Orion (us)	836	37	29.89	618.2
Improvement	1.71×	1.58×	11.2×	2.38×

Table 3. Quantifying the several sources of improvement in ResNet-20 performance over the prior work of Fhelipe [42].

While MobileNet is a *deeper* network than ResNet-18, our automatic bootstrap placement algorithm places *fewer* total bootstraps in it. This occurs because MobileNet has no residual connections, while ResNet-18 has eight. Residual connections place an additional constraint on our bootstrap placement algorithm, typically that the input and output levels of the residual block be the same and doing so generally increases bootstrap counts. This same observation is found in Baruch et al. [6]. Furthermore, since MobileNets are designed for mobile or embedded systems, they contain much cheaper depth-wise separable convolutions. Looking closer, we find that the average level that Orion performs MobileNet convolutions at is $\ell = 6$, whereas in ResNet-18 it is $\ell = 3$. This indicates that our bootstrap algorithm is more aggressively trading off the runtime of (cheaper) convolutions for fewer bootstraps in MobileNet than it is in ResNet-18.

8.4 ImageNet

To demonstrate both the scalability and user-friendliness of Orion, we evaluate ResNet-34 and ResNet-50 on the ImageNet-1k dataset. Importantly, Orion does not require any FHE-specific training or modification to these networks such as

the removal of skip connections or range-aware loss functions (both of which are explored in HeLayers [6]). Additionally, these networks have 81× and 95× more parameters than the largest networks supported by Fhelipe [42]. Here, we directly load the pretrained weights from torchvision and simply finetune both networks after replacing ReLU activations with SiLU and max pooling with average pooling. Since Orion extends PyTorch, this fine-tuning can be performed using existing PyTorch training scripts, and our SiLU models match the accuracy of torchvision's ReLU models.

The (*single-threaded*) end-to-end encrypted inference time for both networks are 3.98 hours and 8.98 hours, respectively. Baruch et al. [6] evaluate a similar ResNet-50 model across 32 CPU threads using HEaaN [15] in 2.53 hours by replacing ReLU with its degree-18 approximation. Since a direct comparison is challenging, we note that our ResNet-50 implementation contains just 351 bootstrap operations, whereas they use 8,480 bootstraps [6]. While we could only evaluate one encrypted inference per network, our results match the cleartext PyTorch output with 8 bits of precision.

8.5 Bootstrap Placement Complexity

Our automatic bootstrap placement algorithm scales well with both network depth and complexity. Table 4 presents

Operation	Res-20	Res-32	Res-44	Res-56	Res-110
Compile (s)	437	654	867	1096	2132
Boot. Place. (s)	6.67	10.6	19.1	24.8	49.4
# Bootstraps	37	61	85	109	217

Table 4. An analysis of the scalability of our automatic bootstrap placement algorithm with network depth.

Figure 6. The first homomorphic object detection and localization results. Labels indicate each object's predicted class and its confidence score in $[0, 1]$.

the compile time, bootstrap placement time, and number of bootstraps for ResNet-20 through ResNet-110 using the same composite approximation to ReLU from Section 7. Bootstrap placement time refers to the time our algorithm takes to determine the location of every bootstrap in the network.

We find a linear increase in bootstrap placement time as network depth increases. We explain this by extending the fully-connected network in Figure 5b to an arbitrary depth d. Each layer contains $L_{\text{eff}}+1$ vertices (V) and is connected to the next layer through L_{eff}^2 edges (E), where L_{eff} is the number of levels remaining after each bootstrap operation. The shortest path, found using a topological sort with complexity $O(|V| + |E|)$, grows linearly with network depth as $O(L_{\text{eff}}^2 \cdot d)$.

8.6 Case Study: Object Localization

We close our evaluation with the first large-scale homomorphic object detection and localization experiments. In these experiments, we train a YOLO-v1 [57] model with a ResNet-34 backbone on the PASCAL-VOC dataset [25], which consists of 20,000 images, each resized to $448 \times 448 \times 3$, spanning 20 classes. Our model has 139 million parameters and is designed to predict any instances of these 20 classes within an image along with their corresponding bounding boxes; a much harder task than direct object classification.

Figure 6 visualizes two FHE output predictions from Orion, each with single-threaded latencies of 17.5 hours. While this is still far from practical, its implementation took just 60 additional lines of code (⬀) and required minimal changes from its analogous PyTorch implementation.

9 Related Works

Prior work in FHE programming fall into two major categories: circuit-level compilers and domain-specific compilers. Circuit-level compilers [4, 17–19, 29, 47, 49–51, 53, 62, 64, 65] represent most of prior work and focus on lower-level optimizations for general programs and typically target smaller workloads. In contrast, domain-specific compilers (e.g., for machine learning) [7–9, 16, 20, 21, 42, 63] often sacrifice this fine-grained control to efficiently support much larger applications. Orion falls into the latter category.

Circuit-level compilers: Early FHE compilers [13, 17, 19] focused on circuit-level optimizations, and more recent tools such as Porcupine [18], Coyote [51], and HECO [64], offer automated solutions for scheduling FHE instructions. HECO adopts the MLIR framework to target a wide variety of FHE backends and hardware. In parallel, Hecate [50] and ELASM [49] propose several rescaling techniques to both improve performance and explore the tradeoff in scale management and latency. However, as the authors note, scaling these techniques to deep learning workloads remains infeasible.

Domain-specific compilers: CHET [21] was one of the first FHE compilers to target machine learning workloads. Its focus included automatically selecting encryption parameters, data layouts, and introduced an intermediate representation to decouple program execution from improvements to cryptography. EVA [20] improved upon CHET by proposing the waterline rescaling technique to efficiently manage scaling factors. Both CHET and EVA were implemented in SEAL [59], which does not natively support bootstrapping and therefore do not target *deep* neural networks. nGraph-HE [8, 9], TenSEAL [7], and SEALion [59] also target machine learning with Python APIs, but similarly lack the the ability to automate bootstrapping.

Recently, Dacapo [16] proposed an automatic bootstrap placement algorithm and integrated their solution into the GPU-accelerated HEaaN library [14]. Their approach involves computing a set of candidate bootstrap locations and then estimating the latency when bootstrapping at different combinations of these locations. HeLayers [2] also automates bootstrap placement and further provides a robust framework for deep learning inference in the most popular FHE backends [1, 5, 14]. To the best of our knowledge, this is the only FHE compiler outside of Orion to support datasets larger than CIFAR-10, and it similarly supports ImageNet.

Fhelipe [42] is perhaps the closest prior work to Orion. Although it is a more general compiler for tensor arithmetic, it is also capable of supporting deep learning applications. Unlike Fhelipe, the goal of Orion is not to compile FHE programs into a list of primitive operations. Instead, we target a higher level of abstraction (e.g., linear transforms) to leverage cryptographic optimizations such as hoisting in our FHE backend. It is here that we find the majority of our performance improvements.

10 Conclusion

In this paper, we propose **Orion**, a framework that completely automates and translates neural networks directly into FHE programs. Orion allows both researchers and practitioners to rapidly iterate on their ideas and understand FHE using a high-level machine learning library such as PyTorch. We propose 1) our *single-shot multiplexed* packing strategy that implements arbitrary convolutions and 2) our automatic bootstrap placement algorithm that requires no user input. We integrate both directly into Orion and achieve state-of-the-art (single-threaded) latency for standard FHE benchmarks. Orion can run large-scale neural networks such as ResNet-50 on ImageNet and even YOLO-v1 object detection on images of size $448 \times 448 \times 3$.

Going forward, we plan to lower Orion to alternative backends such as multi-threaded FHE libraries (e.g., OpenFHE [5]) and GPU systems (e.g., HEaaN-GPU [14], Cheddar [39]). Additionally, our high-level Python interface allows other researchers to extend Orion to support new networks layer types such as self-attention or layer normalization. By lowering the barrier to entry into this field, Orion helps strengthen and embolden research within the FHE community.

Acknowledgements

References

[1] Lattigo v4. Online: https://github.com/tuneinsight/lattigo, August 2022. 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, Hayim Shaul, and Omri Soceanu. HeLayers: A tile tensors framework for large neural networks on encrypted data. *Proceedings on Privacy Enhancing Technologies*, 2023(1):325–342, jan 2023.

[3] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption standard. Cryptology ePrint Archive, Paper 2019/939, 2019.

[4] David W. Archer, José Manuel Calderón Trilla, Jason Dagit, Alex Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard Ryan. Ramparts: A programmer-friendly system for building homomorphic encryption applications. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC'19, page 57–68, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, 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.

[6] Moran Baruch, Nir Drucker, Gilad Ezov, Yoav Goldberg, Eyal Kushnir, Jenny Lerner, Omri Soceanu, and Itamar Zimerman. Training large scale polynomial cnns for e2e inference over homomorphic encryption, 2023.

[7] Ayoub Benaissa, Bilal Retiat, Bogdan Cebere, and Alaa Eddine Belfedhal. Tenseal: A library for encrypted tensor operations using homomorphic encryption, 2021.

[8] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. ngraph-he2: A high-throughput framework for neural network inference on encrypted data, 2019.

[9] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. ngraph-he: a graph compiler for deep learning on homomorphically encrypted data. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, CF '19, page 3–13, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. Cryptology ePrint Archive, Paper 2020/1203, 2020.

[11] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[12] Alon Brutzkus, Oren Elisha, and Ran Gilad-Bachrach. Low latency privacy preserving inference. In *International Conference on Machine Learning*, 2019.

[13] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. Armadillo: a compilation chain for privacy preserving applications. Cryptology ePrint Archive, Paper 2014/988, 2014.

[14] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. Cryptology ePrint Archive, Paper 2016/421, 2016.

[15] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, pages 409–437. Springer, 2017.

[16] 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, Philadelphia, PA, August 2024. USENIX Association.

[17] Eduardo Chielle, Oleg Mazonka, Homer Gamil, Nektarios Georgios Tsoutsos, and Michail Maniatakos. E3: A framework for compiling c++ programs with encrypted operands. Cryptology ePrint Archive, Paper 2018/1013, 2018.

[18] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T. Lee, and Brandon Reagen. Porcupine: A synthesizing compiler for vectorized homomorphic encryption. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 375–389, New York, NY, USA, 2021. Association for Computing Machinery.

[19] Eric Crockett, Chris Peikert, and Chad Sharp. Alchemy: A language and compiler for homomorphic encryption made easy. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1020–1037, New York, NY, USA, 2018. Association for Computing Machinery.

[20] 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*. ACM, jun 2020.

[21] 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*, PLDI 2019, page 142–156, New York, NY, USA, 2019. Association for Computing Machinery.

[22] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[23] Austin Ebel and Brandon Reagen. Osiris: A systolic approach to accelerating fully homomorphic encryption, 2024.

[24] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning, 2017.

[25] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, June 2010.

[26] Axel Feldmann, Nikola Samardzic, Aleksandar Krastev, Srini Devadas, Ron Dreslinski, Karim Eldefrawy, Nicholas Genise, Chris Peikert, and Daniel Sanchez. F1: A fast and programmable accelerator for fully homomorphic encryption (extended version), 2021.

[27] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, AD '11, page 36–47, New York, NY, USA, 2011. Association for Computing Machinery.

[28] Harvey L. Garner. The residue number system. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), page 146–153, New York, NY, USA, 1959. Association for Computing Machinery.

[29] Shruthi Gorantala, Rob Springer, Sean Purser-Haskell, William Lam, Royce Wilson, Asra Ali, Eric P. Astor, Itai Zukerman, Sam Ruth, Christoph Dibak, Phillipp Schoppmann, Sasha Kulankhina, Alain Forget, David Marn, Cameron Tew, Rafael Misoczki, Bernat Guillen, Xinyu Ye, Dennis Kraft, Damien Desfontaines, Aishe Krishnamurthy, Miguel Guevara, Irippuge Milinda Perera, Yurii Sushko, and Bryant Gipson. A general purpose transpiler for fully homomorphic encryption. Technical report, Google LLC, 2021.

[30] Shai Halevi and Victor Shoup. Algorithms in HElib. Cryptology ePrint Archive, Paper 2014/106, 2014.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[32] 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, 2017.

[33] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks, 2018.

[34] Francesco Intoci, Sinem Sav, Apostolos Pyrgelis, Jean-Philippe Bossuat, Juan Ramon Troncoso-Pastoriza, and Jean-Pierre Hubaux. slytherin: An agile framework for encrypted deep neural network inference, 2023.

[35] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669, Baltimore, MD, August 2018. USENIX Association.

[36] Andrey Kim, Antonis Papadimitriou, and Yuriy Polyakov. Approximate homomorphic encryption with reduced approximation error. Cryptology ePrint Archive, Paper 2020/1118, 2020.

[37] Dongwoo Kim and Cyril Guyot. Optimized privacy-preserving cnn inference with fully homomorphic encryption. *IEEE Transactions on Information Forensics and Security*, 18:2175–2187, 2023.

[38] Duhyeong Kim and Yongsoo Song. Approximate homomorphic encryption over the conjugate-invariant ring. In Kwangsu Lee, editor, *Information Security and Cryptology – ICISC 2018*, pages 85–102, Cham, 2019. Springer International Publishing.

[39] Jongmin Kim, Wonseok Choi, and Jung Ho Ahn. Cheddar: A swift fully homomorphic encryption library for cuda gpus, 2024.

[40] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. Sharp: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[41] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. BTS. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ACM, jun 2022.

[42] 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. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024.

[43] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[44] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc.

[45] Ya Le and Xuan S. Yang. Tiny imagenet visual recognition challenge. 2015.

[46] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[47] DongKwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. Optimizing homomorphic evaluation circuits by program synthesis and term rewriting. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 503–518, New York, NY, USA, 2020. Association for Computing Machinery.

[48] 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 Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 12403–12422. PMLR, 17–23 Jul 2022.

[49] 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, Anaheim, CA, August 2023. USENIX Association.

[50] Yongwoo Lee, Seonyeong Heo, Seonyoung Cheon, Shinnung Jeong, Changsu Kim, Eunkyung Kim, Dongyoon Lee, and Hanjun Kim. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–204, 2022.

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

[52] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. Cryptology ePrint Archive, Paper 2017/396, 2017. https://eprint.iacr.org/2017/396.

[53] Sunjae Park, Woosung Song, Seunghyeon Nam, Hyeongyu Kim, Junbum Shin, and Juneyoung Lee. Heaan.mlir: An optimizing compiler for fast ring-based homomorphic encryption. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023.

[54] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[55] Brandon Reagen, Woo-Seok Choi, Yeongil Ko, Vincent T. Lee, Hsien-Hsin S. Lee, Gu-Yeon Wei, and David Brooks. Cheetah: Optimizing and accelerating homomorphic encryption for private inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 26–39, 2021.

[56] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2016.

[57] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2016.

[58] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. Craterlake: A hardware accelerator for efficient unbounded computation on encrypted data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 173–187, New York, NY, USA, 2022. Association for Computing Machinery.

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

[60] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[61] Deepraj Soni, Negar Neda, Naifeng Zhang, Benedict Reynwar, Homer Gamil, Benjamin Heyman, Mohammed Nabeel, Ahmad Al Badawi, Yuriy Polyakov, Kellie Canida, Massoud Pedram, Michail Maniatakos, David Bruce Cousins, Franz Franchetti, Matthew French, Andrew Schmidt, and Brandon Reagen. Rpu: The ring processing unit, 2023.

[62] McKenzie van der Hagen and Brandon Lucia. Client-optimized algorithms and acceleration for encrypted compute offloading. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 683–696, New York, NY, USA, 2022. Association for Computing Machinery.

[63] Tim van Elsloo, Giorgio Patrini, and Hamish Ivey-Law. Sealion: a framework for neural network inference on encrypted data, 2019.

[64] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. Heco: Fully homomorphic encryption compiler, 2023.

[65] Alexander Viand and Hossein Shafagh. Marble: Making fully homomorphic encryption accessible to all. In *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC '18, page 49–60, New York, NY, USA, 2018. Association for Computing Machinery.