Bridging Usability and Performance: A Tensor Compiler for Autovectorizing Homomorphic Encryption

Edward Chen
Carnegie Mellon University
ejchen@cmu.edu

Fraser Brown
Carnegie Mellon University
fraserb@cmu.edu

Wenting Zheng
Carnegie Mellon University
wenting@cmu.edu

Abstract

Homomorphic encryption (HE) offers strong privacy guarantees by enabling computation over encrypted data. However, the performance of tensor operations in HE is highly sensitive to how the plaintext data is packed into ciphertexts. Large tensor programs introduce numerous possible layout assignments, making it both challenging and tedious for users to manually write efficient HE programs.

In this paper, we present Rotom, a compilation framework that autovectorizes tensor programs into optimized HE programs. Rotom systematically explores a wide range of layout assignments, applies state-of-the-art optimizations, and automatically finds an equivalent, efficient HE program. At its core, Rotom utilizes a novel, lightweight `ApplyRoll` layout conversion operator to easily modify the underlying data layouts and unlock new avenues for performance gains. Our evaluation demonstrates that Rotom scalably compiles all benchmarks in under 5 minutes, reduces rotations in manually optimized protocols by up to 4×, and achieves up to 80× performance improvement over prior systems.

1 Introduction

The recent revolution in advanced machine learning and data analytics offers unprecedented opportunities. However, these technologies have also introduced new and significant privacy risks [49,52,53]. For example, inference-as-a-service offerings like ChatGPT have caused many concerns about user data privacy. These concerns have led to bans from major companies like Samsung [33,46] and temporary government restrictions in countries such as Italy [48]. The growing concern over user privacy has led to increasing regulatory pressures, with laws and policies threatening to stifle progress if these challenges are not adequately addressed [7,21,48].

Cryptographic primitives for secure computation are a promising approach. In particular, homomorphic encryption (HE) is one such powerful primitive that enables *computation directly on encrypted data*. HE allows a party P_0 to encrypt its input x using a private key to get a ciphertext $\texttt{Enc}(x)$. A different party P_1 applies a function f over the ciphertext to get $f(\texttt{Enc}((x))$, where $f(\texttt{Enc}(x)) = \texttt{Enc}(f(x))$. P_1 cannot see the decrypted result of $f(x)$ as it does not own the private key. Since P_0 owns the private key, $\texttt{Enc}(f(x))$ can be decrypted by P_0 to get $f(x)$.

There are two variants of HE: partially HE (PHE) and fully HE (FHE). PHE is only viable when the target computation has limited multiplicative depth; FHE removes this restriction by introducing a bootstrapping operation that enables unbounded computation. Both PHE and FHE are important building blocks in end-to-end secure computation protocols, such privacy-preserving inference-as-a-service applications [22,25,26,28,32,40,41,43,44,54].

These applications predominantly use lattice-based HE schemes such as BFV/BGV/CKKS [6,10,20], which provide significant speedups over prior designs. One common trait of the schemes is their *vector* programming model, where multiple plaintext values are encrypted into a single ciphertext with thousands of slots ($n = 4K - 64K$). The schemes support only three homomorphic operations over encrypted vectors: element-wise addition, element-wise multiplication, and cyclic rotations. While single-instruction, multiple data (SIMD) operations improve performance through parallelism, their benefits are only realized through high slot utilization, i.e., by packing as many plaintext values as possible into the ciphertext(s). Most HE programs require a mix of SIMD additions/multiplications and data movement operations to achieve high slot utilization. Unfortunately, these data movement operations are limited to expensive intra-ciphertext rotations. Therefore, different *layouts*—the order in which plaintext values are packed into ciphertext(s)—can result in wildly different data movement and performance. Optimizing layout choices is imperative, but manually doing so is tedious.

Cryptography experts have spent substantial effort [26,32, 41,43,54] carefully crafting packing schemes for various machine learning applications. These workloads support operations over large tensors suitable for the high-parallelized, SIMD operations used in HE. For example, Gazelle [32] optimized packing schemes for ciphertext-plaintext matrix-vector multiplication used in convolutional neural networks. Bolt [43] later showed that such techniques do not directly generalize to efficient ciphertext-plaintext matrix multiplication used in transformers. Naive generalization leads to an overhead of $128\times$ more multiplications and over $2000\times$ more rotations than an optimized scheme. Manually optimized protocols show that different applications can require entirely different packings, and simply generalizing a packing scheme from one application to another is neither easy for a user nor efficient in performance.

Is it possible to design compilers that can *automatically* generate optimized, application-specific vectorized HE implementations? We define this question as the *layout assignment problem*, a search problem where an automated system has to explore numerous possible layouts to find an assignment that leads to the least costly HE program. Prior compiler works [1, 14, 18, 19, 34, 39, 47, 50] proposed a variety of solutions to this problem, but these works all have one of two drawbacks. As with any search problem, the main challenge is the tussle between search speed and search quality. Many works used heuristics and limited, restrictive layout options [1, 18, 19, 34, 50] in order to improve the search speed. However, the restricted layouts can introduce expensive data movement costs (i.e., conversions between layouts) that reduce search quality. One recent work proposed to search more complex input layouts, but suffered from long compilation times—over 13 hours for a single matrix multiplication [47]. In fact, brute-forcing a fully optimal solution is infeasible as realistic tensor workloads oftentimes have hundreds-of-thousands of layout possibilities to consider. The main challenge is devising a way to ensure finding an effective search quality for commonly seen workloads, while minimizing search speed.

In this paper, we present Rotom, a compiler framework that efficiently finds an optimized layout assignment from a high-level tensor program. Rotom's main goal is to resolve the tension between layout search speed and search quality.

To this end, Rotom relies on three key techniques. First, we formalize layout alignment rules that allow Rotom to determine how to implement a tensor operator in HE and derive its associated costs. When searching for an efficient layout assignment, these rules help Rotom easily reason about different layout choices and their respective HE implementations. For a given binary tensor operator, Rotom uses alignment to check whether the operand layouts satisfy the logical requirements to implement a tensor operator in HE; otherwise, it will determine which layout conversions are required. While alignment is used implicitly by prior works, Rotom's contribution extends these rules to also handle alignment for more complex layouts, such as the diagonal packing.

Second, we design a new lightweight layout conversion operator, `ApplyRoll`, that can be used to generate inexpensive layout conversions. In many common scenarios, our new layout conversion operator can provide $3-8\times$ more performance compared to approaches used in prior works. Furthermore, `ApplyRoll` lowers to a very structured set of HE operators—a fixed step-size rotations and multiplications with evenly spaced 0-1 bitmasks—making it amicable to additional optimizations. For example, this structure pattern-matches on the well-known baby-step giant step optimization [32] that can reduce the number of HE rotations by a square root factor.

Lastly, we introduce 4 cost-based, symmetry-breaking heuristics to reduce the search time without compromising search quality. Rotom uses a top-down enumeration search to find an efficient layout assignment. Naively performing this search would be intractable, as there are potentially hundred-of-thousands of different layout choices to consider for real-world application workloads. Instead, Rotom employs these heuristics to help control the search space, reducing the layout choices to just hundreds and enabling Rotom to compile large tensor programs in seconds.

We implement Rotom in $\approx$12,000 lines of Python and evaluate Rotom on tensor programs used in realistic workloads. Rotom efficiently compiles most workloads within a few seconds and all workloads in under 5 minutes. Rotom reduce rotations used in hand-tuned protocols by up to $4\times$ and outperforms prior HE compilers by up to $80\times$.

To summarize, Rotom makes the following contributions:

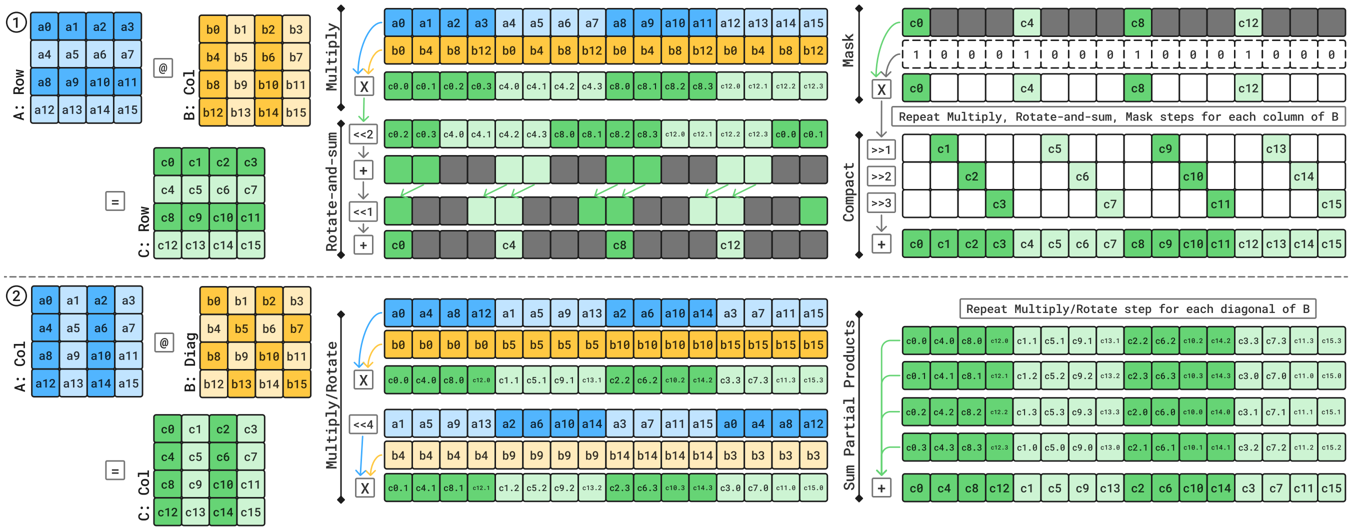
1. A lightweight layout conversion operator, `ApplyRoll`.
2. A formalization of layout alignment with rolled layouts.
3. 4 novel peephole optimizations derived from `ApplyRoll` that enables $3-80\times$ improvement over prior work.
4. 4 cost-based, symmetry-breaking heuristics to scalably perform layout assignment within seconds, where prior work did not finish compiling within 24 hours.
5. An evaluation across 7 tensor workloads, including automatically packing a transformer model's attention layer for the first time.

2 Background

The following sections introduce background preliminaries for our target HE schemes and motivate the importance and difficulty of finding "good" layout assignments.

2.1 Preliminary: Homomorphic Encryption

Lattice-based HE schemes like BFV/BGV/CKKS [6, 10, 20] have recently gained interest thanks to their relative efficiency. A common trait of these schemes is that they support a *vector* programming model, where n plaintext values are in encrypted into a single ciphertext (with n slots). For example, a



A: Row
B: Col
C: Row
Multiply
Rotate-and-sum
Mask
Compact
Repeat Multiply, Rotate-and-sum, Mask steps for each column of B
A: Col
B: Diag
C: Col
Multiply/Rotate
Sum Partial Products
Repeat Multiply/Rotate step for each diagonal of B

Figure 1: ① shows an HE matrix multiplication with simple layouts and expensive data movement. ② shows a more optimized implementation using a diagonalized layout.

single plaintext vector $v = [v_1, v_2, \dots, v_n]$ can be entirely encrypted into a single ciphertext $\texttt{Enc}(v)$. The schemes support only three HE operations: element-wise addition and multiplication (also known as "single instruction, multiple data" or SIMD), and ciphertext rotation that cyclically shifts the underlying plaintext vector. The number of available slots within each vector is typically very large (e.g., $n = 4K - 64K$).

While addition and ciphertext-plaintext (ct-pt) multiplication are relatively low-cost in HE schemes ($\approx$0.1ms), rotation and ciphertext-ciphertext (ct-ct) multiplication are orders of magnitudes more expensive ($\approx$10-30ms). Unfortunately, rotations are the sole operation that facilitates intra-ciphertext data movement. Therefore, minimizing the number of rotations—in other words, minimizing *data movement*—is crucial to HE program performance.

In end-to-end secure computation protocols, such as privacy-preserving inference applications, HE is used in predominantly two ways: partially HE (PHE) and fully HE (FHE). PHE is only viable when the computation has *limited* multiplicative depth. This variant is often used alongside other cryptographic primitives in secure two-party computation (2PC) protocols [26, 28, 32, 40, 43]. FHE removes this restriction by introducing a bootstrapping operation, enabling unbounded computation with arbitrary multiplicative depth; end-to-end protocols can be implemented solely in FHE [22, 25, 41, 44, 54]. The tradeoff between PHE and FHE is that the former exhibits more communication, while the latter exhibits significantly more computation. The performance of PHE + 2PC or FHE for particular application depends on a variety of factors like network bandwidth and latency, the availability of GPU or specialized accelerator for FHE, etc.

2.2 Finding A Good Layout Assignment Is Hard

Manually writing an HE program is extremely tedious. Users must determine how to pack plaintext data into ciphertexts with thousands of slots while working with a limited set of data movement instructions. Unfortunately, different input layouts can result in HE programs with wildly different data movement and costs. Even worse, changing the input layout of an HE program requires rewriting the layouts for all downstream operations. Brute-forcing an optimal solution is practically infeasible as there are numerous layout assignments to explore for real-world applications (e.g., a 64×64 matrix alone can have 12! different tilings).

To illustrate the impact of layouts on HE performance, Figure 1 compares two vectorized implementations of matrix multiplication, a key operation in transformer workloads. In this simplified setup, we use 4×4 matrices and assume each ciphertext has $n = 16$ slots. Matrix A (blue) is encrypted and sent by the client; B (orange) is a private, plaintext server-side matrix; and C (green) is the encrypted result returned to the client. Both A and C are compactly packed into a single ciphertext to minimize communication costs. B is plaintext matrix and can be freely repacked by the server. Each partial product of C is labeled as $ci.j$, where i refers to the final output index and j refers to the index of the partial products for summation. Part ① shows a costly HE program with simple layouts; and part ② presents an optimized implementation using a diagonal layout.

In part ①, A is row-packed, B is column-packed into 4 separate ciphertexts where each column of B is repeated, and C is row-packed. First, the program computes a multiplication the rows of A and the first column of B. The partial prod-

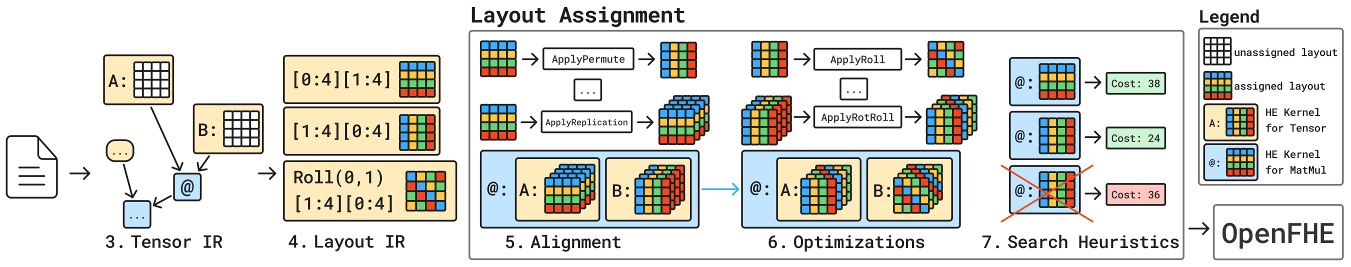


Figure 2: Rotom compilation pipeline. Each component references a section in the paper.

ucts are summed together using a rotate-and-sum routine—a series of rotations and additions to sum elements together within a ciphertext—to get the first column of C. Garbage partial products (marked in gray) are later removed using multiplications with 0-1 bitmasks (masking). The `multiply`, `rotate-and-sum`, and `mask` steps are repeated 3 more times for each column in B. Finally, the resulting ciphertexts are compacted together into a single ciphertext. In total, this implementation results in 4 multiplications, 11 rotations and 11 additions.

In part ②, A is column-packed, B is diagonally-packed where each diagonal value is repeated 4 times, and C is column-packed. In this implementation, each column of A is multiplied by a repeated diagonal in B, resulting in a partial product of C that is aligned for summation across the slots of the ciphertexts. Imperatively, this input packing avoids extraneous data movement operations compared to part ①. Using optimized layouts, part ② reduces the total computation to 4 multiplications, 3 rotations, and 3 additions.

As demonstrated in Figure 1, different input layouts result in different HE implementations, output layouts, and overall costs. Notably, the output layouts are later used in downstream operations; writing an efficient HE program requires careful reasoning and cheap layout conversions between different layout choices. Rotom alleviates these challenges by automatically exploring a wide range of layout choices using its lightweight conversion operator, `ApplyRoll`.

3 Overview

Rotom is a compiler framework that takes in a high-level tensor program, finds an efficient layout assignment, and lowers to an HE program that can be run by an HE backend (see Figure 2). In the following sections, we walk through Rotom's compilation pipeline in more detail.

3.1 Rotom's Compilation Pipeline

We describe Rotom's tensor DSL, its tensor and layout IRs, layout alignment rules it uses for layout conversions, optimizations over layout IR, and its strategy lowering layout IR to HE backends.

```python
from rotom import Tensor, MatMul, Conv2D

# matmul example
A = Tensor("A", (64,64), secret=True)
B = Tensor("B", (64,64), secret=False)
matmul_res = MatMul(A, B)

# convolution example
I = Tensor("I", (3,32,32) secret=True)
W = Tensor("W", (16,3,3), secret=False)
conv_res = Conv2D(I, W, stride=1, "valid")
```

Listing 1: MatMul and Convolution in Rotom DSL.

3.1.1 Tensor DSL and IR

To cut down on programming headaches, Rotom supports a python-based tensor DSL, similar to writing programs using PyTorch [45]. This DSL exposes common tensor operators used in machine learning, like matrix multiplication and convolutions. Listing 1 illustrates how to use Rotom's DSL to write simple tensor programs with only a few lines of code. In the example, the `Tensor` operator defines entry points for public and encrypted tensors; `MatMul` and `Conv2D` describe tensor operations on those inputs. Rotom then parses this program into its tensor IR, a direct acyclic graph (DAG) of tensor operators which are a one-to-one match to the tensor DSL. Appendix A details the tensor operators within this IR.

3.1.2 Layout IR

Rotom represents a layout assignment using layout operators in its layout IR. When searching for a layout assignment, Rotom lowers each tensor operator into an *HE Kernel*—a DAG of layout operators that implements a single tensor operator. Each layout operator extends a tensor operator by maintaining a layout representation (see Section 4) of how tensor elements are packed into the ciphertext(s).

Rotom's layout IR supports conversion operators—such as `ApplyPermute`, `ApplyReplication`, and `ApplyCompact`—which transform one layout to another. `ApplyRoll` is a new conversion operator introduced by Rotom and is the main workhorse behind most of Rotom's layout optimizations. Appendix B details the layout operators in this IR.

3.1.3 Layout Alignment

When Rotom encounters a binary tensor operator, it may require layout conversions to ensure the corresponding HE kernel is both correct and has high-slot utilization. Rotom uses layout alignment rules (see Section 5) to reason about these layout conversions and their associated data movement costs, directly from the layout representation. Additionally, Rotom provides new rules to reason about alignment with its new `ApplyRoll` conversion operator.

3.1.4 IR Optimizations

To improve the performance of an HE kernel, Rotom introduces 4 new peephole optimizations (see Section 6). Many of these peephole optimizations stem from re-writing `ApplyPermute` to `ApplyRoll` in specific, yet common patterns in our layout IR. Moreover, `ApplyRoll` lowers to a structured set of HE operations that makes it amicable to further optimization opportunities.

3.1.5 Search Space Heuristics

Rotom uses a top-down enumeration search to find a layout assignment. However, enumerating all possible layouts is extremely expensive as there are numerous layout options to consider. To prevent search-space explosion, Rotom employs 4 symmetry-breaking, cost-based heuristics (see Section 7.2) that either restrict when layout conversions are applied or prune costly HE kernels that are not along the Pareto frontier of existing solutions.

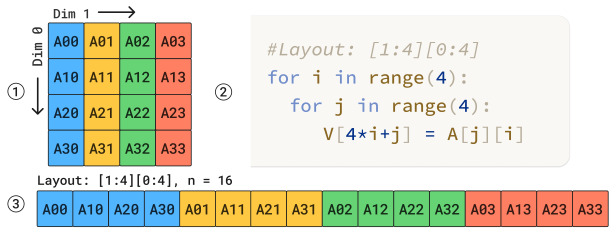
While Rotom does not guarantee finding the optimal solution, as it would be infeasible to search the entire possible space of layouts, it's approach finds the best solution within its restricted search space. Rotom's search heuristics only removes layout assignments that are *strictly worse* than a found solution. These heuristics improves search speed while maintaining search quality.

3.1.6 Lowering to OpenFHE

After finding a layout assignment, Rotom lowers its layout assignment (in layout IR) into an HE program. The HE program is composed of an HE operator IR, which is a one-to-one match of the HE operations exposed by backend HE libraries. This low-level IR acts as a common abstraction to connect Rotom to OpenFHE [2] and other frameworks (e.g., HEIR [13]).

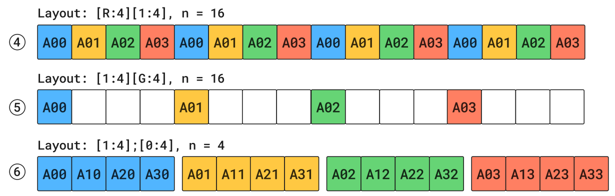
4 Preliminary: Layout Representation

This section describes the layout representation (or "layout" for short) that Rotom uses to define how tensors are packed into ciphertexts. Like prior systems [1, 18, 34, 47], Rotom


Dim 1
Dim 0
①
②
③
A00 A01 A02 A03
A10 A11 A12 A13
A20 A21 A22 A23
A30 A31 A32 A33
Layout: [1:4][0:4], n = 16
A00 A10 A20 A30 A01 A11 A21 A31 A02 A12 A22 A32 A03 A13 A23 A33

```
#Layout: [1:4][0:4]
for i in range(4):
  for j in range(4):
    V[4*i+j] = A[j][i]
```

(a) ① presents tensor A. ② describes the nested loop traversal over A for a column-major packing. ③ shows how A is packed column-wise into a $n = 16$ ciphertext.


Layout: [R:4][1:4], n = 16
④
A00 A01 A02 A03 A00 A01 A02 A03 A00 A01 A02 A03 A00 A01 A02 A03
Layout: [1:4][G:4], n = 16
⑤
A00 A01 A02 A03
Layout: [1:4];[0:4], n = 4
⑥
A00 A10 A20 A30 A01 A11 A21 A31 A02 A12 A22 A32 A03 A13 A23 A33

(b) ④ shows the 1st row of A repeated 4 times. ⑤ shows the 1st row of A separated by gap slots. ⑥ shows how the column packing of A is split into 4 ciphertexts with $n = 4$

Figure 3: Different packings using our layout representation.

uses a high-level layout representation to easily reason about different layout assignments and conversions between layouts. Rotom adopts the layout representation introduced by Viaduct-HE [47] as both systems use rolls (further detailed in Section 4.3). Rotom's main difference is in formalizing a lightweight conversion operator and abstractions that enable Rotom to efficiently apply rolls to any layout.

A layout defines a mapping between ciphertext(s) slots and tensor elements. It is composed of three components: a list of *traversal dimensions*, a *segmentation* operator, and a list of *rolls*. Together, these ingredients help Rotom abstract over a wide range of layouts to find an efficient layout assignment.

To illustrate how to interpret a layout, we use Figure 3 as a running example. Part ① shows a 4×4 tensor, A, with tensor dimensions 0 and 1 for the rows and columns, respectively. As we explain each layout component, we show how to pack A into different layouts.

4.1 Traversal dimensions

A layout representation's most basic goal is to dictate how a plaintext tensor is packed into ciphertexts. We do so by using a *traversal dimension*, which is defined as a sequence of *tensor indices* (materialized into tensor elements) along a tensor dimension.

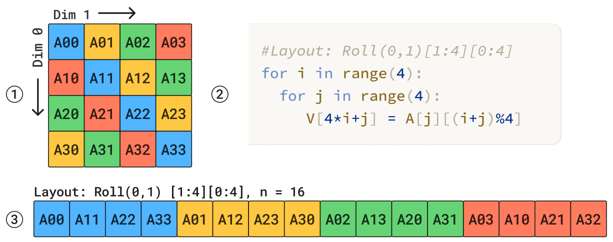
The notation for a traversal dimension is `[dim:extent:stride]`, where `dim` denotes the tensor dimension, `extent` denotes the length of a traversal, and `stride` denotes the step-size of the traversal (a stride of 1 can be omitted from the representation). Note that extents are padded to the closest power-of-two; this practice is commonly used in other systems [34, 47] as it simplifies layout conversions and provides a more natural packing into ciphertexts, which also have a power-of-two number of slots. To represent an iteration over the first column of A (in blue), we write `[0:4:1]` (or `[0:4]`). In this iteration, the `dim` being traversed is 0, since we iterate down a column; the `extent` of the traversal is 4 elements; and the `stride` of the traversal is 1, because we intend to iterate over every element without skipping any of them. Since dimension 1 is not specified in our layout, the default value for the traversal is 0. This layout, `[0:4]`, corresponds to the *tensor indices*: `{(0,0),(1,0),(2,0),(3,0)}`.

A multi-dimensional tensor is compactly represented as a *nested* loop iteration using multiple traversal dimensions. Let's consider packing A column-wise into a single $n = 16$ ciphertext. To do so, we write `[1:4][0:4]`. Part ② shows the tensor indices of the layout represented with a nested loop; i indexes the `dim:1` of A and j indexes `dim:0` of A, shown by $A[j][i]$ in the loop body. The extent and stride of both the loops and traversal dimensions are 4 and 1, respectively. As shown in part ③, this layout results in the *tensor indices*: `{(0,0),(1,0),(2,0),...,(1,3),(2,3),(3,3)}`.

To represent repeated and zeroed-out slots, Rotom also makes use of two special traversal dimensions: repeated and gap dimensions. In both cases, the extent of the traversal dimension defines the length of either the repetition or zero-filled slots; stride is omitted as it has no effect to the layout. A repeated dimension is used to repeat values within a ciphertext, denoted by `[R:extent]`, where the `R` stands for repeated. For example, part ④ shows how `[R:4][1:4]` maps to the first row of A repeated 4 times, since the extent of `[R:4]` is 4. An empty dimension is used to represent zero-filled elements within a ciphertext, denoted by `[G:extent]`, where `G` stands for gap slots. Part ⑤ shows how `[1:4][G:4]` maps to the first row of A spaced out by a step-size of 4. Here, the *non-zero* indices represented by `[G:4]` (e.g., tensor indices: `{(0,1),(0,2),(0,3)}`) are zeroed out.

4.2 Segmentation

A ciphertext has a fixed number of slots, n, and traversal dimensions that iterate more than n elements must be packed into multiple ciphertexts. Rotom uses segmentation to separate traversal dimensions into *vector dimensions* and *slot dimensions*. This is done using the "`;`" operator. Traversal dimensions before "`;`" constitute vector dimensions, and traversal dimensions after "`;`" constitute slot dimensions. The vector dimensions define *which* ciphertext a tensor index is mapped to, and slot dimensions define the slot position *within* that ciphertext. If a layout representation does <u>not</u> have a segmentation operator, then all traversal dimensions are slot dimensions and all data can fit within a single ciphertext. Part ⑥ shows a segmented column-major layout `[1:4];[0:4]`, where each column of A is packed into its own ciphertext.


Dim 1
Dim 0
①
②
③
A00 A01 A02 A03
A10 A11 A12 A13
A20 A21 A22 A23
A30 A31 A32 A33
Layout: Roll(0,1) [1:4][0:4], n = 16
A00 A11 A22 A33 A01 A12 A23 A30 A02 A13 A20 A31 A03 A10 A21 A32

```
#Layout: Roll(0,1)[1:4][0:4]
for i in range(4):
  for j in range(4):
    V[4*i+j] = A[j][(i+j)%4]
```

Figure 4: ① presents the input matrix A with dimensions 0 and 1, each with extent 4. ② describes the diagonal-major layout, using a roll, and the corresponding loop traversal over A. ③ shows how A is packed diagonally into a single ciphertext with $n = 16$.

4.3 Rolls

Originally introduced by Viaduct-HE [47], rolls were used as a preprocessing step to pack ciphertexts into more complex layouts. As demonstrated in Figure 1, complex layouts can result in more efficient layout assignments that reduce data movement costs.

At a high level, rolls cyclically rotate slot elements along a specified traversal dimension in a layout, akin to how the *np.roll* [27] function shifts plaintext tensor elements along a specified axis. In a layout representation, a roll is defined by `Roll(i,j)`, where `i` and `j` represent *the index of* the traversal dimension *to roll* and the traversal dimension to *roll by*. Rolling `i` by `j` will update the tensor indices of `i` to be the result of a modular addition with the tensor indices of both traversal dimensions, modulo the extent of the traversal dimension at `i`.

Figure 4 illustrates how a roll can be used to represent a diagonal layout, `Roll(1,0)[1:4][0:4]`. In `Roll(1,0)`, `i`= 1 which represents `[0:4]`, and `j`= 0 which represents `[1:4]`. `Roll(1,0)` performs a modular addition between the tensor indices of `[0:4]` and `[1:4]`, while leaving `[1:4]` unchanged. Part ② shows how this roll modifies the tensor indices, i.e., `(i+j)%4` in the loop body. As a result, the tensor indices of `[0:4]` changed from `{0,1,2,3,0,1,2,3,...}` to `{0,1,2,3,1,2,3,0,...}`. This roll is the same as performing a cyclic rotation of each column by its column index (i.e., shifting the 0th column by 0, shifting the 1st column by 1, and so on), resulting in the diagonal layout.

5 Layout Alignment

The previous section presents our high-level layout representation that Rotom uses to explore a wide range of layouts. In this section, we explore how Rotom uses layout alignment rules (or "alignment" for short) on its layouts to correctly and deterministically generate an HE kernel from a tensor operator. Alignment checks whether operand layouts satisfy the logical requirements of a tensor operator in HE and determines if layout conversions are needed.

Prior HE compilers use alignment to determine how HE kernels are implemented and their associated cost. By determining the cost, compilers can then reason about different layout choices when searching for a layout assignment. CHET [18], HECO [50], and HE Layers [1] all use alignment implicitly by predefining HE kernel implementations or restricting available layout choices. While this approach makes it easy find a layout assignment, the restricted layouts can result in expensive conversions. To reduce conversion costs, both Fhelipe [34] and Viaduct-HE [47] use a more flexible layout representation. These systems use alignment explicitly, constructing HE kernels in two steps. First, they align operand layouts using layout conversions operators, then use HE operators to perform the tensor operator.

While Viaduct-HE supports rolled layouts (Fhelipe does not), it does not have high-level abstractions to reason about alignment with rolls. To determine the implementation (and cost) of an HE kernel, Viaduct-HE fully materializes the large underlying ciphertext vectors, greatly hindering its compilation speed on large tensor inputs. Rotom alleviates this problem by formalizing alignment over its high-level layout representation.

5.1 How Rotom Uses Alignment

When Rotom constructs an HE kernel, operand layouts for binary tensor operators may require layout conversions to realign the slots values for high slot utilization. Alignment helps determine which traversal dimensions and rolls in a layout are misaligned, and what layout conversions are necessary to realign the layouts. Additionally, alignment standardizes how HE kernels are constructed by separating layout conversion operators (e.g., `ApplyPermute`) from operational operators (e.g., `MatMul`). Since all layout operators have a fixed implementation, Rotom can then easily reason about both the cost and lowering of an HE kernel *directly from its layout representation*.

5.2 Dimension Alignment

Recall that Rotom uses traversal dimensions in a layout to represent how to iterate over a tensor. When encountering a binary tensor operator, the traversal dimensions in the operand layouts have to be *aligned* to guarantee high-slot utilization and good performance. Different tensor operators have different alignment constraints, and these constraints are derived from a nested loop representation of the tensor operator.

To give a concrete example using Einstein's notation, $C_{ij} = \sum_k A_{ik}B_{kj}$ represents a possible nested loop computation for the `MatMul` operator, where i, j, and k are the loop iterators. In this example, assume that all loop iterators have an extent of 4. For tensor A, i indexes the 0th dimension, k indexes the 1st dimension, and j repeats the value denoted by A_{ik}. Following the loop iterator order of i, j, and k, this traversal is represented with the layout A: `[0:4][R:4][1:4]`, where j is denoted with a repeated dimension. Using the same loop iterator order for B, the aligned traversal is represented with the layout B: `[R:4][1:4][0:4]`. Since A and B both share k, the A : `[1:4]` is aligned to the B : `[0:4]`, where these traversal dimensions materialize to the same tensor indices. Notably, alignment does not fix the ordering of a layout's traversal dimensions, but rather the relative positioning of the traversal dimensions. For example, another set of aligned layouts can be implemented by swapping the order of i and k, resulting in A: `[1:4][R:4][0:4]` and B: `[0:4][1:4][R:4]`.

5.3 Roll Alignment

As Rotom is the first compiler to support the `ApplyRoll` operator, Rotom generalizes its alignment rules to handle rolled layouts. Recall that rolls shift the tensor elements within a ciphertext, effectively modifying the tensor indices of the layout. If two operand layouts are dimension aligned but have different rolls applied, then the rolls could break alignment. Rotom's alignment rules for rolls guarantee that shared traversal dimensions must have the same roll applied.

Using the same matrix multiplication example, layouts A: `Roll(0,1)[1:4][R:4][0:4]` and B: `[0:4][1:4][R:4]` are not aligned for `MatMul` because A:`[1:4]` is rolled whereas B:`[0:4]` is not. Since this is a shared dimension, alignment dictates these traversal dimensions need to have the same tensor indices. To align these two layouts, Rotom uses `ApplyRoll` to apply the same roll, `Roll(0,1)`, to B. Not all rolls need to be matched in order to reach alignment. A rolled traversal dimension that is aligned with a repeated dimension does not disrupt alignment, as the modified tensor indices still map to the same repeated tensor indices, despite any shifts. For example, layouts A: `Roll(0,1)[0:4][R:4][1:4]` and B: `[R:4][1:4][0:4]` are aligned for `MatMul`, even though B does not have a rolled layout.

6 Improving Layout Search Quality via Rolls

With alignment, Rotom computes the cost of an HE kernel by summing *conversion costs* (cost to convert a layout into an aligned layout in HE) and *operation costs* (cost to perform the tensor operator in HE). In this section, we explain how Rotom uses our new `ApplyRoll` operator to find optimizations that improves both fronts.

One common component to achieving alignment is by swapping the positions of two traversal dimensions. Rotom's first insight stems from designing a new `ApplyRoll` operator that can inexpensively perform this swap to align layouts (Section 6.1). In addition, we observe that `ApplyRoll` lowers to a structure set of HE operators, namely rotations with a fixed step-size and masking with evenly spaced slots. Rotom's second insight is that this structure makes `ApplyRoll` amicable to more optimization opportunities that can further minimize data movement costs (Section 6.2). We implement all of these insights as peephole optimizations in Rotom's compiler pipeline.

6.1 `ApplyRoll` Swaps Traversal Dimensions

To align layouts, prior works introduced conversion operators that are relatively expensive. Rotom formalizes these operators as `ApplyPermute`, `ApplyReplication`, and `ApplyCompact`. `ApplyPermute` is a conversion operator that can be used to arbitrarily permute the order, extent, and strides of a layout's traversal dimensions. To swap two traversal dimensions with `ApplyPermute`, misaligned slot dimensions are first masked into individual ciphertexts, then the gap slots are filled either by `ApplyCompact` (rotating and adding different ciphertexts together) or `ApplyReplication` (rotating and adding the same ciphertext together).

In Rotom, we find that using `ApplyRoll` is not only useful for creating complex layouts, but also as a cheap layout conversion operator. Our key insight is that rolling any traversal dimension by a *repeated dimension* (commonly seen in tensor operators such as matrix multiplication) can swap the two traversal dimensions. Recall that a roll modifies the tensor indices of a traversal dimension by applying a modular addition with another traversal dimension's indices. The intuition behind this conversion is that modular addition is commutative, thus swapping the order of the traversal dimensions does not change the resulting tensor indices. This property only holds when one of the rolled traversal dimensions is a repeated dimension; otherwise, swapping two traversal dimensions could potentially change the tensor indices of the *roll by* dimension. For example, `Roll(1,0)[R:4];[0:4][1:4]` is equivalent to `Roll(0,1)[0:4];[R:4][1:4]`, where traversal dimensions `[R:4]` and `[0:4]` are swapped. Rotom uses this insight to do local re-writes of `ApplyPermute`: when it encounters an `ApplyPermute` operator that swaps a traversal dimension with a repeated dimension, it will replace `ApplyPermute` with `ApplyRoll`.

While both `ApplyPermute` and `ApplyRoll` can swap dimensions (and thus be used for alignment), these two operators lead to very different conversion costs. Figure 5 illustrates how these operators are used to swap `[1:4]` with the repeated dimension `[R:4]` in a repeated column-major layout `[R:4];[1:4][0:4]`. In both cases, all slot positions end with the same values *across the ciphertexts*. Part ① shows the conversion using `ApplyPermute`; each column (color) is first masked, then replicated within each ciphertext. The conversion results in 4 multiplications, 8 rotations, and 8 additions, as the replicate step requires a rotate-and-sum routine for each ciphertext. Part ② shows the conversion using `ApplyRoll`; each column is rotated internally within the ciphertext. This process requires 1 mask, subtraction, and rotation to get each rotated partition; the partitions are then compacted together. This conversion uses 3 multiplications, 6 rotations, and 6 additions. Crucially, `ApplyRoll` avoids the *Replicate* step used by `ApplyPermute`, which saves on rotations as the extents of tensor dimensions and number of slots *increases*.

In this example, `ApplyRoll` can be further reduced to only use 3 HE rotations. This optimization works when an `ApplyRoll` rolls the leftmost slot dimension by a repeated vector dimension. The intuition is as follows. If we can ensure all tensor indices within a ciphertext are shifted by the same amount, then the conversion can be lowered to just rotations. Since the leftmost slot dimension defines contiguous segments of values within a ciphertext, rolling by a vector dimension adds a fixed offset to each segment, fulfilling our condition. When Rotom encounters this pattern, it re-writes the `ApplyRoll` operator to the `ApplyRotRoll` operator.

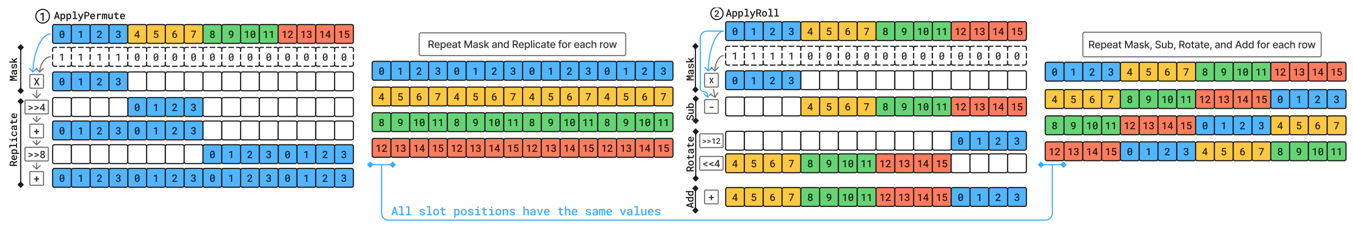
6.2 Reducing Rotations with `ApplyRoll`

With the introduction `ApplyRoll`, Rotom identifies additional opportunities to apply the well-known baby-step giant-step (BSGS) optimization [24] to greatly reduce the number of rotations. The insight behind this optimization stems from the regular structure of HE operators used to implement `ApplyRoll`, which includes a set of rotations with a fixed step-size and multiplications evenly-spaced 0-1 bit-masks. Rotom leverages this structure to create two new peephole optimizations that re-writes `ApplyRoll` into `ApplyBSGSRoll` and `ApplyBSGSMatMul`.

6.2.1 Baby-step Giant-step Optimizations

BSGS is an optimization that reduces the overall number of rotations from $O(n)$ to $O(\sqrt{n})$. BSGS can be used when it pattern-matches on two constraints. First, a ciphertext is rotated multiple times by a progressively increasing offset (i.e., rotations by k, $2k$, ... nk). Second, rotated ciphertexts follow a multiply-and-accumulate pattern (e.g., multiplication with plaintext masks then a compaction step). Rather than performing all individual rotations, this optimization decomposes them into two rotation groups: the baby-step group and the giant-step group. First, the baby-step group is processed by using rotations, multiplications, and partial summations. The intermediate results are then rotated by the giant-step group and summed again to produce the final result. By amortizing rotations across the baby-step group, this optimizations reduces the total number of rotations by a square root factor.

Rotom applies BSGS when `ApplyRoll` is used on two slot dimensions, re-writing the conversion operator to `ApplyBSGSRoll`. In this scenario, `ApplyRoll` is implemented by first masking the original ciphertext into partitions, rotating each partition (with an increasing step-size) to its desired location, and compacting the partitions together. This operation exhibits the rotate, multiply, and accumulate structure that BSGS is designed to optimize. Additionally, when `ApplyRoll` is



① ApplyPermute
Mask
Replicate
Repeat Mask and Replicate for each row
All slot positions have the same values
② ApplyRoll
Mask
Sub
Rotate
Add
Repeat Mask, Sub, Rotate, and Add for each row

Figure 5: Comparison of `ApplyPermute` and `ApplyRoll` to swap two traversal dimensions.

used with a matrix multiplication, Rotom will rewrite both the conversion and `MatMul` operator to `ApplyBSGSMatMul`. The computation also matches the BSGS pattern; `ApplyRoll` creates rotations with an increasing step-size and `MatMul` follows the multiply-and-accumulate structure.

6.2.2 Moving summation dimensions to vector dimensions

Many tensor operators (e.g., matrix multiplication) include a summation of values along a tensor dimension as a sub-operation. Within a layout, we call the traversal dimension to be summed the *summation dimension*. A summation dimension can either be a vector dimension or a slot dimension. We observe summations along vector dimensions is more performant as it can be done by directly adding ciphertexts together. On the other hand, summation along slot dimensions uses a rotate-and-sum routine, which not only requires additional rotations, but also leaves behind garbage values that mandates additional masking. This is one reason why *diagonal packing* is efficient for matrix multiplication.

Taking advantage of this observation requires efficient layout conversion operators that can cheaply move the appropriate traversal dimensions to be vector dimensions. Prior works could not use this observation because their layout conversions (`ApplyPermute`) were too expensive, and the cost of conversion outweighed any performance benefits from moving traversal dimensions. Conversely, `ApplyRoll` is often cheap enough to warrant applying this optimization to generate more efficient HE kernels.

Consider the example $A \cdot B$, where A is the ciphertext matrix and is initially column-packed and replicated, with layout `[R:4];[1:4][0:4]`. B is a plaintext matrix, which means changing the layout of B to align it to A's layout comes at no additional cost. The summation dimension for A is `[1:4]`. Moving it to the vector dimension requires swapping `[1:4]` and `[R:4]`. Using `ApplyPermute` offers no performance benefits compared to summing within the slot dimensions, as both HE kernels result in the same cost. In both cases, the HE kernel requires 4 multiplications, 4 rotate-and-sum routines, 4 masks, and a final compaction step. Instead, Rotom swaps the two traversal dimensions using `ApplyRotRoll`, which uses only 3 rotations and avoids the rotate-and-sum, mask, and compaction step. This operation can be further optimized with BSGS as the size of the input tensors increases.

6.2.3 Strassen's Algorithm

Strassen's algorithm [29] reduces `MatMul` compute costs using a divide-and-conquer approach that recursively operates over tiled input matrices. In Rotom, we take advantage of Strassen's to improve the runtime of ciphertext-ciphertext (ct-ct) matrix multiplication when an input tensor is tiled into 4 square tilings that span 4 ciphertexts. Rather than performing 8 matrix multiplication across all pair-wise tiles, Strassen's algorithm reduces multiplications at the cost of additional addition and subtraction operations. Fortunately, in HE, addition (and subtraction) is much cheaper than multiplication. Using Strassen's reduces the number of ciphertext multiplications in half. While Strassen's algorithm reduces arithmetic complexity, paired with `ApplyRoll` and `ApplyBSGSRoll`, Rotom further reduces cost of conversions. Using `ApplyRoll` and its optimizations, Rotom improves over baseline ct-ct matrix multiplication by $\approx 6\times$, whereas Strassen's with `ApplyPermute` only improves performance by $1.27\times$ (detailed in Table 3).

7 Layout Assignment in Rotom

The previous sections present techniques for aligning and optimizing HE kernels using our `ApplyRoll` operator. In this section, we explore how Rotom uses these techniques in conjunction with search space heuristics to find an efficient layout assignment.

Previous approaches either limited the search space to simple layouts—leading to suboptimal assignments, or employed a bottom-up strategy that exhaustively explored a wide variety of complex layout assignments—resulting in compile times that took more than 24 hours. To achieve competitive search times, Rotom introduces 4 cost-based symmetry-breaking heuristics to prune costly layout assignments. Rotom outperforms prior approaches because it retains search quality using complex rolled layouts without sacrificing search time.

7.1 How Rotom Works

Rotom finds an efficient layout assignment using a top-down enumeration with search space heuristics. To begin, Rotom uses a single topological traversal over its tensor IR. When

Rotom encounters a tensor operator, Rotom generates candidate HE kernels—each with different input/output layouts and costs. Rotom builds each HE kernel *iteratively* by using the output layouts from the previous tensor operator (or with an initial compact row/column layout) as input layouts to the next tensor operator. This process guarantees that each newly generated kernel inherits an input layout that exactly matches the output layout of a previously generated kernel, ensuring consistent layouts across kernel dependencies. Given two input layouts, Rotom uses alignment to deterministically construct correct HE kernels for each tensor operator such that the resulting HE kernel has high slot utilization and good performance. Next, Rotom uses its peephole optimizations to improve the cost of each candidate HE kernel. To avoid long search times, Rotom applies heuristics to prune expensive HE kernels and eliminate symmetries among equivalent ones. Lastly, Rotom picks the layout assignment—a map of each tensor operator to an HE kernel—with the least overall cost.

7.2 Cost-based Symmetry-breaking Heuristics

While the proposed approach finds the optimal solution within the search space, it's still not scalable. This is because when building an HE kernel, Rotom uses the output layouts from the previous kernel as input layouts to the next kernel, causing an exponential blowup based on the depth of the tensor program. To keep layout assignment scalable, Rotom introduces four symmetry-breaking heuristics designed to prune the search space and ensure scalability. The heuristics are as follows:

Cost-based Pruning. To avoid evaluating all possible layout assignment options, Rotom keeps the most efficient HE kernel for each unique output layout pairing. Since Rotom builds each HE kernel iteratively, keeping the most efficient HE kernel for each unique output layout retains the Pareto frontier of HE kernels. Thus, Rotom is only removing HE kernels that are a strictly worse solution that an existing solution. This heuristic does not affect the optimality of the search.

Selective application of `ApplyRoll`. Rotom only uses `ApplyRoll` when they are necessary: either to achieve layout alignment or to move a summation dimension into the vector dimension, which can enable cheaper homomorphic operations. By only applying cost-efficient rolls, Rotom prevents arbitrary applications of rolls from polluting the layout search space. Intuitively, randomly applying arbitrary rolls to HE kernels is not beneficial as it pollutes the search space and adds additional conversion costs with no benefits.

Tiling and Compaction Heuristic. To further constrain layout diversity, Rotom applies a heuristic when encountering `ApplyCompaction`, which compacts multiple ciphertexts with gap slots into fewer ciphertexts. Instead of compacting vector dimensions arbitrarily, Rotom will try to group vector dimensions with slot dimensions that share the same tensor dimension (i.e., the same `dim` in a traversal dimension). For example, given a layout `[1:4][0:4:4];[0:4][1:4:4][G:4]`, Rotom would compact the layout into `[0:4:4];[0:4][1:16]` (a row-major packing) over `[1:4];[0:4][1:4:4][0:4:4]` (a tiled column-major packing). This is because many downstream tensor operations, such as matrix multiplication, perform reductions along a tensor dimensions. Packing same dimensions together will result in less internal fragmentation (uneven runs of gap slots) among traversal dimensions.

Canonical Vector Dimension Ordering. Rotom enforces a strict ascending order on vector dimensions of each layout. This ordering does not affect the packing of each individual ciphertext, but rather standardizes the order of multiple ciphertexts within a layout. For example, layouts `[0:4:4][1:4:4];[0:4][1:4]` and `[1:4:4][0:4:4];[0:4][1:4]` contain the exact same ciphertexts and differ only in their order. By canonicalizing the order of vector dimension, Rotom eliminates redundant layout variants that are semantically equivalent, reducing the number of layout configurations considered during search.

7.3 Limitations

In this section, we describe the design limitations within Rotom. We leave exploring these directions to future work.

Search optimality: Rotom does not guarantee finding the most optimal solution from the space of *all* possible layouts. Rotom's layout representation only supports different tilings and rolled layouts. There could exist new conversion operators apart from `ApplyRoll` that could use unrepresented layouts with reduced data movement costs. Within Rotom's search space, it does guarantee finding the best layout assignment.

Power-of-two padding: Like Fhelipe and Viaduct-HE, Rotom pads each tensor shape to the nearest power-of-two, which works well for common real-world workloads. However, this could theoretically lead to wasted slots and suboptimal packings. For example, a 3×9 tensor could fit within a $n = 32$ ciphertext; however, Rotom will pad to 4×16 and use $n = 64$. One way we would address this limitation is to relax our padding constraints so traversal dimension extents are no longer restricted to a power-of-two. We would also need to update both roll alignment rules and `ApplyRoll` to support rolling by irregular extents. These changes would allow Rotom to find hybrid diagonal packings that could fit a 3×9 tensor in a diagonal packing within a $n = 32$ ciphertext.

Slot-based encoding: Rotom only handles slot-based encoding methods, yet there have been new hand-tuned protocols that use coefficient-based encoding to perform cheaper tensor operations [4,31,38]. Finding a common abstraction that bridges the gap between both encoding formats could lead to even cheaper layout assignments.

Bootstrap placement: Rotom primarily research goal focuses on finding an efficient layout assignment. Rotom does not handle placing noise management operations, such as bootstrap, to enable FHE workloads. Instead, Rotom can be used as a tensor frontend and lowered to other FHE compiler frameworks.

8 How to use Rotom in FHE/MPC compilers

Rotom is a compiler framework for generating efficient layout assignments. It can be used with existing fully homomorphic encryption (FHE) and hybrid MPC compilers, or used as a standalone tool to help users write and optimize secure computation workloads. In this section, we describe how Rotom can be integrated into existing FHE compilers [1, 13, 47, 50], and how a developer can use Rotom to design a 2PC protocol for transformers inference.

Using Rotom in existing FHE compilers There are two types of FHE compilers that exist today. Some compilers [5, 11, 12, 16, 17, 23, 30, 36, 37, 51] do not automatically pack ciphertexts and instead rely on users to explicitly specify the packing schemes. Users write programs directly using primitive HE operators—additions, multiplication, and rotations—and the compiler lowers those programs to an HE operator IR. Then, the compiler runs FHE-related passes, like rescale and bootstrap placement, on the HE operator IR.

To extend to one of these compilers (HEIR [13]) *without* automated packing, we integrate Rotom as a tensor frontend and layout optimizer. Users write programs in Rotom's DSL, and Rotom lowers the resulting layout assignment to the existing compiler's HE operator IR; this is a simple lowering pass, since the target IR consists of standard HE operations with no extra ciphertext maintenance operations (e.g., noise and scale management). For each compiler, Rotom's lowering pass requires less than 250 lines of Python.

A second category of FHE compilers introduces mechanisms to automate packing [3, 8, 9, 14, 18, 19, 34, 39, 47, 50, 51]. They typically compile from a higher-level DSL to an IR that can express ciphertext packings; after running any optimizations on this IR, the compilers lower to an HE program, and run noise management and optimization passes on the HE program. To extend these compilers with a more expressive layout IR and more advanced layout optimizations, we can once again integrate Rotom as a higher-level frontend. Users write programs in Rotom's DSL, and Rotom lowers these programs to its own layout IR, and then to an existing compiler's HE operator IR—yielding a combination of Rotom 's layout optimizations and the existing compiler's backend optimizations. Our example Rotom frontend for HECO requires under 250 lines of Python.

There are two compilers in this category with which Rotom doesn't easily interoperate, since these compilers deviate from the standard automated HE compiler pipeline. Integrating Rotom into Fhelipe would be difficult as Fhelipe's layout representation is limited and does not support rolls. Rotom can generate (better) layouts that cannot be represented in Fhelipe's layout IR. Rotom also cannot connect to Orion, as this framework does not support an IR to target. Instead, Orion compiles tensor operations written in PyTorch to directly Lattigo [42].

Using Rotom in manually optimized protocols Developers and expert cryptographers can use Rotom to design manually optimized 2PC protocols. For example, Bolt [43] presents a hybrid 2PC protocol for transformers inference that uses both HE and secret sharing. One of the paper's contributions is a new ciphertext packing scheme for matrix multiplication in transformers. Rotom can *automatically generate* Bolt's new packing schemes; we expand in our evaluation.

We ran an informal investigation to answer the question, "how easy is it to use Rotom in a manual workflow to generate an optimized ciphertext packing?" To answer this question anecdotally, we asked 2 Ph.D. and 1 undergraduate students to write the attention layer from BERT using Rotom's DSL. On average, the students took around 5 minutes to write this function; where ≈60% of the time was spend on understanding how to use reshape and permute functions. One student remarked on how closely the interface resembled PyTorch and was surprised by Rotom's ease-of-use.

9 Implementation

We implement Rotom in $\approx 12,000$ lines of Python code. Rotom targets the CKKS implementation of the OpenFHE library [2]. Users can define the input scales and the degree of a ciphertext (i.e., n). All other cryptographic parameters are abstracted away from the user and automatically chosen by OpenFHE to ensure 128 bits of security. In addition, Rotom relies on the noise and scale management passes available in OpenFHE to automatically insert ciphertext maintenance operations (rescales, relinearization, and modswitches).

10 Evaluation

To evaluate Rotom, we run benchmarks to determine the *scalability* of Rotom's compilation pass and *search quality* of the layout assignments. We use tensor benchmarks that are already present in the HE compiler literature or exist as a component from a secure inference protocol. These benchmarks use large tensor sizes (64 – 1024) that are equivalent to realistic workloads. Lastly, we evaluate Rotom against two state-of-the-art HE tensor compilers, Fhelipe [34] and Viaduct-HE [47]. To perform a controlled experiment, we run the compiled programs from all systems on OpenFHE, keeping all cryptographic parameters consistent across all benchmarks.

Evaluation Setup We compiled and executed all of our benchmarks on a 64-core Intel Xeon CPU with 256 GB of RAM. To measure the cost of sending ciphertexts over the network, we model a LAN network with bandwith 1Gbps, and a WAN network with bandwidth 100Mbps. Each benchmark result is the average over 5 runs with the relative standard error below 5%.

10.1 Microbenchmarks

To determine the effectiveness of our novel `ApplyRoll` layout operator, we conduct two microbenchmarks that compare the execution time of `ApplyRoll` (and their optimized versions) against `ApplyPermute`. Additionally, we show how `ApplyRoll` reduces conversion costs when using Strassen's algorithm for HE ct-ct `MatMul`.

Operator	Add	Mul	Rot	Time [s]	Speed up
`ApplyPermute`	384	64	447	1.18	-
`ApplyRoll`	126	127	63	0.23	5.13×
`ApplyRotRoll`	0	0	63	0.14	8.43×

Table 1: Performance comparison of swapping the leftmost slot dimension with a repeated vector dimension.

Table 1 compares swapping the leftmost slot dimension with a vector dimension using `ApplyPermute`, `ApplyRoll`, and `ApplyRotRoll`. The benchmark is initialized with a 64×64 tensor, replicated in a row-major layout: `[R:64];[0:64][1:64]`. The goal is to swap the positions of `[R:64]` and `[0:64]`. Our results show how using `ApplyRoll` and `ApplyRotRoll` greatly reduces layout conversion costs, improving conversion performance by 5.13× and 8.43× respectively. From the table, we can clearly see how `ApplyRoll` and `ApplyRotRoll` reduces the total number of HE operations, namely rotations, by avoiding the rotate-and-sum routine used in `ApplyPermute`.

Operator	Add	Mul	Rot	Time [s]	Speedup
`ApplyPermute`	64	69	132	0.36	-
`ApplyRoll`	126	127	126	0.35	1.02×
`ApplyBSGSRoll`	126	127	28	0.11	3.27×

Table 2: Performance comparison of swapping a traversal dimension within slot dimensions.

Table 2 compares swapping two slot dimensions within a single ciphertext. Again, the benchmark is initialized with a row-major 64×64 tensor: `[0:64][1:64]`. The objective is to swap the `[1:64]` into the position of `[0:64]`. For `ApplyPermute`, the resulting layout is `[1:64][R:64]`, whereas for the roll layout operators, the resulting layout is `Roll(1,0)[0:64][1:64]`. The table shows that performing `ApplyPermute` and `ApplyRoll` is quite similar in performance, where much of the runtime is dominated by rotations. However, by applying baby-step giant-step in `ApplyBSGSRoll`, we see a 3.27× improvement by reducing the number of rotations by a square-root factor.

Our final microbenchmark illustrates how Strassen's can be made more applicable in HE applications by reducing conversion costs with `ApplyRoll`. We evaluate a ct-ct `MatMul` workload with two 128×128 tensors and $n = 4096$. For `Strassens`, the inputs are given as four 64×64 squares tiles, whereas for `MatMul`, Rotom is left to specify the input layouts. `Matmul` w/o Roll is comparable to implementations found by prior works.

Operator	Add	Mul	CMul	Rot	Time [s]
`MatMul` w/o Roll	4224	768	512	4344	19.78
`Strassens` w/o Roll	3268	512	256	3328	15.52
`MatMul` w/ Roll	1024	128	512	1148	5.78
`Strassens` w/ Roll	1344	889	256	574	3.12

Table 3: Performance comparison of Strassen's Algorithm.

Table 3 shows how expensive our baseline of `MatMul` w/o Roll is, using thousands of rotations for both operational and conversion costs. By applying only `Strassens`, Rotom can reduce the number of ciphertext multiplications in half, and the number of rotations by ≈1,000, gaining a slight improvement in runtime. Using a combination of our roll optimizations and `Strassens`, Rotom finds an HE implementation that is 6.34× faster than `Matmul` w/o Roll.

10.2 Application-level benchmarks

Rotom evaluates on a suite of 7 tensor workloads that exist in real-world applications and compares the layout assignment to both Fhelipe [34] and Viaduct-HE [47]. Note that Fhelipe is targeted at FHE workloads, and thus they also introduce bootstrap placement. On the other hand, Rotom's goal is to build a compiler framework that automatically vectorizes tensor workloads for HE. Rotom compares to Fhelipe's vectorization pass; though, our contributions are complementary to their bootstrap placement strategy. Our benchmarks are as follows:

1. **Distance** [47] computes the euclidean distance between a target point and 128 test points, each with 64 dimensions.
2. **MatMul** [47] performs a single ct-pt `MatMul` between A and B, where A is encrypted and B is not. For $n = 8K$, A: 128 × 64, B: 64 × 128; for $n = 32K$, all shapes are doubled.
3. **Double-MatMul** [47] performs two consecutive ct-ct `MatMul` between A, B, and C. For $n = 8K$, A, C: 128 × 64, B: 64 × 128. For $n = 32K$, all shapes are doubled.
4. **TTM** [34] is a tensor kernel benchmark introduced by Fhelipe. This benchmark computes the third-order tensor matrix product. All inputs have are 64 × 64.
5. **Convolution** [35] performs a single convolution over an 8-channel input *image* with a filter size of 3 and a stride of 1. For $n = 8K$, *image*: 32 × 32; for $n = 32K$, *image*: 64 × 64.

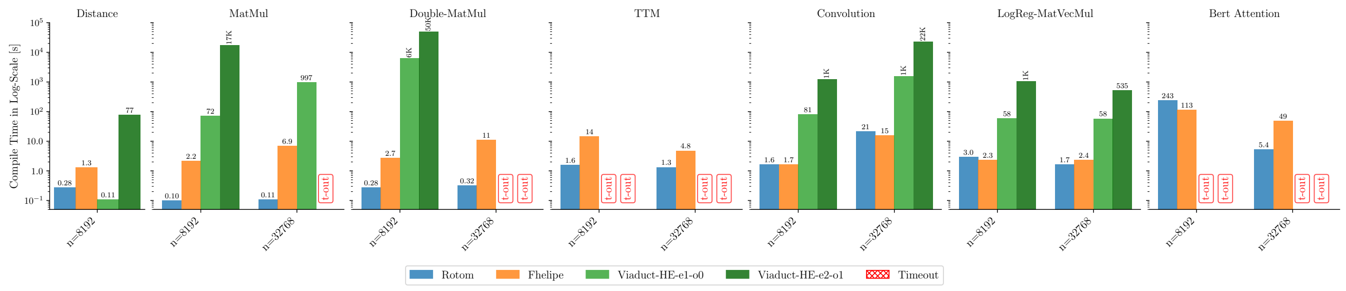


Figure 6: Compile time in Log-Scale. *t-out* indicates the benchmark did not compile within 24hrs.

6. **LogReg-MatVecMul** [25] This workload takes one of the 32 iterations and performs two consecutive matrix-vector multiplications with a batch size of 1024 and 197 features.

7. **Bert Attention** [41,43] performs the attention layer from the BERT-base model, where m=128, d=768, and H=12. This benchmark is highly complex, including both ct-pt and ct-ct `MatMul`, transpose, and tensor addition.

10.2.1 Rotom Achieves Fast Search Speed

Figure 6 details the compile time *in log-scale [s]* of Rotom compared to Fhelipe and Viaduct-HE. *t-out* (in red) denotes benchmarks that did not compile within 24 hours. Viaduct-HE has two compilation modes, e1-o0 and e2-o1, that trades off compilation speed for better search quality and circuit optimizations; we evaluate both modes.

Across all benchmarks, Rotom and Fhelipe find a layout assignment within 5 minutes. Viaduct-HE's layout assignment pass requires array materialization to derive rotated and masked ciphertexts from existing ciphertexts. This derivation and materialization procedure makes their compilation time dependent on the size of the input tensors. Viaduct-HE times out on larger benchmarks, such as BERT, where `MatMul` is performed with 128×768 and 768×768 tensors. By using `ApplyRoll` to derive rotated vectors directly from the layout abstraction, Rotom finds complex derivations from existing ciphertexts without materialization.

Rotom heuristics help control the search space of layout assignments explored. For example, Rotom explores only 7623 on BERT Attention. Without these heuristics, the number of layout assignments grows to over 50,000 after a few `MatMul` operators.

10.2.2 Rotom Achieves High Search Quality

To determine the efficiency of our layout assignments, we compare the execution time of our compiled benchmarks against that of Fhelipe and Viaduct-HE. Figure 7 shows *in log-scale* that Rotom either matches or outperforms Fhelipe and Viaduct-HE across all benchmarks. Each bar includes the communication costs of sending and receiving ciphertexts in both LAN and WAN settings (denoted by the diagonal hash bars). Notably on benchmarks that Viaduct-HE did not compile, Rotom outperforms Fhelipe on Double-MatMul by 8.6× and BERT by 80.3×.

Distance: Rotom matches the packing and performance of Viaduct-HE. Fhelipe doesn't find a replicated input layout, thus requires an additional rotate-and-sum routine for replciation.

MatMul: On $n = 8K$, Rotom outperforms both Fhelipe and Viaduct-HE-e2-o1 by 39× and 1.65× respectively. Both Rotom and Viaduct-HE find a diagonalized input packing. Rotom gains a slight performance gain by employing BSGS to reduce the number of HE rotations. On $n = 32K$, Rotom outperforms both by 20× and 34.29×,

Double-MatMul: On $n = 8K$, Rotom outperforms both Fhelipe and Viaduct-HE-e2-o1 by 7.45× and 1.49× respectively. This is because it is cheaper to send multiple ciphertexts than to convert the layout of a ciphertext in the LAN setting. Rotom and Viaduct-HE both optimize their HE kernels by moving summation dimensions to the vector dimensions. On $n = 32K$, Rotom outperforms Fhelipe by 8.6×.

Convolution: For $n = 8K$, Rotom matches Fhelipe's packing. Viaduct-HE fails to find a plaintext hoisting optimization that combines the masks to the plaintext weight matrix and uses 2× more multiplications. On $n = 32K$, both Rotom and Viaduct-HE-e2-o0 have similar packings and HE kernels that rely on sending multiple ciphertexts.

TTM/LogReg-MatMul: In both of these benchmarks, Rotom outperforms Fhelipe and Viaduct-HE, for similar reasons to the MatMul and Double-MatMul benchmarks.

BERT: For $n = 8K$, Rotom outperforms Fhelipe by 10×. Similarly with previous microbenchmarks, Rotom finds cheaper conversion costs using `ApplyRoll`, that improves the HE kernel costs of ct-pt `MatMul` and ct-ct `MatMul`. On $n = 32K$, Rotom finds a better packing that fully utilizes of all ciphertext slots, whereas Fhelipe's packing uses one ciphertext for each attention head, performing around 80× more rotations.

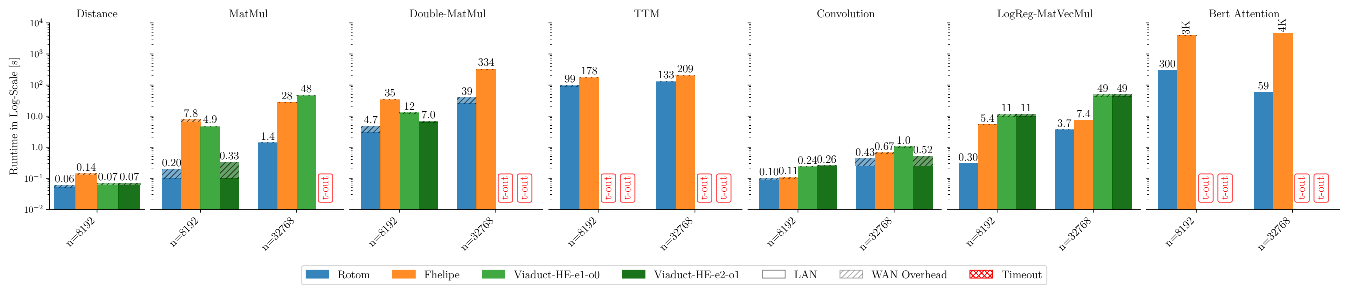


Figure 7: Performance in Log-Scale on OpenFHE for Rotom, Fhelipe, Viaduct-HE-e1-o0, and Viaduct-HE-e2-o1. Diagonal hash bars indicate runtimes in the WAN setting with higher communication overheads. *t-out* indicates the benchmark did not compile within 24hrs.

10.3 Comparison to Hand-tuned Protocols

To determine whether the layout assignments from Rotom can match that of expertly hand-tuned protocols, we compare both the ciphertext-plaintext (ct-pt) and ciphertext-ciphertext (ct-ct) matrix multiplication packings to that of Bolt's and Thor's [41, 43]. Bolt is 2PC transformer model that mixes both HE and MPC; Thor is an FHE transformer model.

When comparing ct-pt matrix multiplication, Rotom matches the exact packing given by Bolt. Comparing to Thor, their ct-pt matrix multiplication algorithm requires an internal rotation on the input ciphertext, incurring an additional multiplication depth. Rotom avoids this initial conversion step and optimizes using BSGS, incurring $4\times$ less rotations and $\approx 1/3$ less multiplications.

For ct-ct matrix multiplication with an input column and row packing, Rotom is finds the exact layout assignment from Bolt. Compared to the proposed packing starting with a diagonal and column packing, Rotom is able to improve the computational cost using its `ApplyRoll` operator. Specifically, Rotom avoids having to replicate the diagonal major packing with a rotate-and-sum routine, but instead chooses to apply a roll to internally rotate the column packing, resulting in less rotations and additions. Rotom also saves on half of the ct-pt multiplication costs by deriving the internal rotation through subtraction rather than from two masks. Compared to Thor, Rotom finds a similar implementation. In fact, their layout conversion into the diagonal packing can be implemented exactly with the `ApplyRoll` operator.

11 Related work

Early iterations of compilers for automatically packing HE ciphertexts [1, 8, 12, 18, 50] used simpler reasoning heuristics and limited layout options. For example, some works restricted to only row-major or column-major layouts in order to scale to larger end-to-end machine learning benchmarks. Using a limited set of layouts makes layout assignment easier to reason about since the compiler can force each tensor operation to begin and end with a specific layout, allowing composability between tensor operations. Unfortunately, limiting layouts also induces more expensive layout conversions that can greatly degrade overall performance.

Fhelipe is a recent work that addresses these limitations by having a flexible layout abstraction, allows arbitrary and intermixed dimension orderings [34]. However, the main limitation with Fhelipe's search approach and layout abstraction is that it doesn't factor in operation costs nor does it support more complex data layouts, such as diagonalization and tiling. These complex layout permutations can greatly reduce the operational costs of frequent tensor operations, such as matrix multiplication, by reducing rotations and moving the summation dimension to vector dimensions.

Viaduct-HE is another recent work that has a similar layout abstraction as Rotom, and provides additional support for more complex data layouts than Fhelipe [47]. However, this system uses a bottom-up enumeration combined with a breadth-first search to explore an immense search space, resulting in long search times. In particular, vector derivation in Viaduct-he is useful in reducing operations by composing one ciphertext efficiently from an existing ciphertext (using a rotation). However, their process for finding vector derivations is quite expensive, especially since this is done by materializing the vector and comparing indexing sites. Orion [19] is another recent FHE framework for deep learning, with a special focus on convolutional neural networks (CNNs). The framework supports optimized but specialized packing strategies for matrix-vector multiplication and convolution. Compared to Orion, Rotom is able to produce ciphertext packing for a variety of models beyond CNNs. Specifically, Rotom's search can efficient convolutions similar to those used in Orion.

Other works such as Porcupine and Coyote use heavyweight search techniques, i.e., program synthesis and simulated annealing, to solve layout assignment [15, 39]. These approaches have a very expensive search procedure and do not scale past small programs.

12 Conclusion

Rotom addresses a key barrier to practical homomorphic encryption by automating the complex, layout-sensitive optimization process for tensor programs. By introducing the novel `ApplyRoll` operator and combining it with systematic layout exploration and cost-based selection, Rotom delivers efficient, vectorized HE implementations with minimal developer effort. Our results show that Rotom not only scales to large tensor programs but also achieves significant performance improvements—paving the way for more practical and performant privacy-preserving computation.

A Tensor Operators

Tensor Operator	**Description**
Tensor	tensor initialization
+	tensor addition
*	tensor multiplication
-	tensor subtraction
@	matrix multiplication
conv2d	2d convolution
sum	summation along a tensor dimension
product	product along a tensor dimension
transpose	tensor transpose
reshape	tensor reshape
permute	permute the dimension order
index	index into a tensor dimension

B Layout Operators

Layout Operator	**Description**
Tensor operators	-
`ApplyReplication`	add a replicated dimension
`ApplyPermute`	apply an arbitrary conversion
`ApplyRoll`	roll one traversal dimension by another
`ApplyCompact`	compact ciphertexts together
`ApplyRotRoll`	optimize ApplyRoll w/ only rotations
`ApplyBSGSRoll`	optimize ApplyRoll w/ BSGS
`ApplyBSGSMatMul`	optimize ct-pt @ w/ BSGS
`ApplyStrassens`	optimize ct-ct @ w/ Strassen's

References

[1] 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.

[2] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, et al. Openfhe: Open-source fully homomorphic encryption library. In *proceedings of the 10th workshop on encrypted computing & applied homomorphic cryptography*, pages 53–63, 2022.

[3] 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*, 2019.

[4] Youngjin Bae, Jung Hee Cheon, Guillaume Hanrot, Jai Hyun Park, and Damien Stehlé. Plaintext-ciphertext matrix multiplication and fhe bootstrapping: fast and fused. In *Annual International Cryptology Conference*, pages 387–421. Springer, 2024.

[5] 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*, 2019.

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

[7] California Consumer Privacy Act (CCPA) 2018. https://oag.ca.gov/privacy/ccpa, 2018.

[8] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. Armadillo: a compilation chain for privacy preserving applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, 2015.

[9] Huili Chen, Rosario Cammarota, Felipe Valencia, Francesco Regazzoni, and Farinaz Koushanfar. Ahec: End-to-end compiler framework for privacy-preserving machine learning acceleration. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[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] 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*, 2018.

[13] HEIR Contributors. HEIR: Homomorphic Encryption Intermediate Representation, 2023. https://github.com/google/heir.

[14] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T Lee, and Brandon Reagen. Porcupine: A synthesizing compiler for vectorized homomorphic encryption. In *PLDI*, 2021.

[15] 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*, pages 375–389, 2021.

[16] 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*, 2018.

[17] 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.

[18] 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.

[19] Austin Ebel, Karthik Garimella, and Brandon Reagen. Orion: A fully homomorphic encryption compiler for private deep neural network inference. *arXiv preprint arXiv:2311.03470*, 2023.

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

[21] GDPR. Official Journal of the European Union '16.

[22] 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. In *ICML*, 2016.

[23] Shruthi Gorantala, Rob Springer, Sean Purser-Haskell, William Lam, Royce Wilson, Asra Ali, Eric P Astor, Itai Zukerman, Sam Ruth, Christoph Dibak, et al. A general purpose transpiler for fully homomorphic encryption. *arXiv preprint arXiv:2106.07893*, 2021.

[24] Shai Halevi and Victor Shoup. Algorithms in helib. In *Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I 34*, pages 554–571. Springer, 2014.

[25] Kyoohyung Han et al. Helr: Homomorphic logistic regression on encrypted data, 2019.

[26] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. Iron: Private inference on transformers. *Advances in neural information processing systems*, 35:15718–15731, 2022.

[27] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

[28] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. Cheetah: Lean and fast secure {Two-Party} deep neural network inference. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 809–826, 2022.

[29] Steven Huss-Lederman, Elaine M Jacobson, Anna Tsao, Thomas Turnbull, and Jeremy R Johnson. Implementation of strassen's algorithm for matrix multiplication. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, pages 32–es, 1996.

[30] Siddharth Jayashankar, Edward Chen, Tom Tang, Wenting Zheng, and Dimitrios Skarlatos. Cinnamon: A framework for scale-out encrypted ai. In *Proceedings of the 30th ACM International Conference on Architectural*

Support for Programming Languages and Operating Systems, Volume 1, pages 133–150, 2025.

[31] 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. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 4361–4375, 2024.

[32] 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, 2018.

[33] Arjun Kharpal. Tech samsung bans use of a.i. like chatgpt for employees after misuse of the chatbot, 2025.

[34] 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.

[35] 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*, pages 503–518, 2020.

[36] 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.

[37] 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.

[38] Wen-jie Lu, Zhicong Huang, Zhen Gu, Jingyu Li, Jian Liu, Cheng Hong, Kui Ren, Tao Wei, and WenGuang Chen. Bumblebee: Secure two-party inference framework for large transformers. *Cryptology ePrint Archive*, 2023.

[39] 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*, pages 118–133, 2023.

[40] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. 2020.

[41] Jungho Moon, Dongwoo Yoo, Xiaoqian Jiang, and Miran Kim. Thor: Secure transformer inference with homomorphic encryption. *Cryptology ePrint Archive*, 2024.

[42] Christian Vincent Mouchet, Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. Lattigo: A multiparty homomorphic encryption library in go. In *Proceedings of the 8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, pages 64–70, 2020.

[43] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. Bolt: Privacy-preserving, accurate and efficient inference for transformers. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4753–4771. IEEE, 2024.

[44] 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*, 2024.

[45] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. arxiv 2019. *arXiv preprint arXiv:1912.01703*, 10, 1912.

[46] Siladitya Ray. Samsung bans chatgpt among employees after sensitive code leak, 2025.

[47] Rolph Recto and Andrew C Myers. A compiler from array programs to vectorized homomorphic encryption. *arXiv preprint arXiv:2311.06142*, 2023.

[48] Adam Satariano. Chatgpt is banned in italy over privacy concerns, 2025.

[49] Mark Stone. Is copilot safe? microsoft copilot security concerns explained, 2025.

[50] 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.

[51] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. Sok: Fully homomorphic encryption compilers. In *2021*

IEEE Symposium on Security and Privacy (SP), pages 1092–1108. IEEE, 2021.

[52] Jeremy White. How strangers got my email address from chatgpt's model, 2025.

[53] Shenglai Zeng, Jiankun Zhang, Pengfei He, Yue Xing, Yiding Liu, Han Xu, Jie Ren, Shuaiqiang Wang, Dawei Yin, Yi Chang, et al. The good and the bad: Exploring privacy issues in retrieval-augmented generation (rag). *arXiv preprint arXiv:2402.16893*, 2024.

[54] 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. *Cryptology ePrint Archive*, 2024.