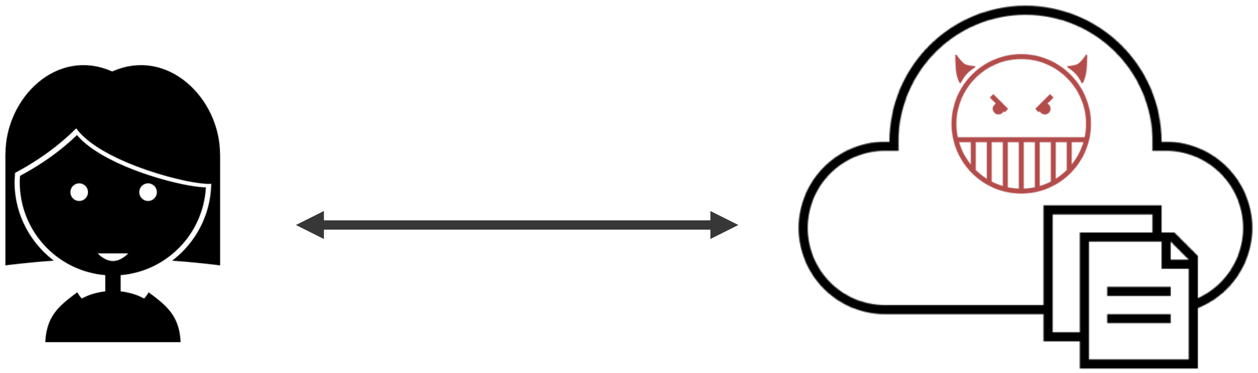
Useable Fully Homomorphic Encryption

Challenges & Opportunities

Anwar Hithnawi, Alexander Viand

ETH zürich

Cloud Computing

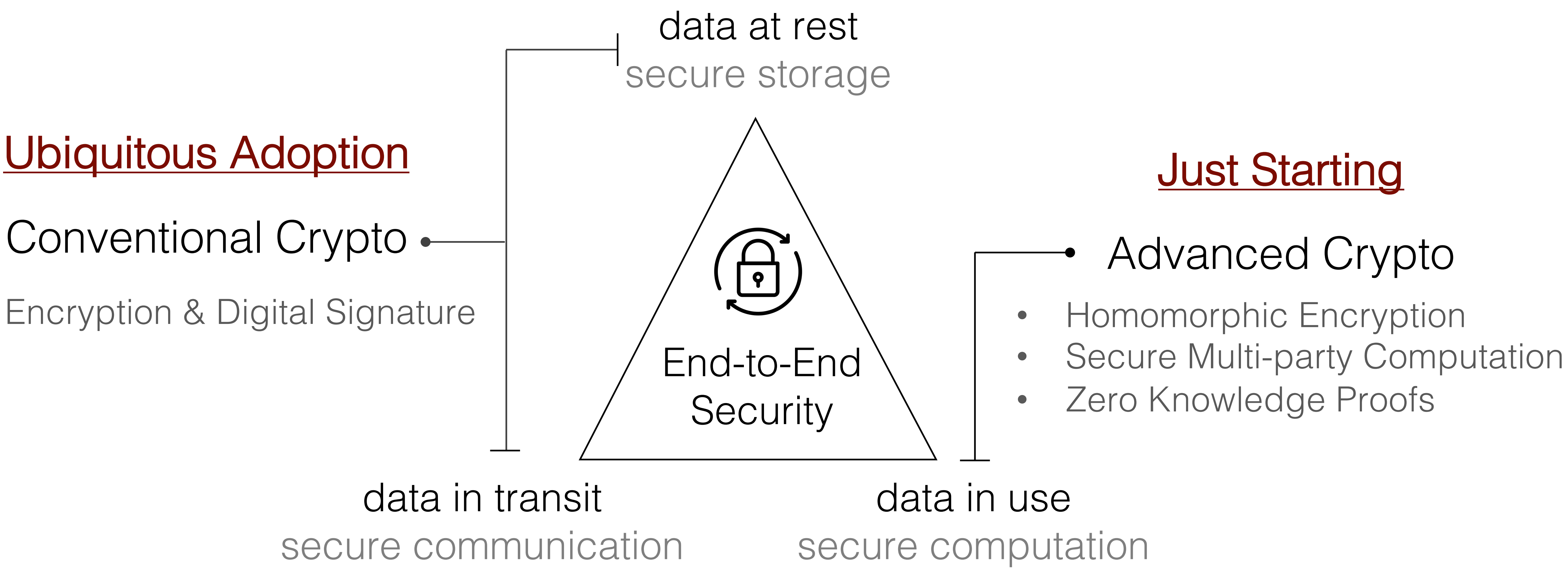


“Where the sensitive information is concentrated, that is where the spies will go. This is just a fact of life.”
former NSA official Ken Silva.

Software Vulnerabilities | Insider Threats | Physical Attacks

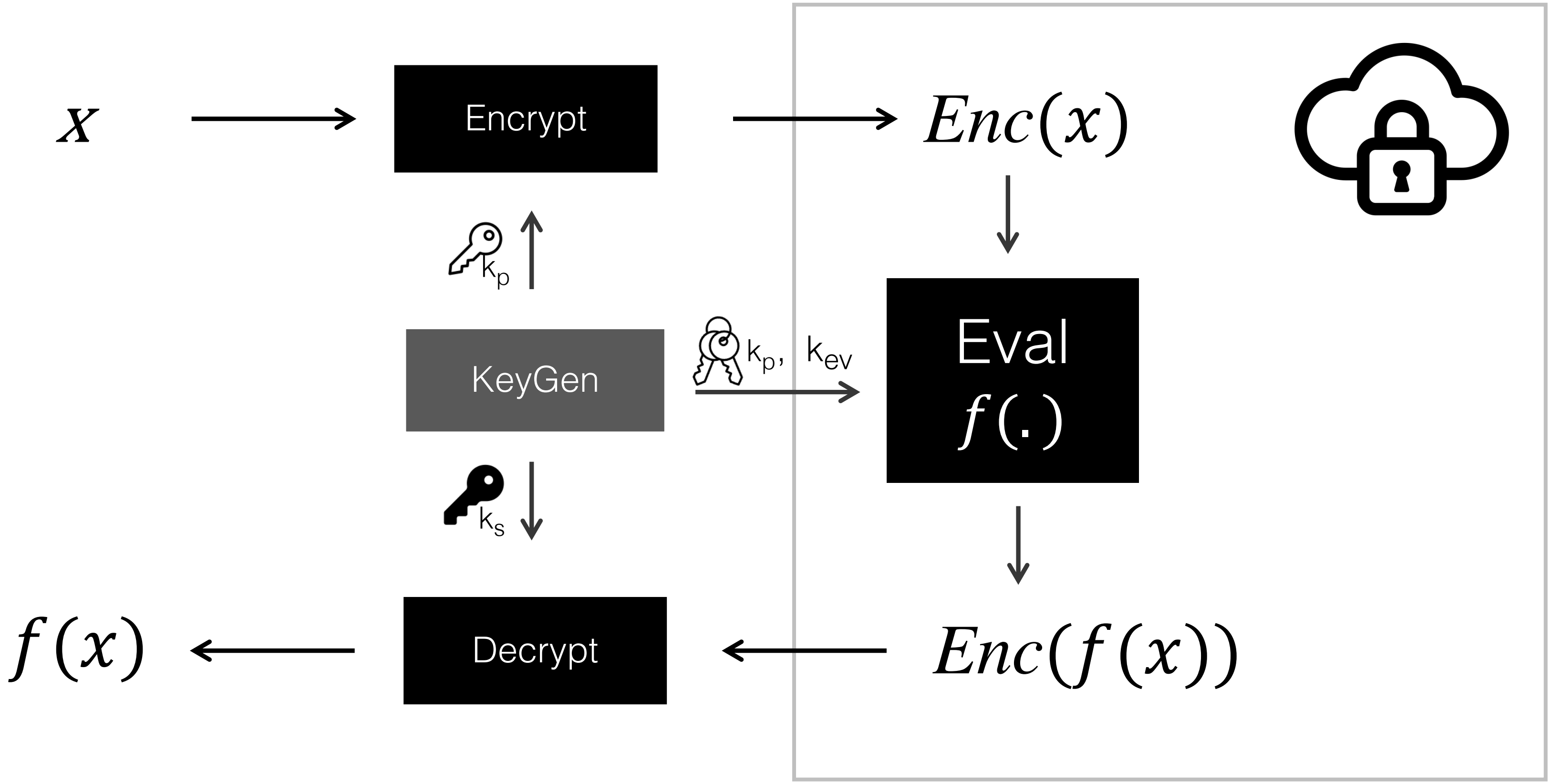
End-to-End Encrypted Systems

Modern Cryptography



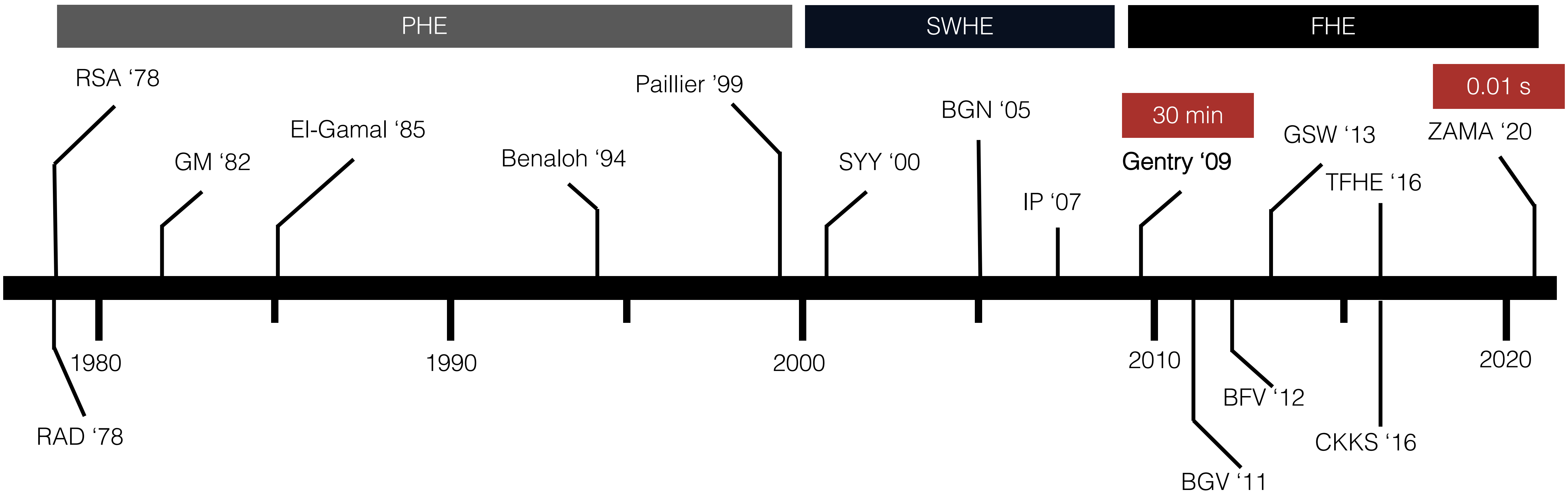
Fully Homomorphic Encryption

Enables computation on encrypted data



Delegate the processing of data without giving away access to it

40 Years of FHE History



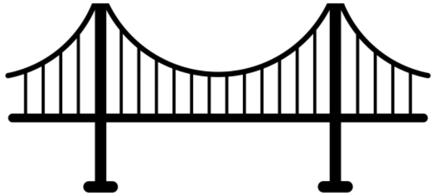
FHE will soon be practical for a wide set of applications, but …

Developing FHE Applications remains

Notoriously Hard

Usable FHE

Advanced Cryptography

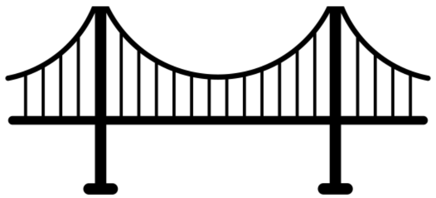


Programming Languages

1. What makes developing FHE applications hard?
2. How are compilers addressing these complexities?
3. Roadmap to End-to-End FHE development
4. HECO: Automatic Code Optimizations for FHE

Usable FHE

Advanced Cryptography

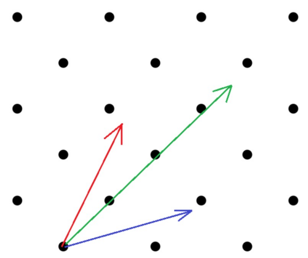


Programming Languages

1. What makes developing FHE applications hard?
2. How are compilers addressing these complexities?
3. Roadmap to End-to-End FHE development
4. HECO: Automatic Code Optimizations for FHE

FHE: Theory to Practice

$Enc(0) \circledcirc Enc(1)$

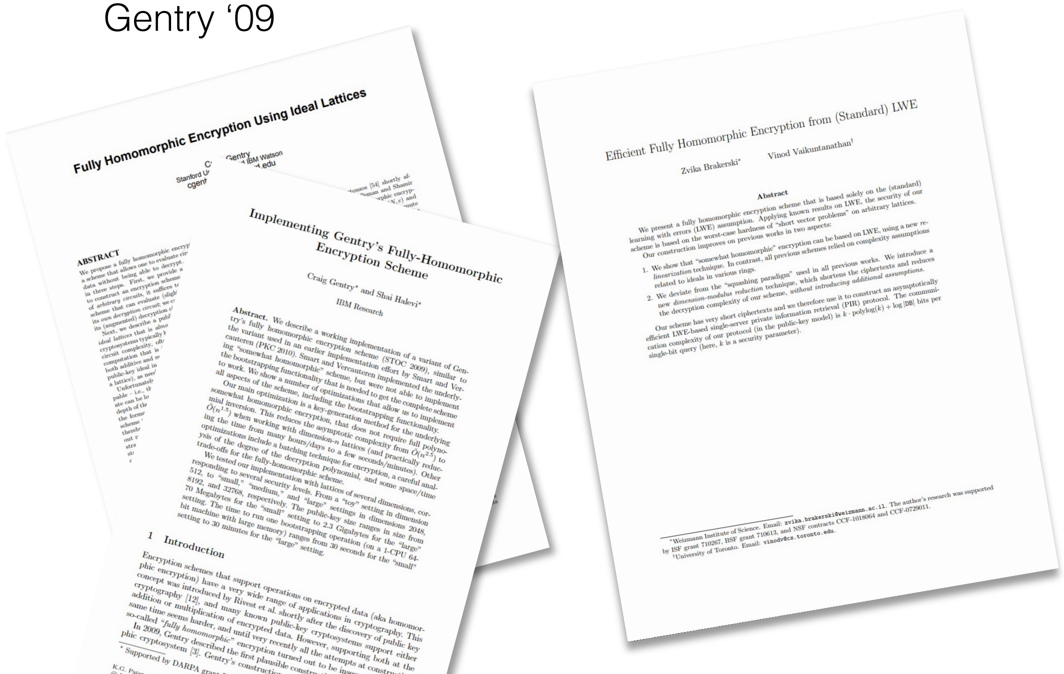


Learning **W**ith **E**rrors

$$c = \sum_{i=1}^{n} a_i s_i + e$$

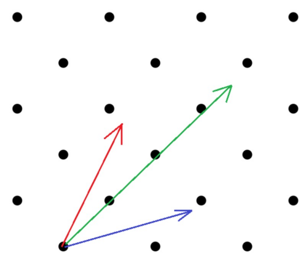
$$where\ \boldsymbol{a} \xleftarrow{\$} \mathbb{Z}_q^n, \boldsymbol{e} \xleftarrow{\chi} \mathbb{Z}_q, \boldsymbol{s} \in \mathbb{Z}_q^n$$

Gentry '09



FHE: Theory to Practice

$Enc(0) \circledcirc Enc(1)$



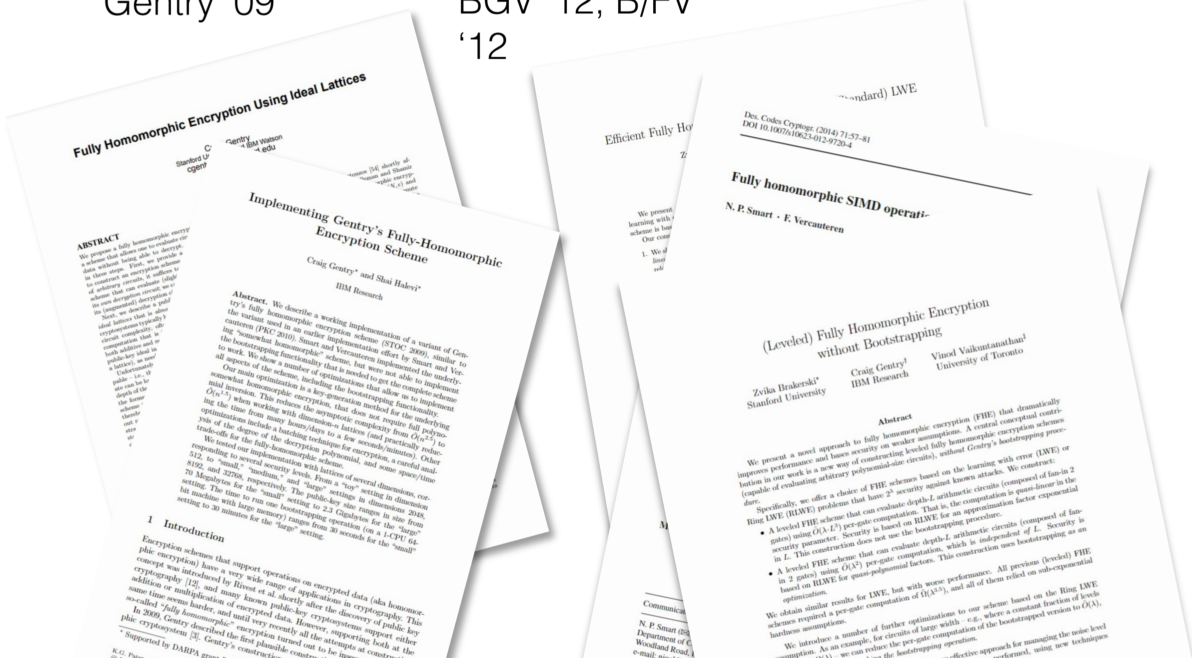
Ring-**L**earning **W**ith **E**rrors

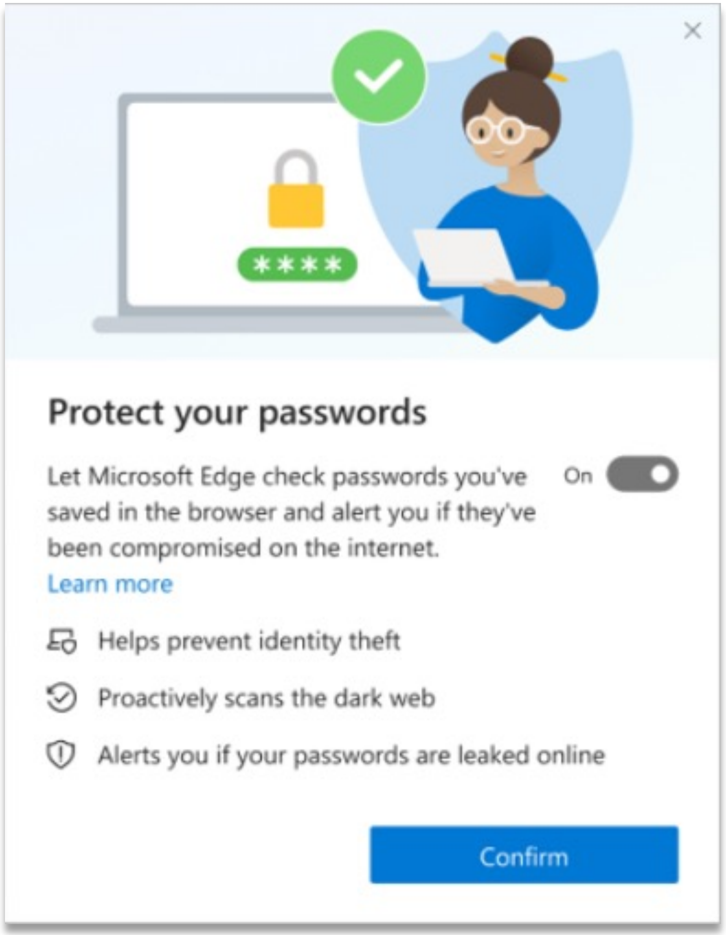
$$\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{e}$$

$where\ \boldsymbol{a} \xleftarrow{\$} R_q, \boldsymbol{e} \xleftarrow{\chi} R_q, \boldsymbol{s} \in R_q$
$R = \mathbb{Z}[X]/(X^n + 1)$

Gentry '09

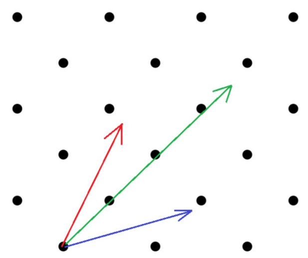
BGV '12, B/FV '12





FHE: Theory to Practice

$Enc(0) \circledcirc Enc(1)$



Ring-**L**earning **W**ith **E**rrors

$$\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{e}$$

$\textit{where } \boldsymbol{a} \stackrel{\$}{\leftarrow} R_q, \boldsymbol{e} \stackrel{\chi}{\leftarrow} R_q, \boldsymbol{s} \in R_q$

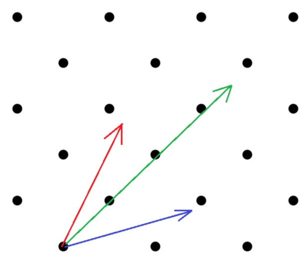
$R = \mathbb{Z}[X]/(X^n + 1)$

Gentry ‘09

BGV ‘12, B/FV ‘12

FHE: Theory to Practice

$Enc(0) \odot Enc(1)$



Ring-**L**earning **W**ith **E**rrors

$$\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{e}$$

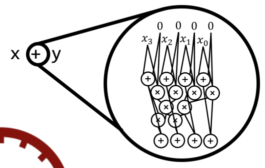
$where\ \boldsymbol{a} \xleftarrow{\$} R_q, \boldsymbol{e} \xleftarrow{\chi} R_q, \boldsymbol{s} \in R_q$

$R = \mathbb{Z}[X]/(X^n + 1)$

Gentry '09

BGV '12, B/FV '12

Fast Binary FHE



~1 OoM more operations
but each operation is incredibly fast

TFHE '16


J Cryptol (2020) 33:34–91
Journal of CRYPTOLOGY

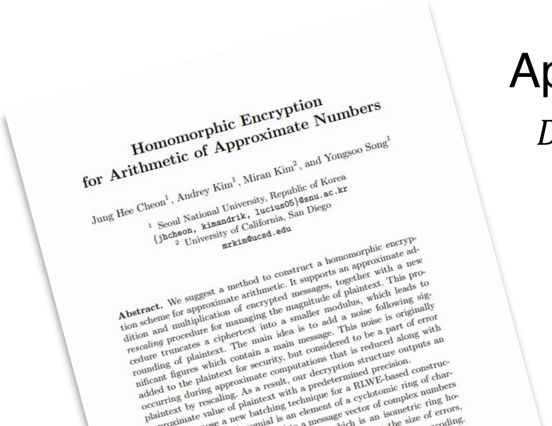
TFHE: Fast Fully Homomorphic Encryption Over the Torus*

Ilaria Chillotti
Nicolas Gama
Mariya Georgieva
Malika Izabachène

CKKS '16

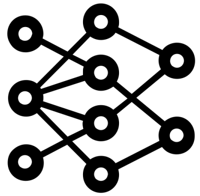
Approximate FHE

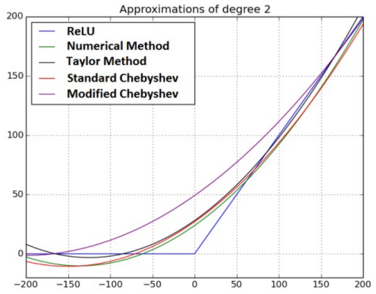
$Dec(Enc(m)) \approx m$


Homomorphic Encryption for Arithmetic of Approximate Numbers

Jung Hee Cheon[1], Andrey Kim[1], Miran Kim[2], and Yongsoo Song[1]

[1] Seoul National University, Republic of Korea
[2] University of California, San Diego



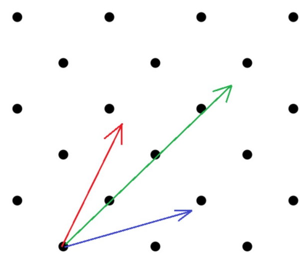


(a) Approximation of ReLU using different methods

Figure from CryptoDL: Deep Neural Networks over Encrypted Data [HTG17]

FHE: Theory to Practice

$Enc(0) \circledcirc Enc(1)$



Ring-**L**earning **W**ith **E**rrors

$$\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{e}$$

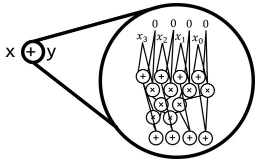
$where\ \boldsymbol{a} \xleftarrow{\$} R_q, \boldsymbol{e} \xleftarrow{\chi} R_q, \boldsymbol{s} \in R_q$

$R = \mathbb{Z}[X]/(X^n + 1)$

Gentry '09

BGV '12, B/FV '12

Fast Binary FHE



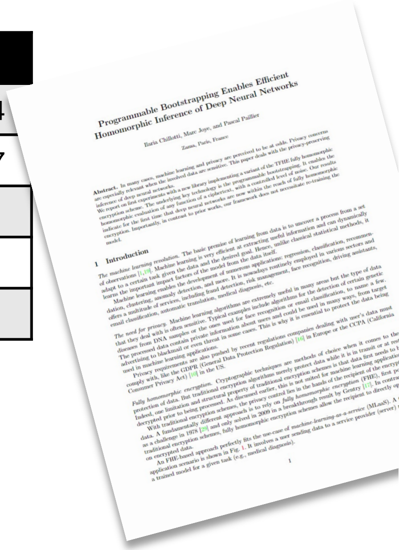
TFHE '16

ZAMA '21

DPRIVE

Look-Up Tables

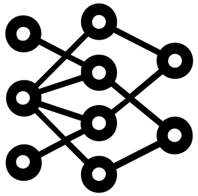
X	f(x)
-1	-0.84
-0.5	-0.47
0	0
0.5	0.47
1	0.84



CKKS '16

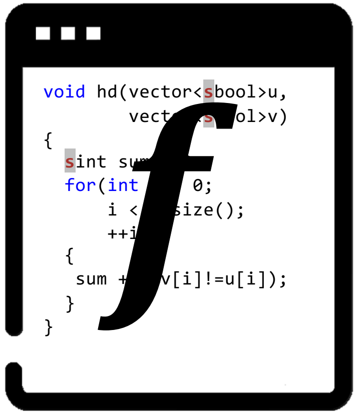
Approximate FHE

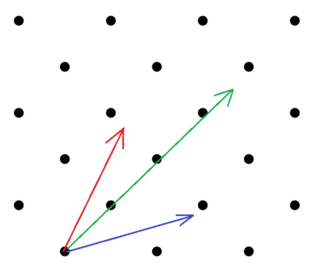
$Dec\big(Enc(m)\big) \approx m$



Hardware Acceleration

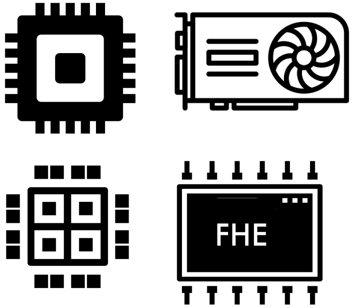
intel. Duality galois SRI International

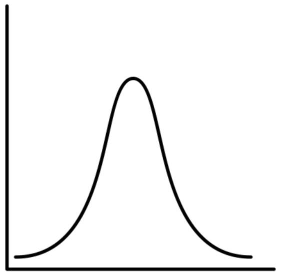




Math

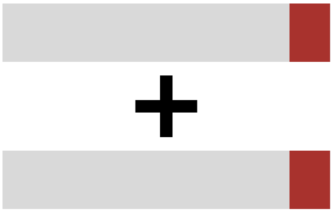
$$R = \mathbb{Z}[X]/(X^n + 1)$$

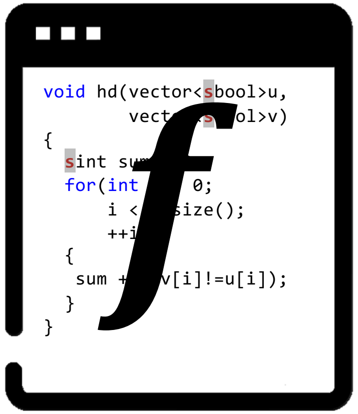


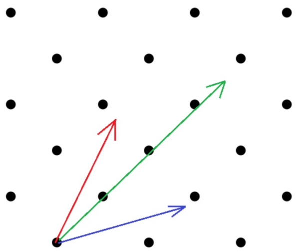


Noise

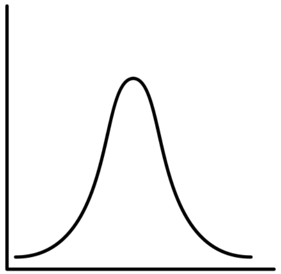
$$\boldsymbol{a} * \boldsymbol{s} + \boldsymbol{m} + \boldsymbol{e}$$





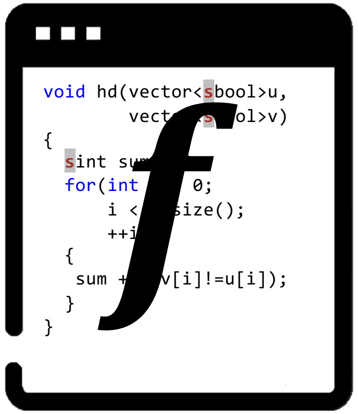


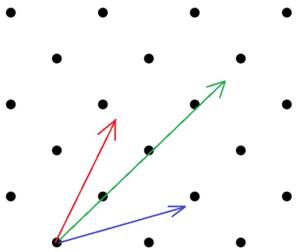
Math



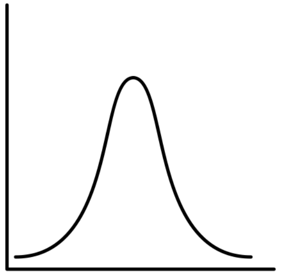
Noise

$$a * s + m + e$$



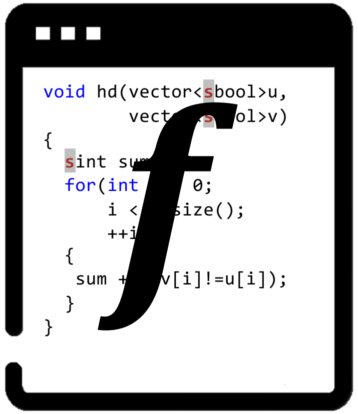


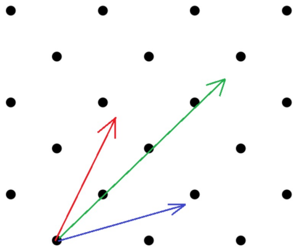
Math



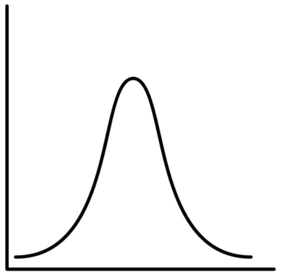
Noise

$$a * s + m + e$$



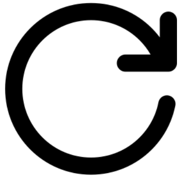


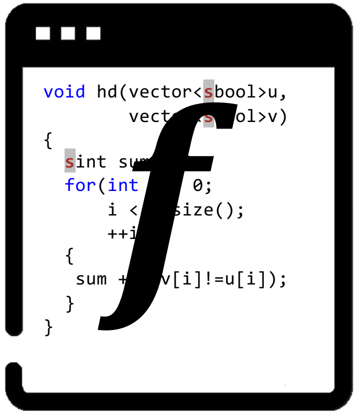
Math

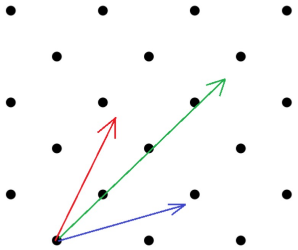


Noise

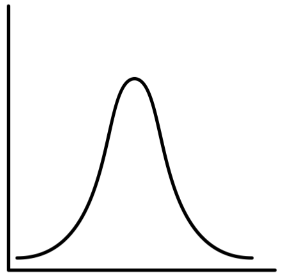
$$a * s + m + e$$





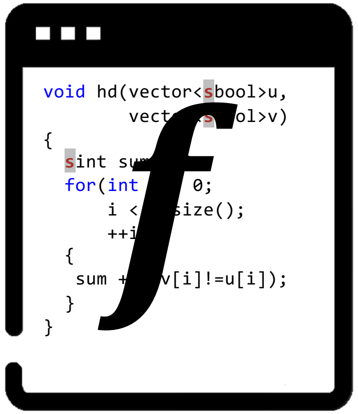


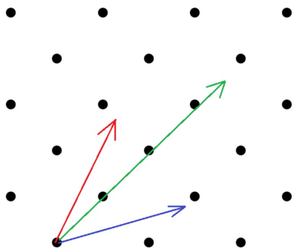
Math

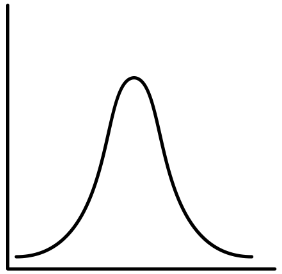


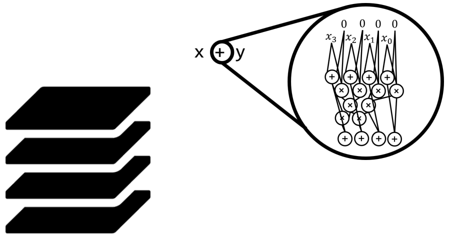
Noise

$$a * s + m + e$$









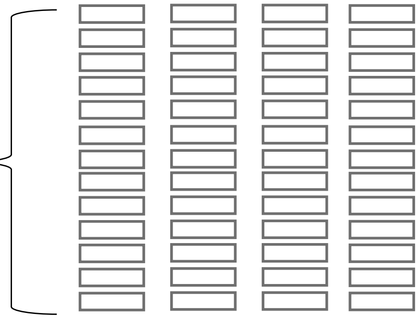
Math

Noise

Representations

x[0],x[1],x[2], …

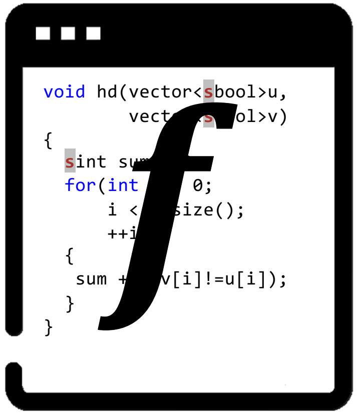
$N = 32768$

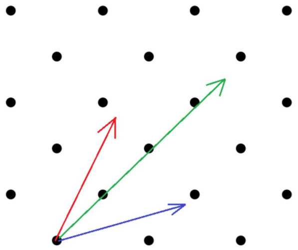


Enc(x[0])

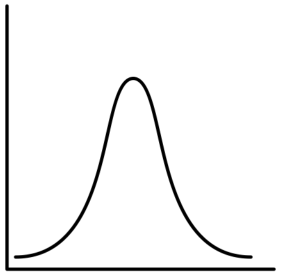
Enc(x[1])

Enc(x[2])…

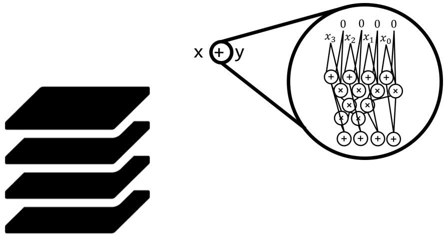




Math



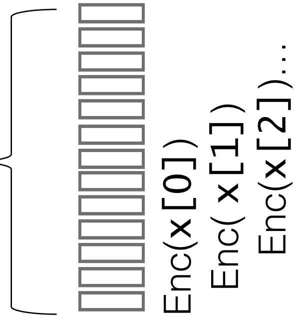
Noise

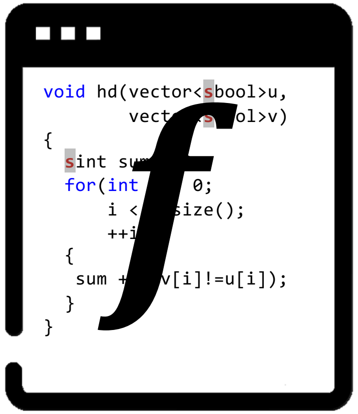


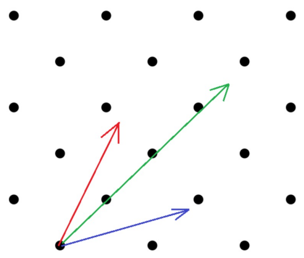
Representations

x[0],x[1],x[2], …

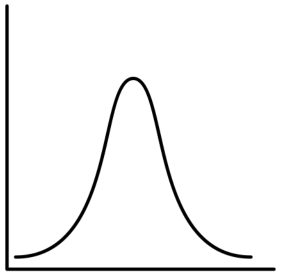
$N = 32768$



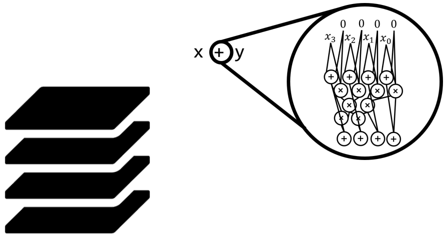




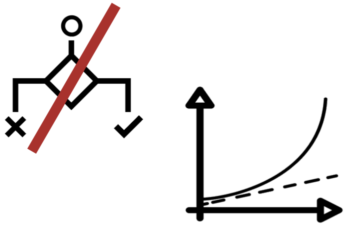
Math



Noise



Representations



Applications

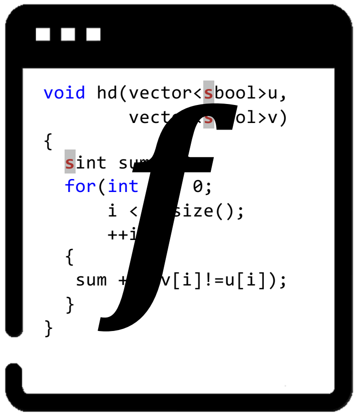
$$\chi^2 = N \sum_{i=1}^{n} \frac{(O_i/N - p_i)^2}{p_i}$$

$\alpha = (4N_0 N_2 - N_1^2)^2$

$\beta_1 = 2(2N_0+N_1)^2$

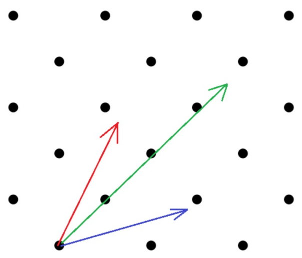
$\beta_2 = (2N_0+N_1)(2N_2+N_1)$

$\beta_3 = 2(2N_2+N_1)^2$

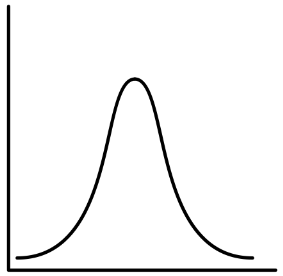


Functionality and performance depend on ***f***’s representation:

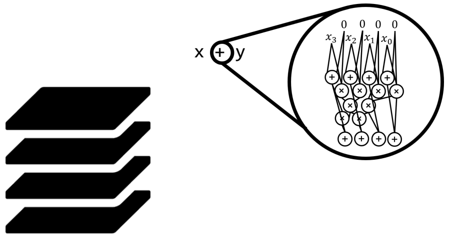
- How do we express ***f***
- How do we optimize ***f***



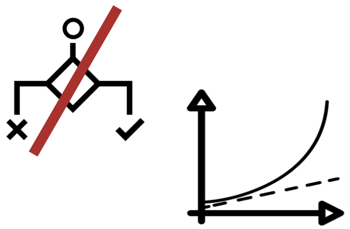
Math



Noise



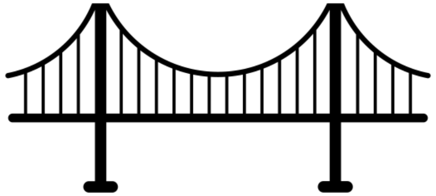
Representations



Applications

Usable FHE

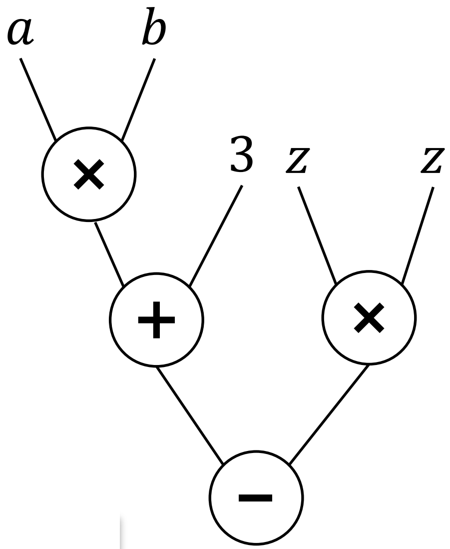
Advanced Cryptography



Programming Languages

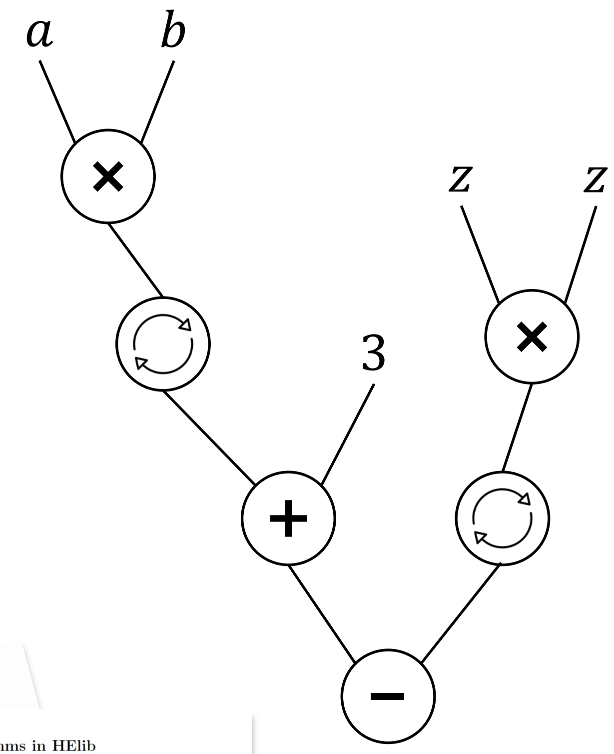
1. What makes developing FHE applications hard?
2. **How are compilers addressing these complexities?**
3. Roadmap to End-to-End FHE development
4. HECO: Automatic Code Optimizations for FHE

Evolution of FHE Tools

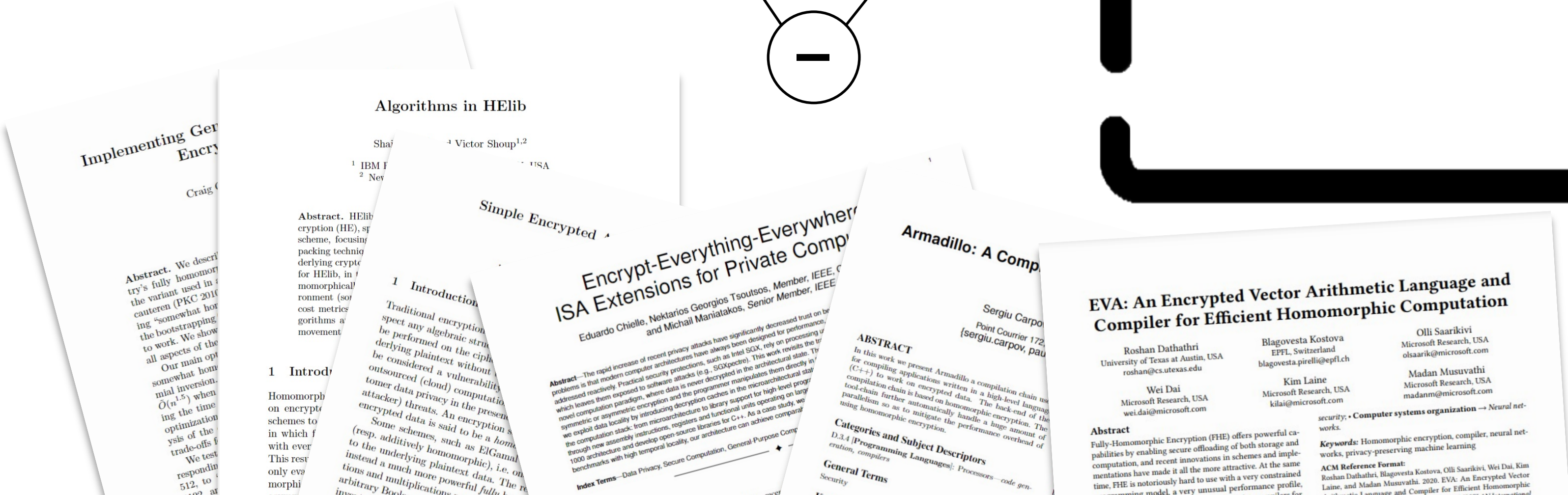



```
void f(...)
{
 mul_inp(a,b);
 relin_inp(a);
 add_plain_inp(a,3)
 square_inp(z,z);
 relin_inp(a);
 sub_inp(a,z);
 return a;
}
```

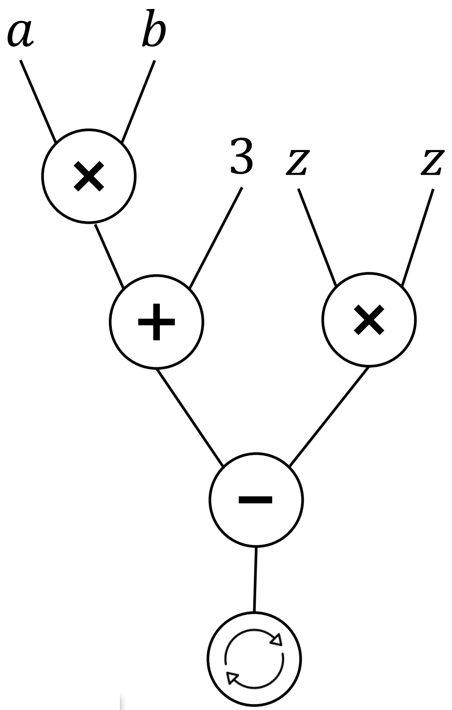
Evolution of FHE Tools

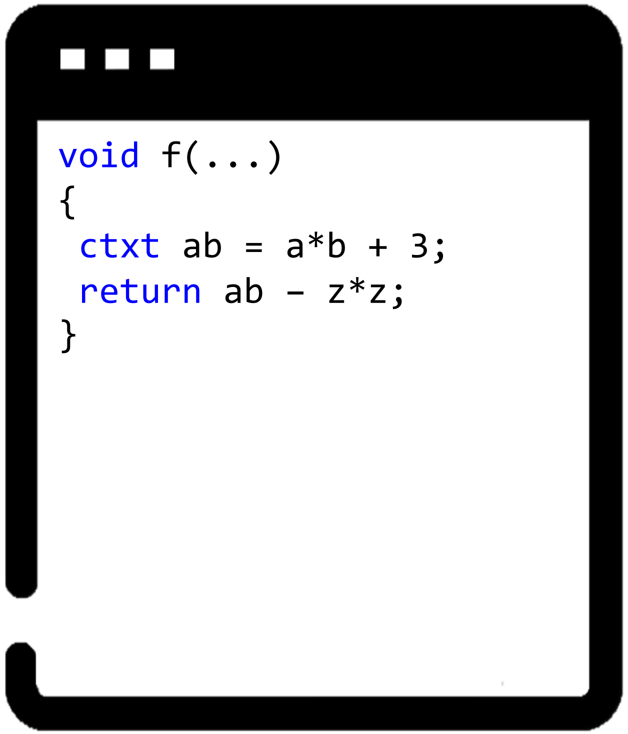



```
void f(...)
{
 ctxt ab = a*b + 3;
 return ab - z*z;
}
```



Evolution of FHE Tools





SoK: Fully Homomorphic Encryption Compilers

Alexander Viand, ETH Zurich, alexander.viand@inf.ethz.ch
Patrick Jattke, ETH Zurich, pjattke@ethz.ch
Anwar Hithnawi, ETH Zurich, anwar.hithnawi@inf.ethz.ch

Abstract—Fully Homomorphic Encryption (FHE) allows a third party to perform arbitrary computations on encrypted data, learning neither the inputs nor the computation results. Hence, it provides resilience in situations where computations are carried out by an untrusted or potentially compromised party. This powerful concept was first conceived by Rivest et al. in the 1970s. However, it remained unrealized until Craig Gentry presented the first feasible FHE scheme in 2009.

The advent of the massive collection of sensitive data in cloud services, coupled with a plague of data breaches, moved highly regulated businesses to increasingly demand confidential and secure computing solutions. This demand, in turn, has led to a recent surge in the development of FHE tools. To understand the landscape of recent FHE tool developments, we conduct an extensive survey and experimental evaluation to explore the current state of the art and identify areas for future development.

In this paper, we survey, evaluate, and systematize FHE tools and compilers. We perform experiments to evaluate these tools' performance and usability aspects on a variety of applications. We conclude with recommendations for developers intending to develop FHE-based applications and a discussion on future directions for FHE tools development.

I. Introduction

Recent years have seen unprecedented growth in the adoption of cloud computing services. More and more highly regulated businesses and organizations (e.g., banks, governments, insurances, health), where data security is paramount, move their data and services to the cloud. This trend has led to a surge in demand for secure and confidential computing solutions that protect data confidentiality while in transit, rest, and in-use. This is an amply justified and expected demand, particularly in the light of the numerous reports of data breaches [1], [2]. Fully Homomorphic Encryption (FHE) is a key technological enabler for secure computation and has recently matured to be practical for real-world use [3]–[9].

FHE allows arbitrary computations to be performed over encrypted data, eliminating the need to decrypt the data and expose it to potential risk while in use. While first proposed in the 1970s [10], FHE was long considered impossible or impractical. However, thanks to advances in the underlying theory, general hardware improvements, and more efficient implementations, it has become increasingly practical. In 2009, breakthrough work from Craig Gentry proposed the first feasible FHE scheme [11]. In the last decade, FHE has gone from a theoretical concept to reality, with performance improving by up to five orders of magnitude. For example, times for a multiplication between ciphertexts dropped from 30 minutes to less than 20 milliseconds. While this is still around seven orders of magnitude slower than an IMUL instruction on a modern CPU, it is sufficient to make many applications practical. Additionally, modern schemes introduced SIMD-style parallelism, encoding thousands of plaintext values into a single ciphertext to further improve throughput [12].

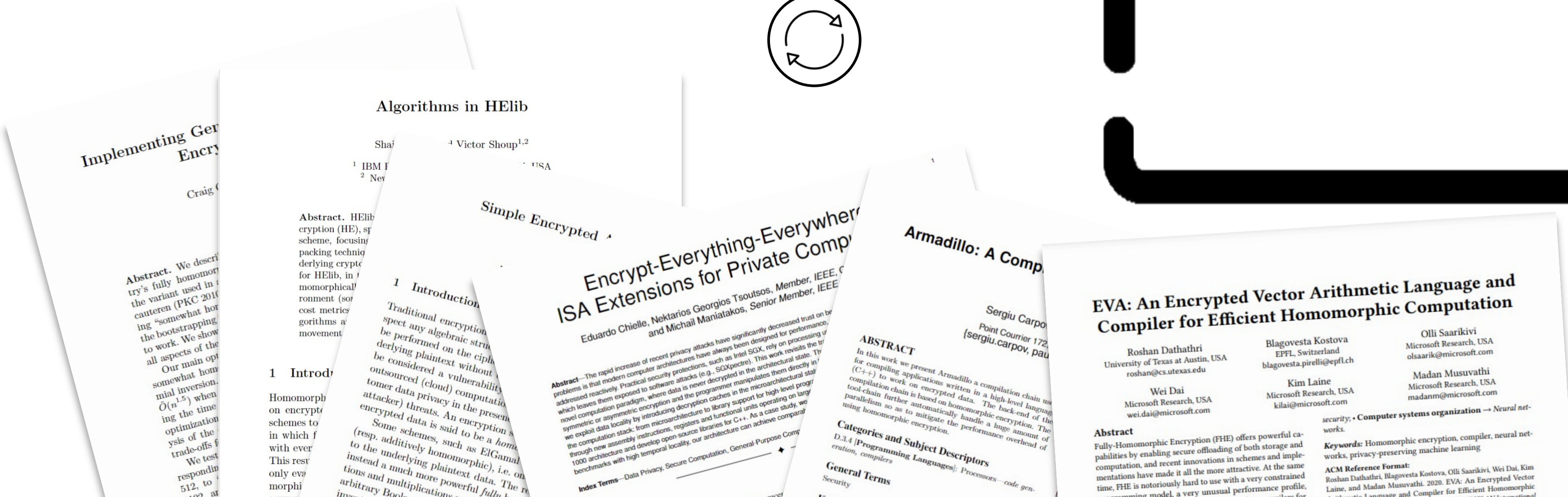
These advances have enabled a wide range of applications covering a wide range of domains. These include mobile applications, where FHE has been used to encrypt the back end of a privacy-preserving fitness app [13], while continuing to provide a real-time experience. In the medical domain, FHE has been used to enable privacy-preserving genome analysis [14] applications over large datasets. More generally, FHE has been used to solve various well-known problems like Private Set Intersection (PSI) [15], outperforming previous solutions by 2× in running time. In the domain of machine learning, FHE has been used for tasks ranging from linear and logistic regression [16] to Encrypted Neural Network inference [17], which can be used to run privacy-preserving ML-as-a-Service applications, for example, for private phishing email detection [18]. As a consequence, there has been increasing interest in FHE-based secure computation solutions [3]–[9]. Gartner projects [19] that "by 2025, at least 20% of companies will have a budget for projects that include fully homomorphic encryption."

Despite these recent breakthroughs, building secure and efficient FHE-based applications remains a challenging task. This is largely attributed to the differences between traditional programming paradigms and FHE's computation model, which poses unique challenges. For example, virtually all standard programming paradigms rely on data-dependent branching, e.g., if/else statements and loops. On the other hand, FHE computations are, by definition, data-independent, or they would violate the privacy guarantees. Working with FHE also introduces significant engineering challenges in practice. Different schemes offer varying performance tradeoffs, and optimal choices are heavily application-dependent. To address some of the engineering challenges in this space, we have seen a surge of work on tools that aim to improve accessibility and reduce barriers to entry in this field.

Without tool support, realizing FHE-based computations by implementing the required mathematical operations directly or using an arbitrary-precision arithmetic library is complex, requiring considerable expertise in both cryptography and high-performance numerical computation. Therefore, FHE libraries like the Simple Encrypted Arithmetic Library (SEAL) [20] or the Fast Fully Homomorphic Encryption Library over the

1

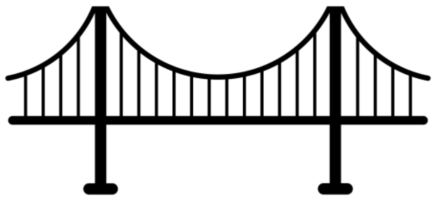
S&P 2021, arXiv 2101.07078



Existing tools make important contributions,
but are too **narrowly focussed**

Usable FHE

Advanced Cryptography

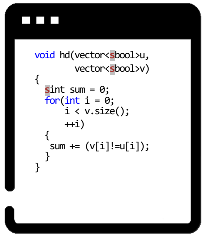


Programming Languages

1. What makes developing FHE applications hard?
2. How are compilers addressing these complexities?
3. Roadmap to End-to-End FHE development
4. HECO: Automatic Code Optimizations for FHE

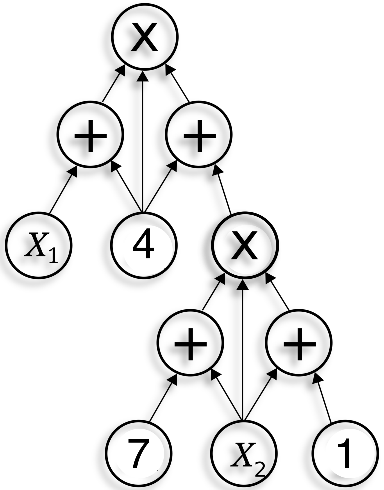
End-to-End FHE Toolchain

Computer Program



FHE Compiler

Arithmetic Circuit

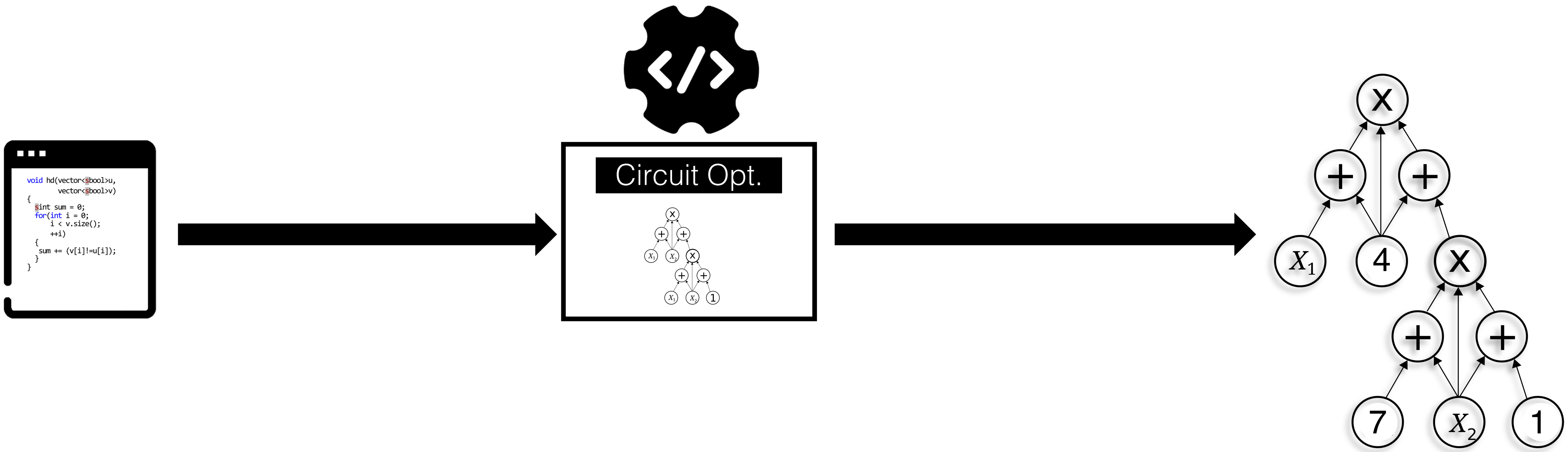


End-to-End FHE Toolchain

Computer Program

FHE Compiler

Arithmetic Circuit

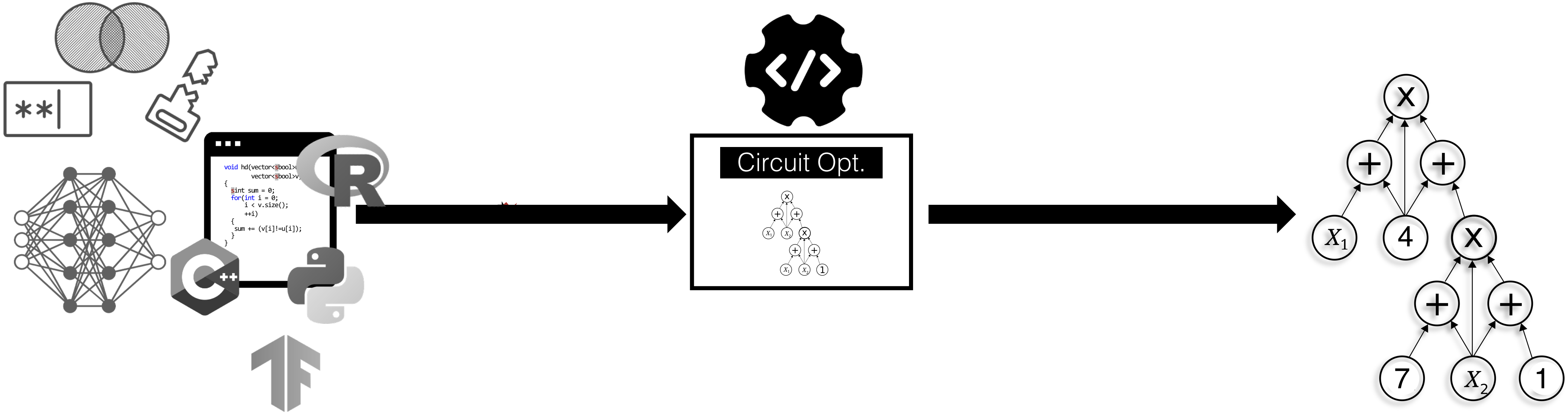


End-to-End FHE Toolchain

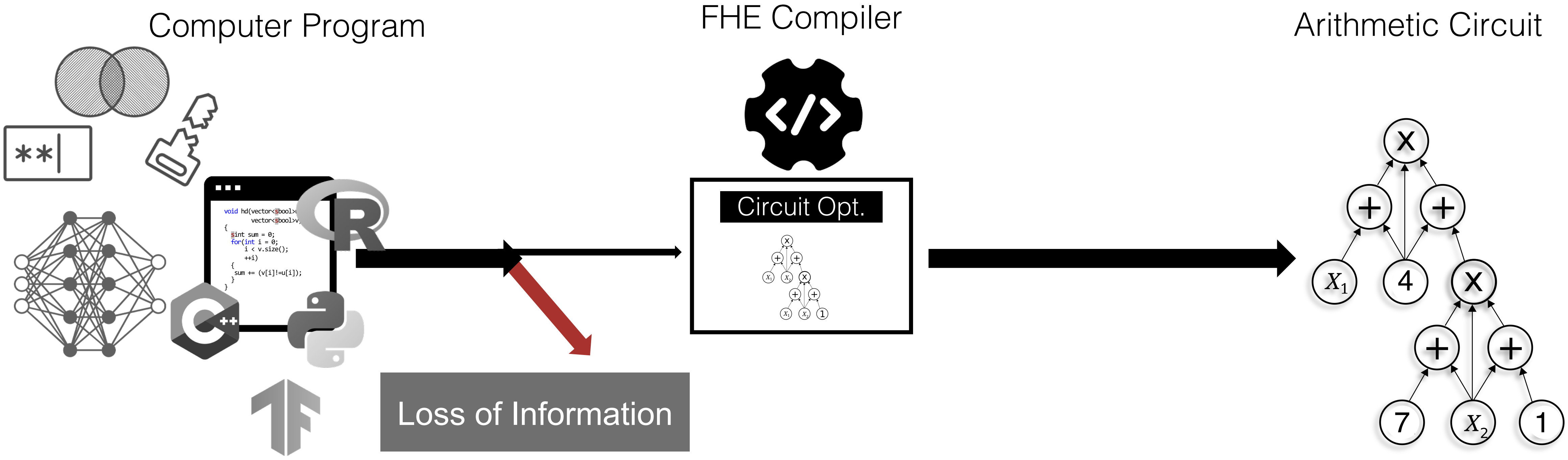
Computer Program

FHE Compiler

Arithmetic Circuit



End-to-End FHE Toolchain

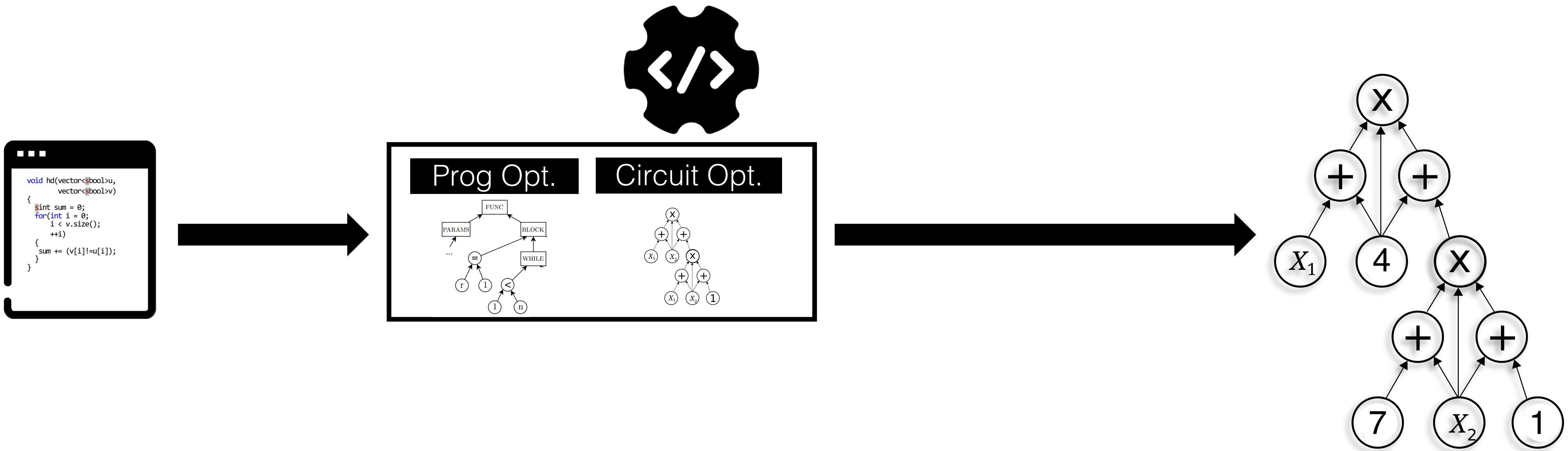


End-to-End FHE Toolchain

Computer Program

FHE Compiler

Arithmetic Circuit

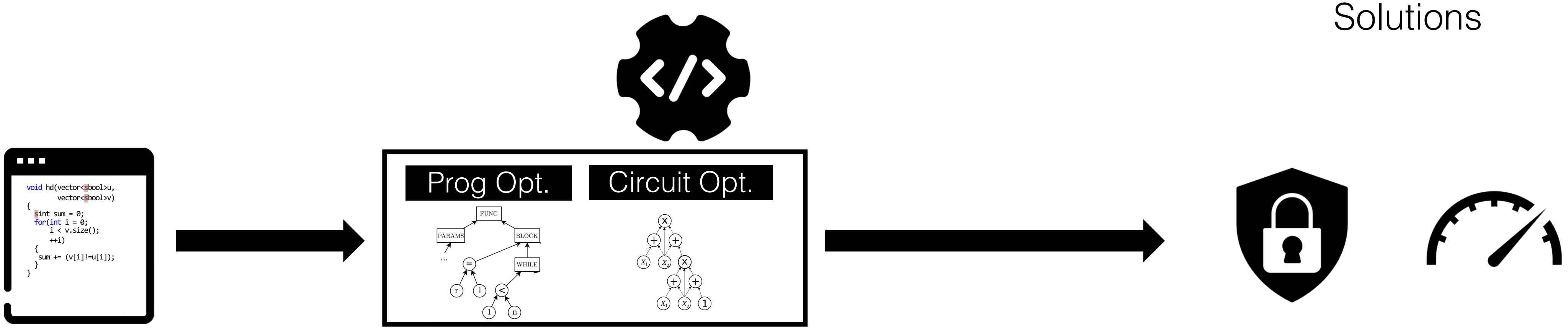


End-to-End FHE Toolchain

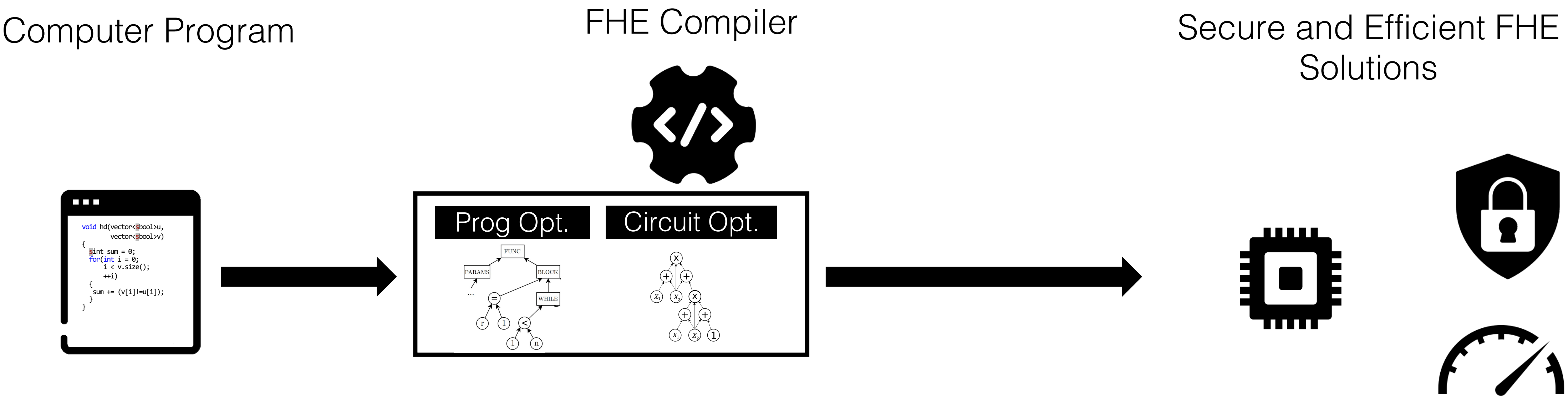
Computer Program

FHE Compiler

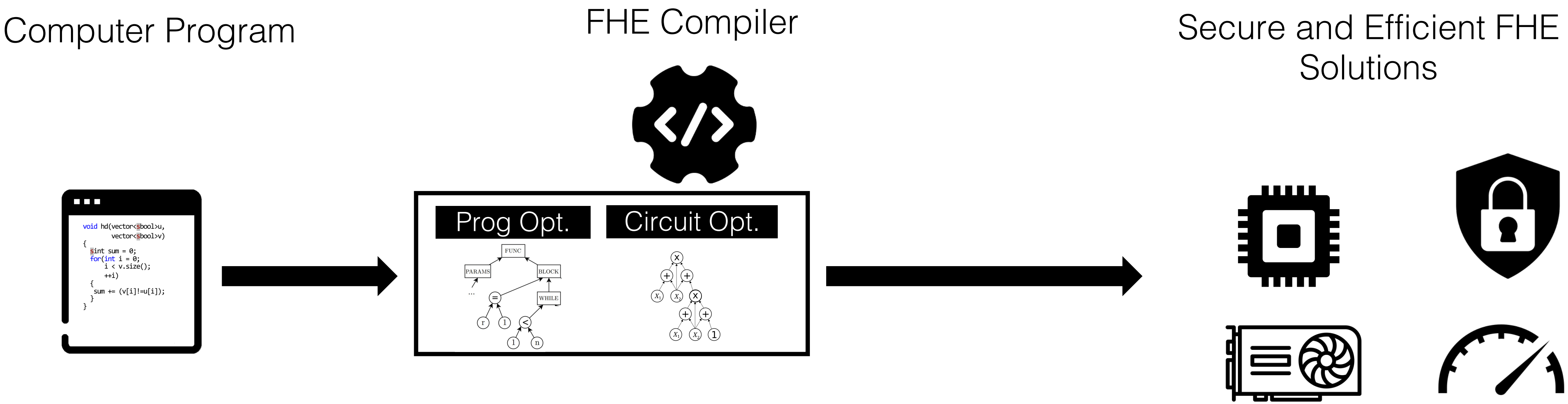
Secure and Efficient FHE Solutions



End-to-End FHE Toolchain



End-to-End FHE Toolchain

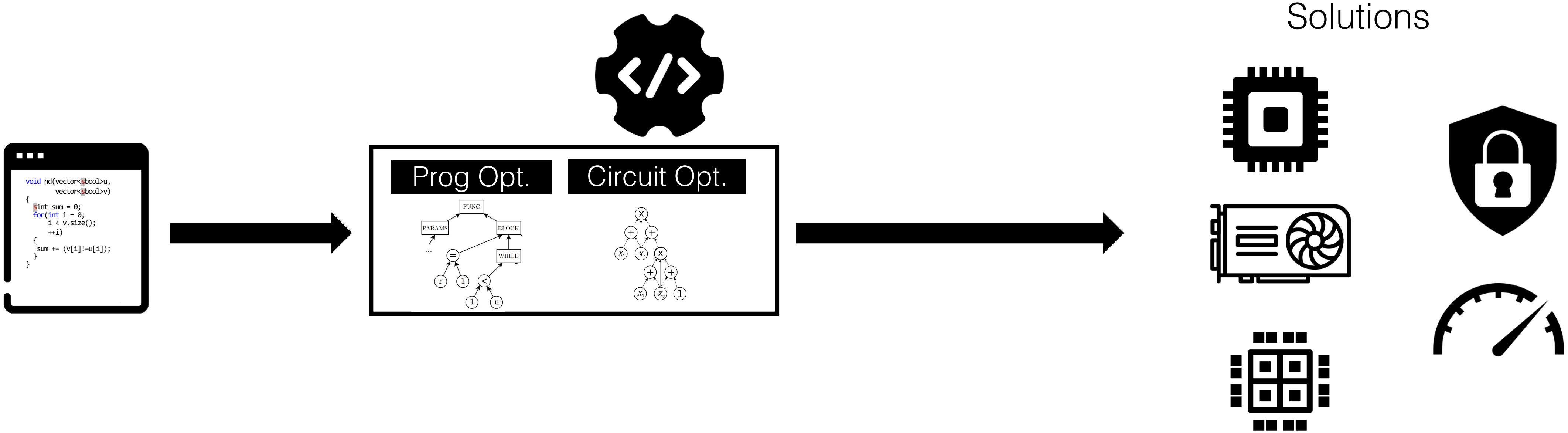


End-to-End FHE Toolchain

Computer Program

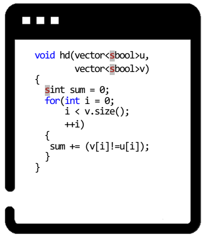
FHE Compiler

Secure and Efficient FHE Solutions

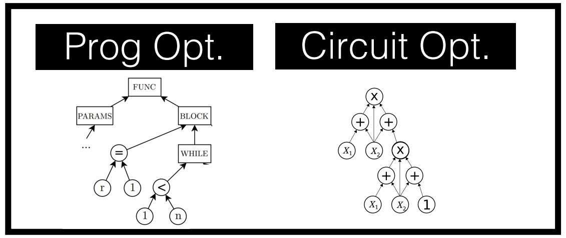


End-to-End FHE Toolchain

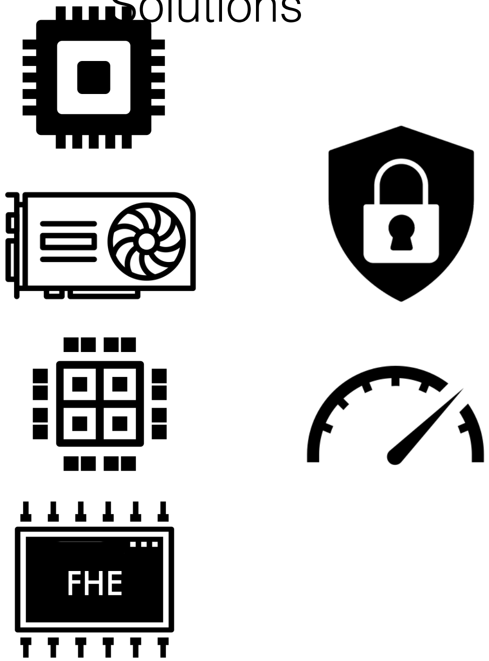
Computer Program



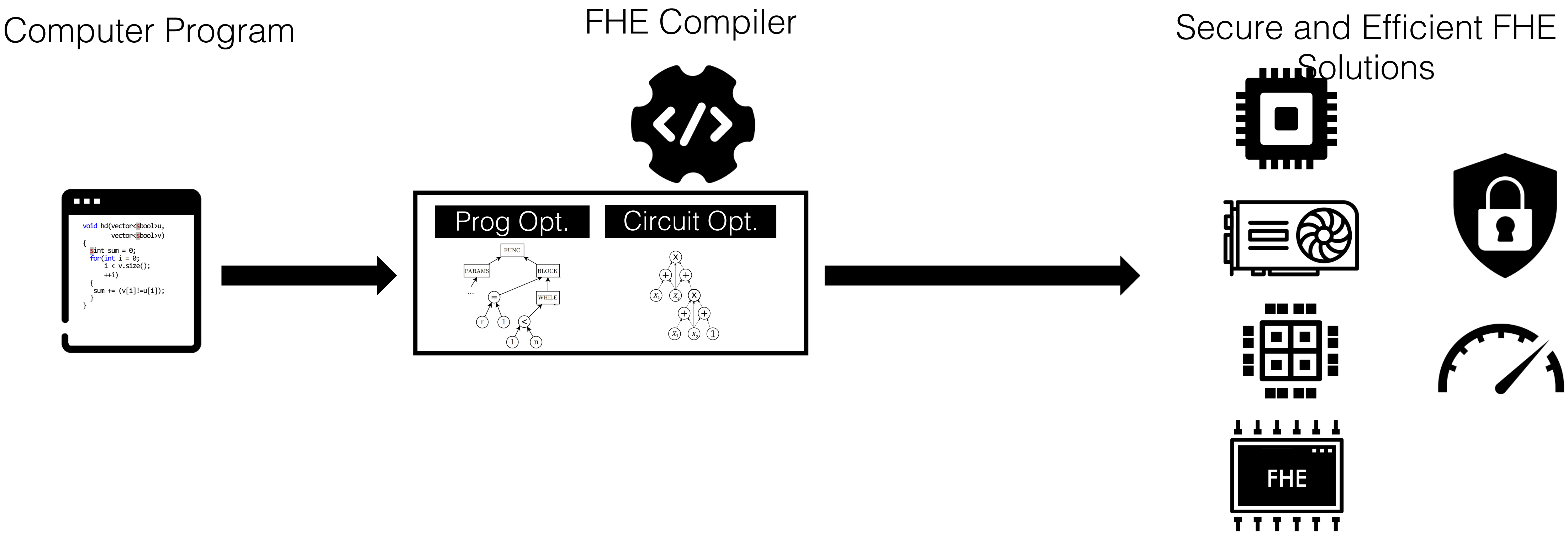
FHE Compiler



Secure and Efficient FHE Solutions



End-to-End FHE Toolchain

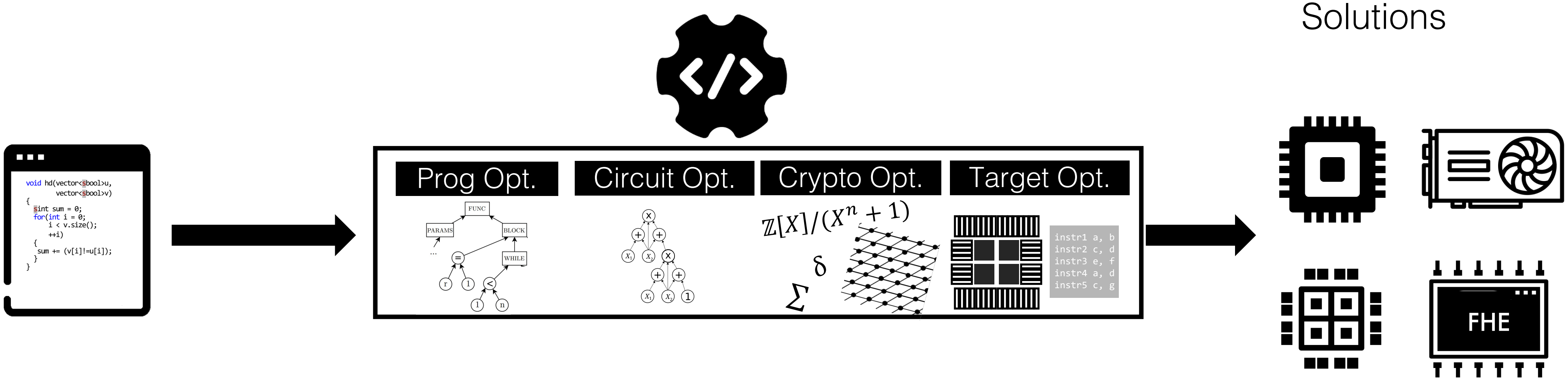


End-to-End FHE Toolchain

Computer Program

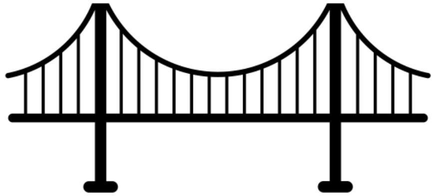
FHE Compiler

Secure and Efficient FHE Solutions



Usable FHE

Advanced Cryptography



Programming Languages

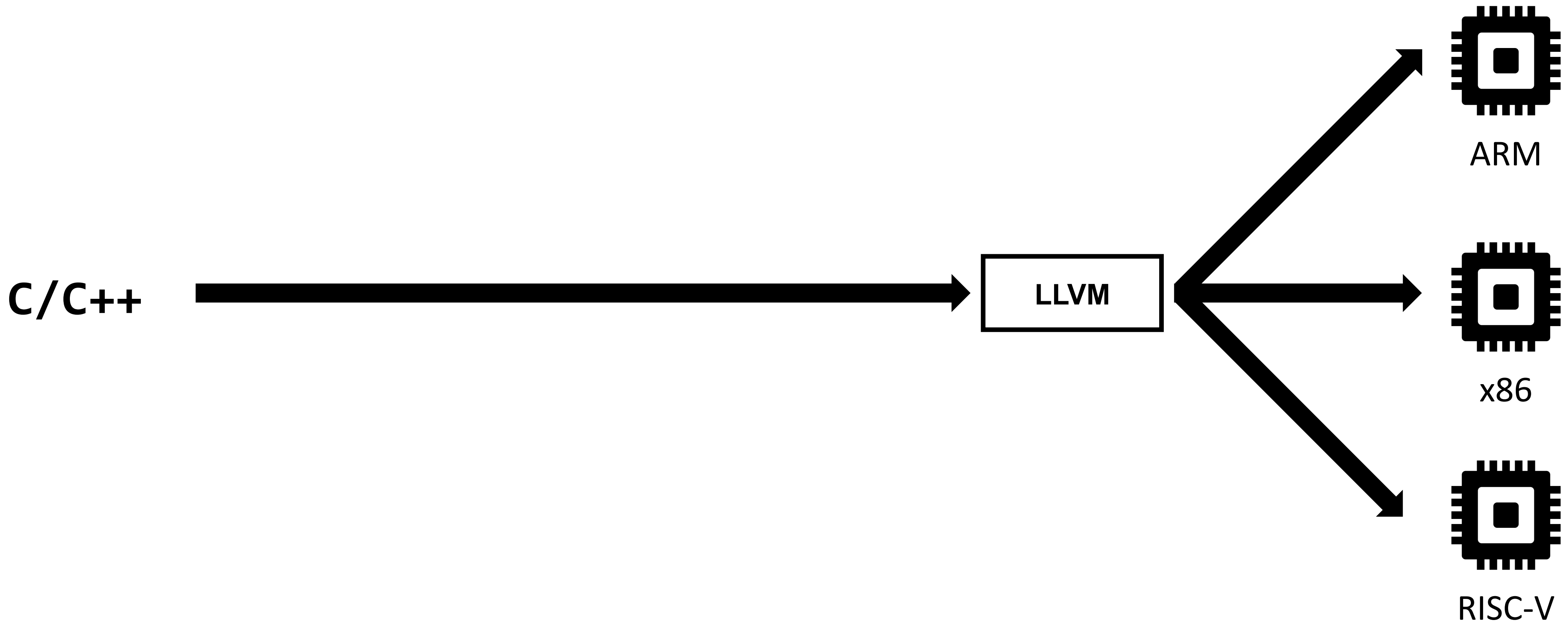
1. What makes developing FHE applications hard?
2. How are compilers addressing these complexities?
3. Roadmap to End-to-End FHE development
4. **HECO: Automatic Code Optimizations for FHE**

Multi-Stage Compilation

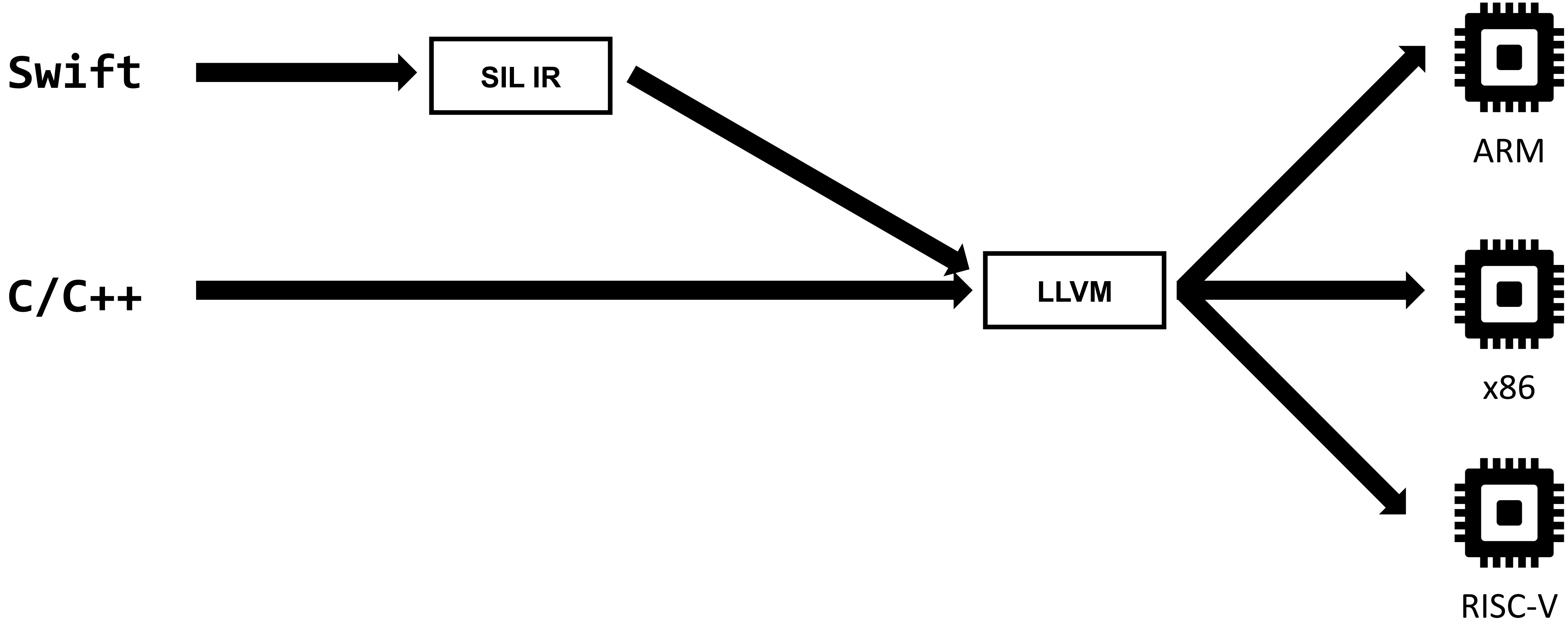
Compiler Pipelines

Compiler Pipelines

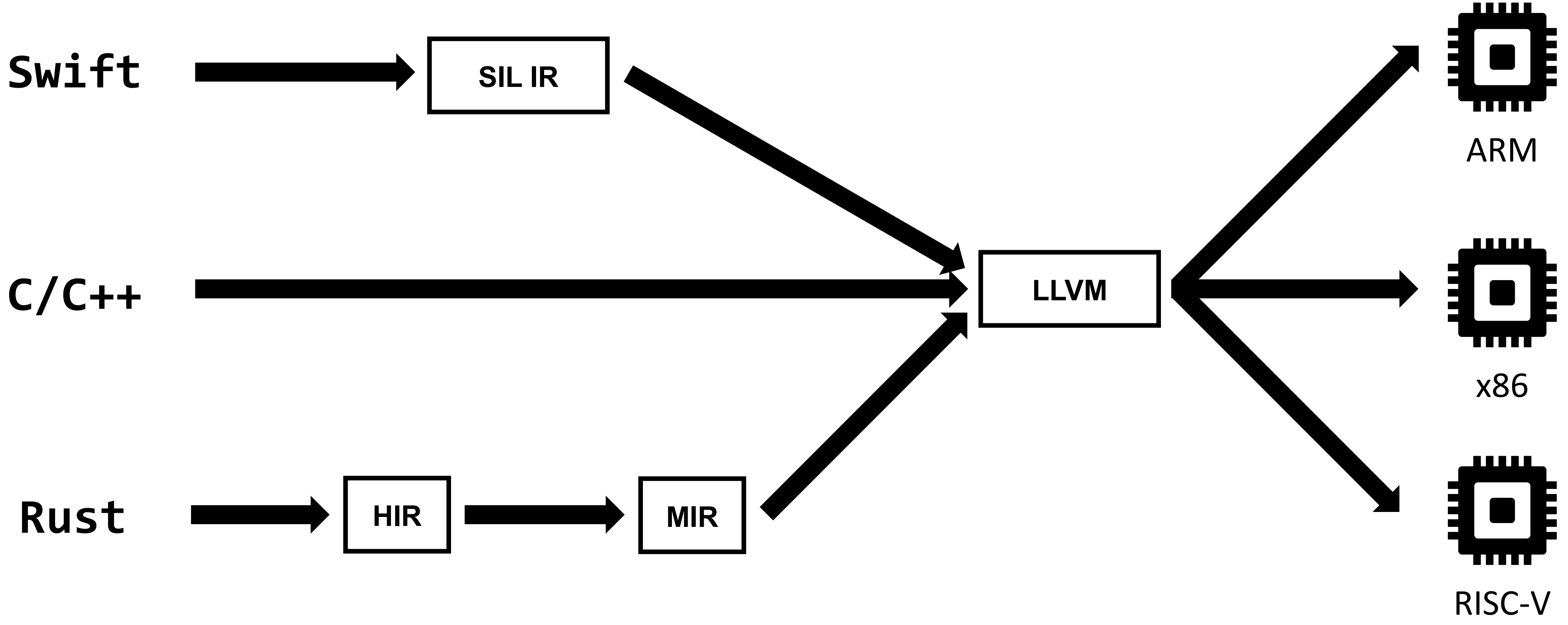
Compiler Pipelines



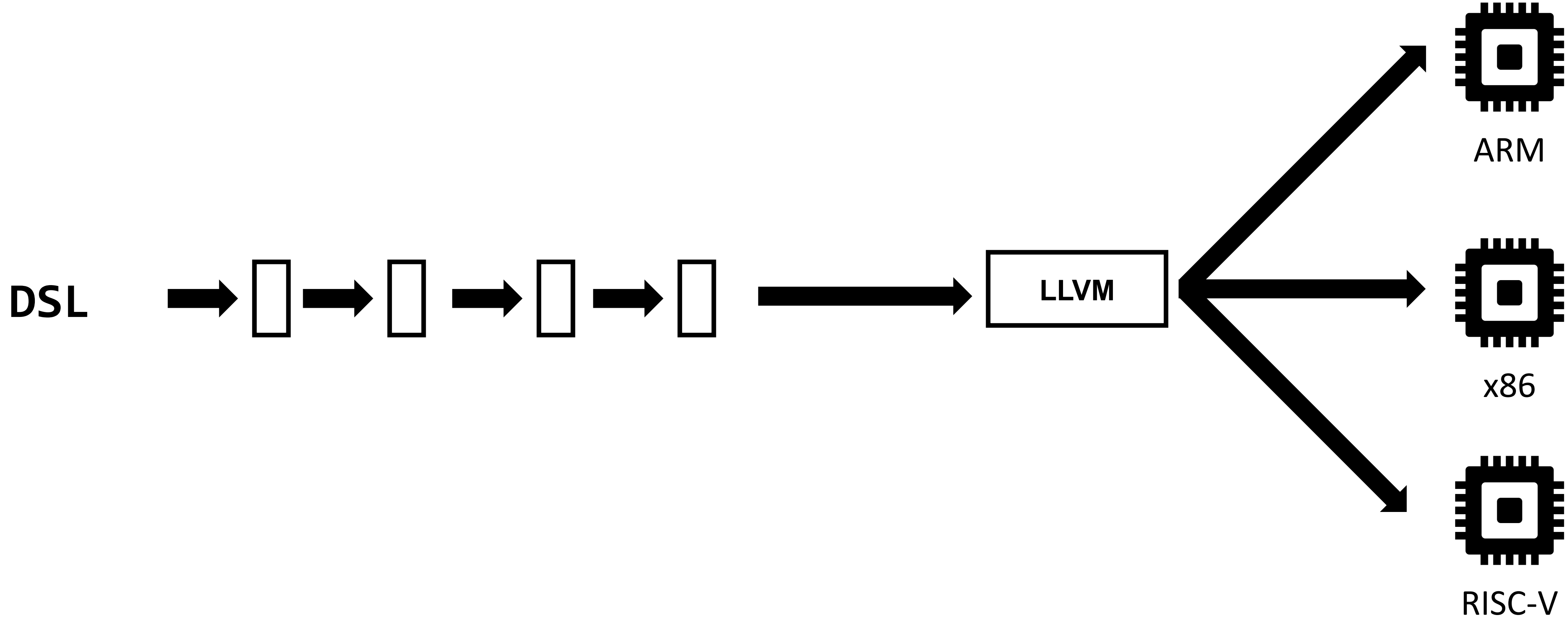
Compiler Pipelines



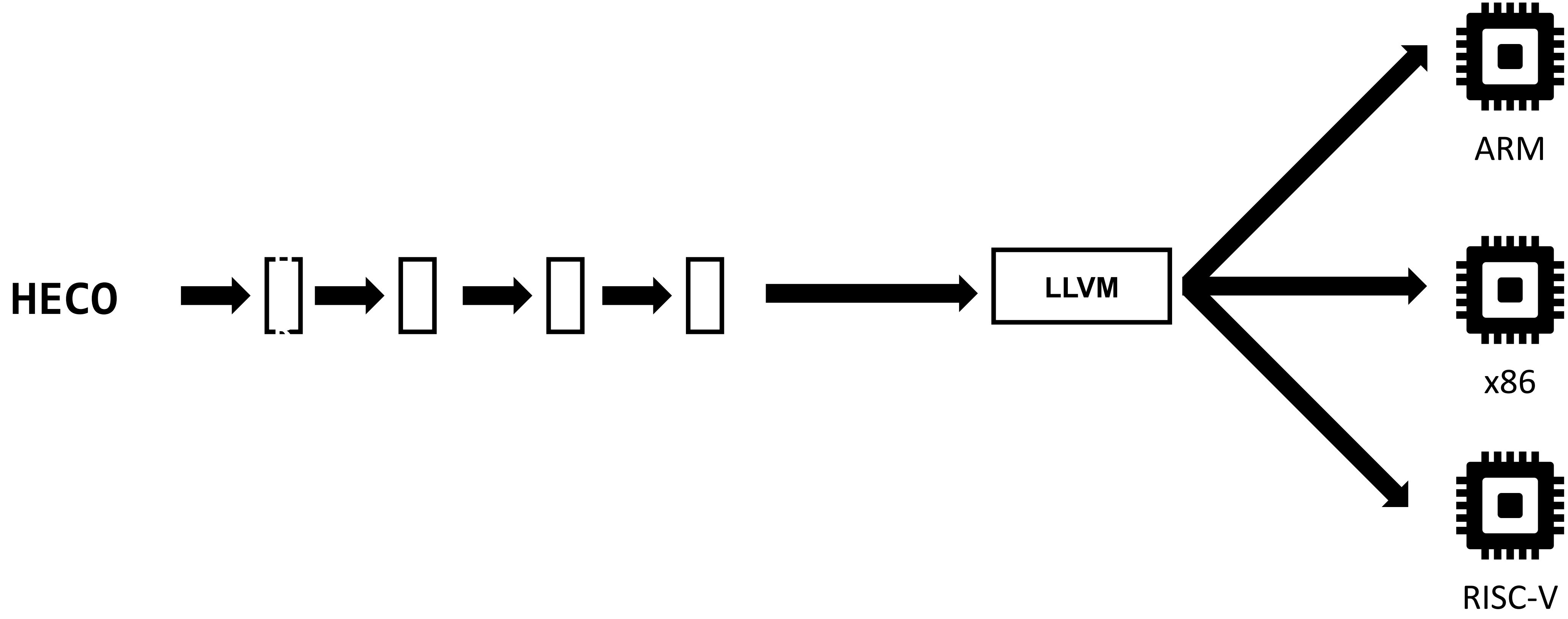
Compiler Pipelines



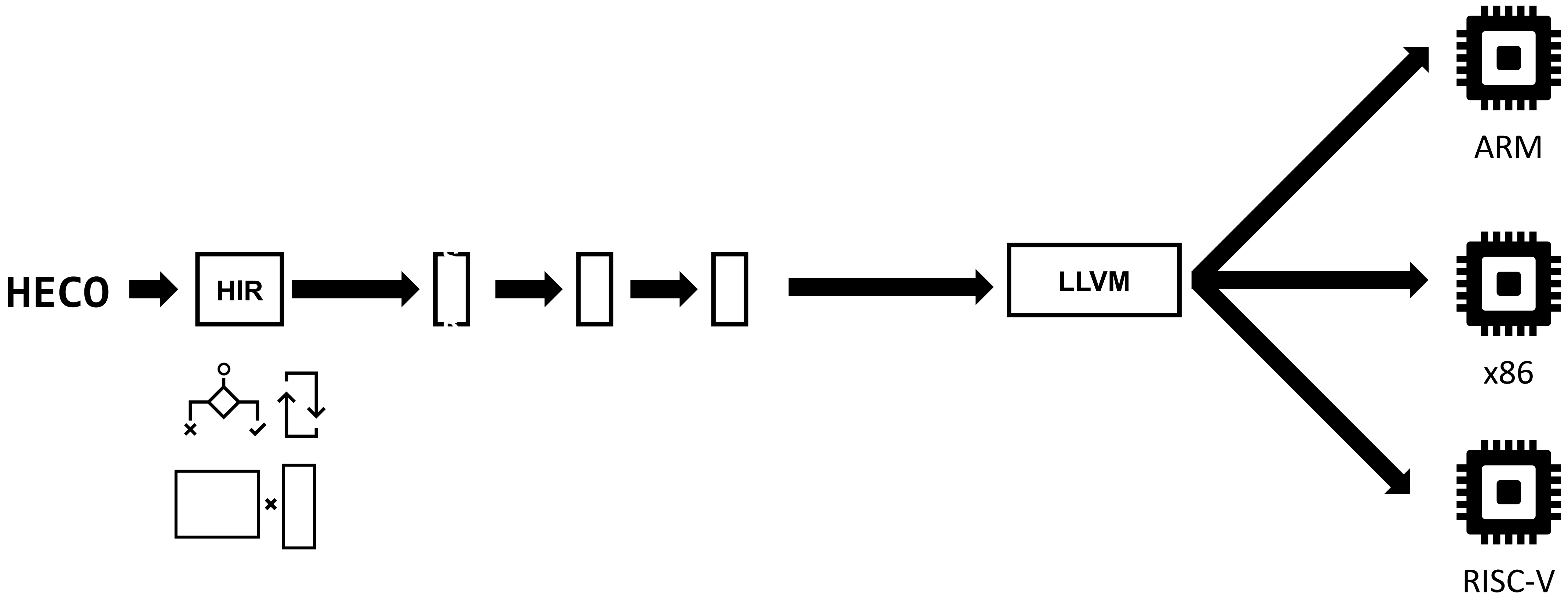
Compiler Pipelines



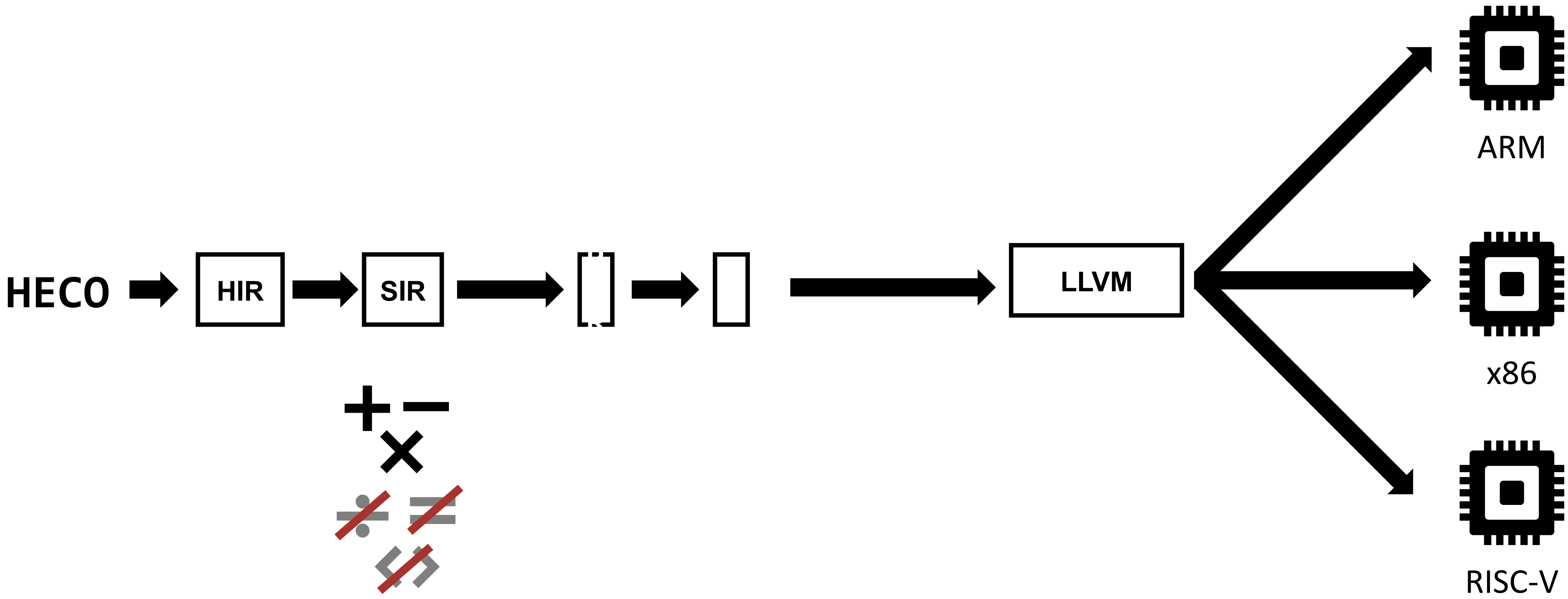
HECO Pipeline



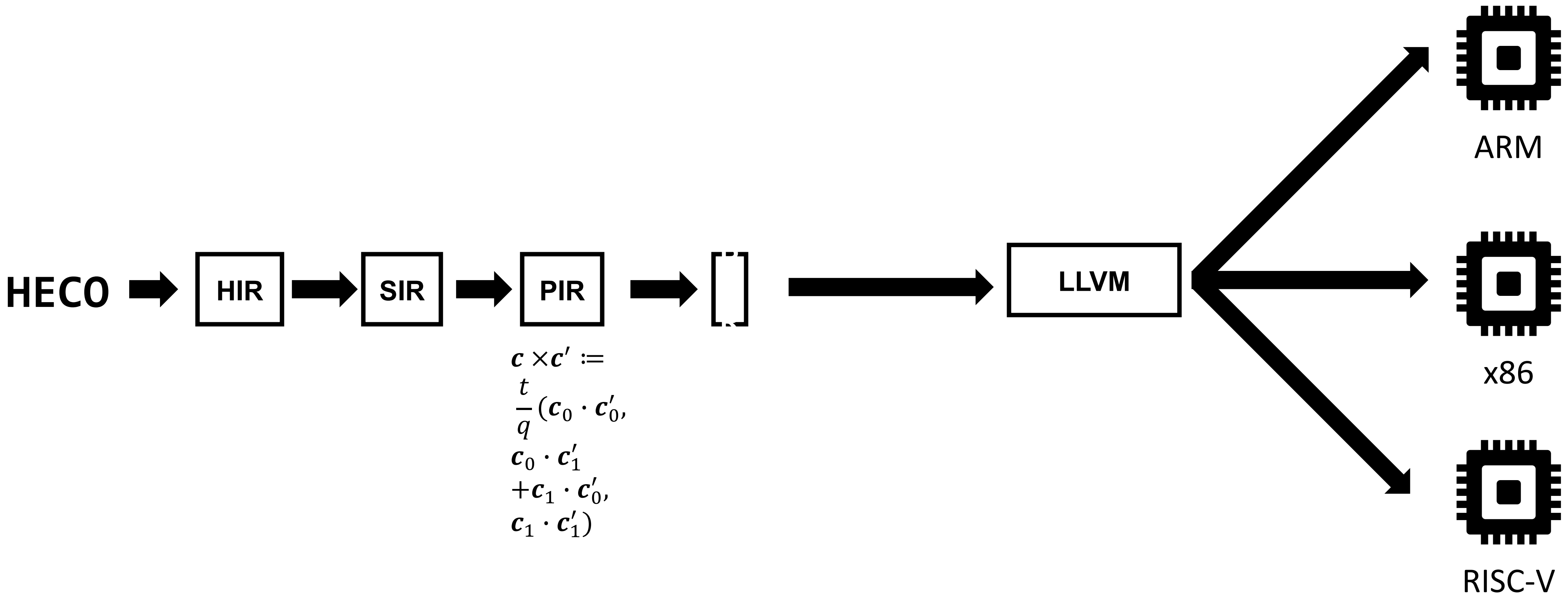
HECO Pipeline



HECO Pipeline

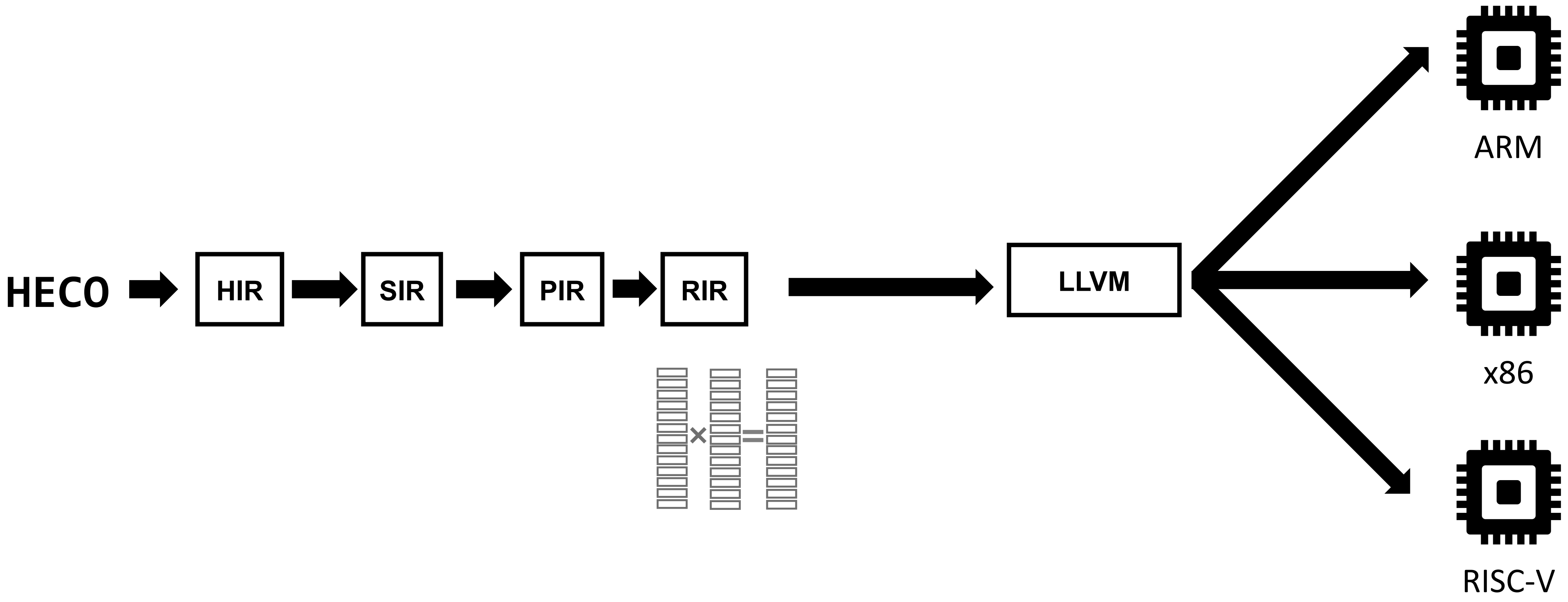


HECO Pipeline

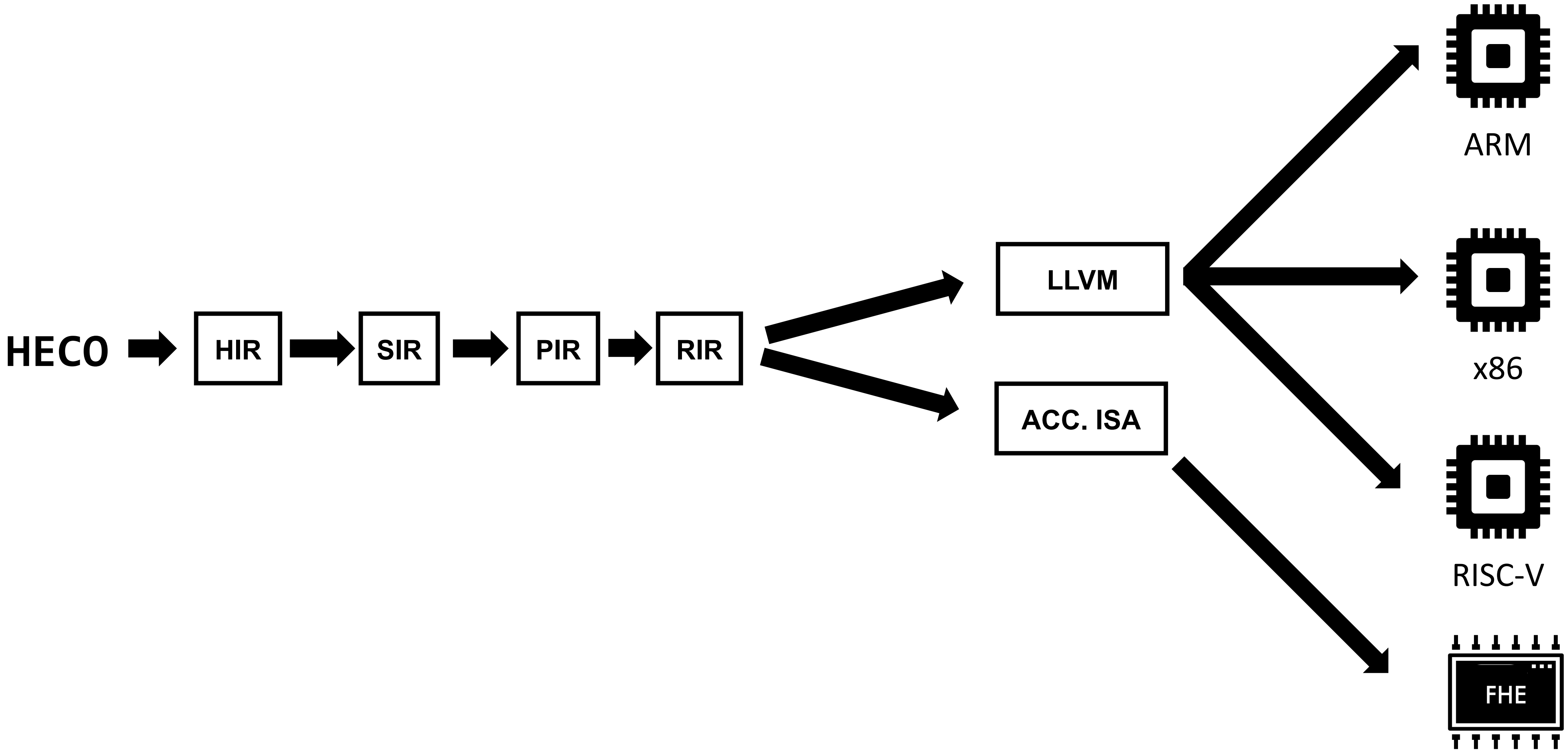


$\boldsymbol{c} \times \boldsymbol{c}' \coloneqq \frac{t}{q}(\boldsymbol{c}_0 \cdot \boldsymbol{c}_0', \boldsymbol{c}_0 \cdot \boldsymbol{c}_1' + \boldsymbol{c}_1 \cdot \boldsymbol{c}_0', \boldsymbol{c}_1 \cdot \boldsymbol{c}_1')$

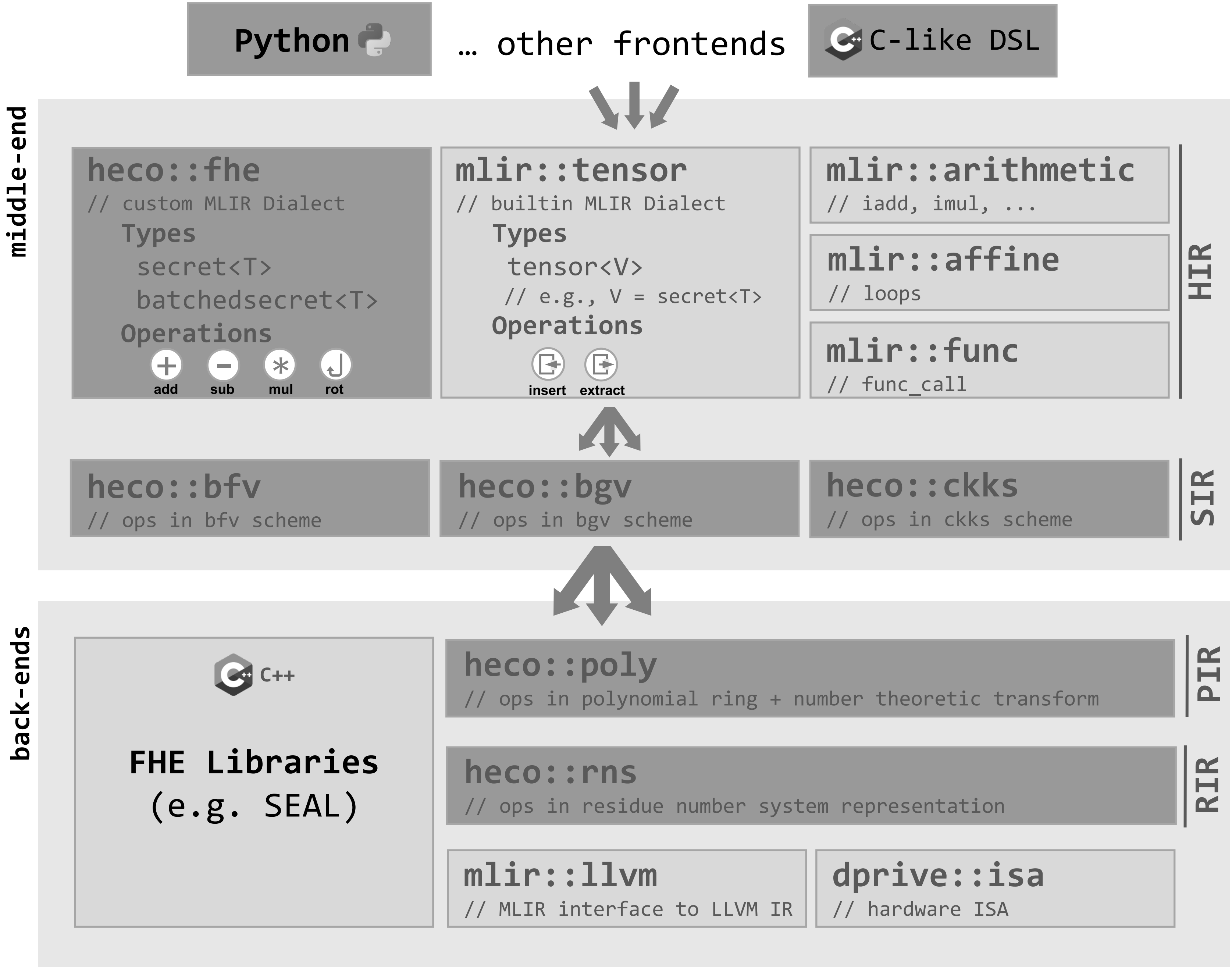
HECO Pipeline



HECO Pipeline



HECO Architecture



FHE Intermediate Representations

What is an Intermediate Representation?

- Operations & Types

```
%res = std.add(%lhs, %rhs) : (!std.i32, !std.i32) -> !std.i32
```

Results

Operation and dialect name

Operands

Operation type

- Semantics defined mostly through lowerings

FHE Compilation Pipeline


```
%r  = bfv.mul(%lhs, %rhs) : (!fhe.ctxt, !fhe.ctxt) -> !fhe.ctxt
```

```
%r0 = poly.mul(%lhs[0], %rhs[0])
%t1 = poly.mul(%lhs[0], %rhs[1])
%t2 = poly.mul(%lhs[1], %rhs[0])
%r1 = poly.add(%t1, %t2)
%r2 = poly.mul(%lhs[1], %rhs[1])
```

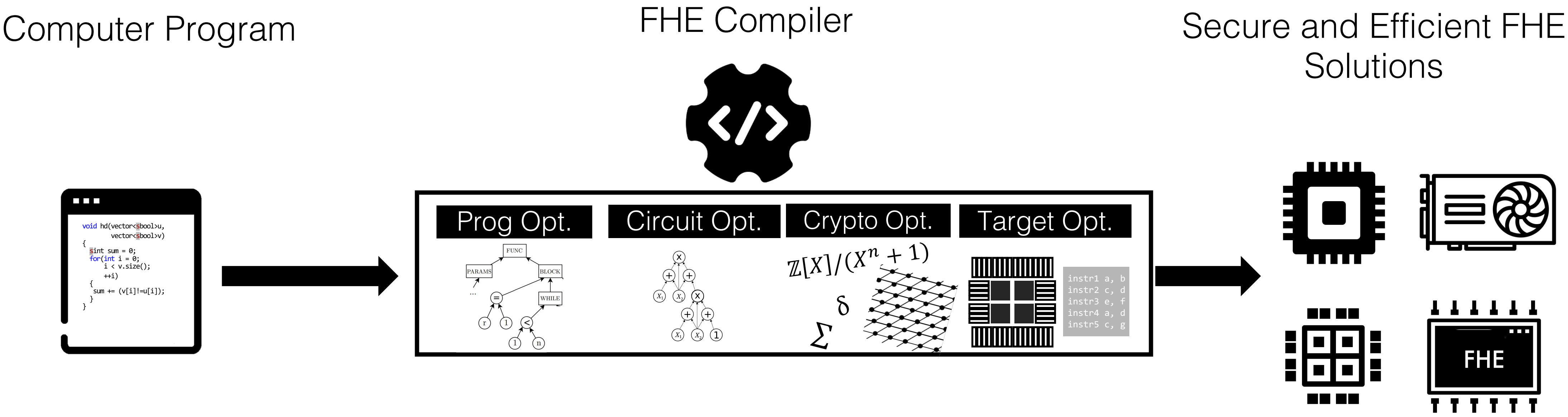
FHE Compilation Pipeline


```
%r  = bfv.mul(%lhs, %rhs) : (!fhe.ctxt, !fhe.ctxt) -> !fhe.ctxt
```

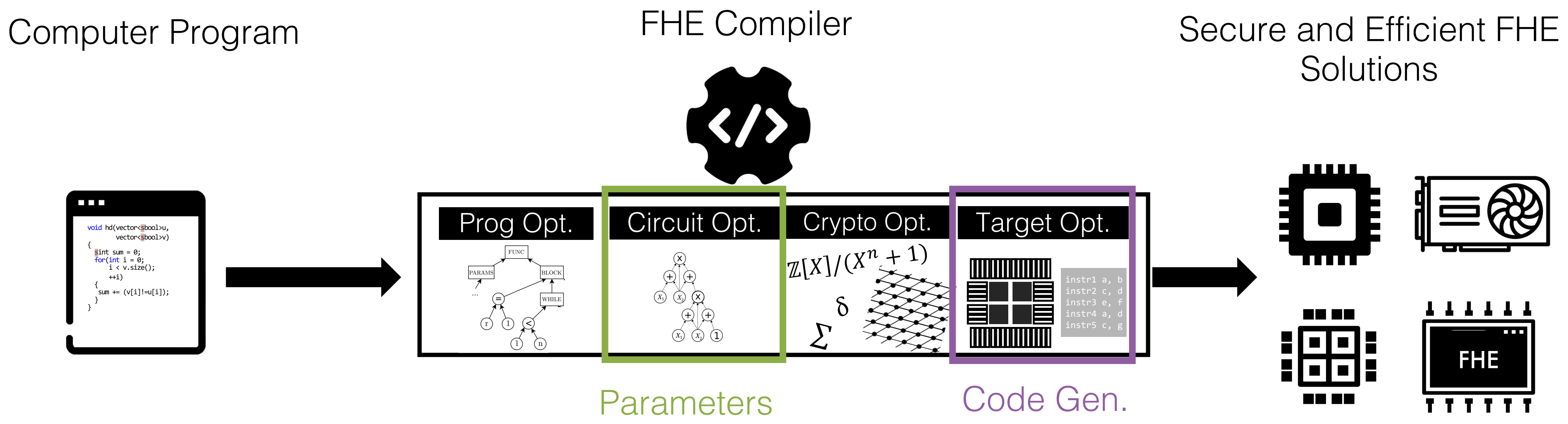
```
%r0 = poly.mul(%lhs[0], %rhs[0])
%t1 = poly.mul(%lhs[0], %rhs[1])
%t2 = poly.mul(%lhs[1], %rhs[0])
%r1 = poly.add(%t1, %t2)
%r2 = poly.mul(%lhs[1], %rhs[1])
```

```
%l1 = poly.ntt(%lhs[1])
%r1 = poly.ntt(%rhs[1])
%t  = poly.elem_mul(%l1, %r1)
%r2 = poly.inv_ntt(%t)
```

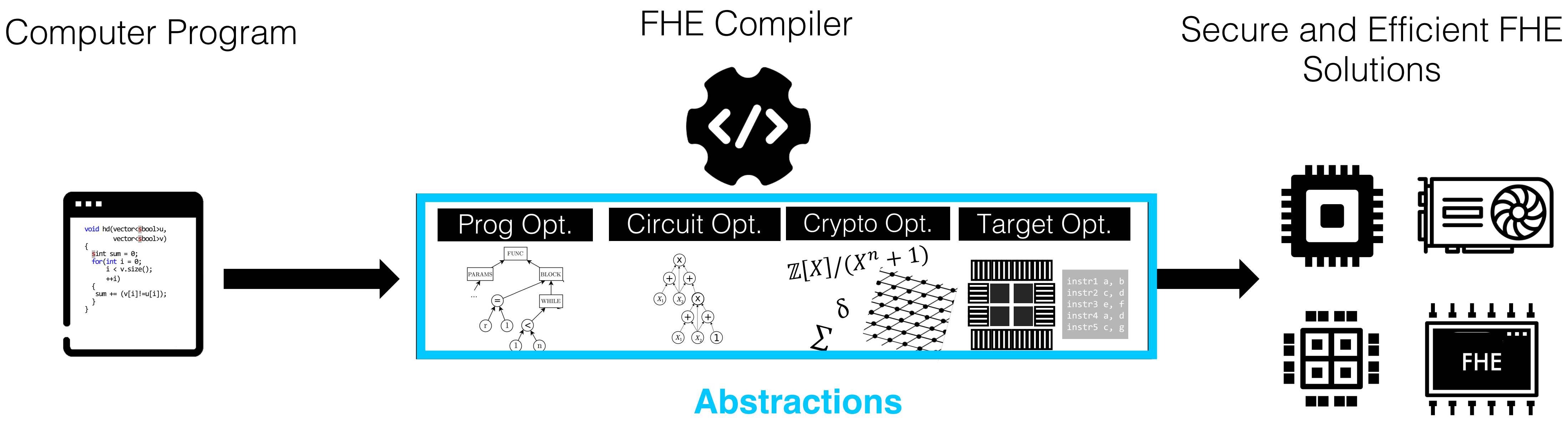
HECO: Modular End-to-End Design



HECO: Modular End-to-End Design



HECO: Modular End-to-End Design



FHE Scheme Standardization
- HomomorphicEncryption.org (2018)
- Now turning into ISO/IEC AWI 18033-8

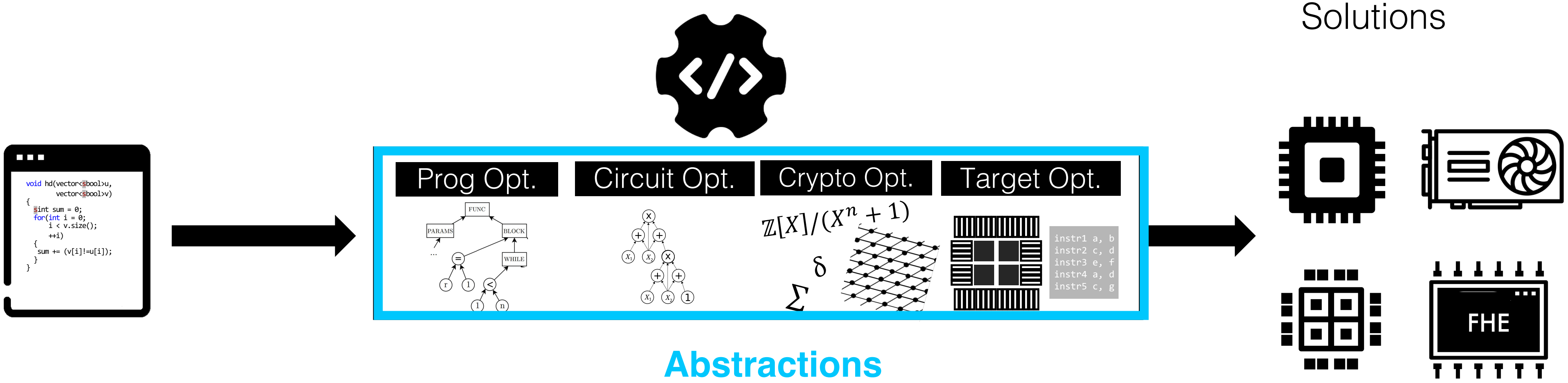
FHE Intermediate Representation Standardization
- 2018 draft API standard not adopted/implemented
- Significant FHE compiler efforts are accelerating
  - Need to re-visit standardization of abstractions
  - Expand beyond “FHE API” abstraction

HECO: Modular End-to-End Design

Computer Program

FHE Compiler

Secure and Efficient FHE Solutions



Abstractions

intel

Microsoft

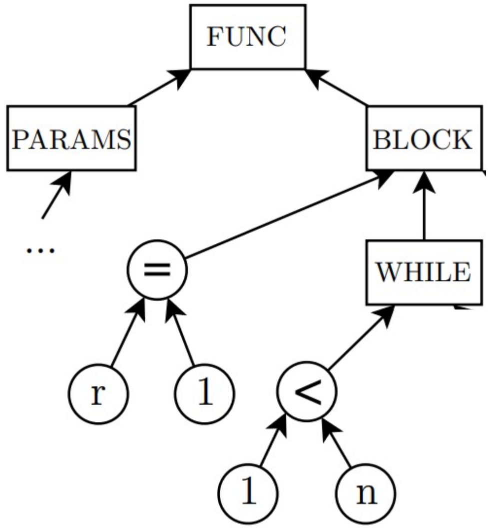
Google

ZAMA

FHE Intermediate Representation Standardization

- 2018 draft API standard not adopted/implemented
- Significant FHE compiler efforts are accelerating
  - Need to re-visit standardization of abstractions
  - Expand beyond “FHE API” abstraction

Program Optimization



SIMD-like Parallelism

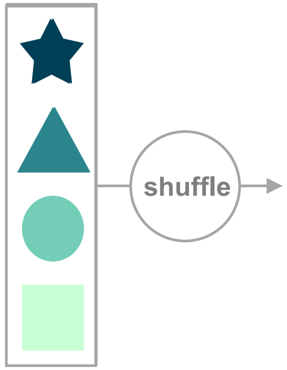
Standard C++

```
int foo(int[] x,int[] y){

   int[] r;
   for(i = 0; i < 6; ++i){
      r[i] = x[i] * y[i]
   }
   return r;

}
```

Batched FHE

```
int foo(int[] a,int[] b){

   return a * b;

}
```

SIMD Batching



No efficient free permutation or scatter/gather

SIMD-like Parallelism

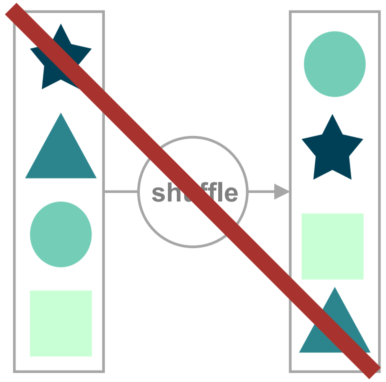
Standard C++

```
int foo(int[] x,int[] y){

   int[] r;
   for(i = 0; i < 6; ++i){
      r[i] = x[i] * y[i]
   }
   return r;

}
```

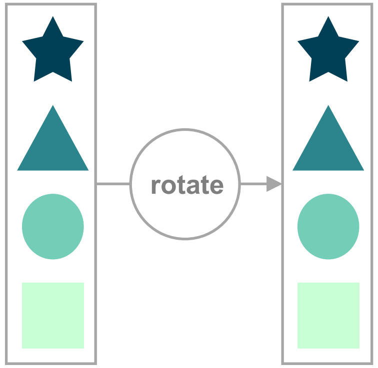
Batched FHE

```
int foo(int[] a,int[] b){

   return a * b;

}
```

SIMD Batching



No efficient free permutation or scatter/gather



Only cyclical rotations

SIMD-like Parallelism

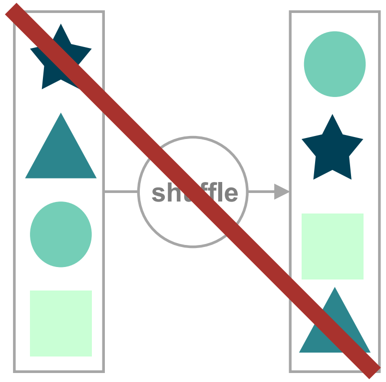
Standard C++

```
int foo(int[] x,int[] y){

   int[] r;
   for(i = 0; i < 6; ++i){
      r[i] = x[i] * y[i]
   }
   return r;

}
```

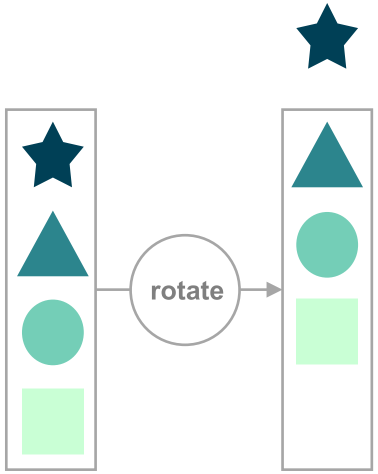
Batched FHE

```
int foo(int[] a,int[] b){

   return a * b;

}
```

SIMD Batching



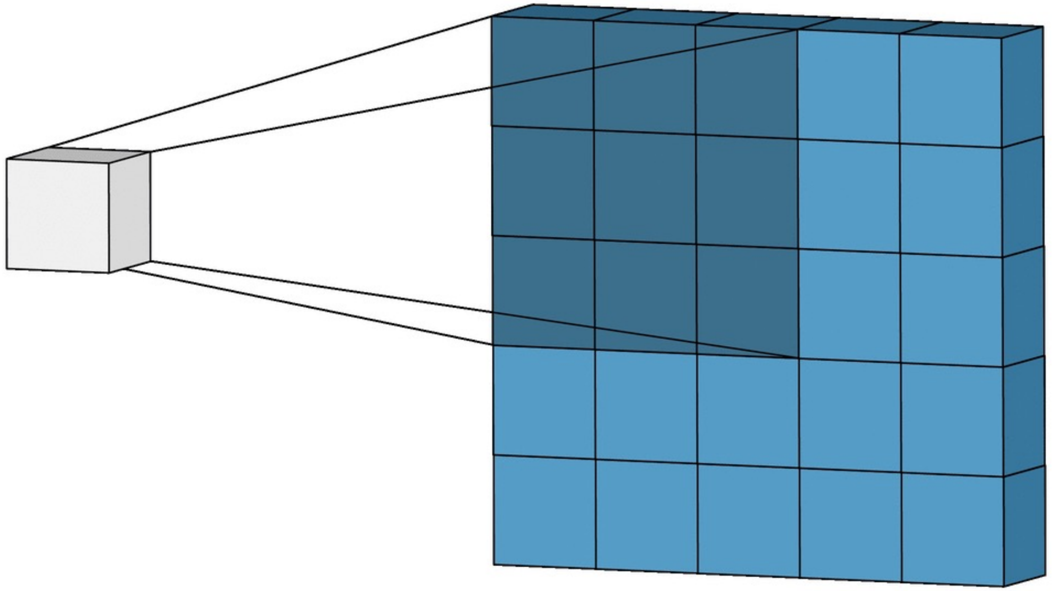
No efficient free permutation or scatter/gather



Only cyclical rotations

Example Input Program

```
LaplacianSharpening(secret_int img[], int imgSize) {
  secret_int img2[];
  int weightMatrix[3][3] = [1 1 1; 1 -8 1; 1 1 1];
  for (int x = 1; x < imgSize-1; ++x) {
    for (int y = 1; y < imgSize-1; ++y) {
      int value = 0;
      for (int j = -1; j < 2; ++j) {
        for (int i = -1; i < 2; ++i) {
          value += weightMatrix[i+1][j+1]
                       * img[imgSize*(x+i)+y+j];
        }
      }
      img2[imgSize*x+y] = 2* img[imgSize*x+y] - value;
    }
  }
  return img2;
}
```



FHE can include dramatic transformations!

```
LaplacianSharpening(secret_int img[], int imgSize) {
 secret_int img2[];
 int weightMatrix[3][3] = [1 1 1; 1 -8 1; 1 1 1];
 for (int x = 1; x < imgSize-1; ++x) {
  for (int y = 1; y < imgSize-1; ++y) {
   int value = 0;
   for (int j = -1; j < 2; ++j) {
    for (int i = -1; i < 2; ++i) {
     value += weightMatrix[i+1][j+1]
                 * img[imgSize*(x+i)+y+j];
    }
   }
   img2[imgSize*x+y] = 2* img[imgSize*x+y] - value;
  }
 }
 return img2;
}
```

```
LaplacianSharpening(secret_int img[], int imgSize) {
 secret_int[] t0 = fhe.rotate (img, -imgSize-1);
 secret_int[] t1 = fhe.rotate (img, -imgSize);
 secret_int[] t2 = fhe.rotate (img, -imgSize+1);
 secret_int[] t3 = fhe.rotate (img, -1);
 secret_int[] t4 = fhe.rotate (img, 1);
 secret_int[] t5 = fhe.rotate (img, imgSize-1);
 secret_int[] t6 = fhe.rotate (img, imgSize);
 secret_int[] t7 = fhe.rotate (img, imgSize+1);
 secret_int[] t8 = fhe.mul (img, -8);
 return fhe.add(t0,t1,t2,t3,t4,t5,t6,t7,t8);
}
```

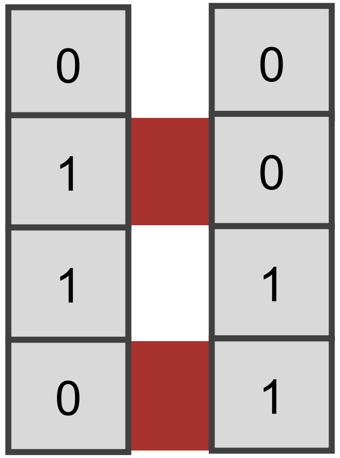
FHE can include dramatic transformations!

```
LaplacianSharpening(secret_int img[], int imgSize) {
  secret_int img2[];
  int weightMatrix[3][3] = [1 1 1; 1 -8 1; 1 1 1];
  for (int x = 1; x < imgSize-1; ++x) {
    for (int y = 1; y < imgSize-1; ++y) {
      int value = 0;
      for (int j = -1; j < 2; ++j) {
        for (int i = -1; i < 2; ++i) {
          value += weightMatrix[i+1][j+1]
                      * img[imgSize*(x+i)+y+j];
        }
      }
      img2[imgSize*x+y] = 2* img[imgSize*x+y] - value;
    }
  }
  return img2;
}
```

```
LaplacianSharpening(secret_int img[], int imgSize) {
  secret_int[] t0 = fhe.rotate (img, -imgSize-1);
  secret_int[] t1 = fhe.rotate (img, -imgSize);
  secret_int[] t2 = fhe.rotate (img, -imgSize+1);
  secret_int[] t3 = fhe.rotate (img, -1);
  secret_int[] t4 = fhe.rotate (img, 1);
  secret_int[] t5 = fhe.rotate (img, imgSize-1);
  secret_int[] t6 = fhe.rotate (img, imgSize);
  secret_int[] t7 = fhe.rotate (img, imgSize+1);
  secret_int[] t8 = fhe.mul (img, -8);
  return fhe.add(t0,t1,t2,t3,t4,t5,t6,t7,t8);
}
```

Simpler Example: Hamming Distance

```
HammingDistance(secret_int x[], secret_int x[], int len) {
  secret_int sum = 0;
  for (int i = 1; i < len-1; ++i) {
      int a = x[i] – y[i];
      int b = a * a;
      sum += b;
   }
  return sum;
}
```



Simpler Example: Hamming Distance


```
func private @HammingDistance(%x: ..., %y: ..., %len ...) -> ... {
 %c0 = fhe.constant 0 : ...
 %0 = affine.for %i = 0 to %len iter_args(%s = %c0) -> (...) {
  %1 = fhe.sub(%x[%i], %y[%i]) : ...
  %2 = fhe.multiply(%1, %1) : ...
  %3 = fhe.add(%s, %3) : ...
  affine.yield %3 : ...
 }
 func.return %0 ...
}
```

Emulating Index Access is expensive!

Simpler Example: Hamming Distance

```
func private @HammingDistance(%x: ..., %y: ..., %len ...) -> ... {
  %c0 = fhe.constant 0 : ...
  %0 = affine.for %i = 0 to %len iter_args(%s = %c0) -> (...) {
    %1 = fhe.sub(%x[%i], %y[%i]) : ...
    %2 = fhe.multiply(%1, %1) : ...
    %3 = fhe.add(%s, %3) : ...
    affine.yield %3 : ...
  }
  func.return %0 ...
}
```

Simpler Example: Hamming Distance

```
func private @HammingDistance4(%x: ..., %y: ...) -> ... {
  %c0 = fhe.constant 0 : ...
  %1 = fhe.sub(%x[0], %y[0]) : ...
  %2 = fhe.multiply(%1, %1) : ...
  %3 = fhe.add(%s, %3) : ...

  func.return %3 ...
}
```

Simpler Example: Hamming Distance


```
func private @HammingDistance4(%x: ..., %y: ...) -> ... {
  %c0 = fhe.constant 0 : ...
  %1 = fhe.sub(%x[0], %y[0]) : ...
  %2 = fhe.multiply(%1, %1) : ...
  %3 = fhe.add(%s, %3) : ...
  %4 = fhe.sub(%x[1], %y[1]) : ...
  %5 = fhe.multiply(%4, %4) : ...
  %6 = fhe.add(%3, %5) : ...
  func.return %6 ...
}
```

Simpler Example: Hamming Distance


```
func private @HammingDistance4(%x: ..., %y: ...) -> ... {
  %c0 = fhe.constant 0 : ...
  %1 = fhe.sub(%x[0], %y[0]) : ...
  %2 = fhe.multiply(%1, %1) : ...
  %3 = fhe.add(%s, %3) : ...
  %4 = fhe.sub(%x[1], %y[1]) : ...
  %5 = fhe.multiply(%4, %4) : ...
  %6 = fhe.add(%3, %5) : ...
  %7 = fhe.sub(%x[2], %y[2]) : ...
  %8 = fhe.multiply(%7, %7) : ...
  %9 = fhe.add(%6, %8) : ...
  %10 = fhe.sub(%x[3], %y[3]) : ...
  %11 = fhe.multiply(%10, %10) : ...
  %12 = fhe.add(%9, %11) : ...
  func.return %12 ...
}
```

Current cost: 4 Multiplications & 8 Index Accesses

Simpler Example: Hamming Distance

```
func private @HammingDistance4(%x: ..., %y: ...) -> ... {
  %c0 = fhe.constant 0 : ...
  %1 = fhe.sub(%x[0], %y[0]) : ...
  %2 = fhe.multiply(%1, %1) : ...
  %3 = fhe.add(%s, %3) : ...
  %4 = fhe.sub(%x[1], %y[1]) : ...
  %5 = fhe.multiply(%4, %4) : ...
  %6 = fhe.add(%3, %5) : ...
  %7 = fhe.sub(%x[2], %y[2]) : ...
  %8 = fhe.multiply(%7, %7) : ...
  %9 = fhe.add(%6, %8) : ...
  %10 = fhe.sub(%x[3], %y[3]) : ...
  %11 = fhe.multiply(%10, %10) : ...
  %12 = fhe.add(%9, %11) : ...
  func.return %12 ...
}
```

Combining Sequential Operations

Simpler Example: Hamming Distance

```
func private @HammingDistance4(%x: ..., %y: ...) -> ... {
  %c0 = fhe.constant 0 : ...
  %1 = fhe.sub(%x[0], %y[0]) : ...
  %2 = fhe.multiply(%1, %1) : ...

  %4 = fhe.sub(%x[1], %y[1]) : ...
  %5 = fhe.multiply(%4, %4) : ...

  %7 = fhe.sub(%x[2], %y[2]) : ...
  %8 = fhe.multiply(%7, %7) : ...

  %10 = fhe.sub(%x[3], %y[3]) : ...
  %11 = fhe.multiply(%10, %10) : ...
  %12 = fhe.add(%c0, %2, %5, %8, %11) : ...
  func.return %12 ...
}
```

Combining Sequential Operations

Simpler Example: Hamming Distance

```
func private @HammingDistance4(%x: ..., %y: ...) -> ... {

  %1 = fhe.sub(%x[0], %y[0]) : ...
  %2 = fhe.multiply(%1, %1) : ...

  %4 = fhe.sub(%x[1], %y[1]) : ...
  %5 = fhe.multiply(%4, %4) : ...

  %7 = fhe.sub(%x[2], %y[2]) : ...
  %8 = fhe.multiply(%7, %7) : ...

  %10 = fhe.sub(%x[3], %y[3]) : ...
  %11 = fhe.multiply(%10, %10) : ...
  %12 = fhe.add(%2, %5, %8, %11) : ...
  func.return %12 ...
}
```

Combining Sequential Operations

Simpler Example: Hamming Distance

```
func private @HammingDistance4(%x: ..., %y: ...) -> ... {
  %1 = fhe.sub(%x[0], %y[0]) : ...
  %2 = fhe.multiply(%1, %1) : ...
  %4 = fhe.sub(%x[1], %y[1]) : ...
  %5 = fhe.multiply(%4, %4) : ...
  %7 = fhe.sub(%x[2], %y[2]) : ...
  %8 = fhe.multiply(%7, %7) : ...
  %10 = fhe.sub(%x[3], %y[3]) : ...
  %11 = fhe.multiply(%10, %10) : ...
  %12 = fhe.add(%2, %5, %8, %11) : ...
  func.return %12 ...
}
```

Apply Operation to entire Vector

Simpler Example: Hamming Distance

```
func private @HammingDistance4(%x: ..., %y: ...) -> ... {
  %1 = fhe.sub(%x, %y) : ...
  %2 = fhe.multiply(%1, %1) : ...
  %4 = fhe.sub(%x[1], %y[1]) : ...
  %5 = fhe.multiply(%4, %4) : ...
  %7 = fhe.sub(%x[2], %y[2]) : ...
  %8 = fhe.multiply(%7, %7) : ...
  %10 = fhe.sub(%x[3], %y[3]) : ...
  %11 = fhe.multiply(%10, %10) : ...
  %12 = fhe.add(%2, %5, %8, %11) : ...
  func.return %12 ...
}
```

Apply Operation to entire Vector

Simpler Example: Hamming Distance

```
func private @HammingDistance4(%x: ..., %y: ...) -> ... {
  %1 = fhe.sub(%x, %y) : ...
  %2 = fhe.multiply(%1[0], %1[0]) : ...

  %5 = fhe.multiply(%4[1], %4[1]) : ...

  %8 = fhe.multiply(%7[2], %7[2]) : ...

  %11 = fhe.multiply(%10[3], %10[3]) : ...
  %12 = fhe.add(%2, %5, %8, %11) : ...
  func.return %12 ...
}
```

Apply Operation to entire Vector

Simpler Example: Hamming Distance

```
func private @HammingDistance4(%x: ..., %y: ...) -> ... {
  %1 = fhe.sub(%x, %y) : ...
  %2 = fhe.multiply(%1[0], %1[0]) : ...
  %5 = fhe.multiply(%4[1], %4[1]) : ...
  %8 = fhe.multiply(%7[2], %7[2]) : ...
  %11 = fhe.multiply(%10[3], %10[3]) : ...
  %12 = fhe.add(%2, %5, %8, %11) : ...
  func.return %12 ...
}
```

Apply Operation to entire Vector

Each use now requires index access ☹

Simpler Example: Hamming Distance

```
func private @HammingDistance4(%x: ..., %y: ...) -> ... {
  %1 = fhe.sub(%x, %y) : ...
  %2 = fhe.multiply(%1, %1) : ...
  %3 = fhe.add(%2[0], %2[1], %2[2], %2[3]) : ...
  func.return %3 ...
}
```

Algorithm 2 Batching Pass

```
Algorithm BATCHPASS(G)
    V, E ← G
    foreach op ∈ V ∧ type(op) = fhe.secret:
        ts ← SELECTTARGETSLOT(op, V, E)
        OPERANDCONVERSION(op, ts, V, E)
        foreach v ∈ V ∧ (op, v) ∈ E:
            u ← fhe.extract[v, ts]
            REPLACE(v, u, V, E)
procedure SELECTTARGETSLOT(op, V, E)
    foreach v ∈ V ∧ (op, v) ∈ E:
        switch v:
            case fhe.insert[_, i]: return i
            case func.return: return 0
    foreach v ∈ V ∧ (v, op) ∈ E:
        switch o:
            case fhe.extract[_, i]:
                return i
    return ⊥
```

Continuing this gives us the first improvement

Target Slot Logic extends this to complex cases

50% faster is nice, but nowhere near sufficient ☹

Simpler Example: Hamming Distance

```
func private @HammingDistance4(%x: ..., %y: ...) -> ... {
  %1 = fhe.sub(%x, %y) : ...
  %2 = fhe.multiply(%1, %1) : ...
  %3 = fhe.add(%2[0], %2[1], %2[2], %2[3]) : ...
  func.return %3 ...
}
```

Translate Index Accesses to Rotations

Simpler Example: Hamming Distance

```
func private @HammingDistance4(%x: ..., %y: ...) -> ... {
 %1 = fhe.sub(%x, %y) : ...
 %2 = fhe.multiply(%1, %1) : ...
 %3 = fhe.rotate(%2, -1)
 %4 = fhe.rotate(%2, -2)
 %5 = fhe.rotate(%2, -3)
 %6 = fhe.add(%2, %3, %4, %5) : ...
 func.return %6[0] ...
}
```

Translate Index Accesses to Rotations

Simpler Example: Hamming Distance


```
func private @HammingDistance4(%x: ..., %y: ...) -> ... {
  %1 = fhe.sub(%x, %y) : ...
  %2 = fhe.multiply(%1, %1) : ...
  %3 = fhe.rotate(%2, -1)
  %4 = fhe.rotate(%2, -2)
  %5 = fhe.rotate(%2, -3)
  %6 = fhe.add(%2, %3, %4, %5) : ...
  func.return %6[0] ...
}
```

Translate Index Accesses to Rotations

Implicit Scalar Encoding: s = s[0]

Simpler Example: Hamming Distance

```
func private @HammingDistance4(%x: ..., %y: ...) -> ... {
  %1 = fhe.sub(%x, %y) : ...
  %2 = fhe.multiply(%1, %1) : ...
  %3 = fhe.rotate(%2, -1)
  %4 = fhe.rotate(%2, -2)
  %5 = fhe.rotate(%2, -3)
  %6 = fhe.add(%2, %3, %4, %5) : ...
  func.return %6 ...
}
```

Translate Index Accesses to Rotations

Implicit Scalar Encoding: s = s[0]

Simpler Example: Hamming Distance


```
func private @HammingDistance4(%x: ..., %y: ...) -> ... {
  %1 = fhe.sub(%x, %y) : ...
  %2 = fhe.multiply(%1, %1) : ...
  %3 = fhe.rotate(%2, -1)
  %4 = fhe.rotate(%2, -2)
  %5 = fhe.rotate(%2, -3)
  %6 = fhe.add(%2, %3, %4, %5) : ...
  func.return %6 ...
}
```

Translate Index Accesses to Rotations

Implicit Scalar Encoding: s = s[0]

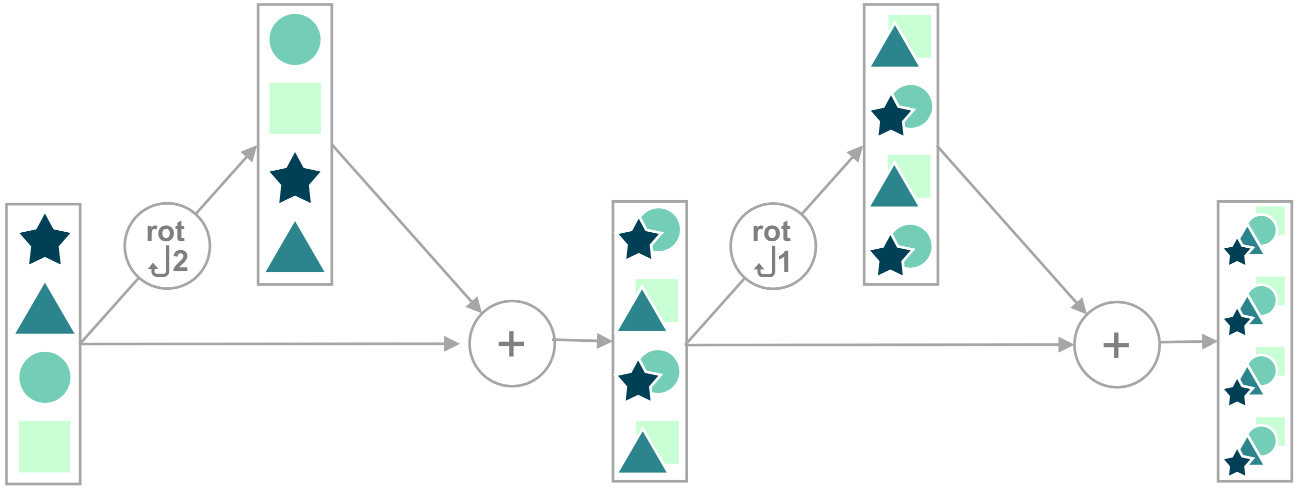
Significantly faster, but still O(n) ☹

Exploit the fact that all addends have same origin!

Simpler Example: Hamming Distance

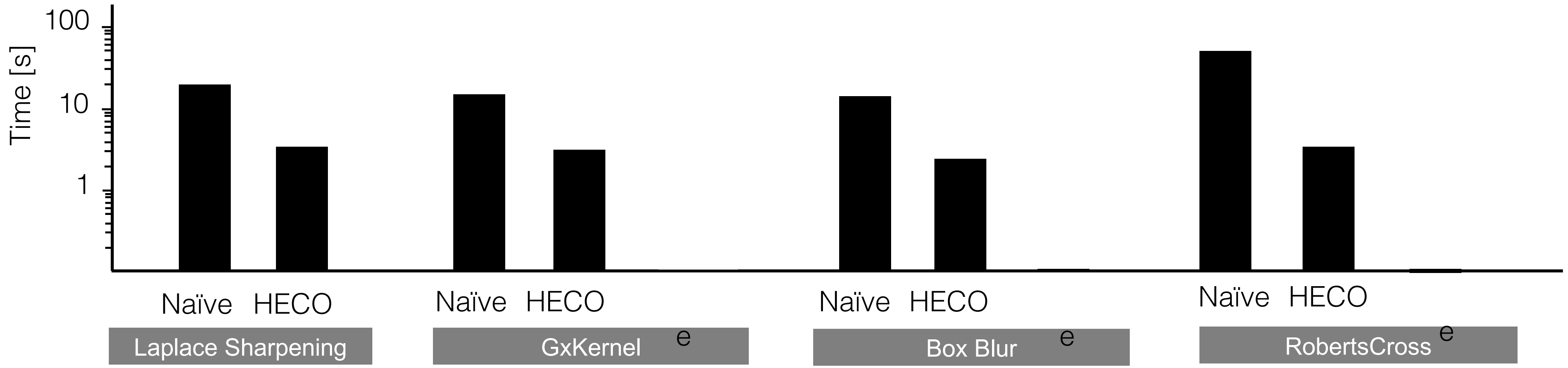
```
func private @HammingDistance4(%x: ..., %y: ...) -> ... {
  %1 = fhe.sub(%x, %y) : ...
  %2 = fhe.multiply(%1, %1) : ...
  %3 = fhe.rotate(%2, -2) : ...
  %4 = fhe.add(%2, %3) : ...
  %5 = fhe.rotate(%4, -1) : ...
  %6 = fhe.add(%4, %5) : ...
  func.return %6 ...
}
```

Exploit FHE Folklore Technique: O(log(n))



Evaluation: Effect of Batching Opimizations

Comparing against Naïve (non-Batched) implementation and “Optimal” synthesis-based solution [CD+21]



[CD+21] Cowan, M. et al. 2021. Porcupine: A Synthesizing Compilerfor Vectorized Homomorphic Encryption – PLDI 2021, 375–389.

arxiv.org/abs/2202.01649

GitHub

github.com/MarbleHE/HECO