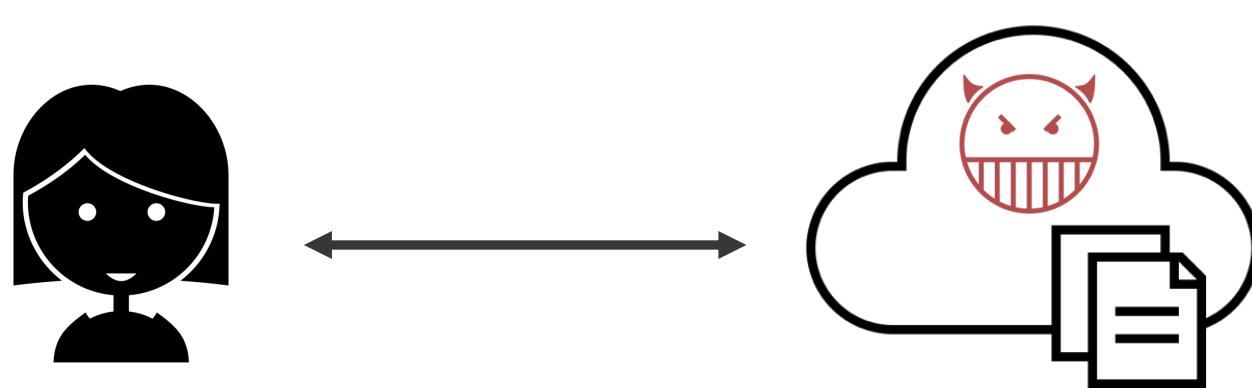


Useable Fully Homomorphic Encryption Challenges & Opportunities

Anwar Hithnawi, Alexander Viand



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

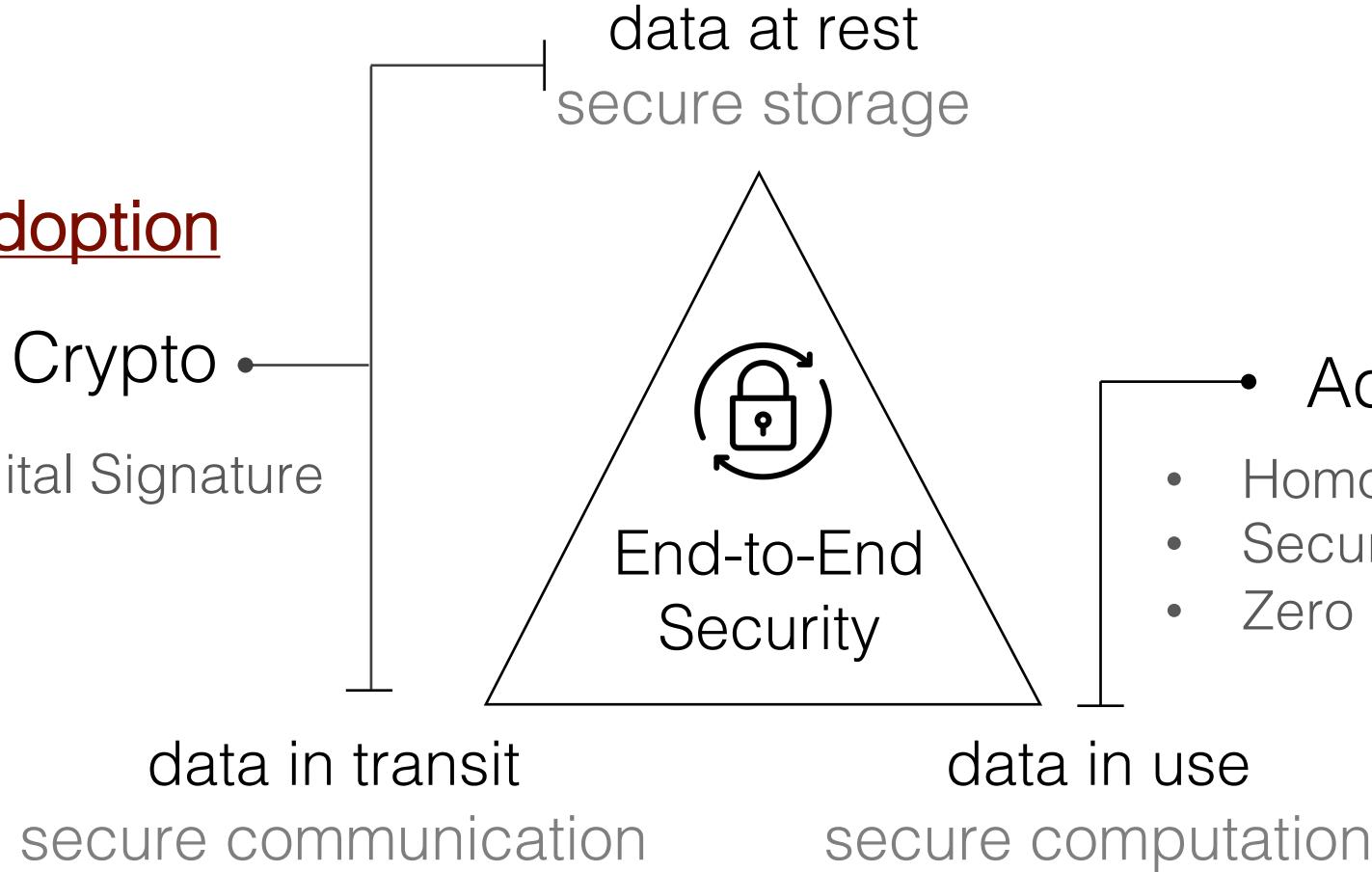


Modern Cryptography

Ubiquitous Adoption

Conventional Crypto

Encryption & Digital Signature



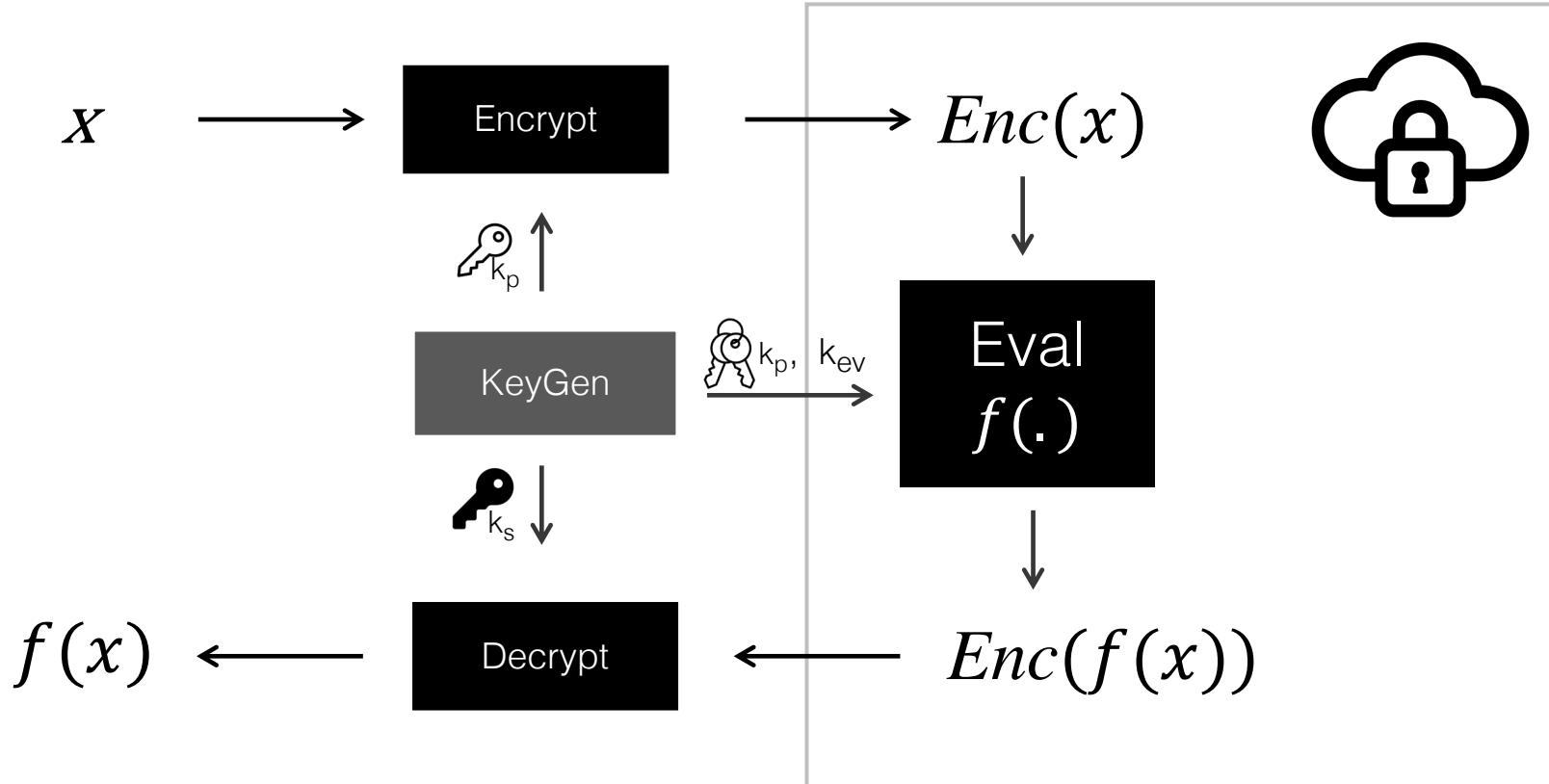
Just Starting

Advanced Crypto

- Homomorphic Encryption
- Secure Multi-party Computation
- Zero Knowledge Proofs

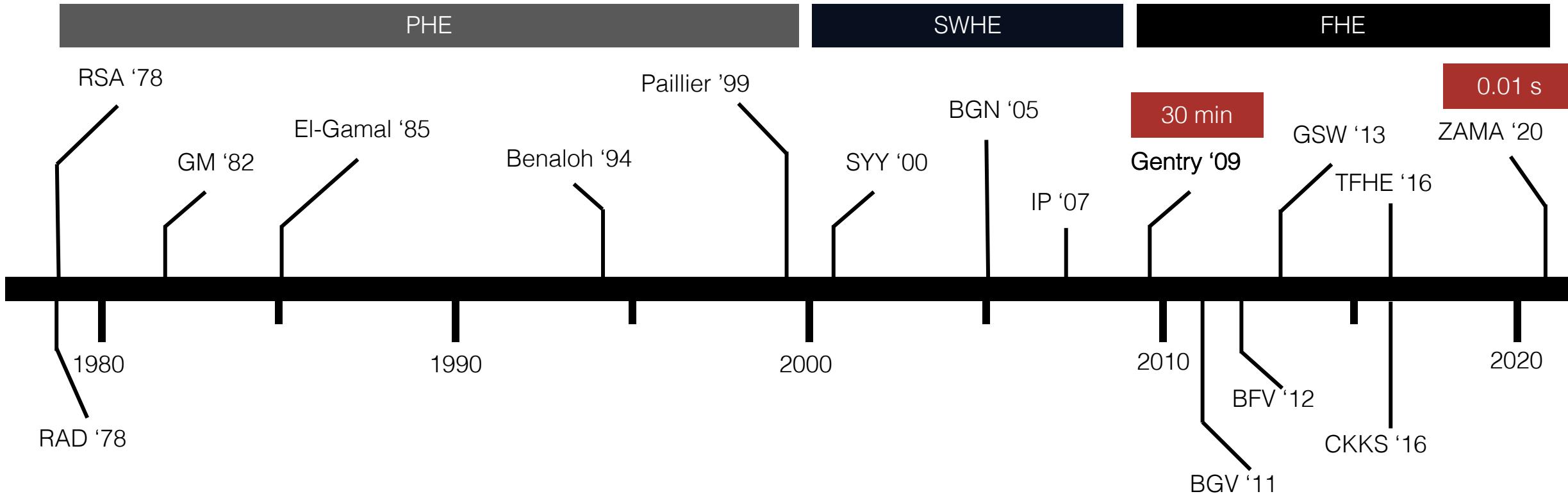
Fully Homomorphic Encryption

Enables **computation** on encrypted data



Delegate the **processing** of data without giving away **access** to it

40 Years of FHE History

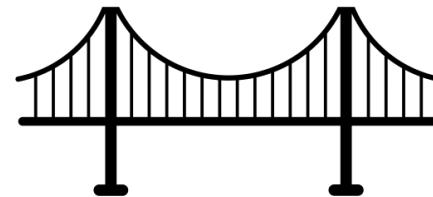


FHE will soon be practical for a wide set of applications, but ...

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



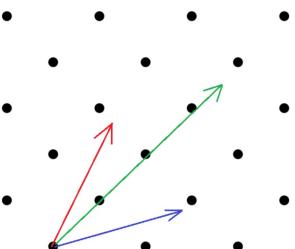
Programming
Languages

- 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

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

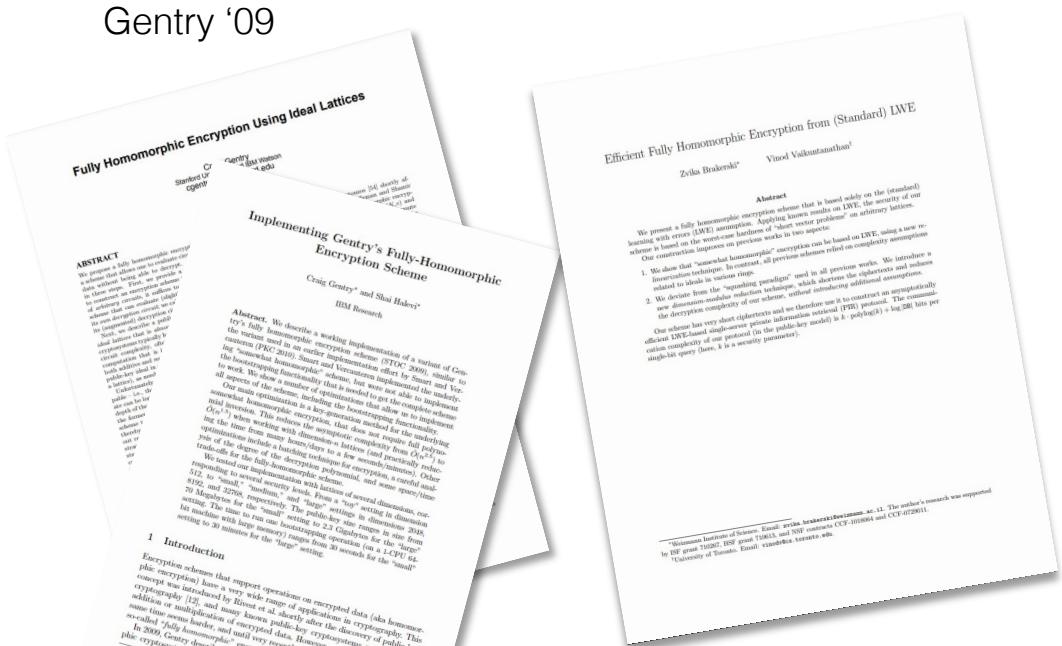
Learning With Errors



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

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

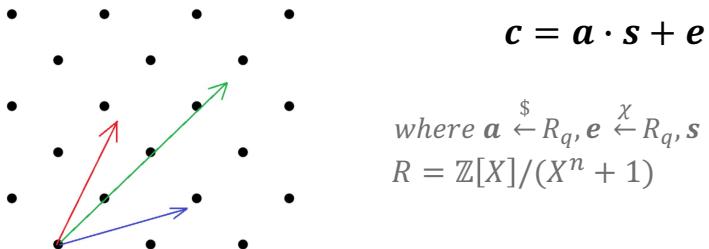
Gentry '09



FHE: Theory to Practice

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

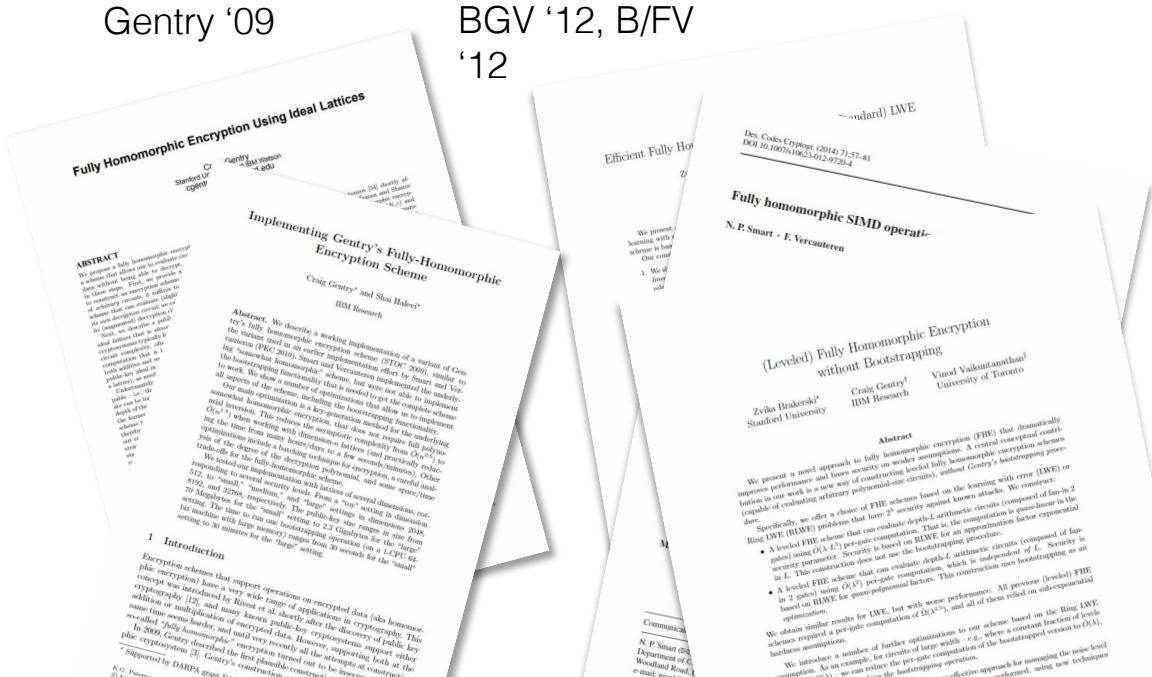
Ring-Learning With Errors



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

Gentry '09

BGV '12, B/FV
'12





Protect your passwords

Let Microsoft Edge check passwords you've saved in the browser and alert you if they've been compromised on the internet.

[On](#)

[Learn more](#)

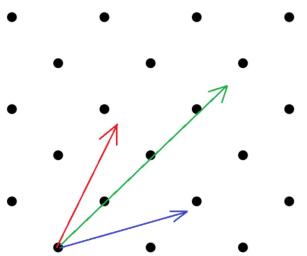
- Helps prevent identity theft
- Proactively scans the dark web
- Alerts you if your passwords are leaked online

Confirm

FHE: Theory to Practice

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

Ring-Learning **W**ith **E**rros

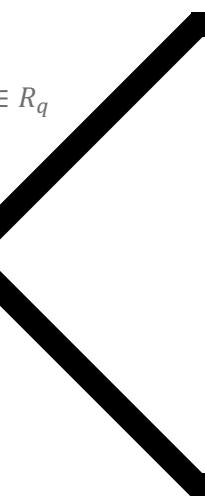


$$\mathbf{c} = \mathbf{a} \cdot \mathbf{s} + \mathbf{e}$$

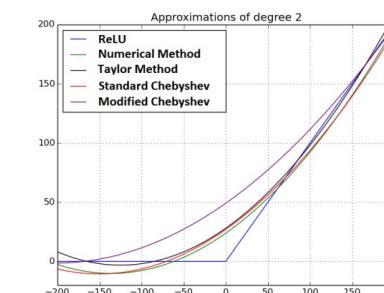
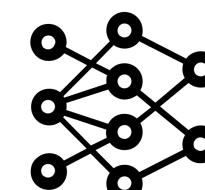
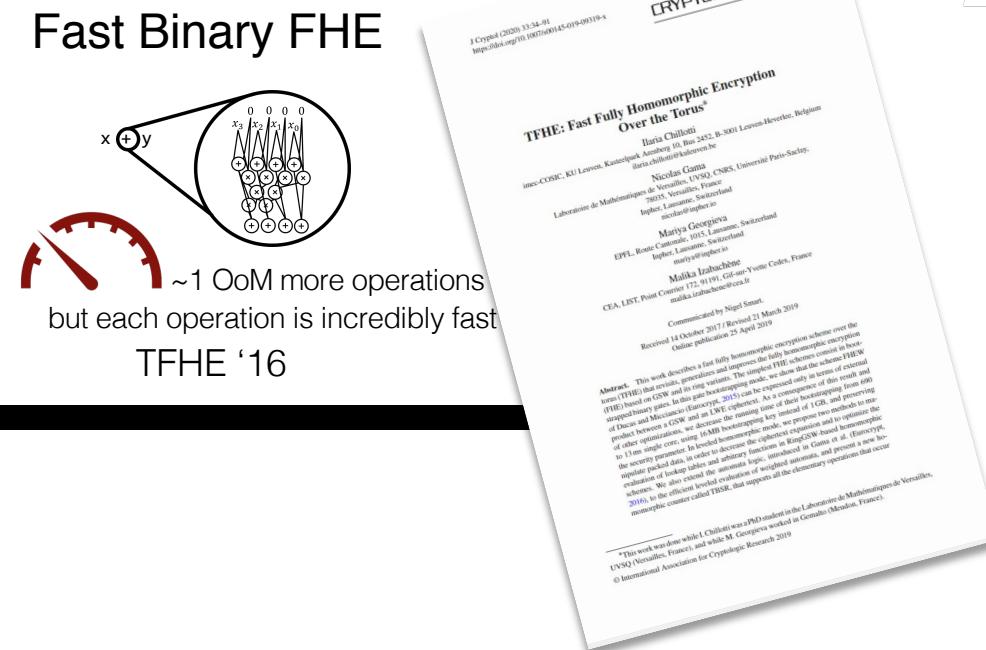
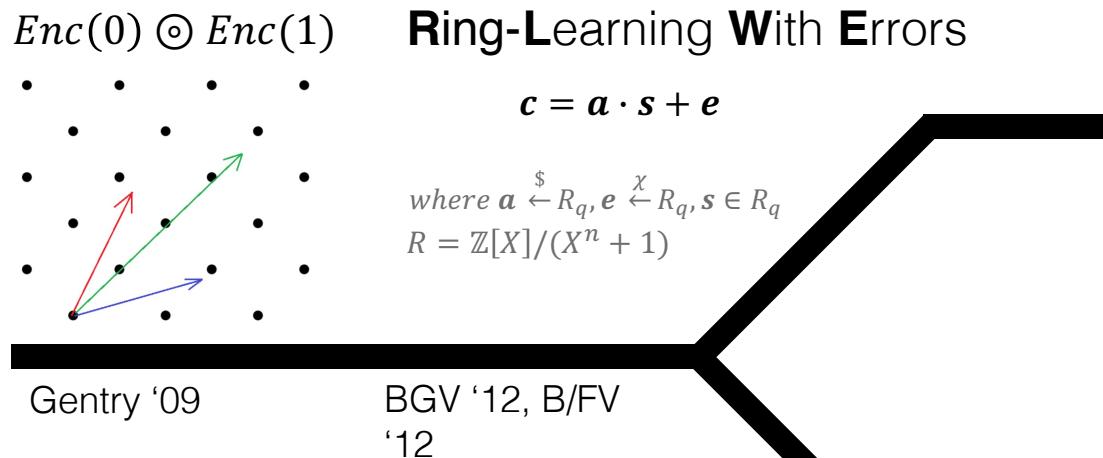
where $\mathbf{a} \xleftarrow{\$} R_q, \mathbf{e} \xleftarrow{\chi} R_q, \mathbf{s} \in R_q$
 $R = \mathbb{Z}[X]/(X^n + 1)$

Gentry '09

BGV '12, B/FV
'12



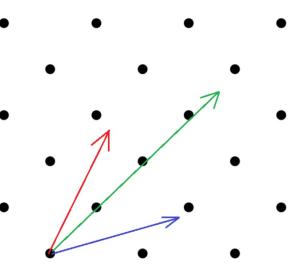
FHE: Theory to Practice



(a) Approximation of ReLU using different methods
Figure from CryptoDL: Deep Neural Networks over Encrypted Data [HTG17]

FHE: Theory to Practice

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



Gentry '09

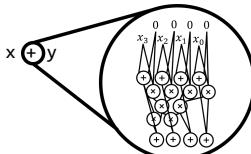
Ring-Learning With Errors

$$c = a \cdot s + e$$

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

BGV '12, B/FV
 '12

Fast Binary FHE



TFHE '16

Look-Up Tables

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

Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks
 Rui Chiffati, Marc Joye, and Pascal Paillier
 Seite 1 von 1

Abstract. In many cases, machine learning and privacy are perceived as being at odds. Privacy concerns usually prevent users from sharing their data with third parties, while machine learning requires massive amounts of data. We argue that this is not necessarily the case. We present a new technique for bootstrapping homomorphic ciphertexts that does not require any secret key evaluation. This allows us to efficiently evaluate neural networks in a homomorphic setting. In contrast to previous work, our framework does not require learning the model.

1. Introduction
 The promise of learning from data is to discover a generic function that can be applied to a wide variety of other data. Machine learning techniques have been very successful in solving tasks such as classification, regression, and clustering. However, learning from sensitive data is often a challenge. In various applications, data is collected and stored in a central location, which makes it vulnerable to attacks. To mitigate these risks, homomorphic encryption provides a way to learn from data without ever having to touch it. This is achieved by applying a function to encrypted data without ever decrypting it. This is called “learning with encrypted data” or “homomorphic inference”. The use of homomorphic encryption is currently limited to some specific applications, such as machine learning, due to its high computational cost. Machine learning algorithms are generally used in a server but the type of data they handle is often sensitive. Therefore, machine learning is often considered to be a high-risk application. In various domains, there is a need for more efficient methods to reduce risk. One approach is to use a different method to estimate sensitive data. Another approach is to use a different method to protect the data. In this paper, we propose a new method to protect data in machine learning. This method is based on a new technique for bootstrapping homomorphic ciphertexts that does not require any secret key evaluation. This allows us to efficiently evaluate neural networks in a homomorphic setting. In contrast to previous work, our framework does not require learning the model.

2. Related Work
 The first work on learning with encrypted data was done by Gentry [1]. He proposed a general framework for learning with encrypted data. This work was followed by several others, such as Brakerski et al. [2], Gentry et al. [3], and Gentry et al. [4]. These works focused on learning with encrypted data in a general setting. They also proposed various applications, such as machine learning, and provided theoretical guarantees. Some of these works were later improved by Gentry et al. [5] and Gentry et al. [6].

3. Our Contribution
 Our contribution is a new technique for bootstrapping homomorphic ciphertexts that does not require any secret key evaluation. This allows us to efficiently evaluate neural networks in a homomorphic setting. In contrast to previous work, our framework does not require learning the model.

ZAMA '21

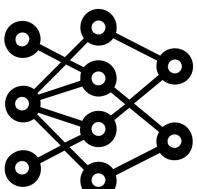


DPRIVE

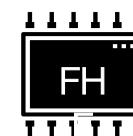
CKKS '16

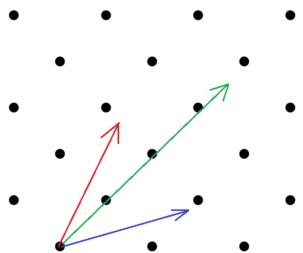
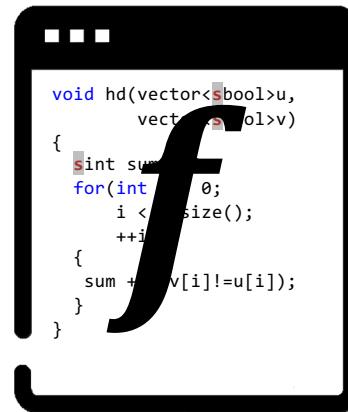
Approximate FHE

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

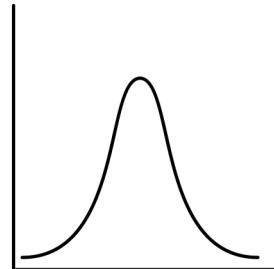


Hardware Acceleration





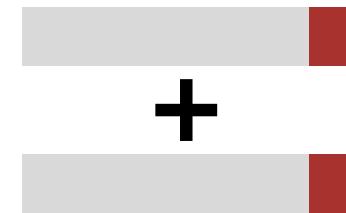
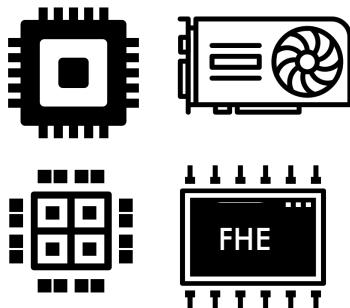
Math

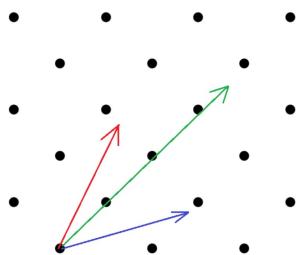
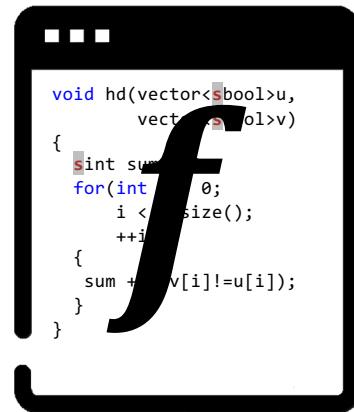


Noise

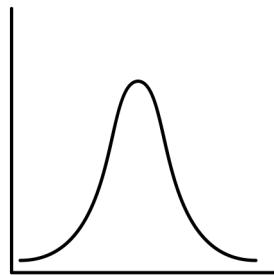
$$R = \mathbb{Z}[X]/(X^n + 1)$$

$$a * s + m + e$$





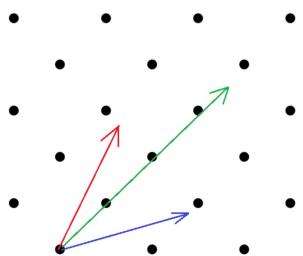
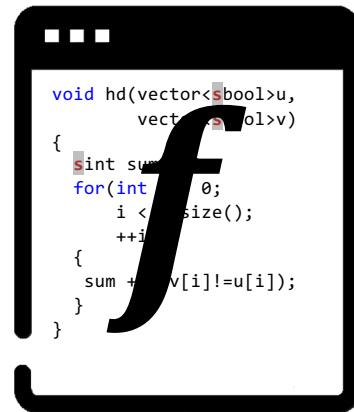
Math



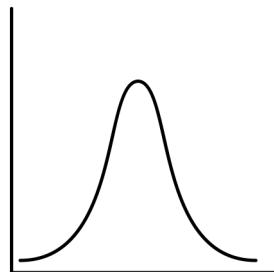
Noise

$$a * s + m + e$$





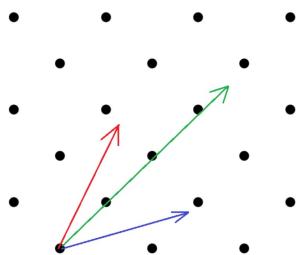
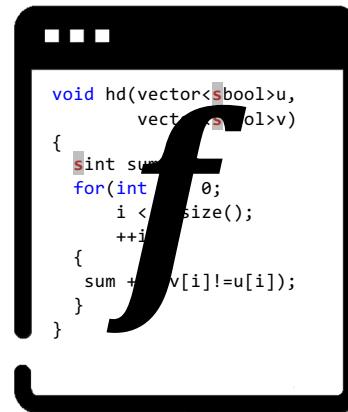
Math



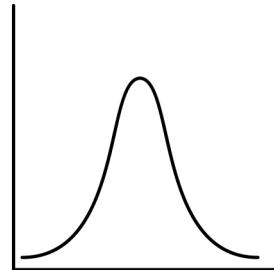
Noise

$$a * s + m + e$$



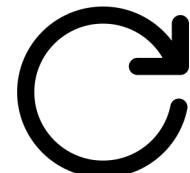


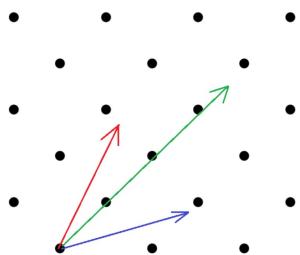
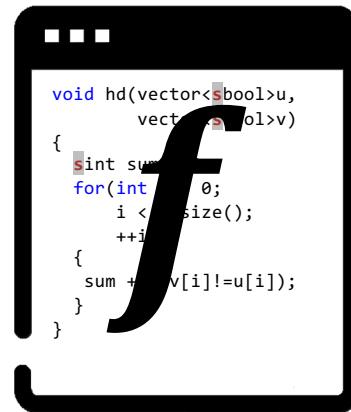
Math



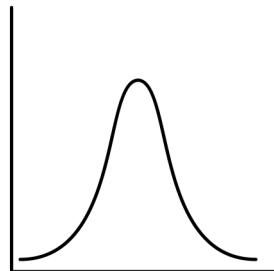
Noise

$$a * s + m + e$$





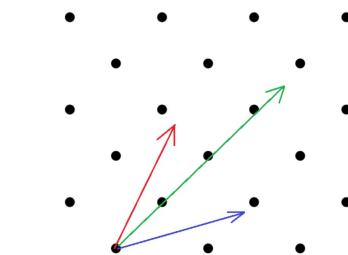
Math



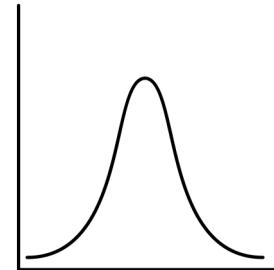
Noise

$$a * s + m + e$$

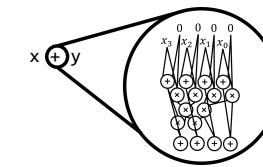
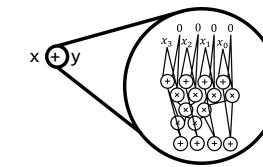
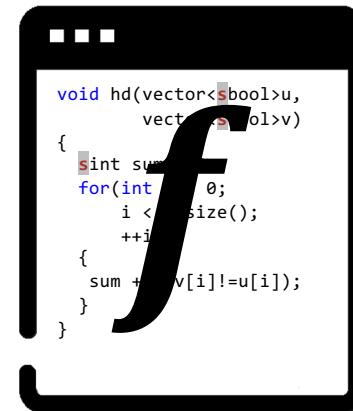




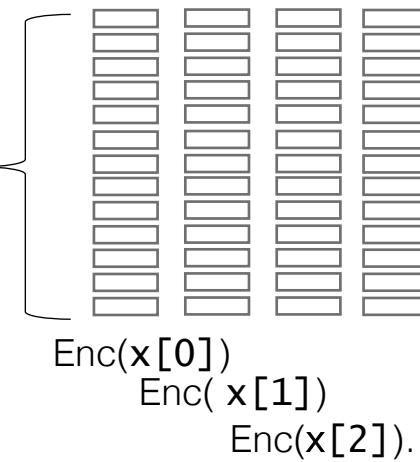
Math

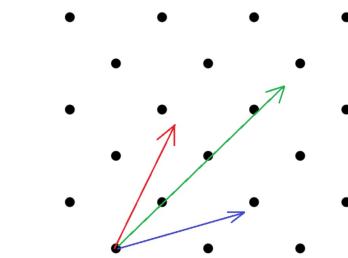


Noise

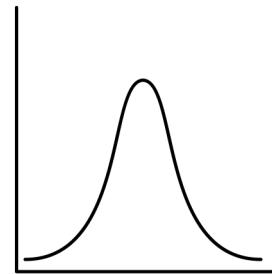


Representations

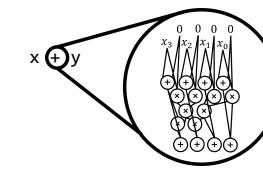
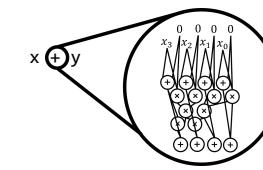
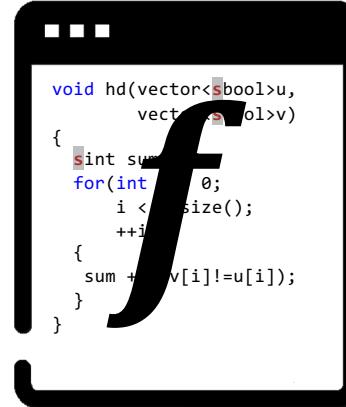
$$x[0], x[1], x[2], \dots$$
$$N = 32768$$




Math

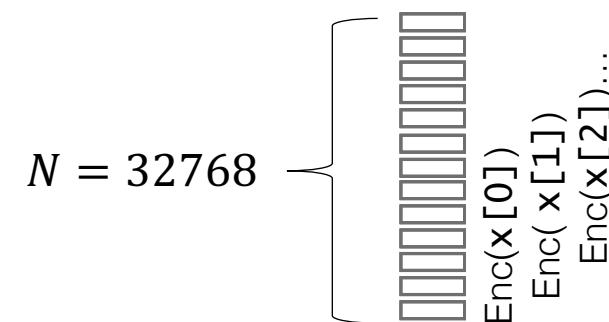


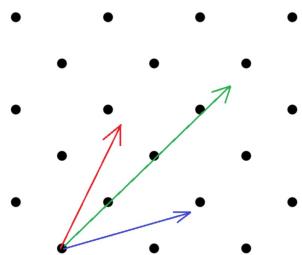
Noise



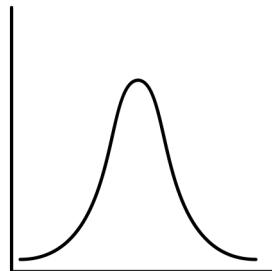
Representations

$x[0], x[1], x[2], \dots$

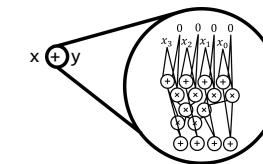
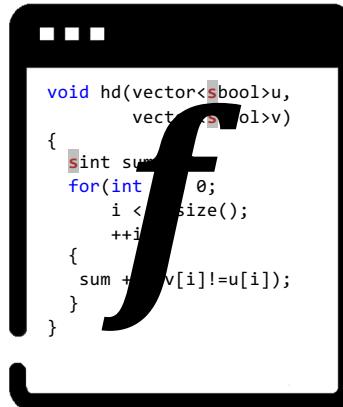




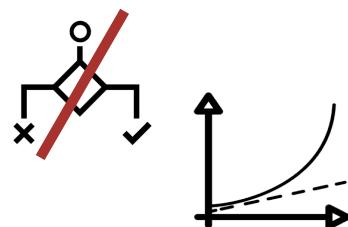
Math



Noise



Representations



Applications



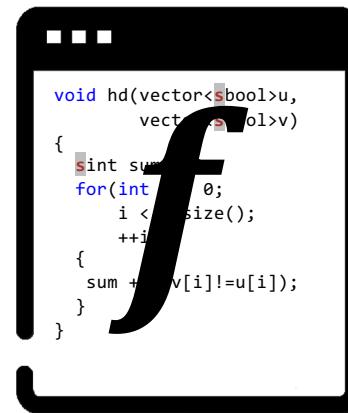
$$\chi^2 = N \sum_{i=1}^n \frac{(O_i/N - p_i)^2}{p_i}$$

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

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

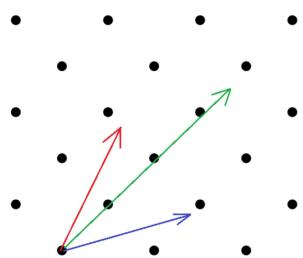
$$\beta_2 = (2N_0 + N_1)(2N_2 + N_1)$$

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

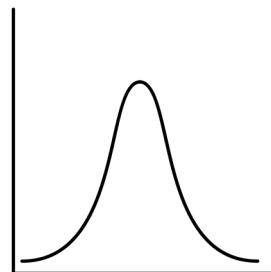


Functionality and performance depend on f 's representation:

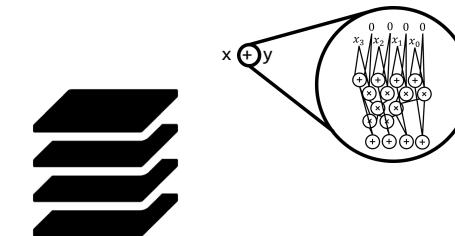
- How do we express f
- How do we optimize f



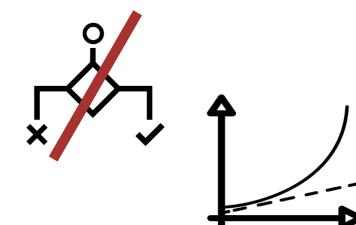
Math



Noise



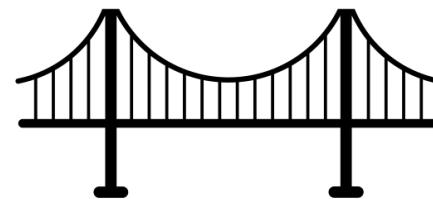
Representations



Applications

Usable FHE

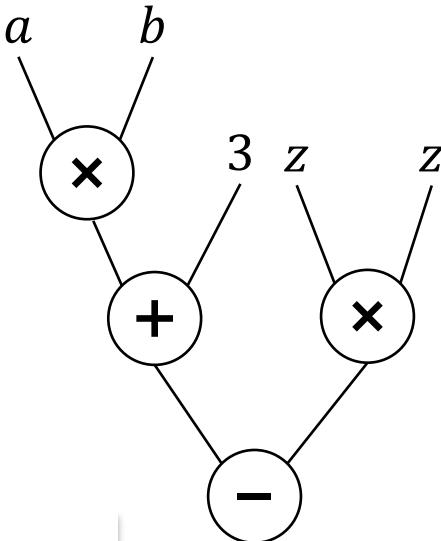
Advanced
Cryptography



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



Algorithms in HElib

Shai Halevi¹
IBM T.J. Watson Research Center
Victor Shoup^{1,2}
USA

¹ IBM T.J. Watson Research Center
² NYU

Abstract. We describe the first implementation of a fully homomorphic encryption scheme, based on the variant used in a recent provably secure FHE scheme. The scheme is somewhat homomorphic, and uses somewhat homomorphic encryption to bootstrap. We show how to work. We give some details on all aspects of the scheme. The main operations are somewhat homomorphic inversion, and multiplication by a constant. The time complexity of the main operations is $O(n^2)$ when using the time optimal algorithm, and $O(n^3)$ when using the time optimal algorithm. The space complexity is $O(n^2)$. We test the performance of the scheme on various benchmarks, and show that it is competitive with other FHE schemes. The scheme is implemented in C++ and is available at <http://www.csail.mit.edu/~shaih/HElib/>.

1 Introduction

Traditional encryption schemes, both symmetric and asymmetric, were not designed to respect any algebraic structure of the plaintext and ciphertext spaces, i.e. no computations can be performed on the ciphertext in a way that would pass through the encryption to the underlying plaintext without using the secret key, and such a property would in many contexts be considered a vulnerability. Nevertheless, this property has powerful applications, e.g. in outsourced (cloud) computation scenarios, the cloud provider could use this to guarantee customer data privacy in the presence of both internal (malicious employee) and external (outside attacker) threats. An encryption scheme that allows computations on encrypted data is said to be a *homomorphic encryption scheme*.

Some schemes, such as ElGamal (resp., e.g. Paillier), are multiplicative (resp., additive) homomorphic, i.e. one algebraic operation (resp., multiplication or addition) instead of multiple operations (resp., addition and multiplication).

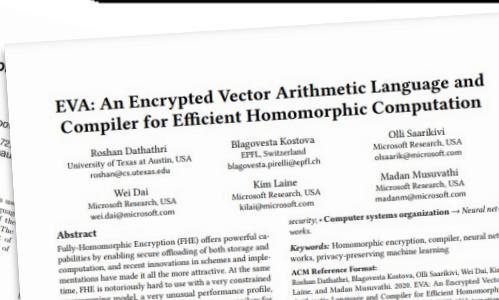
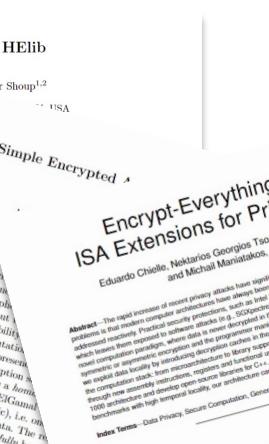
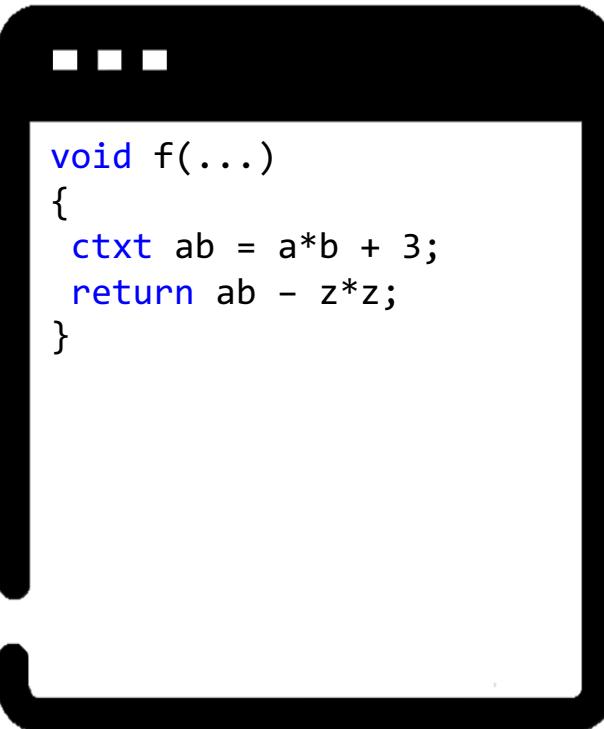
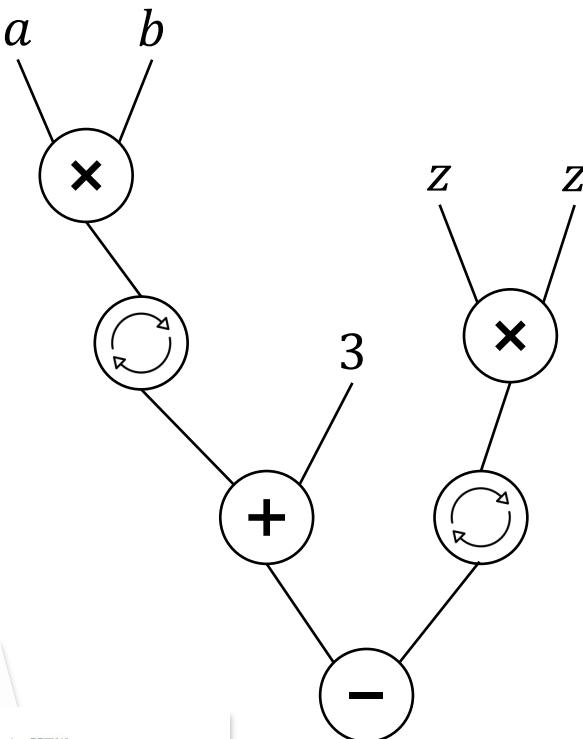
Simple Encrypted Arithmetic Library 2.3.1
Microsoft Research, WA, USA; kia.laine@microsoft.com

```

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



Evolution of FHE Tools

ETH Zürich

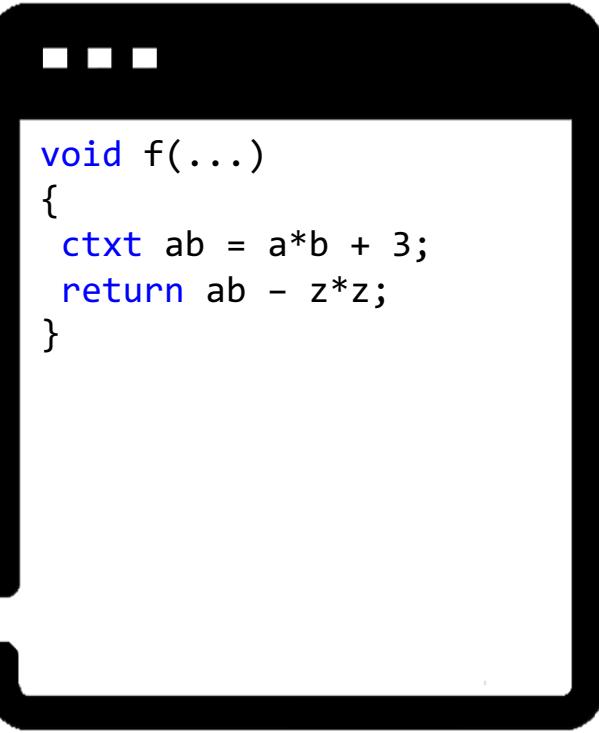
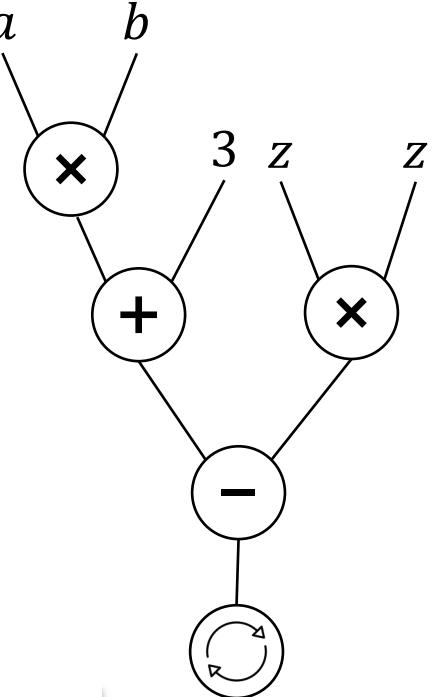


Implementing Fully Homomorphic Encryption

Shai Halevi¹
Craig Gentry²

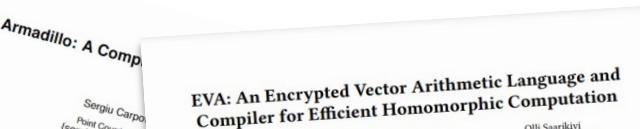
Algorithms in HElib

¹ IBM Research USA
² NYU



ISA Extensions for Private Computation
Eduardo Chaille, Nektarios Georgios Tsoutsos, Member, IEEE, and Michali Maniatis, Senior Member, IEEE

Abstract. We describe the first ISA extensions for private computation. These extensions enable efficient private computation on modern processors. The extensions support both SIMD-style parallelism and SIMD-style parallelism, encoding thousands of plaintext values into a single ciphertext to further improve throughput.



EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation
Sergiu Carpov, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Roshan Dathathri

Abstract. Fully Homomorphic Encryption (FHE) offers powerful capabilities by enabling secure offloading of both storage and computation, and recent advances in theory and implementation have made it more attractive. At the same time, FHE is notoriously hard to use with a very constrained model, a very unusual performance profile, and no standard benchmarks or open source libraries for C/C++. In this study, we propose EVA, an encrypted vector arithmetic language and compiler for efficient homomorphic computation, designed to mitigate the performance overhead of padding and unpadding operations.

Keywords: Homomorphic encryption, compiler, neural networks, privacy-preserving machine learning

ACM Reference Format:
Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Mumtaz, 2020. EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation. In Proceedings of the 2020 ACM SIGPLAN Conference on Programming Languages (PLAS '20), December 1–5, 2020, Virtual Event, CA, USA, Article 12 (2020), 28 pages.

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 unfeasible 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, has 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 systematically review FHE tools and compilers. We perform experiments to evaluate their 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 tool development.

Gartner projects [19] that “by 2025, at least 20% of companies will have a budget for projects that include fully homomorphic encryption.”

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, insurance, 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

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 some of the engineering challenges in this space, we have seen 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]

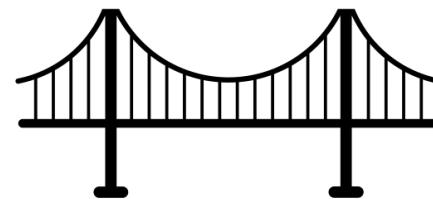
or the Fast Fully Homomorphic Encryption Library over the

S&P 2021, arXiv 2101.07078

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

Usable FHE

Advanced
Cryptography

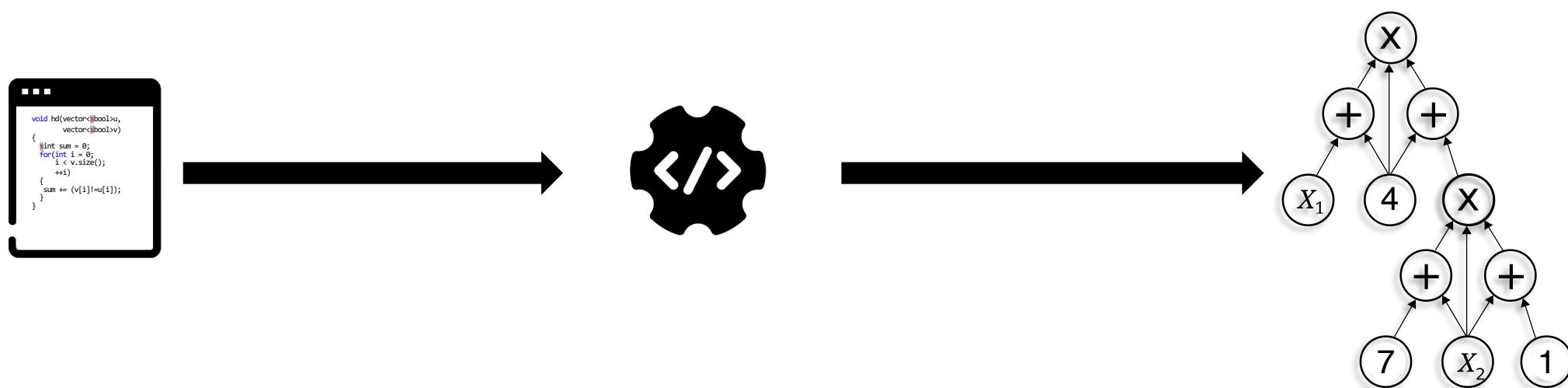


Programming
Languages

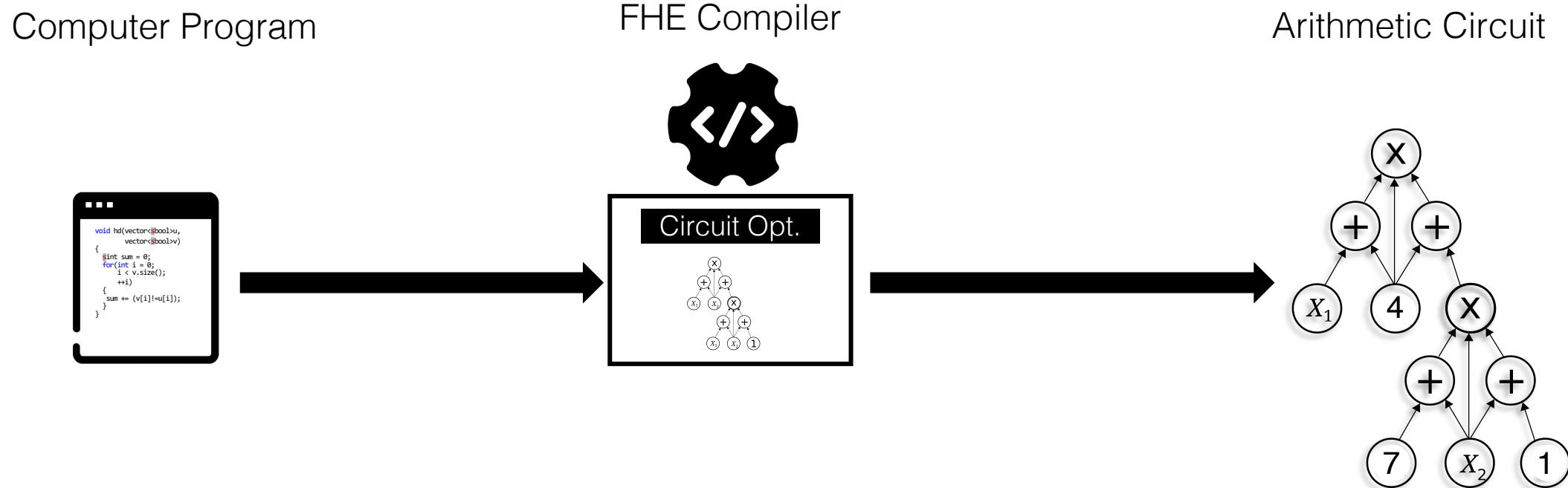
- 1 What makes developing FHE applications hard?
- 2 How are compilers addressing these complexities?
- 3 Roadmap to End-to-End FHE development
- 4 HECO: Automatic Code Optimizations for FHE

End-to-End FHE Toolchain

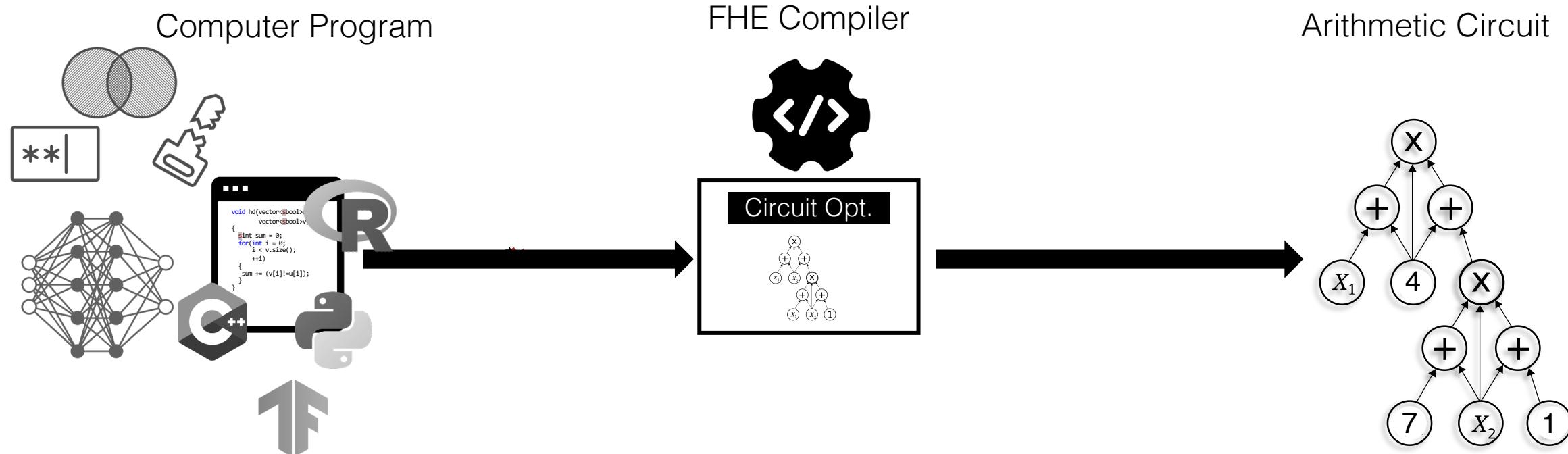
Computer Program FHE Compiler Arithmetic Circuit



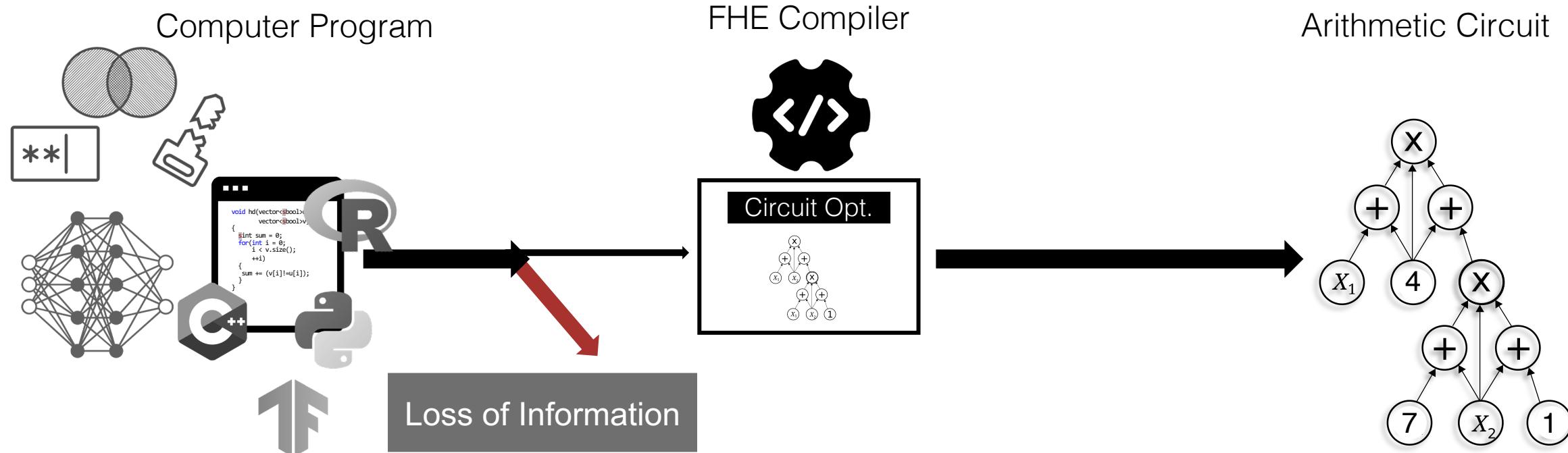
End-to-End FHE Toolchain



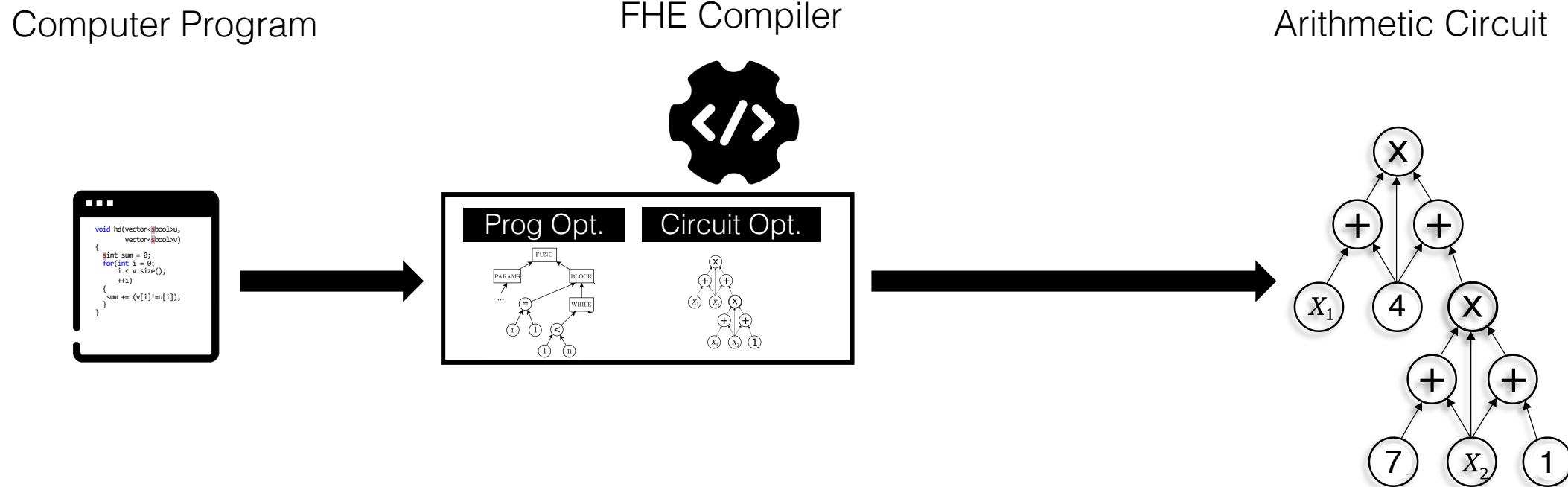
End-to-End FHE Toolchain



End-to-End FHE Toolchain

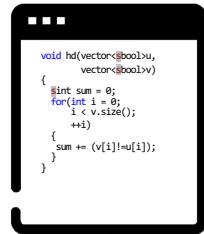


End-to-End FHE Toolchain

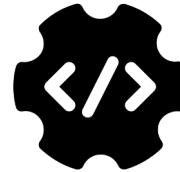


End-to-End FHE Toolchain

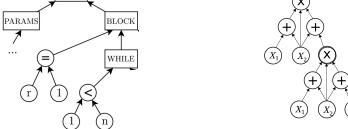
Computer Program



FHE Compiler



Prog Opt. Circuit Opt.

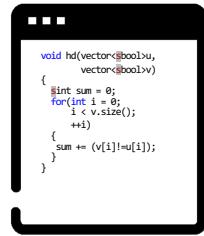


Secure and Efficient FHE Solutions

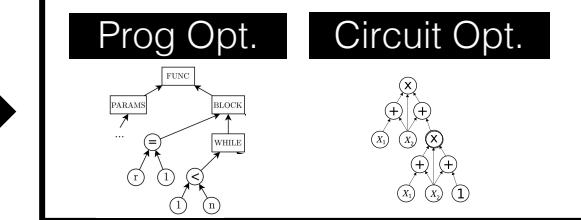
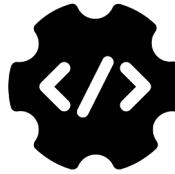


End-to-End FHE Toolchain

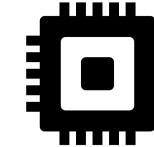
Computer Program



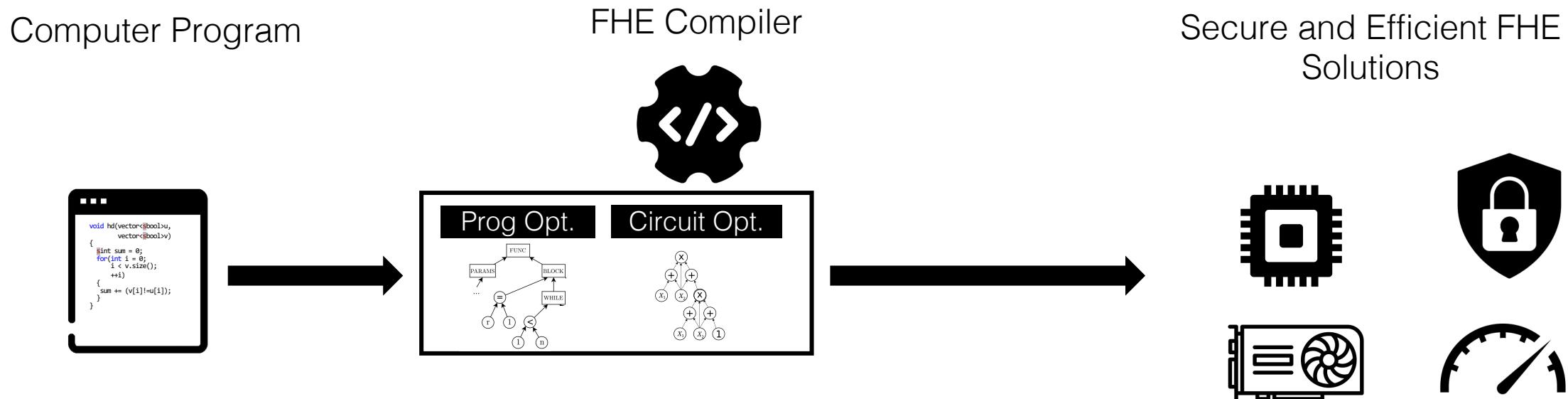
FHE Compiler



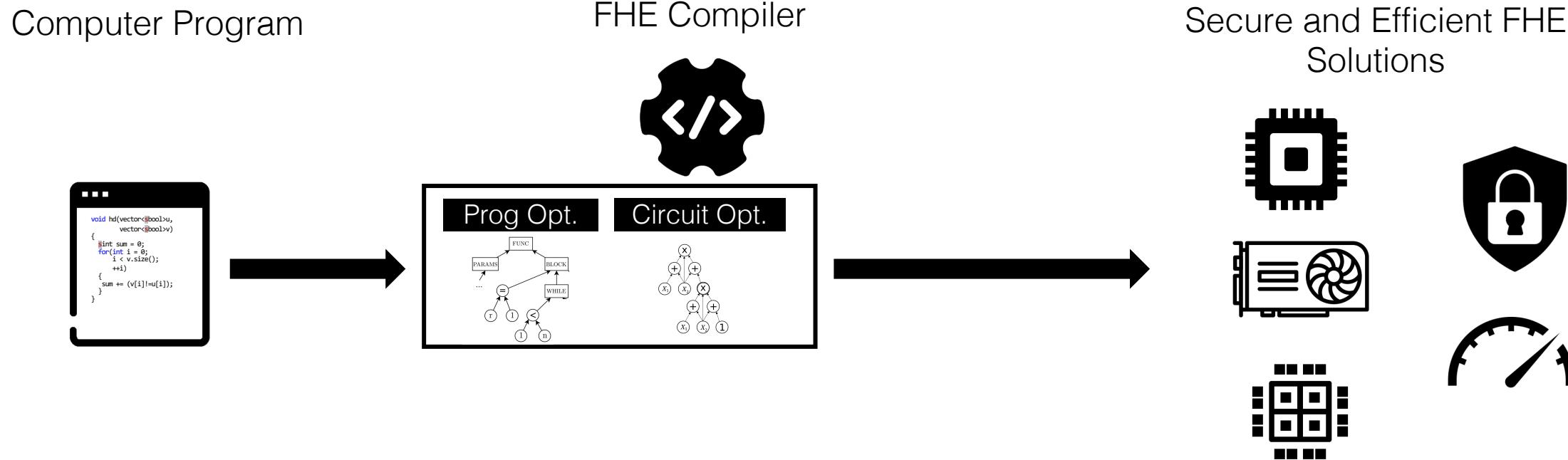
Secure and Efficient FHE
Solutions



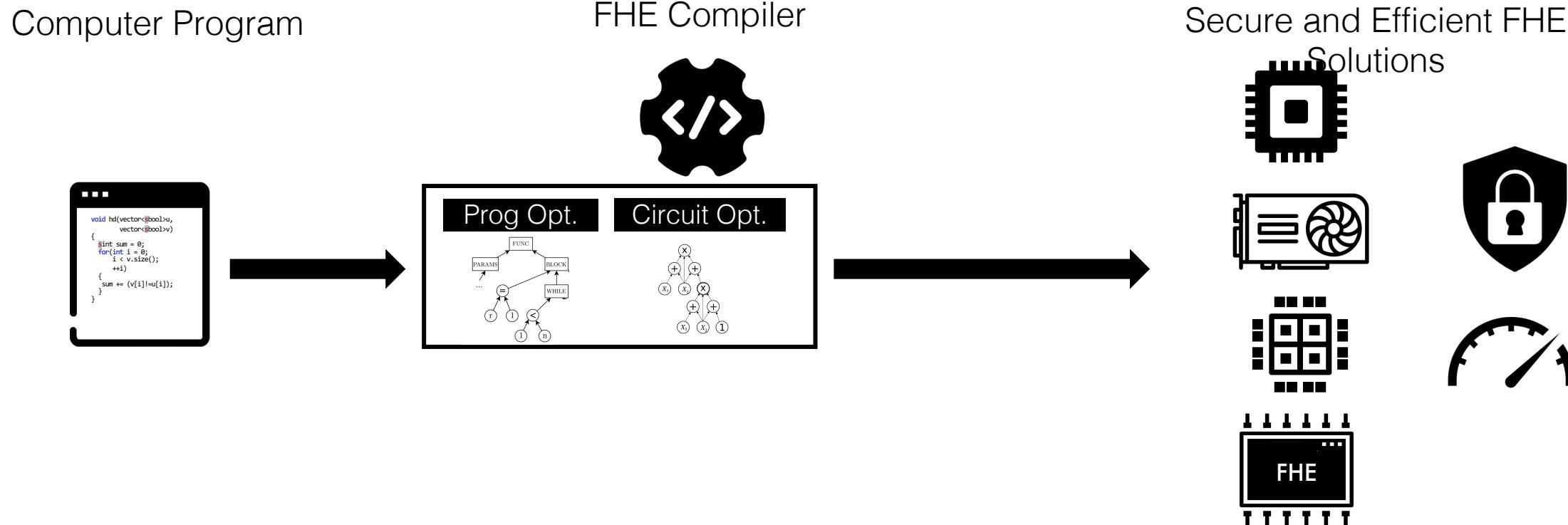
End-to-End FHE Toolchain



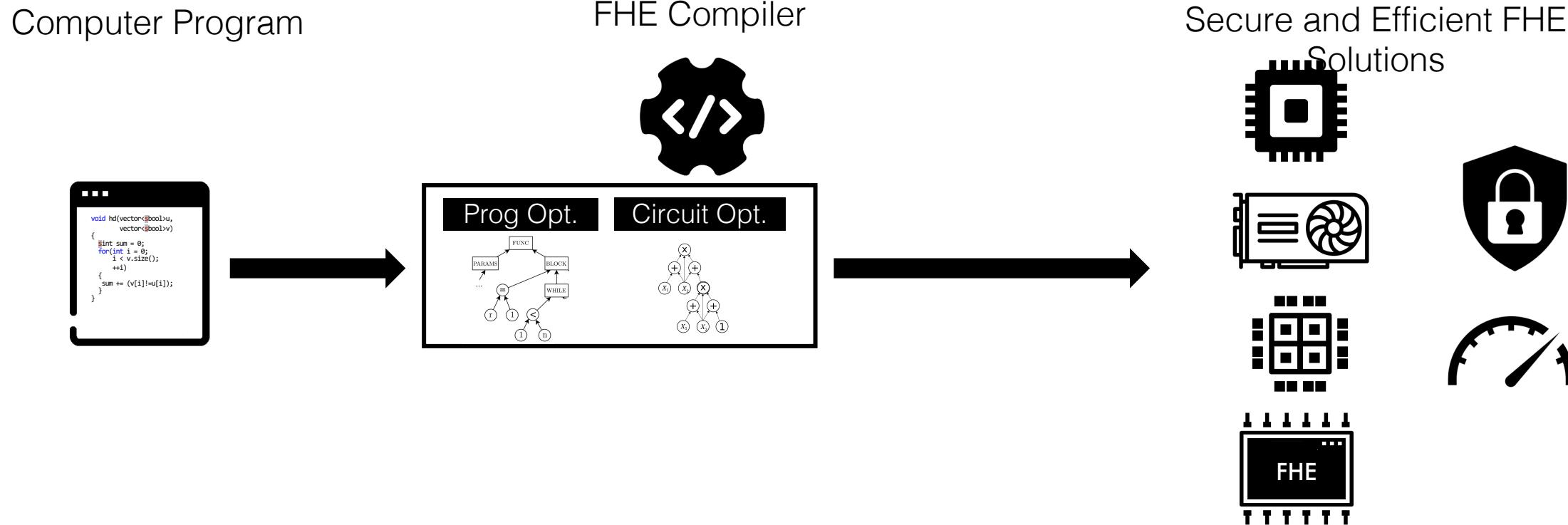
End-to-End FHE Toolchain



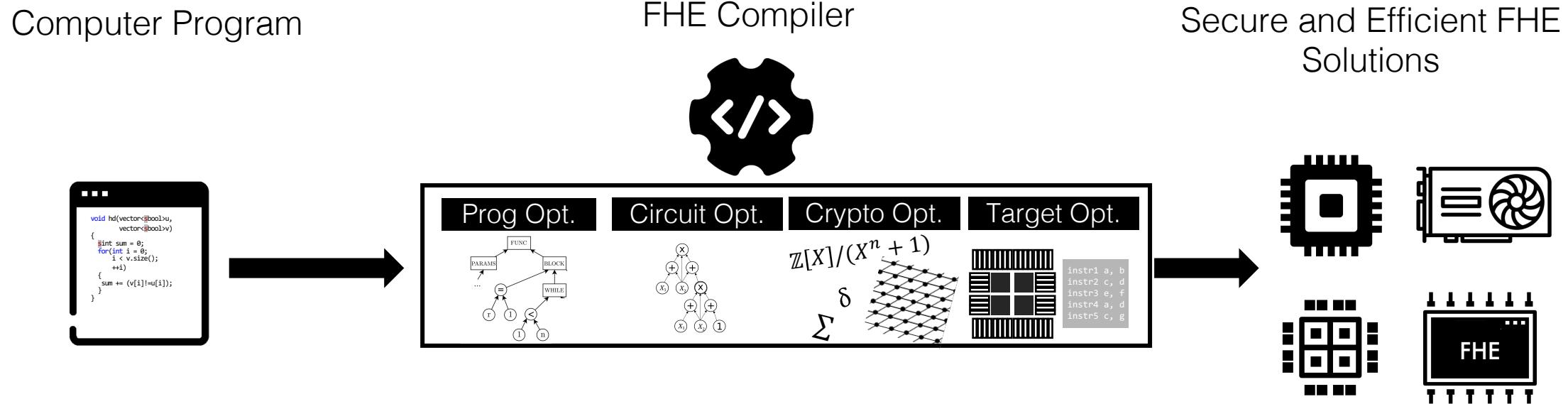
End-to-End FHE Toolchain



End-to-End FHE Toolchain

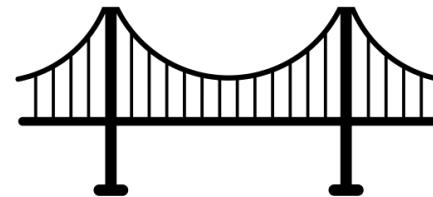


End-to-End FHE Toolchain



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



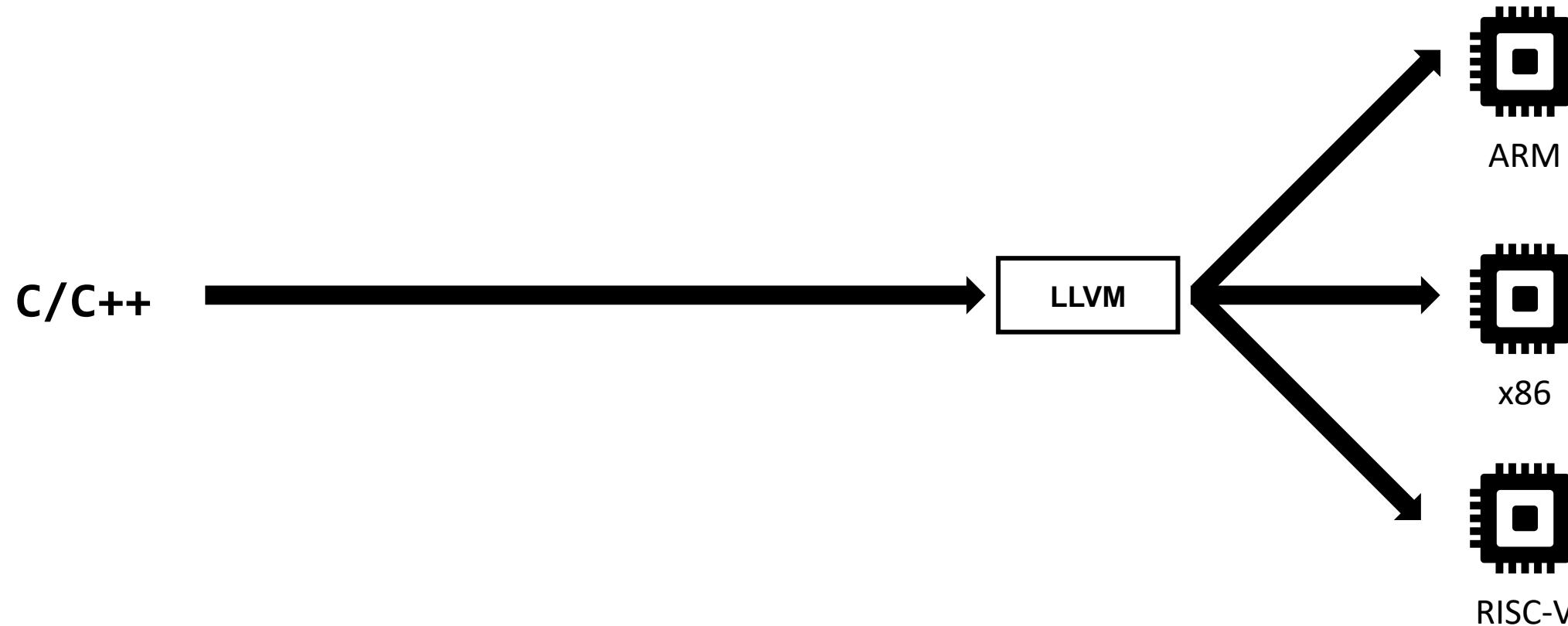
Compiler Pipelines



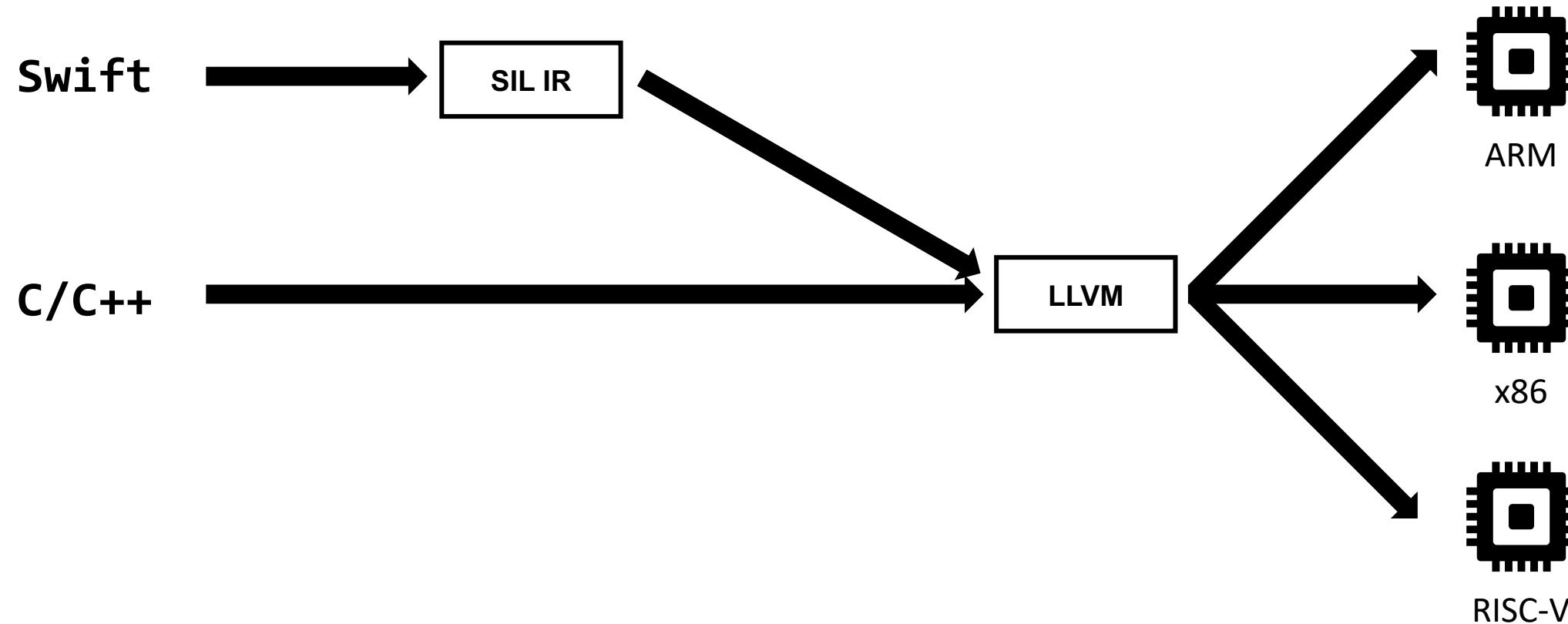
Compiler Pipelines



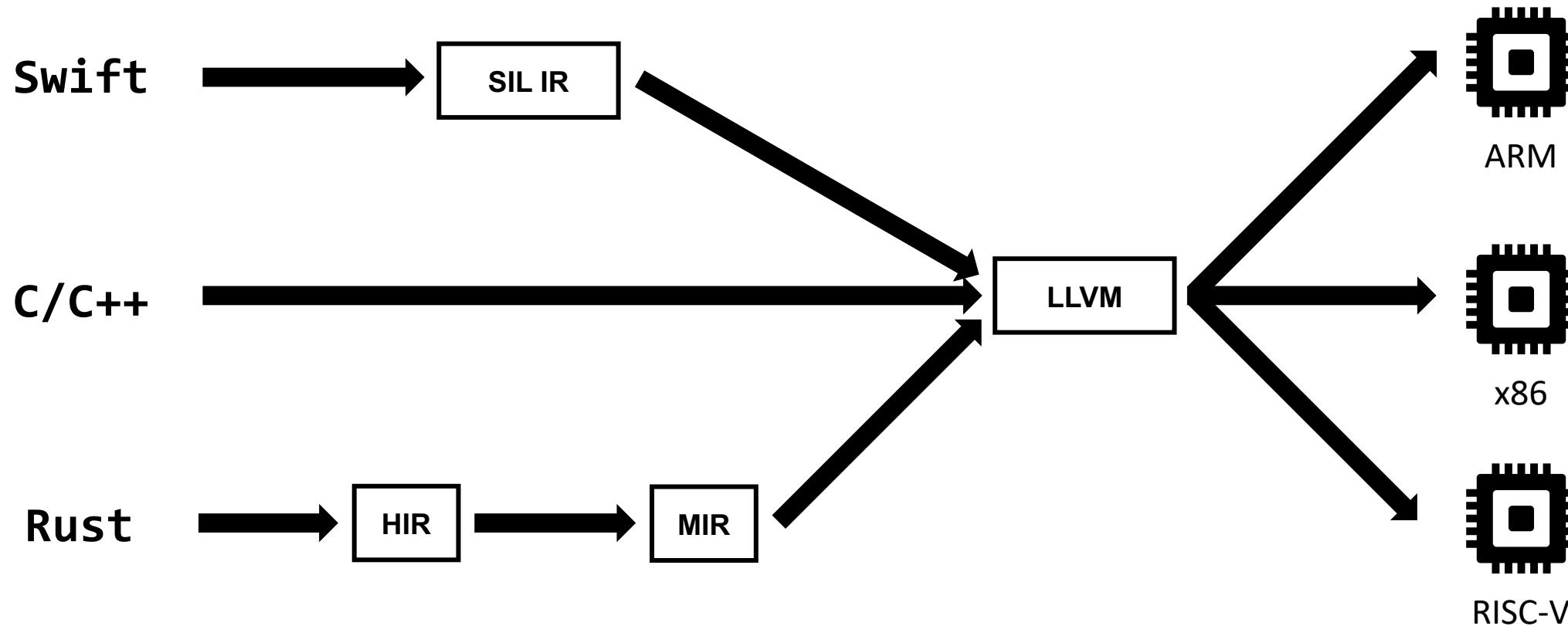
Compiler Pipelines



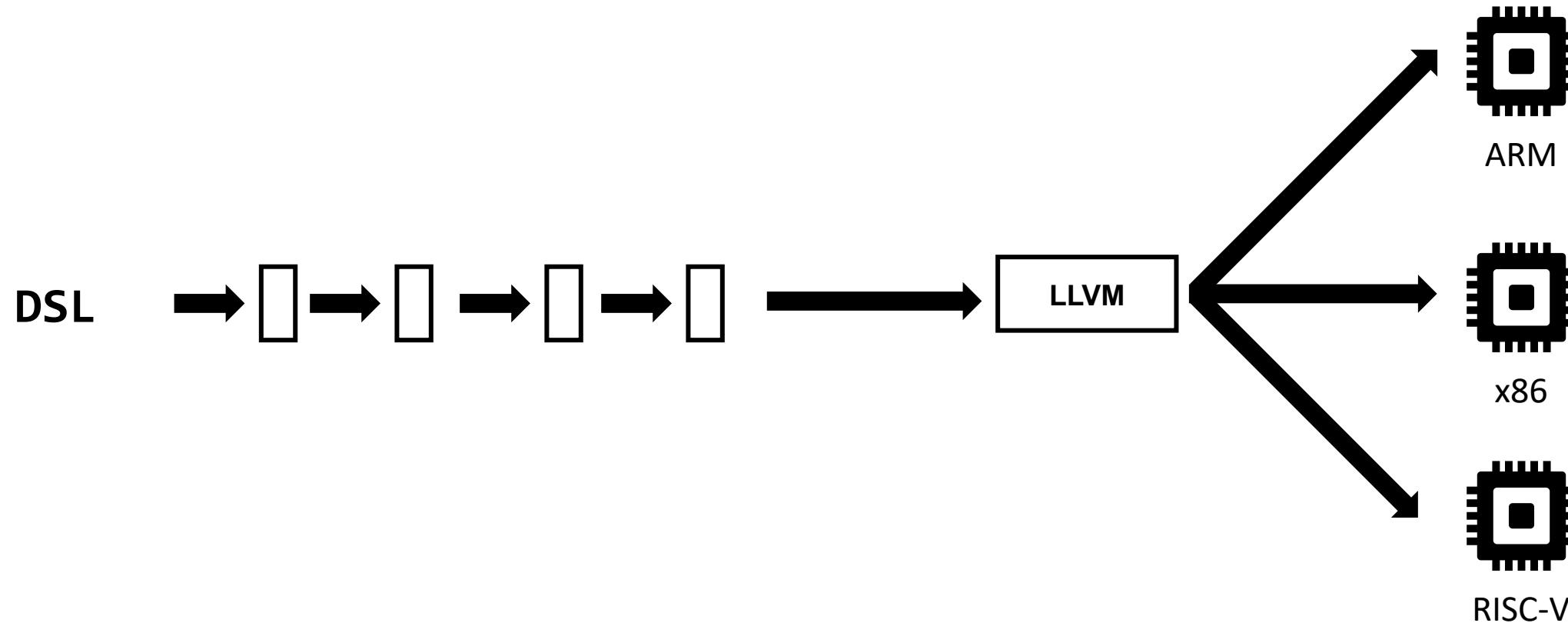
Compiler Pipelines



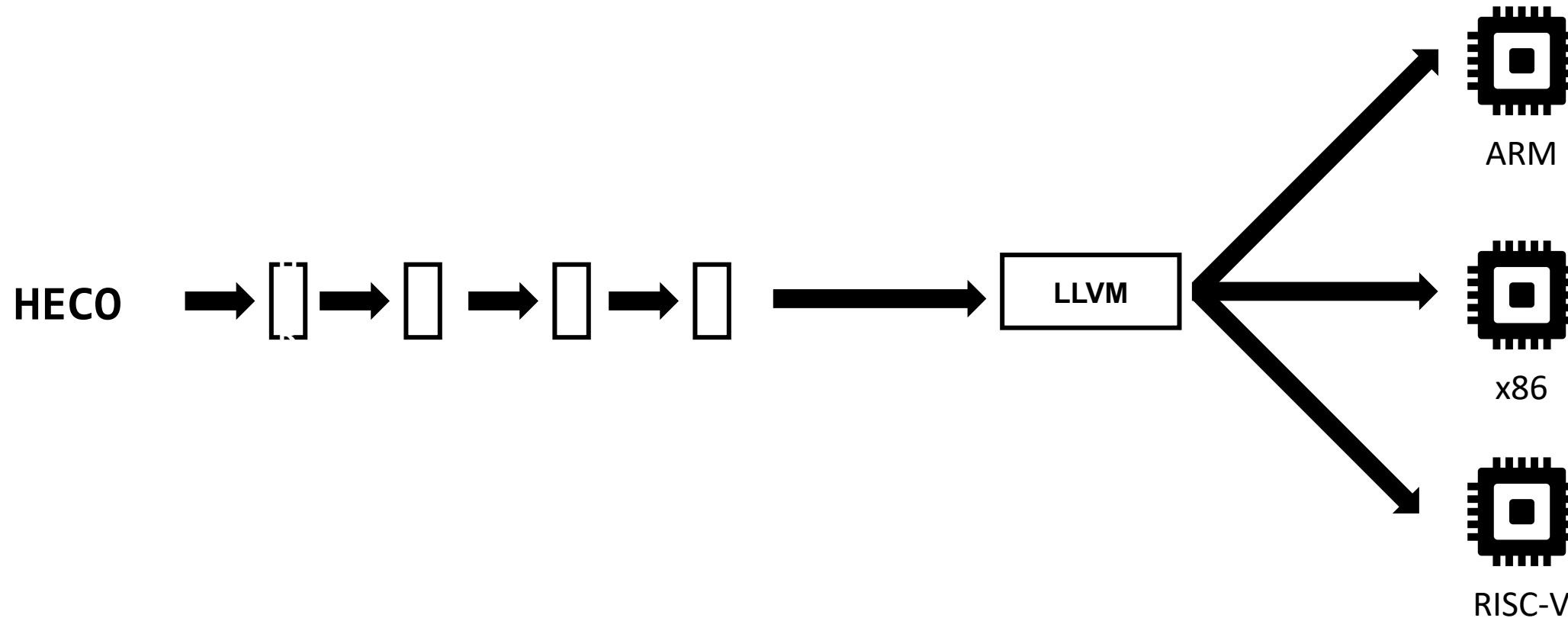
Compiler Pipelines



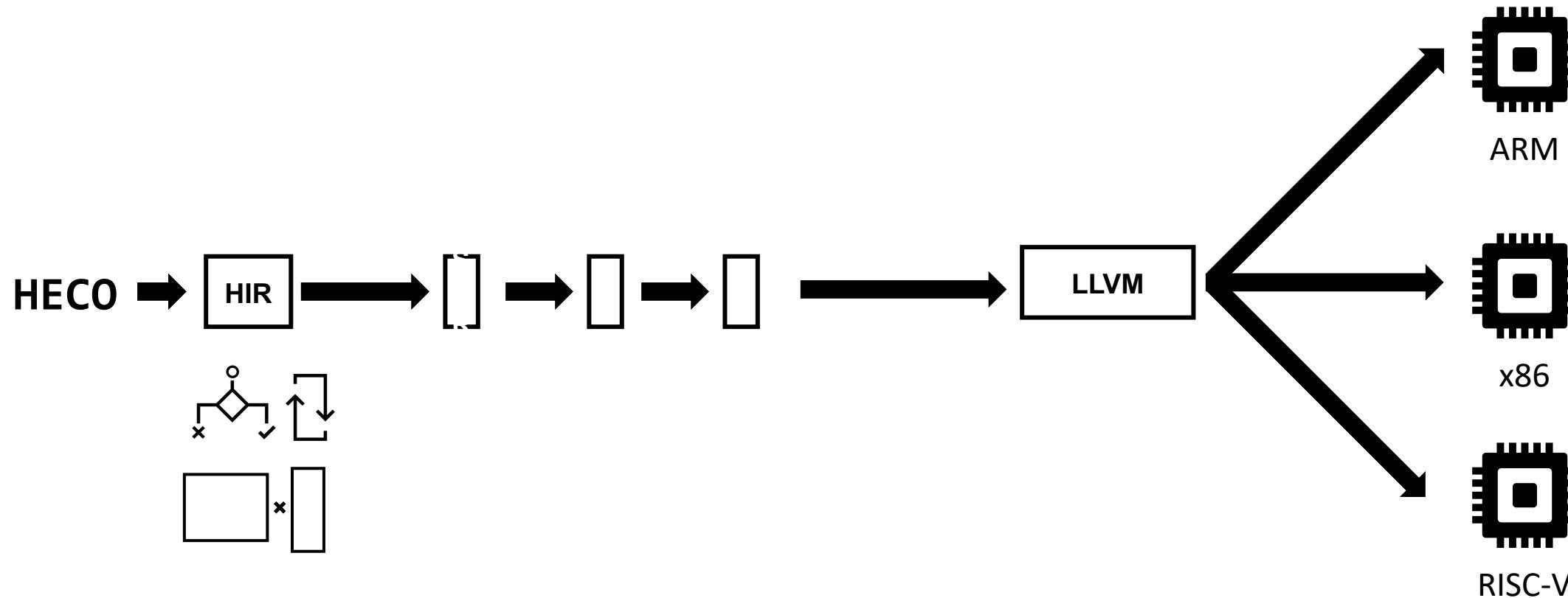
Compiler Pipelines



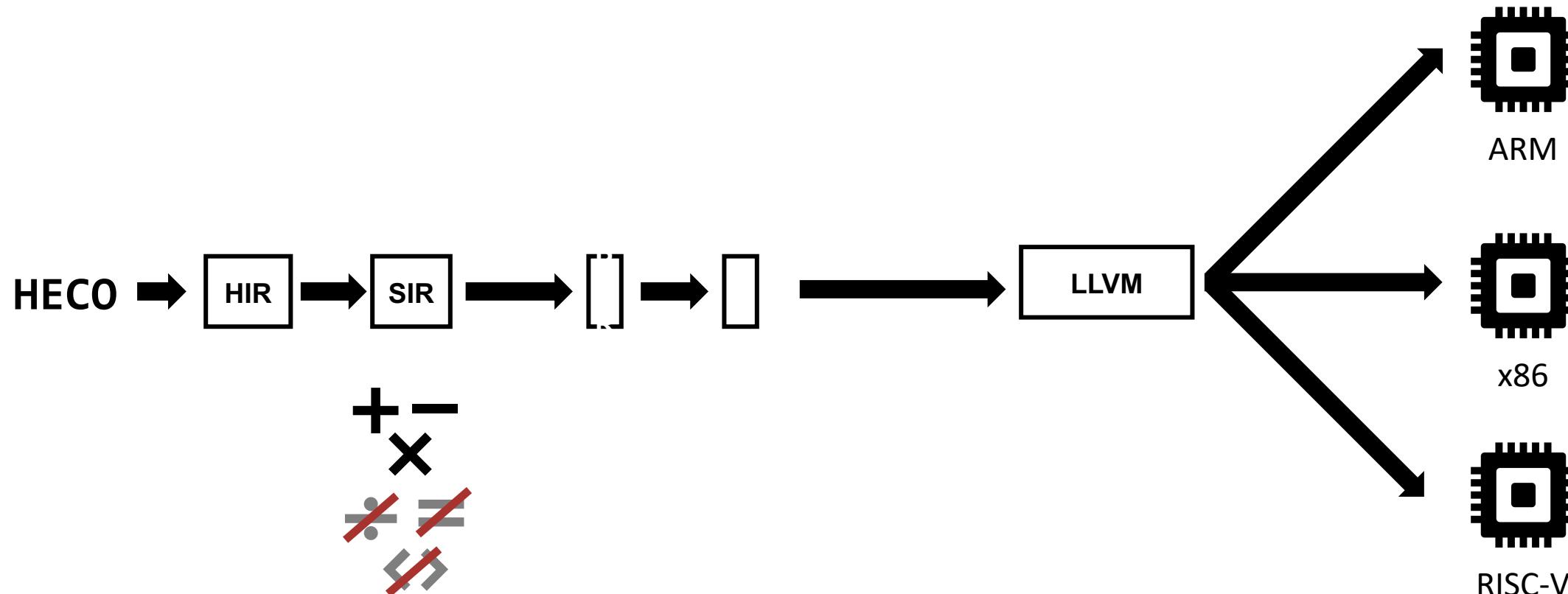
HECO Pipeline



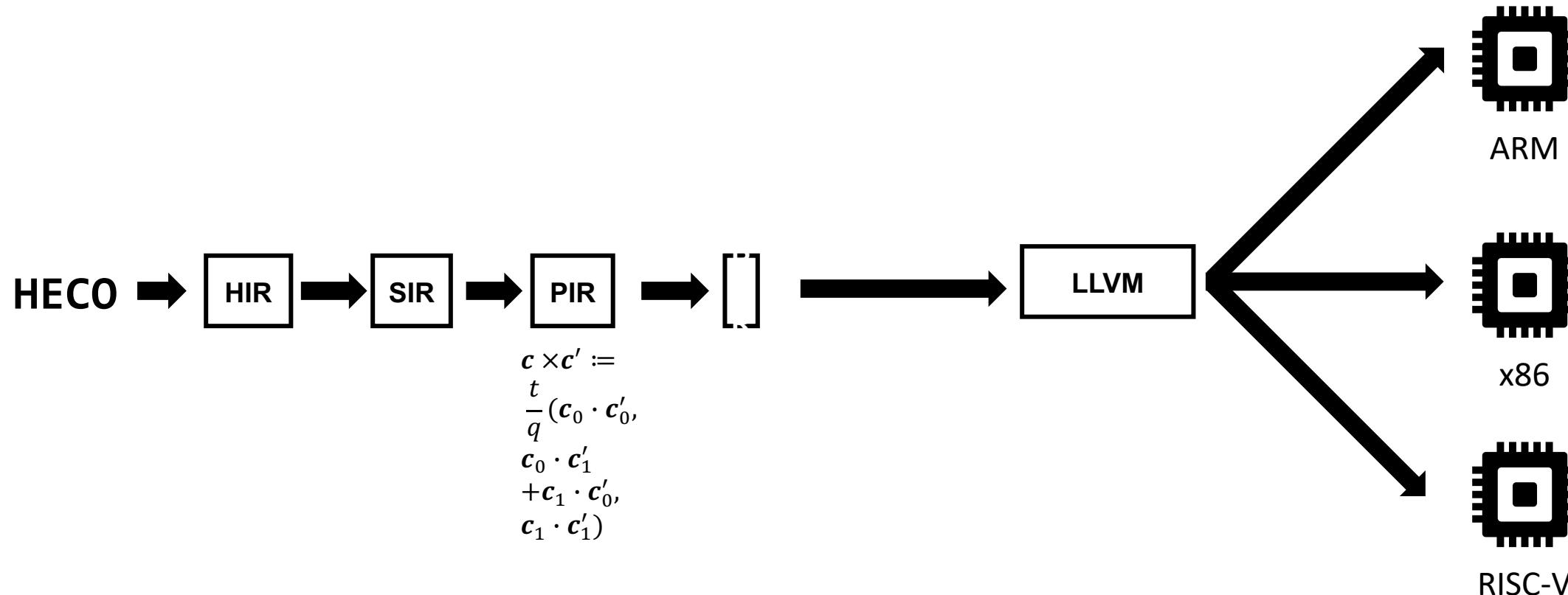
HECO Pipeline



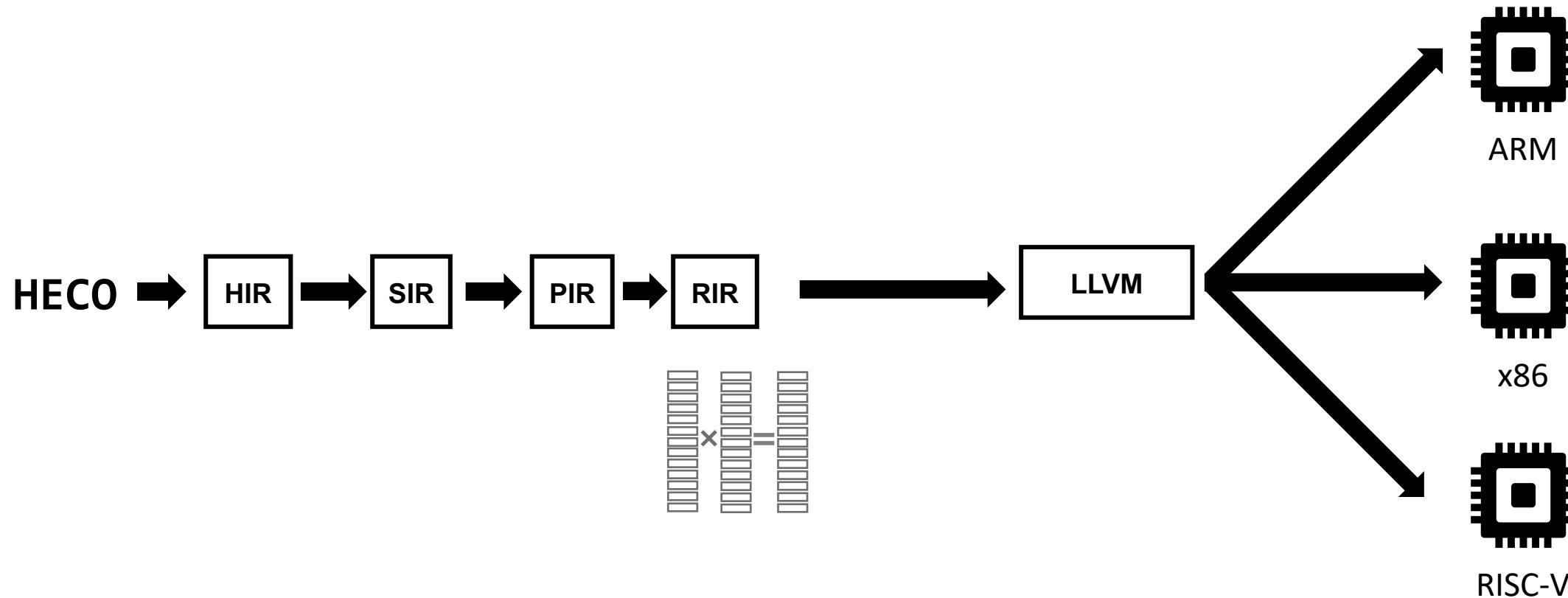
HECO Pipeline



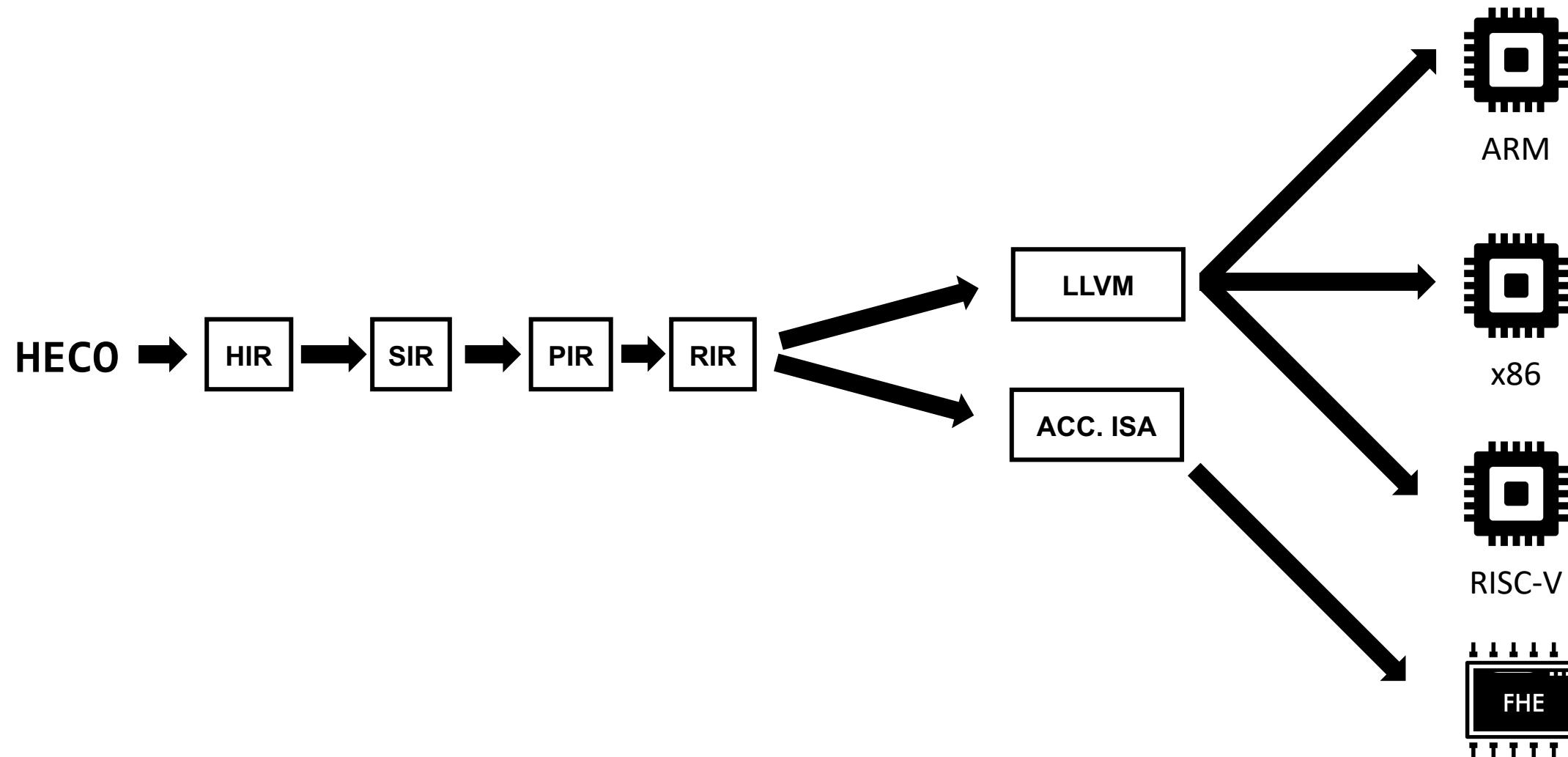
HECO Pipeline



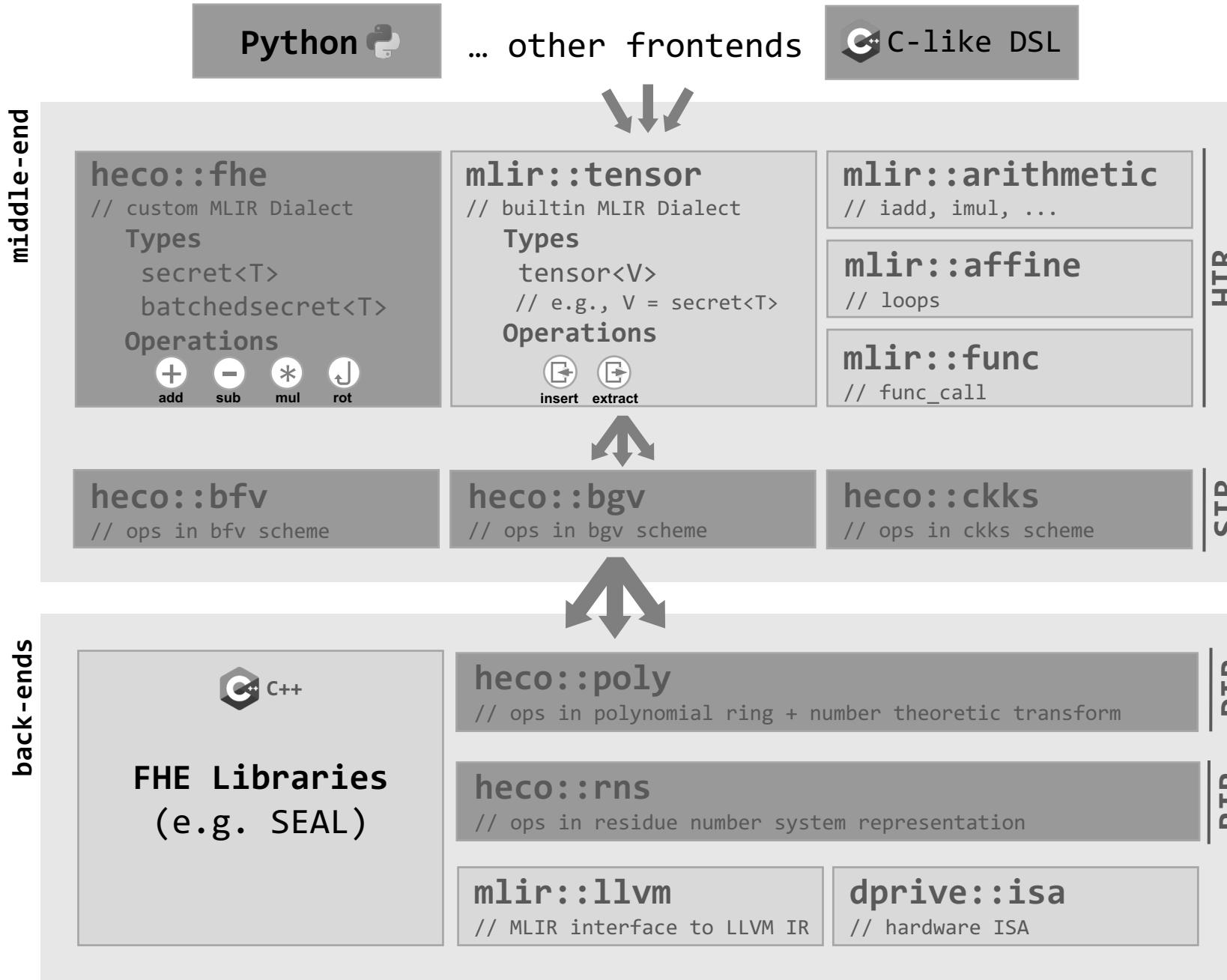
HECO Pipeline



HECO Pipeline



HECO Architecture

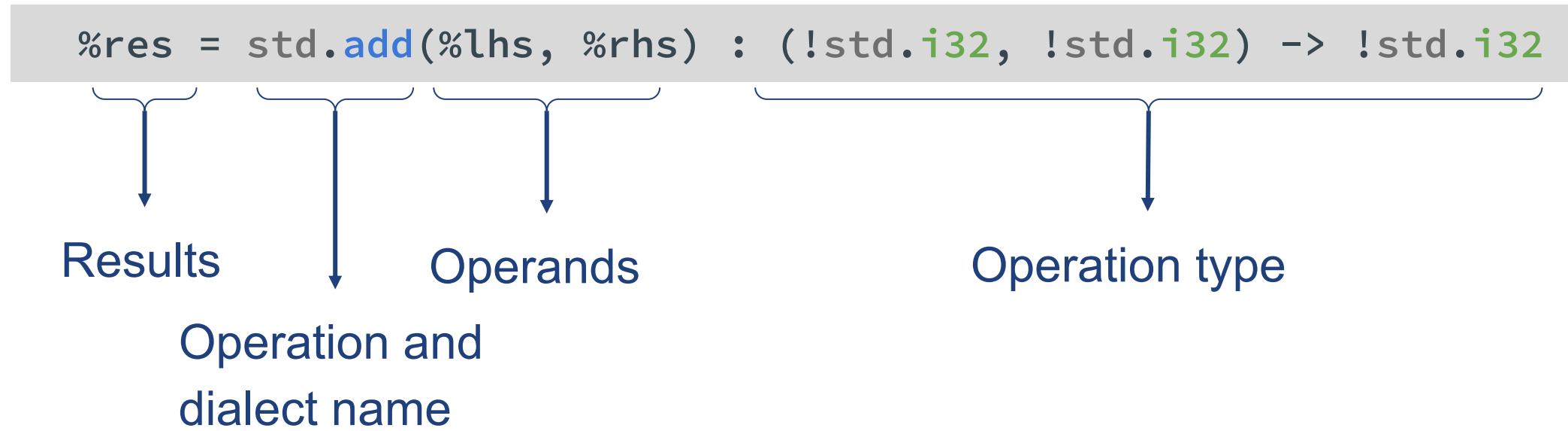


FHE Intermediate Representations



What is an Intermediate Representation?

- Operations & Types



- Semantics defined mostly through lowerings

FHE Compilation Pipeline

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

```
%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.ctx, !fhe.ctx) -> !fhe.ctx
```

```
%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])
```

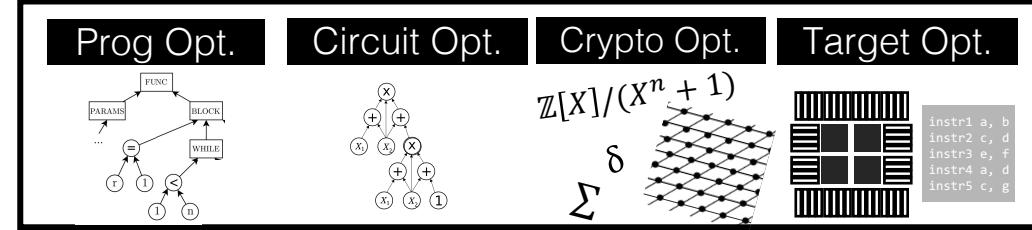
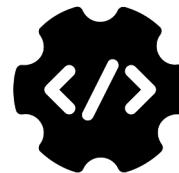
```
%l1 = poly.ntt(%lhs[1])
%r1 = poly.ntt(%rhs[1])
%t = poly.elem_mul(%l1, %r1)
%r2 = poly.inv_ntt(%t)
```

HECO: Modular End-to-End Design

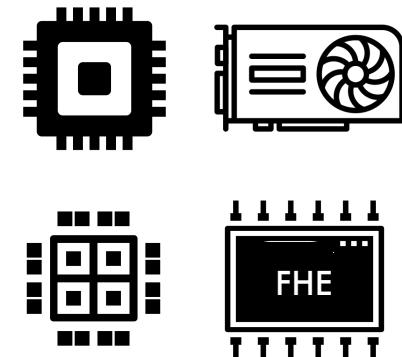
Computer Program

```
void hd(vector<bool>u,  
       vector<bool>v)  
{  
    int sum = 0;  
    for(int i = 0;  
        i < v.size();  
        ++i)  
    {  
        sum += (v[i] != u[i]);  
    }  
}
```

FHE Compiler



Secure and Efficient FHE
Solutions

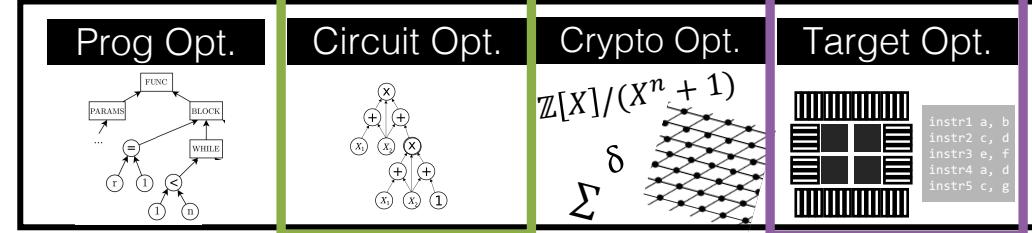


HECO: Modular End-to-End Design

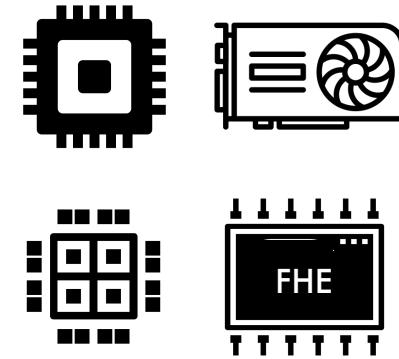
Computer Program

```
void hd(vector<bool>u,  
       vector<bool>v)  
{  
    int sum = 0;  
    for(int i = 0;  
        i < v.size();  
        ++i)  
    {  
        sum += (v[i] != u[i]);  
    }  
}
```

FHE Compiler

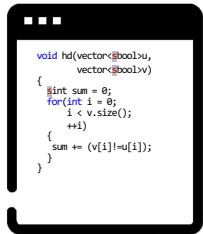


Secure and Efficient FHE
Solutions

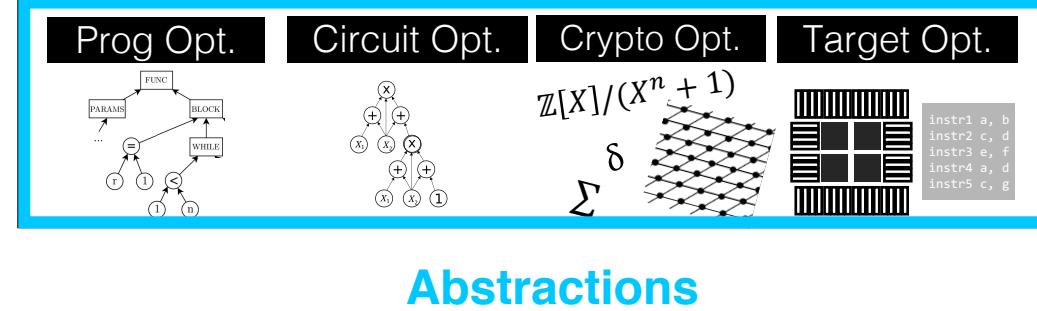
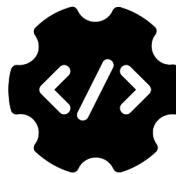


HECO: Modular End-to-End Design

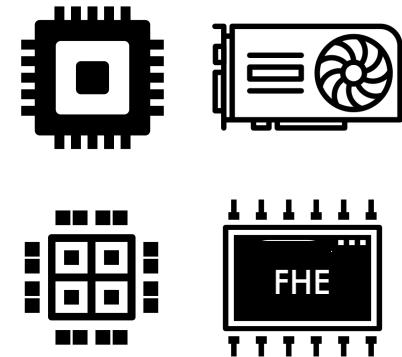
Computer Program



FHE Compiler



Secure and Efficient FHE Solutions



FHE Scheme Standardization

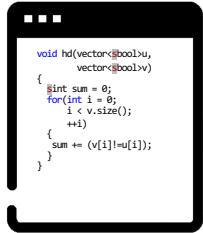
- HomomorphicEncryption.org (2018)
- Now turning into ISO/IEC AWI 18033-8

FHE Intermediate Representation Standardization

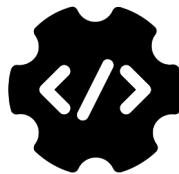
- 2018 draft API standard not adopted/implemented
- Significant FHE compiler efforts are accelerating
 - Need to re-visit standardization of abstractions
 - Expand beyond “FHE API” abstraction

HECO: Modular End-to-End Design

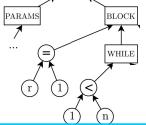
Computer Program



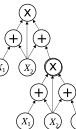
FHE Compiler



Prog Opt.



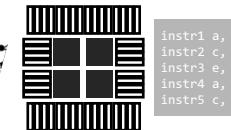
Circuit Opt.



Crypto Opt.

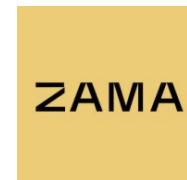
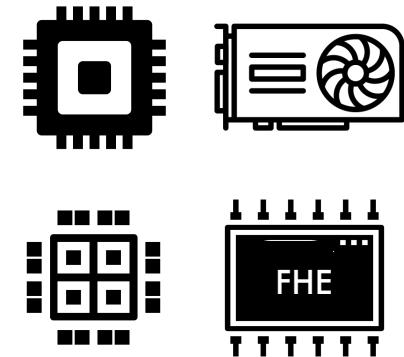
$$\mathbb{Z}[X]/(X^n + 1)$$
$$\sum \delta$$

Target Opt.



Abstractions

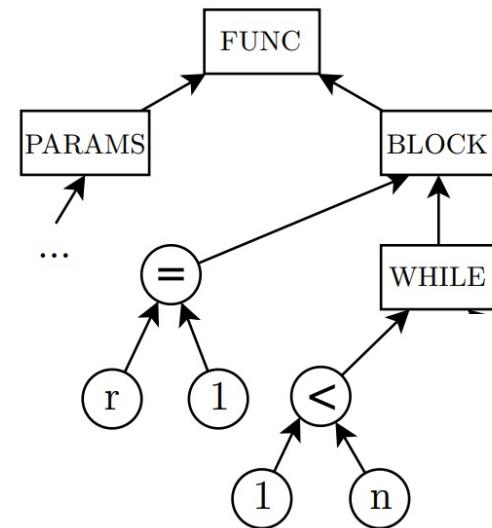
Secure and Efficient FHE Solutions



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

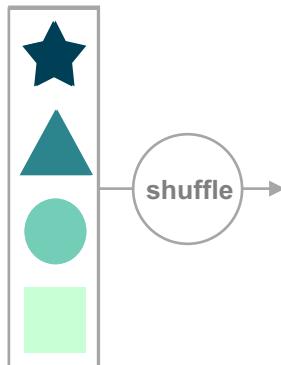


SIMD-like Parallelism

Standard C++	Batched FHE
<pre>int foo(int[] x,int[] y){\n\n int[] r;\n for(i = 0; i < 6; ++i){\n r[i] = x[i] * y[i]\n }\n return r;\n}</pre>	<pre>int foo(int[] a,int[] b){\n\n return a * b;\n}</pre>



SIMD Batching



No efficient free permutation or scatter/gather

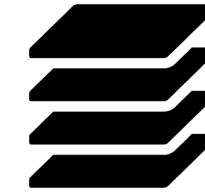
SIMD-like Parallelism

Standard C++

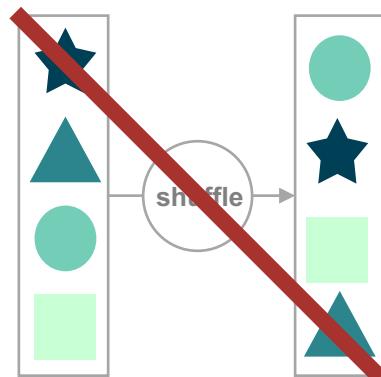
```
int foo(int[] x,int[] y){  
    int[] r;  
    for(i = 0; i < 6; ++i){  
        r[i] = x[i] * y[i]  
    }  
    return r;  
}
```

Batched FHE

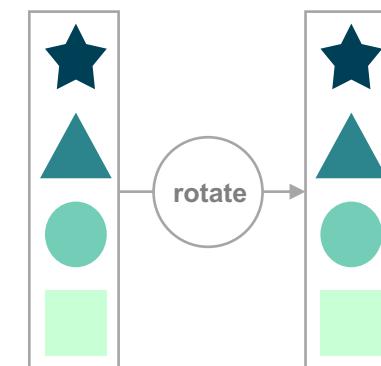
```
int foo(int[] a,int[] b){  
    return a * b;  
}
```



SIMD Batching



No efficient free permutation or scatter/gather



Only cyclical rotations

SIMD-like Parallelism

Standard C++

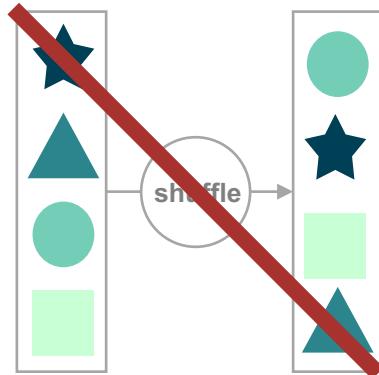
```
int foo(int[] x,int[] y){  
    int[] r;  
    for(i = 0; i < 6; ++i){  
        r[i] = x[i] * y[i]  
    }  
    return r;  
}
```

Batched FHE

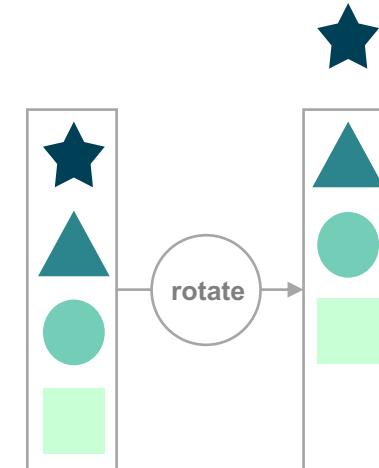
```
int foo(int[] a,int[] b){  
    return a * b;  
}
```



SIMD Batching



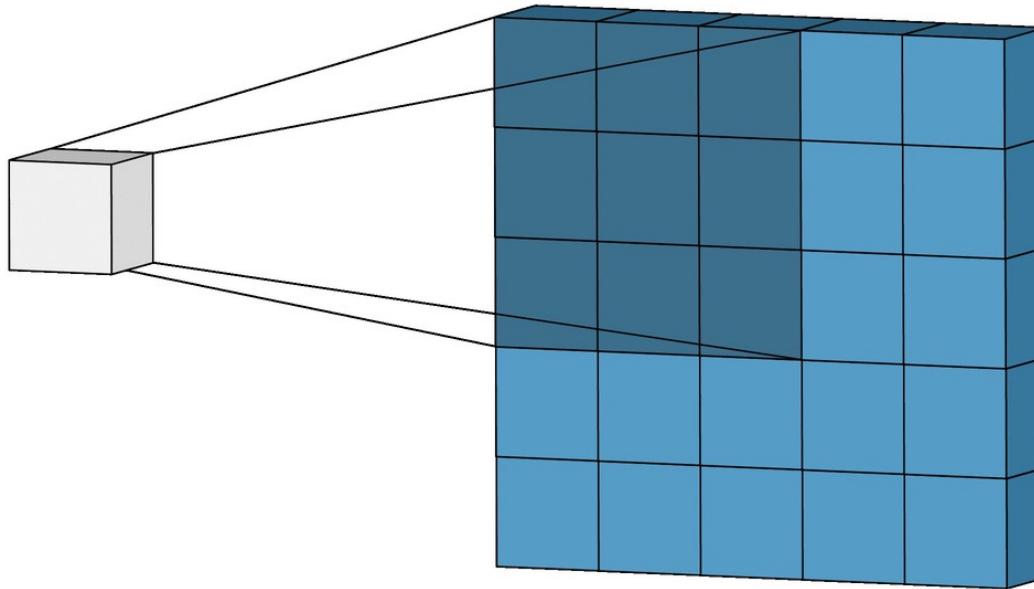
No efficient free permutation or scatter/gather



Only cyclical rotations

Example Input Program

```
1 LaplacianSharpening(secret_int img[], int imgSize) {  
2     secret_int img2[];  
3     int weightMatrix[3][3] = [1 1 1; 1 -8 1; 1 1 1];  
4     for (int x = 1; x < imgSize-1; ++x) {  
5         for (int y = 1; y < imgSize-1; ++y) {  
6             int value = 0;  
7             for (int j = -1; j < 2; ++j) {  
8                 for (int i = -1; i < 2; ++i) {  
9                     value += weightMatrix[i+1][j+1]  
10                      * img[imgSize*(x+i)+y+j];  
11                 }  
12             }  
13             img2[imgSize*x+y] = 2* img[imgSize*x+y] - value;  
14         }  
15     }  
16     return img2;  
17 }
```



FHE can include dramatic transformations!

```
1 LaplacianSharpening(secret_int img[], int imgSize) {  
2     secret_int img2[];  
3     int weightMatrix[3][3] = [1 1 1; 1 -8 1; 1 1 1];  
4     for (int x = 1; x < imgSize-1; ++x) {  
5         for (int y = 1; y < imgSize-1; ++y) {  
6             int value = 0;  
7             for (int j = -1; j < 2; ++j) {  
8                 for (int i = -1; i < 2; ++i) {  
9                     value += weightMatrix[i+1][j+1]  
10                        * img[imgSize*(x+i)+y+j];  
11                 }  
12             }  
13             img2[imgSize*x+y] = 2* img[imgSize*x+y] - value;  
14         }  
15     }  
16     return img2;  
17 }
```



```
1 LaplacianSharpening(secret_int img[], int imgSize) {  
2     secret_int[] t0 = fhe.rotate (img, -imgSize-1);  
3     secret_int[] t1 = fhe.rotate (img, -imgSize);  
4     secret_int[] t2 = fhe.rotate (img, -imgSize+1);  
5     secret_int[] t3 = fhe.rotate (img, -1);  
6     secret_int[] t4 = fhe.rotate (img, 1);  
7     secret_int[] t5 = fhe.rotate (img, imgSize-1);  
8     secret_int[] t6 = fhe.rotate (img, imgSize);  
9     secret_int[] t7 = fhe.rotate (img, imgSize+1);  
10    secret_int[] t8 = fhe.mul (img, -8);  
11    return fhe.add(t0,t1,t2,t3,t4,t5,t6,t7,t8);  
12 }
```

FHE can include dramatic transformations!

```

1 LaplacianSharpening(secret_int img[], int imgSize) {
2     secret_int img2[];
3     int weightMatrix[3][3] = [1 1 1; 1 -8 1; 1 1 1];
4     for (int x = 1; x < imgSize-1; ++x) {
5         for (int y = 1; y < imgSize-1; ++y) {
6             int value = 0;
7             for (int j = -1; j < 2; ++j) {
8                 for (int i = -1; i < 2; ++i) {
9                     value += weightMatrix[i+1][j+1]
10                    * img[imgSize*(x+i)+y+j];
11                }
12            }
13            img2[imgSize*x+y] = 2* img[imgSize*x+y] - value;
14        }
15    }
16    return img2;
17 }
```



```

1 LaplacianSharpening(secret_int img[], int imgSize) {
2     secret_int[] t0 = fhe.rotate (img, -imgSize-1);
3     secret_int[] t1 = fhe.rotate (img, -imgSize);
4     secret_int[] t2 = fhe.rotate (img, -imgSize+1);
5     secret_int[] t3 = fhe.rotate (img, -1);
6     secret_int[] t4 = fhe.rotate (img, 1);
7     secret_int[] t5 = fhe.rotate (img, imgSize-1);
8     secret_int[] t6 = fhe.rotate (img, imgSize);
9     secret_int[] t7 = fhe.rotate (img, imgSize+1);
10    secret_int[] t8 = fhe.mul (img, -8);
11    return fhe.add(t0,t1,t2,t3,t4,t5,t6,t7,t8);
12 }
```

Simpler Example: Hamming Distance

```
1 HammingDistance(secret_int x[], secret_int x[], int len) {  
2     secret_int sum = 0;  
3     for (int i = 1; i < len-1; ++i) {  
4         int a = x[i] - y[i];  
5         int b = a * a;  
6         sum += b;  
7     }  
8     return sum;  
9 }
```

0	0
1	0
1	1
0	1

Simpler Example: Hamming Distance

```
1 func private @HammingDistance(%x: ..., %y: ..., %len ...) -> ... {  
2   %c0 = fhe.constant 0 : ...  
3   %0 = affine.for %i = 0 to %len iter_args(%s = %c0) -> (...) {  
4     %1 = fhe.sub(%x[%i], %y[%i]): ...  
5     %2 = fhe.multiply(%1, %1) : ...  
6     %3 = fhe.add(%s, %3) : ...  
7     affine.yield %3 : ...  
8   }  
9   func.return %0 ...  
10 }
```

Emulating Index Access is expensive!

Simpler Example: Hamming Distance

```
1 func private @HammingDistance(%x: ..., %y: ..., %len ...) -> ... {  
2     %c0 = fhe.constant 0 : ...  
3     %0 = affine.for %i = 0 to %len iter_args(%s = %c0) -> (...) {  
4         %1 = fhe.sub(%x[%i], %y[%i]) : ...  
5         %2 = fhe.multiply(%1, %1) : ...  
6         %3 = fhe.add(%s, %3) : ...  
7         affine.yield %3 : ...  
8     }  
9     func.return %0 ...  
10 }
```

11
12
13
14
15
16

Simpler Example: Hamming Distance

```
1 func private @HammingDistance4(%x: ..., %y: ...) -> ... {  
2   %c0 = fhe.constant 0 : ...  
3   %1 = fhe.sub(%x[0], %y[0]) : ...  
4   %2 = fhe.multiply(%1, %1) : ...  
5   %3 = fhe.add(%s, %3) : ...  
6  
7   func.return %3 ...  
8 }  
9  
10  
11  
12  
13  
14  
15  
16
```

Simpler Example: Hamming Distance

```
1 func private @HammingDistance4(%x: ..., %y: ...) -> ... {  
2     %c0 = fhe.constant 0 : ...  
3     %1 = fhe.sub(%x[0], %y[0]) : ...  
4     %2 = fhe.multiply(%1, %1) : ...  
5     %3 = fhe.add(%s, %3) : ...  
6     %4 = fhe.sub(%x[1], %y[1]) : ...  
7     %5 = fhe.multiply(%4, %4) : ...  
8     %6 = fhe.add(%3, %5) : ...  
9     func.return %6 ...  
10 }  
11  
12  
13  
14  
15  
16
```

Simpler Example: Hamming Distance

```
1 func private @HammingDistance4(%x: ..., %y: ...) -> ... {  
2     %c0 = fhe.constant 0 : ...  
3     %1 = fhe.sub(%x[0], %y[0]) : ...  
4     %2 = fhe.multiply(%1, %1) : ...  
5     %3 = fhe.add(%s, %3) : ...  
6     %4 = fhe.sub(%x[1], %y[1]) : ...  
7     %5 = fhe.multiply(%4, %4) : ...  
8     %6 = fhe.add(%3, %5) : ...  
9     %7 = fhe.sub(%x[2], %y[2]) : ...  
10    %8 = fhe.multiply(%7, %7) : ...  
11    %9 = fhe.add(%6, %8) : ...  
12    %10 = fhe.sub(%x[3], %y[3]) : ...  
13    %11 = fhe.multiply(%10, %10) : ...  
14    %12 = fhe.add(%9, %11) : ...  
15    func.return %12 ...  
16 }
```

Current cost: 4 Multiplications & 8 Index Accesses

Simpler Example: Hamming Distance

```
1 func private @HammingDistance4(%x: ..., %y: ...) -> ... {  
2     %c0 = fhe.constant 0 : ...  
3     %1 = fhe.sub(%x[0], %y[0]) : ...  
4     %2 = fhe.multiply(%1, %1) : ...  
5     %3 = fhe.add(%s, %3) : ...  
6     %4 = fhe.sub(%x[1], %y[1]) : ...  
7     %5 = fhe.multiply(%4, %4) : ...  
8     %6 = fhe.add(%3, %5) : ...  
9     %7 = fhe.sub(%x[2], %y[2]) : ...  
10    %8 = fhe.multiply(%7, %7) : ...  
11    %9 = fhe.add(%6, %8) : ...  
12    %10 = fhe.sub(%x[3], %y[3]) : ...  
13    %11 = fhe.multiply(%10, %10) : ...  
14    %12 = fhe.add(%9, %11) : ...  
15    func.return %12 ...  
16 }
```

Combining Sequential Operations

Simpler Example: Hamming Distance

```
1 func private @HammingDistance4(%x: ..., %y: ...) -> ... {  
2     %c0 = fhe.constant 0 : ...  
3     %1 = fhe.sub(%x[0], %y[0]) : ...  
4     %2 = fhe.multiply(%1, %1) : ...  
5  
6     %4 = fhe.sub(%x[1], %y[1]) : ...  
7     %5 = fhe.multiply(%4, %4) : ...  
8  
9     %7 = fhe.sub(%x[2], %y[2]) : ...  
10    %8 = fhe.multiply(%7, %7) : ...  
11  
12    %10 = fhe.sub(%x[3], %y[3]) : ...  
13    %11 = fhe.multiply(%10, %10) : ...  
14    %12 = fhe.add(%c0, %2, %5, %8, %11) : ...  
15    func.return %12 ...  
16 }
```

Combining Sequential Operations

Simpler Example: Hamming Distance

```
1 func private @HammingDistance4(%x: ..., %y: ...) -> ... {  
2  
3     %1 = fhe.sub(%x[0], %y[0]) : ...  
4     %2 = fhe.multiply(%1, %1) : ...  
5  
6     %4 = fhe.sub(%x[1], %y[1]) : ...  
7     %5 = fhe.multiply(%4, %4) : ...  
8  
9     %7 = fhe.sub(%x[2], %y[2]) : ...  
10    %8 = fhe.multiply(%7, %7) : ...  
11  
12    %10 = fhe.sub(%x[3], %y[3]) : ...  
13    %11 = fhe.multiply(%10, %10) : ...  
14    %12 = fhe.add(%2, %5, %8, %11) ...  
15    func.return %12 ...  
16 }
```

Combining Sequential Operations

Simpler Example: Hamming Distance

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

Apply Operation to entire Vector

Simpler Example: Hamming Distance

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

Apply Operation to entire Vector

Simpler Example: Hamming Distance

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

Apply Operation to entire Vector

Simpler Example: Hamming Distance

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

Apply Operation to entire Vector

Each use now requires index access ☹

Simpler Example: Hamming Distance

```

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

Algorithm 2 Batching Pass

```

1: Algorithm BATCHPASS( $\mathcal{G}$ )
2:    $\mathcal{V}, \mathcal{E} \leftarrow \mathcal{G}$ 
3:   foreach  $op \in \mathcal{V} \wedge \text{type}(op) = \text{fhe.secret}$ :
4:      $ts \leftarrow \text{SELECTTARGETSLOT}(op, \mathcal{V}, \mathcal{E})$ 
5:     OPERANDCONVERSION( $op, ts, \mathcal{V}, \mathcal{E}$ )
6:     foreach  $v \in \mathcal{V} \wedge (op, v) \in \mathcal{E}$ :
7:        $u \leftarrow \text{fhe.extract}[v, ts]$ 
8:       REPLACE( $v, u, \mathcal{V}, \mathcal{E}$ )
9: procedure SELECTTARGETSLOT( $op, \mathcal{V}, \mathcal{E}$ )
10:  foreach  $v \in \mathcal{V} \wedge (op, v) \in \mathcal{E}$ :
11:    switch  $v$ :
12:      case fhe.insert $[_, i]$ : return  $i$ 
13:      case func.return: return 0
14:    foreach  $v \in \mathcal{V} \wedge (v, op) \in \mathcal{E}$ :
15:      switch  $o$ :
16:        case fhe.extract $[_, i]$ :
17:          return  $i$ 
18:    return  $\perp$ 
19: procedure OPERANDCONVERSION( $op, ts, \mathcal{V}, \mathcal{E}$ )
```

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

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

Translate Index Accesses to Rotations

Simpler Example: Hamming Distance

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

Translate Index Accesses to Rotations

Simpler Example: Hamming Distance

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

Translate Index Accesses to Rotations

Implicit Scalar Encoding: $s = s[0]$

Simpler Example: Hamming Distance

```
1 func private @HammingDistance4(%x: ..., %y: ...) -> ... {  
2   %1 = fhe.sub(%x, %y) : ...  
3   %2 = fhe.multiply(%1, %1) : ...  
4   %3 = fhe.rotate(%2, -1)  
5   %4 = fhe.rotate(%2, -2)  
6   %5 = fhe.rotate(%2, -3)  
7   %6 = fhe.add(%2, %3, %4, %5) : ...  
8   func.return %6 ...  
9 }
```

Translate Index Accesses to Rotations

Implicit Scalar Encoding: $s = s[0]$

Simpler Example: Hamming Distance

```
1 func private @HammingDistance4(%x: ..., %y: ...) -> ... {  
2   %1 = fhe.sub(%x, %y) : ...  
3   %2 = fhe.multiply(%1, %1) : ...  
4   %3 = fhe.rotate(%2, -1) |  
5   %4 = fhe.rotate(%2, -2) |  
6   %5 = fhe.rotate(%2, -3) |  
7   %6 = fhe.add(%2, %3, %4, %5) : ...  
8   func.return %6 ...  
9 }
```

Translate Index Accesses to Rotations

Implicit Scalar Encoding: $s = s[0]$

Significantly faster, but still $O(n)$ ☹

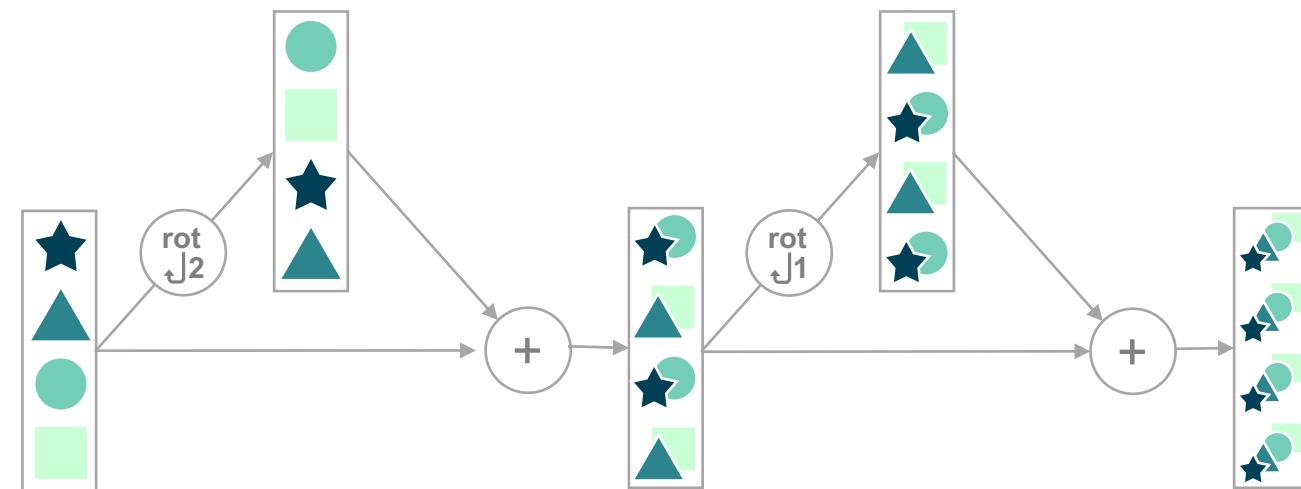
Exploit the fact that all addends have same origin!

Simpler Example: Hamming Distance

```

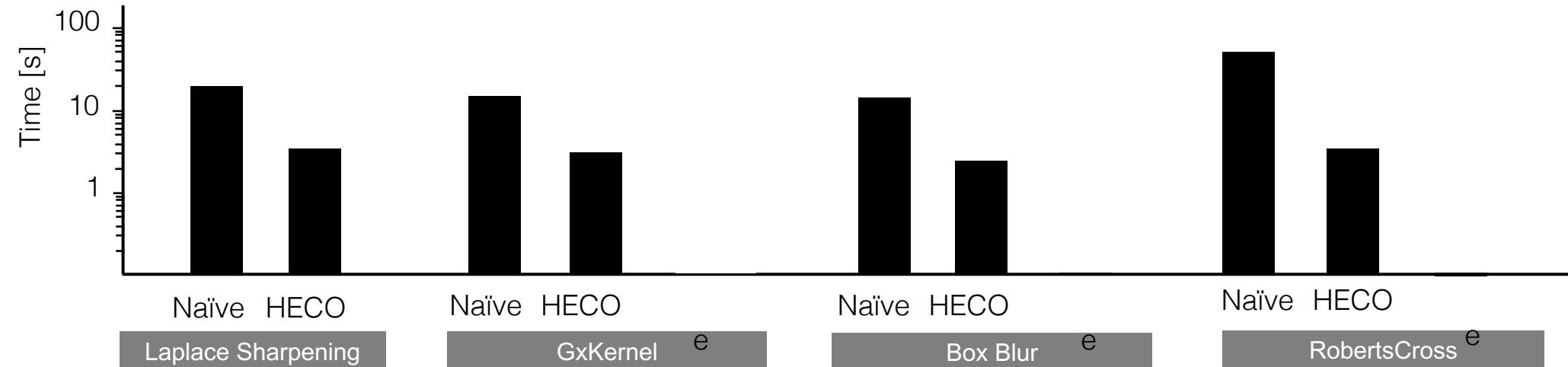
1 func private @HammingDistance4(%x: ..., %y: ...) -> ... {
2   %1 = fhe.sub(%x, %y) : ...
3   %2 = fhe.multiply(%1, %1) : ...
4   %3 = fhe.rotate(%2, -2) : ...
5   %4 = fhe.add(%2, %3) : ...
6   %5 = fhe.rotate(%4, -1) : ...
7   %6 = fhe.add(%4, %5) : ...
8   func.return %6 ...
9 }
```

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



Evaluation: Effect of Batching Optimizations

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



HECO: Automatic Code Optimizations for Efficient Fully Homomorphic Encryption

Alexander Viand, Patrick Jatke, Miro Haller, Anwar Hithnawi
ETH Zürich

Abstract

In recent years, Fully Homomorphic Encryption (FHE) has undergone several breakthroughs. Today, performance is no longer a major barrier to adoption. Instead, it is the complexity of deploying FHE in practice that is the primary limitation. Several frameworks have emerged recently to ease FHE development. However, most of these suffer from automation limits and impermeable boundaries to secure and efficient FHE implementations. This is a fundamental issue that needs to be addressed before we can realistically expect broader use of FHE. Automating these transformations is challenging because the requirements set of optimizations in FHE and our low-level tools are monolithic, and focus on individual optimizations. Therefore they fail to fully address the needs of end-to-end FHE development. In this paper, we present HECO, a new imperative program and emits efficient and secure FHE implementations, programs and units efficient and secure FHE implementation, extending the scope of optimizations beyond development, extending the scope of optimizations beyond the cryptographic challenges existing tools focus on.

1 Introduction

Privacy and security are gaining tremendous importance across all organizations, as public perception of these issues have shifted and expectations, as well as regulatory demands, have increased. This has led to a surge in demand for secure and confidential computing solutions that protect data confidentiality while it is in transit, rest, and in-use. Fully Homomorphic Encryption (FHE) is a key secure computation technology that enables systems to preserve the confidentiality of data end-to-end, including while in use. Hence, allowing outsourcing of computations without having to grant access to the data. In the last decade, advances in FHE schemes have



arxiv.org/abs/2202.01649

