

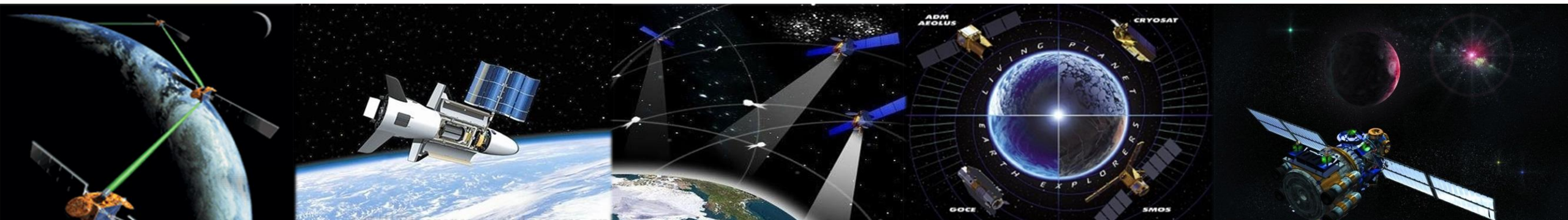


北京航空航天大学  
BEIHANG UNIVERSITY

# HEIR: A Unified Representation for Cross-Scheme Compilation of Fully Homomorphic Computation

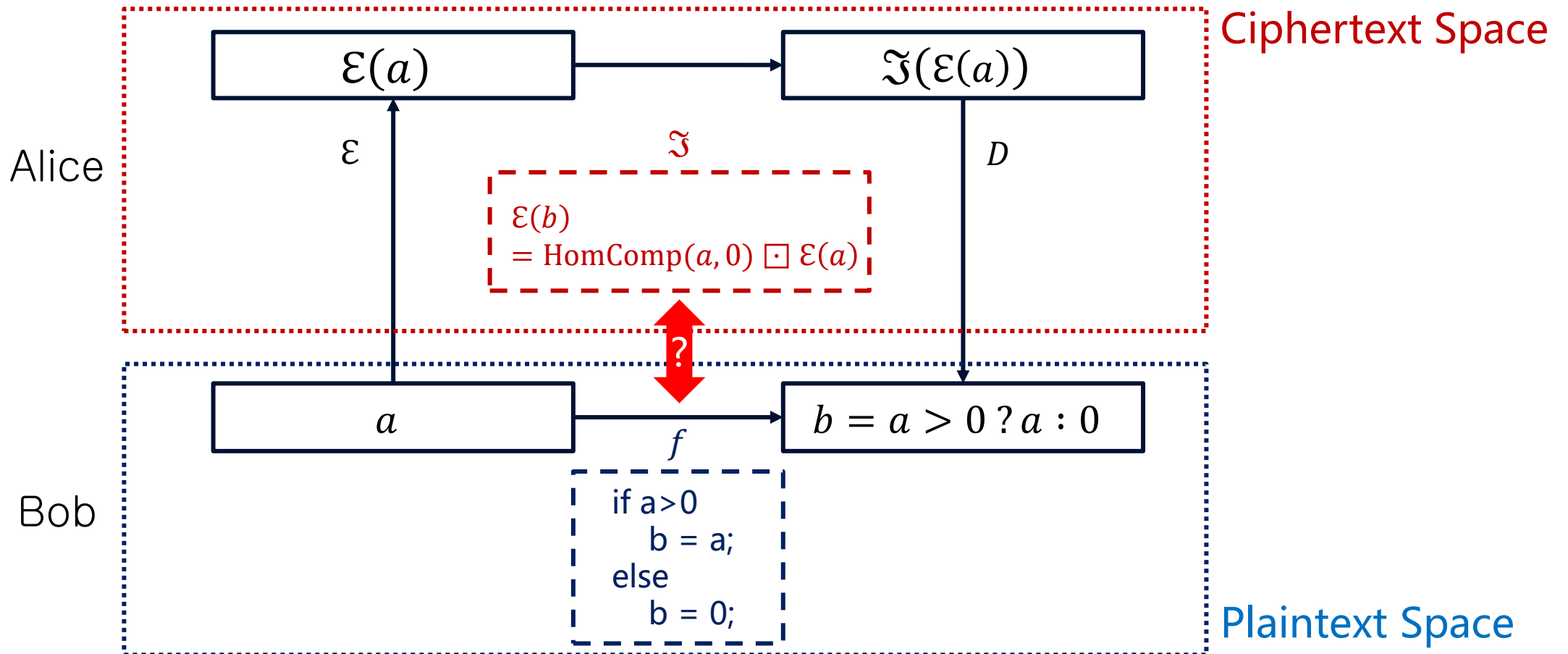
Song Bian, **Zian Zhao**, Zhou Zhang, Ran Mao,  
Kohei Suenaga, Yier Jin, Zhenyu Guan, Jianwei Liu

Beihang University, Kyoto University, University of Science and Technology of China



# Fully Homomorphic Encryption

Q: How to automate such design?



# FHE Compiler Overview

User Program



FHE Compiler



FHE Library



**Standard C**

```
int foo(int [] x, int[] y){
  int[] r;
  for(i = 0; i < 4; ++i){
    r[i] = x[i] * y[i]
  }
  return r;
}
```

**Compiler's IR**

```
func foo(%arg0:RLWECipher, %arg1:RLWECipher){
  RLWECipher v0 = heir.define() : RLWECipher
  RLWECipher v1 = heir.rlwemult(%arg0, %arg1) :
RLWECipher
  return v1;
}
```

**SEAL Library**

```
Ciphertext foo(Ciphertext x,
               Ciphertext y){
  Ciphertext r;
  evaluator.multiply(x, y, r);
  return r;
}
```

# Why do we need FHE compilers?

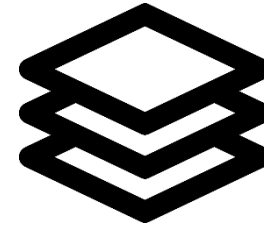
## FHE programming Paradigm



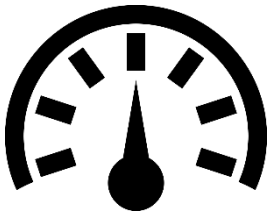
**No If/Else**



**No Loops**



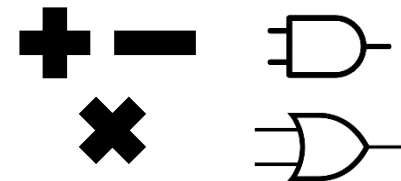
**SIMD Batching**



**Parameter Selection**



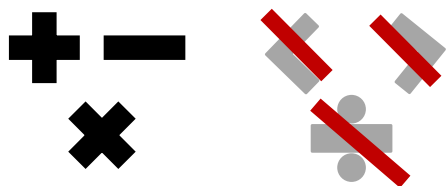
**Noise Management**



**Limited Operators**

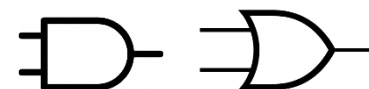
# Why do we need FHE compilers?

## FHE schemes

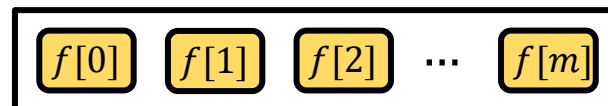


### Arithmetic FHE

- Schemes: BFV/BGV/CKKS
- RLWE-based ciphertext
- Support SIMD parallelism
- Only support arithmetic operations



Boolean Gates



Look-up Table

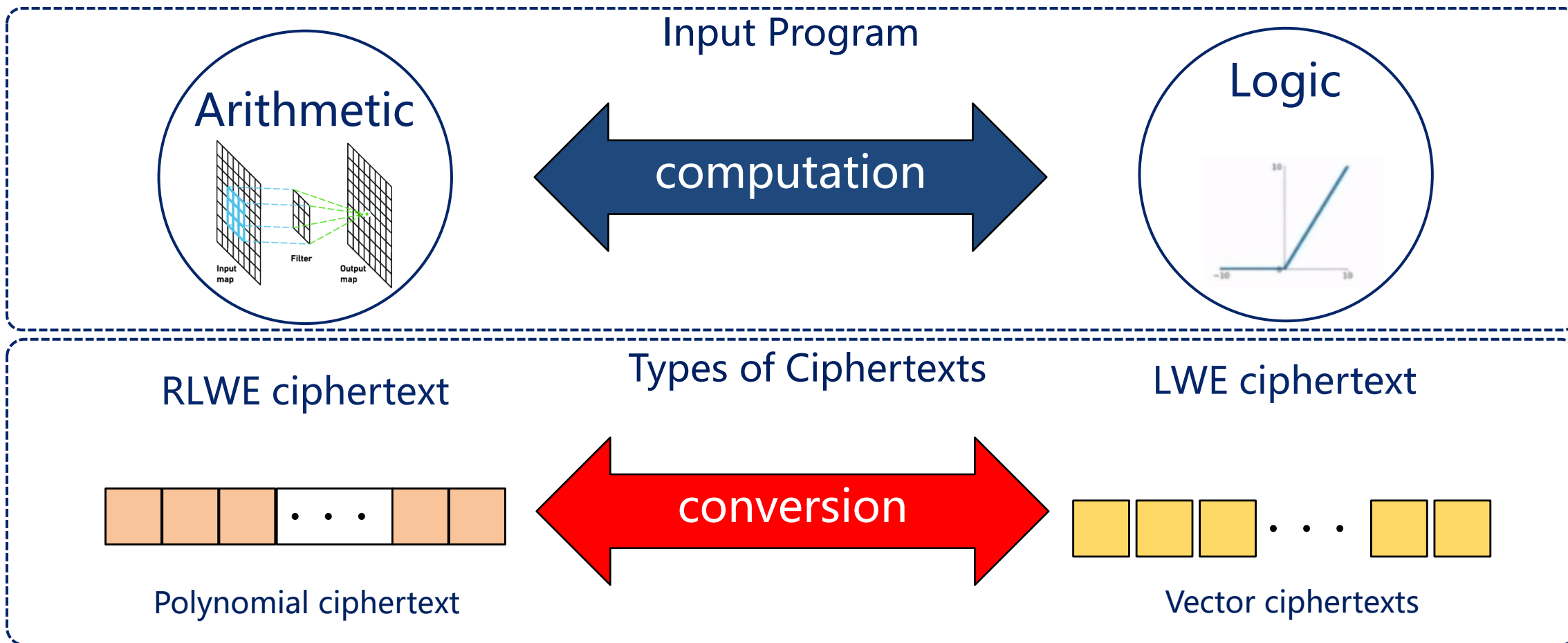
### Logic FHE

- Schemes: FHEW/TFHE
- LWE-based ciphertext
- Support logic operations through LUT and Boolean gates (Bootstrapping)
- Slow multiplication operations

***The existing FHE compilers only support compilation for single FHE scheme***

# Why do we need FHE compilers?

Automatic scheduling for different FHE schemes

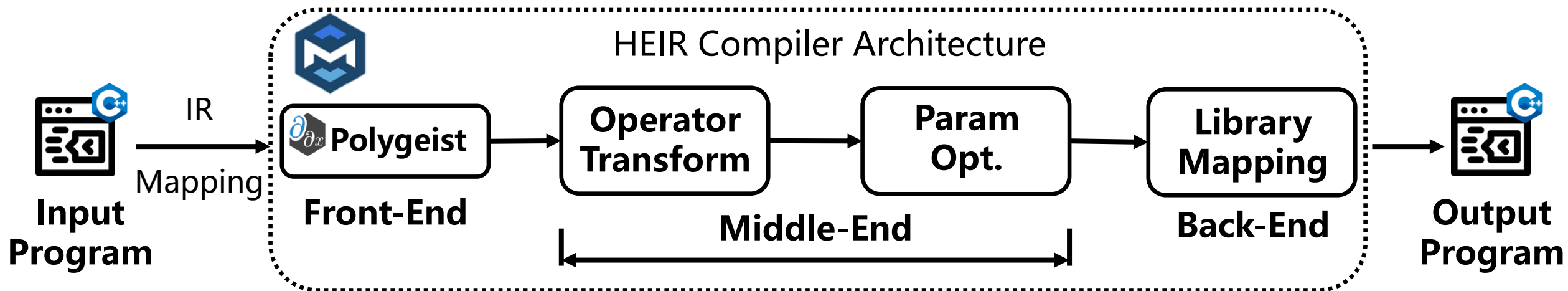


# Motivation

- ❑ Build a DSL-free functional complete FHE compiler that does not solely rely on Boolean-circuit representation
- ❑ Propose an unified intermediate representation for FHE to support cross-scheme FHE program scheduling and optimization
- ❑ Support compilation for large-scale computation tasks that contain both arithmetic and logic operations

# HEIR Compiler Pipeline

## HEIR: Compiler for FHE

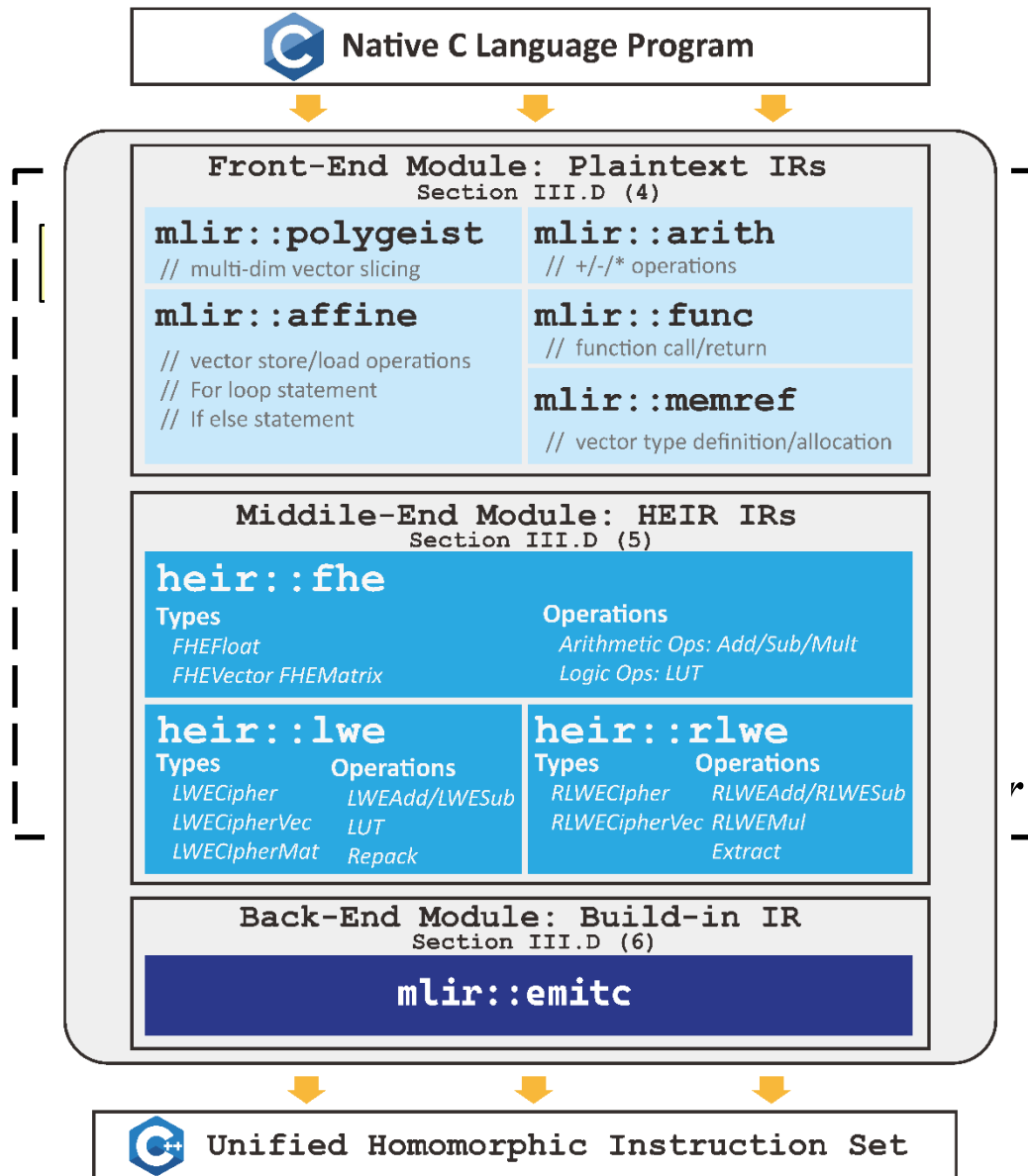


### Key components:

- **Program Segmentation:** cut operators into regions according to arithmetic or logic circuit
- **Encoding Optimization:** cut data into types according to plaintext encoding
- **Parameter Management:** Setting and optimizing encryption parameters



# IR stack definition in HEIR



- Front-End tool (Polygeist) converts C source code to MLIR **Standard dialects**
- Lowering MLIR Standard dialects to the unified **FHE Dialect** representation
- Segmenting the FHE IR program and lowering to **LWE** and **RLWE Dialects**
- Lowering to **EmitC Dialect** to dock with APIs/instructions in FHE library

# Example Program

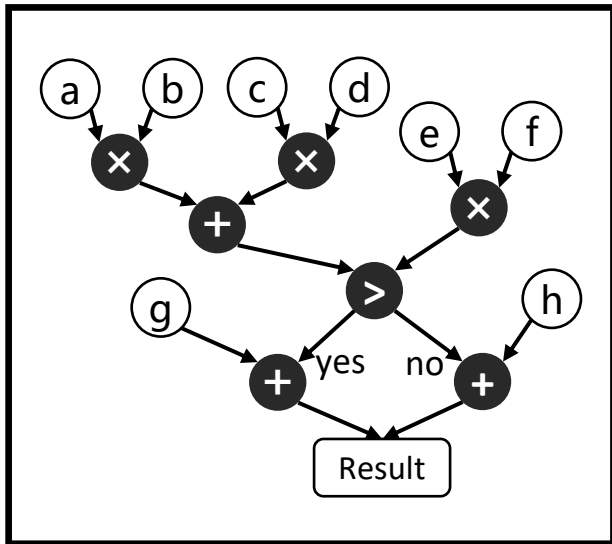
## Program Transformation

```
1  #include<stdio.h>
2  int min_dist(int data[5][4], cent[4]) {
3      int min;
4      int dist[5];
5      for (int i=0; i < 5; i++) {
6          for (int j = 0; j < 4; j++)
7              dist[i]+=(data[i][j]-cent[j])^2
8      }
9      min = dist[0];
10     for (int i = 1; i < 5; i++) {
11         if (min > dist[i]) min = dist[i];
12     }
13     return min;}
```

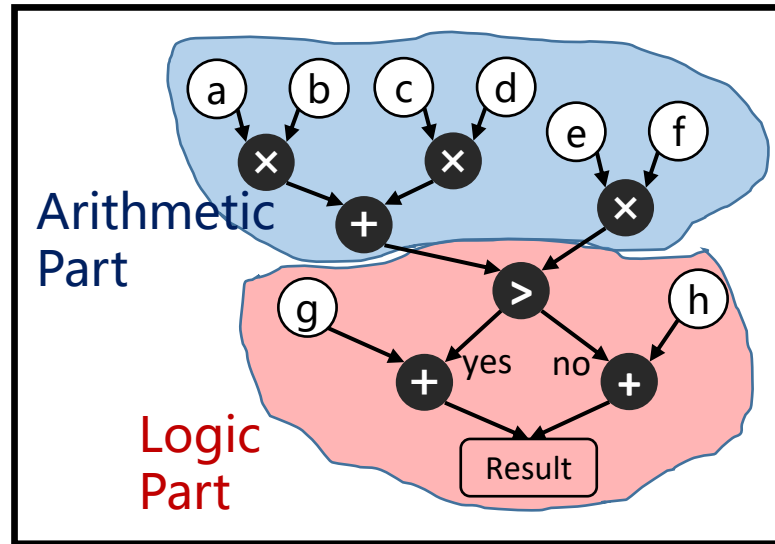
```
1  LWECipher min_dist(vector<RLWECipher> data[5][4],
2                          RLWECipher cent) {
3      LWECipher min;
4      vector<LWECipher> dist;
5      for (int i=0; i < 5; i++)
6          dist[i] = inner_prod(data[i], data[i]) +
7                  inner_prod(cent, cent)-2*inner_prod(data[i], cent);
8      min = dist[0]
9      for (int i = 1; i < 5; i++) {
10         LWECipher msb = dist[i] - min;
11         msb = LUT(msb, x < 0 ? 1 : 0);
12         min = msb * dist[i] + (1 - msb) * min
13     } return min;}
```

# Program Segmentation

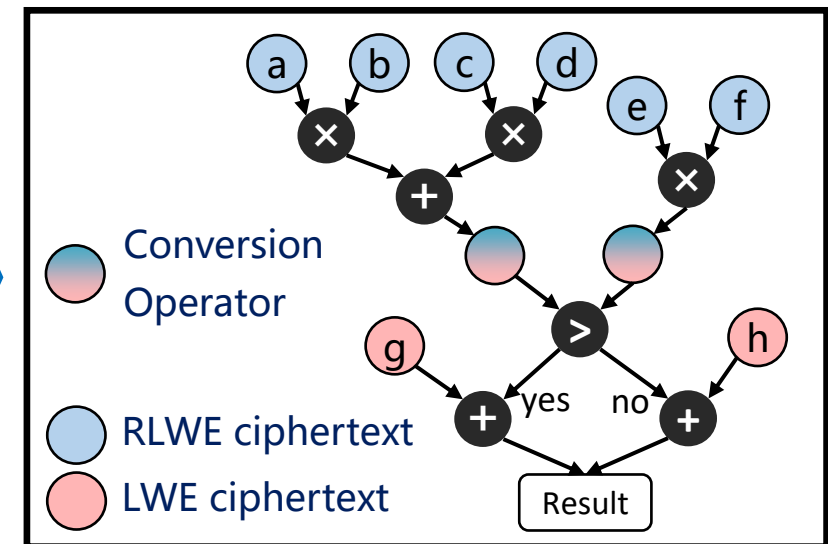
## Operator-based Type Inference



Data flow graph of input program



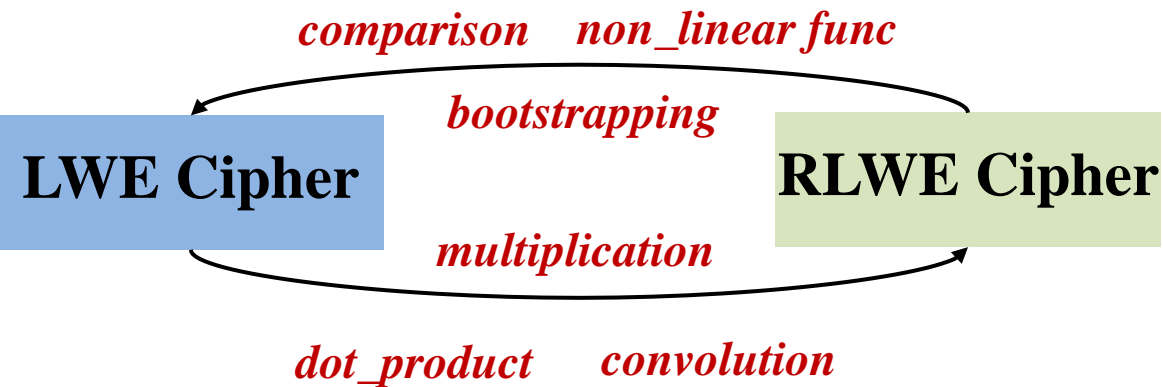
Program Segmentation



Data type Alignment

# Program Segmentation

## Ciphertext Type Conversion in HEIR



Specific Operator	Constraint Type	Operator Insertion
fhe.CompareOp	LWE	fhe.ExtractOp
fhe.LUTOp	LWE	fhe.ExtractOp
fhe.MultOp	RLWE	fhe.RepackOp
fhe.DotProductOp	RLWE	fhe.RepackOp

$$\text{FHE} - \text{LWE} \frac{u \in \{\text{Inputs}, \text{Variables}\} \quad u.type \in \{\text{FHEVector}, \text{FHEMatrix}\}}{u.type \leftarrow \{\text{LWECipherVec}, \text{LWECipherMat}\}}$$

$$\text{LWE} - \text{RLWE} \frac{u \in \{\text{Inputs}, \text{Variables}\} \quad u.type \in \{\text{LWECipherVec}, \text{LWECipherMat}\} \quad u.op \in \{\text{MULTIPLY}, \text{SLICE}\}}{u.type \leftarrow \{\text{RLWECipher}, \text{RLWECipherVec}\} \quad u \leftarrow \text{REPACK}(u)}$$

$$\text{RLWE} - \text{LWE} \frac{u \in \{\text{Inputs}, \text{Variables}\} \quad u.type \in \{\text{RLWECipher}, \text{RLWECipherVec}\} \quad u.op \in \{\text{LUT}, \text{LOAD}, \text{STORE}\}}{u.type \leftarrow \{\text{LWECipherVec}, \text{LWECipherMat}\} \quad u \leftarrow \text{SAMPLEEXTRACT}(u)}$$

# Encoding Optimization

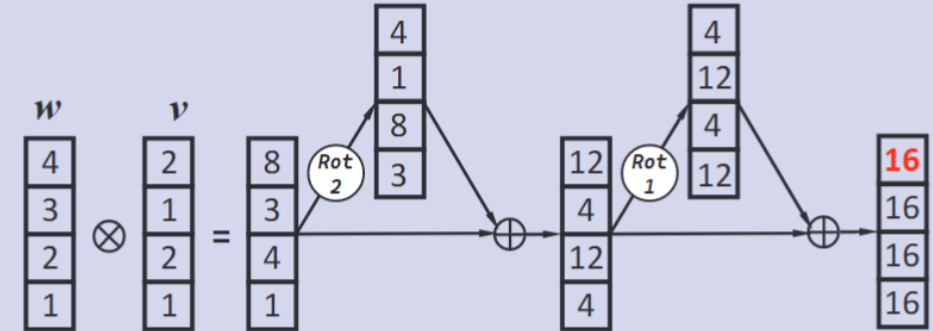
```

1  LWECipher min_dist(vector<RLWECipher> data[5][4],
   RLWECipher cent) {
2  LWECipher min;
3  vector<LWECipher> dist;
4  for (int i=0; i < 5; i++)
5    dist[i] = inner_prod(data[i], data[i]) +
6    inner_prod(cent, cent)-2*inner_prod(data[i], cent);
7  min = dist[0]
8  for (int i = 1; i < 5; i++) {
9    LWECipher msb = dist[i] - min;
10   msb = LUT(msb, x < 0 ? 1 : 0);
11   min = msb * dist[i] + (1 - msb) * min
12  } return min;}

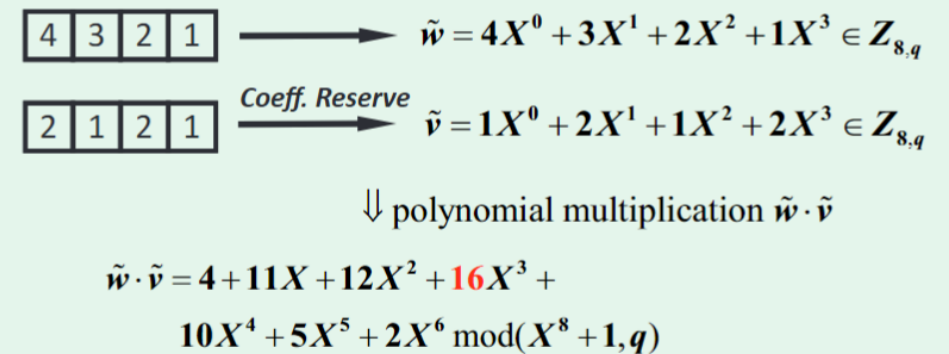
```



(1) Inner product in slot-encoding ciphertext



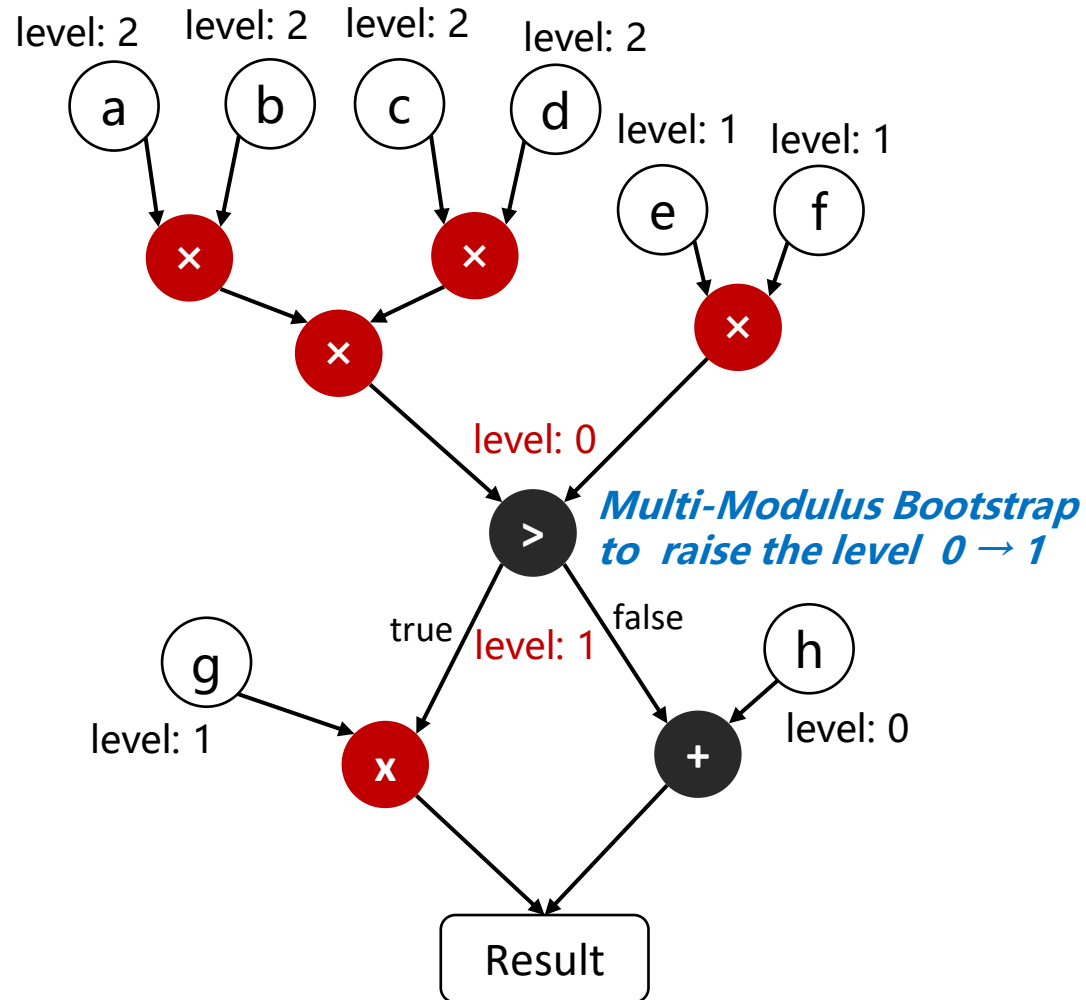
(2) Inner product in coefficient-encoding ciphertext



- Rule-based method (Pattern Matching) to accelerate computations, e.g., accumulation/inner product/convolution

# Parameter Selection

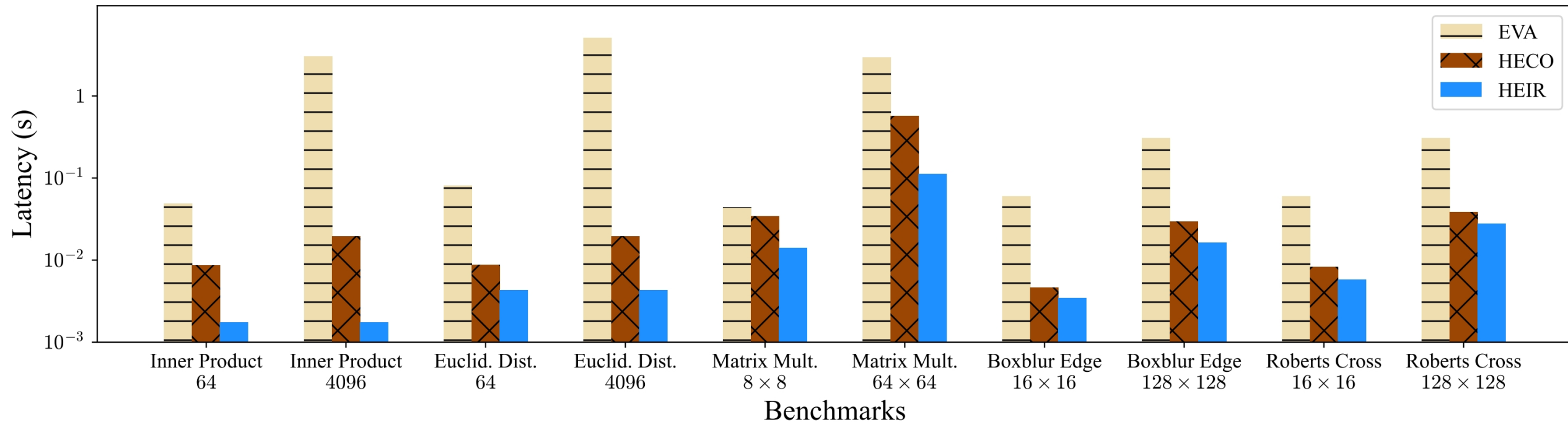
## Level Management



- The **level** of a ciphertext is associated with its “modulus”, indicating the **multiplicative depth** that can be supported.
- **Multi-Modulus Bootstrapping** is proposed to enhance the level of a ciphertext while enabling non-polynomial operations.

# Experiments

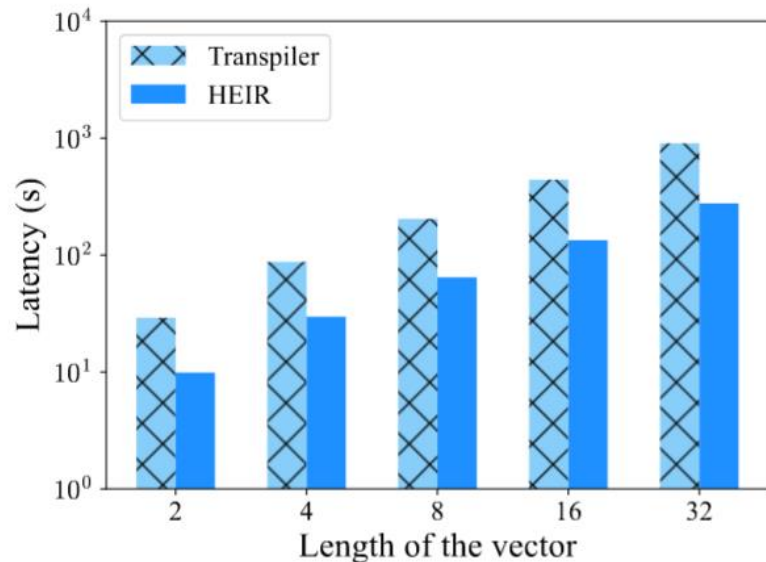
## 1) Arithmetic Circuit Evaluation



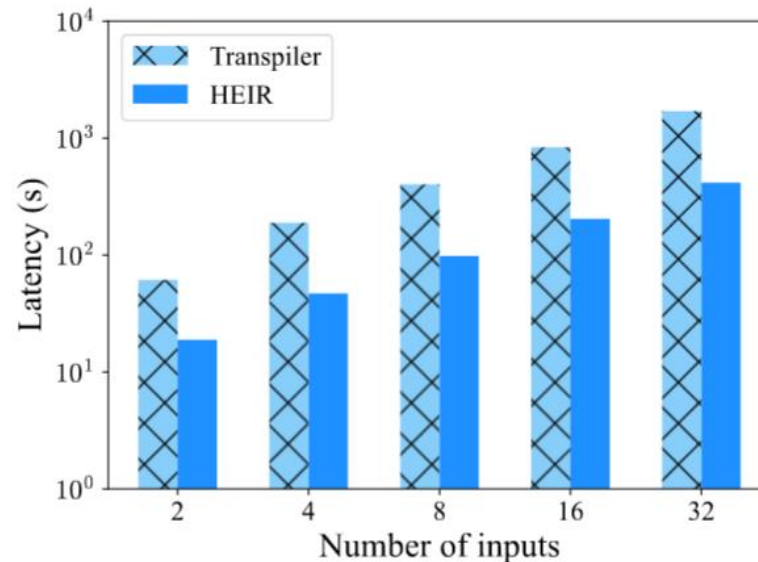
➤ Compared with EVA and HECO, HEIR introduces **encoding optimization**

# Experiments

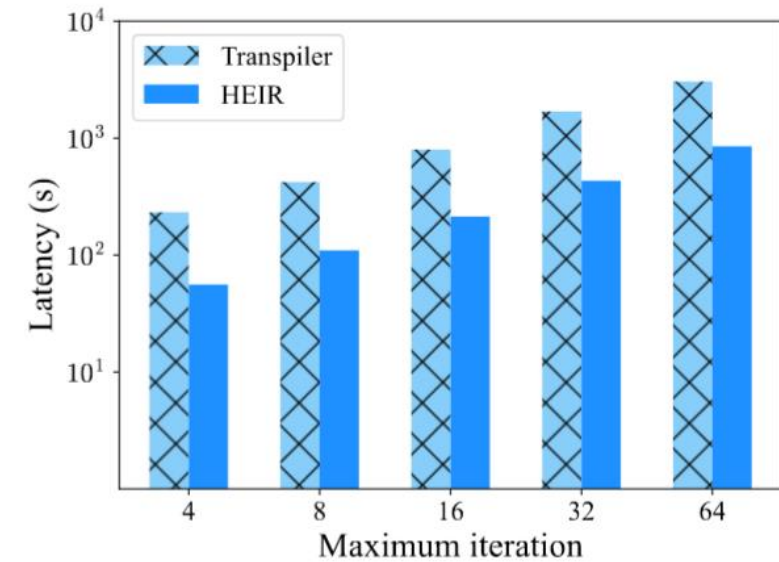
## 2) Logic Circuit Evaluation



(a) Minimum value in the the vector



(b) Minimum index in the the vector



(c) Result of a Fibonacci sequence

➤ **Transpiler** compiles the program to a logic circuit while **HEIR** evaluates **non-polynomial functions with Look-Up Tables**



# Experiments

## Compiling a K-Means Program

(1) 5 data points, 3 features, 2 centroids

(2) 10 data points, 3 features, 2 centroids

Implementaions	Latency (s)
Transpiler	<b>24321.191</b>
HEIR	<b>135.634</b>
Hand-tuned	<b>59.338</b>

Implementations	Euclid Dist	Euclid Dist + Min Value	Centroids Update	Latency (s)
Transpiler	2842.126	5406.806	2385.829	<b>Compilation fail</b>
HEIR	--	--	--	<b>248.823</b>
Hand-tuned	--	--	--	<b>105.621</b>

**Generated program runs ~180x faster than Google Transpiler**  
**Supports larger-scale program compilation**



GitHub



docker

[github.com/heir-compiler/HEIR](https://github.com/heir-compiler/HEIR)

[hub.docker.com/repository/docker/zhaozian/heir](https://hub.docker.com/repository/docker/zhaozian/heir)

# HEIR: A Unified Representation for Cross-Scheme Compilation of Fully Homomorphic Computation

Song Bai\*, Zhan Zhao\*, Zhen Zhang\*, Ran Mao\*, Kobei Suenaga<sup>†</sup>, Yier Jin<sup>‡</sup>, Zhenyu Guan<sup>§</sup>, Jianwei Liu<sup>||</sup>  
\*Beihang University, Email: {bai, zhan, zhen, ran, kobei, yier, zhenyu, guan, liujianwei}@buaa.edu.cn  
<sup>†</sup>Kyoto University, Email: suenaga@gmail.com  
<sup>‡</sup>University of California, Los Angeles, Email: jin@ucla.edu  
<sup>§</sup>University of Science and Technology of China, Email: jinyier@gmail.com  
<sup>||</sup>University of Science and Technology of China, Email: liujianwei@ustc.edu.cn

**Abstract**—We propose a new compiler framework that automates code generation over multiple fully homomorphic encryption (FHE) schemes. While it was recently shown that algorithms combining multiple FHE schemes (e.g., CKKS and HEAL) achieve high execution efficiency and low latency in the same time, developing fast cross-scheme FHE algorithms for real-world applications generally require heavy hand-crafted optimizations by cryptographers, we design and implement a compiler framework based on multi-level intermediate representation (IR). To achieve cross-scheme compilation of FHE, we develop a cross-scheme code-to-code translation (CCAT) to convert FHE programs to a common IR dialect. First, the abstract IR is converted to a concrete IR, and then the concrete IR is converted to the target FHE hardware. In the experiment, we implement the compiler in a friendly dialect into our bottom-level FHE program along with the associated data types and the data layout. In this paper, we demonstrate that complex code is lower the translation time and can be directly executed on binary stack for HEAL and homomorphic K-Means clustering software, such as homomorphic data aggregation in databases, and homogeneous data aggregation in databases, can be compiled to run 2.2-32% faster than the program generated by the state-of-the-art FHE compilers.

**1. INTRODUCTION**  
Fully homomorphic encryption (FHE) is a type of general secure multi-party computing (MPC) techniques that enable participating parties to jointly evaluate arbitrary functions securely. Due to its low round complexity in two-party secure function evaluation, FHE finds applications in two-party secure function evaluation, such as secure computation outsourcing, e.g., secure evaluation, such as secure computation outsourcing over FHE [1–4], and secure evaluation of programs over HEAL [5–7] and database [8] and program deployment at scale can range from hours [9] to days [7, 8] to complete different design. First, the running times of tasks at different product of design. First, the running times of tasks at different product of design. First, the running times of tasks at different product of design. First, the running times of tasks at different product of design.

...we observe a usability-efficiency trade-off, where the design of FHE protocols either require cryptographic experts to hand-tune the exact homomorphic operators, or the designed protocols suffer from significant performance penalties.

In addition to operator choices, the design of FHE programs is further complicated by the ciphertext-operator mismatch problem. As mentioned above, computing homomorphic matrix-vector product using the coefficient representation is incompatible with certain FHE operators, such as homomorphic squaring, homomorphic filtering, and non-polynomial operators [12–17]. Neither dot nor coefficient representations are efficient, some recent polyhomomorphic operators. In particular, some recent polyhomomorphic operators (e.g., HEAL [12–17]) are not well suited for the target application.

The limitation of the development of secure MPC protocols for non-trivial design goals can be seen in the development of secure MPC protocols for non-trivial design goals. In fact, without the flexibility of MPC protocols, many MPC protocols are not well suited for the target application.

Table I summarizes the features of various FHE compilers. HEIR can achieve homomorphic for the operators in the state-of-the-art FHE compilers, but the operators are implemented in state-of-the-art FHE compilers, but the operators are implemented in state-of-the-art FHE compilers, but the operators are implemented in state-of-the-art FHE compilers.

Compiler	HEAL [12]	HEAL [13]	HEAL [14]	HEAL [15]	HEAL [16]	HEAL [17]	HEIR [18]	HEIR [19]	HEIR [20]
Supports	✓	✓	✓	✓	✓	✓	✓	✓	✓
Arithmetic	✓	✓	✓	✓	✓	✓	✓	✓	✓
Logic	✓	✓	✓	✓	✓	✓	✓	✓	✓
Arithmetic	✓	✓	✓	✓	✓	✓	✓	✓	✓
HEAL	✓	✓	✓	✓	✓	✓	✓	✓	✓

Network and Distributed System Security (NDSS) Symposium 2024  
26 February – 1 March 2024, San Diego, CA, USA  
ISBN 978-1-969-92-117-4  
https://doi.org/10.1145/3658593.3658597  
www.ndss.symposium.org