# Authority-Aware FHE Program Partitioning Compiler for Edge-Cloud Systems

**Dongkwan Kim**

**Department of Electrical and Electronic Engineering**

**Graduate School**

**Yonsei University**

# Authority-Aware FHE Program Partitioning Compiler for Edge-Cloud Systems

**Advisor : Prof. Hanjun Kim**

**A Dissertation Submitted**
**to the Department of Electrical and Electronic Engineering**
**and the Committee of the Graduate School**
**of Yonsei University in Partial Fulfillment of the**
**Requirements for the Degree of**
**Doctor of Philosophy in Electrical and Electronic Engineering**

**Dongkwan Kim**

**August 2026**

Authority-Aware FHE Program Partitioning Compiler for Edge-Cloud Systems

This Certifies that the Dissertation of KIM,DONGKWAN is Approved

Committee Chair KIM HANJUN
2026-07-08 21:04:37

Committee Member 이용우
2026-07-06 15:13:53

Committee Member Ro Won Woo
2026-07-07 13:17:32

Committee Member Song William Jinho
2026-07-02 09:53:50

Committee Member 이동윤
2026-07-02 23:59:53

Department of ELECTRICAL AND ELECTRONIC ENGINEERING
Graduate School
Yonsei University
August 2026

# 감사의 글

어느새 이런 날이 왔습니다. 이 학위논문을 완성하기까지 정말 많은 분들의 도움을 받았습니다. 이 글을 빌려 지난 시간 동안 저를 도와주시고 응원해주신 모든 분들께 진심으로 감사의 말씀을 드리고 싶습니다.

학부생 시절부터 박사과정을 마무리하는 지금까지 많은 가르침을 주신 김한준 교수님께 진심으로 감사드립니다. 처음 연구실에 들어왔을 때를 생각해보면, 저는 컴파일러가 무엇인지도 잘 몰랐고, 논문 한 편을 읽는 데도 한참이 걸리던 학생이었습니다. 그런 저에게 교수님께서는 항상 새로운 길을 보여주시고, 쉽게 해보지 못했을 많은 경험과 기회를 주셨습니다. 아직도 부족한 점이 많지만, 앞으로도 교수님께 배운 것들을 마음 깊이 새기며 좋은 연구자가 될 수 있도록 노력하겠습니다. 동형암호 연구를 하면서 오랜 시간 많은 도움을 주신 이동윤 교수님께도 깊이 감사드립니다. 매주 미팅에 참여하며 정말 많은 것을 배웠고, 연구가 잘 풀리지 않을 때마다 문제를 다시 바라볼 수 있도록 큰 도움을 받았습니다. 여러 프로젝트를 함께 진행하면서 주신 조언과 도움이 큰 힘이 되었습니다. 진심으로 감사드립니다. 학부 때부터 많은 가르침을 주시고, 이번 학위논문 심사도 맡아주신 노원우 교수님과 송진호 교수님께도 감사의 말씀을 드리고 싶습니다. 노원우 교수님의 컴퓨터 구조 수업은 제가 컴퓨터 시스템 분야에 관심을 갖게 된 중요한 계기였습니다. 수업을 들으면서 처음으로 나는 컴퓨터 쪽을 해야 하나보다라고 생각했던 기억이 납니다. 송진호 교수님의 운영체제와 고급 프로그래밍 수업을 통해서 처음으로 큰 프로그램을 어떻게 작성하고, 어떻게 디버깅하는지를 배웠습니다. 지금도 종종 구현을 하다가 예전 수업자료나 과제를 다시 찾아보는 경우가 있습니다. 연구실 선배이자 동형암호 연구를 함께 이끌어주셨고, 이번 학위논문 심사위원으로도 많은 도움을 주신 이용우 교수님께도 진심으로 감사드립니다. 연구뿐만 아니라, 대학원 생활 전체에서 정말 많은 도움을 받았습니다. 주신 조언들이 큰 힘이 되었고 함께 연구했던 시간들이 매우 소중한 기억입니다. 바쁘신 와중에도

귀한 시간을 내어 제 학위논문을 심사해주신 모든 심사위원 교수님들께 다시 한 번 감사드립니다. 좋은 피드백과 조언을 주신 덕분에 논문을 더 나은 방향으로 마무리할 수 있었습니다. 주신 말씀들을 잊지 않고, 앞으로도 계속 발전하는 연구자가 되도록 노력하겠습니다.

컴파일러 최적화 연구실에서 함께한 선후배님들께도 감사의 마음을 전합니다. 대학원 생활을 행복하게 지낼 수 있었던 것은 결국 좋은 사람들과 함께했기 때문이라 생각합니다. 이경민 박사님, 김봉준 박사님, 허선영 교수님, 김창수 박사님, 조성준 박사님, 송승빈 박사님, 정신녕 박사님, 이재호 박사님, 최희림 연구원님, 김근우 연구원님, 천선영 박사님, 윤성우 연구원님, 박현준 연구원님, 이주민 연구원님, 권현호 연구원님, 염호윤 연구원님, 이찬 연구원님, 정건모 연구원님, 정해은 연구원님, 김성호 연구원님, 김재민 연구원님께 감사드립니다. 함께 보낸 많은 시간들이 제게 큰 의미로 남아 있습니다. 늘 세심하게 챙겨주신 남주현 선생님께도 깊은 감사드립니다. 한 분 한 분에 대해 모두 쓰고 싶은데, 그러면 너무 길어질 것 같아요.

가족들에게도 감사의 마음을 전합니다. 항상 저를 믿어주시고 응원해주신 부모님께 진심으로 감사드립니다. 부모님의 사랑과 지지가 있었기에 박사과정 동안 좋은 시간들을 많이 보낼 수 있었고, 이렇게 의미 있는 마무리를 할 수 있었습니다. 형에게도 고맙다는 말을 전하고 싶습니다. 항상 제 편이 되어주고 필요할 때마다 도와주어 고맙습니다. 이모들, 이모부들, 사촌형, 누나, 동생, 장인장모님을 비롯한 가족들께도 감사드립니다. 가족들의 응원과 관심이 제게 큰 힘이 되었습니다. 언제나 감사하고 사랑하고 있습니다.

사랑하는 호노라에게 정말 고맙다는 말을 전하고 싶습니다. 모든 시간을 함께 보내며 제 곁에서 가장 가까이 응원해주고, 가장 큰 위로가 되어주었습니다. 늘 제 편이 되어주어서 고맙습니다. 앞으로의 새로운 여정도 함께할 수 있음에 감사하고, 더 좋은 사람이 될 수 있도록 노력하겠습니다. 그리고 우리집 야옹이 찌미. 간식으로 감사를 대신하도록 하겠습니다.

혼자 걸어온 길이 아니었습니다. 사랑하는 스승님들, 동료들, 가족들이 있었기에 여기까지 올 수 있었습니다. 모든 분들께 진심으로 감사드립니다.

# TABLE OF CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

## LIST OF ALGORITHMS

**ABSTRACT**

**Authority-Aware FHE Program Partitioning Compiler**

**for Edge-Cloud Systems**

Privacy-preserving edge-cloud tensor execution is difficult to develop without exposing intermediate values to unauthorized devices because user data and service-side model parameters often belong to different privacy authorities. Fully homomorphic encryption enables computation on encrypted data and allows a cloud server to process data that belong to the edge. Recently proposed FHE compilers support for deciding where encrypted execution is needed, translating tensor operators into ciphertext-level execution, and generating coordinated edge and cloud programs. Fully homomorphic encryption enables computation on encrypted data and allows a cloud server to process edge-owned data without accessing it in plaintext. However, they mainly focus on fully encrypted execution, while existing edge-cloud partitioning methods do not directly handle authority conflicts between private data owned by different parties.

This dissertation presents AION, an end-to-end authority-aware compiler for privacy-preserving edge-cloud tensor execution. AION allows developers to express tensor computation together with privacy intent through source-level placement annotations and authority annotations. From the annotated program, AION analyzes how privacy authority propagates through the program and identifies the values that must be protected during edge-cloud execution. AION lowers the protected tensor computation into

FHE primitive operations using tensor-to-slot mapping information, and selects an edge-cloud partitioning plan by estimating encrypted computation and ciphertext communication costs. Finally, AION generates coordinated edge and cloud programs with the required encryption, decryption, and communication operations.

The evaluation shows that AION improves the efficiency and programmability of privacy-preserving edge-cloud tensor execution. AION achieves 4.92× and 3.69× geometric mean speedups over EVA and Hecate, respectively, while reducing edge and cloud memory usage by up to 65.5% and 59.1%. AION also reduces the burden of developers by replacing manual encrypted-region selection, communication insertion, encryption and decryption management, and tensor-to-FHE primitive rewriting with source-level annotations and compiler-generated edge-cloud programs.

Keywords: Homomorphic Encryption, Edge-Cloud Computing, Program Partitioning, Type Systems

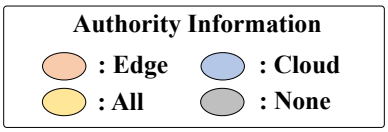
# 1. Introduction

Edge-cloud computing provides a practical execution model in which edge devices collect local data and cloud servers process the collected data to provide computational services. Mobile devices, embedded systems, IoT sensors, and wearable platforms continuously observe the physical world near users, while cloud servers provide the computational resources required for machine learning inference, signal processing, and large-scale analytics. By splitting execution between local devices and remote servers, applications can use powerful cloud-side resources without placing the entire computational burden on constrained edge devices.

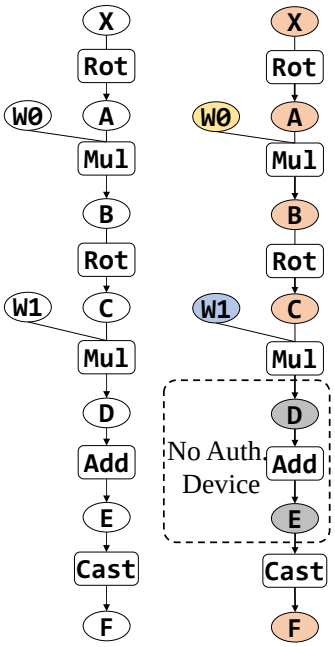
Privacy-preserving edge–cloud execution is difficult to develop without exposing intermediate values to unauthorized devices because the data used by an application may belong to different authorities. A healthcare application may process physiological signals owned by the user with a service-provider-owned diagnosis model. A smart-home application may analyze private sensor traces using proprietary cloud-side parameters. An industrial monitoring application may combine device-side measurements with cloud-side models or rules that should not be disclosed to the device. These applications need collaboration between the edge and the cloud, while each side must avoid exposing its private data to the other side in plaintext.

```
# X is for Edge
# W0 is for All
# W1 is for Cloud
func rotNmul(X, W0, W1) :
 For i in [0,3):
  A[i] = Rotate(X, i)
  B[i] = A[i]*W0[i]
 E = 0
 For i in [0,3):
  For j in [0,2):
   C[i][j] = Rotate(B[i],j)
   D[i][j] = C[i][j]*W1[j]
   E = E + D[i][j]
 F = Cast(E)
 return F
```

(a) Program




(b) Legend




(c) DFG (d) Tainted DFG

Figure 1.1: Example program and its data flow and tainted data flow graphs. If the opposite authorities of the edge and the cloud taint the same data such as D, none of the edge and the cloud can access the tainted data.

## 1.1 Privacy-Preserving Edge-Cloud Partitioning

Conventional edge–cloud partitioning [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] does not directly address privacy authority constraints. Existing partitioning methods usually choose where to execute each operation based on latency, bandwidth, energy consumption, or device capability. They are effective when intermediate values can move between devices as ordinary plaintext data. Their assumption becomes invalid when an intermediate value is derived from private data owned by different parties, because the value may no longer be readable by either device in plaintext.

Ownership-based partitioning [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21] provides a privacy-aware model for edge–cloud execution by assigning data values to authorized devices. Prior schemes allow developers to specify which device is permitted to access each sensitive value, and then partition the program according to these ownership annotations. In this model, privacy is expressed as a property of data rather than only as a property of code regions. The compiler can therefore reason about whether a value may be placed on the edge, placed on the cloud, or shared by both devices in plaintext.

Figure 1.1 shows how ownership-based partitioning propagates authority through a program. In the example program in Figure 1.1a, the input `X` is annotated as `Edge`, the weight `W0` is annotated as `All`, and the weight `W1` is annotated as `Cloud`. From these annotations, the compiler constructs the data-flow graph shown in Figure 1.1d and infers the authority of derived values by following data dependencies. For each operation, the authority of the result is determined from the authorities of its operands, so the compiler can track how privacy constraints flow through intermediate values.

The authority of a derived value is obtained from the common plaintext authority of its contributing operands. For example, the value `A` is derived from `X`, so it remains readable by the edge. The value `B` is computed from `A` and `W0`, where `A` has `Edge` authority and `W0` has `All` authority. Their common authorized device is the edge, so `B` receives `Edge` authority. This propagation rule prevents a value from being assigned to a device unless that device is authorized to read all plaintext data that contribute to the value.

Ownership-based partitioning exposes an execution gap when a value has no common plaintext-authorized device. In Figure 1.1d, the authorized device of `D` becomes `None` because its incoming edges come from `W1` with authority `Cloud` and `C` with au-

thority Edge, and the intersection of these authorities is empty. To continue execution in a plaintext-only partitioning model, the developer may have to relax the privacy constraint by explicitly allowing one device to access the other party's private data through a `Cast` operation. Such a workaround weakens the privacy guarantee. Ownership-based partitioning can identify privacy conflicts, but it cannot by itself support the joint computation of edge-private and cloud-private data without sacrificing privacy.

Several privacy-preserving mechanisms can be used to support edge-cloud execution. Trusted Execution Environment (TEE) places sensitive computation inside isolated hardware regions [15, 16]. This approach protects selected code and data under a trusted hardware environment. Differential Privacy (DP) protects information released through computation results [22, 23, 24, 25, 26]. It is effective for limiting information leakage from released outputs or model updates, and is commonly used with settings such as federated learning. Secure Multi-Party Computation (MPC) supports joint computation among parties through secure protocols [27, 28, 29]. It can compute over edge-private and cloud-private inputs without revealing them in plaintext, but its protocol execution requires message exchanges between parties while the computation is running. In an edge-cloud setting, these exchanges make communication latency a central execution cost. This dissertation focuses on collaborative edge-cloud execution, where the compiler selects the protected region, generates the encrypted computation for that region, and inserts the required communication between the edge and the cloud.

Fully Homomorphic Encryption (FHE) [30, 31, 32, 33, 34, 35] provides the encrypted execution mechanism. If a value has no plaintext-authorized device, the value can be represented as a ciphertext and processed without exposing its contents. This capability

allows AION to preserve the collaborative edge-cloud execution model while generating protected computation for authority-conflicting values. The compiler must determine which values can remain plaintext, which values must become ciphertexts, where encryption and communication should be inserted, and where decryption is required. This dissertation uses FHE because authority-conflicting values must remain computable even when neither the edge nor the cloud is allowed to read them in plaintext.

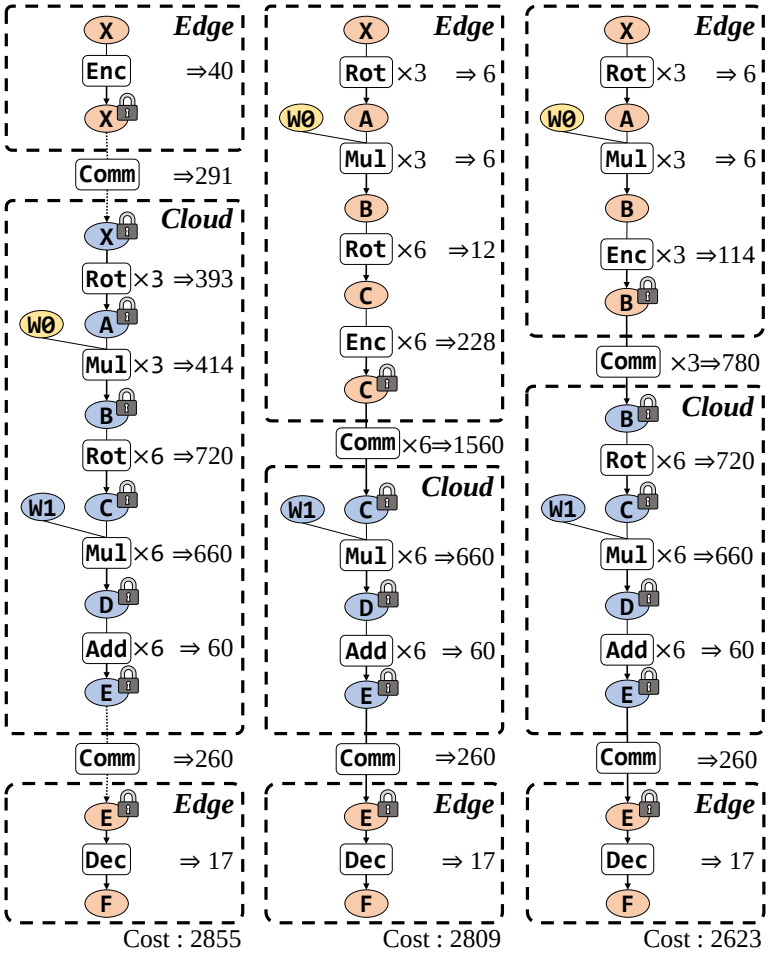
## 1.2 Encrypted Tensor Partitioning Challenges and Solutions

This dissertation targets privacy-preserving tensor execution across the edge and the cloud when intermediate values may lose all plaintext-authorized devices. FHE-enabled edge-cloud execution requires more than applying encryption to privacy-conflicting values. FHE provides the cryptographic mechanism for computing on protected data, but the compiler must still decide where encrypted execution should be introduced, how much computation should remain in plaintext, and how the encrypted region should be translated into executable FHE operations. In this dissertation, these requirements arise from two connected challenges.

**Challenge A: Selecting encrypted regions under authority constraints.** The first challenge is selecting where encrypted execution should begin and end under privacy authority constraints. A value with no plaintext-authorized device cannot be exposed to either the edge or the cloud in plaintext. The compiler must therefore introduce encrypted representation before such a value is processed by an unauthorized device. At the same time, values that can legally remain on the edge should not be forced into FHE,

| | Edge | | Comm. | Cloud | | |
|---|---|---|---|---|---|---|
| | Enc | Dec | | Add | Mul | Rot |
| **Plain** | - | - | 5 | 1 | 1 | 1 |
| **Lv.1** | 38 | 17 | 260 | 10 | 110 | 120 |
| **Lv.2** | 40 | 19 | 291 | 12 | 138 | 131 |

(a) Profiled latency for different level and FHE operations




(b) All-in-cloud (c) Minimal-in-cloud (d) Cost-aware (AION)

Figure 1.2: FHE-enabled program partitioning with different partitioning policies. The all-in-cloud policy assigns all the operations to the cloud, and the minimal-in-cloud policy assigns only required operations to the cloud. The cost-aware policy that this work proposes partitions the program with the minimal estimated latency.

because plaintext edge execution is much cheaper than homomorphic cloud execution.

Figure 1.2 shows FHE-enabled partitioned programs and their estimated costs for the same example in Figure 1.1a under different partitioning policies. Unlike ownership-based program partitioning in Figure 1.1d, FHE-enabled partitioning can make authority-conflicting values executable by representing them as ciphertexts. However, the point at which encryption is introduced has a large impact on end-to-end latency because both FHE operations and ciphertext communication are expensive.

The all-in-cloud policy in Figure 1.2b preserves privacy by executing the full program on the cloud using FHE. This policy avoids unauthorized plaintext exposure, but it also applies homomorphic computation to values that could have been processed on the trusted edge device. As a result, the estimated latency is high due to excessive use of encrypted operations. A more selective policy is shown in Figure 1.2c. This policy minimizes the amount of cloud-side FHE computation by delaying encryption until it is required by authority constraints. It reduces the computation cost compared with all-in-cloud execution, but it does not necessarily minimize the total program latency.

The limitation comes from ciphertext communication between edge and cloud. Encrypting a value increases its size, and the level of a ciphertext affects both operation latency and communication size. Therefore, a partitioning policy that minimizes only the number of FHE operations may still incur a large communication cost if it transfers many ciphertexts or transfers them at high levels. Efficient FHE-enabled partitioning must evaluate encrypted computation and ciphertext communication together.

Existing FHE compilers automate important parts of encrypted program generation, including ciphertext parameter management and scale management [36, 37, 38, 39, 40,

41, 42, 43, 44, 45, 46, 47, 48]. However, these compilers typically translate the given computation into an FHE program without distinguishing between values that require encrypted execution and values that can remain in plaintext on the edge. They are therefore not designed to choose an authority-aware edge-cloud partition boundary. In addition, manually extracting the FHE-required segment from a program is time-consuming and error-prone. The manual process requires partitioning the program into edge and cloud sub-programs, inserting encryption, decryption, and communication operations, and configuring FHE parameters for the cloud-side sub-program with in-depth scheme knowledge. Annotation-guided code partitioning schemes [49, 50, 51, 52, 53] can drive program offloading, but they do not understand FHE semantics and therefore cannot generate an FHE-enabled cloud sub-program.

**Solution A: Authority-Aware Cost-guided Partition Selection** AION addresses this challenge through cost-guided partition selection. Figure 1.2d illustrates a partitioned program selected by considering both computation and communication cost. Although this plan executes more operations homomorphically than the minimal-in-cloud plan, it reduces the communication cost more significantly and achieves a lower total estimated latency. AION therefore treats partitioning as a compiler search over legal encrypted-region candidates. Each candidate must satisfy the authority constraints, and the compiler selects the candidate with the lowest estimated end-to-end cost based on edge plaintext computation, cloud-side FHE computation, and ciphertext communication.

**Challenge B: Lowering tensor computation into encrypted execution.** The second challenge is making the selected encrypted region executable when the source program is written with tensor abstractions. Edge-cloud workloads targeted in this dissertation are naturally expressed using tensor operators such as convolution, matrix multiplication, pooling, normalization, and element-wise operations. FHE libraries do not execute these tensor operators directly. They provide ciphertext containers and primitive operations such as rotation, multiplication, rescaling, encryption, and decryption [54, 55, 56, 57]. The two levels of abstraction are mismatched, and bridging them correctly requires FHE-specific expertise that most application developers do not have.

Bridging this gap requires a non-trivial realization of tensor semantics at the ciphertext level. The compiler must decide how logical tensor values are represented inside ciphertext packs. The compiler must map tensor coordinates to ciphertext identifiers and slot identifiers, determine whether a tensor fits within one ciphertext or must be partitioned across multiple ciphertexts, and preserve or transform layouts across neighboring operators. These decisions determine the number of ciphertexts, the rotation structure, the reduction pattern, and the edge-cloud communication volume.

Prior encrypted tensor systems also show that tensor-to-FHE translation is a substantial compiler problem in its own right. Lee et al. [58] demonstrate that efficient encrypted tensor execution depends critically on packing strategy. Fhelipe [59] explores automatic layout selection and transformation across encrypted tensor computations. Qiwu [60] improves ciphertext utilization by analyzing valid and invalid slot regions, and Orion [61] provides a user-friendly frontend for encrypted neural-network inference. More recently, HEIR [62] shows that homomorphic compilation can be organized

around MLIR and structured tensor abstractions. These works significantly advance encrypted tensor compilation, but practical edge-cloud deployment still requires connecting tensor lowering with authority-aware partitioning. Much of the tensor-to-FHE gap is still bridged through workload-specific packing scripts, backend wrappers, or handcrafted encrypted kernels. Such implementations are often specialized to a particular model architecture, ciphertext layout, or backend library. As a result, the compiler has limited visibility into how tensor layout choices affect FHE-level cost and communication cost across the selected partition boundary.

**Solution B: Compiler-visible encrypted tensor realization.** This challenge can be addressed by treating tensor-to-FHE translation as a compiler lowering problem rather than as backend-specific manual engineering. The compiler should not leave the selected encrypted region as a high-level tensor fragment that developers must rewrite by hand. Instead, it should preserve tensor semantics in an intermediate representation, use that structure to determine how tensor values are represented as ciphertext packs, and lower supported tensor operators into executable FHE primitive operations. Such a lowering stage must make the encrypted representation visible to the compiler. It should expose how tensor elements are mapped to ciphertext slots, how many ciphertexts are needed to represent each value, which layouts are preserved across operators, and where layout conversion is required. With this information, the compiler can generate the encrypted computation for the selected region and also estimate its cost in terms of ciphertext count, rotations, multiplications, and communication size.

This approach connects encrypted tensor execution with edge-cloud partitioning.

Once tensor operations are lowered into ciphertext-level execution plans, the partitioning stage can reason about the actual cost of each encrypted-region candidate rather than treating the cloud-side tensor computation as an opaque block. Thus, a practical solution requires a compiler-visible lowering path from tensor semantics to encrypted execution, so that the selected FHE region is both executable and cost-aware.

## 1.3 Dissertation Contributions

This dissertation presents AION, an end-to-end compiler for privacy-preserving edge-cloud execution. Given a tensor program and a lightweight expression of privacy intent, AION performs structured lowering from tensor semantics to encrypted execution, authority analysis, cost-aware partition optimization, and distributed code generation within one compilation flow, producing standalone edge and cloud programs together.

Developers can express privacy intent through explicit placement annotations that designate edge or cloud execution regions, or through data authority annotations that declare which device may access each sensitive value in plaintext. From the annotations, the compiler derives legal encrypted-region candidates without requiring developers to manually split the program.

To bridge the gap between tensor programs and FHE execution, AION preserves tensor semantics in a compiler-visible representation and lowers supported tensor operators into ciphertext-level execution plans. The lowering stage exposes how tensor values are represented as ciphertext packs, how many ciphertexts are required for each value, and how supported tensor operators are realized through FHE primitive operations.

AION also introduces a type system in which every value in the program carries a

type that records the set of devices authorized to access it and its current encryption status. Forward and backward propagation of these types through the program identifies which values must be encrypted at any legal partition boundary. The compiler then enumerates all legal partition candidates, estimates the end-to-end latency of each by modeling encrypted computation cost and ciphertext communication cost jointly, and selects the partition with the lowest estimated cost. In addition, AION inserts encrypt, decrypt, and communication operations at the selected boundary and emits separate edge and cloud programs together with key-deployment artifacts, transforming a single annotated source into a fully deployable distributed system.

This work evaluates AION on representative privacy-preserving edge-cloud tensor workloads. The AION compiler achieves 4.92× and 3.69× geometric mean speedups over EVA and Hecate, respectively, and reduces memory usage by up to 65.5% at the edge and 59.1% at the cloud. The evaluation also shows that AION reduces developer burden by replacing manual encrypted-region selection, communication insertion, encryption and decryption management, and tensor-to-FHE primitive rewriting with source-level annotations and compiler-generated edge-cloud programs. These results show that authority-aware partitioning and encrypted tensor lowering improve the efficiency and practicality of FHE-enabled edge-cloud execution.

This dissertation makes the following contributions.

- This work presents a tensor DSL for expressing tensor computation together with privacy intent. The interface supports explicit placement annotations for developers who want to guide execution placement and authority annotations that al-

low the compiler to reason about plaintext accessibility.

- This work designs a structured lowering pipeline that translates high-level tensor operators into coordinated ciphertext-level FHE execution plans. The compiler uses tensor-to-slot metadata to lower supported tensor operators into FHE primitive operations while preserving tensor structure for compiler analysis.

- This work introduces a type system that tracks, for each value, the set of devices authorized to access it together with its current encryption status. Forward and backward analyses over these types identify the values that must be protected by homomorphic encryption at each legal partition boundary.

- This work proposes an optimization framework that evaluates legal partition candidates by jointly modeling encrypted computation cost and ciphertext communication cost. Using this cost model, the compiler selects the partition point that minimizes the end-to-end latency of privacy-preserving execution.

- This work implements distributed code generation for FHE-enabled edge-cloud execution. The compiler inserts encryption, decryption, and communication operations at the selected boundary and emits coordinated edge and cloud programs with the artifacts required for deployment.

## 1.4 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 reviews prior work on edge-cloud partitioning, ownership-based privacy control, homomorphic-encryption

systems, and encrypted tensor compilation. Chapter 3 presents an overview of AION, including the compiler pipeline, runtime workflow, and threat model. Chapter 4 introduces the frontend programming model, describing the tensor DSL, privacy-specification interfaces, and structured tensor semantic representation. Chapter 5 presents encrypted tensor lowering and explains how structured tensor operators are lowered into FHE execution plans. Chapter 6 describes authority analysis, partition selection, code generation and security analysis. Chapter 7 explains the runtime and deployment flow for collaborative edge-cloud execution. Chapter 8 evaluates AION in terms of latency, memory usage, estimation accuracy, programmability, and case studies. Chapter 9 discusses broader deployment considerations, and Chapter 10 concludes this dissertation.

# 2. Related Work

This chapter reviews conventional edge-cloud partitioning and ownership-based partitioning, which motivate the authority-aware partitioning problem. It also reviews privacy-preserving cloud computing techniques, homomorphic-encryption systems in edge-cloud settings, and FHE compilation systems for encrypted tensor execution.

## 2.1 Edge-Cloud Program Partitioning without Privacy Conflict

Existing edge-cloud partitioning systems [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 63] aim to balance computational load between the edge and the cloud for conventional applications, especially deep neural network workloads. Neurosurgeon [1] is a pioneering work that identifies an optimal partitioning layer within a DNN model to minimize total latency or energy consumption. Wang et al. [7] and Li et al. [64] analyze DNN models to find optimal splitting points that reduce inference latency. JointDNN [63] introduces a collaborative execution framework between mobile devices and the cloud and shows that relying solely on either local or cloud execution is suboptimal in terms of inference time and energy consumption. The partial offloading approach [3] proposes a privacy-preserving computation-offloading method that obscures the original data captured by the client from the edge server. Although these approaches are effective for regular edge-cloud programs, they do not adequately account for privacy-preserving computation under

explicit data-authorization constraints. In contrast, AION incorporates privacy authority directly into the compilation process, enabling optimized edge-cloud partitioning while preserving data protection requirements.

## 2.2 Ownership-based Partitioning

Prior research has extensively studied ownership-based partitioning [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21] using annotations on sensitive data or code regions. Secure program partitioning [11] introduced the idea of manually annotating programs with security types. Privtrans [12] proposed automatic privilege separation by dividing a program into privileged and unprivileged components based on developer annotations, but it does not consider the integrity of private data. Programcutter [13] automatically partitions monolithic programs, although users must manually insert global variables into the partitioned result. Autoslicer [14] automates partitioning for protecting sensitive data using dynamic data-dependency analysis, but it is not applicable to programs consisting of only a single function. Al-Mutawa et al. [21] automatically detect private portions of user data, but they process private sections only at trusted nodes. Existing approaches mainly consider restricted partitioning points because the execution environment is determined by data ownership. Furthermore, automatic partitioning for trusted execution environments (TEE) [15, 16] relies on developer annotations to protect security-sensitive data and functions. In contrast to these approaches, AION can consider all program points by leveraging FHE without violating privacy authority, thereby eliminating the need for specialized trusted hardware such as TEEs.

## 2.3 Privacy-Preserving Cloud Computing

Existing works [65, 66] employ federated learning to preserve privacy in cloud computing. Federated learning enables deep learning model training without directly sharing sensitive data, but the exchanged model updates may still leak private information. Moreover, federated learning is inherently targeted at training workloads and does not generally support outsourced cloud execution of arbitrary inference or tensor programs. Other works [22, 23, 24, 25, 26, 67, 68] adopt differential privacy to protect information in cloud-based machine learning. Although differential privacy helps prevent information leakage from computation results, it does not solve the general problem of securely outsourcing computation itself. Another line of work [27, 28, 29, 69, 70, 71, 72] uses secure multi-party computation (MPC) to protect privacy. MPC enables arbitrary computation while preserving privacy, but it requires large amounts of communication between parties during execution. In contrast, AION targets fully homomorphic encryption (FHE), which avoids such intensive online interaction and is therefore better suited to edge-cloud computing.

## 2.4 Fully Homomorphic Encryption Fundamentals

Fully homomorphic encryption enables computations on encrypted data without the need for decryption. As shown in Figure 2.1, in edge-cloud computing, the edge device encrypts private data and sends the encrypted data to the cloud. The cloud processes the data according to the given computation without decrypting it and sends the encrypted result back to the edge. The edge device that has the secret key decrypts the result and

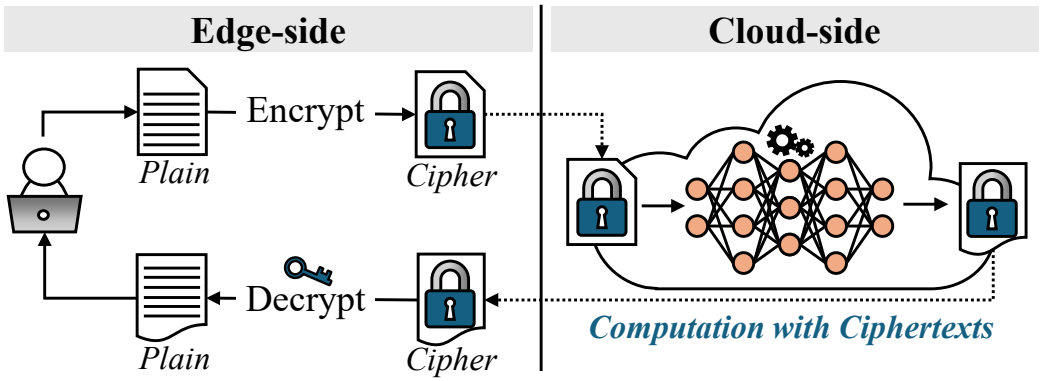



Figure 2.1: Execution model of homomorphic encryption. FHE enables computation with ciphertexts without decryption.

accesses the plaintext output. During cloud-side computation, the data and intermediate results remain encrypted, so the cloud cannot access their contents.

The FHE schemes such as BGV [31], BFV [32], and CKKS [35] allow arbitrary computation based on the Ring-Learning with Errors (R-LWE) problem. The schemes first embed the given message $m$ into a plaintext $\mathcal{P}_m$ in a polynomial ring ($\mathcal{P}_m \in \mathbb{Z}_Q[X]/(X^N+1)$) and then encrypt it as a ciphertext $\mathcal{C}_m = (-as + \mathcal{P}_m + b, a)$, where $a$ and $b$ are random polynomials and $s$ is a secret key.

The encryption parameters $Q$ and $N$ are called the coefficient modulus and polynomial modulus degree, respectively. Because the value embedded in a plaintext is stored as a coefficient of a polynomial, the value is bounded by the coefficient modulus $Q$. Recent works [33, 73, 74] propose constructing the coefficient modulus as a product of small moduli ($Q = \prod q_i$), and the number of moduli is called the level $l$. In practice, the small moduli are selected within a machine word size (e.g., 64-bit integers), and the level $l$ determines the latency and size of FHE operations.

Table 2.1: Symbols and Definitions in Fully Homomorphic Encryption

| Symbol | Definition |
|---|---|
| | Message in FHE |
| $m$ | Original Message |
| $\mathscr{P}_m$ | Plaintext which message $m$ is encoded |
| $\mathscr{C}_m$ | Ciphertext which plaintext $\mathscr{P}_m$ is encrypted |
| | Encryption Parameters |
| $N$ | Polynomial modulus degree |
| $L$ | Initial level of ciphertext $\mathscr{C}_m$ |
| | Ciphertext Parameters |
| $a, b$ | Random polynomials used in encryption |
| $s$ | Secret key |
| $Q$ | Coefficient modulus as a product of small moduli ($Q = \Pi q_i$) |
| $q_i$ | $i$-th prime number |
| $l$ | Level of ciphertext $\mathscr{C}_m$ |

Because ciphertexts in FHE contain noise to protect the plaintext and secret key, homomorphic operations increase the noise level. In particular, the multiplication depth of a ciphertext is bounded by the coefficient modulus $Q$. The depth of multiplication refers to the number of multiplications applied to a ciphertext.

Because the coefficients of ciphertext polynomials increase exponentially with the multiplication depth, existing schemes employ a modulus switching technique, called *rescale*, which reduces both the coefficients and their modulus to keep the noise growth manageable. Even with the rescale operation, the size of the coefficient modulus limits the maximum multiplication depth. In other words, the depth of multiplication of a program determines the required coefficient modulus $Q$ and the level $l$, which represents the number of rescale operations applicable to a ciphertext. Several works [41, 43,

44] propose methods for selecting efficient encryption parameters and managing them with modulus switching.

Figure 2.2 illustrates that the level of an FHE ciphertext greatly affects its computation latency. Starting from the initial level $L$, each rescale operation decreases the level by 1. If the level becomes 0, the noise grows larger than the coefficient modulus $Q$, leaving the ciphertext with no meaningful information. Thus, increasing the initial level enables deeper multiplication but also increases the computation latency. This characteristic makes partitioning an FHE program with lower ciphertext levels crucial for improving performance.

FHE parameters make privacy-preserving partitioning a cost-sensitive compiler problem. Encrypting a value changes its representation and communication size. Moving more operations into the encrypted cloud region changes the level required by the ciphertexts and the number of homomorphic operations. Moving fewer operations into the cloud may reduce encrypted computation but increase boundary communication. These trade-offs motivate an FHE-aware partitioning strategy that evaluates encrypted computation and ciphertext communication together.

## 2.5 Homomorphic Encryption in Edge-Cloud Computing

P$^2$-SWAN [75] presents a secure mobile computing framework that applies the Paillier cryptosystem [76] to sensor data. However, the study reports only encryption latency on the edge device and does not analyze practical computation latency. PL-FedIPEC [77] proposes low-latency federated learning using an improved Paillier cryptosystem, but it focuses on reducing decryption latency at the edge during training rather than parti-

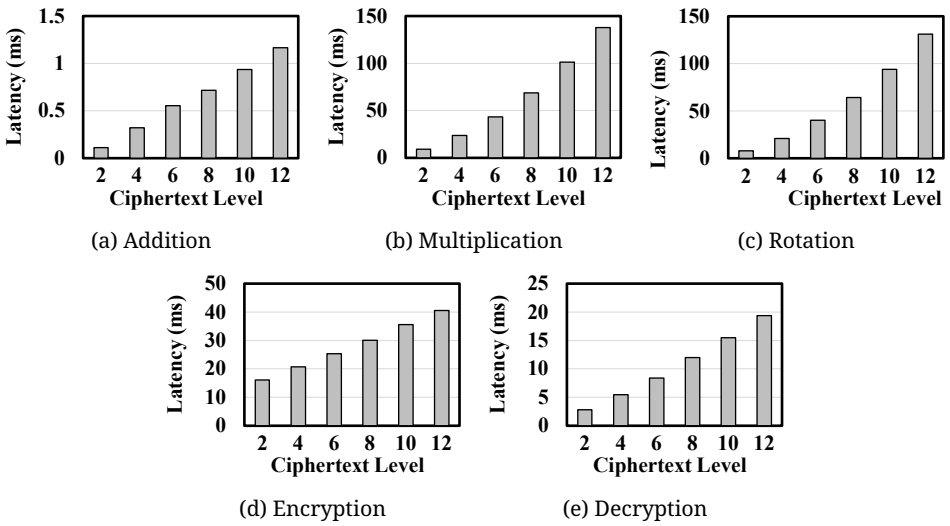



Figure 2.2: Latency according to ciphertext level and FHE operations. The latency increases as the level grows. A higher level enables deeper multiplication depth in an FHE program, which increases computation time.

tioning neural-network execution for performance. Rahman et al. [78] divide large AI tasks into multiple subtasks and offload them to edge devices. By encrypting quality-of-service (QoS) data, privacy is preserved during AI-based service composition, but the framework specifically assumes that QoS data are the sensitive component. In more general settings where edge inputs themselves are private, encrypted multiplication between QoS data and edge data introduces significant overhead. Li et al. [79] manage data from each sensor using FHE with labels in industrial IoT environments, but they do not consider optimized partitioning for performance. SelectiveCrypt [80] and Ramesh and Govindarasu [81] explore the use of HE in edge-cloud settings, but they do not account for the impact of HE levels on communication and computation costs. SelectiveCrypt also encrypts all sensitive data and does not provide users with control over the of-

Table 2.2: Comparison of tensor and IR support in FHE compilation systems and encrypted tensor execution systems.

| Work | Input | Tensor Semantics | Explicit IR Structure | Standard IR | Layout / Packing | Large Tensor |
|---|---|---|---|---|---|---|
| EVA / CHET [39, 41] | FHE Primitive | ○ | ○ | ○ | ○ | ○ |
| Hecate [43] | FHE Primitive | ○ | ● | ○ | ○ | ○ |
| ELASM [44] | FHE Primitive | ○ | ● | ○ | ○ | ○ |
| Lee et al. [47] | FHE Primitive | ○ | ● | ○ | ○ | ○ |
| DaCapo [48] | Custom DSL | ◐ | ● | ○ | ◐ | ○ |
| HALO [82] | Custom DSL | ◐ | ● | ○ | ◐ | ○ |
| QiWu [60] | Custom DSL | ◐ | ○ | ○ | ◐ | ○ |
| Fhelipe [59] | NumPy-style DSL | ◐ | ○ | ○ | ● | ◐ |
| Orion [61] | PyTorch-style DSL | ● | ○ | ○ | ● | ● |
| Ant-Ace [58, 83] | ONNX | ● | ● | ◐ | ● | ● |
| HEIR [62] | Multiple frontends + MLIR | ◐ | ● | ● | ◐ | ◐ |
| **AION** | PyTorch-style DSL + MLIR | ● | ● | ● | ● | ● |

floading point. More broadly, existing HE-based edge-cloud studies protect data from the opposing device or another party, but they do not jointly optimize balanced computation offloading under encryption. In contrast, AION reflects FHE parameters that influence performance, identifies plaintext-computable points through privacy-aware analysis, and estimates both computation and communication cost to reduce latency.

## 2.6 FHE Compilation Systems and Encrypted Tensor Execution

FHE compilation systems reduce the burden of writing encrypted programs by automating the low-level decisions required for homomorphic execution. CHET [39] and EVA [41] provide compiler support for CKKS programs and automate parameter and level management. EVA further inserts scale-management operations to maintain numerical correctness during encrypted execution. Hecate [43] and ELASM [44] improve FHE program performance through fine-grained scale management. These systems explore ex-

Table 2.3: Comparison of FHE parameter management and partitioning support in FHE compilation systems and encrypted tensor execution systems.

| **Work** | **Input** | **Level/Scale Mgmt.** | **Program Partitioning** | **Authority-Aware** |
|---|---|---|---|---|
| EVA / CHET [39, 41] | FHE Primitive | ● | ○ | ○ |
| Hecate [43] | FHE Primitive | ● | ○ | ○ |
| ELASM [44] | FHE Primitive | ● | ○ | ○ |
| Lee et al. [47] | FHE Primitive | ● | ○ | ○ |
| DaCapo [48] | Custom DSL | ● | ○ | ○ |
| HALO [82] | Custom DSL | ● | ○ | ○ |
| QiWu [60] | Custom DSL | ○ | ○ | ○ |
| Fhelipe [59] | NumPy-style DSL | ◐ | ○ | ○ |
| Orion [61] | PyTorch-style DSL | ◐ | ○ | ○ |
| Ant-Ace [58, 83] | ONNX | ◐ | ○ | ○ |
| HEIR [62] | Multiple frontends + MLIR | ◐ | ○ | ○ |
| **AION** | PyTorch-style DSL + MLIR | ● | ● | ● |

ecution plans that satisfy accuracy and latency requirements, but this process can be expensive. Lee et al. [47] address this overhead by using reverse analysis to reason about the remaining scale and generate optimized plans without exhaustive search.

Recent work also studies deeper encrypted programs that require bootstrapping and level-aware execution planning. DaCapo [48] proposes bootstrap optimization for efficient scale management, and HALO [82] studies loop-aware bootstrapping management and primitive-level optimization after low-level encrypted programs have been constructed. These systems make FHE execution more practical by improving how encrypted primitive programs are parameterized and optimized.

Other FHE compilers and program translators target broader input languages or backend generation. HECO [45] automates ciphertext-parameter management and low-

ers high-level programs to target FHE backends. Coyote [46] optimizes rotation operations and automatically vectorizes encrypted circuits. Cingulata [36, 84], E3 [85], and Marble [86] compile general applications into Boolean or arithmetic circuits for encrypted execution. RAMPARTS [38] converts Julia functions into optimized arithmetic circuits. nGraph-HE [40] and AHEC [42] use deep-learning graph representations as frontends and apply FHE-aware optimizations such as vectorization, tiling, and data-layout selection. Porcupine [87] provides an HE domain-specific language and maps plaintext kernels to HE instructions with data-layout optimization.

A separate line of work focuses more directly on encrypted tensor execution. Lee et al. [58] show that efficient encrypted tensor execution can be formulated as a layout-mapping problem and demonstrate multiplexed packing for tensor-like workloads. This packing strategy shows how multi-dimensional tensor data can be embedded into SIMD ciphertext slots so that tensor operators can be implemented with regular rotation and multiplication patterns. Fhelipe [59] explores automatic layout selection and layout conversion for encrypted array programs written in a NumPy-style interface. QiWu [60] analyzes valid and invalid slot regions to improve ciphertext utilization and enable profitable ciphertext fusion. Orion [61] improves programmability by providing a PyTorch-style interface and generating specialized encrypted kernels for neural-network workloads. Ant-Ace [58, 83] starts from ONNX-level input and uses layout-aware encrypted kernel generation for tensor operators. HEIR [62] demonstrates that homomorphic compilation can be organized around MLIR [88], providing reusable compiler abstractions for encrypted computation.

These systems cover important parts of the encrypted execution stack. Some sys-

tems focus on FHE primitive programs and provide level, scale, parameter, and bootstrapping management. Others raise the input abstraction to tensor programs and automate packing, layout reasoning, and kernel generation. Tables 2.2 and 2.3 summarizes these differences across FHE compilation systems and encrypted tensor execution systems. The table uses filled, half-filled, and empty circles to indicate direct support, restricted or indirect support, and unsupported features, respectively.

AION builds on this line of work but targets a different compilation problem in privacy-preserving edge–cloud execution. Existing FHE compilers usually assume that the encrypted computation has already been selected or that the input program will be compiled as an encrypted program. Tensor-oriented systems automate encrypted tensor execution, but they do not decide where plaintext execution should end, where encrypted execution should begin, or which device may access each intermediate value in plaintext. AION connects these concerns in one compiler flow. Starting from a tensor program with privacy intent, AION preserves tensor semantics, lowers tensor operators into ciphertext-level execution schedules, tracks data authority and encryption state, selects a partitioning plan using computation and communication costs, and generates coordinated edge and cloud programs.

## 2.7 Threat Model

AION targets an edge-cloud computing environment consisting of `Edge` and `Cloud`. The program code itself is assumed to be shareable without leaking private information, but the data manipulated by the program may include edge-private inputs and cloud-private parameters. The edge-private inputs represent user-side sensitive data, while the cloud-

private parameters represent service-side proprietary data such as model weights. Both parties may execute parts of the program, but each party must not access the other party's private data in plaintext.

This work assumes that the cloud is `honest-but-curious`, as in prior studies [89, 90, 91, 92, 93, 94, 95]. The cloud faithfully executes the assigned program, but it may attempt to inspect intermediate data, inputs, outputs, or model-related values that become available during execution. Therefore, edge-private data that must be processed by the cloud are protected with fully homomorphic encryption. The cloud can evaluate the assigned computation over ciphertexts, but it cannot read their plaintext contents. Similarly, cloud-private values are not transferred to the edge in plaintext unless the developer explicitly authorizes the edge to read them.

The security goal of AION is to prevent unauthorized plaintext access in the compiler-generated edge-cloud program. A private value may be placed on a device only when the device is included in the value's privacy authority set, or when the value is represented as a ciphertext. In particular, AION does not send an edge-private value to the cloud in plaintext, and it does not send a cloud-private value to the edge in plaintext. If a value must cross an edge-cloud boundary without being readable by the receiving side, the compiler inserts the required encryption and communication operations so that the transferred value remains protected.

The guarantee of AION is defined with respect to the privacy intent explicitly specified by the developer. If a value is annotated as edge-private, AION prevents the value from being exposed to the cloud in plaintext. If a value is annotated as cloud-private, AION prevents the value from being exposed to the edge in plaintext. If a value is an-

notated as `All`, AION may communicate or execute the value in plaintext because the annotation states that both devices are authorized to read it. This annotation-based design avoids unnecessary ciphertext communication and homomorphic computation for values that are intentionally shared.

Incorrect or malicious privacy annotations are outside the formal guarantee of AION. If a developer intentionally annotates an actually private value as `All`, the compiler cannot infer the hidden privacy intent from the program alone. In such a case, the generated program is correct with respect to the given annotation, but the annotation itself is incorrect. A more conservative design could encrypt every value that crosses the edge-cloud boundary regardless of its authority annotation. This design would reduce the impact of incorrect annotations, but it also communicates legitimately shared values as ciphertexts and increases communication and computation overhead. AION instead adopts the annotation-based design, which enforces the specified privacy policy while allowing plaintext execution and communication when both devices are authorized.

This threat model assumes that the AION runtime is correctly installed on the edge and cloud, and that key generation, distribution, and deployment are secure. The edge keeps the secret key, while the cloud receives only the keys required for homomorphic evaluation. This work also assumes secure communication channels with standard authentication and integrity protection. Side-channel attacks, application-level inference attacks such as model inversion [96, 97] and model extraction [98, 99], and private-data-dependent control flow are outside the scope of this work. In addition, this work focuses on applications whose outsourced encrypted computation does not depend on control flow determined by private data.

# 3. Overview of AION

This chapter gives an overview of AION and its execution model. This work takes a tensor program with privacy intent as input and produces coordinated edge and cloud programs. The compiler pipeline consists of frontend processing, tensor-to-slot lowering, authority analysis, partitioning point optimization, and code generation. The runtime then prepares keys and executes the generated programs across the edge and the cloud under the threat model described in Chapter 2.7.

## 3.1 AION Compiler

AION is an end-to-end compiler designed to transform a single privacy-sensitive tensor program into coordinated edge and cloud executions under privacy authority constraints. The compiler is designed to integrate two previously separate concerns within a single workflow. On the one hand, it allows developers to describe computation at the level of tensor semantics rather than manually writing low-level FHE operations. On the other hand, it preserves the authority-aware partitioning framework developed in prior work, enabling the compiler to infer which values must remain protected, where encrypted execution is required, and how the computation should be divided between the edge and the cloud.

Figure 3.1 shows the overall structure of the AION compiler. The AION compiler con-

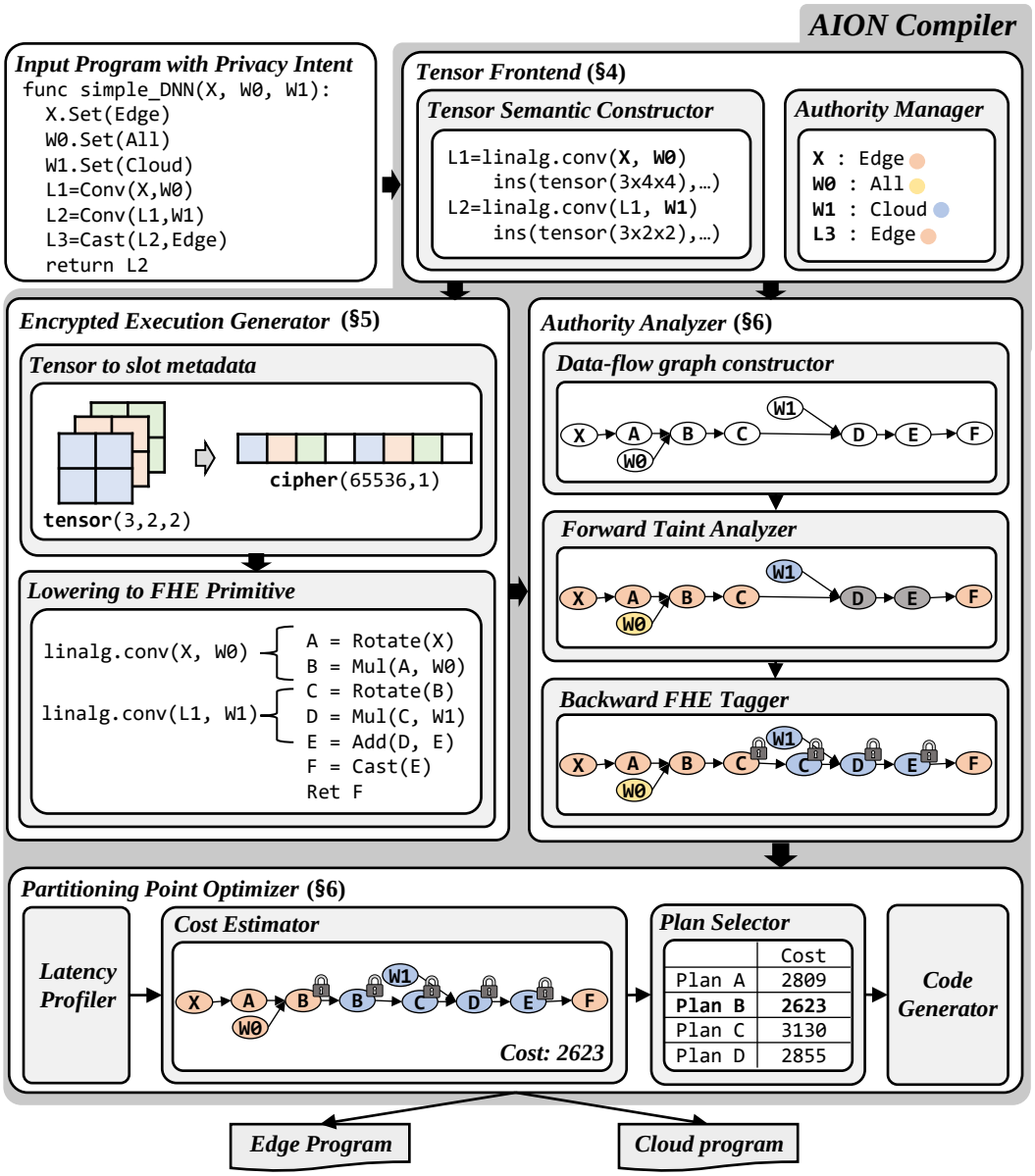



Figure 3.1: AION Compiler Overview

sists of **Tensor Frontend**, **Encrypted Execution Generator**, **Authority Analyzer**, and **Partitioning Point Optimizer**. Starting from an annotated tensor program, the compiler first constructs a tensor-level program representation and collects privacy authority information from the input annotations. Then, the AION compiler lowers supported tensor operators into compiler-visible ciphertext-level execution plans through the encrypted execution generator. These plans describe how tensor values are represented as ciphertext packs and how tensor operators are realized by FHE primitive operations. AION compiler then analyzes privacy authority of derived values, tags the required variables with FHE encryption, explores partitioning points with inference latency and communication cost estimation, and finally generates partitioned edge-cloud programs.

### 3.1.1 Input Program with Privacy Intent

The input to AION is a high-level tensor program written in a torch-like domain-specific language. This frontend allows developers to express machine learning workloads using familiar abstractions such as convolution, matrix multiplication, and element-wise operations. In addition to functional correctness, the program includes lightweight privacy intent, which specifies how data should be treated under authority constraints. Privacy intent is expressed through lightweight annotations such as device ownership specifications and explicit casts, allowing the developer to indicate which inputs are owned by the edge, the cloud, or both, without manually reasoning about encrypted execution or performance-aware partitioning.

### 3.1.2 Structured Tensor Representation Frontend

Upon receiving the input program, AION translates the input program into an internal tensor representation while preserving the privacy intent attached to the program. The Tensor Semantic Constructor extracts tensor operator structure, tensor shapes, and operand relationships from the source program. For example, a convolution in the source program is represented as a structured tensor operation with explicit input, weight, and output shapes. This representation preserves tensor semantics before the program is lowered to ciphertext-level execution.

Alongside this tensor representation, the Authority Manager records frontend privacy metadata. The metadata includes placement annotations, authority annotations on input values, and explicit cast points. Later compiler stages use this metadata to constrain partitioning and to determine which values may appear in plaintext per device.

### 3.1.3 Encrypted Tensor Lowering

Encrypted Tensor Lowering bridges tensor semantics and FHE execution. Tensor programs operate on multi-dimensional arrays, while FHE backends operate on ciphertexts, plaintexts, and primitive operations such as rotation, multiplication, and addition. In Figure 3.1, this stage is implemented by the Encrypted Lowering Generator, which records tensor-to-slot metadata and uses it to lower supported tensor operators into FHE primitives.

Tensor-to-Slot Metadata records how tensor values are represented in ciphertext-level execution. It maps logical tensor coordinates to ciphertext and slot identifiers,

constructs ciphertext packs when a tensor exceeds the slot capacity of a single ciphertext, and propagates layout metadata required by neighboring operators. For windowed operators, this metadata includes the layout information used to determine rotation amounts, plaintext vector construction, selection masks, and output packing.

Lowering to FHE Primitive uses this metadata to translate supported tensor operators into FHE primitive schedules. The compiler recognizes structured tensor operators from supported `linalg` patterns and emits rotations, plaintext constants, multiplications, additions, and pack assembly operations according to the selected lowering rule. In Figure 3.1, a `linalg.conv` operator is lowered by using tensor-to-slot metadata to align spatial offsets with ciphertext rotations and to construct plaintext weight vectors for the corresponding slots.

### 3.1.4 Authority Analysis

Authority Analyzer constructs data-flow graph through encrypted execution graph, propagate authority constraints through the encrypted execution with the privacy metadata, and identifies which intermediate values must remain protected under any legal partitioning scheme. Data-flow graph constructor generates a data-flow graph from the encrypted execution graph, explicitly representing the producer-consumer relationships among intermediate values and operations. Forward Taint Analyzer assigns and propagates privacy authority through the graph according to the AION type system in Chapter 6.1.2. Values that depend on both edge and cloud data may become inaccessible to either side, and are thus assigned a restricted authority state. Backward FHE Tagger revisits the graph from the outputs toward the inputs and marks the regions that should be

computed under homomorphic encryption. Through this backward pass, values whose data authority would be invalid can instead be made available to the cloud in encrypted form without violating the privacy authority. The output is an authority tagged FHE data-flow graph, which serves as AION's basic partitioning plan.

### 3.1.5 Partitioning Point Optimization

AION compiler can explore alternative partitioning strategies and choose the one with the lowest estimated end-to-end cost. Latency Profiler provides profiled execution costs for primitive encrypted operations and communication. The Cost Estimator aggregates these costs over each candidate plan, accounting for both encrypted computation latency and communication overhead. Because moving an operation from the edge to the cloud may still reduce the overall cost if doing so decreases the number or size of cross-boundary transfers, the planner compares candidate partition points with evaluating the total cost of each plan, and selects the minimum-cost one. Code Generator materializes this plan into executable programs for edge device and cloud server.

## 3.2 AION Runtime

AION runtime consists of an offline preparation phase and an online execution phase. The offline phase prepares the partitioned programs, cryptographic keys, and runtime metadata required for collaborative execution. The online phase executes the generated edge and cloud programs cooperatively to provide privacy-preserving inference.

### 3.2.1 Offline Preparation

During the offline phase, AION compiler generates partitioned edge and cloud programs from the annotated source program. Key Generator creates the cryptographic keys required for FHE execution, including a secret key, a public key, and an evaluation key. Deploy Manager collects the generated programs, keys, and runtime metadata, and deploys them to the edge device and the cloud server. The edge receives the public key and the secret key for encryption and decryption, while the cloud uses the evaluation key to execute homomorphic operations over encrypted data.

### 3.2.2 Online Collaborative Execution

During the online phase, the edge and the cloud cooperatively execute the partitioned program generated during the offline phase. The edge processes the plaintext portion of the computation assigned to it, while the cloud executes the encrypted portion on ciphertext data. Whenever data must cross the partition boundary, the runtime performs the required encryption, decryption, and communication according to the generated plan.

In this execution model, the edge handles user interaction and plaintext computation, and the cloud provides encrypted computation without accessing the contents of protected data. The runtime realizes privacy-preserving collaborative execution by coordinating the edge and cloud programs under AION's partitioning strategy.

# 4. AION Frontend: Tensor DSL and Privacy Specification

This chapter presents the frontend programming model of AION. The frontend is designed as a domain-specific language for privacy-preserving edge-cloud tensor execution. In the AION DSL, developers express tensor computation using high-level operators such as convolution, matrix multiplication, pooling, and element-wise arithmetic, and express privacy intent using placement and authority constructs.

The DSL-level constructs are intentionally separated from FHE-specific execution details. Developers do not describe ciphertext slots, rotations, encrypted multiplications, or communication queues. Instead, they describe what the program computes and which values or computations have privacy constraints. The frontend lowers this program into a structured tensor representation together with privacy metadata. Subsequent compiler stages use this metadata to perform encrypted tensor lowering, authority analysis, partition selection, and distributed code generation.

## 4.1 AION Frontend DSL

The AION frontend provides a domain-specific language for describing tensor and model computations at a high level. Instead of manually expressing a program in terms of FHE primitives, developers write applications using familiar deep-learning operators such as convolution, matrix multiplication, pooling, normalization, and arithmetic.

Table 4.1: Operators supported in the AION frontend DSL

| **Operator** | **Description** |
|---|---|
| `AdaptiveAvgPool2d` | Adaptive average pooling |
| `Add` | Elementwise addition |
| `AvgPool2d` | Average pooling |
| `BatchNorm1d` | One-dimensional batch normalization |
| `BatchNorm2d` | Two-dimensional batch normalization |
| `Concat` | Tensor concatenation |
| `Conv2d` | Two-dimensional convolution |
| `Downsample2d` | Spatial downsampling |
| `Linear` | Fully connected layer |
| `Matmul` | Matrix multiplication |
| `Mul` | Elementwise multiplication |
| `Pad` | Tensor padding |
| `Rotate` | Slot rotation abstraction |
| `SiLU` | Sigmoid linear unit activation |
| `Sub` | Elementwise subtraction |

Table 4.1 shows the operators supported in the AION frontend DSL. The operators are intentionally exposed at the level of tensor semantics rather than encrypted execution. Developers can describe the intended computation without reasoning about ciphertext slots, homomorphic patterns, or low-level encrypted kernels.

The supported operators are chosen to cover the common structure of DNN inference workloads. Convolution, matrix multiplication, pooling, normalization, concatenation, and elementwise operators are sufficient to express a wide range of neural-network models while keeping the frontend compact.

## 4.2 Frontend Privacy Specification Interfaces

AION provides two privacy constructs in its DSL. The first is a placement annotation, which constrains where a region or an operation should execute. The second is an authority annotation, which specifies which device is allowed to observe a value in plaintext. Together, these constructs allow developers to express privacy intent without manually rewriting the program into edge and cloud parts or manually inserting FHE operations. Table 4.2 shows the annotations of AION. The privacy intent specified through these interfaces is collected and managed by **Authority Manager** in the frontend. Authority Manager records the source-level placement annotations, input authority annotations, and explicit cast points as frontend privacy metadata.

### 4.2.1 Placement Annotation

The placement annotation construct allows developers to provide guided execution placement at the source level. A region-level placement annotation marks a contiguous computation region as edge-side or cloud-side. An operation-level placement annotation refines this decision for an individual operation.

Placement annotations express the developer's placement intent in the AION DSL. For example, a developer may place a sequence of tensor operators on the cloud using a cloud placement region, while keeping a specific operation on the edge inside that region. The frontend records these annotations as placement metadata. The backend then treats the metadata as partition constraints when constructing the execution plan.

Region-level placement annotations describe coarse-grained execution regions. The

Table 4.2: Privacy-specification constructs in the AION DSL.

| DSL construct | Metadata | Meaning |
|---|---|---|
| `place Edge {...}` | Region placement | Edge-side region. |
| `place Cloud {...}` | Region placement | Cloud-side region. |
| `place Edge op` | Operation placement | Edge-side operation override. |
| `place Cloud op` | Operation placement | Cloud-side operation override. |
| `Set(Edge)` | Value authority | Plaintext access only at the edge. |
| `Set(Cloud)` | Value authority | Plaintext access only at the cloud. |
| `Set(All)` | Value authority | Plaintext access at both devices. |
| `Cast` | Visibility request | Result readable at the edge. |

AION DSL provides two region-level placement constructs, `place Edge {...}` and `place Cloud {...}`. A developer can use `place Cloud {...}` to mark a contiguous region of tensor operations as cloud-side computation, and can use `place Edge {...}` to mark a region that should remain on the edge. These region-level constructs allow developers to describe high-level execution placement without manually rewriting the program into separate edge and cloud variants.

Operation-level placement annotations provide finer-grained control. The AION DSL also allows an individual operation to be annotated with `place Edge` or `place Cloud`. An operation-level edge placement can preserve selected operations at the edge inside an otherwise cloud-side region. Similarly, an operation-level cloud placement can force a selected operation to execute on the cloud even when the surrounding code remains edge-side. In both cases, the annotation is recorded as operation-level placement metadata and later checked when constructing a legal partition plan.

The placement annotation construct corresponds to the guided specification path of AION. It is useful when the developer already has domain knowledge about desirable execution placement and wants to express that knowledge directly in the source pro-

gram. The compiler still remains responsible for materializing the resulting edge-cloud execution. It inserts communication, encryption, and decryption operations where necessary, and it verifies that the resulting program does not violate the privacy authority constraints described by the authority annotations.

### 4.2.2 Authority Annotation Interface

The authority annotation construct is based on data authority annotations. Instead of directly marking where execution should move, the developer declares which device is authorized to read each input or parameter. For example, an input tensor may be edge-private, a model parameter may be cloud-private, and another value may be shared by both devices. The frontend records such declarations using source-level authority annotations such as `Set`. The AION DSL also allows the developer to indicate points where a result should become readable at the edge using `Cast`. At the source level, the interface remains intentionally simple. The developer specifies only ownership information that is already known and the points where plaintext visibility is required. In this way, privacy intent is expressed directly in the user program together with the tensor computation itself. The developer does not need to manually insert encryption, decryption, communication, or FHE operations. Those operations are generated by later compiler stages based on the authority metadata and the selected partition plan.

### 4.2.3 Frontend Examples

Figure 4.1 shows an example DNN program written in the AION DSL. The original DNN program executes two convolution layers and two average pooling layers. The

```
1  Conv2(I, W) {
2    res = 0
3    for k in [0, 6):
4      for a in [0, 25):
5        R = place Edge rot(I[k], a)
6        H = R ∗ W[k][a]
7        res = res + H
8    return res
9  }
10
```

```
11 DNN(x) {
12   A = Conv1(x, W1)
13   B = AvgPool(A)
14   place Cloud {
15     C = Conv2(B, W2)
16     D = AvgPool(C)
17   }
18   return D
19 }
20
```

Figure 4.1: Example DNN pseudocode. Developers add placement constructs to the AION DSL to describe cloud-side regions and edge-side operation overrides without manually splitting the program into edge and cloud code.

developer adds only three placement constructs, shown in blue at lines 5, 14, and 17.

With the `place Cloud` region at lines 14 and 17, the developer specifies `Conv2(B, W2)` and `AvgPool(C)` as a cloud-side execution region in a coarse-grained way. If the developer wants to keep a specific operation on the edge inside this cloud-side region, the developer can use an operation-level `place Edge` construct. In the example, the rotation operation at line 5 (`R = place Edge rot(I[k], a)`) is constrained to execute on the edge. Thus, while the remaining operations in `Conv2` are executed on the cloud server, the rotation operation is executed on the edge.

## 4.3 Structured Tensor Semantic Representation

After the developer writes a program in the AION DSL, the frontend translates the source program into a structured tensor semantic representation. At this level, operators such as convolution are represented as internal tensor operators. Table 4.3 shows representative internal operators used in the tensor semantic representation of AION.

**Tensor Semantic Constructor** extracts tensor-level operator structure, tensor dimen-

Table 4.3: Internal operators used in the tensor semantic representation of AION

| **Operator** | **Role** |
|---|---|
| `constant` | Constant tensor or weight representation |
| `linalg.conv_2d_nchw_fchw` | Convolution |
| `linalg.generic` | Generic elementwise computation |
| `linalg.matmul` | Matrix multiplication / linear layer |
| `linalg.pooling_nchw_sum` | Pooling |
| `tensor.concat` | Tensor concatenation |
| `tensor.collapse_shape` | Shape collapsing |
| `tensor.expand_shape` | Shape expansion |
| `tensor.extract_slice` | Tensor slicing |
| `tensor.pad` | Tensor padding |
| `tensor.transpose` | Tensor transposition |

sions, and operand relationships from the source program, and converts them into internal tensor operators. For example, source-level tensor operators such as convolution and pooling are rewritten into structured internal operations in the tensor and linalg dialects, while preserving their tensor semantics. As a result, AION frontend produces a structured tensor semantic representation of the input program. The representation captures the computation in terms of internal tensor operators, while the privacy intent specified at the source level is separately maintained by the Authority Manager.

## 4.4 Developer Burden Reduced by AION

The programming difficulty of FHE-enabled tensor execution is not fully captured by the number of source lines. A small amount of manual FHE code can encode a large amount of developer responsibility. For example, the developer may have to decide how a tensor is packed into ciphertext slots, which slot rotation corresponds to each spatial offset of a

convolution window, how plaintext weights should be aligned with rotated ciphertexts, and how partial sums are accumulated across ciphertext slots. In an edge-cloud setting, the developer must additionally decide which values should cross the device boundary, when encryption and decryption are required, and which values may be exposed to each device in plaintext. These decisions require knowledge of both tensor execution and FHE execution.

AION reduces this burden by keeping the developer-facing program close to the original tensor program. The developer writes computation using tensor operators such as `Conv2d`, `BatchNorm2d`, `SiLU`, `AvgPool2d`, and `Linear`, which are shown in Table 4.1. Privacy intent is expressed through source-level authority annotations such as `set(Edge)` and `set(Cloud)`. From this program, the compiler constructs the structured tensor representation, lowers supported tensor operators into encrypted execution, performs authority analysis, selects an edge-cloud partition plan, and generates distributed code.

Figure 4.2 compares the developer-facing code structure under four programming styles. The original program in Figure 4.2(a) is shown as tensor-level pseudocode that summarizes the model structure and inference path. This is the part of a PyTorch-style model that the developer would normally write to describe the computation. The program is expressed as a sequence of tensor operators, and the residual block is written using the same high-level operators as the rest of the model.

In a manual FHE implementation, the same tensor computation must be decomposed into ciphertext-level operations. As shown in Figure 4.2(b), the developer directly writes operations such as slot masking, ciphertext rotation, plaintext construction, and accumulation. The code no longer follows the tensor-level structure of the original

```
func ResNet(x):
    out = Conv2d(x)
    out = BatchNorm2d(out)
    out = SiLU(out)

    for block in layer1:
        dsout = out
        out = Conv2d(out)
        out = BatchNorm2d(out)
        out = SiLU(out)
        out = Conv2d(out)
        out = BatchNorm2d(out)
        out = SiLU(out + dsout)

    out = AvgPool2d(out, (8, 8))
    out = Linear(out)
    ...
```

(a) Original tensor program

```
func ResNet_Manual(x, weight, shape):
    mask = make_output_mask(shape)

    for c in range(shape.Cin):
        x[c] = x[c] * mask

    result = []
    for n in range(shape.Cout):
        acc = 0
        for c in range(shape.Cin):
            for kh in range(3):
                for kw in range(3):
                    r = shape.slot_offset(kh, kw)
                    rot = x[c].rotate(r)
                    pt = weight[n][c][kh][kw]
                    acc = acc + rot * pt

        result.append(acc)

    ...
```

(b) Manual FHE code

```
func ResNet_DaCapo(x, model):

    st = {"nt": 2**16, "bb": 32,
          "ko": 1, "ho": 32, "wo": 32}

    s1 = CascadeConv(st, model.conv1)
    close = shapeClosure(**s1)
    out = HE_ConvBN(close, x,
                    model.conv1, model.bn1)
    out = HE_SiLU(out)

    s2 = CascadeConv(s1, model.layer1[0].conv1)
    close = shapeClosure(**s2)
    out = HE_ConvBN(close, out,
                    model.layer1[0].conv1,
                    model.layer1[0].bn1)
    out = HE_SiLU(out)

    s3 = CascadeConv(s2, model.layer1[0].conv2)
    close = shapeClosure(**s3)
    ...
```

(c) Tensor FHE wrapper code

```
func ResNet_AION(x, model):
    x.set(Edge)
    model.set(Cloud)

    out = HE.Conv2d(x, model.conv1)
    out = HE.BatchNorm2d(out, model.bn1)
    out = HE.SiLU(out)

    for block in model.layer1:
        dsout = out
        out = HE.Conv2d(out, block.conv1)
        out = HE.BatchNorm2d(out, block.bn1)
        out = HE.SiLU(out)
        out = HE.Conv2d(out, block.conv2)
        out = HE.BatchNorm2d(out, block.bn2)
        out = HE.SiLU(out + dsout)

    out = HE.AvgPool2d(out, (8, 8))
    out = HE.Linear(out, model.linear)
    ...
```

(d) AION frontend code

Figure 4.2: Programming interfaces for FHE-enabled tensor execution. Manual FHE code exposes primitive operations, while tensor FHE wrappers require framework-specific shape and layout constructs. AION keeps the code close to the original tensor program with only source-level authority annotations.

model. Instead, the developer must reason about the ciphertext layout and manually implement how each tensor operation is realized over packed ciphertext slots.

Tensor-oriented FHE wrappers reduce part of this primitive-level burden, but they still require the developer to use framework-specific constructs. In Figure 4.2(c), the developer no longer writes each rotation and multiplication manually, but must still manage constructs such as `CascadeConv` and `shapeClosure`. These constructs encode shape, layout, and packing information that is specific to the encrypted execution framework. The developer therefore needs to understand the wrapper's execution model in addition to the original tensor program.

The AION frontend keeps the source structure close to the original tensor program. As shown in Figure 4.2(d), the AION code follows the same operator order as the original model code. The main difference is that tensor operators are invoked through the AION frontend, and the developer adds source-level authority annotations for the input and model parameters. The compiler then handles the remaining FHE-specific and edge-cloud-specific tasks, including ciphertext layout construction, encrypted tensor lowering, authority propagation, partition selection, encryption and communication insertion, and edge-cloud code generation. This design separates developer responsibility from compiler responsibility. The developer specifies the tensor computation and the privacy authority of the data. The compiler determines the encrypted execution and generates the edge and cloud programs.

# 5. AION Middleend: Lowering Tensor Semantics with FHE

This chapter presents the tensor lowering stage of AION. The frontend described in the Chapter 4 produces a structured tensor semantic representation using operators in the `linalg`, `tensor`, and `arith` dialects. The representation preserves tensor shapes, operator semantics, and data-flow relationships. This stage translates the supported tensor operators into ciphertext-level FHE execution plans.

FHE libraries provide primitive operations over ciphertext slots. The compiler maps tensor operators such as convolution, pooling, matrix multiplication, and batch normalization to these primitive operations. It also maps tensor values to ciphertext packs. The generated operations include `HE.rotate`, `HE.mul`, `HE.add`, plaintext constants, selection masks, and ciphertext-pack assembly.

Each lowering rule corresponds to the ciphertext operations required by a tensor operator. Pointwise operators apply primitive operations to aligned ciphertext slots. Windowed operators use Tensor-to-Slot Metadata to generate rotations for spatial offsets and accumulate the rotated values. Contraction operators generate rotated activation views and reductions for the contracted dimension. Structural tensor operators reorganize ciphertext packs with rotations and masks.

## 5.1 Structured Tensor Operator Recognition

This chapter explains how AION identifies the tensor semantics that guide encrypted tensor lowering. The Encrypted Execution Generator first recognizes supported tensor operators from structured `linalg` operators or from semantic patterns in `linalg.generic`. It then selects a Lowering to FHE Primitive rule and reads the tensor shapes, iterator structure, and Tensor-to-Slot Metadata required by that rule.

### 5.1.1 Linalg Operator Structure

AION uses the structured information in linalg IR to select a lowering rule. Convolution, depthwise convolution, pooling, and matrix multiplication are represented by dedicated linalg operators. The compiler selects the corresponding lowering rule from the operator kind and its tensor shapes.

Other tensor semantics are represented by `linalg.generic`. A generic operator contains affine indexing maps, iterator types, and a scalar region body. The compiler uses this information to identify a supported semantic pattern.

For a pointwise binary operator, the input and output indexing maps preserve the same coordinates, all iterators are parallel, and the region body yields an arithmetic operation over corresponding elements. For example, the compiler can identify a four-

dimensional channelwise bias addition from its indexing maps pattern.

$$(d_0, d_1, d_2, d_3) \mapsto (d_0, d_1, d_2, d_3),$$

$$(d_0, d_1, d_2, d_3) \mapsto (d_1),$$

$$(d_0, d_1, d_2, d_3) \mapsto (d_0, d_1, d_2, d_3)$$

The first and third maps preserve the tensor coordinates. The second map projects the channel coordinate and indicates that the one-dimensional operand is broadcast across the batch and spatial dimensions. Batch normalization is identified in the same way using two channel-projected operands and a multiply-add scalar body.

For example, a matrix multiplication can also be described by a generic contraction. The corresponding indexing maps pattern has the following form.

$$(d_0, d_1, d_2) \mapsto (d_0, d_2),$$

$$(d_0, d_1, d_2) \mapsto (d_2, d_1),$$

$$(d_0, d_1, d_2) \mapsto (d_0, d_1)$$

Its `iterator_type` is [`parallel`, `parallel`, `reduction`]. The reduction iterator identifies the contracted dimension. The affine maps identify the coordinates preserved in the output and the coordinates consumed by the contraction.

Figure 5.1 shows the contraction pattern. The linalg representation exposes the two output dimensions and the reduction dimension. The encrypted execution plan contains plaintext weight vectors, rotated activation views, multiplications, and additions for the reduction.

```
%0 = linalg.generic
  ins(%A, %B : tensor<1x128xf64>, tensor<128x10xf64>)
  outs(%C : tensor<1x10xf64>)
  indexing_maps = [
    affine_map<(d0, d1, d2) -> (d0, d2)>,
    affine_map<(d0, d1, d2) -> (d2, d1)>,
    affine_map<(d0, d1, d2) -> (d0, d1)>
  ]
  iterator_types = ["parallel", "parallel", "reduction"]
  {
    ^bb0(%a: f64, %b: f64, %c: f64):
      %mul = arith.mulf %a, %b : f64
      %add = arith.addf %c, %mul : f64
      linalg.yield %add : f64
  } -> tensor<1x10xf64>
```

(a) Matmul operation using linalg.generic operator

```
%A0 = HE.mul %A, %S
%A1 = HE.rotate %A0, [-K]
%A2 = HE.add %A0, %A1

%U0 = HE.constant ...
%U1 = HE.constant ...
...
%R0 = HE.mul (HE.rotate %A2, [0]), %U0
%R1 = HE.mul (HE.rotate %A2, [1]), %U1
...
%R  = HE.add %R0, %R1
%R_1= HE.add %R, (HE.rotate %R, [M])
%R_2= HE.add %R, (HE.rotate %R_1, [2*M])

%B  = HE.constant ...
%O  = HE.add %R_2, %B
```

(b) Lowered encrypted primitive operators

Figure 5.1: Example of linalg lowering for a matrix multiplication operator. The iterator types and affine indexing maps expose a matmul contraction structure, which AION lowers into homomorphic rotations, multiplications, and additions.

**Algorithm 5.1:** Lowering structured tensor operators into FHE primitive operations

**Input :** Structured tensor semantic representation $P$
**Output** Ciphertext-level FHE execution plan $P_{\text{FHE}}$
**:**

**foreach** *op in* $P$ **do**
  **if** *op has a dedicated linalg operator* **then**
    $rule \leftarrow$ SELECT_DEDICATEDRULE($op$)
  **else if** *op is linalg.generic* **then**
    $rule \leftarrow$ MATCH_GENERICRULE($op.maps, op.iterators, op.body$)
  **else**
    $rule \leftarrow$ SELECT_TENSORRULE($op$)
  **end**
  $state \leftarrow$ READ_SHAPE_LAYOUT($op$)
  $constants \leftarrow$ PLAINTEXT_CONSTANTS($rule, state$)
  $result \leftarrow$ EMIT_FHEPRIMITIVES($rule, op.operands, constants, state$)
  PROPAGATE_LAYOUTMETADATA($op, result$)
  Replace $op$ with $result$
**end**
**return** $P_{\text{FHE}}$

### 5.1.2 Lowering Rule Selection

Algorithm 5.1 describes the lowering procedure. The compiler first selects a rule from a dedicated linalg operator or a supported generic pattern. It then reads the Tensor-to-Slot Metadata which have tensor shapes and layout metadata required by that rule. The selected rule generates compile-time plaintext vectors and emits the FHE primitive operations for the output pack.

Table 5.1 summarizes the supported lowering rules. The source patterns remain visible in linalg IR until encrypted tensor lowering. This allows the compiler to choose a rule using tensor semantics before emitting primitive operations.

Table 5.1: Representative tensor-to-FHE primitive lowering rules in AION

| **Tensor semantic pattern** | **Recognition information** | **Generated FHE execution plan** |
|---|---|---|
| Dense constant | Floating-point tensor constant | Partition the tensor into plaintext slot vectors and pad the final vector when required. |
| Pointwise arithmetic | Identity indexing maps, parallel iterators, and arithmetic scalar body | Apply HE.add, HE.mul, or negation to aligned ciphertext-pack elements. |
| Channelwise bias and normalization | Identity tensor maps, channel-projection maps, parallel iterators, and add or multiply-add scalar body | Construct channelwise plaintext constants and apply packwise multiplication and addition. |
| Convolution and depthwise convolution | Dedicated linalg operator, tensor shapes, static kernel, stride, and layout metadata | Generate kernel-offset rotations, plaintext weights, accumulations, selection masks, and output-pack assembly. |
| Pooling | Dedicated linalg pooling operator, window shape, stride, and layout metadata | Generate window rotations, selection masks, accumulations, and output-pack assembly. |
| Matrix multiplication and linear layer | Dedicated matmul operator or contraction maps with parallel and reduction iterators | Generate rotated activation views, plaintext weight vectors, reductions, and optional bias addition. |
| Channel concatenation | Channel-dimension tensor.concat with compatible tensor shapes | Generate rotations, boundary masks, and output-pack assembly. |
| Downsampling with channel padding | tensor.extract_slice followed by channel tensor.pad | Generate selection masks, rotations, and output-pack assembly. |

## 5.2 Ciphertext Packs for Large Tensor Values

The tensor shape remains available during tensor lowering. Let $T$ be a tensor value and let $N$ be the number of slots in one ciphertext. The compiler calculates the number of ciphertexts required for $T$ using the following expression.

$$n_T = \left\lceil \frac{|T|}{N} \right\rceil$$

Here, $|T|$ is the number of tensor elements. The converted value is represented as an ordered ciphertext pack.

$$\text{Pack}(T) = \langle C_0, C_1, \dots, C_{n_T-1} \rangle$$

For the basic partition used by packwise operators, each flattened tensor element at offset $j$ is assigned to a ciphertext and slot by the following mapping.

$$\text{PackIndex}(j) = \left( \left\lfloor \frac{j}{N} \right\rfloor, j \bmod N \right)$$

The first component identifies a ciphertext in the pack, and the second component identifies a slot in that ciphertext.

The same calculation is applied to compile-time plaintext tensors. The compiler partitions a dense constant into chunks of $N$ elements and pads the last chunk with zeros when the number of elements is not a multiple of $N$. This produces plaintext packs aligned with the ciphertext packs consumed by the lowering rules.

Algorithm 5.2 describes the common lowering procedure for a same-shaped packwise operator. The compiler decomposes each operand pack into individual ciphertexts, emits the selected FHE primitive operation for aligned pack elements, and assembles the result pack. An existing pack assembly can be reused directly. Otherwise, the compiler extracts one-ciphertext slices from the converted tensor value.

The ciphertext-pack representation is also used by windowed operators. The multiplexed packing method determines the channel arrangement and spatial slot positions for these operators. When a mapped tensor value exceeds the slot capacity, the lowering rule decomposes the input pack, generates operations for the required input and

**Algorithm 5.2:** Packwise lowering for tensor values larger than one ciphertext

**Input:** Tensor operator $op$, operand packs $P_0, \ldots, P_m$, and slot capacity $N$
**Output:** Output ciphertext pack $P_{\text{out}}$

$n \leftarrow \lceil \text{NumElements}(op.output) / N \rceil$
**foreach** $P_i$ *in* $\{P_0, \ldots, P_m\}$ **do**
  $pieces_i \leftarrow \text{DecomposePack}(P_i, n)$
**end**
**for** $j \leftarrow 0$ **to** $n - 1$ **do**
  $C_j \leftarrow \text{EmitPrimitive}(op, pieces_0[j], \ldots, pieces_m[j])$
**end**
$P_{\text{out}} \leftarrow \text{AssemblePack}(C_0, \ldots, C_{n-1})$
**return** $P_{\text{out}}$

Table 5.2: Ciphertext-pack management during encrypted tensor lowering

| Compilation point | Ciphertext-pack handling |
|---|---|
| Tensor type conversion | Calculate the pack length from the ranked tensor shape and slot capacity. |
| Dense constant | Partition the constant into slot-capacity chunks and pad the last chunk with zeros. |
| Pack decomposition | Reuse an existing pack assembly or extract one-ciphertext slices from the converted tensor value. |
| Packwise primitive | Emit one primitive operation for each aligned pack element. |
| Pack assembly | Assemble the generated ciphertexts as an ordered tensor value for the following lowering rule. |

output ciphertexts, and assembles the output pack. The same representation therefore carries large tensor values across supported packwise and windowed lowering rules.

## 5.3 Layout Metadata Propagation

Windowed lowering uses multiplexed packing [58]. AION carries the required layout information as compiler metadata. The layout determines rotation amounts, plaintext

vector construction, and the pack structure of the generated execution plan.

The layout attribute records the packing strategy and the two-dimensional spacing factor $k = (k_h, k_w)$ use by multiplexed packing.

$$\texttt{hecate.layout} = \{\texttt{strategy} = \texttt{multiplexed}, \texttt{k} = [k_h, k_w]\}.$$

The spacing factor records the spatial spacing used by the packed tensor. The lowering rule reads the attribute together with tensor shape and stride information when it constructs rotations, plaintext constants, masks, and output packs.

Table 5.3 shows metadata propagation. Pointwise and channelwise operators preserve the incoming spacing factor. Channel concatenation requires compatible input spacing factors and preserves the factor for its output. For convolution, pooling, and downsampling, the output factor reflects the stride of the operator. Let the input factor be $k^{\text{in}} = (k_h^{\text{in}}, k_w^{\text{in}})$ and the stride be $s = (s_h, s_w)$. The output factor is computed as follows.

$$k^{\text{out}} = (k_h^{\text{in}} s_h,\ k_w^{\text{in}} s_w).$$

The propagated attribute allows the next lowering rule to construct its ciphertext-level execution plan using the packed layout produced by the previous rule.

## 5.4 Lowering to FHE Primitive for Supported Tensor Operators

This chapter describes the Lowering to FHE Primitive rules used for supported tensor operators. Each rule consumes a recognized tensor operator and the associated

Table 5.3: Propagation of the spacing factor in the encrypted tensor lowering stage

| **Tensor operator** | **Output spacing factor** |
|---|---|
| Pointwise arithmetic | $k^{\text{out}} = k^{\text{in}}$ |
| Channelwise bias and normalization | $k^{\text{out}} = k^{\text{in}}$ |
| Convolution and depthwise convolution | $k^{\text{out}} = (k_h^{\text{in}} s_h, k_w^{\text{in}} s_w)$ |
| Pooling and downsampling | $k^{\text{out}} = (k_h^{\text{in}} s_h, k_w^{\text{in}} s_w)$ |
| Channel concatenation | $k^{\text{out}} = k^{\text{in}}$ |

Tensor-to-Slot Metadata, then emits a ciphertext-level FHE execution plan. The rules are grouped by tensor semantics because pointwise, channelwise, windowed, contraction, and structural operators require different ciphertext operations and layout handling.

### 5.4.1 Pointwise and Channelwise Operators

Pointwise arithmetic preserves the slot order of its inputs. The compiler matches the scalar region body of `linalg.generic` and emits the corresponding primitive operation for each element of the ciphertext pack. Addition emits `HE.add`. Multiplication emits `HE.mul`. Subtraction emits negation followed by `HE.add`. Division by a scalar constant emits multiplication with a plaintext vector containing the reciprocal value.

Channelwise operators use affine indexing maps to identify broadcast semantics. Bias addition constructs a plaintext vector that places each bias value in the slots assigned to its channel and emits packwise `HE.add`. Batch normalization constructs plaintext vectors for the channelwise scale and bias values and emits packwise `HE.mul` and `HE.add`. These rules preserve the incoming spacing factor.

### 5.4.2 Windowed Operators

Convolution, depthwise convolution, and pooling use rotations to align spatial neighborhoods. For convolution, the compiler reads the input shape, output shape, kernel shape, stride, and layout metadata. It generates a plaintext weight vector for each required kernel-offset rotation and output-channel group. Each plaintext vector aligns the corresponding kernel weights with the slots of the rotated input ciphertext.

For each kernel offset, the compiler generates a rotated view of the input ciphertext pack and multiplies it by a compile-time plaintext weight vector. It accumulates the weighted ciphertexts for the input channels and kernel positions. The generated plan selects the slots assigned to each output channel and assembles the output ciphertext pack. The compiler instantiates the ciphertext-level convolution schedule using the multiplexed packing method [58]. The linalg-level tensor shape, stride, and layout metadata provide the information required to construct the schedule during lowering.

Figure 5.2 shows the generated primitive structure. A single convolution operator becomes rotations for kernel offsets, multiplications with plaintext weights, additions, and output pack assembly. The rotation amounts and plaintext vectors are generated from the tensor shapes, stride, and layout metadata.

Depthwise convolution follows the same lowering structure. The plaintext weight construction reflects the channelwise kernel relation of depthwise convolution. Pooling generates rotations and selection masks for the pooling window and accumulates the selected values. The stride is reflected in the output layout metadata.

```
%0 = linalg.conv_2d_nchw_fchw
  ins(%I, %W : tensor<1xCxHxWxf64>, tensor<KxCxRxSxf64>)
  outs(%O : tensor<1xKxHoxWo xf64>)
  strides = dense<[sh, sw]> : tensor<2xi64>
  dilations = dense<[1, 1]> : tensor<2xi64>
  -> tensor<1xKxHoxWoxf64>
```

(a) Conv2D operator using linalg representation

```
%Weight = HE.constant ...
%Sel = HE.constant ...

// kernel-offset rotations
%R00 = HE.rotate %A, [off(0,0)]
%R01 = HE.rotate %A, [off(0,1)]
...
%P00 = HE.mul %R00, %Weight00
%P01 = HE.mul %R01, %Weight01
...
%B   = HE.add %P00, %P01
...
%C1  = Summation(%B, ...)
%C2  = Summation(%C1, ...)
%C3  = Summation(%C2, ...)

// output placement
%Cr  = HE.rotate %C3, [out_off]
%D   = HE.mul %Cr, %Sel
%Out = tensor.concat %D0, %D1, ...
```

(b) Lowered encrypted primitive operators

Figure 5.2: Linalg lowering for a convolution operator. The convolution access pattern is lowered into a deterministic encrypted schedule consisting of kernel-offset rotations, compile-time plaintext weights, structured slot summations, and explicit output-pack placement.

### 5.4.3 Contraction Operators

Matrix multiplication and linear layers generate a ciphertext-level contraction plan. The compiler reads the input length, output length, constant weight tensor, and optional bias tensor. If a preceding shape operation connects a four-dimensional packed tensor to the linear layer, the compiler uses the layout metadata to construct plaintext weight vectors.

The generated plan rotates the activation ciphertext, multiplies each rotated view by a compile-time plaintext weight vector, and accumulates the partial results. The reduction dimension in the linalg iterator types determines the accumulation structure. An optional bias addition is emitted after the reduction, as shown in Figure 5.1.

### 5.4.4 Structural Tensor Operators

Structural tensor operators preserve tensor semantics while changing the organization of ciphertext packs. The supported channel concatenation rule reads two input packs with compatible tensor shapes and spacing factors. It rotates the second pack, applies plaintext masks at ciphertext boundaries, and assembles the output pack.

Convolution and pooling padding is incorporated into plaintext vector construction. The generated vectors suppress slots outside the logical tensor boundary. The compiler also recognizes downsampling followed by channel padding. It emits selection masks, rotations, and output-pack assembly for the combined tensor transformation.

## 5.5 Result of Encrypted Tensor Lowering

The output of this stage is a compiler-visible ciphertext-level FHE execution plan. The Encrypted Execution Generator produces this plan by combining Tensor-to-Slot Metadata with Lowering to FHE Primitive. Tensor values are represented as ciphertext packs. Supported tensor operators are represented with FHE primitive operations. Layout metadata is propagated through the structured tensor data flow and used to construct the primitive schedules of subsequent operators.

This representation connects the tensor frontend to the authority-aware backend described in Chapter 6. The backend can analyze the generated data-flow graph, determine which values require encrypted execution, estimate the cost of FHE operations and ciphertext communication, and construct the partitioned programs.

# 6. AION Backend: Authority Analysis and Partitioning

This chapter presents the compiler backend of AION. The frontend of AION captures tensor semantics and source-level privacy intent, and the lowering stage translates that program into an encrypted-execution representation. The backend converts the lowered computation into a legal and efficient edge-cloud execution plan that respects privacy authority, inserts homomorphic operations only where it is necessary, and emits programs for distributed execution.

Figure 6.1 shows the overall structure of the authority-driven AION flow with the same running example in Figure 1.1. The implementation consists of the forward taint analyzer, the backward FHE tagging analyzer, the partitioning point optimizer, and the partitioned code generator. The forward taint analyzer generates a tainted data-flow graph (DFG) with the privacy authority type (Chapter 6.2). Based on Rules 6.1-6.4 in Figure 6.7, the forward taint analysis infers the privacy authority types of non-annotated variables. The backward FHE tagging analyzer tags variables with the `None` type as `Cipher` in the tainted DFG (Chapter 6.3). By using FHE, AION compiler can convert `None`-typed variables to be accessible by an edge and a cloud without violating the privacy authority type. The partitioning point optimizer adjusts the partitioning point of the `Cipher`-tagged tainted DFG (Chapter 6.4) to reduce the overall latency of the FHE-enabled partitioned program. The optimizer estimates the total latency of all the pos-

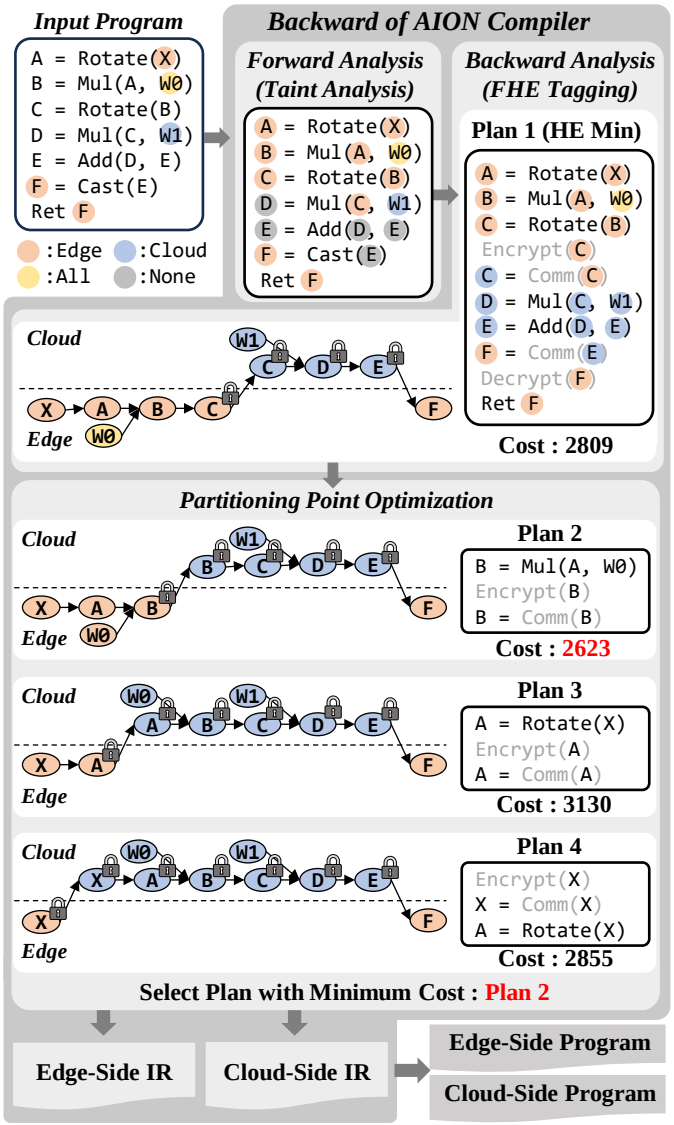



Figure 6.1: Overall design of the AION backend. AION compiler taints all the variables with the privacy authority types according to Rules in Figure 6.7 in a forward direction (*Forward Taint Analysis*), and analyzes proper authority types for the None typed variables using Rules in Figure 6.8 and Figure 6.9 in a backward direction (*Backward FHE Tagging*). To optimize edge-cloud partitioning, AION explores several partitioning points and selects a plan with minimum cost (*Partitioning Point Optimization*).

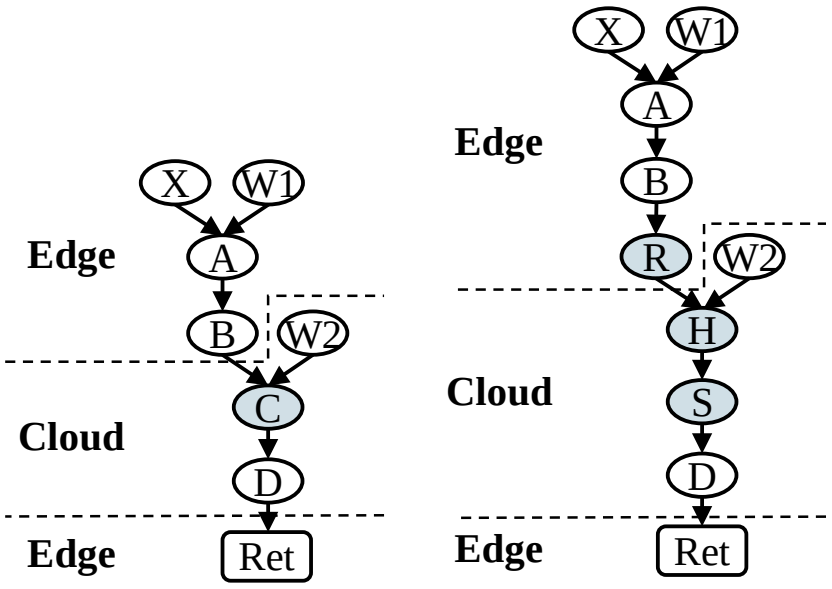
sible partitioning points, and chooses the plan with the minimum estimated latency. Finally, the partitioned code generator inserts the encryption and communication operations and partitions the original program into partitioned programs for the edge and the cloud according to the selected plan (Chapter 6.5).

## 6.1 Authority Analysis

The AION compiler translates an input program into a secure, partitioned execution plan with homomorphic encryption based on the privacy intent specified by the developer. The privacy intent can be articulated from two complementary perspectives, the explicit placement of program operations, and the privacy authority of the data itself. The AION frontend provides developers the flexibility to convey their intent using either or both of these methods. From the operational perspective, developers can manually specify whether particular computations should execute on the edge or the cloud. From the data perspective, developers can restrict variables with privacy authority types, enabling the compiler to automatically deduce the permissible execution environments for the dependent data and operations.

### 6.1.1 Placement Annotation as Partition Constraints

Besides data-level authority annotations, AION allows developers to express privacy intent from the perspective of program operations through placement annotations. These annotations become partition constraints that specify where selected operations should be executed. The backend incorporates these constraints when constructing the edge-cloud execution plan.




(a) Coarse-grained partitioning (b) Fine-grained partitioning

Figure 6.2: Program Dependence Graphs (PDGs) of the example program with the coarse- and fine-grained partitioning plans.

Region-level placement annotations provide coarse-grained partition constraints. As introduced in the frontend interface, the AION DSL provides `place Edge ...` and `place Cloud ...` constructs. A `place Cloud ...` region marks the enclosed operations as cloud-side computation, while a `place Edge ...` region marks the enclosed operations as edge-side computation. From these annotations, the compiler constructs an initial coarse-grained partitioning plan over the program dependence graph.

Operation-level placement annotations further refine the initial plan. An operation annotated with `place Edge` is treated as a constraint that keeps the operation in the edge partition, whereas an operation annotated with `place Cloud` is treated as a constraint that places the operation in the cloud partition. These operation-level annota-

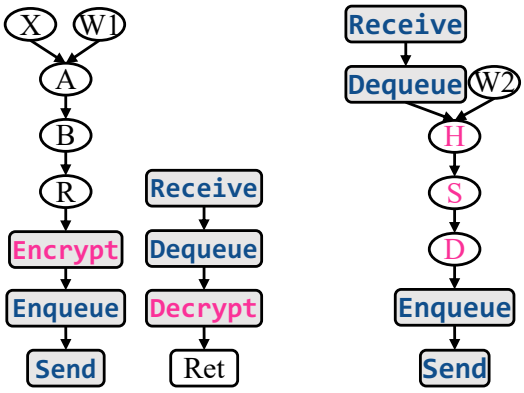
**Algorithm 6.1:** Analyze PDG flow

```
   Input  : Annotated PDG
   Output : Partitioning Plan
 1 Function Partitioning(PDG) :
 2     Initialize(PDG)
 3     for Node in PDG do
 4         if Node is between Annot.Begin and End then
 5             if Node.Annot is Edge and Node is not bi-directional then
 6                 SetEdge(Node)
 7             end
 8             else
 9                 SetCloud(Node)
10             end
11         end
12         if Node.Annot = Cloud then
13             SetCloud(Node)
14         end
15     end
16     PartitioningPlan ← PDG
17     return PartitioningPlan
18 end
```

tions allow developers to express fine-grained placement intent for individual operations, while leaving consistency checking and program transformation to the compiler.

Algorithm 6.1 and Figure 6.2 describe the analysis. As shown in Figure 6.2a, AION first constructs a coarse-grained partitioning plan from the region-level placement annotations. The compiler marks all nodes enclosed by `place Cloud {...}` as part of the cloud partition, while leaving the other nodes in the edge partition unless another placement constraint applies (Lines 4-11). Then, as shown in Figure 6.2b, AION refines the partitioning plan with operation-level placement annotations. The compiler keeps operations annotated with `place Edge` in the edge partition, and places operations annotated with `place Cloud` in the cloud partition (Lines 5-7, 12-14).

If an operation-level placement annotation appears inside an invoked function such




(a) Edge PDG (b) Cloud PDG

**Communication List**

| Variable | Data Size | Direction | Queue |
|---|---|---|---|
| R | 12 MB | Edge to Cloud | Q1 |
| D | 1 MB | Cloud to Edge | Q2 |

**HE Variable List**

| Variable |
|---|
| R, H, S, D |

(c) Metadata tables

Figure 6.3: Partitioned graphs and Metadata Tables. Instructions and variables related to FHE are marked in red, and instructions for communication are indicated in blue.

as `conv2` in Figure 4.1, the annotated operation may not appear in the coarse-grained graph in Figure 6.2a. In this case, AION inlines the function and generates a fine-grained graph as shown in Figure 6.2b. If an operation annotated with `place Edge` appears in the middle of a cloud region and would require bi-directional communication between the edge and the cloud, the compiler may reject the conflicting placement and issue a warning to the developer (Line 5). Through placement analysis, AION produces a partitioning plan that serves as the basis for subsequent authority analysis and program decomposition (Line 16).

Using the partitioning plan, AION splits the PDG in Figure 6.2b into multiple partitioned PDGs like Figure 6.3a and Figure 6.3b. If there is a live-out edge between two

partitions, AION compiler inserts Enqueue and Dequeue at the corresponding end-points of the edge, and adds the live-out variable to the communication list in Figure 6.3c with its direction and estimated communication size. By inserting a Send node at the end of a partition and a Receive node at the beginning of the other partition, the partitioned program can send the enqueued variables from one partition to the other through a communication queue. The received variables are then dequeued by Dequeue at the corresponding program position. AION compiler estimates the communication size from the variable types and the encryption parameters. The estimated communication size is used by the AION runtime to batch communication and amortize overhead.

For example, the live-out edges from R and D in Figure 6.2b are located between different partitions. The compiler duplicates the live-out variables into the source and destination partition, so there are two duplicated R and D nodes in the edge and cloud graphs. Then, AION compiler inserts Enqueue nodes after the variables R and D in the source partitions, and inserts Dequeue nodes before the variables R and D in the destination partitions like Figure 6.3a and Figure 6.3b. Moreover, the compiler adds the variables to the communication list with their direction like Figure 6.3c. The communication size is updated after analyzing FHE variables because encryption changes the data size. At the end of the source partitions and at the beginning of the destination partitions, the compiler inserts Send and Receive nodes. By batching the variables in the communication queues, the AION compiler reduces communication counts between the edge and the cloud.

Second, AION analyzes variables in the cloud partition and marks them as FHE variables if their values are affected by the edge. Starting from the incoming values of the

```
1  conv2_Edge(Q, I) {
2    for k in [0, 6):
3      for a in [0, 25):
4        R[k][a] = rot(I[k], a)
5        Encrypt(R[k][a])
6        Enqueue(Q, R[k][a])   }
7
8
9
10
11
12
```

```
13 DNN_Edge(x) {
14   Q_1 = InitQueue()
15   A = conv1(x, W1)
16   B = AvgPool(A)
17   conv2_Edge(Q_1, B)
18   Send(Q_1)
19   // Offloaded code
20   // executed at cloud
21   Q_2 = Receive()
22   D = Dequeue(Q_2)
23   ret = Decrypt(D)
24   return ret }
```

Figure 6.4: Partitioned code for the edge. AION compiler inserts communication codes and encryption/decryption codes.

```
1  conv2_cloud(Q, W) {
2    S = 0
3    for k in [0, 6):
4      for a in [0, 25):
5        R = Dequeue(Q)
6        H = HE_mul(R, W[k][a])
7        S = HE_add(S, H)
8    return S }
```

```
9  DNN_cloud() {
10   Q_2 = InitQueue()
11   Q_1 = Receive()
12   C = conv2_cloud(Q_1, W2)
13   D = HE_AvgPool(C)
14   Enqueue(Q_2, D)
15   Send(Q_2)  }
16
```

Figure 6.5: Partitioned code for the cloud. AION compiler inserts communication codes and transforms the plain operations into homomorphic operations.

cloud partition such as `R`, the AION compiler follows dependence flows like `H`, `S`, and `D`, and marks those variables as FHE variables. By encrypting these variables, the cloud server cannot infer their values from the edge. Then, if the HE variables are located in the edge graph, the compiler inserts `Encrypt` and `Decrypt` operations for the variables. Finally, the compiler calculates the encrypted data size for the variables in the communication list based on their original data types and encryption parameters.

Figure 6.4 and Figure 6.5 show the edge and cloud partitioned code from the partitioning plan. The compiler places all the instructions in the edge (cloud) graph and inserts communication codes such as `Enqueue`, `Dequeue`, `Send` and `Receive` like the

blue colored instructions in Figure 6.4 and Figure 6.5. Moreover, at the beginning of the partitioned codes, the compiler inserts queue initialization codes like line 14 in Figure 6.4 and line 10 in Figure 6.5. Since there exist two communication directions such as edge to cloud and cloud to edge, the compiler generates two communication queues such as `Q_1` and `Q_2` and assigns the queues for the variables in the communication list. For example, since `R` is sent from the edge to the cloud, the compiler assigns `Q_1` for `R`, while assigning `Q_2` for `D` because `D` is sent from the cloud to the edge.

AION compiler inserts encryption and decryption code at the edge side. For each variable in the homomorphic encryption variable list, the compiler inserts an encryption or decryption operation according to its communication direction. If a variable moves from the edge to the cloud, the compiler inserts an encryption operation like line 5 in Figure 6.4. On the other hand, if the variable is communicated from the cloud to the edge, the compiler inserts a decryption operation like line 23 in Figure 6.4. If a variable is not in the communication list, it remains only on the cloud side. Since that variable is already a result of a homomorphic operation and will be consumed by another, no additional encryption or decryption is unnecessary.

### 6.1.2 Authority Semantics and Inference Rules

This work introduces the authority-aware language and AION type system used in AION for privacy-preserving edge-cloud execution with FHE. The AION language allows developers to write a privacy-preserved cloud application by only annotating the privacy authority of input variables without considering performance-aware partitioning or homomorphic encryption. Chapter 6.1.2 describes the syntax of the AION language with

$$
\begin{array}{rcl}
Prg & ::= & \overline{F} \\
F & ::= & \texttt{func}\ fid\ (\ \overline{v : T}\ )\ \{stmt; \overline{e}\} \\
stmt & ::= & v := e \mid stmt; stmt \\
e & ::= & const \mid v \mid Op \mid Cast(e) \\
 & & \mid Encrypt(e) \mid Decrypt(e) \mid Comm(e) \\
const & ::= & (k_1, k_2, ..., k_n) : T \\
Op & ::= & -e \mid e + e \mid e \times e \mid Rotate(e, i) \\
T & ::= & (Enc, Auth) \\
Enc & ::= & Cipher \mid Plain \\
Auth & ::= & Edge \mid Cloud \mid All \mid None
\end{array}
$$

$$
i \in \mathbb{Z}^+ \quad n \in \mathbb{N} \quad k \in constants
$$

Figure 6.6: AION language. Symbols in the gray boxes are internally used in the AION compiler. $T$, $Enc$ and $Auth$ mean type, encryption and privacy authority, respectively. $All$ in $Auth$ means $\{Edge, Cloud\}$.

the AION type system, including symbols that are internally used in the AION compiler, and Chapter 6.1.2 details how the AION type system infers privacy authority of non-annotated variables and how homomorphic encryption changes the authority type.

**AION Syntax and Internal Types** By extending an existing FHE language [43], this work introduces the AION language for privacy-preserving edge-cloud execution. Figure 6.6 shows the syntax of the AION language. Like the existing language, the AION language also provides `Rotate`, `Addition`, and `Multiplication` operations. To manage the privacy authority of each variable, AION additionally provides an authority type `Auth`, whose values include `Edge`, `Cloud`, and `All`. The privacy authority type describes the device authorized to access the variable. The `Edge` and `Cloud` authorities indicate variables accessible only by the edge and cloud, respectively, whereas `All` indicates a

variable accessible by both devices. In addition, the AION language provides a new operation called `Cast` that changes the privacy authority of a variable, thus allowing the variable to be accessed by a certain device.

Figure 6.10a shows an example code written by a developer with the AION syntax. Like the existing program in Figure 1.1a, the AION syntax requires the developer to annotate privacy authority types for private variables. If the privacy authority type of a variable is not annotated, the AION infers its privacy authority type with AION typing rules, or initializes its privacy authority type as the `All` type. Thus, the developer can annotate the privacy authority type only at the required variables.

AION syntax also includes operations and types that are internally used in AION compiler. The privacy authority type is extended to a pair that consists of encryption (`Enc`) and privacy authority (`Auth`). The encryption describes if a variable is encrypted in FHE or not, so the encryption type consists of `Cipher` and `Plain`. The authority analysis additionally introduces the `None` type to `Auth` for the internal authority type analysis, which means that no device is authorized to access the variable.

**Authority Inference Rules** This work proposes privacy authority type inference rules for the AION language, as shown in Figure 6.7, to guarantee the correctness of the privacy authority analysis. While developers define a variable type only with the privacy authority type, the AION infers variables to be encrypted at each program point, and extends the privacy authority type to the encryption-authority pair type. At the end of the forward and backward analyses, there should be no variable with `None` in `Auth` in the program. The AION removes the non-authorized variables by applying FHE, and

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash -e : T} \tag{6.1}$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash Rotate(e, i) : T} \tag{6.2}$$

$$\frac{\Gamma \vdash e_1 : (E, A_1) \quad \Gamma \vdash e_2 : (E, A_2)}{\Gamma \vdash e \oplus e : (E, A_1 \cap A_2)} \tag{6.3}$$

$$\frac{\Gamma \vdash e : (E, A)}{\Gamma \vdash Cast(e) : (E, Edge)} \tag{6.4}$$

$$\frac{\Gamma \vdash e : (Plain, Edge)}{\Gamma \vdash Encrypt(e) : (Cipher, Edge)} \tag{6.5}$$

$$\frac{\Gamma \vdash e : (Cipher, Edge)}{\Gamma \vdash Decrypt(e) : (Plain, Edge)} \tag{6.6}$$

$$\frac{\Gamma \vdash e : (Cipher, A)}{\Gamma \vdash Comm(e) : (Cipher, All - A)} \tag{6.7}$$

$$\frac{\Gamma \vdash e : (Plain, All)}{\Gamma \vdash Comm(e) : (Plain, All)} \tag{6.8}$$

Figure 6.7: AION inference rules aspect of $(Enc, Auth)$

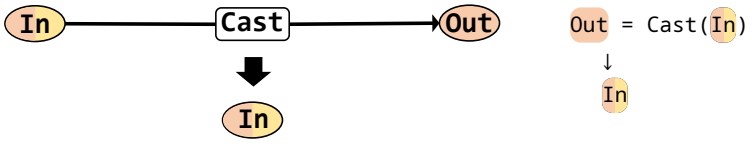
ensures that all program instructions respect the privacy authority type inference rules.

While a unary operation like negation (Rule 6.1) or rotation (Rule 6.2) keeps its result type as its operand type, a binary operation assigns an intersect of its operand types to its result type (Rule 6.3). The cast operation makes a variable accessible to the edge device by changing its privacy authority type (Rule 6.4). It allows a developer to assign authorized device to see results by decrypting. Since the cast operation weakens the privacy protection level, the privacy-set type system recommends developers to use the cast only for limited variables such as the final results.

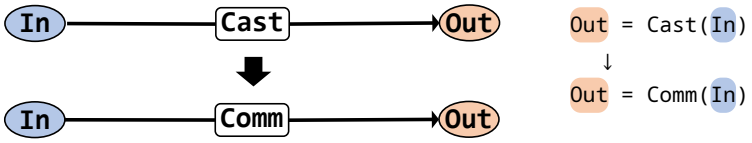
The encrypt operation encrypts its argument, and changes its encryption type from `Plain` to `Cipher` (Rule 6.5). Although the operation does not change its privacy authority type, since its encryption type is `Cipher`, non-authorized devices can also access the variable. The decrypt operation decrypts the data and changes its encryption type from `Cipher` to `Plain` (Rule 6.6). Since the secret key is only available in the edge, only the edge device can encrypt and decrypt the variable. Last, the communication operation describes the data communication operation. If the data is `Cipher`, the data can be transferred while changing its privacy authority type (Rule 6.7). If the data is `Plain`, the data can be transferred only when its authority is the `All` type (Rule 6.8).

## 6.2 Forward Taint Analysis

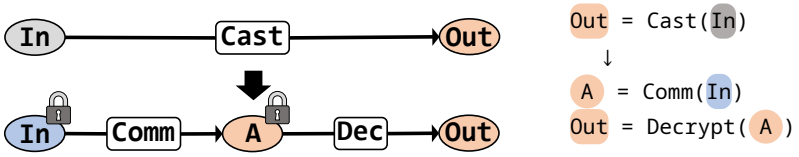
The forward taint analyzer generates a tainted data-flow graph (DFG) based on the annotated types. First, the analyzer extends the privacy authority type to a tuple of the encryption type and the privacy authority type. All the variables have the `Plain` type by default. Then, using Rules 6.1-6.4 in Figure 6.7, the analyzer infers the privacy authority types for the non-annotated variables in a forward direction. While the unary operations keep their operand type as their result type according to Rules 6.1 and 6.2, the binary operation computes the result type as an intersect of its operand types according to Rule 6.3. The cast operation enforces the result as `Edge` type according to Rule 6.4. Figure 6.10b and Figure 1.1d show the example code and the tainted DFG after the forward taint analysis. Due to the `None` typed variables such as `D` and `E`, the tainted DFG is not sufficient for program partitioning yet.




(a) Case 1: When the input privacy authority type is inclusive




(b) Case 2: When the input privacy authority type is opposite



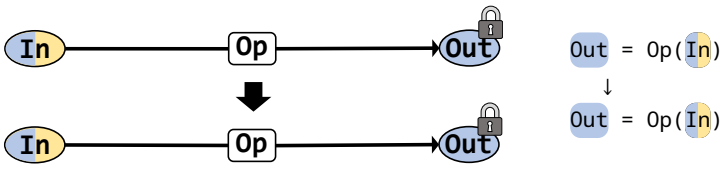

(c) Case 3: When the input privacy authority type is None

Figure 6.8: Backward FHE tagging rules for the `cast` operation

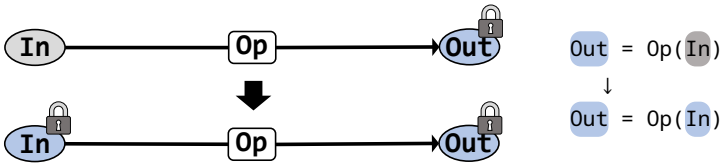
## 6.3 Backward FHE Tagging

The backward FHE tagging analyzer transforms the tainted DFG into an FHE-enabled tainted DFG, which does not include a None typed variable. According to the backward tagging rules in Figure 6.8 and Figure 6.9, the analyzer changes the None type to `Cloud` or Edge type by marking them with `Cipher` types, which means homomorphically encrypted and thus accessible by `Cloud` or Edge. If there is a None type in the final result, the compiler notifies programmers of the remaining variables, so developers can check their privacy authority and insert `Cast` operations at a proper location.

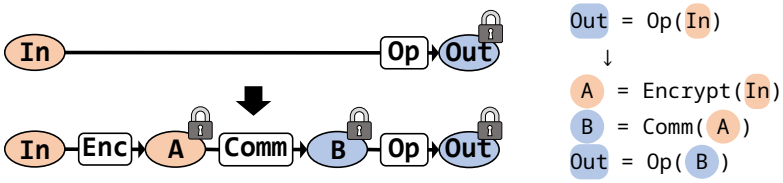
Starting from a `Cast` operation that explicitly changes the privacy authority type




(a) Case 1: When the input privacy authority type is inclusive




(b) Case 2: When the input privacy authority type is None




(c) Case 3: When the input privacy authority type is opposite

Figure 6.9: Backward FHE tagging rules for the unary and binary operations

of a variable, the analyzer infers the beginning and the end points of `Cipher` types in a backward way. Since the `Cast` operation explicitly specifies the target device of its result, the analysis infers the privacy authority type of its operand in a backward way. Figure 6.8 illustrates how the analyzer transforms the privacy authority type of an input variable of the `Cast` operation for all the cases. If the input type includes its output type (Figure 6.8a), the analyzer only removes the `Cast` operation while keeping the input type (`Edge` or `All`). If the input type is opposite to its output type (Figure 6.8b), the analyzer transforms the `Cast` operation to a `Comm` operation that transfers the input variable from the input type device to the output type device. Finally, if the input type is the `None` type that requires homomorphic encryption (Figure 6.8c), since the `Cast` operation makes

```
func rotNmul(X:(Edge), W0:(All),
             W1:(Cloud)):
 For i in [0,3):
  A[i] = Rotate(X, i)
  B[i] = A[i]*W0[i]
 E = 0
 For i in [0,3):
  For j in [0,2):
   C[i][j] = Rotate(B[i],j)
   D[i][j] = C[i][j]*W1[j]
   E = E + D[i][j]
 F = Cast(E)
 return F
```

(a) User-level example code with AION syntax

```
func rotNmul(X:(Plain,Edge), W0:(Plain,All),
             W1:(Plain,Cloud)):
 For i in [0,3):
  A[i]:(Plain,Edge) = Rotate(X, i)
  B[i]:(Plain,Edge) = A[i]*W0[i]
 E:(Plain,All) = 0
 For i in [0,3):
  For j in [0,2):
   C[i][j]:(Plain,Edge) = Rotate(B[i],j)
   D[i][j]:(Plain,None) = C[i][j]*W1[j]
   E:(Plain,None) = E + D[i][j]
 F:(Plain,Edge) = Cast(E)
 return F
```

(b) Tainted code after forward analysis

Figure 6.10: User code and the result of forward analysis.

```
1  func rotNmul(X:(Plain,Edge), W0:(Plain,All),
2               W1:(Plain,Cloud)):
3   For i in [0,3):
4    A[i]:(Plain,Edge) = Rotate(X, i)
5    B[i]:(Plain,Edge) = A[i]*W0[i]
6   E:(Plain,All) = 0
7   For i in [0,3):
8    For j in [0,2):
9     C[i][j]:(Plain,Edge) = Rotate(B[i],j)
10    C1[i][j]:(Cipher,Edge) = Encrypt(C[i][j])
11    C2[i][j]:(Cipher,Cloud) = Comm(C1[i][j])
12    D[i][j]:(Cipher,Cloud) = C2[i][j]*W1[j]
13    E:(Cipher,Cloud) = E + D[i][j]
14  E1:(Cipher,Edge) = Comm(E)
15  F:(Plain,Edge) = Decrypt(E1)
16  return F
```

(a) FHE-tagged code after backward analysis

```
1  func rotNmul(X:(Plain,Edge), W0:(Plain,Edge),
2               W1:(Plain,Cloud)):
3   For i in [0,3):
4    A[i]:(Plain,Edge) = Rotate(X, i)
5    B[i]:(Plain,Edge) = A[i]*W0[i]
6    B1[i]:(Cipher,Edge) = Encrypt(B[i])
7    B2[i]:(Cipher,Cloud) = Comm(B1[i])
8   E:(Plain,Cloud) = 0
9   For i in [0,3):
10   For j in [0,2):
11    C[i][j]:(Cipher,Cloud) = Rotate(B2[i],j)
12    D[i][j]:(Cipher,Cloud) = C[i][j]*W1[j]
13    E:(Cipher,Cloud) = E + D[i][j]
14  E1:(Cipher,Edge) = Comm(E)
15  F:(Plain,Edge) = Decrypt(E1)
16  return F
```

(b) FHE-tagged code after partitioning point optimization

Figure 6.11: FHE tagging and partitioning-point optimization results.

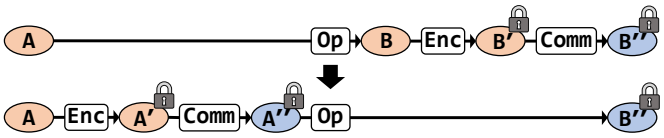
the output variable a plaintext at Edge, the analyzer inserts a ciphertext variable A with the Decrypt operation before the output variable. Moreover, to keep the contents of the input variable secret from the Edge device that has the decryption key, the analyzer inserts the Comm operation between the input variable and the ciphertext A, thus placing the input variable at the Cloud device.

Figure 6.9 illustrates how the analyzer manipulates the ciphertext variable at Cloud in a backward way. If the input type includes its output type (Figure 6.9a), the analyzer keeps the input type as the output type (Cloud) without encryption. By keeping the input variable in the same device (Cloud), the analyzer can save unnecessary communication between Edge and Cloud. If the input type is None (Figure 6.9b), which means that the input variable should be encrypted, the analyzer marks the input type with (Cipher, Cloud). Since the input variable is homomorphically encrypted, and Cloud does not have the private key, Cloud cannot read the contents, although the input variable is located at Cloud. Finally, if the input type is Edge that has the private key (Figure 6.9c), the analyzer inserts the Encrypt operation that homomorphically encrypts the input variable, and sends the ciphertext from Edge to Cloud. Since the variable is homomorphically encrypted, the Cloud device can execute the operation without reading the contents of the input variable.

Figure 6.1 and Figure 6.11a show the FHE-enabled taint DFG and the FHE-tagged code after the backward FHE tagging analysis. In the example, the backward FHE tagging analyzer inserts Comm and Decrypt operations in place of a Cast operation between E and F, thus changing the None type of E to Cloud type. According to Figure 6.9b, the analyzer also changes the None type of D into the Cloud type while assigning the Cipher

```
B :(Plain,Edge) = Op( A :(Plain,Edge))
B':(Cipher,Edge) = Enc( B :(Plain,Edge))
B'':(Cipher,Cloud) = Comm(B':(Cipher,Edge))
                     ↓
A':(Cipher,Edge) = Enc( A :(Plain,Edge))
A'':(Cipher,Cloud) = Comm(A':(Cipher,Edge))
B'':(Cipher,Cloud) = Op(A'':(Cipher,Cloud))
```

(a) Codes before and after the optimization




(b) DFGs before and after the optimization

Figure 6.12: FHE tagging rule for partitioning point optimization. The `Encrypt` and `Comm` operations are moved across the unary or binary operation.

type. Since `C` has the (`Plain`, `Edge`) type, the analyzer inserts `Encrypt` and `Comm` operations between `C` and `D` according to Figure 6.9c. The final FHE-tagged code does not include any `None` type, thus meaning that all the variables and instructions can be assigned to proper devices without compromising the privacy protection level.

## 6.4 Partitioning Point Optimization

The backward FHE tagging analysis generates a valid FHE-tagged code for program partitioning. Since the backward FHE tagging analysis applies homomorphic encryption only to required variables such as `None` typed variables, the analysis minimizes the amount of homomorphic computation. However, this locally minimal use of homomorphic encryption does not always minimize the end-to-end latency. Homomorphic encryption increases the size of values that cross the edge-cloud boundary, and the communication cost of ciphertexts can dominate the latency even when the number

of homomorphic operations is small. Therefore, the FHE-tagged code generated by the backward analysis is treated as an initial legal partitioning plan, not necessarily as the final optimized plan.

The partitioning point optimizer improves this initial plan by trading off communication cost against additional homomorphic computation. It moves the encryption point earlier across selected operations, so that more operations are executed homomorphically on the cloud. This movement can increase cloud computation cost, but it can also reduce the number or size of communicated ciphertexts. AION evaluates this trade-off using profiled operation latency and network bandwidth, and selects the partitioning plan with the lowest estimated end-to-end latency among the explored candidates.

To find a better partitioning point, the partitioning point optimizer explores partitioning points by increasing the number of homomorphic operations using the optimization rule depicted in Figure 6.12. The rule moves the `Encrypt` and `Comm` operations before a preceding unary or binary operation. Before the transformation, the preceding operation is executed on the edge over plaintext data, and its result is encrypted and communicated to the cloud. After the transformation, the input of the operation is encrypted and communicated first, and the operation itself is executed on the cloud over ciphertexts. Thus, the transformation moves the boundary between plaintext edge execution and encrypted cloud execution earlier by one operation.

For each partitioning point, the optimizer estimates the overall latency by reflecting both computation and communication overheads. The total cost of a partitioning plan

$P$ is defined by the following model.

$$Cost(P) = Cost_{edge}(P) + Cost_{cloud}(P) + Cost_{comm}(P) \tag{6.9}$$

The edge-side cost accumulates the operations executed on the edge device.

$$Cost_{edge}(P) = \sum_{op \in Edge(P)} Latency_{Edge}(op) \tag{6.10}$$

Here, $Edge(P)$ denotes the set of operations assigned to the edge in partitioning plan $P$, and $Latency_{Edge}(op)$ denotes the profiled latency of operation $op$ on the edge device. This term includes the latency of ordinary plaintext operations as well as encryption and decryption operations inserted at the partition boundary.

The cloud-side cost accumulates the operations executed on the cloud.

$$Cost_{cloud}(P) = \sum_{op \in Cloud(P)} Latency_{cloud}(op) \tag{6.11}$$

The latency term depends on plaintext or ciphertext.

$$Latency_{cloud}(op) = \begin{cases} Latency_{plain}(op), & \text{if } op \text{ is executed in plaintext} \\ Latency_{FHE}(op, level(op)), & \text{if } op \text{ is executed in ciphertext} \end{cases} \tag{6.12}$$

$Cloud(P)$ denotes the set of operations assigned to the cloud in partitioning plan $P$. A cloud-side operation is executed in plaintext only when the authority type of the involved values allows the cloud to access them in plaintext. Otherwise, the operation is

executed homomorphically over ciphertexts. $Latency_{plain}(op)$ denotes the profiled latency of plaintext operation $op$ on the cloud, and $Latency_{FHE}(op, level(op))$ denotes the profiled latency of homomorphic operation $op$ at ciphertext level $level(op)$. Since FHE operation latency depends on both the operation type and the ciphertext level, AION tracks the level of each ciphertext value and accumulates the corresponding profiled latency for homomorphic operations.

The communication cost is estimated from the amount of transferred data and the profiled network bandwidth.

$$Cost_{comm}(P) = \sum_{v \in Comm(P)} \left( \frac{Size(v)}{BW} + Latency_{msg} \right) \tag{6.13}$$

Here, $Comm(P)$ denotes the set of values communicated between the edge and the cloud in partitioning plan $P$, $BW$ denotes the profiled network bandwidth, and $Latency_{msg}$ represents the per-message communication overhead. For plaintext values, $Size(v)$ is calculated from the tensor shape and element type. For ciphertext values, $Size(v)$ is determined by the number of ciphertexts required to represent the value and the ciphertext size at the corresponding level.

$$Size(v) = NumCtxt(v) \times SizeCtxt(level(v)) \tag{6.14}$$

Here, $NumCtxt(v)$ denotes the number of ciphertexts used to represent value $v$, and $SizeCtxt(level(v))$ denotes the size of one ciphertext at the level of $v$. Since the ciphertext size depends on its level, the optimizer first analyzes the level of communicated

ciphertexts from the homomorphic operations, and then calculates the communication amount. This cost model allows AION to compare a plan that minimizes homomorphic computation with a plan that performs additional homomorphic operations but reduces ciphertext communication.

Algorithm 6.2 describes how the optimizer explores partitioning points. The algorithm adjusts the encryption point because the decryption point is fixed by the `Cast` operation. The `Cast` operation represents the point where the final result becomes readable at the edge, so the optimizer does not move the decryption side. Instead, it searches for a better point where plaintext edge execution changes to encrypted cloud execution.

As in Figure 6.12, the algorithm marks the operation before the `Encrypt` operation with `Pivot`, and moves the `Pivot` operation after the `Encrypt` and `Comm` operations (Line 5-8). This transformation makes the pivot operation execute on ciphertexts at the cloud instead of plaintexts at the edge. Therefore, each movement increases the encrypted computation region while potentially reducing communication overhead.

For the newly generated DFG, the algorithm reassigns the encryption and the privacy authority type not to violate privacy-set inference rules (Line 9). This reassignment also checks whether the transformed DFG is still a legal partitioning plan. A legal plan must ensure that a plaintext value is accessed only by a device included in its authority type, and that a value crossing the edge-cloud boundary is transferred through a `Comm` operation with the proper encryption state. Thus, the optimizer explores only partitioning points that preserve the privacy authority rules.

Then, the algorithm estimates the latency of the candidate plan using the cost model described above (Line 10), and chooses the plan with minimal estimated latency for

**Algorithm 6.2:** Partitioning point optimization

```
   Input  : Tainted DFG
   Output : OptimizedPlan
 1 Function PartitioningPointOptimization(DFG) :
 2     Cost ← CostEstimate(DFG)
 3     OptimizedPlan ← DFG
 4     for EncryptOp in DFG do
           // Find Communication Operation
 5         CommOp ← getNextOp(DFG, EncryptOp)
 6         Pivot ← getPrevOp(DFG, EncryptOp)
           // Swap Pivot Position
 7         DFG ← DFG.swap(Pivot, EncryptOp)
 8         DFG ← DFG.swap(Pivot, CommOp)
           // Re-assign Type to the DFG
 9         AssignTypes(DFG)
           // Estimate Cost of the DFG
10         CandCost ← CostEstimate(DFG)
           // Take Plan with Min. Cost
11         if CandCost < Cost then
12             Cost ← CandCost
13             OptimizedPlan ← DFG
14         end
15     end
       // Assign Device to All Type Data
16     for Op in DFG do
17         if Op.Operand is All Type then
18             SetTypeFrom(Op.Result)
19         end
20     end
21     return OptimizedPlan
22 end
```

`OptimizedPlan` (Line 11-14). Finally, the algorithm assigns the target device of the `All` typed variables as the same device of its use in order to avoid unnecessary communication because the `All` typed variable can be located either in `Edge` or `Cloud` (Line 16-20).

Figure 6.1 shows how the partitioning point optimizer explores the partitioning points and estimates their latency for the example code in Figure 6.11. Starting from the FHE-enabled taint DFG (Plan 1 in Figure 6.1) generated by the backward FHE tagging analysis, the optimizer moves the `Encrypt` and `Comm` operations to the beginning of the program one by one. The optimizer estimates the overall latency for each partitioning plan (Plans 2-4 in Figure 6.1) and chooses Plan 2, which has minimal latency among the plans. Figure 6.11b shows the FHE-tagged code for Plan 2.

## 6.5 Code Generation

The code generation stage materializes the selected partition plan into executable edge and cloud programs. The previous stages determine the encryption state of each value, the device assignment of each operation, and the communication boundary between the edge and the cloud. Backward FHE tagging in Chapter 6.3 identifies the values that must be represented as ciphertexts, and partitioning point optimization in Chapter 6.4 selects the partition plan with the lowest estimated cost among legal candidates. Code generation preserves these decisions by emitting program fragments, communication operations, and runtime metadata for collaborative edge-cloud execution.

The input to the code generator is the optimized FHE-enabled data-flow graph produced by the partitioning point optimizer. Each operation in the graph is assigned to either the edge or the cloud, and each value is annotated with its encryption state and

privacy authority. The graph also contains explicit `Encrypt`, `Decrypt`, and `Comm` operations inserted by the previous stages. From this graph, the code generator emits edge and cloud programs with communication and encrypted variable metadata.

The code generator first separates the optimized graph into an edge-side program and a cloud-side program. Operations assigned to the edge are emitted into the edge program, and operations assigned to the cloud are emitted into the cloud program. Data dependences whose producer and consumer are assigned to the same device remain local to one generated program. Data dependences that cross the device boundary are represented by `Comm` operations in the optimized graph.

A `Comm` operation is an abstract compiler-level marker for inter-device data transfer. The code generator lowers each `Comm` operation into concrete communication operations in the generated programs. On the producer side, the generator inserts an `Enqueue` operation to place the transferred value into a communication queue. After the values in the queue are prepared, the generator inserts a `Send` operation to transmit the queue to the other device. On the consumer side, the generator inserts a matching `Receive` operation to obtain the queue and a `Dequeue` operation to recover the transferred value.

The direction of each generated communication sequence is determined by the producer and consumer of the corresponding data dependence. If an edge-produced value is consumed by the cloud, the generated sequence is emitted as edge-side `Enqueue`/`Send` and cloud-side `Receive`/`Dequeue`. If a cloud-produced value is consumed by the edge, the sequence is emitted in the opposite direction. Multiple values transferred in the same direction at the same boundary can share a communication phase, allowing the runtime to batch values in a queue instead of communicating each variable separately.

The generator emits encryption and decryption operations according to the optimized graph. If an `Encrypt` operation appears before a communication boundary, the edge-side program emits encryption before the value is enqueued for transfer. If a `Decrypt` operation appears after a returned ciphertext, the edge-side program emits decryption after the value is received and dequeued. Code generation therefore materializes the encryption and decryption decisions made by the authority analysis and partitioning optimization stages, while the decision of where encryption should be introduced remains in the previous analysis stages.

For cloud-side encrypted computation, operations over `Cipher` values are emitted as homomorphic operations. Tensor-level operators lowered by the encrypted tensor lowering stage are emitted as ciphertext-level primitive schedules, including rotations, multiplications, additions, and plaintext constants. AION uses Hecate [43] as the FHE backend for encrypted operation generation and scale management. Under this organization, AION constructs the distributed edge-cloud program and its boundary communication, while the FHE backend generates valid encrypted operations inside the cloud-side encrypted region.

The code generator also emits metadata required by the runtime. The communication metadata records each transferred value, its transfer direction, queue identifier, and estimated data size. This metadata is used to match generated `Send` and `Receive` operations during execution. The FHE variable metadata records values represented as ciphertexts in the cloud program. It corresponds to the FHE variable list shown in Chapter 6.3 and initializes cloud-side encrypted execution.

Finally, AION compiles the generated programs separately for their target devices.

The edge program is compiled for the edge-side platform, and the cloud program is compiled for the cloud-side platform. This separate compilation allows each program to reflect its execution environment while preserving the partition boundary selected by the compiler. The backend output is therefore a deployable pair of programs together with the metadata required for collaborative edge-cloud execution.

## 6.6 Security and Correctness of Generated Code

Assuming that the cloud is `honest-but-curious`, this chapter analyzes the security property enforced by the AION compiler. The cryptographic security of ciphertexts relies on the underlying FHE scheme [30, 31, 32, 33, 34, 35] and backend compiler. AION adopts existing FHE compilers [41, 43] as its backend for FHE operations and does not modify the polynomial modulus degree $N$ or the coefficient modulus $Q$ used to determine the security level $\lambda$ [100]. This chapter analyzes the compiler-level safety property of the generated edge and cloud programs. Specifically, it shows that private values are not exposed in plaintext to devices outside their privacy authority.

The proof follows a type-safety argument. AION assigns each value a type that contains both its encryption state and its privacy authority. The encryption state indicates whether the value is represented as `Plain` or `Cipher`, and the privacy authority indicates which device is allowed to access the value in plaintext. The key safety property is that a device may access a value in plaintext only when the device is included in the value's privacy authority. If a value is processed by a device outside its plaintext authority, the value must be represented as ciphertext.

The security guarantee is proved by showing that each compiler stage preserves this

type-safety property. Lemma 1 shows that the forward analysis does not expand plaintext authority while propagating authority types through data flow. Lemma 2 shows that the backward FHE tagging analysis resolves authority-conflicting values by converting them into ciphertexts rather than exposing them in plaintext. Lemma 3 shows that the partitioning point optimization preserves the same property while moving the encryption boundary. Together, these lemmas imply that the generated edge and cloud programs do not expose unauthorized plaintext values.

**Problem Statement.** *The partitioned programs generated by AION do not violate the privacy authority specified by developers. For every value processed by* `Edge` *or* `Cloud`*, the processing device either belongs to the value's privacy authority or accesses the value only in ciphertext form.*

**Assumption.** For the security analysis, this work assumes the followings.

1. The devices are `honest-but-curious`. They faithfully execute the generated programs, but may attempt to inspect the contents of values available at runtime.

2. FHE guarantees that devices cannot read the plaintext contents of ciphertexts without the secret key.

3. Developers correctly annotate the privacy authority of private data. In other words, the sources of privacy-sensitive data flows are properly annotated, and the final results are properly casted when they should become readable by `Edge`.

4. The secret key is available only to `Edge`. `Cloud` receives only the evaluation key required for homomorphic evaluation.

**Equation Setup.** For the security analysis, this work uses the following notation.

- Let $A(v)$ be the privacy authority type of $v$, such as `Edge`, `Cloud`, `All`, and `None`.
- Let $E(v)$ be the encryption type of a variable $v$, such as `Plain` and `Cipher`.
- Let $D(v)$ represent the processing device of a variable $v$, such as `Edge` and `Cloud`.

**Definition 1** (Type-safety property)**.** *For every variable $v$ processed by a device $D(v)$, AION maintains the following type-safety property. If $E(v) =$* `Plain`*, then $D(v) \in A(v)$. If $D(v) \notin A(v)$, then $E(v) =$* `Cipher`*. Thus, an unauthorized device may manipulate a value only in ciphertext form.*

**Lemma 1.** *The forward analysis of AION compiler preserves the type-safety property by never expanding the plaintext authority of derived variables, except for explicit* `Cast` *operations specified by developers. For $v_d = v_1 \oplus v_2, A(v_d) \subseteq A(v_1)$ and $A(v_d) \subseteq A(v_2)$.*

*Proof.* For the annotated variables by the developers, the forward analyzer keeps their annotated privacy authority types, thus not violating the privacy authority.

For the non-annotated variables, the forward analyzer infers their privacy authority types using Rules 6.1-6.4 in Figure 6.7.

**Case 1** (Rules 6.1 and 6.2 in Figure 6.7)**.** The rules are for unary operators such as negation and rotation, and do not change the privacy authority. Since $A(v_d) = A(v_s)$ for $v_d = op(v_s), A(v_d) \subseteq A(v_s)$.

**Case 2** (Rule 6.3 in Figure 6.7)**.** The rule is for binary operators such as addition and multiplication. Since $A(v_d) = A(v_1) \bigcap A(v_2)$ for $v_d = v_1 \oplus v_2, A(v_d) \subset A(v_1)$ and $A(v_d) \subset A(v_2)$.

**Case 3** (Rule 6.4 in Figure 6.7)**.** The rule is for `Cast` operator that the developers explicitly annotate. $A(v_d) = \{\texttt{Edge}\}$ by the rule, and it is intended by the developers.

For all cases in the forward analysis, the privacy authority of the destination is bounded by the authorities of its sources, or is `Edge` as the result of the `Cast` operation inserted by the developer. Thus, the forward analysis of AION does not violate the privacy authority. □

**Lemma 2.** *For the tainted DFG generated by the forward analysis, the backward FHE tagging analysis of AION preserves the type-safety property. The analysis either preserves the existing plaintext authority of a value or changes the value into ciphertext form before it is processed by a device outside its plaintext authority.*

*Proof.* From Lemma 1, the input of the backward analysis does not violate the privacy authority. This work will prove that the backward analysis does not weaken the privacy authority. The backward analysis changes the privacy authority types and the encryption types for the two kinds of operators such as `Cast` and unary/binary operator according to the rules in Figure 6.8 and Figure 6.9.

**Case 1** (`Cast`)**.** The rules in Figure 6.8 covers all the possible cases of the `Cast` operations. While the privacy authority type of the casting result is fixed as `Edge`, the privacy authority type of its input can be `All`, `Edge`, `Cloud` and `None`.

1. `All`, `Edge`: The backward analysis keeps its privacy authority type, thus not weakening its privacy authority.

2. Cloud: The backward analysis replaces the Cast operation to the Comm operation that makes the processing devices of the input and output opposite. Thus, for input $v$, $D(v)$ is the same to $A(v)$ as Cloud.

3. None: The backward analysis changes its encryption type from Plain to Cipher and inserts the Comm and Decrypt operations. Since the input variable is encrypted, Cloud can access it without violating the privacy authority.

Thus, the backward analysis does not weaken its privacy authority for the Cast rule.

**Case 2** (Unary/binary operator)**.** The rules in Figure 6.9 covers all the possible cases of the unary and binary operations. The privacy authority type and encryption type of the output are fixed as (Cipher, Cloud). While the encryption type of the input is Plain, its privacy authority type can be All, Cloud, None and Edge.

1. All, Cloud: The backward analysis keeps its privacy authority type, thus not weakening its privacy authority.

2. None: The backward analysis changes its type from (Plain, None) to (Cipher, Cloud). Since the variable is encrypted, it does not violate the privacy authority.

3. Edge: The backward analysis inserts the Encrypt and Comm operations. Since the analysis keeps the privacy authority and encryption types of the input the same as (Plain, Edge), it does not weaken the privacy authority. The inserted Encrypt operation changes the encryption type, and the Comm operation changes the processing device, bridging the input type (Plain, Edge) and the output type (Cipher, Cloud) without violating the privacy authority.

Thus, the backward analysis does not weaken its privacy authority for the unary and binary operations.

For all the cases, the backward analysis does not weaken its privacy authority. Moreover, if the authority types of the input and the output are opposite as `Cloud` and `Edge`, the analysis inserts the `Comm` operation to change its processing device. Thus, the backward analysis of AION does not violate the privacy authority. □

**Lemma 3.** *For the given FHE-tagged code, the partitioning point optimization of AION preserves the type-safety property while moving* `Encrypt` *and* `Comm` *operations across preceding operations.*

*Proof.* From Lemma 2, the input of the partitioning point optimization does not violate the privacy authority. We will prove that the partitioning point optimization does not weaken the privacy authority.

During the optimization step, the partitioning point optimization moves `Encrypt` and `Comm` operations as Figure 6.12 describes.

$$v_2 = Comm(Encrypt(Op(v_1)))$$
$$\rightarrow v_2 = Op(Comm(Encrypt(v_1)))$$

The partitioning point optimization does not change the privacy authority type and the encryption type of the input and the output. It only changes the privacy authority type and the encryption type of the `Op` from (`Plain`, `Edge`) to (`Cipher`, `Cloud`). Although its processing device becomes changed, since the variable is encrypted as `Cipher`, it

does not violate the privacy authority. Thus, the partitioning point optimization does not weaken the privacy authority.

□

**Theorem.** *The edge and cloud programs generated by AION do not expose private values in plaintext to unauthorized devices.*

*Proof.* From Lemma 1, the forward analysis preserves the type-safety property while propagating privacy authority through data flow. From Lemma 2, the backward FHE tagging analysis resolves authority-conflicting values by inserting encryption, communication, and decryption operations without exposing those values in plaintext to unauthorized devices. From Lemma 3, the partitioning point optimization preserves the same type-safety property while moving the encryption boundary.

Therefore, every compiler transformation performed by AION preserves the type-safety property. Since the code generator partitions the program at `Comm` operations and emits encryption and decryption operations according to the final FHE-tagged graph, the generated edge and cloud programs allow each device to access only authorized plaintext values or ciphertexts. Thus, AION does not expose private values in plaintext to unauthorized devices in the generated partitioned programs. □

# 7. AION Runtime: Deployment and Execution

The AION runtime executes the edge and cloud programs generated by the compiler as a single collaborative application. The compiler determines the partition plan, inserts encryption, decryption, and communication operations, and emits the runtime metadata described in Chapter 6.5. The runtime is responsible for deploying these artifacts to the proper devices, preparing homomorphic encryption keys, initializing communication channels, and executing the generated programs according to the compiler-generated communication protocol.

The runtime consists of an offline preparation phase and an online execution phase. The offline phase is performed before user input is processed. It prepares the partitioned programs, cryptographic keys, and runtime metadata required for collaborative execution. The online phase is performed whenever a user requests inference. During this phase, the edge executes the plaintext part of the program, the cloud executes the encrypted part of the program, and both sides exchange intermediate values through the generated communication queues.

Figure 7.1 illustrates the overall runtime process. The white circled numbers show the offline preparation phase, and the black circled numbers show the online collaborative execution phase.

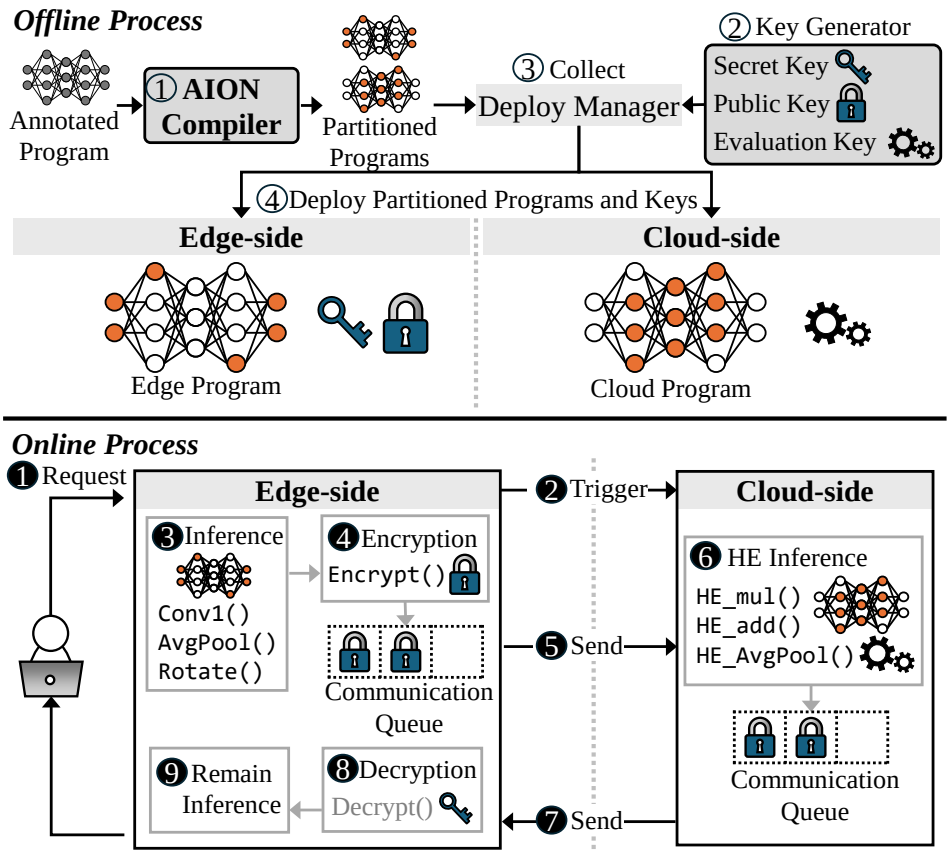



Figure 7.1: Key management and execution process of the AION runtime. The key generation and distribution take place in an offline process. The online process shows how computation is performed on both the edge side and the cloud side.

## 7.1 Offline Preparation

The offline phase prepares all artifacts required to execute the partitioned program. First, the AION compiler translates the annotated source program into two partitioned programs, one for the edge and one for the cloud (Step ①). The generated edge program contains the plaintext computation assigned to the edge, encryption and decryption operations, and communication operations for interacting with the cloud. The generated cloud program contains the computation assigned to the cloud, including homomorphic operations over ciphertext values. Along with these programs, the compiler emits metadata that describes communication queues and encrypted variables.

Second, the key generator creates the keys required for homomorphic execution (Step ②). The generated keys include a secret key, a public key, and an evaluation key. The secret key is used to decrypt ciphertext results and is kept only at the edge. The public key is used by the edge to encrypt values before sending them to the cloud. The evaluation key is used by the cloud to evaluate homomorphic operations over ciphertexts. Because the cloud receives only the key material required for evaluation, it can execute the encrypted sub-program without obtaining the secret key.

Third, the deploy manager collects the generated programs, keys, and metadata (Step ③). The deploy manager packages the edge program with the key material required for edge-side encryption and decryption. It also packages the cloud program with the evaluation key and cloud-side encrypted execution metadata. The communication metadata is provided to both sides so that the generated Send, `Receive`, Enqueue, and Dequeue operations can be matched during execution.

Finally, the deploy manager installs the prepared artifacts on the target devices (Step ④). The edge receives the edge program, the public key, the secret key, and the metadata needed to communicate with the cloud. The cloud receives the cloud program, the evaluation key, and the metadata needed to initialize the encrypted execution context. After deployment, both programs are ready for online execution.

## 7.2 Online Collaborative Execution

The online phase begins when a user invokes the application with input data. The edge acts as the entry point of the execution because user-side private data are initially available at the edge. The edge starts a new execution session and triggers the cloud program so that both sides initialize their communication queues and execution state (Steps ❶ and ❷). The queue layout follows the communication metadata generated by the compiler, so each send operation has a corresponding receive operation on the other side.

After initialization, the edge executes the edge-side part of the partitioned program using plaintext data (Step ❸). This part may include the plaintext prefix of the original program and selected operations that the compiler keeps on the edge. For example, in Figure 7.1, the edge executes operations such as `Conv1`, `AvgPool`, and `Rotate` before transferring data to the cloud. A fine-grained placement annotation can also keep an operation on the edge even when neighboring operations are assigned to the cloud.

When the execution reaches a compiler-generated encrypted boundary, the edge encrypts the intermediate value using the public key (Step ❹). The encrypted value is inserted into the communication queue using the generated enqueue operation. The edge then sends the queued ciphertexts to the cloud according to the generated commu-

nication metadata (Step ❺). The queue abstraction allows multiple boundary-crossing values to be transferred as one communication phase when they belong to the same transfer direction and boundary.

The cloud receives the queued ciphertexts, dequeues them into the variables expected by the cloud program, and executes the cloud-side encrypted computation (Step ❻). The cloud-side program evaluates homomorphic operations such as `HE_mul`, `HE_add`, and `HE_AvgPool`. These operations are performed over ciphertexts using the evaluation key. Since the transferred data remain homomorphically encrypted, the cloud can process them without reading their plaintext contents.

After completing the cloud sub-program, the cloud enqueues the encrypted result and sends it back to the edge (Step ❼). The returned value remains ciphertext it should become readable at the edge. Upon receiving the result, the edge dequeues the ciphertext and decrypts it with the secret key (Step ❽). Because the secret key is available only at the edge, only the edge performs decryption.

The edge then resumes the remaining edge-side computation using the decrypted plaintext value (Step ❾). This suffix computation may include post-processing operations or the final part of the original model. After the edge-side suffix finishes, the edge returns the final result to the user.

### 7.3 Runtime Responsibilities

The runtime does not perform partitioning or decide where encryption is required. Those decisions are made by the compiler through authority analysis, backward FHE tagging, and partitioning point optimization. The runtime instead follows the gener-

ated programs and metadata. Its main responsibilities are to distribute the correct keys to the correct devices, allocate and match communication queues, invoke the edge and cloud programs in the proper order, and provide the encrypted execution context required by the cloud.

This separation between compilation and runtime keeps online execution simple. Expensive compiler analyses, partition search, and code generation are performed offline. During online execution, the edge and cloud only execute the generated programs and follow the precomputed communication protocol. As a result, AION can provide privacy-preserving edge–cloud inference while avoiding runtime decisions about partition placement or encrypted-region construction.

# 8. Evaluation of AION

This work evaluates AION as a complete compiler and runtime stack for secure edge-cloud execution, covering the frontend interface, structured lowering from tensor programs to encrypted execution, authority-aware partitioning, and the generation of distributed execution code. The evaluation examines both performance and practicality through end-to-end latency, memory utilization, program correctness, partitioning-point analysis, cost-estimation accuracy, programmability, and real-world case studies.

## 8.1 Evaluation Setting

This work evaluates the current implementation of AION in a heterogeneous edge-cloud environment. The `Edge` platform is an Odroid N2 embedded board with ARM big-LITTLE architecture, consisting of ARM Cortex-A73 big cores and ARM Cortex-A53 LITTLE cores, with 4 GB DRAM. It serves as the client-side device that issues job requests and executes the edge-side program locally. The `Cloud` platform is a desktop server equipped with an Intel® Core™ i7-8700 processor and 64 GB DRAM. It executes the cloud-side programs and performs homomorphic encryption computation. The two platforms are connected through a local network with a peak bandwidth of 944 Mbps.

The implementation of AION is built on MLIR and generates LLVM intermediate representation code. For encrypted execution, AION uses the Microsoft SEAL library [54]

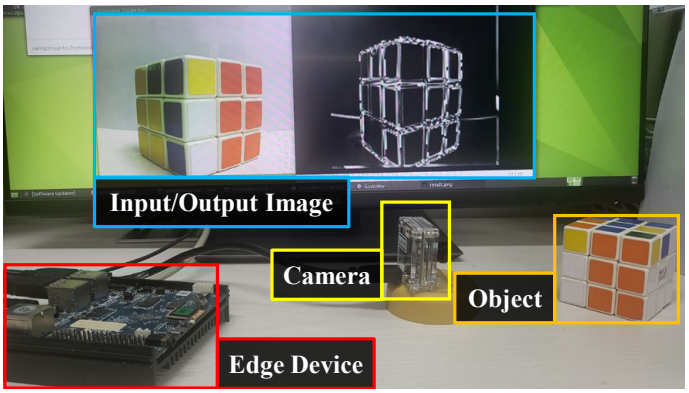



Figure 8.1: Evaluation setting for the Fault Detector scenario

and backend of Hecate [43] to compile FHE operations for the cloud server.

For the encrypted tensor experiments, the cloud-side encrypted execution is evaluated on a GPU-equipped server with an Intel® Core™ i7-12700 processor, 128 GB DRAM, and an NVIDIA RTX A6000 GPU. In this configuration, AION uses HEonGPU [57] as the FHE library and backend execution engine. The tensor experiments focus on the encrypted execution generated from high-level tensor programs, and compare AION with DaCapo [48], which supports bootstrapping-capable encrypted execution. Figure 8.1 shows the experimental setup for the Fault Detector benchmark scenario.

## 8.2 Benchmark Description

This work evaluates AION using two benchmark groups. The first group consists of six edge-cloud application benchmarks that represent practical privacy-preserving computing scenarios. The benchmark includes both machine learning and deep learning workloads, and is derived from widely used applications and prior FHE compiler studies, including EVA [41] and Hecate [43]. The second group consists of five neural-network

benchmarks that evaluate encrypted tensor lowering and GPU-backed FHE execution. These tensor benchmarks are compared with DaCapo [48].

**Edge-cloud application benchmarks.** The edge-cloud application benchmarks evaluate how AION selects encrypted regions, inserts communication and encryption operations, and executes the generated edge and cloud programs.

- **Fault Detector (Sobel Filter, SF).** This benchmark represents an image-based fault detection scenario in which the `Edge` device preprocesses an image and the `Cloud` computes the remaining part. It uses the Sobel Filter, which detects border lines by applying convolution with Sobel kernels and approximates the gradient of image intensity.

- **Temperature Prediction (Multi-Layer Perceptron, MLP).** This benchmark represents a sensor-data prediction scenario in which the `Edge` collects sequential measurements and the `Cloud` predicts future values. It uses a Multi-Layer Perceptron, a neural network widely used for approximating non-linear functions in time-series prediction tasks.

- **Animal Classifier (LeNet).** This benchmark represents an image classification scenario in which the `Edge` computes the former part of the network and the `Cloud` executes the latter part. It uses LeNet, a classical convolutional neural network composed of convolution, pooling, and fully connected layers.

- **Data Training with Linear Regression (LR).** This benchmark represents a distributed training scenario in which the `Edge` and `Cloud` independently process

separate data while collaboratively supporting privacy-aware learning. Linear Regression models linear relationships between input data and target values.

- **Data Training with Polynomial Regression (PR).** This benchmark extends the data-training scenario to polynomial relationships. Polynomial Regression models non-linear trends by fitting polynomial equations to the input data.

- **Data Training with Multivariate Regression (MR).** This benchmark represents a more complex distributed learning workload involving multiple predictor and response variables. Multivariate Regression captures richer relationships across multiple inputs and outputs.

**Encrypted tensor benchmarks.** The tensor benchmarks evaluate whether AION can lower larger neural-network programs into executable encrypted tensor computation. These benchmarks stress convolutional layers, element-wise operations, pooling, linear layers, tensor concatenation, and deep encrypted execution requiring bootstraps.

- **ResNet.** This benchmark represents a residual convolutional neural network. It includes convolutional blocks and skip connections, and evaluates whether encrypted tensor lowering can preserve tensor alignment across residual additions.

- **AlexNet.** This benchmark represents a convolutional image-classification network with stacked convolutional and fully connected layers. It provides a medium-sized tensor workload for encrypted convolution and linear-layer execution.

- **SqueezeNet.** This benchmark represents a compact convolutional network built

from Fire modules. It evaluates encrypted tensor execution across repeated small convolution blocks and channel-wise tensor composition.

- **MobileNet.** This benchmark represents a lightweight neural network designed for efficient inference. It stresses the compiler's ability to lower compact convolutional structures and pointwise tensor transformations into encrypted execution.

- **VGG16.** This benchmark represents a deep sequential convolutional network with repeated convolution and pooling blocks. It evaluates the scalability of encrypted tensor lowering and bootstrapping-capable execution for deeper neural-network workloads.

## 8.3 Performance Evaluation

This work evaluates the practical performance of AION in comparison with existing FHE compilation frameworks, including EVA [41] and Hecate [43]. The evaluation focuses on the generated edge-cloud executions and examines how much the proposed compilation of AION improves runtime behavior such as latency on cloud, communication overhead in realistic deployment settings. In particular, this work reports end-to-end latency, memory utilization, and program error, and discusses the practicality of the programming model through the benchmark implementations and generated executions. For the encrypted tensor benchmarks, AION is compared with DaCapo [48]. This part focuses on the tensor lowering path and evaluates whether AION can generate more efficient encrypted execution for larger neural-network workloads.

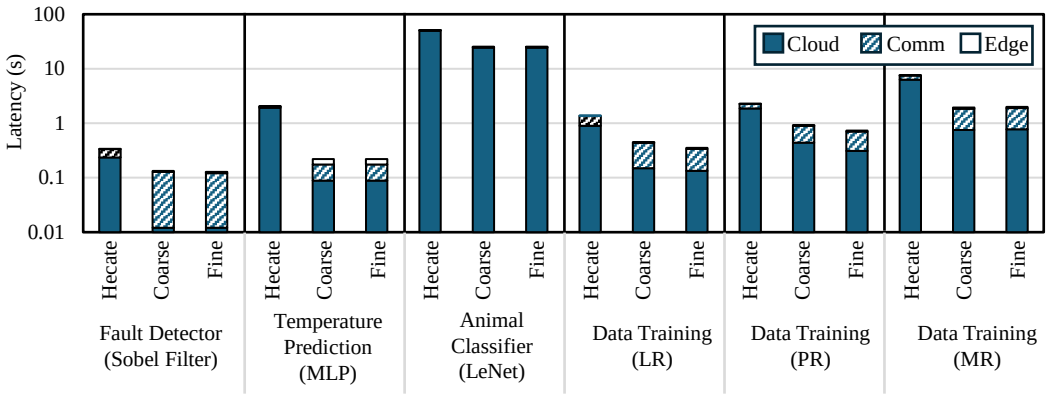



Figure 8.2: Latency of benchmarks about coarse-grained and fine-grained partitioning under the performance governor

### 8.3.1 End-to-end Latency

This work presents the end-to-end latency results of AION from three viewpoints. The first viewpoint is the effect of partitioning granularity, in which developers explicitly specify authority regions and manually shape the partition boundary at coarse-grained or fine-grained levels. The second viewpoint is automatic authority-aware partitioning, in which authority is assigned to data and the compiler determines the encrypted execution and optimized partitioning behavior automatically. The third viewpoint is encrypted tensor execution, in which high-level tensor programs are lowered into FHE primitive execution and compared with a bootstrapping-capable FHE compiler. In all results, end-to-end latency includes edge-side computation time, cloud-side computation time, and communication time between the two devices.

**Effect of partitioning granularity** This part evaluates the latency of manually guided partitioning using explicit authority regions. The comparison includes Hecate [43], which

executes the program as a fully encrypted cloud program without code partitioning, a coarse-grained partitioning scheme in which the application is divided at manually chosen large regions, and a fine-grained partitioning scheme in which the developer specifies more precise authority regions through the frontend annotations of AION.

Figure 8.2 shows the end-to-end latency results across the six benchmarks. The latency includes edge execution time, cloud execution time, and communication time for each benchmark. Both partitioned executions show clear improvements over the Hecate baseline. The coarse-grained scheme achieves a 3.32× geomean speedup over Hecate, while the fine-grained scheme achieves a 3.61× geomean speedup. Although the `Edge` device is much weaker than the `Cloud` server in raw computing capability, moving selected operations to the edge reduces the number of expensive homomorphic operations performed in the cloud. Since homomorphic computation dominates the execution cost, the reduction leads to a substantial decrease in overall latency.

The fine-grained result is also 1.09× faster than the coarse-grained result on geomean. The improvement comes from the ability of the frontend annotations to describe cloud execution regions more precisely, allowing the developer to control communication volume and partition placement more carefully. In Multi Layer Perceptron and LeNet, the coarse-grained and fine-grained schemes show the same latency because both schemes effectively choose the same partition points. In Sobel Filter and Multivariate Regression, the two schemes choose slightly different but nearby partition positions, which leads to similar end-to-end latency.

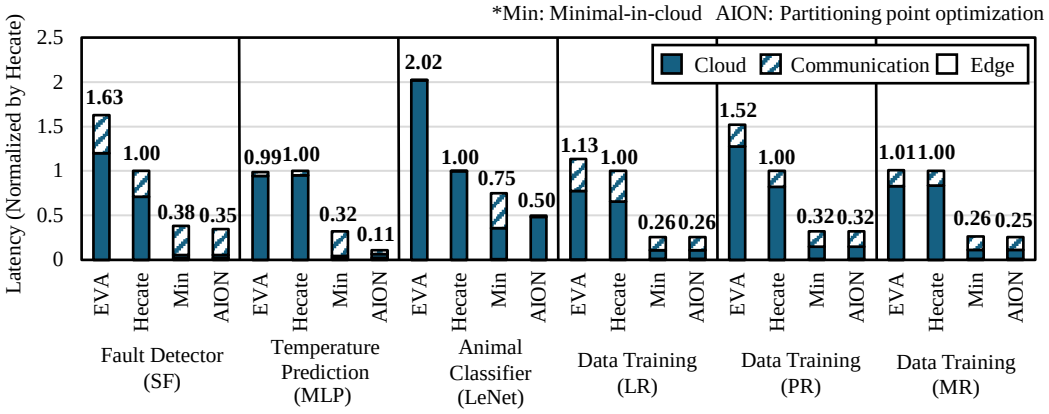



Figure 8.3: End-to-end latencies of AION (Figure 1.2d and Figure 6.11b) compared to the existing FHE compilers such as EVA and Hecate, and a different partitioning policy which is the minimal-in-cloud policy (Figure 1.2c and Figure 6.11a). Each latency is normalized to the end-to-end latency of Hecate, and stacked with the computation latency at the edge and cloud devices, as well as communication latency.

**Automatic Authority-Aware Partitioning** This part evaluates the authority-aware compilation of AION when authority is assigned to data rather than to explicit execution regions. AION compiler derives the encrypted execution path from the data authority, constructs the partitioned edge-cloud program, and applies cost-aware partitioning-point optimization. The comparison includes EVA [41] and Hecate [43], which follow the conventional all-in-cloud(Figure 1.2b), and with the minimal-in-cloud (Figure 1.2c and Figure 6.11a), which is derived from backward FHE tagging without partitioning-point optimization, while AION compiler policy applies cost-aware partitioning optimization(Figure 1.2d and Figure 6.11b).

Figure 8.3 shows the normalized end-to-end latency results. Each latency is normalized to Hecate and includes edge-side computation time, cloud-side computation time, and communication time. The minimal-in-cloud policy reduces the latency substan-

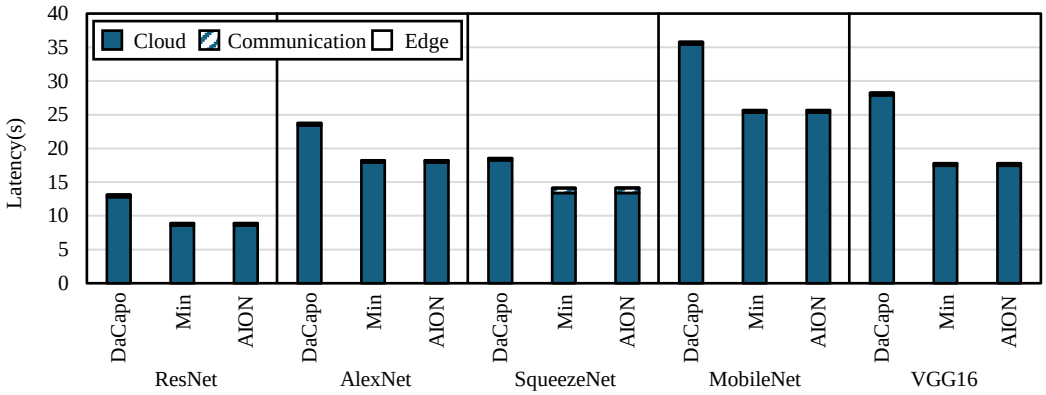



Figure 8.4: End-to-end latency of encrypted tensor benchmarks compared with DaCapo [48].

tially, achieving 3.76× geomean speedup over EVA and 2.81× geomean speedup over Hecate. The result shows that minimizing homomorphic execution in the cloud is highly effective even when part of the computation is moved to a weaker edge device.

The optimized authority-aware compilation of AION improves the result further. Compared with the minimal-in-cloud policy, it achieves an additional 1.31× geomean speedup, corresponding to 4.92× and 3.69× geomean speedups over EVA and Hecate, respectively. The gap between these two results reflects the benefit of partitioning-point optimization. In several benchmarks, the optimized partition may increase the amount of cloud-side homomorphic computation, but it reduces communication time enough to lower the overall end-to-end latency. The effect is especially clear in Multi Layer Perceptron and LeNet. For example, the optimized compilation path achieves a 2.96× speedup over the minimal-in-cloud policy for Multi Layer Perceptron. In Linear Regression and Polynomial Regression, the optimized result is identical to the minimal-in-cloud result because the minimal-in-cloud placement is already the estimated best partition.

**Encrypted Tensor Execution** This part evaluates the encrypted tensor lowering of AION on larger neural-network workloads. The comparison uses DaCapo [48] as the baseline because DaCapo supports bootstrapping-capable encrypted execution. As in the edge-cloud partitioning evaluation, this experiment also includes the minimal-in-cloud policy as a reference partitioning policy. The benchmark set consists of ResNet, AlexNet, SqueezeNet, MobileNet, and VGG16. These models include convolutional layers, pooling layers, linear layers, element-wise operations, and deeper execution paths that stress encrypted tensor lowering.

Figure 8.4 shows the end-to-end latency of the encrypted tensor benchmarks. AION reduces the latency for all five models and achieves a 1.41× geomean speedup over DaCapo. The minimal-in-cloud policy and AION show the same latency in these tensor benchmarks because the cost-guided partitioning of AION selects the same partitioning point as the minimal-in-cloud policy. The result shows that the encrypted tensor lowering of AION can generate more efficient FHE execution not only for small operators, but also for larger neural-network workloads with convolutional, pooling, linear, and element-wise layers.

The improvement mainly comes from reducing the cloud-side encrypted execution time. Across the five tensor benchmarks, AION achieves a 1.43× geomean speedup in cloud-side execution. Since the edge-side computation time is identical for the two systems and communication accounts for a small portion of the total latency in most benchmarks, the reduction in encrypted cloud execution dominates the end-to-end improvement. SqueezeNet is the only case where AION increases communication time, but the reduced cloud-side execution time still improves the total latency.

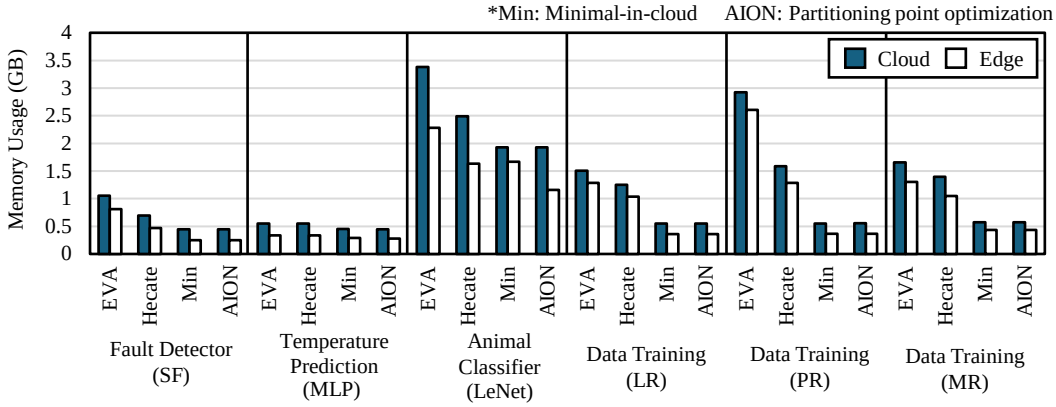
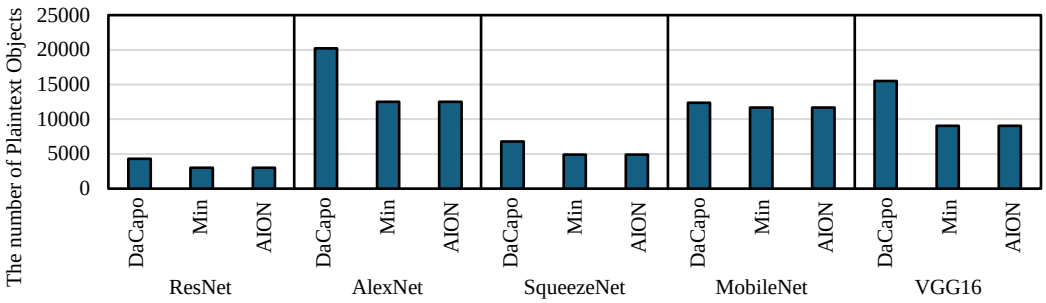



Figure 8.5: Memory usages of the partitioning point optimization of AION (Figure 1.2d and Figure 6.11b) compared to the existing FHE compilers such as EVA and Hecate, and a different partitioning policies which is the minimal-in-cloud policy (Figure 1.2c and Figure 6.11a).

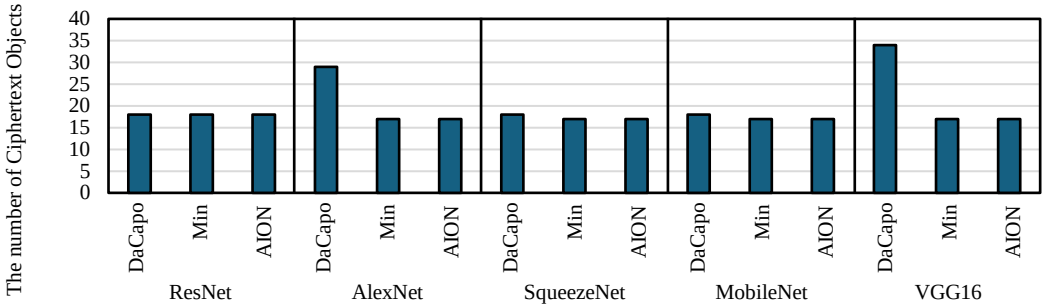
### 8.3.2 Memory Utilization

Figure 8.5 illustrates memory usages of EVA, Hecate, the minimal-in-cloud policy and the optimized result of AION. Both the minimal-in-cloud and the partitioning point optimization reduce memory usage in the edge and the cloud. Specifically, compared to EVA and Hecate, optimized partitioning point of AION reduces edge memory usage by 65.5% and 51.7%, and cloud memory usage by 59.1% and 45.8%, respectively. The program partitioning reduces the depth of multiplication on a ciphertext, leading to a lower maximum coefficient modulus $Q$. This characteristic enables the edge to utilize a smaller key set due to the lower coefficient modulus $Q$, resulting in lower memory usage. Additionally, the ciphertexts with lower level $l$ have smaller sizes, reducing the memory usage of both the edge and the cloud.

For the encrypted tensor benchmarks, this work evaluates memory-related behav-




(a) Number of plaintext objects used by encrypted tensor benchmarks.




(b) Number of ciphertext objects used by encrypted tensor benchmarks.

Figure 8.6: Number of plaintext and ciphertext objects used by encrypted tensor benchmarks.

ior using the number of plaintext and ciphertext objects. The tensor experiments use a different cloud-side execution configuration and a GPU-backed FHE library, so the object count provides a backend-independent view of the encrypted execution generated by the compiler. A lower number of plaintext objects indicates fewer encoded constants or plaintext intermediates managed by the backend, while a lower number of ciphertext objects indicates fewer encrypted tensor packs or encrypted intermediates.

Figure 8.6 shows the number of plaintext and ciphertext objects used by DaCapo and AION. AION reduces both types of objects across the tensor benchmark suite, reducing

Table 8.1: RMS Error of edge-cloud application benchmarks

| Benchmark | EVA | Hecate | Min | AION |
|---|---|---|---|---|
| Fault Detector (SF) | 6.915E-06 | 2.264E-04 | 3.247E-06 | 3.305E-06 |
| Temperature Prediction (MLP) | 3.864E-03 | 4.025E-03 | 4.180E-03 | 4.124E-03 |
| Animal Classifier (LeNet) | 7.497E-04 | 5.869E-03 | 7.366E-03 | 7.252E-03 |
| Data Training (LR) | 7.238E-06 | 3.747E-04 | 1.697E-05 | 2.574E-05 |
| Data Training (PR) | 2.020E-04 | 2.419E-04 | 1.948E-04 | 4.708E-04 |
| Data Training (MR) | 2.339E-03 | 2.140E-03 | 2.221E-03 | 2.383E-03 |
| Gmean | 2.023E-04 | 1.006E-03 | 3.004E-04 | 2.566E-04 |

Table 8.2: RMS error of encrypted tensor benchmarks

| Benchmark | DaCapo | Min | AION |
|---|---|---|---|
| ResNet | 9.440E-04 | 9.360E-04 | 9.360E-04 |
| AlexNet | 3.647E-03 | 1.730E-04 | 1.730E-04 |
| SqueezeNet | 5.980E-04 | 4.430E-04 | 4.430E-04 |
| MobileNet | 2.250E-02 | 8.810E-04 | 8.810E-04 |
| VGG16 | 9.140E-04 | 8.930E-04 | 8.930E-04 |
| Gmean | 2.115E-03 | 5.627E-04 | 5.627E-04 |

plaintext objects by 30.6% and ciphertext objects by 26.5% in total. This result shows that the encrypted tensor lowering of AION generates a more compact encrypted execution, in addition to reducing the end-to-end latency.

The reduction is more visible in models where the baseline generates many encoded constants or encrypted intermediate objects. In particular, AlexNet and VGG16 show clear reductions in ciphertext objects, while ResNet keeps the same ciphertext count but still reduces plaintext objects. These results indicate that AION reduces memory-related pressure by simplifying the generated encrypted tensor execution rather than relying only on faster backend execution.

### 8.3.3 Program Error

Since FHE is a kind of approximate computing, the computation output has some errors depending on the ciphertext parameters. Table 8.1 and Table 8.2 present the root-mean-square error of the partitioned programs compared to the plain programs. Although the error values are slightly changed, since the partitioning algorithm does not alter the structure of DNN model and parameters, the authority-driven compilation and partitioning point optimization of AION maintains the model accuracy.

## 8.4 Partitioning Point

To evaluate the effectiveness of the partitioning points that AION compiler finds, this work measures the latency of the partitioned programs for all the possible partitioning points. Figure 8.7 shows the end-to-end latency of each benchmark for different partitioning points. The x-axis illustrates the partitioning points from the beginning to the end of available partitioning points. `0` refers to the partitioning point of the all-in-cloud policy, and the largest number refers to the partitioning point of the minimal-in-cloud policy. The partitioning points that AION compiler chooses are marked with triangles.

As Figure 8.7 illustrates, AION compiler successfully chooses the partitioning point with the minimum estimated latency, reflecting the computation and communication overheads. Especially for Sobel Filter, Multi Layer Perceptron, LeNet, and Multivariate Regression programs, AION compiler chooses different partitioning points compared to the minimal-in-cloud policy because reducing estimated communication overheads is more beneficial to minimizing computation overheads. In other words, reflecting the

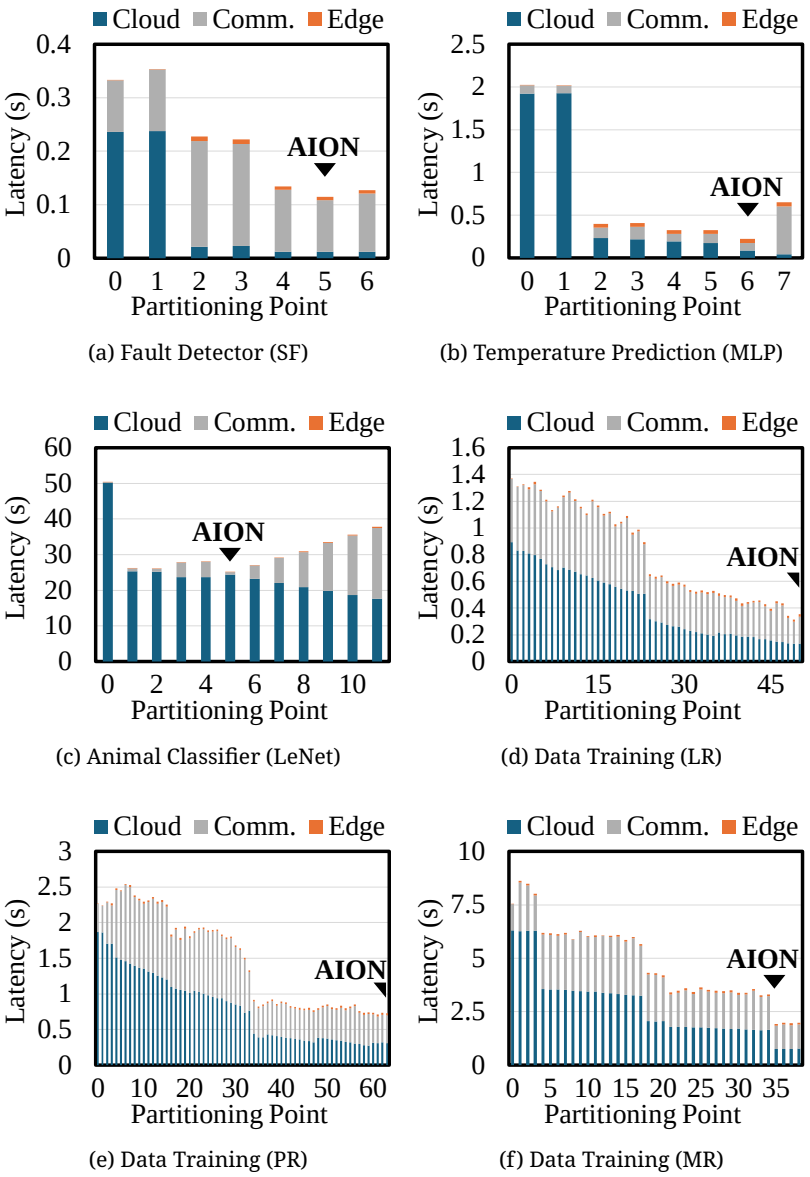



Figure 8.7: End-to-end latency depending on partitioning points. The x-axis orders candidate partitioning points from the all-in-cloud policy to the minimal-in-cloud policy. The triangle marks the partitioning point selected by AION.

communication overheads is crucial to fully optimizing the FHE-based privacy-preserving cloud computing. On the other hand, for Linear Regression and Polynomial Regression, AION compiler chooses the same partitioning points to the minimal-in-cloud policy because their estimated latency is minimal.

## 8.5 Accuracy of Cost Estimation

Since AION compiler highly relies on the estimated latency to choose optimal partitioning points, precise latency estimation is crucial for the compiler to find better partitioning points. This work evaluates the accuracy of the latency estimation by comparing the estimated latency and the actual latency for each benchmark. Figure 8.8 shows the error rate between the estimated and actual execution. Across all benchmarks, the average error in estimating computation latency is 9.53%, while the average error in estimating communication latency is 16.47%. The computation latency estimation is relatively accurate for the edge-cloud benchmarks, where the compiler estimates FHE operation latency from profiled operation costs and statically known ciphertext levels. The tensor benchmarks show larger computation-side errors because they execute deeper encrypted workloads on a GPU-backed FHE backend, where kernel scheduling, memory movement, and backend-level execution behavior add runtime overheads.

The communication latency estimation has a larger error than computation latency estimation. Communication time is affected by the transferred ciphertext size, which can be estimated from the ciphertext level and number of ciphertexts, and by runtime network behavior, which may fluctuate during execution. Therefore, communication latency has a larger gap between estimation and measurement. Even with this varia-

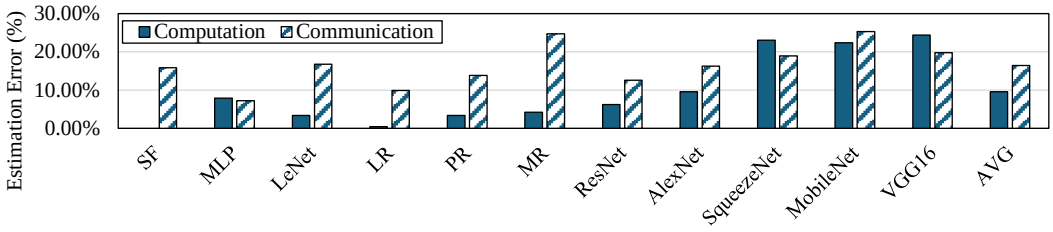



Figure 8.8: Error of latency estimation based on measurement in terms of computation and communication.

tion, the estimation is accurate enough to distinguish the major cost differences among partitioning candidates and guide the partitioning-point selection of AION.

## 8.6 Programmability

This work evaluates the programmability benefit of AION from the perspective of developer burden. In a manual implementation, developers must decide which part of the program should be executed with FHE, insert encryption and decryption operations, manage communication between the edge and the cloud, and express tensor operations as executable FHE operations. These tasks require FHE-specific knowledge as well as distributed execution logic.

Table 8.3 summarizes the developer tasks required by different implementation approaches. In manual implementation, the developer is responsible for both the partitioned edge-cloud execution and the FHE implementation. With Hecate [43], scale management and FHE-level optimization are automated by the backend compiler, but the developer still needs to select the encrypted region, insert encryption and decryption operations, and write the communication logic manually. For tensor workloads, DaCapo [48] reduces the burden of constructing encrypted tensor computation by pro-

Table 8.3: Developer tasks required for FHE-enabled edge-cloud execution.

| **Developer task** | **Manual** | **Hecate [43]** | **DaCapo [48]** | **AION** |
|---|---|---|---|---|
| Frontend Interface | FHE Primitives | FHE Primitives | Custom DSL | PyTorch-Style |
| Select Encrypted Region | Manual | Manual | Manual | Auto |
| Choose Partition Boundary | Manual | Manual | Manual | Cost-guided |
| Insert Encryption/Decryption | Manual | Semi | Semi | Auto |
| Insert Communication | Manual | Manual | Manual | Auto |
| Manage Scale and Level | Manual | Auto | Auto | Auto |
| Generate Edge/Cloud Programs | Manual | Manual | Manual | Auto |

viding tensor-level compilation support, but developers still need additional wrapper code and the system does not address authority-aware edge-cloud partitioning. In contrast, AION allows the developer to express the tensor program and privacy intent at the source level, and the compiler generates the FHE-enabled edge-cloud execution.

The table shows that the main programmability benefit of AION is not limited to reducing the number of lines written by the developer. AION removes several error-prone tasks that are specific to privacy-preserving edge-cloud execution. In particular, developers do not need to manually derive the FHE-required region from privacy constraints, insert communication operations at the partition boundary, or rewrite supported tensor operators into ciphertext-level operations.

To complement this task-level comparison, Table 8.4 compares the line of code of the original code, manual FHE implementation, manual implementation with Hecate [43], and annotated code with AION for edge-cloud benchmarks. Table 8.5 further compares the line of code for tensor benchmarks, including DaCapo with wrapper code. These LoC results should be interpreted as secondary indicators of programmability, while the primary benefit of AION is the removal of the manual tasks summarized in Table 8.3.

Table 8.4: Line-of-code comparison for edge-cloud benchmarks.

| Benchmark | Original | Manual | Manual +Hecate [43] | AION |
|---|---|---|---|---|
| Fault Detector (SF) | 25 | 54 | 39 | 28 |
| Temperature Prediction (MLP) | 64 | 90 | 81 | 66 |
| Animal Classifier (LeNet) | 266 | 387 | 280 | 268 |
| Data Training (LR) | 48 | 102 | 76 | 52 |
| Data Training (PR) | 44 | 122 | 75 | 49 |
| Data Training (MR) | 53 | 123 | 88 | 58 |

Table 8.5: Line-of-code comparison for tensor benchmarks.

| Benchmark | Original (Plain) | Manual | DaCapo [48] (with Wrapper) | AION |
|---|---|---|---|---|
| ResNet | 98 | 12791 | 107 | 60 |
| AlexNet | 65 | 80369 | 148 | 39 |
| SqueezeNet | 100 | 21385 | 320 | 69 |
| MobileNet | 74 | 46027 | 116 | 76 |
| VGG16 | 112 | 65537 | 236 | 128 |

For the edge-cloud benchmarks in Table 8.4, the manual implementation requires more code than the original program because the developer must write communication, encryption, decryption, and FHE operation management code. Hecate reduces part of this burden by automating scale management and FHE-level optimization. However, the developer still manually selects the FHE-executed region and writes the communication, encryption, and decryption logic required for privacy-preserving edge-cloud execution. In contrast, the AION-annotated code remains close to the original program because the compiler derives the partitioned execution from source-level privacy intent.

For the tensor benchmarks in Table 8.5, the manual implementation requires substantially more code because the developer must explicitly construct ciphertext-level tensor execution. DaCapo significantly reduces this burden by providing tensor-oriented

FHE compilation support with wrapper code. AION further keeps the source program close to the original plain tensor program by combining tensor-level lowering with authority-aware partitioning and edge-cloud code generation.

Overall, the programmability results show two complementary benefits. For edge-cloud workloads, AION reduces the need to write partitioning, communication, and encryption code manually. For tensor workloads, AION reduces the need to manually express tensor semantics at the ciphertext level. These results support the claim that AION improves programmability not simply by reducing LoC, but by removing FHE-specific and edge-cloud-specific implementation tasks from the developer.

## 8.7 Case Study

This work evaluates AION under practical edge-cloud conditions. Real deployments may use lower-bandwidth wireless networks, reduced edge-side CPU frequency, and sensor-driven workloads with different data complexity. This work examines how network condition, edge CPU governor, and healthcare monitoring scenarios affect latency, partitioning behavior, and practicality of FHE-enabled edge-cloud execution.

### 8.7.1 Network Condition

This work evaluates AION on a local network with a wired connection whose maximum bandwidth is 944Mbps. It also evaluates two 802.11/n WiFi settings, a 2.4GHz network with a maximum bandwidth of 60.4Mbps and a 5GHz network with a maximum bandwidth of 88.5Mbps. Figure 8.9 shows the end-to-end latency under these network conditions. Each result is normalized to the wired-network latency of the same benchmark.

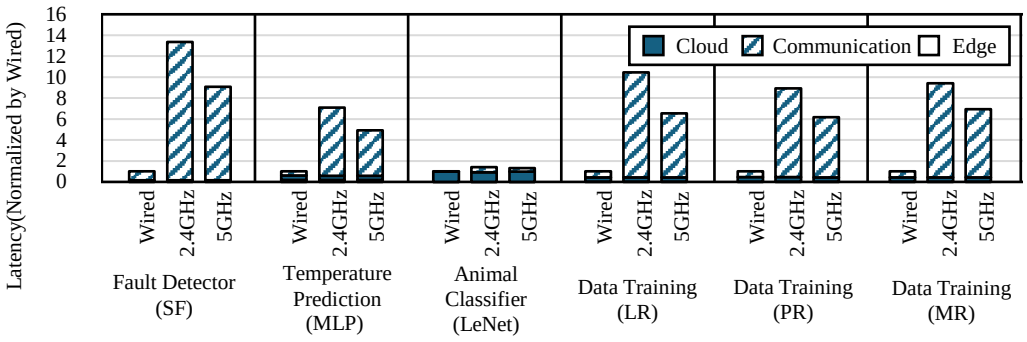



Figure 8.9: Performance comparison result with 2.4GHz and 5GHz 802.11/n WiFi network, as well as wired network. Each latency is normalized to the end-to-end latency of wired network, and stacked with the computation latency at the edge and cloud devices and communication latency.

Lower bandwidth increases end-to-end latency because ciphertext communication becomes slower. Compared with the wired network, the geometric-mean latency becomes 7.04 times slower on the 2.4GHz WiFi network and 5.05 times slower on the 5GHz WiFi network. The increase mainly appears in the communication component, while the edge and cloud computation components remain unchanged because the same edge and cloud devices execute the generated programs. This result shows that network bandwidth is an important deployment factor for FHE-enabled edge-cloud execution.

The impact of bandwidth reduction differs across benchmarks. In LeNet, the end-to-end latency increases by only 1.42 times on the 2.4GHz network and 1.33 times on the 5GHz network. In this benchmark, computation accounts for a large portion of total latency, so increased communication latency has a smaller effect on the normalized end-to-end latency. For the other benchmarks, the normalized slowdown is larger because communication accounts for a larger fraction of end-to-end latency under WiFi. These results show that lower bandwidth can increase communication cost and reduce

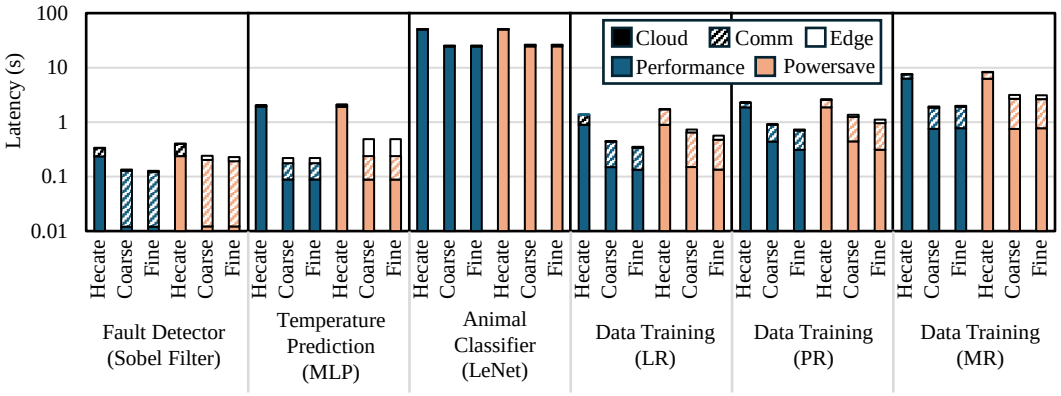



Figure 8.10: Latency of benchmarks under performance and powersave CPU governors

the benefit of AION's edge-cloud collaboration. When AION is integrated into a real-world IoT system, edge devices may communicate with a local proxy over WiFi, while the proxy communicates with the cloud to mitigate communication overhead.

### 8.7.2 Impact of Edge CPU Governor

This work compares latency under two Edge CPU governors, the performance governor and the powersave governor. The performance governor allows the Edge device to run at a higher frequency for maximal computation capability, whereas the powersave governor reduces the operating frequency to save energy at the cost of higher latency.

Figure 8.10 displays the latency result of performance governor and powersave governor. Under the performance governor, the fine-grained scheme achieves 3.61× speedup over Hecate and 1.09× speedup over the coarse-grained scheme. Under the powersave governor, the fine-grained scheme still achieves 2.55× speedup over Hecate and the same 1.09× speedup over the coarse-grained scheme. The overall speedup over Hecate becomes smaller in the powersave setting because lower edge frequency

increases both local computation time and communication-related overhead. Nevertheless, the relative gain of fine-grained partitioning over coarse-grained partitioning remains consistent, indicating that the benefit of precise partition control is robust even when the Edge device operates with reduced computing power.

Figure 8.11 shows the end-to-end latency of each benchmark for different partitioning points under the powersave governor. Compared with the performance-governor result in Figure 8.7, the latency generally increases because the lower frequency of the Edge device enlarges the cost of local execution. This change also affects the relative advantage of each partitioning point, since partitioning points that assign more computation to the edge become less favorable in the powersave setting. Even under this changed condition, however, AION compiler still chooses partitioning points near the minimum-latency points of the benchmarks. This result shows that the partitioning-point optimization of AION effectively reflects the execution environment as well as the computation and communication overheads.

### 8.7.3 Integrating in Healthcare Monitoring Systems

To demonstrate its practical implications, this work evaluates the authority-driven compilation of AION in IoT applications. Based on equations for disease diagnosis [101, 102, 103, 104, 105], this work implements two kinds of smart healthcare monitoring applications using a wearable device such as Apple Watch series 8 (Figure 8.12) that continuously monitors vital signs of a user. The basic vital signs monitoring application utilizes the patient's personal information such as age, height and weight, and collected data such as heart rate and blood oxygen level. The enhanced application includes ad-

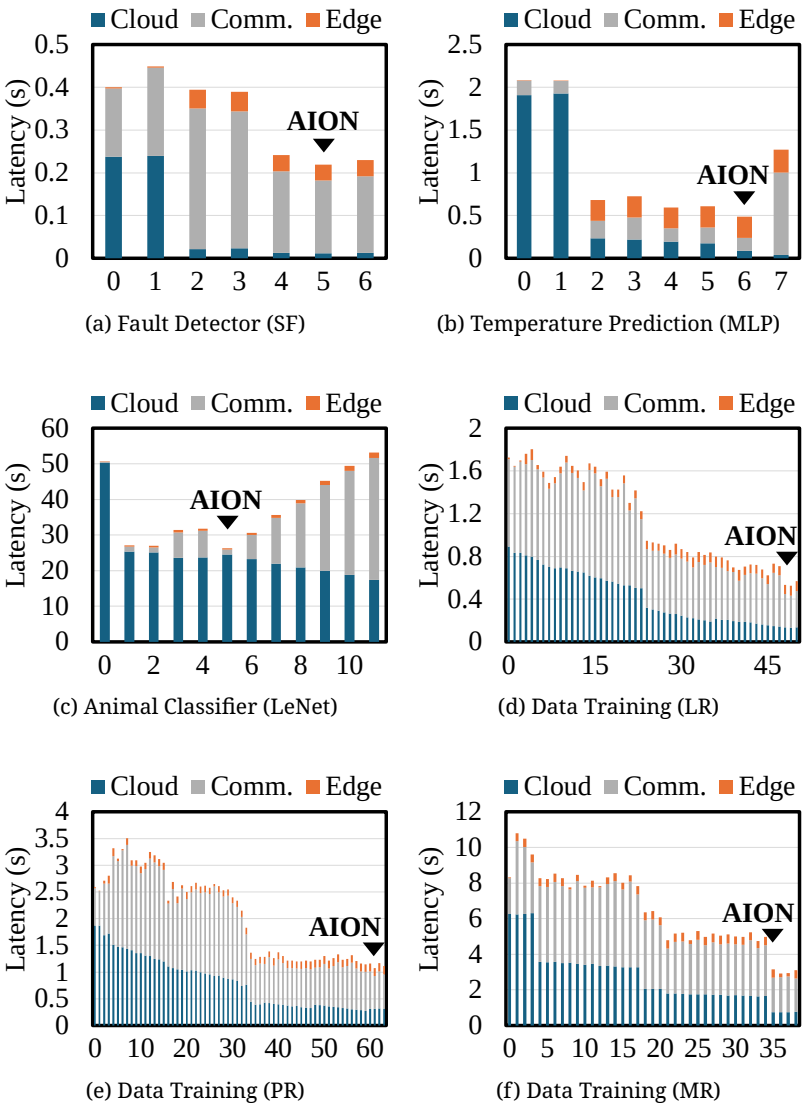



Figure 8.11: End-to-end latency depending on partitioning points with powersave governor on edge device. The x-axis shows the partitioning point from the beginning of the all-in-cloud policy to the end of the minimal-in-cloud policy. The partitioning point that AION compiler chooses is marked with a triangle.

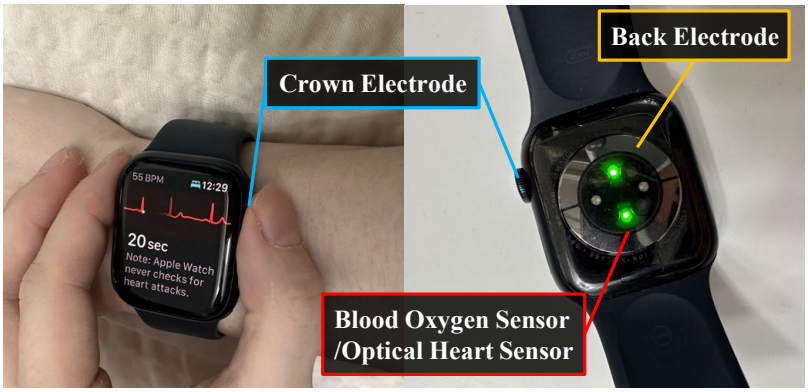



Figure 8.12: Apple Watch Series 8 and its sensors used in the case study

ditional measurements such as an electrocardiogram. This extension shows how well the authority-driven compilation of AION handles increased data complexity.

As an edge device, the wearable device performs preliminary data processing to filter out noise while collecting real-time health data. The data from sensors are sent to a local edge proxy, implemented on the Odroid N2. The edge proxy performs initial analysis to detect abnormal patterns and preprocesses the data.

The authority-driven compilation of AION automatically manages the data according to the annotated privacy authority and partitions the program. Existing compilers like EVA [41] and Hecate [43] adopt the all-in-cloud policy (Figure 1.2b), and Min adopts the minimal-in-cloud policy (Figure 1.2c and Figure 6.11a).

Table 8.6 shows performance, memory usage, and the RMS error of the cases. Latency measurements indicate that the authority-driven compilation of AION achieves the lowest latency by optimizing the computational overhead. For the basic vital signs monitoring application (Table 8.6a), AION achieves a total latency of 685.42ms, which is 2.77× faster than EVA [41]. In memory utilization, edge memory usage is reduced

Table 8.6: Performance in Case Study using Health Data

(a) Case 1: Basic Vital Signs Monitoring

| Compiler | Latency (ms) | | | Mem. Usage (GB) | | Error |
|---|---|---|---|---|---|---|
| | Edge | Cloud | Comm. | Edge | Cloud | |
| EVA | 5.33 | 584.83 | 1312.60 | 2.22 | 2.49 | 3.98E-05 |
| Hecate | 4.32 | 416.36 | 1179.07 | 1.29 | 1.53 | 4.14E-04 |
| Min | 6.63 | 76.60 | 1104.23 | 0.63 | 0.84 | 1.21E-04 |
| AION | 4.43 | 290.84 | 390.15 | 1.03 | 1.27 | 2.72E-05 |

(b) Case 2: Enhanced Monitoring with Electrocardiogram

| Compiler | Latency (ms) | | | Mem. Usage (GB) | | Error |
|---|---|---|---|---|---|---|
| | Edge | Cloud | Comm. | Edge | Cloud | |
| EVA | 6.12 | 1126.31 | 2039.47 | 2.27 | 2.51 | 2.79E-06 |
| Hecate | 5.23 | 775.27 | 1681.52 | 1.29 | 1.55 | 1.88E-04 |
| Min | 7.39 | 215.94 | 2511.93 | 0.64 | 0.86 | 6.48E-06 |
| AION | 6.52 | 583.83 | 1679.13 | 1.29 | 1.54 | 5.60E-06 |

from 2.22GB to 1.03GB (46.4%), and cloud memory usage is reduced from 2.49GB to 1.27GB (51.0%). For the enhanced monitoring application with electrocardiogram data (Table 8.6b), AION achieves a total latency of 2269.49ms, which is 1.396× faster than EVA. Edge memory usage is reduced from 2.27GB to 1.29GB (58.1%), and cloud memory usage is reduced from 2.51GB to 1.54GB (61.6%). The root-mean-square errors in both cases indicate that AION does not affect the model accuracy. In addition, the results show that AION effectively supports the increased number of sensors.

# 9. Discussion

AION currently uses Hecate [43] as the FHE back-end compiler. The authority analysis and partition selection method, however, is independent of a specific FHE compiler backend. A different backend, such as EVA [41], can be used if it provides the required FHE operation generation path and cost profiles. In this sense, the partitioning method is complementary to existing FHE compilers. Existing FHE compilers generate encrypted computation for a given FHE program, while AION determines which part of an edge-cloud program should be encrypted and where communication should occur.

The same authority-based formulation is also useful for IoT deployment scenarios. IoT systems often include constrained sensor devices, local gateways, proxy devices, and cloud services. Previous systems such as CryptDB [106] and fog-computing scenarios [107, 108] show that an intermediate trusted device can help manage data before it reaches the cloud. In such a setting, AION can represent the edge device and a trusted proxy as belonging to the same plaintext-authorized region. The developer can annotate data with edge authority, and the compiler can preserve the rule that only devices in the trusted edge-side region access the data in plaintext. The cloud can still participate in computation through FHE when a value crosses the trust boundary.

This deployment model is especially relevant when small IoT devices cannot afford the full cost of encryption, decryption, or local preprocessing. A trusted proxy can per-

form these tasks on behalf of the sensor device, while AION still reasons about privacy at the level of data authority. The developer specifies which values are edge-private, cloud-private, shared, or inaccessible in plaintext, and the compiler inserts the required encryption and communication operations for the generated edge and cloud programs.

A practical limitation of the current formulation is that AION assumes two execution authorities, `Edge` and `Cloud`, and a single-key FHE execution model. Real IoT deployments may involve multiple mutually untrusted edge devices, several cloud services, or a proxy that is trusted for some data but not for others. Supporting such settings would require extending the authority type to represent more than two parties and connecting the compiler to FHE schemes or protocols that support the corresponding key structure. The current design provides a natural starting point for this extension because authority is already represented as a set of devices allowed to access each value in plaintext.

# 10. Conclusion

This dissertation presented AION, an end-to-end authority-aware compiler for privacy-preserving edge-cloud tensor execution. Edge-cloud execution is widely used because it combines data collected near users with cloud-side computation. However, privacy-preserving execution becomes difficult when user data and service-side model parameters belong to different privacy authorities. Fully homomorphic encryption makes it possible to compute on encrypted data, but applying FHE to the entire program introduces large computation and memory overhead. Practical deployment therefore requires a compiler that can determine where encrypted execution is needed, translate tensor computation into ciphertext-level execution, and generate edge-cloud programs.

AION addresses this problem by allowing developers to express tensor computation together with privacy intent at the source level. The frontend provides a tensor DSL with placement annotations and authority annotations. From the annotated program, AION analyzes how privacy authority propagates through data dependencies and identifies the values that must remain protected during edge-cloud execution. This analysis allows the compiler to preserve plaintext execution where it is legally allowed, while introducing encrypted execution only for values that require protection.

AION also supports the execution of protected tensor computation over FHE. Tensor programs are written with high-level operators, while FHE libraries provide primitive

operations over ciphertext slots. AION bridges this gap by lowering protected tensor computation into FHE primitive operations using tensor-to-slot metadata. This lowering process makes the relationship between tensor values, ciphertext packs, and FHE operations explicit in the compiler, reducing the need for developers to manually rewrite tensor operators into ciphertext-level code.

Based on the authority analysis and encrypted tensor lowering, AION selects an edge-cloud partitioning plan using a cost model that considers encrypted computation and ciphertext communication. The compiler inserts the required encryption, decryption, and communication operations at the selected boundary and emits coordinated edge and cloud programs. The runtime then supports deployment by managing keys, program artifacts, and online collaborative execution between the edge and cloud.

The evaluation showed that AION improves the efficiency and programmability of privacy-preserving edge-cloud tensor execution. AION achieved 4.92× and 3.69× geometric-mean speedups over EVA and Hecate, respectively, and reduced edge and cloud memory usage by up to 65.5% and 59.1%. AION also reduced developer burden by replacing manual encrypted-region selection, communication insertion, encryption and decryption management, and tensor-to-FHE primitive rewriting with source-level annotations and compiler-generated edge-cloud programs. These results show that authority-aware partitioning and encrypted tensor lowering should be addressed together in a compiler flow to make FHE-enabled edge-cloud tensor execution practical.

## References


[1] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China: Association for Computing Machinery, 2017, pp. 615–629, ISBN: 9781450344654.

[2] M. Gao, W. Cui, D. Gao, R. Shen, J. Li, and Y. Zhou, "Deep neural network task partitioning and offloading for mobile edge computing," in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1–6.

[3] H.-J. Jeong, I. Jeong, H.-J. Lee, and S.-M. Moon, "Computation offloading for machine learning web apps in the edge server environment," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2018, pp. 1492–1499.

[4] H. Choi and I. V. Bajić, "Deep feature compression for collaborative object detection," in *2018 25th IEEE International Conference on Image Processing (ICIP)*, IEEE, 2018, pp. 3743–3747.

[5] H. Li, K. Ota, and M. Dong, “Learning iot in edge: Deep learning for the internet of things with edge computing,” *IEEE network*, vol. 32, no. 1, pp. 96–101, 2018.

[6] S. Itahara, T. Nishio, and K. Yamamoto, “Packet-loss-tolerant split inference for delay-sensitive deep learning in lossy wireless networks,” in *2021 IEEE Global Communications Conference (GLOBECOM)*, IEEE, 2021, pp. 1–6.

[7] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan, “When edge meets learning: Adaptive control for resource-constrained distributed machine learning,” in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 63–71.

[8] L. Zeng, E. Li, Z. Zhou, and X. Chen, “Boomerang: On-demand cooperative deep neural network inference for edge intelligence on the industrial internet of things,” *IEEE Network*, vol. 33, no. 5, pp. 96–103, 2019.

[9] P. Ren, X. Qiao, Y. Huang, L. Liu, C. Pu, and S. Dustdar, “Fine-grained elastic partitioning for distributed dnn towards mobile web ar services in the 5g era,” *IEEE Transactions on Services Computing*, vol. 15, no. 6, pp. 3260–3274, 2022.

[10] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane, “Spinn: Synergistic progressive inference of neural networks over device and cloud,” in *Proceedings of the 26th Annual International Conference on Mobile Computing*

*and Networking (MobiCom)*, London, United Kingdom: Association for Computing Machinery, 2020, ISBN: 9781450370851.

[11] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, "Secure program partitioning," *ACM Trans. Comput. Syst.*, vol. 20, no. 3, pp. 283–328, Aug. 2002.

[12] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *USENIX Security Symposium*, vol. 57, 2004.

[13] Y. Wu, J. Sun, Y. Liu, and J. S. Dong, "Automatically partition software into least privilege components using dynamic data dependency analysis," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2013, pp. 323–333.

[14] W. Qiang and H. Luo, "Autoslicer: Automatic program partitioning for securing sensitive data based-on data dependency analysis and code refactoring," in *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, IEEE, 2022, pp. 239–247.

[15] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, et al., "Glamdring: Automatic application partitioning for intel sgx," in *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 17)*, USENIX Association, 2017, pp. 285–298.

[16] K. Rubinov, L. Rosculete, T. Mitra, and A. Roychoudhury, “Automated partitioning of android applications for trusted execution environments,” in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 923–934.

[17] L. Lin, P. Li, X. Liao, H. Jin, and Y. Zhang, “Echo: An edge-centric code offloading system with quality of service guarantee,” *IEEE Access*, vol. 7, pp. 5905–5917, 2019.

[18] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “Maui: Making smartphones last longer with code offload,” in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, 2010, pp. 49–62.

[19] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, “Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading,” in *2012 Proceedings IEEE INFOCOM*, 2012, pp. 945–953.

[20] Y. Duan, M. Zhang, H. Yin, and Y. Tang, “Privacy-Preserving offloading of mobile app to the public cloud,” in *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA: USENIX Association, Jul. 2015.

[21] M. Al-Mutawa and S. Mishra, “Data partitioning: An approach to preserving data privacy in computation offload in pervasive computing systems,” in *Proceedings*

*of the 10th ACM Symposium on QoS and Security for Wireless and Mobile Networks (Q2SWinet)*, Montreal, QC, Canada: Association for Computing Machinery, 2014, pp. 51–60, ISBN: 9781450330275.

[22] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor, "Our data, ourselves: Privacy via distributed noise generation," in *Advances in Cryptology-EUROCRYPT 2006*, Springer, 2006.

[23] J. C. Duchi, M. I. Jordan, and M. J. Wainwright, "Local privacy and statistical minimax rates," in *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, IEEE, 2013, pp. 429–438.

[24] N. Holohan, D. J. Leith, and O. Mason, "Optimal differentially private mechanisms for randomised response," *IEEE Transactions on Information Forensics and Security*, pp. 2726–2735, 2017.

[25] P. Kairouz, K. Bonawitz, and D. Ramage, "Discrete distribution estimation under local privacy," in *International Conference on Machine Learning*, PMLR, 2016, pp. 2436–2444.

[26] M. S. Alvim, M. E. Andrés, K. Chatzikokolakis, P. Degano, and C. Palamidessi, "Differential privacy: On the trade-off between utility and information leakage," in *Formal Aspects of Security and Trust: 8th International Workshop, FAST 2011,*

*Leuven, Belgium, September 12-14, 2011. Revised Selected Papers 8*, Springer, 2012, pp. 39–54.

[27] A. Shamir, “How to share a secret,” *Commun. ACM*, 1979.

[28] A. C.-C. Yao, “How to generate and exchange secrets,” in *27th annual symposium on foundations of computer science (Sfcs 1986)*, IEEE, 1986, pp. 162–167.

[29] V. Kolesnikov and T. Schneider, “Improved garbled circuit: Free xor gates and applications,” in *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part II (ICALP)*, Springer-Verlag, 2008, pp. 486–498, ISBN: 9783540705826.

[30] C. Gentry, “Fully homomorphic encryption using ideal lattices,” in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.

[31] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, “(leveled) fully homomorphic encryption without bootstrapping,” in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS)*, Cambridge, Massachusetts: Association for Computing Machinery, 2012, pp. 309–325.

[32] J. Fan and F. Vercauteren, “Somewhat practical fully homomorphic encryption,” *Cryptology ePrint Archive*, vol. 2012, p. 144, 2012.

[33] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, “A full rns variant of approximate homomorphic encryption,” in *Selected Areas in Cryptography–SAC 2018: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers 25*, Springer, 2019, pp. 347–368.

[34] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, “TFHE: Fast fully homomorphic encryption over the torus,” *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020, https://doi.org/10.1007/s00145-019-09319-x.

[35] J. H. Cheon, A. Kim, M. Kim, and Y. Song, “Homomorphic encryption for arithmetic of approximate numbers,” in *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, Springer, 2017, pp. 409–437.

[36] S. Carpov, P. Dubrulle, and R. Sirdey, “Armadillo: A compilation chain for privacy preserving applications,” in *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, 2015.

[37] E. Crockett, C. Peikert, and C. Sharp, “Alchemy: A language and compiler for homomorphic encryption made easy,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, ACM, 2018, pp. 1020–1037, ISBN: 9781450356930.

[38] D. W. Archer, J. M. Calderón Trilla, J. Dagit, A. Malozemoff, Y. Polyakov, K. Rohloff, and G. Ryan, “Ramparts: A programmer-friendly system for building homomorphic encryption applications,” in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC)*, ACM, 2019, pp. 57–68, ISBN: 9781450368292.

[39] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, “Chet: An optimizing compiler for fully-homomorphic neural-network inferencing,” in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, ACM, 2019, ISBN: 9781450367127.

[40] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski, “Ngraph-he: A graph compiler for deep learning on homomorphically encrypted data,” in *Proceedings of the 16th ACM International Conference on Computing Frontiers (CF)*, ACM, 2019, pp. 3–13, ISBN: 9781450366854.

[41] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. 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)*, London, UK: Association for Computing Machinery, 2020, pp. 546–561, ISBN: 9781450376136.

[42] H. Chen, R. Cammarota, F. Valencia, F. Regazzoni, and F. Koushanfar, “Ahec: End-to-end compiler framework for privacy-preserving machine learning acceleration,” in *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference (DAC)*, IEEE Press, 2020, ISBN: 9781450367257.

[43] Y. Lee, S. Heo, S. Cheon, S. Jeong, C. Kim, E. Kim, D. Lee, and H. Kim, “Hecate: Performance-aware scale optimization for homomorphic encryption compiler,” in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2022, pp. 193–204.

[44] Y. Lee, S. Cheon, D. Kim, D. Lee, and H. Kim, “ELASM: Error-latency-aware scale management for fully homomorphic encryption,” in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023.

[45] A. Viand, P. Jattke, M. Haller, and A. Hithnawi, “Heco: Automatic code optimizations for efficient fully homomorphic encryption,” in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023.

[46] R. Malik, K. Sheth, and M. Kulkarni, “Coyote: A compiler for vectorizing encrypted arithmetic circuits,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS)*, ACM, 2023, pp. 118–133, ISBN: 9781450399180.

[47] Y. Lee, S. Cheon, D. Kim, D. Lee, and H. Kim, “Performance-aware scale analysis with reserve for homomorphic encryption,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2024, pp. 302–317.

[48] S. Cheon, Y. Lee, D. Kim, J. M. Lee, S. Jung, T. Kim, D. Lee, and H. Kim, “Dacapo: Automatic bootstrapping management for efficient fully homomorphic encryption,” in *33rd USENIX Security Symposium (USENIX Security 24)*, Philadelphia, PA: USENIX Association, Aug. 2024, pp. 6993–7010.

[49] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe, “Mobile fog: A programming model for large-scale applications on the internet of things,” in *Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing*, 2013.

[50] N. K. Giang, M. Blackstock, R. Lea, and V. C. M. Leung, “Distributed data flow: A programming model for the crowdsourced internet of things,” in *Proceedings of the Doctoral Symposium of the 16th International Middleware Conference*, 2015.

[51] N. K. Giang, M. Blackstock, R. Lea, and V. C. Leung, “Developing IoT applications in the fog: A distributed dataflow approach,” in *Proceedings of 2015 International Conference on the Internet of Things (IOT ’15)*, 2015.

[52] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden, “Wishbone: Profile-based partitioning for sensornet applications,” in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[53] T. Szydlo, R. Brzoza-Woch, J. Sendorek, M. Windak, and C. Gniady, “Flow-based programming for iot leveraging fog computing,” in *2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, 2017.

[54] *Microsoft SEAL (release 3.5.9)*, `https://github.com/Microsoft/SEAL`, Microsoft Research, Redmond, WA., Apr. 2020.

[55] *HEaaN HE Library*, `https://heaan.it`, 2022.

[56] A. Al Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, I. Quah, Y. Polyakov, S. R.V., K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca, “OpenFHE: Open-source fully homomorphic encryption library,” in *Proceedings*

*of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC)*, Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 53–63.

[57] A. Ş. Özcan and E. Savaş, "HEonGPU: A GPU-based fully homomorphic encryption library 1.0," *Cryptology ePrint Archive*, vol. 2024, p. 1543, 2024.

[58] E. Lee, J.-W. Lee, J. Lee, Y.-S. Kim, Y. Kim, J.-S. No, and W. Choi, "Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions," in *International Conference on Machine Learning*, PMLR, 2022, pp. 12 403–12 422.

[59] A. Krastev, N. Samardzic, S. Langowski, S. Devadas, and D. Sanchez, "A tensor compiler with automatic data packing for simple and efficient fully homomorphic encryption," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, Jun. 2024.

[60] Z. Zhang, Y. Liu, Y. Zhang, Z. Chen, J. Zhao, X. Feng, H. Cui, and J. Xue, "Qiwu: Exploiting ciphertext-level simd parallelism in homomorphic encryption programs," in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*, Las Vegas, NV, USA: Association for Computing Machinery, 2025, pp. 523–537, ISBN: 9798400712753.

[61] A. Ebel, K. Garimella, and B. 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)*, Rotterdam, Netherlands: Association for Computing Machinery, 2025, pp. 734–749, ISBN: 9798400710797.

[62] A. Ali, J. Choi, B. Gipson, S. Gorantala, J. Kun, W. Legiest, L. Lim, A. Viand, M. Z. Demissie, and H. Zheng, "Heir: A universal compiler for homomorphic encryption," *arXiv preprint arXiv:2508.11095*, 2025.

[63] A. E. Eshratifar, M. S. Abrishami, and M. Pedram, "Jointdnn: An efficient training and inference engine for intelligent mobile cloud computing services," *IEEE Transactions on Mobile Computing*, vol. 20, no. 2, pp. 565–576, 2019.

[64] G. Li, L. Liu, X. Wang, X. Dong, P. Zhao, and X. Feng, "Auto-tuning neural network quantization framework for collaborative inference between the cloud and edge," in *Artificial Neural Networks and Machine Learning–ICANN 2018: 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4-7, 2018, Proceedings, Part I 27*, Springer, 2018, pp. 402–411.

[65] Y. Otoum, S. K. Yadlapalli, and A. Nayak, "Ftliot: A federated transfer learning framework for securing iot," in *GLOBECOM 2022-2022 IEEE Global Communications Conference*, IEEE, 2022.

[66] P. Singh, M. Masud, M. S. Hossain, A. Kaur, G. Muhammad, and A. Ghoneim, "Privacy-preserving serverless computing using federated learning for smart grids," *IEEE Transactions on Industrial Informatics*, pp. 7843–7852, 2021.

[67] A. Grafberger, M. Chadha, A. Jindal, J. Gu, and M. Gerndt, "Fedless: Secure and scalable federated learning using serverless computing," in *2021 IEEE International Conference on Big Data (Big Data)*, IEEE, 2021, pp. 164–173.

[68] P. Goyal, M. Schtern, S. Mukhopadhyay, and M. Litoiu, "Towards a differential privacy machine learning edge-cloud architecture-an experimental study," in *Proceedings of the 32nd Annual International Conference on Computer Science and Software Engineering*, 2022.

[69] L. Zhong, L. Zhang, L. Xu, and L. Wang, "Mpc-based privacy-preserving serverless federated learning," in *2022 3rd International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE)*, IEEE, 2022, pp. 493–497.

[70] Y. Wu, X. Wang, W. Susilo, G. Yang, Z. L. Jiang, S.-M. Yiu, and H. Wang, "Generic server-aided secure multi-party computation in cloud computing," *Computer Standards & Interfaces*, p. 103 552, 2022.

[71] S. Truex, N. Baracaldo, A. Anwar, T. Steinke, H. Ludwig, R. Zhang, and Y. Zhou, "A hybrid approach to privacy-preserving federated learning," in *Proceedings of the 12th ACM workshop on artificial intelligence and security*, 2019, pp. 1–11.

[72] M. Gong, Y. Xie, K. Pan, K. Feng, and A. K. Qin, "A survey on differentially private machine learning," *IEEE computational intelligence magazine*, pp. 49–64, 2020.

[73] J. Kim, M. S. Lee, A. Yun, and J. H. Cheon, "Crt-based fully homomorphic encryption over the integers," *Cryptology ePrint Archive*, 2013.

[74] J. H. Cheon, J. Kim, M. S. Lee, and A. Yun, "Crt-based fully homomorphic encryption over the integers," *Information Sciences*, vol. 310, pp. 149–162, 2015.

[75] M. X. Makkes, A. Uta, R. B. Das, V. N. Bozdog, and H. Bal, "P̂ 2-swan: Real-time privacy preserving computation for iot ecosystems," in *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, IEEE, 2017, pp. 1–10.

[76] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in Cryptology—EUROCRYPT'99: International Conference on the Theory and Application of Cryptographic Techniques Prague, Czech Republic, May 2–6, 1999 Proceedings 18*, Springer, 1999, pp. 223–238.

[77] C. He, G. Liu, S. Guo, and Y. Yang, “Privacy-preserving and low-latency federated learning in edge computing,” *IEEE Internet of Things Journal*, vol. 9, no. 20, pp. 20 149–20 159, 2022.

[78] M. S. Rahman, I. Khalil, M. Atiquzzaman, and X. Yi, “Towards privacy preserving ai based composition framework in edge networks using fully homomorphic encryption,” *Engineering Applications of Artificial Intelligence*, vol. 94, p. 103 737, 2020.

[79] S. Li, S. Zhao, G. Min, L. Qi, and G. Liu, “Lightweight privacy-preserving scheme using homomorphic encryption in industrial internet of things,” *IEEE Internet of Things Journal*, vol. 9, no. 16, pp. 14 542–14 550, 2022.

[80] B. Kim, S. Heo, J. Lee, S. Jeong, Y. Lee, and H. Kim, “Compiler-assisted semantic-aware encryption for efficient and secure serverless computing,” *IEEE Internet of Things Journal*, vol. 8, no. 7, pp. 5645–5656, 2021.

[81] S. Ramesh and M. Govindarasu, “An efficient framework for privacy-preserving computations on encrypted iot data,” *IEEE Internet of Things Journal*, vol. 7, no. 9, pp. 8700–8708, 2020.

[82] S. Cheon, Y. Lee, H. Youm, D. Kim, S. Yun, K. Jeong, D. Lee, and H. Kim, “Halo: Loop-aware bootstrapping management for fully homomorphic encryption,” in

*Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS)*, Rotterdam, Netherlands: Association for Computing Machinery, 2025, pp. 572–585, ISBN: 9798400706981.

[83] P. Yuan, Y. Liu, J. Lai, L. Li, T. Sui, L. Xiao, X. Zhang, Q. Zhu, and J. Xue, "Metakernel: Enabling efficient encrypted neural network inference through unified mvm and convolution," *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA2, Oct. 2025.

[84] *Cingulata*, `https://github.com/CEA-LIST/Cingulata`, 2020.

[85] E. Chielle, O. Mazonka, H. Gamil, N. G. Tsoutsos, and M. Maniatakos, "E3: A framework for compiling c++ programs with encrypted operands," *Cryptology ePrint Archive*, vol. 2018, p. 1013, 2018.

[86] A. Viand and H. Shafagh, "Marble: Making fully homomorphic encryption accessible to all," in *Proceedings of the 6th Workshop on Encrypted Computing; Applied Homomorphic Cryptography*, ACM, 2018.

[87] M. Cowan, D. Dangwal, A. Alaghi, C. Trippel, V. T. Lee, and B. Reagen, "Porcupine: A synthesizing compiler for vectorized homomorphic encryption," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Program-*

*ming Language Design and Implementation (PLDI)*, ACM, 2021, pp. 375–389, ISBN: 9781450383912.

[88] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Virtual Event, Republic of Korea: IEEE Press, 2021, pp. 2–14, ISBN: 9781728186139.

[89] A. Vishwanath, R. Peruri, and J. He, "Security in fog computing through encryption," *International Journal of Information Technology and Computer Science*, 2016.

[90] Z. Mo, Y. Qiao, and S. Chen, "Two-party fine-grained assured deletion of outsourced data in cloud systems," in *2014 IEEE 34th International Conference on Distributed Computing Systems*, 2014.

[91] Y. Tang, P. P. C. Lee, J. C. S. Lui, and R. Perlman, "Secure overlay cloud storage with access control and assured deletion," *IEEE Transactions on Dependable and Secure Computing*, 2012.

[92] A. K. Ranjan, V. Kumar, and M. Hussain, “Security analysis of cloud storage with access control and file assured deletion (fade),” in *2015 Second International Conference on Advances in Computing and Communication Engineering*, 2015.

[93] Z. Xia, X. Wang, X. Sun, and Q. Wang, “A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 340–352, Feb. 2016.

[94] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, “Privacy-preserving multi-keyword ranked search over encrypted cloud data,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 222–233, Jan. 2014.

[95] W. Sun, B. Wang, N. Cao, M. Li, W. Lou, Y. T. Hou, and H. Li, “Privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking,” in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS)*, ACM, 2013, pp. 71–82.

[96] M. Fredrikson, S. Jha, and T. Ristenpart, “Model inversion attacks that exploit confidence information and basic countermeasures,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, ACM, 2015, pp. 1322–1333, ISBN: 9781450338325.

[97] Y. Zhang, R. Jia, H. Pei, W. Wang, B. Li, and D. Song, "The secret revealer: Generative model-inversion attacks against deep neural networks," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE Computer Society, 2020, pp. 250–258.

[98] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction apis," in *25th USENIX Security Symposium (USENIX Security 16)*, USENIX Association, 2016, pp. 601–618, ISBN: 9781931971324.

[99] R. Lehmkuhl, P. Mishra, A. Srinivasan, and R. A. Popa, "Muse: Secure inference resilient to malicious clients," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, 2021, pp. 2201–2218, ISBN: 978-1-939133-24-3.

[100] M. Furka, M. Kalúz, M. Fikar, and M. Klaučo, "Guidelines for secure process control: Harnessing the power of homomorphic encryption and state feedback control," *IEEE Access*, vol. 11, pp. 110 328–110 341, 2023.

[101] J. Bos, K. Lauter, and M. Naehrig, "Private predictive analysis on encrypted medical data.," *Journal of biomedical informatics*, vol. 50, May 2014.

[102] R. D'Agostino, M. Pencina, J. Massaro, and S. Coady, "Cardiovascular disease risk assessment: Insights from framingham," *Global heart*, vol. 8, pp. 11–23, Mar. 2013.

[103] B. Tabaei and W. Herman, "A multivariate logistic regression equation to screen for diabetes : Development and validation," *Diabetes care*, vol. 25, pp. 1999–2003, Nov. 2002.

[104] L. S. FRIDERICIA, "Die systolendauer im elektrokardiogramm bei normalen menschen und bei herzkranken," *Acta Medica Scandinavica*, vol. 53, no. 1, pp. 489–506, 1920. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.0954-6820.1920.tb18267.x.

[105] A. Page, O. Kocabas, S. Ames, M. Venkitasubramaniam, and T. Soyata, "Cloud-based secure health monitoring: Optimizing fully-homomorphic encryption for streaming algorithms," in *2014 IEEE Global Communications Conference Workshops (GLOBECOM Workshops)*, Dec. 2014, pp. 48–52.

[106] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal: Association for Computing Machinery, 2011, pp. 85–100, ISBN: 9781450309776.

[107] Y. Guan, J. Shao, G. Wei, and M. Xie, “Data security and privacy in fog computing,” *IEEE Network*, vol. 32, no. 5, pp. 106–111, 2018.

[108] J.-S. Fu, Y. Liu, H.-C. Chao, B. K. Bhargava, and Z.-J. Zhang, “Secure data storage and searching for industrial iot by integrating fog computing and cloud computing,” *IEEE Transactions on Industrial Informatics*, vol. 14, no. 10, pp. 4519–4528, 2018.

**Abstract in Korean**

## 엣지-클라우드 시스템에서의 권한 인식 동형암호 프로그램 분할 컴파일러

엣지-클라우드 컴퓨팅은 엣지 장치가 수집한 데이터를 클라우드의 높은 연산 성능과 함께 활용할 수 있어 다양한 응용에서 널리 사용되고 있다. 그러나 사용자 데이터와 서비스 제공자의 모델 매개변수가 함께 사용되는 환경에서는 프라이버시 문제가 발생한다. 사용자 데이터는 클라우드에 노출되지 않아야 하며, 서비스 제공자의 모델 정보도 사용자 장치에 노출되지 않아야 한다. 완전 동형암호(FHE)는 암호화된 데이터 위에서 직접 연산을 수행할 수 있어 이러한 문제를 해결할 수 있는 방법을 제공한다.

완전 동형암호를 엣지-클라우드 실행에 실용적으로 적용하기 위해서는 여전히 컴파일러 지원이 필요하다. 프로그램 전체를 암호화된 상태로 실행하면 프라이버시는 보호할 수 있지만, 불필요한 암호문 연산으로 인해 큰 실행 시간과 메모리 오버헤드가 발생한다. 반대로 프로그램의 일부만 암호화하여 실행하려면 어떤 값이 어느 장치에서 평문으로 접근 가능한지 분석하고, 암호화 실행이 필요한 지점을 정확히 결정해야 한다. 또한 텐서 연산으로 작성된 프로그램은 FHE 라이브러리가 제공하는 기본 암호문 연산으로 바로 실행될 수 없기 때문에, 텐서 연산을 암호문 수준의 실행으로 변환하는 과정도 필요하다.

본 학위논문은 프라이버시 보존 엣지-클라우드 텐서 실행을 위한 권한 인식 컴파일러인 AION을 제안한다. AION은 개발자가 소스 수준의 배치 주석과 권한 주석을 사용하여 텐서 계산과 프라이버시 의도를 함께 표현할 수 있도록 한다. 주석이 포함된 프로그램으로부터 AION은 프라이버시 권한이 프로그램을 따라 어떻게 전파되는지 분석하고, 엣지-클라우드 실행 중 보호되어야 하는 값을 식별한다. 텐서 계산에 대해서는 텐서 값이 암호문 슬롯 위에서 어떻게 표현

되는지를 고려하여 FHE 기본 연산으로 변환한다. 이후 AION은 암호화 연산 비용과 암호문 통신 비용을 추정하여 엣지-클라우드 분할 계획을 선택하고, 필요한 암호화, 복호화, 통신 연산을 포함하는 엣지와 클라우드 프로그램을 생성한다.

실험 결과는 AION이 프라이버시 보존 엣지-클라우드 텐서 실행의 효율성과 프로그래밍 생산성을 함께 개선함을 보여준다. AION은 EVA와 Hecate 대비 각각 4.92배와 3.69배의 기하평균 속도 향상을 달성하였으며, 엣지와 클라우드의 메모리 사용량을 각각 최대 65.5%와 59.1% 줄였다. 또한 AION은 수작업 암호화 구간 선택, 통신 삽입, 암호화와 복호화 관리, 텐서 연산의 FHE 기본 연산 변환을 소스 수준의 주석과 컴파일러가 생성한 엣지-클라우드 프로그램으로 대체하여 개발자 부담을 줄인다. 이러한 결과는 FHE 기반 엣지-클라우드 텐서 실행을 실용적으로 만들기 위해 권한 인식 분할과 암호화 텐서 하향 변환이 같은 컴파일러 흐름 안에서 함께 다루어져야 함을 보여준다.

핵심되는 말: 동형암호, 엣지-클라우드 컴퓨팅, 프로그램 분할, 타입 시스템