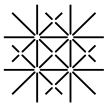


University of Basel

A Compiler for Homomorphically Encrypted Permutations

Bachelor's Thesis

University of Basel – Faculty of Science
Department of Mathematics and Computer Science
Computer Networks Group

Examiner: Prof. Dr. Christian Tschudin
Supervisor: Ali Ajorian, MSc

Heinrich Ahrend
h.ahrend@unibas.ch

July 31, 2026

Abstract

Homomorphic encryption (HE) is capable of processing encrypted data without revealing secret information. While HE is emerging as a powerful tool for outsourced computation, developing efficient algorithms is challenging because HE schemes only offer a limited set of basic operations. Despite this, the vectorized message space of arithmetic HE schemes is well suited for parallelized computations and has been utilized for homomorphic matrix algorithms. However, evaluating permutations in a privacy-preserving context remains unexplored. We therefore build upon existing homomorphic matrix algorithms to efficiently compose and apply encrypted permutations. In addition, we propose an algorithm that homomorphically computes the inversion number of a permutation at a multiplicative depth of 3. To deploy these methods in practice, we implement a compiler that automatically maps programs from a high-level language to optimized homomorphic routines for outsourced computation. We demonstrate its effectiveness by analyzing the achieved runtime improvements.

Table of Contents

1 Introduction

Homomorphic encryption (HE) allows untrusted parties to perform computations directly on encrypted data. This unlocks new opportunities for outsourced data processing in critical domains, which has led to significant developments within the field in recent years [28]. However, the basic operations that HE schemes provide to process encrypted data are fairly limited. Hence, the deployment of HE in real-world applications remains challenging due to substantial computational overhead.

To overcome this issue, arithmetic HE schemes leverage the ability to pack an array of values into a single ciphertext by dividing messages into multiple slots. Subsequently, the homomorphic operations offered by these schemes are performed slot-wise in a single instruction, multiple data (SIMD) fashion [22]. This allows multiple values to be processed simultaneously within one ciphertext. However, homomorphic computations that require interaction between different slots of a message are difficult to implement due to the limited set of basic operations. Therefore, optimally arranging the data within a message becomes an important but non-trivial and highly domain-specific task. As a solution, the responsibility of choosing a layout and transforming a program into the corresponding homomorphic routines can be delegated to a compiler. Although this approach has already been widely adopted for machine learning algorithms [1, 7, 16, 25], applying it to the homomorphic processing of permutations remains unexplored. However, permutations play a crucial role in problems like homomorphic sorting [33] and rank aggregation [26].

To fill this gap, we present a Python framework for automatically outsourcing the homomorphic evaluation of permutations based on the BFV scheme [17]. More specifically, the remaining chapters are structured as follows. In Chapter 2, the theoretical background is introduced. Chapter 3 then describes how compositions, applications and inversion numbers of permutations are homomorphically computed. Chapter 4 details our implementation of a compiler that transforms programs from a domain-specific language (DSL) for permutations into optimized homomorphic routines. We evaluate this compiler in Chapter 5 by benchmarking the achieved speedup. Finally, Chapter 6 summarizes our results and outlines future work.

2
Background

2.1 Notation

For a field F and a positive integer n, we write $F^{n\times n}$ for the set of $n \times n$ matrices with entries in F. Given a matrix $A \in F^{n\times n}$, we use lower-case a_{ij} to refer to the entry in the i-th row and j-th column of A. Similarly, we write v_i for the i-th entry of a vector $v \in F^n$. We use $\mathbf{0}$ to denote the zero vector in F^n.

Moreover, the ring of polynomials in X over a field F is denoted by $F[X]$. Given a polynomial $P \in F[X]$, we write $F[X]/(P)$ for the quotient ring of $F[X]$ modulo P. We use $R \cong S$ to denote that two rings R and S are isomorphic.

For a prime power q (i.e., $q = p^n$ for some prime p and a positive integer n), a finite field of order q is denoted by $\mathbb{F}_q$. Let N be a power of 2. We write $\overline{\Phi}_{2N}$ for the polynomial $X^N + 1 \in \mathbb{F}_q[X]$.[1]

We write $x \leftarrow \mathsf{A}$ to denote that x is assigned the output of the (possibly randomized) algorithm A.

2.2 Homomorphic Encryption

Let $\mathcal{M}$ and $\mathcal{C}$ be sets, called the message space and ciphertext space, respectively. Let $\mathcal{F}$ be a set of functions $f\colon \mathcal{M}^d \to \mathcal{M}$ for various $d \in \mathbb{N}$. A (public key) homomorphic encryption scheme [28] is a tuple of probabilistic polynomial-time algorithms $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ where:

- The key generation algorithm KeyGen takes as input the security parameter λ. It outputs a secret key sk, a public key pk and an evaluation key evk.
- The encryption algorithm Enc takes as input the public key pk and a message $m \in \mathcal{M}$. It outputs a ciphertext $c \in \mathcal{C}$.
- The decryption algorithm Dec takes as input the secret key sk and a ciphertext $c \in \mathcal{C}$. It outputs a plaintext $m \in \mathcal{M}$. For correctness, we require that $\mathsf{Dec}_{\mathsf{sk}}(\mathsf{Enc}_{\mathsf{pk}}(m)) = m$ for all $m \in \mathcal{M}$.

[1] This is the image of the $2N$-th cyclotomic polynomial under the natural map to $\mathbb{F}_q[X]$.

- The evaluation algorithm Eval takes as input the evaluation key evk, a function $f \in \mathcal{F}$ of arity d and ciphertexts $c_1, \ldots, c_d \in \mathcal{C}$. It outputs a ciphertext $c \in \mathcal{C}$.

We say that the scheme is $\mathcal{F}$-homomorphic if

$$\mathsf{Dec}_{\mathsf{sk}}(\mathsf{Eval}_{\mathsf{evk}}(f, c_1, \ldots, c_d)) = f(m_1, \ldots, m_d) \quad \text{for every } f \in \mathcal{F},$$

where d is the arity of f and $c_i \leftarrow \mathsf{Enc}_{\mathsf{pk}}(m_i)$ for all $i \in \{1, \ldots, d\}$. In other words, the diagram in Figure 2.1 commutes.

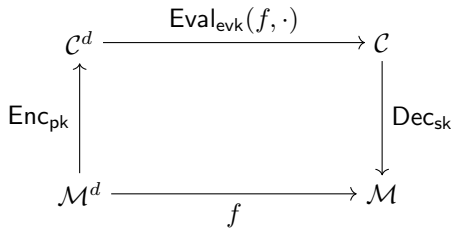


Figure 2.1: Commutative diagram illustrating the homomorphic property

A scheme is called a fully homomorphic encryption (FHE) scheme if it is $\mathcal{F}$-homomorphic for $\mathcal{F}$ being the set of all (efficiently computable) functions. The first FHE scheme was presented by Gentry in [18] (which we also refer to for a more elaborated definition of HE). In practice, most FHE schemes offer a set of basic operations that is functionally complete, typically including some kind of addition (Add) and multiplication (Mult). To evaluate a function $f \in \mathcal{F}$, the function is then represented as a circuit based on these elementary operations.

Modern FHE schemes are generally based on the ring learning with errors (RLWE) problem [27] and thus rely on adding random noise to a message. Computing homomorphic operations on a ciphertext c, increases the noise carried by c. As long as the noise stays small enough, $\mathsf{Dec}_{\mathsf{sk}}(c)$ will be able to reconstruct the original message. Hence, the circuit depth of f becomes a limitation when working with HE. As a solution, Gentry introduced a technique called bootstrapping, which is able to reset the noise by evaluating a scheme's decryption algorithm homomorphically. However, bootstrapping is computationally expensive and therefore often not feasible in practice. This has led to the notion of leveled fully homomorphic encryption, in which the parameter size depends polynomially on the depth of functions in $\mathcal{F}$. That allows us to evaluate arbitrary functions, as long as their depth can be bounded before running KeyGen.

2.3 Arithmetic HE Schemes

Current HE schemes can be divided into boolean and arithmetic schemes [4].

In boolean schemes such as FHEW [15] and TFHE [12], ciphertexts usually represent a single bit and functions are expressed as boolean circuits. The bootstrapping offered by these schemes is comparably fast and often allows the simultaneous evaluation of univariate functions (known as programmable bootstrapping).

Arithmetic schemes typically encrypt integers (BFV [17], BGV [9]) or floating point numbers (CKKS [11]) and support addition and multiplication as the basic operations. In contrast to boolean schemes, bootstrapping is considerably slower and non-linear functions are difficult to evaluate due to the lack of programmable bootstrapping. Hence, these schemes are often used in the context of leveled FHE [9]. However, arithmetic schemes can leverage a technique called batching to operate on multiple values in a SIMD manner.

We illustrate the idea of batching using the BFV/BGV scheme, although similar concepts apply to CCKS. Typically, the message space is chosen to be $\mathcal{M} = \mathbb{F}_p[X]/(\overline{\Phi}_{2N})$, where the plaintext modulus p is an odd prime number and the ring dimension N is a power of 2. It can be shown [22] that the polynomial $\overline{\Phi}_{2N} \in \mathbb{F}_p[X]$ factors as

$$\overline{\Phi}_{2N} = P_1 \cdots P_k$$

for some integer k, with distinct irreducible polynomials $P_i \in \mathbb{F}_p[X]$ of equal degree $d = N/k$. Moreover, d can be characterized as the multiplicative order of p modulo $2N$ (i.e., the smallest positive integer satisfying $p^d \equiv 1 \pmod{2N}$). Because $\mathbb{F}_p[X]/(P_i) \cong \mathbb{F}_{p^d}$, the Chinese remainder theorem yields

$$\mathcal{M} \cong \mathbb{F}_p[X]/(P_1) \times \cdots \times \mathbb{F}_p[X]/(P_k) \cong \mathbb{F}_{p^d} \times \cdots \times \mathbb{F}_{p^d} = \mathbb{F}_{p^d}^k.$$

Hence, messages can be interpreted as vectors of length k over $\mathbb{F}_{p^d}$. This allows us to store multiple values from $\mathbb{F}_{p^d}$ in separate slots of a message. As depicted in Figure 2.2, addition and multiplication in $\mathcal{M}$ now correspond to the slot-wise addition and multiplication in $\mathbb{F}_{p^d}^k$. To maximize the degree of parallelism k, we usually require that $p \equiv 1 \pmod{2N}$. In that case, $d = 1$ and therefore $k = N$, so we obtain $\mathcal{M} \cong \mathbb{F}_p^N$. For simplicity, we will often identify the message space as $\mathcal{M} = \mathbb{F}_p^N$.

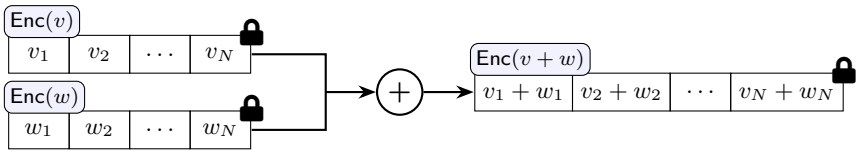


Figure 2.2: Homomorphic slot-wise addition of two messages $v, w \in \mathbb{F}_p^N$

Apart from addition and multiplication, arithmetic HE schemes offer a third basic operation Rot, which enables interaction between different slots: By exploiting the Galois automorphisms on $\mathcal{M}$, we can realize cyclic rotations of the plaintext slots. For a ciphertext $c \in \mathcal{C}$ and an index $l \in \mathbb{Z}$, evaluating $\mathsf{Rot}_{\mathsf{evk}}(c, l)$ shifts the slots underlying c by the amount l (with cyclic wrap-around). If $l > 0$, the slots are shifted to the left, otherwise they are shifted to the right. In BFV/BGV this requires some care, since rotations are applied to the slots $1, \ldots, N/2$ and $N/2 + 1, \ldots, N$ separately (see Figure 2.3).

2.4 Permutations

A permutation [5] on a set X is a bijection $X \to X$. The symmetric group S_n of degree n is the group consisting of all permutations on $\{1, \ldots, n\}$. The group operation is function composition $\circ$ and the neutral element is the identity mapping id.

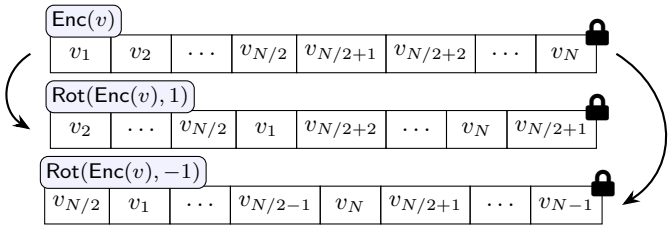


Figure 2.3: Homomorphic rotation of plaintext slots in BFV/BGV

Given a permutation $\sigma \in S_n$ and a vector $v \in F^n$, we can reorder the entries of v by mapping v_i to the $\sigma(i)$-th entry. Formally,

$$\mathsf{Apply}(\sigma, v) = \left(v_{\sigma^{-1}(1)}, \ldots, v_{\sigma^{-1}(n)}\right).$$

We can also represent $\sigma \in S_n$ by its permutation matrix $\varphi(\sigma) = (a_{ij}) \in F^{n \times n}$, where

$$a_{ij} = \begin{cases} 1 & \text{if } i = \sigma(j), \\ 0 & \text{otherwise.} \end{cases}$$

Computing $\mathsf{Apply}(\sigma, v)$ now corresponds to the matrix-vector multiplication $\varphi(\sigma)v$. Additionally, the mapping φ is a group homomorphism, i.e., $\varphi(\sigma)\varphi(\eta) = \varphi(\sigma \circ \eta)$ for all $\sigma, \eta \in S_n$. Representing permutations in this way, allows us to exploit the vectorized structure of $\mathcal{M}$ to optimize homomorphic computations involving permutations.

Furthermore, we call $(i, j) \in \{1, \ldots, n\}^2$ an inversion [24] of the permutation $\sigma \in S_n$ if $i < j$ and $\sigma(i) > \sigma(j)$. In other words, an inversion of σ is a pair of positions whose corresponding values are out of order. The inversion number $\mathrm{inv}(\sigma)$ counts all such pairs and can be expressed as

$$\mathrm{inv}(\sigma) = \#\{(i, j) \in \{1, \ldots, n\}^2 \mid i < j \text{ and } \sigma(i) > \sigma(j)\}.$$

As an example, consider the permutation $\sigma \in S_3$ defined by $\sigma(1) = 2$, $\sigma(2) = 3$ and $\sigma(3) = 1$. The pairs $(1, 3)$ and $(2, 3)$ are inversions of σ since $\sigma(1) > \sigma(3)$ and $\sigma(2) > \sigma(3)$, respectively. In contrast, $(1, 2)$ is not an inversion because $\sigma(1) < \sigma(2)$. Therefore, the inversion number of σ is $\mathrm{inv}(\sigma) = 2$.

2.5 Efficient Computation

Due to the limited set of basic operations, efficient homomorphic computation requires new approaches in algorithm and program design. In arithmetic schemes, multiplications increase the noise significantly more than additions or rotations, while multiplications and rotations exceed additions in computational overhead. Additionally, multiplying a ciphertext by a constant plaintext (CMult) is computationally cheaper than a regular multiplication (see Figure 4.6). Therefore, designing homomorphic algorithms becomes a challenge of balancing the multiplicative depth against the number of multiplications and rotations.

This is highly dependent on how we pack our data into a vector from the message space $\mathcal{M}$, because after encryption, it is difficult to restructure the packed data by using only homomorphic operations. In recent years, considerable effort has been devoted to finding

efficient layouts (packings) and algorithms for homomorphic matrix computation, which we can apply to optimize the evaluation of permutations. In the following, we present a selection of these approaches.

2.5.1 Diagonal Packing

Matrix-vector multiplication in the homomorphic setting has been studied by Halevi and Shoup in [20] and [21]. Let $A \in F^{n \times n}$ be a matrix let $v \in F^n$ be a vector. For $n = 3$ we can express their product using diagonals of A and rotations of v by

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} a_{11} \\ a_{22} \\ a_{33} \end{pmatrix} \odot \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} + \begin{pmatrix} a_{12} \\ a_{23} \\ a_{31} \end{pmatrix} \odot \begin{pmatrix} v_2 \\ v_3 \\ v_1 \end{pmatrix} + \begin{pmatrix} a_{13} \\ a_{21} \\ a_{32} \end{pmatrix} \odot \begin{pmatrix} v_3 \\ v_1 \\ v_2 \end{pmatrix},$$

where $\odot$ denotes entry-wise multiplication. To put this in more formal terms, for every $i \in \{0, \ldots, n-1\}$ we define

$$D_i(A) = (a_{1,1+i}, \ldots, a_{n-i,n}, a_{n-i+1,1}, \ldots, a_{n,i}) \in F^n,$$

which is the i-th diagonal of A (with cyclic wrap-around). Moreover, let $\pi^i \in S_n$ be the cyclic rotation to the left by i positions. Then it holds that

$$Av = \sum_{i=0}^{n-1} D_i(A) \odot \mathsf{Apply}(\pi^i, v).$$

Since all involved operations are available in arithmetic HE, this approach can be implemented by packing every diagonal vector $D_i(A) \in F^n$ into a separate plaintext. However, in the original work it is assumed that the matrix dimension n matches the amount of slots N, while in practice n is usually much smaller than N. To make the cyclic rotations work nonetheless, we can replicate v once (under the assumption $2n \leq N/2$) and fill the other plaintext slots with zeros. This replicated version $\tilde{v}$ can be computed homomorphically by adding a rotated version of v to itself. The resulting circuit is shown in Figure 2.4.

In [30] this approach is extended to support matrix-matrix multiplications. We slightly modify their algorithm, since it computes the lower diagonals of the resulting matrix, whereas our previously defined packing corresponds to the upper diagonals.[2] To match the setting of permutations, we also restrict $A, B \in F^{n \times n}$ to be square matrices. The i-th diagonal of their product $C = AB$, where $i \in \{0, \ldots, n-1\}$, is then given by

$$D_i(C) = \sum_{j=0}^{n-1} D_j(A) \odot \mathsf{Apply}(\pi^j, D_{i-j \bmod n}(B)).$$

We have implemented this analogously to the homomorphic matrix-vector multiplication.

2.5.2 Row-major Packing

Another technique often used for matrix-matrix multiplication is the row-major packing [2, 23, 32]. As illustrated in Figure 2.5, this method concatenates the rows of a matrix and

[2] Lower diagonals start below the main diagonal (i.e., at element $a_{1+i,1}$), while upper diagonals start above the main diagonal. They are identical up to a cyclic shift.

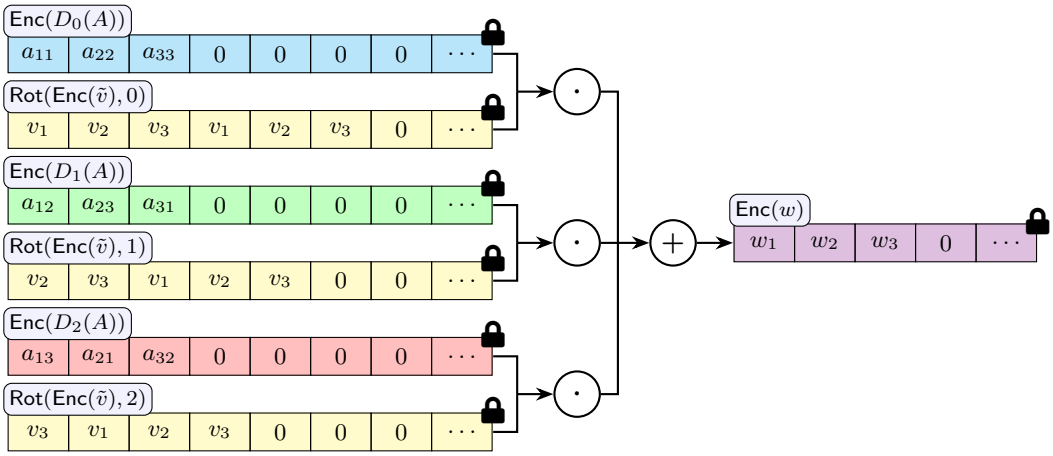


Figure 2.4: Homomorphic computation of $w = Av$ for a matrix $A \in F^{3\times 3}$ and a vector $v \in F^3$. The matrix A is diagonally packed and $\tilde{v}$ is the replicated version of v.

stores them in a single plaintext. Clearly, the amount of slots N must be at least n^2 to store matrices of size $n \times n$. Since algorithms from the literature usually assume that n is a power of 2, we also add a zero padding to obtain a matrix of size $\tilde{n} \times \tilde{n}$ where $\tilde{n} = 2^{\lceil \log_2 n \rceil}$.

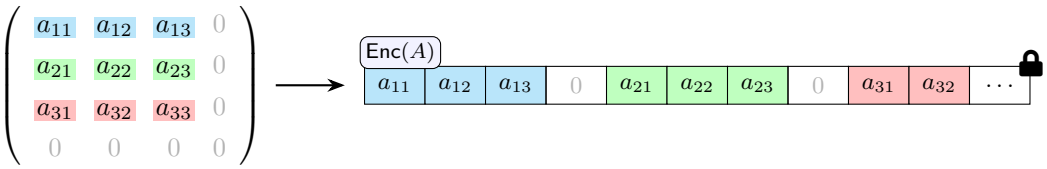


Figure 2.5: Row-major packing of a matrix $A \in F^{3\times 3}$

To multiply two matrices $A, B \in F^{n\times n}$, the mentioned works express their product as

$$AB = \sum_{i=1}^{n} \hat{A}_i \odot \hat{B}_i,$$

where $\hat{A}_i \in F^{n\times n}$ and $\hat{B}_i \in F^{n\times n}$ are matrices that can be precomputed separately based on row-major packed A and B. Note that from a compiler standpoint this allows us reuse these matrices if A or B is used in several multiplications. For example, in [2] and [32] each $\hat{A}_i$ is obtained by replicating the i-th column of A, and each $\hat{B}_i$ is obtained by replicating the i-th row of B. For $n = 3$ that results in

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{11} & a_{11} \\ a_{21} & a_{21} & a_{21} \\ a_{31} & a_{31} & a_{31} \end{pmatrix} \odot \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{11} & b_{12} & b_{13} \\ b_{11} & b_{12} & b_{13} \end{pmatrix}$$
$$+ \begin{pmatrix} a_{12} & a_{12} & a_{12} \\ a_{22} & a_{22} & a_{22} \\ a_{32} & a_{32} & a_{32} \end{pmatrix} \odot \begin{pmatrix} b_{21} & b_{22} & b_{23} \\ b_{21} & b_{22} & b_{23} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} + \begin{pmatrix} a_{13} & a_{13} & a_{13} \\ a_{23} & a_{23} & a_{23} \\ a_{33} & a_{33} & a_{33} \end{pmatrix} \odot \begin{pmatrix} b_{31} & b_{32} & b_{33} \\ b_{31} & b_{32} & b_{33} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}.$$

Although this packing is mostly used for matrix-matrix multiplication, we can easily support the multiplication of a matrix $A \in F^{n\times n}$ with a vector $v \in F^n$ as a special case. By embedding v inside the first column of

$$V = \begin{pmatrix} v & \mathbf{0} & \cdots & \mathbf{0} \end{pmatrix} \in F^{n\times n},$$

the result can be extracted from the matrix-matrix product

$$AV = \begin{pmatrix} Av & \mathbf{0} & \cdots & \mathbf{0} \end{pmatrix} \in F^{n\times n}.$$

2.5.3 Complexity

The survey [6] describes the diagonal algorithm of [20] and the row-major algorithm of [23] as state-of-the-art solutions for homomorphic matrix computation. For row-major packing, the algorithm of [23] has since been improved by [2]. Roughly speaking, diagonal packing is preferable for matrix-vector products, whereas row-major packing is more suitable for matrix-matrix products. Their properties are summarized in Table 2.1.

Table 2.1: Complexity of matrix computations under different packings. The second and third column refer to the maximal dimension n and the amount of ciphertexts needed to store a matrix in $F^{n\times n}$, respectively. Again, $\tilde{n} = 2^{\lceil \log_2 n \rceil}$ is the next power of 2.

Packing	**Max. n**	**# Ctxts**	**Algorithm**	**# CMult**	**# Mult**	**# Rot**	**Depth**
Diagonal	$N/4$	n	Matrix-vector [20]	0	n	n	1
			Matrix-matrix [30]	0	n^2	n^2	1
Row-major	$2^{\lfloor \log_2 \sqrt{N/2} \rfloor}$	1	Matrix-vector & matrix-matrix [2]	$2\tilde{n}$	$\tilde{n}$	$2\tilde{n}(1+\log_2 \tilde{n}) - 2$	2

2.6 HE Compilers

As we have seen, writing programs for HE can be quite challenging due to the required expertise and the limited set of available operations. Consequently, numerous compilers have been developed in recent years to automate this task.

While approaches to support general-purpose programs exist [4, 35], most HE compilers are tailored to specific applications and impose restrictions on the control-flow and data types of source programs. These compilers usually focus on vector and tensor arithmetic in the context of machine learning [1, 7, 14, 16, 25]. However, there is currently no framework dedicated to performing operations that involve encrypted permutations.

The main task of an HE compiler is to translate source programs into homomorphic circuits. This process includes automated parameter selection, noise management and efficient data packing. For simplicity, our work primarily considers diagonal and row-major packing in the context of permutations, leaving specialized noise management and more advanced tensor packing strategies [1, 16, 25] for future exploration.

3

Encrypted Permutations

Since the basic operations of arithmetic HE schemes are applied in a SIMD fashion, moving data between individual ciphertext slots becomes notably difficult. Consequently, permutations play a fundamental role in arithmetic HE because they allow restructuring a ciphertext after encryption. For instance, efficiently permuting ciphertext slots when the permutation is public has already been investigated in [20] and [36]. However, applications such as permutation-based homomorphic sorting [33] may require the permutation to remain private. We therefore explore homomorphic computation with encrypted permutations in the following sections.

3.1 Application and Composition

As noted in Section 2.4, a permutation $\sigma \in S_n$ can be represented by its permutation matrix $\varphi(\sigma)$. This way, application and composition of permutations become special cases of matrix-vector and matrix-matrix multiplication, respectively. Given the vectorized structure of the message space $\mathcal{M}$, we thus encrypt $\varphi(\sigma)$ to match the setting of arithmetic HE. For homomorphic application and composition of permutations, this allows us to directly utilize the existing matrix algorithms from Section 2.5, which only reveal the permutation size n.[3]

3.2 Inversion Number

In addition to application and composition, we present a dedicated algorithm for homomorphically computing the inversion number of a permutation. Considering that the definition of an inversion is based on comparisons, which are generally inefficient to evaluate under arithmetic HE,[4] this may seem difficult to realize. However, using an approach inspired by Rothe diagrams [24], we now demonstrate how the need for comparisons can be bypassed. To achieve this, we first establish the mathematical basis and then show how to homomorphically compute the inversion number based on that concept.

[3] While a detailed security analysis is outside the scope of this work, such leakage is widely accepted in HE under the IND-CPA security model [23, 28].

[4] Homomorphic comparisons usually rely on polynomial approximation of a difference's sign [33].

For a permutation $\sigma \in S_n$ and $i \in \{1, \dots, n-1\}$, we define $M_i = (m_{kj}) \in F^{n \times n}$ by

$$m_{kj} = \begin{cases} 1 & \text{if } i < j \text{ and } k < \sigma(i), \\ 0 & \text{otherwise.} \end{cases}$$

Visually, each matrix M_i acts as a mask when forming the entry-wise product with another matrix: it selects the block of entries located to the right of the i-th column and above the $\sigma(i)$-th row. So, if we let $\varphi(\sigma) = (a_{kj})$ be the permutation matrix of σ, then an entry $a_{kj} \cdot m_{kj}$ of the product $\varphi(\sigma) \odot M_i$ equals 1 if and only if $a_{kj} = m_{kj} = 1$. By definition, this is equivalent to

$$i < j \quad \text{and} \quad \sigma(i) > k = \sigma(j),$$

which precisely means that (i, j) is an inversion. Hence, each inversion of the form (i, j) contributes a 1 to the matrix $\varphi(\sigma) \odot M_i$, while the other entries remain 0. Therefore, the inversion number of σ is obtained by summing all entries of the matrix Q, defined by

$$Q = \sum_{i=1}^{n-1} \varphi(\sigma) \odot M_i = \varphi(\sigma) \odot \sum_{i=1}^{n-1} M_i. \tag{3.1}$$

As an example, consider the permutation $\sigma \in S_3$ that swaps 1 and 3. Its inversions $(1, 2)$, $(1, 3)$ and $(2, 3)$ correspond to the non-zero entries of

$$Q = \varphi(\sigma) \odot (M_1 + M_2) = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \odot \left(\begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \right) = \begin{pmatrix} 0 & 0 & 2 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Consequently, the sum of the entries of Q is equal to $\text{inv}(\sigma) = 3$.

We now turn to the homomorphic realization of this technique. For the moment, assume that $\varphi(\sigma)$ is encrypted under row-major packing, resulting in a ciphertext denoted by P. Also recall from Section 2.5.2 that in this case, matrices of size $n \times n$ are padded to a size of $\tilde{n} \times \tilde{n}$ with $\tilde{n} = 2^{\lceil \log_2 n \rceil}$.

To obtain M_i from P for each $i \in \{1, \dots, n-1\}$, we utilize the matrices $C_i = (c_{kj}) \in F^{n \times n}$ and $R_i = (r_{kj}) \in F^{n \times n}$, where

$$c_{kj} = \begin{cases} 1 & \text{if } j = i, \\ 0 & \text{otherwise.} \end{cases} \quad \text{and} \quad r_{kj} = \begin{cases} 1 & \text{if } j > i, \\ 0 & \text{otherwise.} \end{cases}$$

First, the i-th column of $\varphi(\sigma)$ is extracted via homomorphic multiplication with the constant matrix C_i (see Figure 3.1). Its entries are then replicated to the left, yielding a matrix denoted by P_i. More precisely, this matrix is obtained using a standard rotate-and-sum algorithm [35], where the ciphertext is iteratively rotated by increasing powers of 2 and added to itself, doubling the number of replicated entries in each step. However, to ensure that data wrapping around from the right does not interfere with the occupied $\tilde{n}^2$ slots, this requires a BFV/BGV ring dimension of at least $4\tilde{n}^2$. Subsequently, all columns located to the right of the i-th column are extracted from P_i by multiplying with R_i, which yields a row-major packed version of M_i in the ciphertext space (see the last row of Figure 3.1).

Once the masks M_i have been obtained, we build Q according to Equation 3.1 via homomorphic multiplication of their sum with P. Finally, $\text{inv}(\sigma)$ is computed by summing

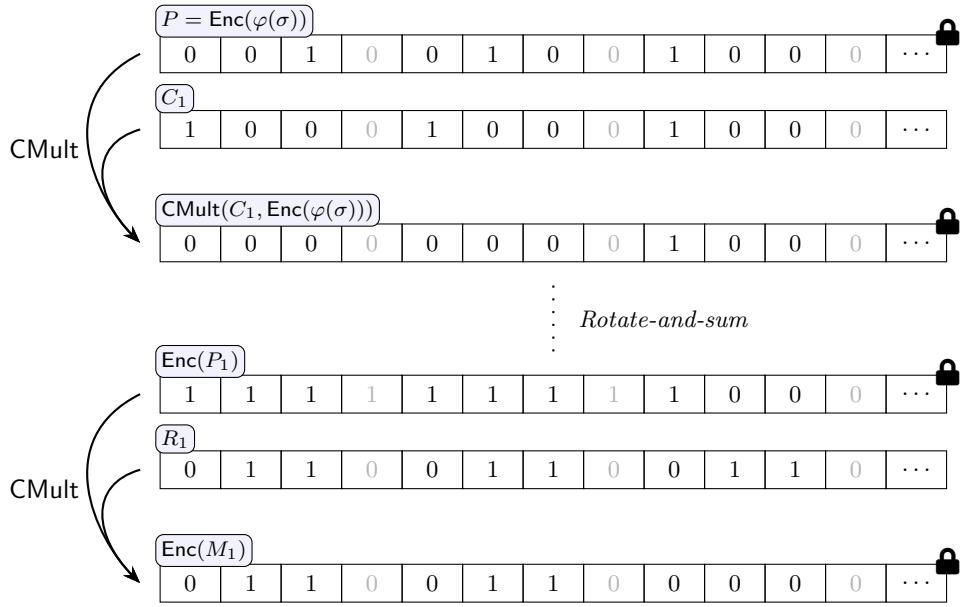


Figure 3.1: Homomorphic construction of the mask M_1, where $\sigma \in S_3$ is again the permutation swapping 1 and 3.

the entries of Q, using the same series of rotations and additions as in the replication step. This routine is illustrated in Figure 3.2 and places the inversion number into the first slot of the resulting ciphertext. The complete procedure is presented in Algorithm 1. Details regarding its implementation are discussed in Chapter 4.

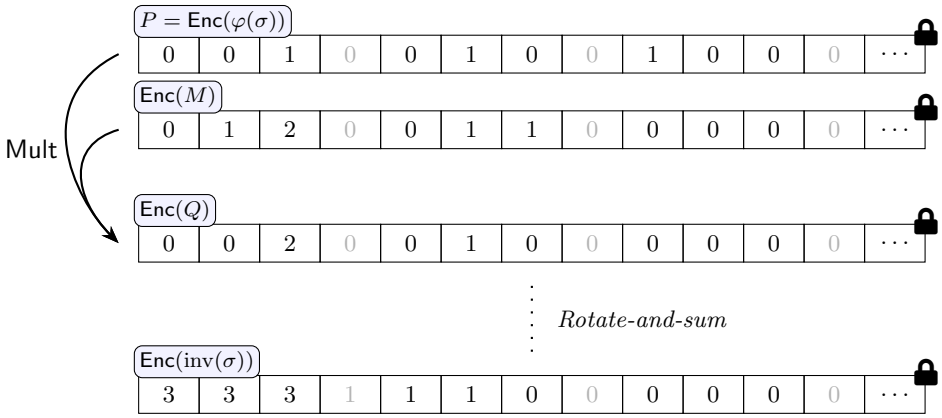


Figure 3.2: Computation of the inversion number for the running example from Figure 3.1. The matrix M is defined as the sum of M_1 and M_2.

In terms of complexity, Algorithm 1 requires only a single ciphertext-ciphertext multiplication (Mult) and $2(n-1)$ constant multiplications (CMult), while operating at a multiplicative depth of 3. The number of rotations (Rot) is given by $2n \log_2(\tilde{n})$.

However, when $\varphi(\sigma)$ is provided in diagonal packing instead of row-major packing, an efficient homomorphic application of the previous approach is no longer straightforward, as the required matrix entries are distributed across multiple ciphertexts. Therefore, we do not implement a dedicated algorithm to homomorphically compute the inversion number under

Algorithm 1 Computing the inversion number under row-major packing

Require: Row-major packed permutation matrix $P \leftarrow \mathsf{Enc}(\varphi(\sigma))$ for $\sigma \in S_n$. This matrix is padded to a size of $\tilde{n} \times \tilde{n}$ where $\tilde{n} = 2^{\lceil \log_2 n \rceil}$. The BFV/BGV ring dimension must be at least $4\tilde{n}^2$.

Ensure: The first slot of Q contains the inversion number $\mathrm{inv}(\sigma)$.

$M \leftarrow \mathbf{0}$
for $i = 1$ to $n - 1$ **do** ▷ *Create masks M_i*
  $P_i \leftarrow \mathsf{CMult}(C_i, P)$ ▷ *Extract i-th column*
  for $j = 0$ to $\log_2(\tilde{n}^2) - 1$ **do** ▷ *Replicate to the left*
    $P_i \leftarrow \mathsf{Add}(P_i, \mathsf{Rot}(C, 2^j))$
  $M_i \leftarrow \mathsf{CMult}(R_i, P_i)$ ▷ *Extract right columns*
  $M \leftarrow \mathsf{Add}(M, M_i)$ ▷ *Add M_i to the existing masks*
$Q \leftarrow \mathsf{Mult}(P, M)$ ▷ *Multiply by the sum of all M_i*
for $i = 0$ to $\log_2(\tilde{n}^2) - 1$ **do** ▷ *Sum the entries of Q*
  $Q \leftarrow \mathsf{Add}(Q, \mathsf{Rot}(Q, 2^i))$
return Q

diagonal packing. Instead, $\varphi(\sigma)$ is homomorphically repacked from diagonal to row-major layout when necessary, so that Algorithm 1 can be used. This transformation is achieved by extracting each matrix entry from its ciphertext via masking, rotating it to the correct position and accumulating the results. The introduced overhead consists of n^2 constant multiplications (CMult) and $n^2 - 1$ rotations (Rot), adding a multiplicative depth of 1.

4

Implementation

We implement a Python framework for working with homomorphically encrypted permutations. It consists of a source language, two target languages and a compiler.

The source language allows expressing programs that operate on permutations. Such programs can either be executed directly by a trusted party without encryption or can be compiled to delegate their execution to an untrusted party as shown in Figure 4.1. In the latter case, the compiler will encrypt all constant values and generate two programs in their respective target languages (Step 1): one program for computation (`computation.py`) and one program for retrieving the output (`display.py`). The evaluation keys are stored in the `computation/keys/` directory, whereas the `display/keys/` directory contains the secret key. The `computation/` directory, which holds the `computation.py` program, the encrypted inputs and the evaluation keys, can be outsourced to an untrusted party (Step 2). After homomorphically evaluating the program in Step 3, the untrusted party returns the obtained results to the trusted party (Step 4), who stores them in `display/results/` during Step 5. Finally, the trusted party, holding the secret key, retrieves the output by executing the `display.py` program in Step 6.

In the following sections, we present the framework's three main components.

4.1 Source Language

The source language $\mathcal{L}_{\mathsf{Src}}$ is a DSL that can be used to define vectors, permutations and their operations, namely permutation composition, applying permutations to vectors and computing the inversion number. Its syntax follows Python rules and is defined in Figure 4.2 using extended Backus-Naur form. The non-terminal symbol *int* denotes an element of $\mathbb{F}_p$ for $p = 2^{16}+1$.[5] The non-terminal *var* refers to an arbitrary Python identifier. Furthermore, we note that not all strings derived from the grammar represent valid programs (for instance, `[1, 2]([3, 4])` is an expression that is not supported by the interpreter).

The basic data types of $\mathcal{L}_{\mathsf{Src}}$ are integers, integer vectors and permutations, all of which can be printed and assigned to variables. A permutation $\sigma \in S_n$ is instantiated by specifying

[5] This choice is justified in Section 4.3.5.

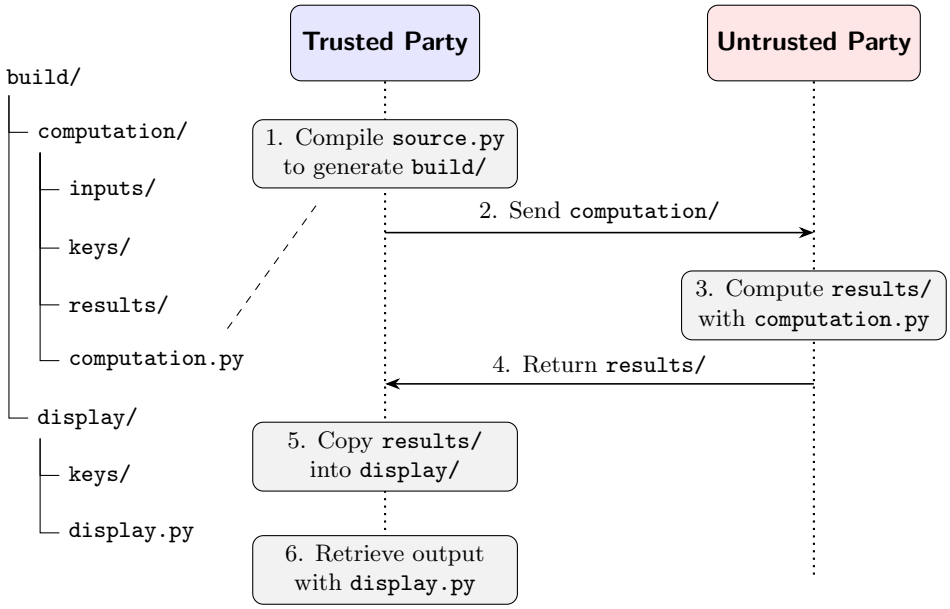


Figure 4.1: Sequence diagram illustrating the delegated execution

$$
\begin{array}{rcl}
\mathit{list} & ::= & \mathit{int} \mid \mathit{int}\texttt{, }\mathit{list} \\
\mathit{exp} & ::= & \texttt{[}\mathit{list}\texttt{]} \mid \texttt{Permutation([}\mathit{list}\texttt{])} \mid \mathit{var} \mid \texttt{(}\mathit{exp}\texttt{)} \\
 & & \mid \mathit{exp}\texttt{(}\mathit{exp}\texttt{)} \mid \mathit{exp}\texttt{ * }\mathit{exp} \mid \texttt{inv(}\mathit{exp}\texttt{)} \\
\mathit{stmt} & ::= & \mathit{var}\texttt{ = }\mathit{exp} \mid \texttt{print(}\mathit{exp}\texttt{)} \\
\mathcal{L}_{\mathsf{Src}} & ::= & \mathit{stmt}^*
\end{array}
$$

Figure 4.2: The concrete syntax of $\mathcal{L}_{\mathsf{Src}}$ in extended Backus-Naur form

the images $\sigma(1), \ldots, \sigma(n)$. For example, `Permutation([2, 1, 3])` creates a permutation in S_3 that maps the first element to position 2, the second to position 1 and the third to position 3. One can apply a permutation $\sigma \in S_n$ to a vector $v \in \mathbb{F}_p^n$ of matching size, which results in the permuted vector $\mathsf{Apply}(\sigma, v)$. Two permutations of the same size can also be composed with the `*` operator to obtain a new permutation of the same size. The inversion number of a permutation is computed by applying the `inv` function to it.[6]

As seen before, HE imposes restrictions that prevent the direct use of some classical programming paradigms. Most prominently, the control-flow of HE programs cannot be data-dependent because that would contradict the security properties [35]. To avoid such challenges, we restrict $\mathcal{L}_{\mathsf{Src}}$ to only include linear-flow programs.

We implemented an interpreter for the $\mathcal{L}_{\mathsf{Src}}$ language on top of Python by introducing a custom `Permutation` class along with an `inv` function. To support the syntax defined above, this class defines special methods for operator overloading, such as `__mul__` to concatenate permutations with the `*` operator. Hence, programs written in $\mathcal{L}_{\mathsf{Src}}$ are essentially Python programs, which can be executed by the Python interpreter, given that the `Permutation`

[6] A compiler could use the fact that all data in $\mathcal{L}_{\mathsf{Src}}$ is static to perform partial evaluation. However, since we aim to explore homomorphic evaluation, we do not consider this type of high-level optimization.

class and `inv` function are imported. Examples of $\mathcal{L}_{\mathsf{Src}}$ programs can be found in Figures 4.3 and 4.5.

$\mathcal{L}_{\mathsf{Src}}$

```
p = Permutation([1, 3, 2, 4, 5, 6])
q = Permutation([5, 3, 2, 6, 4, 1])

print(p * q)
q = q * q * q
print(inv(p * (q * p)))

v = q(p([100, -2, 0, 8, -5, 1]))
print(v)
```

Figure 4.3: Example of an $\mathcal{L}_{\mathsf{Src}}$ program

4.2 Target Languages

Because the compiler generates one program for computation and one program for decryption, we also define two separate target languages. Each of them provides a high-level interface for routines needed to work with the homomorphically encrypted data. Again, both languages are built on top of Python by implementing these routines as Python functions. For homomorphic operations, we use the OpenFHE library [3] and its implementation of the BFV scheme.

The $\mathcal{L}_{\mathsf{Comp}}$ language allows to express the outsourced program `computation.py`. Its interpreter implements the homomorphic matrix algorithms discussed in Section 2.5 and Chapter 3. Moreover, the language provides functions which deserialize evaluation keys and ciphertexts from files in the `keys/` and `inputs/` directories. In replacement of the `print` function from $\mathcal{L}_{\mathsf{Src}}$, we serialize ciphertexts to files in the `results/` directory for later decryption.

Correspondingly, the $\mathcal{L}_{\mathsf{Disp}}$ language is used to design the `display.py` program for retrieving the results. Any such program can deserialize the private key and ciphertexts from the `keys/` and `results/` directories, respectively. Subsequently, ciphertexts can be decrypted and printed based on the applied packing.

Example programs for $\mathcal{L}_{\mathsf{Comp}}$ and $\mathcal{L}_{\mathsf{Disp}}$ are shown in Figures 4.12 and 4.13, respectively. Their semantics are further clarified in the following section.

4.3 Compiler

Since the source and target languages follow standard Python syntax, we implement the compiler via the Python `ast` module, adopting the approach described in [34]. The compilation process is divided into the passes illustrated in Figure 4.4, each of which we now describe in detail.

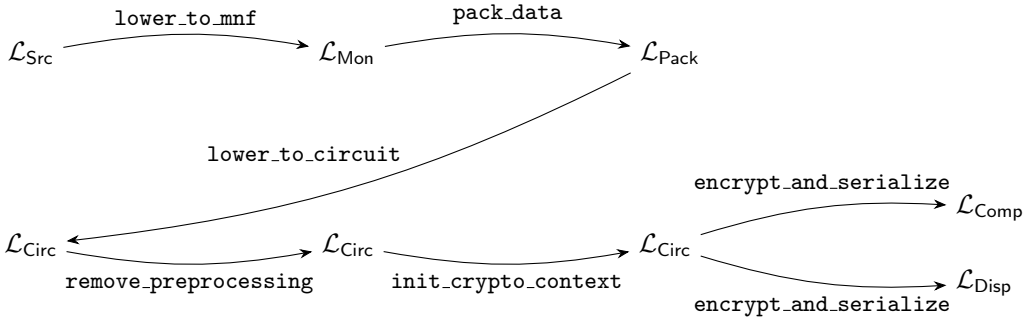


Figure 4.4: Diagram of the compiler passes and intermediate languages

4.3.1 Lower to Monadic Normal Form

For simplification of subsequent passes, the source program is first transformed into monadic normal form [13, 29]. The target language $\mathcal{L}_{\mathsf{Mon}}$ is very similar to $\mathcal{L}_{\mathsf{Src}}$ but only allows atomic subexpressions. This ensures that subexpressions are restricted to variables and constant values instead of complex expressions. We achieve this by assigning nested computations to temporary variables, as shown in Figure 4.5. In the resulting $\mathcal{L}_{\mathsf{Mon}}$ program, every statement only performs a single operation.

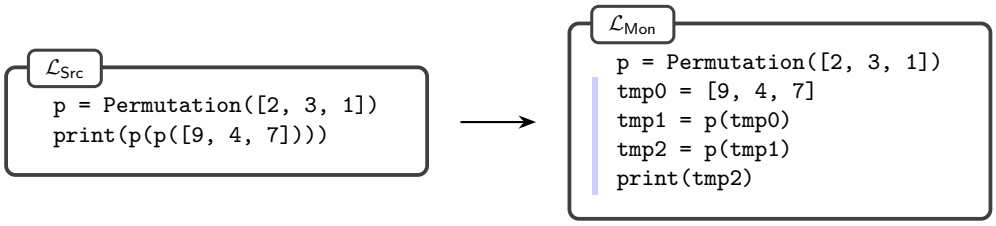


Figure 4.5: Compilation of an $\mathcal{L}_{\mathsf{Src}}$ program to $\mathcal{L}_{\mathsf{Mon}}$

4.3.2 Pack Data

As discussed in Section 2.5, choosing a layout for packing data in arithmetic HE is a critical decision. Hence, this pass determines a preferred layout for all vectors and permutations before packing them accordingly.

The layout is chosen by comparing the estimated program runtimes for the diagonal and row-major options. To estimate the runtime, the compiler determines the expected multiplicative depth while counting the total number of required basic operations (Add, Mult, CMult and Rot; cf. Chapter 2) for the respective option. At this point, the multiplicative depth does not account for potential optimizations described in Section 4.3.4. The final cost is then computed by summing the average runtimes of all basic operations for the given depth, which are presented in Figure 4.6. A higher multiplicative depth increases the runtime since it leads to a larger ring dimension as well as larger moduli (cf. Section 4.3.5). Moreover, rotations and multiplications are significantly more expensive than additions and constant multiplications because they require a costly key switching operation [19].

In the pass's target language $\mathcal{L}_{\mathsf{Pack}}$, constant vectors and permutations are replaced by

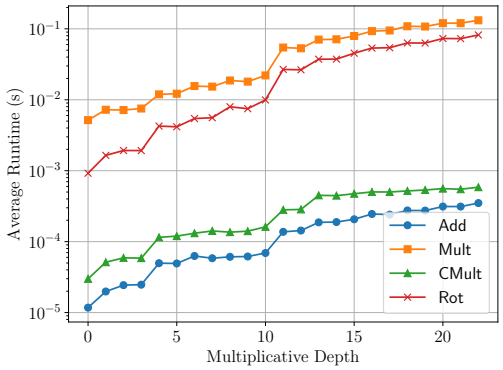


Figure 4.6: Runtimes of basic BFV operations in OpenFHE, averaged over 1000 iterations for each operation and depth. The plaintext modulus is $p = 2^{16} + 1$. See Section 5.1 for setup details.

tuples that contain an instance of the `Packing` class and the packed data (as one integer vector or as an array of integer vectors, depending on the layout). As illustrated in Figure 4.7, the `Packing` class is used to store metadata such as the original data type, the chosen layout, the original size and the padded size.

$\mathcal{L}_{\mathsf{Pack}}$

```
p = (Packing(data_type=DataType.PERMUTATION, layout=Layout.ROW, size=3,
    padded_size=4), [0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0])
tmp0 = (Packing(data_type=DataType.LIST, layout=Layout.ROW, size=3,
    padded_size=4), [9, 0, 0, 0, 4, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0])
tmp1 = p(tmp0)
tmp2 = p(tmp1)
print(tmp2)
```

Figure 4.7: The compiled $\mathcal{L}_{\mathsf{Pack}}$ version of the $\mathcal{L}_{\mathsf{Mon}}$ program from Figure 4.5. The packing is fixed to row-major (which may not be optimal in practice) for illustrative purposes.

4.3.3 Lower to Circuit

After packing the data, all operations are mapped to the appropriate homomorphic routines. Thus, the program fundamentally becomes a homomorphic circuit, expressed in $\mathcal{L}_{\mathsf{Circ}}$. This language distinguishes itself from $\mathcal{L}_{\mathsf{Pack}}$ by using function calls to homomorphic routines for composition, application and inversion counting (see Figure 4.8 for an example program). Applications and compositions are split into preprocessing steps and final computation, while for inversion counting, an additional call to homomorphically repack the data may be inserted if necessary.

To find the proper function calls, the compiler must keep track of how the data has been packed. Therefore, it builds a dictionary that maps each variable to the `Packing` instance associated with its value.

$\mathcal{L}_{\text{Circ}}$

```
p = (Packing(data_type=DataType.PERMUTATION, layout=Layout.ROW, size=3,
    padded_size=4), [0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0])
tmp0 = (Packing(data_type=DataType.LIST, layout=Layout.ROW, size=3,
    padded_size=4), [9, 0, 0, 0, 4, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0])
tmp3 = preprocess_left_matrix_row(cc, p, 4)
tmp4 = preprocess_right_matrix_row(cc, tmp0, 4)
tmp1 = matrix_matrix_mult_row(cc, tmp3, tmp4, 4)
tmp5 = preprocess_left_matrix_row(cc, p, 4)
tmp6 = preprocess_right_matrix_row(cc, tmp1, 4)
tmp2 = matrix_matrix_mult_row(cc, tmp5, tmp6, 4)
print(tmp2)
```

Figure 4.8: The compiled $\mathcal{L}_{\text{Circ}}$ version of the $\mathcal{L}_{\text{Pack}}$ program from Figure 4.7

4.3.4 Remove Preprocessing

The `lower_to_circuit` pass may introduce redundant preprocessing steps if a variable is used for multiple computations. For example, in Figure 4.8, the permutation stored in `p` is preprocessed identically for the assignments to `tmp3` and `tmp5`. To optimize the program's runtime, we can instead reuse the value stored in `tmp3`, as shown in Figure 4.9.

$\mathcal{L}_{\text{Circ}}$

```
p = (Packing(data_type=DataType.PERMUTATION, layout=Layout.ROW, size=3,
    padded_size=4), [0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0])
tmp0 = (Packing(data_type=DataType.LIST, layout=Layout.ROW, size=3,
    padded_size=4), [9, 0, 0, 0, 4, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0])
tmp3 = preprocess_left_matrix_row(cc, p, 4)
tmp4 = preprocess_right_matrix_row(cc, tmp0, 4)
tmp1 = matrix_matrix_mult_row(cc, tmp3, tmp4, 4)
tmp5 = tmp3
tmp6 = preprocess_right_matrix_row(cc, tmp1, 4)
tmp2 = matrix_matrix_mult_row(cc, tmp5, tmp6, 4)
print(tmp2)
```

Figure 4.9: The $\mathcal{L}_{\text{Circ}}$ program from Figure 4.8 without redundant preprocessing

Hence, this pass will replace all redundant preprocessing function calls with variables that already contain the previously computed result. This is achieved by building a dictionary that maps each variable and preprocessing function to a variable holding the previously computed result (provided such a variable exists). It should be noted that the dictionary entries become invalid if a variable is reassigned. In this case, the dictionary entries for the variable are either removed or, if available, inherited from the new value.

4.3.5 Initialize Crypto Context

This pass does not modify the $\mathcal{L}_{\text{Circ}}$ program. Instead, it analyzes the program to configure the OpenFHE `CryptoContext` together with its keys. Following this, the `CryptoContext` is serialized to the `computation/keys/` and `display/keys/` directories. The former also stores the serialized evaluation keys, while the latter holds the secret key.

To create a `CryptoContext` for BFV, one has to determine the multiplicative depth

and specify a plaintext modulus p. Based on these parameters, OpenFHE then computes a ciphertext modulus and the ring dimension N. A larger ciphertext modulus enables a higher multiplicative depth but reduces the security. Therefore, OpenFHE appropriately increases the ring dimension to maintain the targeted security level (by default, 128-bit security is used). However, as discussed in Section 2.3, the plaintext modulus p must be a prime number satisfying $p \equiv 1 \pmod{2N}$. So when the ring dimension N grows too large, it may become incompatible with the previously specified plaintext modulus. This relationship is illustrated in Figure 4.10 for $p = 2^{16} + 1$. To support a multiplicative depth greater than 22, the ring dimension is increased to $N = 2^{16}$, which results in $p \not\equiv 1 \pmod{2N}$.

As a solution, one could try to find a larger plaintext modulus, but the key sizes in these scenarios typically become impractical (see Figure 4.11, where $p = 6 \cdot 2^{17} + 1$ is the smallest modulus that supports a larger depth than $p = 2^{16} + 1$). Thus, we choose a fixed plaintext modulus of $p = 2^{16} + 1$, which is recommended for most applications [31] and supports a multiplicative depth of up to 22. Still, parameter selection for HE remains an active area of research, and a more detailed treatment can be found in [8].

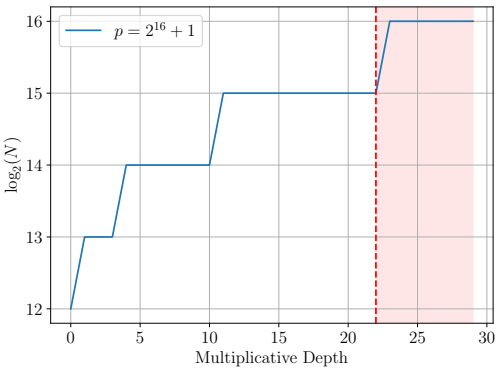


Figure 4.10: Ring dimension N chosen by OpenFHE for $p = 2^{16}+1$ and varying multiplicative depth. They become incompatible once $\log_2(N) > 15$.

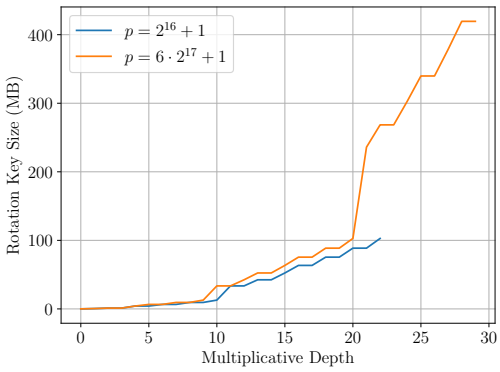


Figure 4.11: Size of a single rotation key in OpenFHE for different multiplicative depths and plaintext moduli. For $p = 2^{16} + 1$, the depth is capped at 22.

Since each index used by Rot requires its own evaluation key, this pass also collects all rotation indices occurring in the program to create the corresponding rotation keys.

4.3.6 Encrypt and Serialize

Finally, the $\mathcal{L}_{\mathsf{Circ}}$ program is split into an $\mathcal{L}_{\mathsf{Comp}}$ and an $\mathcal{L}_{\mathsf{Disp}}$ program. For this purpose, all constant data is encrypted and written to `computation/inputs/`.

The $\mathcal{L}_{\mathsf{Comp}}$ program, which will be evaluated by an untrusted party, deserializes the `CryptoContext`, its evaluation keys and the encrypted data from the respective directories in `computation/`. Based on these inputs, it computes the homomorphic circuit. Furthermore, this pass handles the program's output behavior: Since the untrusted party does not hold the secret key, it cannot evaluate `print` statements. Thus, whenever a `print` statement is encountered, the compiler replaces it with instructions such that the $\mathcal{L}_{\mathsf{Comp}}$ program instead

serializes the corresponding ciphertext to a file in the `computation/results/` directory. Figure 4.12 provides an example of the resulting program.

$\mathcal{L}_{\mathsf{Comp}}$

```
cc = deserialize_crypto_context('keys/crypto_context.enc', BINARY)
deserialize_mult_key('keys/key_mult.enc', BINARY, cc)
deserialize_rotation_key('keys/key_rotation.enc', BINARY, cc)
p = deserialize('inputs/input0.enc', BINARY)
tmp0 = deserialize('inputs/input1.enc', BINARY)
tmp3 = preprocess_left_matrix_row(cc, p, 4)
tmp4 = preprocess_right_matrix_row(cc, tmp0, 4)
tmp1 = matrix_matrix_mult_row(cc, tmp3, tmp4, 4)
tmp5 = tmp3
tmp6 = preprocess_right_matrix_row(cc, tmp1, 4)
tmp2 = matrix_matrix_mult_row(cc, tmp5, tmp6, 4)
serialize('results/output0.enc', tmp2, BINARY)
```

Figure 4.12: The compiled $\mathcal{L}_{\mathsf{Comp}}$ version of the $\mathcal{L}_{\mathsf{Circ}}$ program from Figure 4.9

Subsequently, the `computation/results` directory must be returned to the trusted party, who saves it to `display/results`. As illustrated in Figure 4.13, the $\mathcal{L}_{\mathsf{Disp}}$ program deserializes these results along with the `CryptoContext` and the secret key from their respective directories in `display/`. Following this, it decrypts the ciphertexts and prints the underlying data based on the stored packing.

$\mathcal{L}_{\mathsf{Disp}}$

```
cc = deserialize_crypto_context('keys/crypto_context.enc', BINARY)
sk = deserialize_private_key('keys/key_private.enc', BINARY)
print_ciphertext('results/output0.enc', Packing(data_type=DataType.LIST,
    layout=Layout.ROW, size=3, padded_size=4), BINARY, cc, sk)
```

Figure 4.13: The compiled $\mathcal{L}_{\mathsf{Disp}}$ version of the $\mathcal{L}_{\mathsf{Circ}}$ program from Figure 4.9

5 Evaluation

To evaluate our compiler, we conduct an empirical analysis that focuses on the runtime of the compiled programs.

5.1 Experimental Setup

The benchmarks were performed on a system equipped with an AMD Ryzen 7 7840HS processor (8 cores, max. 5.1 GHz, boost enabled) and 16 GB of RAM. The system runs Fedora 43 with Linux kernel 7.0. Programs were executed in Python 3.14 using version 1.5.0 of the OpenFHE Python wrapper [3].

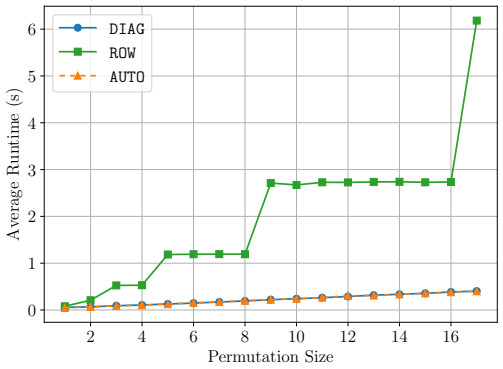
5.2 Packing Strategy

We start by analyzing how different packing strategies affect the performance. Their impact on computing applications, compositions and inversion numbers is illustrated in Figure 5.1, where `AUTO` refers to the packing automatically chosen by the compiler.

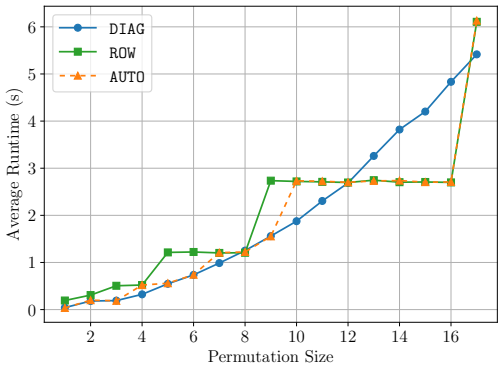
For applying a permutation to a vector, it is evident from Table 2.1 that diagonal packing is superior to row-major packing because it requires lower operational cost and reduces the multiplicative depth. This observation is supported by Figure 5.1(a), which demonstrates that with growing permutation size, diagonal packing increasingly outperforms row-major packing for computing two nested applications.

On the other hand, Figure 5.1(c) shows that row-major packing achieves a significantly lower runtime when computing the inversion number of a permutation. As expected, this performance difference is caused by the additional overhead of homomorphically converting diagonal packing to row-major packing, discussed in Section 3.2.

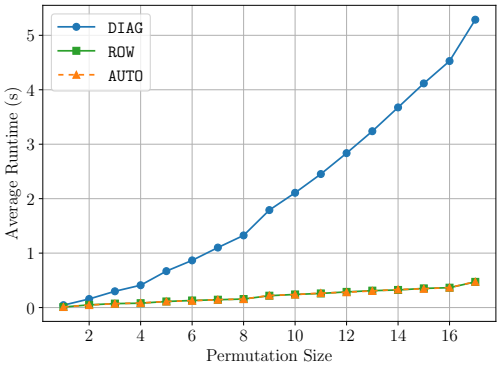
At the same time, row-major packing comes with the disadvantage that sizes must be padded to the next power of 2, which leads to sudden jumps in the runtime whenever a power of 2 is surpassed (see Figure 5.1 for permutation sizes of 2, 4, 8 and 16). However, when the size is close to the next power of 2, row-major packing can become preferable for permutation composition, as shown in Figure 5.1(b), since it uses fewer operations than diagonal packing in that case. Nonetheless, row-major packing doubles the multiplicative



(a) Application (`p(q(v))`)



(b) Composition (`p * q * r`)



(c) Inversion Number (`inv(p)`)

Figure 5.1: Runtime comparison between different packing strategies. The test programs compute the respective operations for permutations `p`, `q`, `r` and a vector `v` of varying size. The runtimes were averaged over 50 iterations.

depth compared to diagonal packing. Hence, when the nesting level (i.e., the number of nested compositions) increases, diagonal packing eventually dominates row-major packing, even when the permutation size is a power of 2. Particularly, when the multiplicative depth requires a larger ring dimension N, the runtime increases rapidly. This is illustrated in Figure 5.2 for a permutation size of 4, where the multiplicative depth exceeds 10 when going from nesting level 5 to 6, which in turn requires a larger ring dimension (see Figure 4.10).

Furthermore, Figures 5.1 and 5.2 show that the packing chosen by the compiler (`AUTO`) is usually close to optimal. However, in some cases the compiler favors row-major packing for permutation composition, even though diagonal packing would be slightly more efficient. We hypothesize that this is caused by not taking the number of distinct rotation indices into account when estimating the runtime. Figure 4.11 illustrates that rotation keys significantly impact memory consumption. Since row-major packing requires more rotation indices (see Figure 5.3), this limitation presumably leads to the compiler underestimating its runtime compared to diagonal packing. Future work could address this by counting the number of required rotation indices in the `pack_data` pass and incorporating their performance impact into the compiler's runtime estimation.

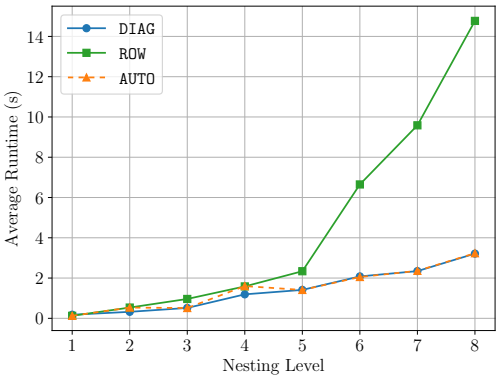


Figure 5.2: Runtime comparison between different packing strategies for permutation composition at varying nesting level. The permutations have a fixed size of 4. The runtimes were averaged over 10 iterations.

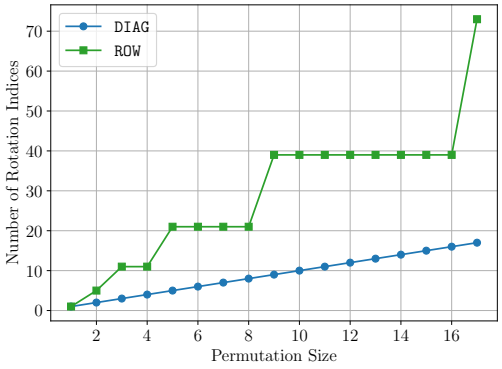


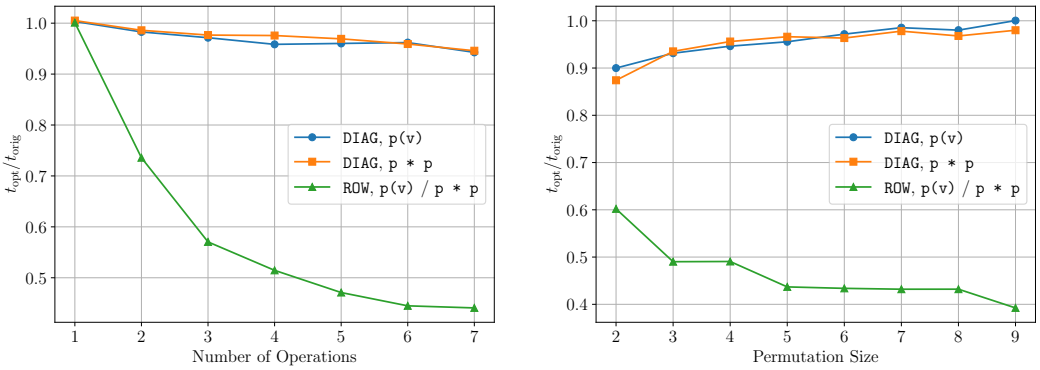
Figure 5.3: Number of distinct rotation indices required by different packings for composing permutations of varying size.

5.3 Preprocessing Optimization

In the `remove_preprocessing` pass, the compiler eliminates redundant preprocessing calls for both diagonal packing and row-major packing. As described in Section 2.5, preprocessing for matrix-vector multiplication under diagonal packing consists of replicating the vector once to support cyclic rotations. This is realized with one homomorphic addition and one rotation, which, in particular, only introduces a constant overhead. Hence, the performance gain of removing preprocessing calls is limited, as the primary computational effort lies in subsequent steps. This is confirmed by Figure 5.4(a), which shows that the achieved speedup when permuting the same vector of size 4 multiple times is less than 10%. Furthermore, as illustrated in Figure 5.4(b), this percentage decreases with larger permutation sizes, because the effort for subsequent steps grows linearly, while the effort for preprocessing steps remains constant.

Since the multiplication of two $n \times n$ matrices under diagonal packing essentially consists of performing n matrix-vector multiplications, the speedup for permutation composition behaves similarly.

In contrast, under row-major packing, a large proportion of the computational effort can be attributed to the preprocessing steps (in particular, all rotations and plaintext multiplications). Thus, eliminating redundant preprocessing calls can lead to a performance gain of more than 50%, as seen in Figure 5.4(a). Moreover, since the effort for preprocessing steps grows faster than the effort for subsequent steps ($\mathcal{O}(\tilde{n} \log_2 \tilde{n})$ rotations compared to $\mathcal{O}(\tilde{n})$ multiplications; see Table 2.1), its relative share scales with larger permutation sizes. Consequently, Figure 5.4(b) illustrates that the speedup increases as permutation sizes grow, demonstrating the effectiveness of the optimization for row-major packing.



(a) Fixed permutation size of 4

(b) Fixed operation count of 5

Figure 5.4: Ratio of optimized runtime t_{opt} (using `remove_preprocessing`) to unoptimized runtime t_{orig} for different packings and operations. The test programs compute `p(v)` or `p * p` a given number of times (not nested) for a permutation `p` and a vector `v` of the respective size. The runtimes were averaged over 250 iterations.

5.4 Discussion

Our runtime analysis confirms the significant performance impact of packings in the context of arithmetic HE. Figures 5.1 and 5.2 indicate that diagonal packing makes a good default choice for permutation application and composition, combining low multiplicative depth with no padding overhead. However, row-major packing may become preferable in more specific scenarios, such as inversion counting or when permutation sizes are close to the next power of 2. Moreover, as shown in Figure 5.4, row-major packing benefits substantially from the elimination of redundant preprocessing steps in cases where the same permutation is used multiple times.

While the compiler already improves performance significantly by incorporating these optimizations, it still has limitations. In particular, to determine the optimal packing choice in the `pack_data` pass, the runtime estimation could account for the number of required rotation indices as well as subsequent preprocessing optimizations. Future work could also explore how homomorphically repacking data between operations can reduce the overall runtime.

6
Conclusion

This thesis presents a functional framework for outsourcing the evaluation of permutations to an untrusted party. We successfully applied existing arithmetic HE algorithms to permute vectors and compose permutations without revealing either. We extended these procedures by an efficient algorithm that utilizes the vectorized message space to compute the inversion number of an encrypted permutation at a constant multiplicative depth of 3. To facilitate their practical deployment, we implemented a compiler that automatically transforms programs written in a user-friendly DSL into optimized homomorphic routines ready for outsourced computation. Our evaluation shows that the packing strategy of the compiler scales efficiently and already achieves runtime savings of several seconds for small permutation sizes. Moreover, the automatic elimination of redundant preprocessing steps reduces the execution time for row-major packing by more than 50% when the same permutation is used in multiple computations.

However, as discussed in Chapter 5, the accuracy of the compiler's runtime estimation can still be improved by taking the number of rotation indices and subsequent preprocessing optimizations into account. Also, due to the computational overhead of HE, the framework is only practically viable for small programs, as the multiplicative depth is capped at 22 and runtimes grow rapidly with increasing permutations sizes. As a solution, one could incorporate more advanced features from homomorphic tensor compilers [1, 16, 25], such as homomorphically repacking data between operations or employing bootstrapping to support arbitrary multiplicative depth. Moreover, because feasible permutation sizes are relatively small, many ciphertext slots frequently remain empty. These slots could instead be utilized to store intermediate results [2] or to pack multiple matrices into a single ciphertext [23]. Future work could also explore how advanced algorithms like baby-step/giant-step matrix-vector multiplication [21] or new packings like the bicyclic layout [10] can minimize runtime. Finally, the framework has potential applications in real-world scenarios such as privacy-preserving search engines or genome analysis, leveraging its capabilities for homomorphic sorting [33] and rank aggregation [26].

Bibliography

[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. *Proceedings on Privacy Enhancing Technologies*, 2023(1):325–342, January 2023. ISSN 2299-0984. doi: 10.56553/popets-2023-0020. URL http://dx.doi.org/10.56553/popets-2023-0020.

[2] Aikata Aikata and Sujoy Sinha Roy. Secure and efficient outsourced matrix multiplication with homomorphic encryption. In Sourav Mukhopadhyay and Pantelimon Stănică, editors, *Progress in Cryptology – INDOCRYPT 2024*, pages 51–74, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-031-80308-6.

[3] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. OpenFHE: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC'22, page 53–63, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450398770. doi: 10.1145/3560827.3563379. URL https://doi.org/10.1145/3560827.3563379.

[4] Asra Ali, Jaeho Choi, Bryant Gipson, Shruthi Gorantala, Jeremy Kun, Wouter Legiest, Lawrence Lim, Alexander Viand, Meron Zerihun Demissie, and Hongren Zheng. HEIR: A universal compiler for homomorphic encryption, 2025. URL https://arxiv.org/abs/2508.11095.

[5] Michael Artin. *Algebra*. Grundstudium Mathematik. Birkhäuser, 1998.

[6] Mikhail Babenko, Elena Golimblevskaia, Andrei Tchernykh, Egor Shiriaev, Tatiana Ermakova, Luis Bernardo Pulido-Gaytan, Georgii Valuev, Arutyun Avetisyan, and Lana A. Gagloeva. A comparative study of secure outsourced matrix multiplication based on homomorphic encryption. *Big Data and Cognitive Computing*, 7(2), 2023. ISSN 2504-2289. doi: 10.3390/bdcc7020084. URL https://www.mdpi.com/2504-2289/7/2/84.

[7] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. nGraph-HE2: A high-throughput framework for neural network inference on encrypted

data. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC'19, page 45–56, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368292. doi: 10.1145/3338469.3358944. URL https://doi.org/10.1145/3338469.3358944.

[8] Jean-Philippe Bossuat, Rosario Cammarota, Ilaria Chillotti, Benjamin R. Curtis, Wei Dai, Huijing Gong, Erin Hales, Duhyeong Kim, Bryan Kumara, Changmin Lee, Xianhui Lu, Carsten Maple, Alberto Pedrouzo-Ulloa, Rachel Player, Yuriy Polyakov, Luis Antonio Ruiz Lopez, Yongsoo Song, and Donggeon Yhee. Security guidelines for implementing homomorphic encryption. *IACR Communications in Cryptology*, 1(4), 2025. ISSN 3006-5496. doi: 10.62056/anxra69p1.

[9] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, page 309–325, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311151. doi: 10.1145/2090236.2090262. URL https://doi.org/10.1145/2090236.2090262.

[10] Jingwei Chen, Linhan Yang, Wenyuan Wu, Yang Liu, and Yong Feng. Homomorphic matrix operations under bicyclic encoding. *IEEE Transactions on Information Forensics and Security*, 20:1390–1404, 2025. doi: 10.1109/TIFS.2024.3490862.

[11] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 409–437, Cham, 2017. Springer International Publishing. ISBN 978-3-319-70694-8.

[12] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020. URL https://doi.org/10.1007/s00145-019-09319-x.

[13] Olivier Danvy. A new one-pass transformation into monadic normal form. *BRICS Report Series*, 9(52), 2002. doi: 10.7146/brics.v9i52.21767. URL https://tidsskrift.dk/brics/article/view/21767.

[14] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. EVA: an encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '20, page 546–561. ACM, June 2020. doi: 10.1145/3385412.3386023. URL http://dx.doi.org/10.1145/3385412.3386023.

[15] Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 617–640, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-46800-5.

[16] 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*, ASPLOS '25, page 734–749, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400710797. doi: 10.1145/3676641.3716008. URL https://doi.org/10.1145/3676641.3716008.

[17] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, 2012. URL https://eprint.iacr.org/2012/144.

[18] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. URL https://crypto.stanford.edu/craig.

[19] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pages 465–482, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-29011-4.

[20] Shai Halevi and Victor Shoup. Algorithms in HElib. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pages 554–571, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-44371-2.

[21] Shai Halevi and Victor Shoup. Faster homomorphic linear transformations in HElib. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 93–120, Cham, 2018. Springer International Publishing. ISBN 978-3-319-96884-1.

[22] Shai Halevi and Victor Shoup. Design and implementation of HElib: a homomorphic encryption library. Cryptology ePrint Archive, Paper 2020/1481, 2020. URL https://eprint.iacr.org/2020/1481.

[23] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1209–1222, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243837. URL https://doi.org/10.1145/3243734.3243837.

[24] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1998.

[25] Aleksandar Krastev, Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. A tensor compiler with automatic data packing for simple and efficient fully homomorphic encryption. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024. doi: 10.1145/3656382. URL https://doi.org/10.1145/3656382.

[26] Shili Lin. Rank aggregation methods. *WIREs Computational Statistics*, 2(5):555–570, 2010. doi: https://doi.org/10.1002/wics.111. URL https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/wics.111.

[27] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 1–23, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-13190-5.

[28] Chiara Marcolla, Victor Sucasas, Marc Manzano, Riccardo Bassoli, Frank H. P. Fitzek, and Najwa Aaraj. Survey on fully homomorphic encryption, theory, and applications. *Proceedings of the IEEE*, 110(10):1572–1609, 2022. doi: 10.1109/JPROC.2022.3205665.

[29] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93 (1):55–92, 1991. ISSN 0890-5401. doi: https://doi.org/10.1016/0890-5401(91)90052-4. URL https://www.sciencedirect.com/science/article/pii/0890540191900524. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[30] Jungho Moon, Dongwoo Yoo, Xiaoqian Jiang, and Miran Kim. THOR: Secure transformer inference with homomorphic encryption. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, CCS '25, page 3765–3779, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400715259. doi: 10.1145/3719027.3765150. URL https://doi.org/10.1145/3719027.3765150.

[31] Yuriy Polyakov. Homomorphic encryption for OpenFHE users - tutorial with applications part 3 – integer arithmetic. OpenFHE Webinar, 2020. URL https://openfhe.org/portfolio-item/homomorphic-encryption-for-palisade-users-3/. Accessed: 2026-06-28.

[32] Panagiotis Rizomiliotis and Aikaterini Triakosia. On matrix multiplication with homomorphic encryption. In *Proceedings of the 2022 on Cloud Computing Security Workshop*, CCSW'22, page 53–61, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450398756. doi: 10.1145/3560810.3564267. URL https://doi.org/10.1145/3560810.3564267.

[33] Lorenzo Rovida, Alberto Leporati, and Simone Basile. Lightweight sorting in approximate homomorphic encryption. *IACR Communications in Cryptology*, 3(1), 2026. ISSN 3006-5496. doi: 10.62056/a3ksdkmol.

[34] Jeremy G. Siek. *Essentials of Compilation: An Incremental Approach in Python.* MIT Press, 2023.

[35] Alexander Viand. *Usable Fully Homomorphic Encryption.* PhD thesis, ETH Zurich, 2023. URL https://doi.org/https://doi.org/10.3929/ethz-b-000613734.

[36] Jelle Vos, Daniël Vos, and Zekeriya Erkin. Efficient circuits for permuting and mapping packed values across leveled homomorphic ciphertexts. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian D. Jensen, and Weizhi Meng, editors, *Computer Security – ESORICS 2022*, pages 408–423, Cham, 2022. Springer International Publishing.