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 generates 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 Rotom scalably compiles all tensor workloads in under 5 minutes, reduces rotations in hand-tuned protocols by up to 3×, and achieves up to 80× performance improvement over prior autovectorization 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 [9, 22, 48].

Cryptographic primitives for secure computation are a promising solution. 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 $\texttt{Enc}(f(x))$. 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 [23, 26–28, 32, 40, 41, 43, 44, 54].

The most commonly used HE schemes used today are lattice-based HE schemes such as BFV/BGV/CKKS [7, 12, 21]. One trait that makes these schemes efficient 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 single-instruction, multiple data (SIMD) operations over ciphertexts: element-wise addition/multiplication and cyclic rotations. While 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). Unfortunately, most HE programs cannot be trivially lowered into this vector programming model and require expensive intra-ciphertext data movement operations to achieve high slot utilization.

Different data *layouts*—the order in which plaintext values are packed into ciphertext(s)—can result in wildly different data movement costs and performance. Optimizing layout choices is imperative to performance, but manually doing so is tedious. Cryptography experts have spent substantial effort [27, 32, 41, 43, 54] carefully crafting specialized packing schemes for a variety of applications, primarily for machine learning. These workloads support operations over large tensors suitable for the highly-parallelized, SIMD operations used in HE. However, different applications may require an entirely different ciphertext packing, and naive generalization

can lead to significant overhead. 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× more multiplications and over 2000× more rotations than an optimized scheme.

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 explores numerous possible layouts to find an assignment that leads to the least costly HE program. Prior works [1, 15, 19, 20, 34, 39, 47, 50] proposed a variety of solutions, but they all have one of two drawbacks. As with any search problem, the main challenge is the tussle between search speed (compilation time) and search quality (execution time). Naively brute-forcing a fully optimal solution is infeasible as realistic tensor workloads can have hundreds-of-thousands of layout possibilities to consider. One recent work [47] proposed searching over a wide set of complex layouts to improve search quality, but suffered from long compilation times—over 13 hours for a single matrix multiplication. Limiting the number of layout options [1, 19, 20, 34, 50] can improve search speed, but can also introduce expensive data movement costs (i.e., conversions between layouts) that hinder performance. An effective approach should find a high-performance, optimized layout assignment while maintaining fast search speed.

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

One of the challenges in improving search speed is: given a tensor program, how does Rotom accurately determine the corresponding HE implementation and cost for each tensor operator? Being able to easily evaluate the cost of different HE implementations over a wide range of input/output layouts is a crucial first step to a fast search procedure. Unfortunately, this is a challenging problem because previous work has either constrained layout options to simplify cost modeling [34], which sacrifices search quality, or depended on array materialization to calculate HE costs [47]. Materialization suffers from poor scalability, as performance degrades significantly with tensor input size. Rotom's first technique addresses this problem by establishing the concept of layout alignment, a way to standardize how high-level tensor operators are implemented in HE. Alignment not only makes it easy to generate correct HE implementations from thousands of layout choices, but also provides a "cost oracle" that can be quickly used to evaluate their efficiency. To generate HE implementations for a binary tensor operator, Rotom uses a novel set of rules to determine whether operands are aligned. If not, then additional layout conversions are inserted to align the operand layouts.

Data movement in HE is limited and can be expensive. Unfortunately, such data movement operations are often unavoidable; many real-world applications with complex tensor operator chaining need layout conversions to alter the packing to ensure high ciphertext slot utilization. To address this challenge, Rotom's second key contribution is to introduce new layout conversions that lead to cheap data movement costs. By carefully designing a new lightweight layout conversion operator, `ApplyRoll`, Rotom can perform inexpensive conversions while creating structured HE operation patterns that enable additional optimization opportunities. For example, this structure pattern-matches on the well-known baby-step giant step optimization [32] that reduces the number of HE rotations by a square root factor. In many common scenarios, our new layout conversion operator can provide 3-8× speedup compared to other conversions used in prior works.

Lastly, naively searching over all possible layout assignments would be intractable, as there are potentially hundreds-of-thousands of different layout choices to consider for real-world application workloads. To address this search space explosion challenge, we introduce 4 cost-based, pruning heuristics to reduce the search space without compromising search quality. These heuristics to can reduce the layout choices from hundreds-of-thousand just hundreds, enabling Rotom to compile large tensor programs in seconds.

We implement Rotom in ≈12,000 lines of Python and evaluate Rotom on tensor programs used in realistic workloads. Rotom efficiently compiles most workloads within a few seconds (all in under 5 minutes). Rotom reduces rotations in hand-tuned protocols by up to 3× and outperforms prior HE compilers by up to 80× on our benchmarks. Our open-source implementation is available at https://github.com/cmu-cryptosystems/Rotom, and our artifact is archived at https://zenodo.org/records/17957733.

2 Background

2.1 Preliminary: Homomorphic Encryption

Lattice-based HE schemes like BFV/BGV/CKKS [7, 12, 21] have recently gained interest thanks to their efficiency. A common trait of these schemes is their *vector* programming model, where n plaintext values are in encrypted into a single ciphertext (with n slots). For example, a single plaintext vector $v = [v_1, v_2, \ldots, v_n]$ 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 (i.e., n is often between $4K$ and $64K$).

Data movement is expensive in HE. While addition and ciphertext-plaintext (ct-pt) multiplication are relatively

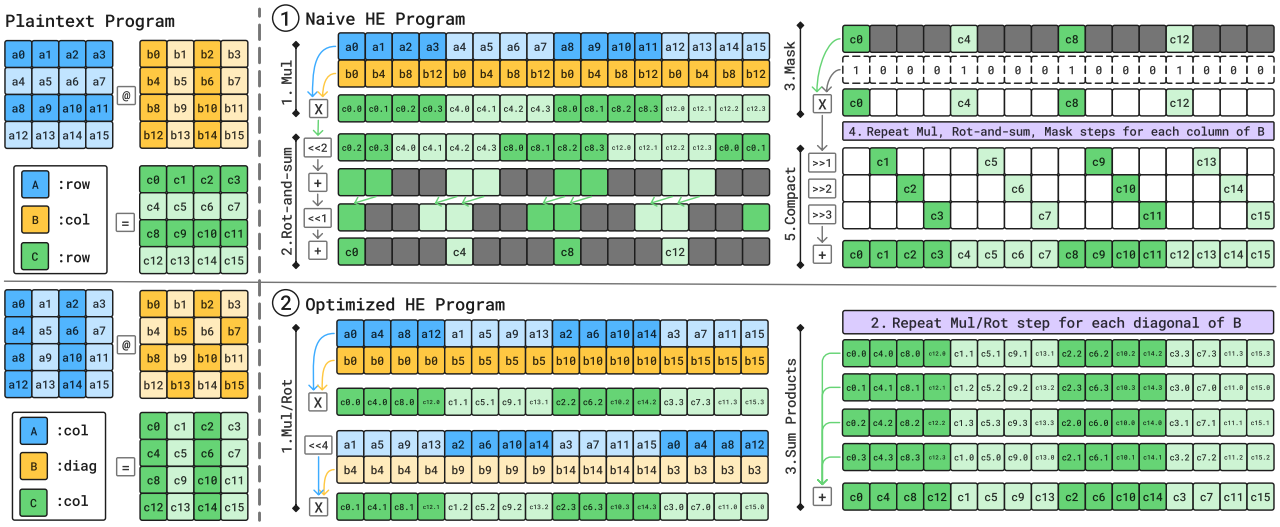


Figure 1: ① shows an HE matrix multiplication with simple layouts and expensive data movement. ② shows a more optimized implementation using a diagonalized layout.

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

2.2 Quality 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 limited data movement instructions. 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 usually requires rewriting the layouts for all downstream operations. Brute-forcing an optimal solution from all possible layout choices is infeasible in practice, since there are numerous layout assignments to explore for real applications. For example, a 64×64 matrix can be tiled in 12! different ways. This happens because each matrix dimension (row and column) can be split into 6 traversals of 2 elements each, creating 12 total segments that can be reordered in any sequence.

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. The plaintext/HE program is illustrated on the left/right side of the dotted line in the figure respectively. In the HE program, partial products of C are labeled as $ci.j$, where i is the output index and j is the partial product index for summation. Part ① shows a costly HE program with simple layouts; and part ② presents an optimized implementation using a diagonal layout.

In ① (top-right), A is row-packed, B is column-packed into 4 separate ciphertexts where each column of B is repeated, and C is row-packed. To save space, we only illustrate 1 of the 4 packed plaintexts representing B. In ①.1-①.3, the program multiplies A's rows with each of B's columns, uses rotate-and-sum to aggregate partial products, and multiplies by 0-1 bitmasks to remove garbage values (marked in gray). In ①.4, these steps repeat for B's remaining columns. Finally in ①.5, the resulting ciphertexts are compacted together (rotated and added). This totals 8 multiplications (one per column, plus masking), 11 rotations (for rotate-and-sum and compaction), and 11 additions.

In ② (bottom-right), A is column-packed, B is diagonally-packed where each diagonal value is repeated 4 times, and C is column-packed. In ②.1 and ②.2, each column of A is rotated to align with B's diagonals, then multiplied together, totaling 3 rotations and 4 multiplications. This diagonal layout naturally aligns partial products across ciphertext slots for direct summation (3 additions), crucially avoiding the rotate-and-sum, masking, and compaction steps required in ①. Using optimized layouts, the total operations are 4 multiplications, 3 rotations, and 3 additions.

As demonstrated in Figure 1, different input layouts result in different HE programs, output layouts, and overall costs. Unfortunately, finding optimal layouts is challenging: the search space of possible layouts is extremely vast, data movement operations in HE are expensive, and modifying

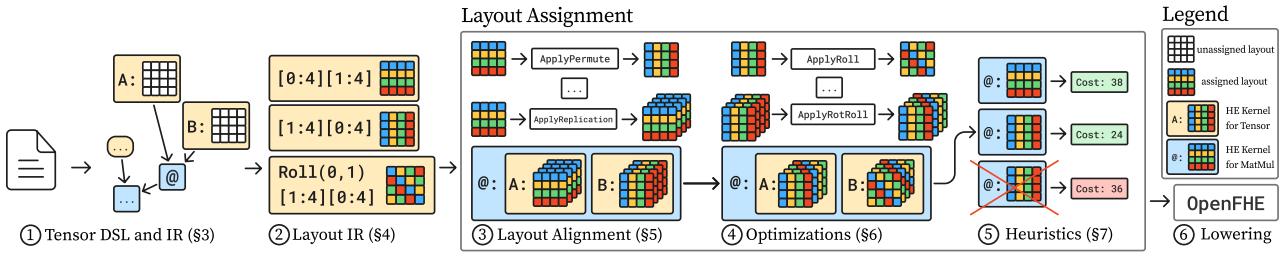


Figure 2: Rotom compilation pipeline. Each component references a section in the paper.

input layouts often necessitates recomputing the entire layout assignment—making exhaustive search intractable. Rotom alleviates these challenges by automatically exploring a wide range of layout options using its layout alignment rules, lightweight conversion operator (`ApplyRoll`), and search space heuristics.

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 executed by an HE backend. In the following sections, we walk through Rotom's compilation pipeline in detail, following Figure 2. Additionally, we refer the reader to Appendix F on how to integrate Rotom as a layout assignment component in existing secure two-party computation and fully homomorphic encryption pipelines.

3.1 Rotom's DSL and IRs

① Tensor DSL and IR (§3) To cut down on programming headaches, Rotom supports a tensor-based domain-specific language (DSL) that is similar to writing programs in PyTorch [45]. This DSL exposes common tensor operators used in machine learning, such as 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 (`secret=False`) and encrypted (`secret=True`) input tensors. `MatMul` and `Conv2D` describe binary tensor operators on `Tensor` operands and return a `Tensor`. Rotom parses the DSL into its tensor intermediate representation (IR), a directed acyclic graph (DAG) where each node is a one-to-one match with a tensor DSL operator. Appendix A details the tensor operators within this IR.

② Layout IR (§4) Given a tensor IR, Rotom performs layout assignment by defining the layouts used for each tensor operator, eventually lowering the tensor IR into Rotom's layout IR. The layout IR is composed of layout operators, each maintaining a high-level layout representation (see Section 4) of how tensor elements are packed into ciphertexts(s). There are two classes of layout operators: computation and

```
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.

```
At = Tensor("A",layout=[0:4][1:4])
Bt = Tensor("B",layout=[1:4];[4][0:4])
A = ApplyReplicate(At,layout=[4];[0:4][1:4])
T = MatMul(A,Bt,layout=[1:4];[0:4][G:4])
C = ApplyCompact(T,layout=[0:4][1:4])
```

Listing 2: An HE Kernel using layout operators.

conversion operators. Computation operators implement tensor computations (e.g., `MatMul`) and are a one-to-one match with tensor operators. Conversion operators are used to transform one layout to another. Rotom's layout IR supports 4 conversion operators—`ApplyPermute`, `ApplyReplicate`, `ApplyCompact`, and `ApplyRoll`. `ApplyRoll` is a new conversion operator introduced by Rotom and is the main workhorse behind most of Rotom's layout optimizations (see Section 6). Appendix B details the layout operators in this IR.

When lowering a tensor operator into HE, Rotom may need to add conversion operators to align operand layouts for the tensor computation. Rotom lowers each tensor operator into an *HE kernel*: a DAG of layout operators that consists of zero or more conversion operators followed by a single computation operator. Listing 2 showcases the HE kernel for the plaintext matrix multiplication example in Figure 1. ①.

Algorithm 1: Layout Assignment

Input: Tensor IR: $G = (V, E)$
Output: Layout assignment: $V \rightarrow$ HE kernel

1 Initialize $C[v] \leftarrow \{\emptyset\}$ for all $v \in V$;
2 **foreach** *input operator* $v \in V$ **do**
3 | $C[v] \leftarrow \text{SEEDLAYOUT}(v)$;
4 **end**
5 **foreach** *non-input operator* $v \in V$ *(topological order)* **do**
6 | $C[v] \leftarrow \text{GENCANDIDATES}(v, C)$;
7 | $C[v] \leftarrow \text{APPLYOPTIMIZATIONS}(C[v])$;
8 | $C[v] \leftarrow \text{PRUNESEARCH}(C[v])$;
9 **end**
10 **return** $\text{SELECTASSIGNMENT}(C)$;

Algorithm 2: GenCandidates

Input: Operator v, candidate HE kernels C
Output: Candidate kernels S

1 $S \leftarrow \emptyset$;
2 **foreach** $K \in \prod_{u \in v.children} C[u]$ **do**
3 | **if** $\neg \text{LAYOUTSALIGNED}(v, K.layouts)$ **then**
4 | | $K \leftarrow \text{APPLYCONVERSION}(v, K)$;
5 | **end**
6 | $S \leftarrow S \cup \{\text{APPLYCOMPUTATION}(v, K)\}$;
7 **end**
8 **return** S;

3.2 How Rotom Works

To find a layout assignment, Rotom efficiently searches over a wide range of layout choices using top-down enumeration aided by search space heuristics. In this process, Rotom maps each tensor operator to an HE kernel such that the overall assignment has the least overall cost.

Algorithm 1 presents our layout assignment algorithm. To begin, all input `Tensor` operators are initially seeded with a compact row/column-major layout (line 2-4). Next, Rotom performs a topological traversal over its tensor IR and generates candidate HE kernels—each with different input/output layouts and costs—for each tensor operator (line 5). Rotom builds the candidate HE kernels *iteratively* using the output layouts from the previous tensor operator as input layouts to the next tensor operator (line 6). 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. Rotom uses its layout alignment rules to construct each HE kernel deterministically, allowing Rotom to easily derive the costs of each kernel. Then, Rotom applies its optimization and search space heuristics to maintain only the Pareto frontier of HE kernels (lines 7-8). The following sections detail how Rotom constructs, optimizes, and prunes HE kernels before lowering to OpenFHE [2].

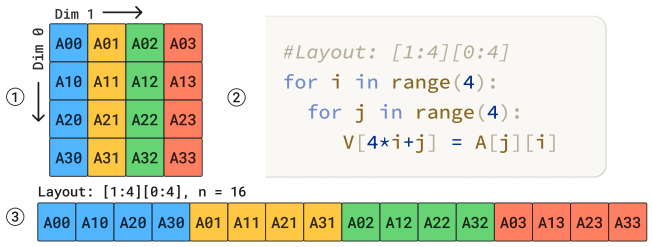
③ Layout Alignment (§5) To construct an HE kernel for a binary tensor operator, Rotom checks whether the input operands' layouts (inherited from the output layouts of the preceding HE kernels) are *aligned* using its alignment rules (see Section 5). This alignment check is shown in Algorithm 2 line 3. Alignment allows Rotom to deterministically generate a corresponding HE kernel, as well as reason about the kernel's data movement costs directly from its high-level layout representation. Rotom uses layout conversion operators to align layouts (line 4). These operators change how tensors are packed into ciphertexts, oftentimes by replicating, swapping, or masking out slot values. `ApplyReplicate` replicates values within a ciphertext by rotating and adding the *same* ciphertext together; `ApplyCompact` performs compaction by rotating and adding *different* ciphertexts together. `ApplyPermute` enables arbitrary layout tiling and dimension reordering; misaligned values are first masked into individual ciphertexts, then recompacted together. Lastly, `ApplyRoll` is a new conversion operator that applies a roll to a layout (see Section 4.3), a transformation that performs circular shifts along a given dimension. Once layouts are aligned, Rotom applies the computation operator to generate the output kernel (line 6).

④ HE Kernel Optimizations (§6) To improve the performance of an HE kernel, Rotom introduces 5 new peephole optimizations (see Section 6). These peephole optimizations include re-writing `ApplyPermute` to `ApplyRoll` (a cheaper layout conversion operator) and using baby-step giant-step to reduce the number of rotations for common layout IR patterns.

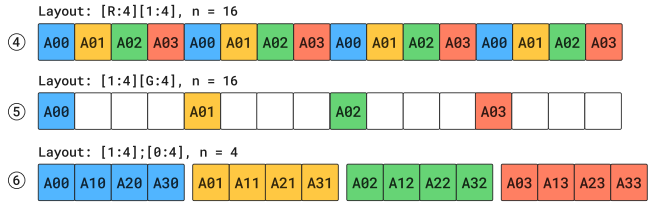
⑤ Search Space Heuristics (§7) Rotom uses a top-down enumeration search to find a layout assignment. To prevent search-space explosion, Rotom employs 4 symmetry-breaking, cost-based heuristics (see Section 7) that either restrict when layout conversions are applied or prune costly HE kernels that are not along the Pareto frontier of existing solutions.

Rotom does not guarantee finding the optimal solution with the space of all possible layouts, as enumerating through this space is practically infeasible. However, its approach finds a practical solution within its restricted search space. One class of heuristics removes candidate HE kernels that are *strictly more expensive* than an existing solution, thereby maintaining only the Pareto frontier of candidate HE kernels. The other class of heuristics reduces the search space by lazily applying layout conversions when necessary.

⑥ Lowering to OpenFHE After finding a layout assignment, a map of tensor operators to HE kernels, Rotom lowers the HE kernels into an HE program. This program is composed of a low-level IR that matches the HE operations exposed by backend libraries. The IR acts as a common abstraction to connect Rotom to OpenFHE [2] and other frameworks (e.g., HEIR [3]).



(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.



(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.

4 Preliminary: Layout Representation

This section describes the layout representation (or "layout" for short) that Rotom uses in its layout IR to define how tensors are packed into ciphertexts. Like prior systems [1, 19, 34, 47], Rotom uses a high-level layout representation to easily reason about different layout assignments and conversions between layouts. Rotom adapts the layout representation introduced by Viaduct-HE [47] as both systems use rolls (further detailed in Section 4.3). Two of Rotom's contributions are a new lightweight conversion operator, `ApplyRoll`, and alignment rules that lets Rotom to easily construct HE kernels with rolled layouts.

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 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 dictates 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* that represent the order of tensor elements along a tensor dimension. For example, to iterate through the first column of A, the traversal dimension should translate to the tensor indices: `{(0,0),(1,0),(2,0),(3,0)}`.

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). 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[1]. Although A is a two-dimensional tensor, since the 1st dimension (`dim=1`) is not specified in our layout, the default value for the traversal along `dim=1` is 0. Therefore, the layout `[0:4]` corresponds to the tensor indices: `{(0,0),(1,0),(2,0),(3,0)}`.

Traversing over a multi-dimensional tensor can be represented using a list of traversal dimensions, which conceptually corresponds to a nested loop iteration over this tensor. Let's consider packing A column-wise into a single $n = 16$ ciphertext. To do so, we write `[1:4][0:4]`, where `[1:4]` is the 0th traversal dimension and `[0:4]` is the 1st traversal dimension in the list. 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 uses 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 on 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. A gap 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. Gap dimensions are similar repeated dimensions, but with all repeated values set to zero except for the first value.

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 sep-

[1]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.

arate traversal dimensions into *vector dimensions* and *slot dimensions*. Vector dimensions define *which* ciphertext a tensor index is mapped to, and slot dimensions define the slot position *within* that ciphertext. Segmentation is done using the "`;`" operator. Traversal dimensions before "`;`" constitute vector dimensions, and traversal dimensions after "`;`" constitute slot dimensions. If a layout representation does *not* 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.

4.3 Rolls

A roll is a transformation of a layout that cyclically shifts the slot elements along a specified traversal dimension. For example, roll can represent cyclically shifting each column of a two-dimensional matrix 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. Rolls were originally introduced by Viaduct-HE [47] only as a means to pack input ciphertexts into more complex layouts. Rotom generalizes rolls by introducing `ApplyRoll`, an operator that can apply a roll to any layout. Rolled layouts can significantly reduce data movement costs and improve HE program performance.

A roll operation **Roll(i,j)** shifts the tensor indices of one traversal dimension by adding the indices of another. Formally, given layout L with traversal dimensions $[td_0, ..., td_n]$, $Roll(i,j)$ modifies the tensor indices of td_i as $ind(td_i) = (ind(td_i) + ind(td_j))\ \%\ td_i.extent$, where $ind(td)$ returns the tensor indices of td. Rotom uses the `ApplyRoll` operator to convert an HE kernel from layout L_1 to layout $L_2 = Roll(i,j)(L_1)$.

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`$=1$ represents `[0:4]`, and `j`$=0$ represents `[1:4]`. `Roll(1,0)` performs a modular addition between the tensor indices of `[0:4]` and `[1:4]`, replaces `[0:4]` with these indices, and leaves `[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,...}`.

5 Layout Alignment

The previous section presents the high-level layout representation that Rotom uses to explore a wide range of layouts. In this section, we discuss how Rotom uses layout alignment rules (or "alignment" for short) to correctly and deterministically generate an HE kernel from a tensor operator. Alignment ensures that the traversal patterns of input tensors—as encoded in their layouts—are compatible with the iteration structure of the target tensor operation. Rotom uses alignment directly on the layout representation of layout operators, checks whether

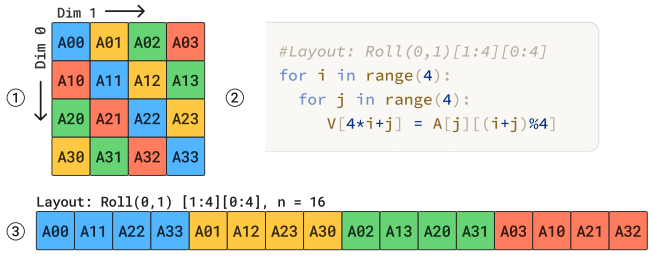


Figure 4: ① presents the input matrix A. ② describes the diagonal-major layout, using a roll. ③ shows how A is packed diagonally into a single ciphertext with $n = 16$.

the layouts satisfy the specific constraints for a tensor operator, and determines if layout conversions are needed.

Prior works with constrained layouts treat alignment as an implementation detail rather than an explicit concern. This is achieved through either rigid HE kernel implementations with fixed input/output layout [1, 19, 50] or by requiring users to manually specify alignment within the frontend interface [34]. However, these alignment strategies prove inadequate for managing alignment with rolled layouts. Our contributions are as follows. First, we establish the alignment rules used by prior compilers, creating a simple and intuitive checks over our layout representation that makes alignment easy to reason about and apply systematically. Second, we develop new alignment rules to handle rolled layouts created by our novel `ApplyRoll` operator. These rules let Rotom construct HE kernels directly from rolled layouts without materializing intermediate circuit representations, significantly improving search speeds over prior work [47]. Appendix C provides examples of Rotom's layout alignment rules.

5.1 How Rotom Uses Alignment

When Rotom constructs an HE kernel for a binary tensor operator, Rotom needs to *align* the operand layouts to achieve high slot utilization and good performance. Rotom uses a set of rules to help determine which traversal dimensions and rolls in a layout are misaligned, and what layout conversions are necessary to realign the layouts. Alignment standardizes how HE kernels are constructed by separating layout conversion operators from the layout computation operators. These operators all have a deterministic lowering based on the operand and target layout, allowing Rotom to easily compute the cost of conversions and HE kernels from alignment decisions.

5.2 Traversal Dimension Alignment

Recall that Rotom uses traversal dimensions in a layout to represent how to iterate over a tensor. Intuitively, the alignment rules try to match the traversals in a layout to the traversals of an equivalent nested loop structure for a tensor operator. We first define traversal dimension alignment *without considering*

rolls, which establishes the baseline alignment requirements based solely on a traversal dimension's dimension, extent, and stride. For a binary tensor operator Op with input layouts L_0 and L_1, let $td_i^{(L_j)}$ denote the i-th traversal dimension in layout L_j. We define a dimension alignment map $\mathcal{M}_{Op}$ that specifies how tensor dimensions from L_0 correspond to tensor dimensions in L_1 for operation Op. For example, element-wise addition requires $\mathcal{M}_{\texttt{Add}} = \{0{:}0, 1{:}1\}$ (identical dimension alignment), whereas matrix multiplication requires $\mathcal{M}_{\texttt{MatMul}} = \{0{:}\text{R}, 1{:}0, \text{R}{:}1\}$ (inner dimension alignment with outer dimensions repeated).

Definition 5.1 (Traversal Dimension Alignment)**.** Two traversal dimensions $td_i^{(L_0)}$ and $td_i^{(L_1)}$ are aligned for operation Op, if and only if:

1. $\mathcal{M}_{Op}[td_i^{(L_0)}.\text{dim}] = td_i^{(L_1)}.\text{dim}$
2. $td_i^{(L_0)}.\text{extent} = td_i^{(L_1)}.\text{extent}$
3. $td_i^{(L_0)}.\text{stride} = td_i^{(L_1)}.\text{stride}$

Thus, two operand layouts are aligned for operation Op if all corresponding traversal dimensions are aligned. Notably, alignment does not fix the order of the traversal dimensions, but rather their relative positioning across both layouts.

5.3 Roll Alignment

As Rotom is the first compiler to support the `ApplyRoll` operator, Rotom extends its alignment rules to handle rolled layouts. Recall that rolls shift tensor elements within a ciphertext, modifying the tensor indices accessed by a layout (Section 4.3). If two operand layouts have aligned traversal dimensions (see Definition 5.1) but different rolls, the rolls may break alignment by causing traversal dimensions to iterate over misaligned tensor indices.

For a layout L with traversal dimension td_i rolled by td_j, we denote the roll as $roll(i,j)^{(L)}$.

Definition 5.2 (Roll Alignment)**.** Two traversal dimensions $td_i^{(L_0)}$ and $td_i^{(L_1)}$ are aligned for operation Op in the presence of rolls if and only if:

1. $td_i^{(L_0)}$ and $td_i^{(L_1)}$ satisfy traversal dimension alignment (see Definition 5.1)
2. For any roll $roll(i,j)$ applied to either layout:
   - If both $td_i^{(L_0)}$ and $td_i^{(L_1)}$ are non-repeated dimensions, then $roll(i,j)$ must be applied to both layouts
   - If either $td_i^{(L_0)}$ or $td_i^{(L_1)}$ is a repeated dimension, no matching roll is required

This definition captures two key properties. Non-repeated dimensions must have matching rolls because they iterate over distinct tensor indices that would be misaligned by different shifts. In contrast, repeated dimensions tolerate mismatched rolls because all indices map to the same replicated value.

6 Improving Layout Search Quality via Rolls

Once a tensor operator's input operands are aligned, Rotom can easily compute the cost of an HE kernel by summing layout conversion and layout computation costs. In this section, we explain how Rotom uses our new `ApplyRoll` operator to find optimizations that improve both fronts.

One common way to achieve alignment is to swap the positions of two traversal dimensions. Rotom's first insight is the design of a new `ApplyRoll` operator that can inexpensively perform this swap to align layouts (Section 6.1). Our second insight is that `ApplyRoll` lowers to a structured set of HE operators (rotations with a fixed step-size and masking with evenly spaced slots), and such structure is amenable to more optimization opportunities that can further minimize data movement costs (Section 6.2). We implement our insights as peephole optimizations in Rotom's compiler pipeline.

6.1 `ApplyRoll` Swaps Traversal Dimensions

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 both the traversal dimensions and roll does not change the resulting tensor indices. This property only holds because a repeated dimension contributes the same constant offset to every tensor index during the modular addition; otherwise, swapping two non-repeated 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 proactively use `ApplyRoll` when it can swap a traversal dimension with a repeated dimension.

While both `ApplyPermute` and `ApplyRoll` can swap dimensions, 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 cipher-

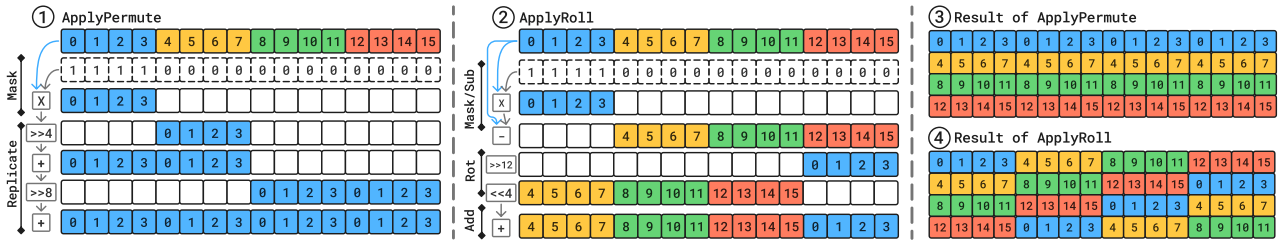


Figure 5: Comparison of `ApplyPermute` and `ApplyRoll` to swap two traversal dimensions.

text. 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, 1 sub/add, and 2 rotations 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 *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 if we can ensure all tensor indices are shifted by the same offset, 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, where the latter consists of only rotations and no extra masking.

A careful reader might observe that `ApplyRoll` always generates rolled output layouts, whereas `ApplyPermute` generates non-rolled layouts (as seen in Figure 5). This could potentially be sub-optimal if rolled layouts lead to very expensive layout computation costs for some subsequent tensor operator. However, we observe in most linear algebra operations, rolled layouts achieves comparable or better layout computation costs than non-rolled layouts. Since both conversions maintain the same relative traversal dimension order for alignment, `ApplyRoll` provides a cheaper conversion overhead while preserving equivalent computation performance.

6.2 Reducing Rotations with `ApplyRoll`

With the introduction of `ApplyRoll`, Rotom identifies opportunities to apply the well-known baby-step giant-step (BSGS) optimization [25] to significantly reduce the number of rotations. The insight behind this optimization stems from the regular structure of HE operators used to implement `ApplyRoll`, which lowers to rotations with a fixed step-size and evenly-spaced masking. Rotom leverages this structure to create two new peephole optimizations that re-write `ApplyRoll` into `ApplyBSGSRoll` and `ApplyBSGSMatMul`. By using these cheaper conversion operations, Rotom finds an efficient way to automatically apply Strassen's algorithm in HE.

Baby-step Giant-step Optimizations BSGS is an optimization that reduces the overall number of rotations from $O(n)$ to $O(\sqrt{n})$. Rotom uses BSGS when it pattern-matches on two HE implementation 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, BSGS 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`. This optimization occurs when two slot dimensions need to be swapped (e.g., `Transpose`). `ApplyRoll` is lowered to HE operators by first masking the original ciphertext into partitions, rotating each partition (with an increasing step-size), and compacting the partitions together. This operation exhibits the rotate, multiply, and accumulate structure that BSGS is designed to optimize. Also, when `ApplyRotRoll` is used with `MatMul`, Rotom will rewrite both operators to `ApplyBSGSMatMul`. The computation also matches the BSGS pattern; `ApplyRotRoll` introduces rotations and `MatMul` follows the multiply-and-accumulate structure.

Moving summation dimensions to vector dimensions Many tensor operators (e.g., `MatMul`) include a summation of values along a tensor dimension as a sub-operation. We call this traversal dimension the *summation dimension*, which can be either a vector or slot dimension. The observation underlying our optimization is that summation along vector dimensions is more performant as it can be done by directly adding ciphertexts together. In contrast, summation along slot dimensions uses a rotate-and-sum routine, which needs additional rotations and leaves behind garbage values that require masking.

To optimize the computation cost of HE kernels, Rotom will actively try to move summation dimension to vectors dimensions using its lightweight `ApplyRoll` operator. Prior works did not use this optimization because their layout conversions (`ApplyPermute`) were too expensive, and the conversion costs outweighed any performance benefits. Appendix D details the cost trade off of `ApplyPermute` and `ApplyRoll` when applied to `MatMul`.

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) `MatMul` when an input tensor is tiled into 4 square tilings that span 4 ciphertexts. Rather than performing 8 `MatMul` 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 improves computation costs by reducing the number of ciphertext multiplications by half. In conjunction with `ApplyRoll` and its optimizations, Rotom improves over baseline ct-ct `MatMul` by $\approx 6\times$, whereas Strassen's alone only improves performance by $1.27\times$ (detailed in Table 3). The benefit of Strassen's is not very apparent with `ApplyPermute`, as the cost is dominated by conversions.

7 Cost-based Symmetry-breaking Heuristics

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 with search space heuristics to find an efficient layout assignment.

Previous approaches either limited the layout search space that lead to suboptimal assignments [34], or exhaustively explored a wide variety of complex layouts that resulted in long compile times (over 24 hours) [47]. 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. In addition, Appendix E addresses Rotom's design limitations with regards to its layout search space.

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 only removes HE kernels that are strictly worse than 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, arbitrary rolls can create an exponentially large search space, since each traversal dimension can be rolled by any other dimension, making search intractable. While this selective application could theoretically reduce search optimality, we found that our benchmarks do not benefit from additional roll operations beyond the two cases listed.

Tiling and Compaction Heuristic. To further constrain layout diversity, Rotom applies a heuristic when encountering `ApplyCompaction`, which compacts multiple ciphertexts with gap slots together. Instead of compacting vector dimensions arbitrarily, Rotom tries to group vector dimensions with slot dimensions that share the same tensor dimension (i.e., the same `dim` in a traversal dimension). This is because many downstream tensor operations, such as matrix multiplication, perform summations along a tensor dimension. Packing same dimensions together will result in less internal fragmentation (uneven runs of gap slots) among traversal dimensions. While this heuristic does not guarantee optimal compaction choices, it often produces efficient layouts in practice.

Canonical Vector Dimension Ordering. A layout with with multiple vector dimensions could have potentially multiple equivalent representations. Rotom eliminates redundant layout variants that are semantically equivalent by canonicalizing the order of vector dimensions. 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. This heuristic reduces the number of layout configurations considered during search and has no affect on search optimality.

8 Evaluation

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 automatically chosen by OpenFHE to ensure 128 bits of security. Rotom relies on the noise management passes available in OpenFHE to automatically insert ciphertext maintenance operations (rescales, relinearization, and modswitches).

To evaluate Rotom, we run benchmarks to determine the *scalability* of Rotom's compilation pass and *search quality* of the layout assignments. Our macro-benchmarks consist of tensor benchmarks from existing HE compiler literature and components of state-of-the-art secure inference protocols from a set of 5 papers [26, 34, 35, 43, 47]. These benchmarks collectively cover all major tensor operator categories (e.g., `MatMul` and `Convolution`) that are used frequently in real-world HE applications. 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 [2], with

the same cryptographic parameters.

Evaluation Setup We compiled and executed all 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 of 5 runs with the relative standard error $< 5\%$.

8.1 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	126	0.23	5.13×
`ApplyRotRoll`	0	0	63	0.14	8.43×
`MatMul-ApplyPermute`	447	128	447	1.893	-
`MatMul-ApplyRoll`	189	190	126	0.634	2.99×
`MatMul-ApplyRotRoll`	63	64	63	0.215	8.80×
`ApplyBSGSMatMul`	63	64	14	0.063	30.05×

Table 1: Comparison of `ApplyPermute`, `ApplyRoll`, and roll optimizations on swapping two dimensions and `MatMul`.

Our first micro-benchmark in Table 1 compares the cost of `ApplyPermute`, `ApplyRoll`, and `ApplyRotRoll` when swapping two dimension and the performance effects it has on `MatMul`. The input is initialized as a 64×64 tensor, replicated in a column-major layout: `[R:64];[1:64][0:64]`. The goal is to swap the leftmost slot dimension with a vector dimension, e.g., swapping the positions of `[R:64]` and `[1:64]`. Our results show that using rolls greatly reduces layout conversion costs, improving conversion performance by 5.13× and 8.43× respectively. These improvements mainly stem from how roll conversions reduce the total number of HE rotations by avoiding the rotate-and-sum routine used in `ApplyPermute`. The bottom half of the table shows the effects of these conversions on `MatMul`. Note how the computation costs for each benchmark are identical (an additional 63 Add and 64 Mul operation). By using `ApplyRoll` and its optimizations, Rotom achieves huge performance benefits over the baseline—26.08× using BSGS to further reduce rotations by a square root factor.

Table 2 compares swapping two slot dimensions within a single ciphertext. Again, the micro-benchmark is initialized with a column-major 64×64 tensor: `[1:64][0:64]`. The objective is to swap `[0:64]` into the position of `[1:64]`. For `ApplyPermute`, the resulting layout is `[0:64][R:64]`, whereas for the roll layout operators, the resulting layout is `Roll(1,0)[1:64][0:64]`. 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.

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: Comparison of swapping two slot dimensions.

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

Table 3: Performance comparison of Strassen's Algorithm.

Our final micro-benchmark 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 $n = 4096$. `Matmul-ApplyPermute` is comparable to implementations found by both Fhelipe and Viaduct-HE.

Table 3 shows how expensive `MatMul-ApplyPermute` is, using thousands of rotations for both computation and conversion costs. Using `Strassens-ApplyPermute`, Rotom can reduce the number of ciphertext multiplications in half, and the number of rotations by $\approx 1,000$, 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 the baseline.

8.2 Application-level benchmarks

Rotom evaluates on a suite of 6 tensor workloads and compares the layout assignment to both Fhelipe [34] and Viaduct-HE [47]. Fhelipe is targeted at FHE workloads, and thus they also introduce bootstrap placement. Contrary, Rotom's goal is to build a compiler framework that automatically vectorizes tensor workloads. Rotom compares to Fhelipe's vectorization pass, but our contributions are complementary to their bootstrap placement strategy. Our benchmarks are as follows:

1. **MatMul** [47] performs a ct-pt `MatMul` between A and B. For $n = 8K$, A: 128×64, B: 64×128; for $n = 32K$, all shapes are doubled.

2. **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.

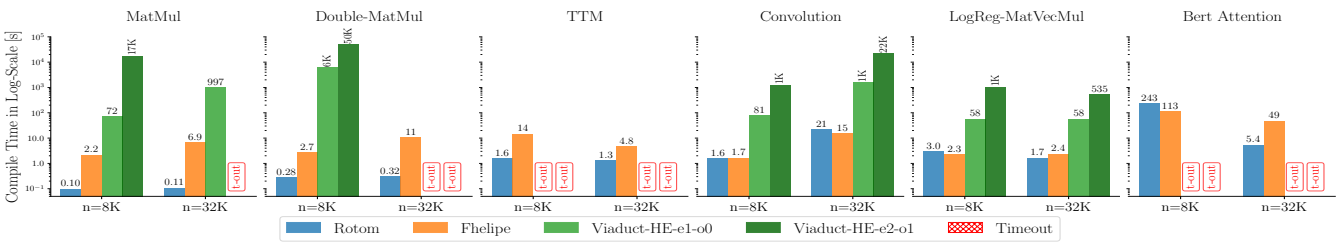


Figure 6: Compile time in Log-Scale. *t-out* indicates the benchmark did not compile within 24hrs.

3. **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.

4. **Convolution** [35] performs a convolution over an 8-channel input *image* with a filter=3 and a stride=1. For $n = 8K$, *image*: 32×32; for $n = 32K$, *image*: 64×64.

5. **LogReg-MatVecMul** [26] This workload takes one iteration and performs two consecutive matrix-vector multiplications with a size of 1024×197 features.

6. **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. To our knowledge, we are the first system to automatically find a layout assignment for a transformer workload.

8.2.1 Rotom Achieves Fast Search Speed

Rotom compiles most benchmarks in seconds, and all benchmarks <5 minutes. 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.

Rotom heuristics help control the search space of layout assignments explored. For example, Rotom explores only 7,623 layout assignments on BERT Attention. Without these heuristics, the number of layout assignments quickly grows to over 50,000 after a few `MatMul` operators. Although Fhelipe achieves fast compilation, its lowering process generates complicated operation structures to preserve high-level layout and metadata information. In contrast, Rotom improves the compilation time by employing straightforward modifications to its layout operators. It's important to note that Fhelipe also handles bootstrap placement which adds to the complexity of their data structures; these noise management passes were turned off during the compilation time comparison.

Viaduct-HE's layout assignment pass requires array materialization—explicitly constructing the complete array of slot indices for each candidate layout—to derive layout conversions from existing ciphertexts. Viaduct-HE employs a bottom-up enumeration strategy that systematically generates candidate layouts by exploring how vectors can be derived from one another. For each candidate generated during this search, the system must materialize the full vector and perform element-wise index comparisons across thousand-element vectors to determine which layout derivations are valid. This expensive per-candidate check, combined with an exponentially growing search space, results in long compilation times. Viaduct-HE times out on larger benchmarks, such as BERT, where `MatMul` is performed with large tensors. By using its alignment rules directly on the high-level layout representation, Rotom can find complex layout conversions using `ApplyRoll` without materialization.

8.2.2 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). In the following comparisons, we compare Rotom to the faster run from both Viaduct-HE compilation settings.

MatMul: On $n = 8K$, Rotom outperforms both Fhelipe and Viaduct-HE by 39× and 1.65× respectively using `ApplyRotRoll` in conjunction with `ApplyBSGSMatMul`. Compared to Fhelipe, both Rotom and Viaduct-HE find a diagonalized input packing that moves summation dimensions to the vector dimensions, avoiding the costly rotate-and-sum routine. Over Viaduct-HE, 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×. This larger performance gap occurs because both baselines resort to performing summations along slots.

Double-MatMul: On $n = 8K$, Rotom outperforms both Fhelipe and Viaduct-HE by 7.45× and 1.49× respectively. On the LAN network, both Rotom and Viaduct-HE identify that it is cheaper to send multiple ciphertexts than paying the cost of a layout conversion, providing a significant advantage over Fhelipe. On $n = 32K$, Rotom outperforms Fhelipe by 8.6×

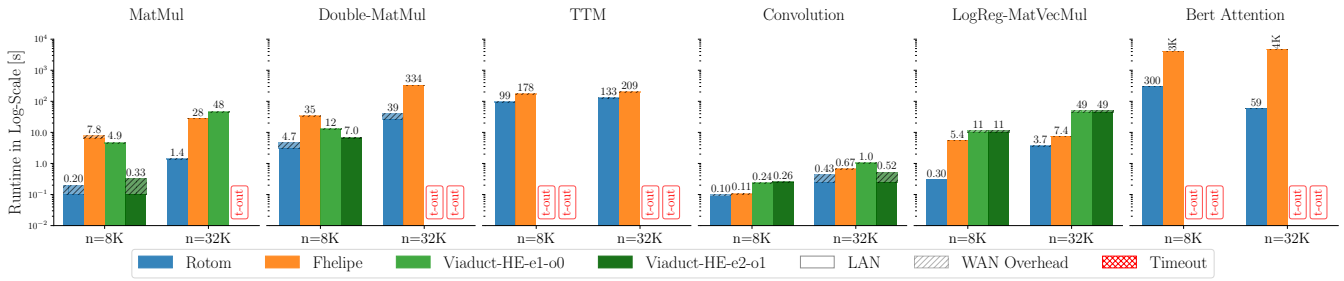


Figure 7: Performance in Log-Scale on OpenFHE for Rotom, Fhelipe, Viaduct-HE. Diagonal hash bars indicate runtimes in the WAN setting with higher communication overheads. *t-out* indicates the benchmark did not compile within 24hrs.

using `ApplyRoll` to cheaply swap the summation dimensions to the vector dimensions.

Convolution: For $n = 8K$, Rotom uses `ApplyRoll` to find summations along vector dimensions, and Fhelipe uses its `shift` operation, which rotates ciphertexts, to find a similar implementation. Viaduct-HE fails to find a plaintext hoisting optimization that combines the masks to the plaintext weight matrix and uses $2\times$ more multiplications. On $n = 32K$, both Rotom and Viaduct-HE have similar packings and HE kernels that rely on sending multiple ciphertexts.

TTM/LogReg-MatMul: In both of these benchmarks, Rotom outperforms both baselines by using its optimized `ApplyRotRoll` and `ApplyBSGSMatMul` operators to greatly reduce the number of rotations.

BERT: For $n = 8K$, Rotom outperforms Fhelipe by $10\times$. Rotom finds cheaper conversion costs using `ApplyRoll`, improving 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 $80\times$ more rotations.

Rotom outperforms both Fhelipe and Viaduct-HE through its expressive layout conversion operators and peephole optimizations that identify low-cost HE kernels. Compared to Fhelipe, Rotom supports complex layouts like diagonalization and rolls that enable BSGS optimizations. Since Fhelipe's layout representation lacks these primitives, it cannot leverage these optimizations and achieves lower efficiency. Viaduct-HE's layout assignment system also lacks BSGS primitives, resulting in O(n) rotations even with diagonalized layouts. These advantages are evident across all benchmarks, where Rotom typically reduces rotations to achieve an order-of-magnitude speedup over both Fhelipe and Viaduct-HE.

8.3 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) `MatMul` packings to that of Bolt's and Thor's [41, 43]. Bolt is 2PC transformer model that mixes both HE and secret sharing; Thor is an FHE transformer model.

Setting	System	# of ct	Add	Rot	pMul	cMul
ct-pt `MatMul`	Bolt	12	9204	1092	9216	-
[128,768]@[768,768]	Thor	96	12272	320	12384	-
layout: col-diag	Rotom	12	9204	1092	9216	-
ct-ct `MatMul`	Bolt	2	63	1,022	896	64
[128,64]@[64,128]	Thor	8	528	544	1536	544
layout: col-row	Rotom	2	379	348	380	64

Table 4: Comparison of `MatMul` with hand-tuned protocols. pMul is masking, cMul is ct-ct multiplication, and $n = 8192$.

When comparing ct-pt `MatMul`, Rotom matches the exact packing given by Bolt. Comparing to Thor, their ct-pt matrix multiplication algorithm significantly reduces the number of rotations at the cost of increased computation and communication costs by sending $8\times$ more ciphertexts. Rotom is not able to generate this packing from Thor, however it does provide a more compact packing which could benefit future work in incorporating bootstrapping passes.

For ct-ct matrix multiplication with an input column and row packing, Rotom improves the computational cost using its `ApplyRoll` operator compared to both Bolt and Thor by reducing both the number of rotations and the number of pMul operations. The increased number of additions has a negligible effect on runtime since additions are cheap. Specifically, when applying a roll along slot dimensions, Rotom's `ApplyRoll` can be optimized to `ApplyBSGSRoll`, using BSGS to reduce the number of rotations by $\approx 3\times$. Additionally, the conversion operator uses subtraction to get the partial rotations, further reducing the number of masks.

9 Related work

Vectorization-focused compilers. Early compilers for HE [1, 10, 14, 50] primarily focused on vectorizing loop-nest-style code using simpler reasoning heuristics or provided domain-specific abstractions with limited layout options, such as row-major or column-major orderings. Using a limited set of layouts makes layout assignment tractable since the compiler can enforce specific layouts at operation boundaries, ensur-

ing composability between tensor operations. However, this restriction induces expensive layout conversions that can significantly degrade overall performance.

ML-first frameworks with hand-designed layouts. More recent works targeting machine learning applications, including EVA [18], CHET [19], LoLa [8], and Orion [20], employ more sophisticated but still limited sets of hand-designed layouts optimized for specific operations. For example, Orion provides specialized packing strategies for matrix-vector multiplication and convolution operations commonly found in CNNs, achieving strong performance on these specific workloads. While these frameworks demonstrate the value of operation-specific layout optimization, their reliance on predetermined packing schemes limits their ability to adapt to diverse tensor computation patterns. Compared to these approaches, Rotom's automated search can discover efficient layouts for a broader variety of models and operations, while also generalizing to transformers and other architectures.

Full layout optimization compilers. Recent compilers have moved toward supporting nearly arbitrary layouts with automated optimization. Fhelipe is a recent work that introduces a flexible layout abstraction, allowing arbitrary and intermixed dimension orderings [34]. However, the main limitation with Fhelipe's search approach and layout abstraction is that it doesn't support more complex data layouts, such as diagonalization. Without diagonalization, Fhelipe cannot leverage baby-step giant-step (BSGS) optimizations for tensor operators, which decompose large rotation distances into combinations of smaller rotations, reducing the total rotation count from $O(n)$ to $O(\sqrt{n})$. These complex layouts can greatly reduce the operational costs of frequent tensor operations by eliminating expensive rotations. Rotom alleviates the layout limitation by supporting rolled layouts and roll-specific optimizations.

Viaduct-HE is another recent work that provides additional support for more complex data layouts [47]. Rotom borrows Viaduct-HE's roll-based layout notation but differs fundamentally in its optimization approach and use of rolls. Viaduct-HE uses bottom-up enumeration to explore the search space, where vector derivation composes one ciphertext efficiently from an existing ciphertext through rotation. However, this derivation process is expensive: Viaduct-HE finds layout derivations concretely by instantiating each potential vector layout and performing element-wise index comparisons to determine overlap. This materialization overhead compounds with the bottom-up enumeration strategy—for each intermediate layout generated during the search, the system must perform expensive index comparisons across all derivation candidates, with each comparison requiring element-wise matching across vectors of thousands of elements ($n = 4K$-$64K$). In contrast, Rotom employs a top-down layout assignment approach using its high-level layout alignment rules and conversion operators that compose without requiring concrete instantiation. Critically, while Viaduct-HE uses rolls only for input tensors, Rotom treats them as first-class conversions through its `ApplyRoll` operator. This enables Rotom to uses `ApplyRoll` as a lightweight layout conversion operator and derive additional optimizations, such as BSGS, from this conversion.

Other works such as Porcupine and Coyote use heavyweight search techniques, i.e., program synthesis and simulated annealing, to solve layout assignment [16, 39]. While these approaches can find high-quality solutions for small programs, their expensive search procedures do not scale to the larger benchmarks we target.

10 Conclusion

Rotom addresses a key barrier to practical HE by automatically finding efficient layout assignments for tensor programs. By automating layout optimization, this work makes HE more accessible for practical applications. Future work will focus on discovering additional novel abstractions, similar to the `ApplyRoll` operator, that can further enhance layout assignment strategies. More broadly, we believe similar compiler-driven optimization techniques can address other performance bottlenecks across the HE compilation pipeline.

Acknowledgments

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

Ethical Considerations

Rotom is a tensor compiler designed to simplify the programming challenges and improve the performance of HE programs. While our work is purely technical, we recognize that any advancement in privacy-enhancing technologies has dual-use potential. We carefully evaluated the ethical implications of making HE computations more efficient and accessible.

Potential Negative Use Cases. More efficient HE could enable organizations to evade legitimate regulatory oversight, e.g., financial institutions hiding discriminatory lending algorithms from regulators, or healthcare providers processing patient data in violation of informed consent. Enhanced performance could create privacy asymmetries where powerful entities analyze user data while preventing users from understanding what computations are being performed, protecting the data processor rather than the data subject. It could also

facilitate illegal marketplaces or enable adversarial actors to process stolen encrypted data without detection.

Weighing Benefits Against Harms. Efficient HE enables privacy-preserving analysis of sensitive medical data across institutions—genomic studies, disease prediction, treatment optimization—that would otherwise be impossible due to privacy regulations. Private financial computations protect individuals from data breaches and enable secure credit scoring without exposing personal information. Making encrypted inference practical allows users to benefit from ML models without surrendering private data, critical for sensitive domains like mental health and personal finance. Improving HE performance also democratizes access to privacy-preserving computation, allowing smaller organizations and research institutions to deploy these protections rather than limiting them to well-resourced entities.

Our Assessment and Future Considerations. We conclude that publishing Rotom serves the greater good. The programming and performance improvements from layout optimization do not create fundamentally new capabilities, as they primarily serve to make HE programs easier to write and operate faster. The negative use cases we identified are already theoretically possible; Rotom merely changes the programming and performance characteristics. Withholding this research would not prevent adversarial actors from developing similar optimizations independently, as the underlying techniques are standard HE compiler optimization problems. In contrast, the positive applications have clear and substantial societal benefits, and the current programming and performance gap remains a major barrier to adoption of privacy-preserving technologies in legitimate contexts. We publish our work through academic peer review, provide sufficient technical detail for reproducibility, and engage with the HE compiler community to responsibly integrate these optimizations into widely-used open-source infrastructure (e.g., HEIR [3]). We believe that advancing the efficiency and accessibility of HE through compiler optimization is essential for realizing the promise of privacy-preserving computation in critical domains, and that responsible publication and engagement with the research community best serves societal interests.

Open Science

We fully support the principles of the Open Science Policy. We open source the research artifacts produced for this project under the Berkeley Software Distribution (BSD) license, including the source code of Rotom and ways to reproduce all the evaluation results reported in this paper. We also encourage the scientific community unrestricted access to review, validate, and expand upon our work. Our open-source implementation is available at https://github.com/cmu-cryptosystems/Rotom, and our artifact is archived at https://zenodo.org/records/17957733.

A Tensor Operators

Tensor Operator	Description
`Tensor`	tensor initialization
`Add`	tensor addition
`Mul`	tensor multiplication
`Sub`	tensor subtraction
`MatMul`	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

Tensor Operator	Signature
`Tensor`	`TensorOp -> TensorOp`
`Add`	`(TensorOp, TensorOp) -> TensorOp`
`Mul`	`(TensorOp, TensorOp) -> TensorOp`
`Sub`	`(TensorOp, TensorOp) -> TensorOp`
`MatMul`	`(TensorOp, TensorOp) -> TensorOp`
`Conv2D`	`(TensorOp, TensorOp) -> TensorOp`
`Sum`	`(TensorOp, Int) -> TensorOp`
`Product`	`(TensorOp, Int) -> TensorOp`
`Transpose`	`TensorOp -> TensorOp`
`Reshape`	`(TensorOp, [Int]) -> TensorOp`
`Permute`	`(TensorOp, [Int]) -> TensorOp`
`Index`	`(TensorOp, Int) -> TensorOp`

B Layout Operators

Layout Operator	Description
`LayoutOp`	Initialize tensor with layout
`Tensor Ops`	Layout operator equivalent of tensor ops
`ApplyReplicate`	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

Layout Operator	Signature
`LayoutOp`	(`Tensor`, `Representation`) -> `LayoutOp`
`Tensor Ops`	(`LayoutOp`, ...) -> `LayoutOp`
`ApplyReplicate`	(`LayoutOp`, `Dim`) -> `LayoutOp`
`ApplyPermute`	(`LayoutOp`, `Representation`) -> `LayoutOp`
`ApplyRoll`	(`LayoutOp`, `Roll`) -> `LayoutOp`
`ApplyCompact`	(`LayoutOp`, `Representation`) -> `LayoutOp`
`ApplyRotRoll`	(`LayoutOp`, `Roll`) -> `LayoutOp`
`ApplyBSGSRoll`	(`LayoutOp`, `Roll`) -> `LayoutOp`
`ApplyBSGSMatMul`	(`LayoutOp`, `LayoutOp`) -> `LayoutOp`
`ApplyStrassens`	(`LayoutOp`, `LayoutOp`) -> `LayoutOp`

C Alignment Rules & Examples

In this section, we provide examples on how to easily check for alignment using Rotom's layout alignment rules.

Traversal Dimension Alignment To give an example of traversal dimension alignment for `MatMul`, consider an example with two 4×4 matrices, A and B. By the dimension alignment map, the 1st dimension A should be aligned with the 0th dimension of B; all other dimensions are aligned with a repeated dimension. Using Einstein's notation, $C_{ij} = \sum_k A_{ik} B_{kj}$ represents a possible nested loop computation for `MatMul`; i is the outermost loop iterator, followed by j, then k. For A, i indexes the 0th dimension, k indexes the 1st dimension, and j repeats the value denoted by A_{ik}. This traversal is represented as A: `[0:4][R:4][1:4]`, where j is a repeated dimension. For B, the aligned layout is represented as B: `[R:4][1:4][0:4]`. In this example, we can see how all of the traversal dimension alignment rules are satisfied: 1. dimensions are aligned according to the dimension alignment map and 2. all traversals have the same extent and stride.

Roll Alignment 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`. This is because A:`[1:4]` is rolled whereas B:`[0:4]` is not. Roll alignment dictates both traversal dimensions need to iterate over the same tensor indices. To align these two layouts, Rotom uses `ApplyRoll` to apply the same roll, `Roll(0,1)`, to B. However, not all rolls need to be matched: a rolled traversal dimension aligned with a repeated dimension preserves roll alignment, as the modified tensor indices still map to the same repeated tensor indices. 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.

D `ApplyPermute` vs `ApplyRoll` Example

To illustrate the difference in layout conversion costs, consider a matrix multiplication workload, where A is encrypted and begins with a repeated column-major packing (A: `[64];[1:64][0:64]`). B is a 64×64 plaintext matrix, so changing the layout of B comes at no additional cost. To move the summation dimension to vector dimensions, we can use `ApplyPermute` to swap the traversal dimensions `[1:64]` and `[64]` in A, to get A: `[1:64];[64][0:64]`. This layout conversion would require 64 masks to isolate every column in A, followed by $64 \times log_2(64)$ rotations and additions to replicate values in each ciphertext. On the other hand, Rotom can use `ApplyRoll` to swap `[1:64]` by `[64]`, converting the layout to A: `Roll(0,1) [1:64];[64][0:64]`. This roll transforms A by internally rotating each column by its column index—the first column is rotated by 0, the second column is rotated by 1, etc. Performing this roll requires 2×64 masks to isolate the two partitions within each column, 2×64 rotations to internally rotate each column, and 2×64 additions to combine each rotated partition. In total, `ApplyPermute` uses 64 multiplications, 384 rotations, and 384 additions, whereas `ApplyRoll` uses 63 multiplications, 126 rotations, and 189 additions. Thus, the total cost of using `ApplyRoll` requires $3\times$ fewer rotations compared to `ApplyPermute`.

E Design 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 search space only supports different tiled and rolled layouts, due to constraints on its layout representation. There could potentially exist new conversion operators apart from `ApplyRoll` that uses unrepresented layouts with reduced data movement costs. Using Rotom search space heuristics, Rotom does find a practically efficient the layout assignment within its search space.

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 [5, 31, 38]. Finding a common abstraction that bridges the gap between both encoding formats could lead to even cheaper layout assignments.

F 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 [**?**, **?**, 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 [6, 13, 14, 17, 18, 24, 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 [3]) *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 [4, 10, 11, 15, 19, 20, 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.

References

[1] 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. In *Proceedings on Privacy Enhancing Technologies*, volume 2023, pages 325–342, 2023.

[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] Asra Ali, Jaeho Choi, Bryant Gipson, Shruthi Gorantala, Jeremy Kun, Wouter Legiest, Lawrence Lim, Alexander Viand, Meron Zerihun Demissie, and Hongren Zheng. Heir: A universal compiler for homomorphic encryption. *arXiv preprint arXiv:2508.11095*, 2024.

[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*, pages 57–68, 2019.

[5] 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.

[6] 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*, pages 3–13, 2019.

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

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

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

[10] 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*, pages 13–19, 2015.

[11] 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.

[12] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology – ASIACRYPT*, pages 409–437. Springer, 2017.

[13] 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 *USENIX Security Symposium*, pages 6993–7010, 2024.

[14] 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.

[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 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 375–389, 2021.

[16] 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 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 375–389, 2021.

[17] Eric Crockett, Chris Peikert, and Chad Sharp. Alchemy: A language and compiler for homomorphic encryption made easy. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 1020–1037, 2018.

[18] 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 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 546–561, 2020.

[19] 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 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–156, 2019.

[20] Austin Ebel, Karthik Garimella, and Brandon Reagen. Orion: A fully homomorphic encryption framework for deep learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, Rotterdam, Netherlands, March 2025. Association for Computing Machinery.

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

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

[23] 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 *International Conference on Machine Learning*, 2016.

[24] 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. In *Proceedings of the 3rd Workshop on Privacy-Preserving Machine Learning in Practice*, pages 1–12. ACM, 2023.

[25] Shai Halevi and Victor Shoup. Algorithms in helib. In *Annual International Cryptology Conference*, pages 554–571. Springer, 2014.

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

[27] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. Iron: Private inference on transformers. In *Advances in Neural Information Processing Systems*, volume 35, pages 15718–15731, 2022.

[28] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. Cheetah: Lean and fast secure {Two-Party} deep neural network inference. In *USENIX Security Symposium*, 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 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 *USENIX Security Symposium*, pages 1651–1669, 2018.

[33] Arjun Kharpal. Samsung bans use of a.i. like chatgpt for employees after misuse of the chatbot. *CNBC*, 2023. https://www.cnbc.com/2023/05/02/samsung-bans-use-of-ai-like-chatgpt-for-staff-after-misuse-of-chatbot.html.

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

[36] Yongwoo Lee, Seonyoung Cheon, Dongkwan Kim, Dongyoon Lee, and Hanjun Kim. {ELASM}:{Error-Latency-Aware} scale management for fully homomorphic encryption. In *USENIX Security Symposium*, 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. In *Network and Distributed System Security Symposium*. Internet Society, 2025.

[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. In *USENIX Security Symposium*, pages 2505–2522, 2020.

[41] Jungho Moon, Dongwoo Yoo, Xiaoqian Jiang, and Miran Kim. Thor: Secure transformer inference with homomorphic encryption. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2025.

[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 *IEEE*

Symposium on Security and Privacy, 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, Paper 2024/1921*, 2024.

[45] 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, Zachary 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. In *Advances in Neural Information Processing Systems*, volume 32, pages 8024–8035, 2019.

[46] Siladitya Ray. Samsung bans chatgpt among employees after sensitive code leak. *Forbes*, 2023. https://www.forbes.com/sites/siladityaray/2023/05/02/samsung-bans-chatgpt-and-other-chatbots-for-employees-after-sensitive-code-leak/.

[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. *The New York Times*, 2023. https://www.nytimes.com/2023/03/31/technology/chatgpt-italy-ban.html.

[49] Mark Stone. Is copilot safe? microsoft copilot security concerns explained. *Concentric AI*, 2024. https://concentric.ai/too-much-access-microsoft-copilot-data-risks-explained/.

[50] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. {HECO}: Fully homomorphic encryption compiler. In *USENIX Security Symposium*, pages 4715–4732, 2023.

[51] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. Sok: Fully homomorphic encryption compilers. In *IEEE Symposium on Security and Privacy*, pages 1092–1108. IEEE, 2021.

[52] Jeremy White. How strangers got my email address from chatgpt's model. *The New York Times*, 2023. https://www.nytimes.com/interactive/2023/12/22/technology/openai-chatgpt-privacy-exploit.html.

[53] Shenglai Zeng, Jiankun Zhang, Pengfei He, Yiding Liu, Yue Xing, Han Xu, Jie Ren, Yi Chang, Shuaiqiang Wang, Dawei Yin, and Jiliang Tang. The good and the bad: Exploring privacy issues in retrieval-augmented generation (rag). In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 4505–4524, 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. In *Network and Distributed System Security Symposium*. Internet Society, 2025.