HECO: Fully **H**omomorphic **E**ncryption **Co**mpiler

Alexander Viand, Patrick Jattke, Miro Haller, Anwar Hithnawi

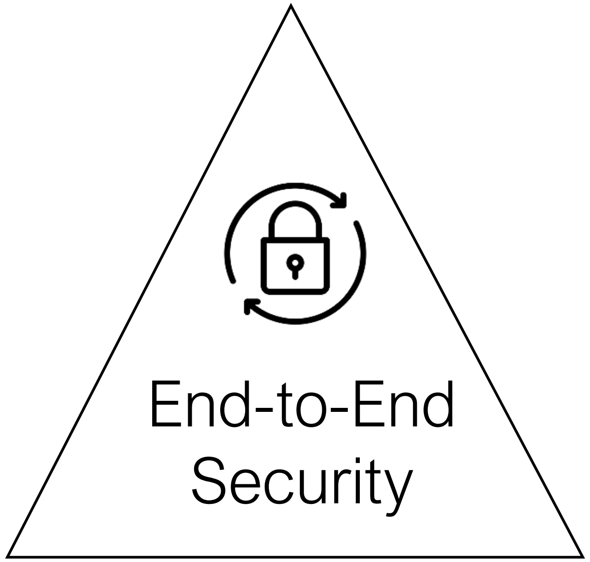
ETH zürich

Modern Cryptography

Data at Rest
Secure Storage

Conventional Crypto

Encryption & Digital Signatures



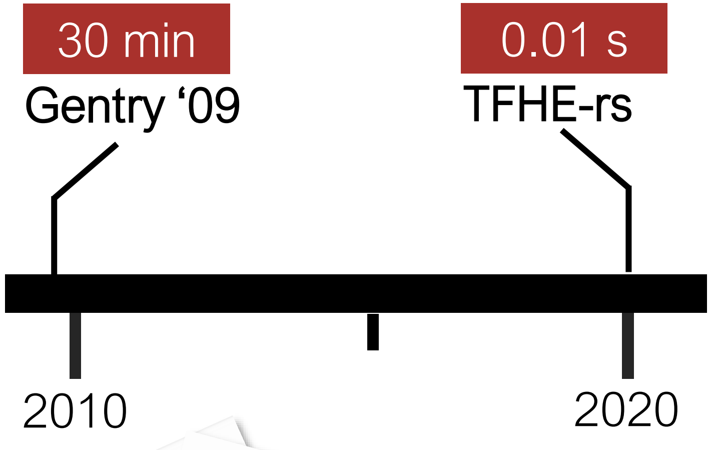
Advanced Crypto

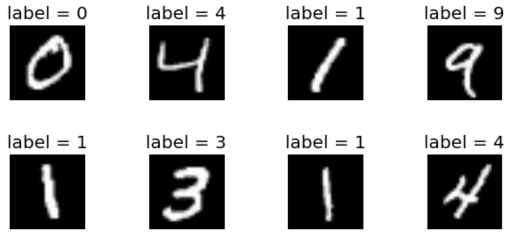
- Secure Multi-Party Computation
- Zero Knowledge Proofs
- Fully Homomorphic Encryption

Data in Transit
Secure Communication

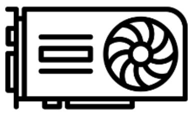
Data in Use
Secure Computation

FHE in Practice



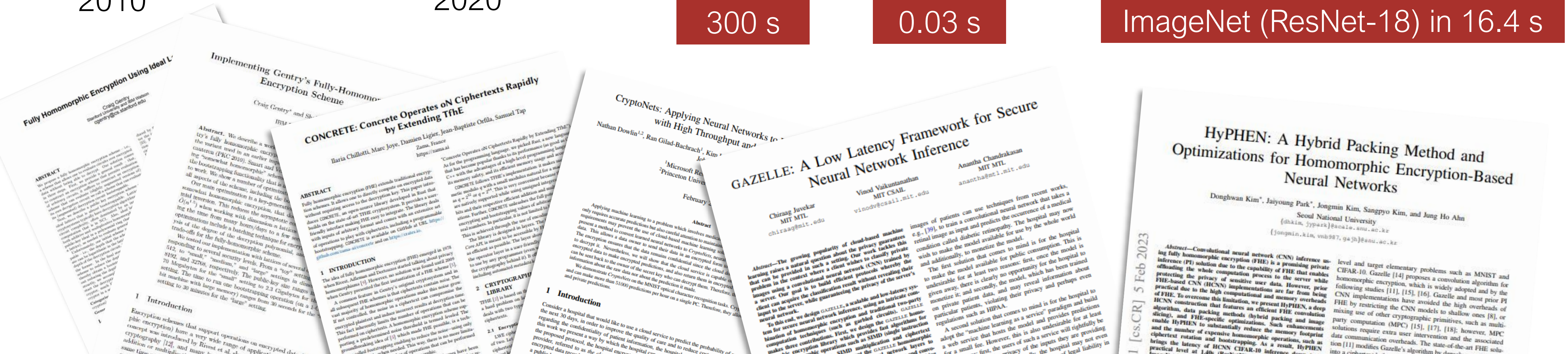


300 s



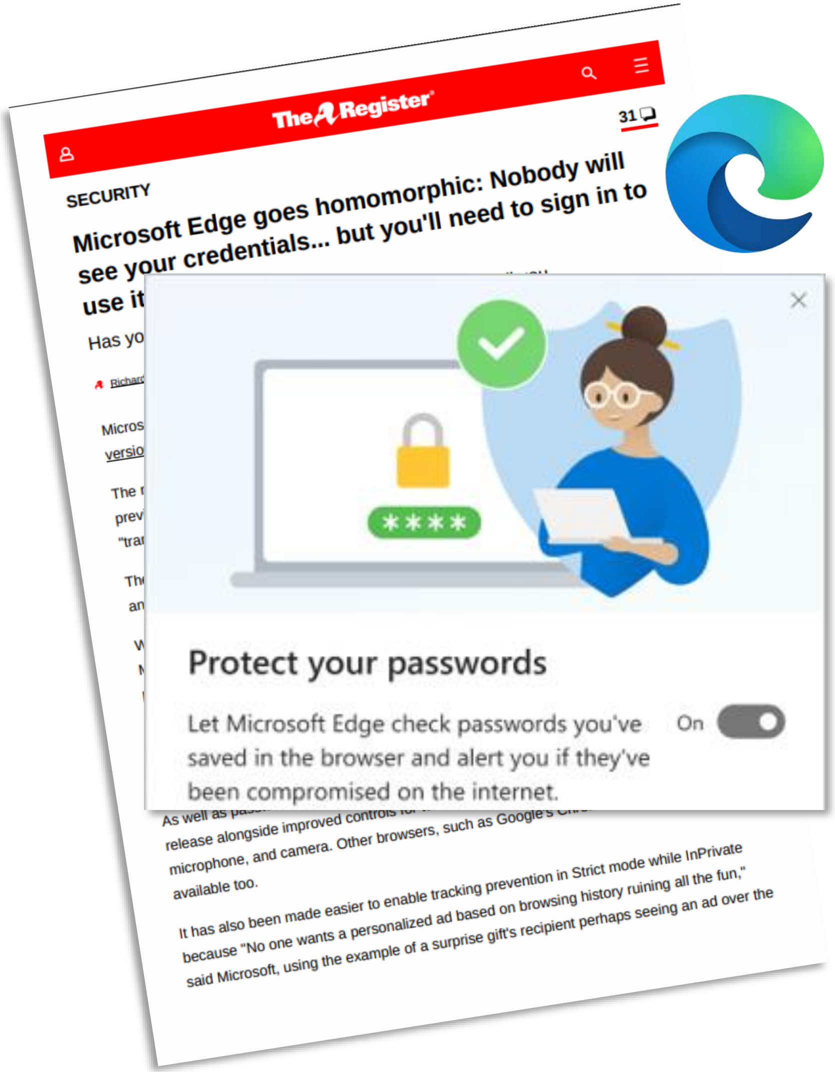
0.03 s

ImageNet (ResNet-18) in 16.4 s



FHE Deployment

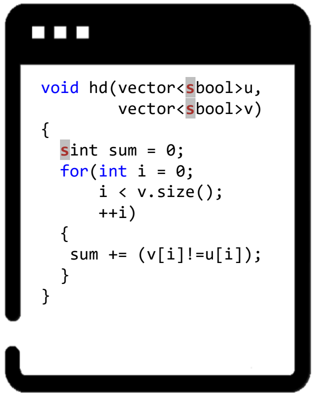
FHE Deployment



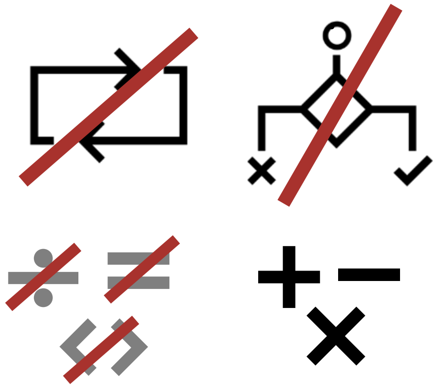
Developing and Deploying FHE Applications is

Notoriously Hard

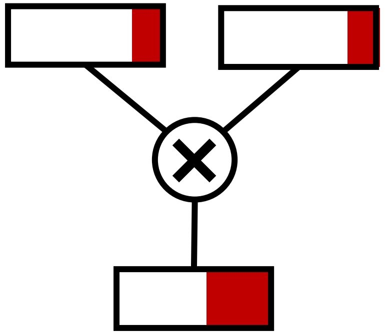
FHE Programming Paradigm



Application Dependent



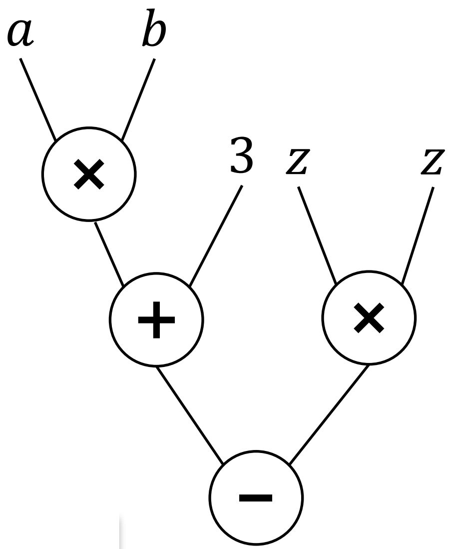
Limited Expressiveness



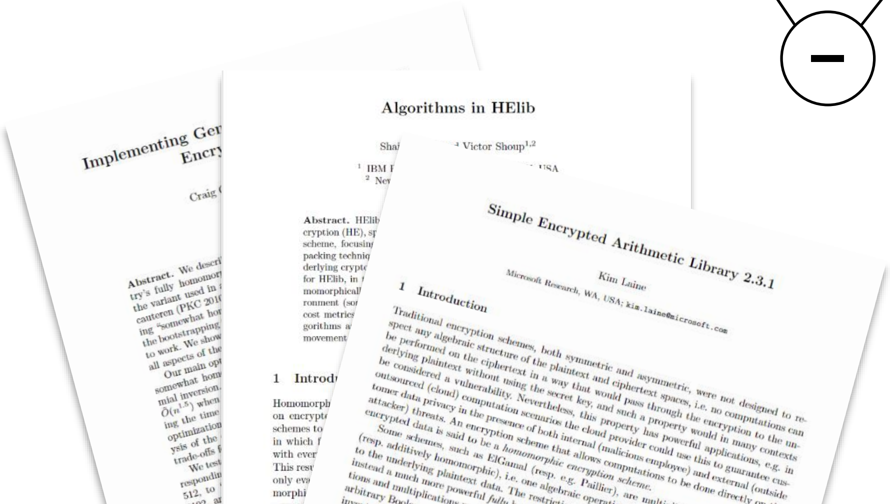
Noise Management

SIMD Batching

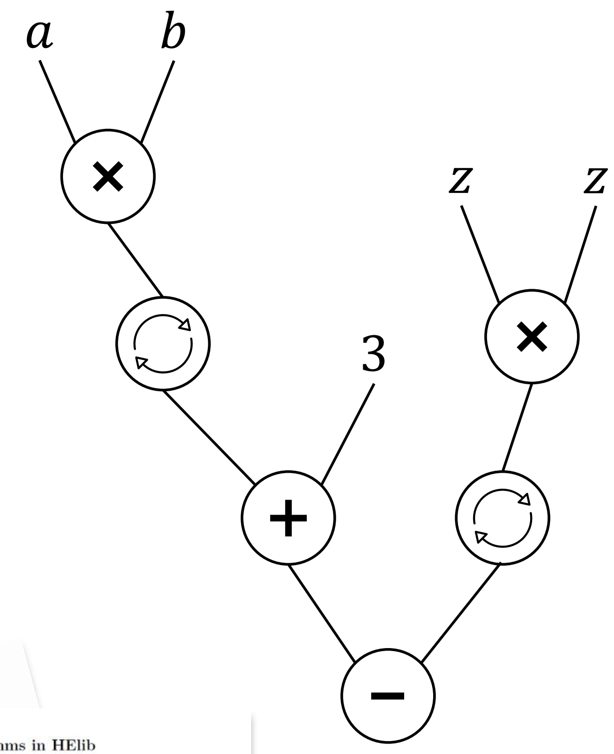
Evolution of FHE Development

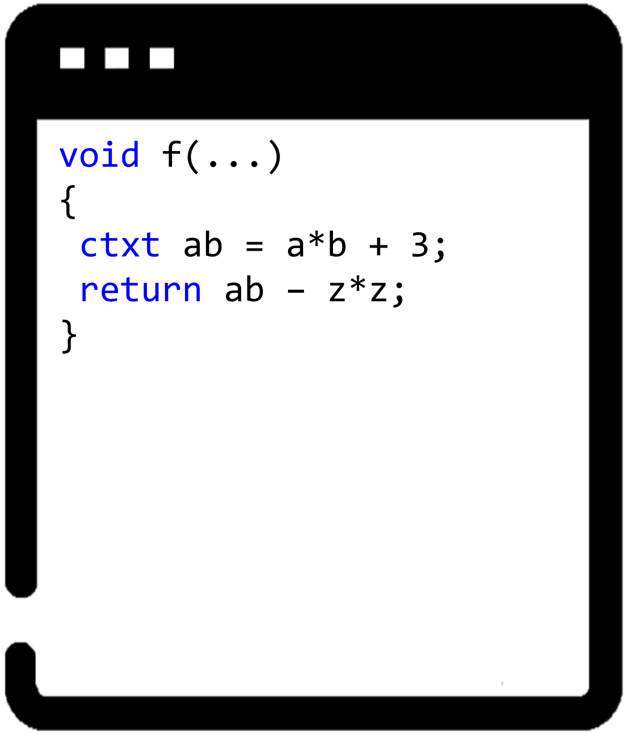


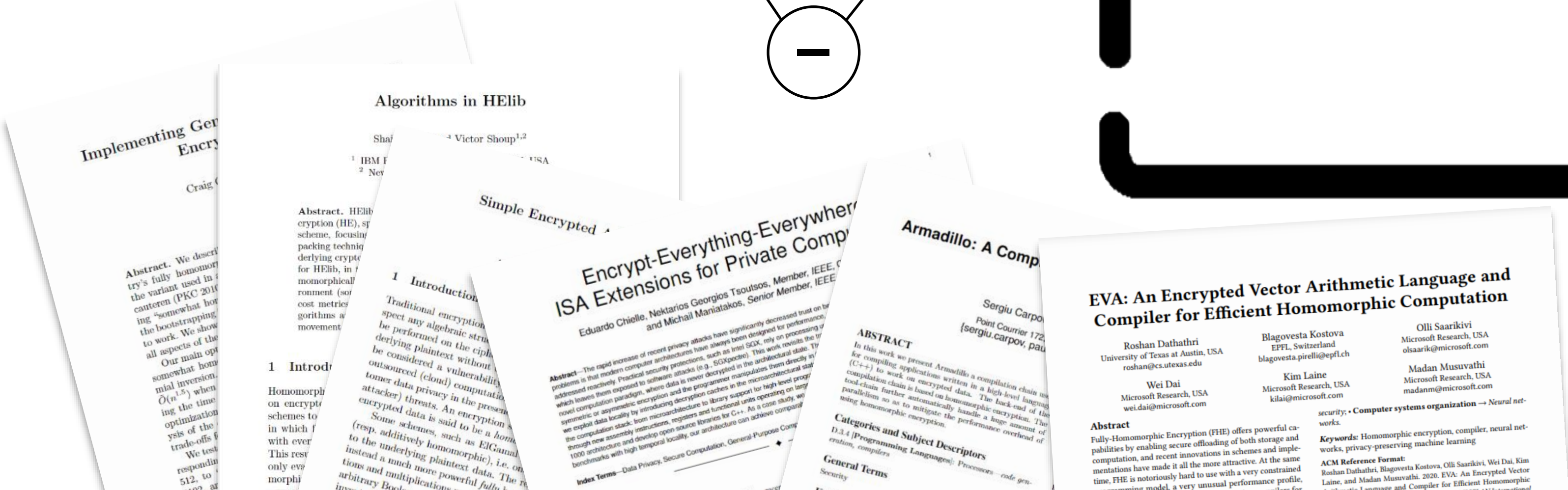

```
void f(...)
{
 mul_inp(a,b);
 relin_inp(a);
 add_plain_inp(a,3)
 square_inp(z,z);
 relin_inp(a);
 sub_inp(a,z);
 return a;
}
```



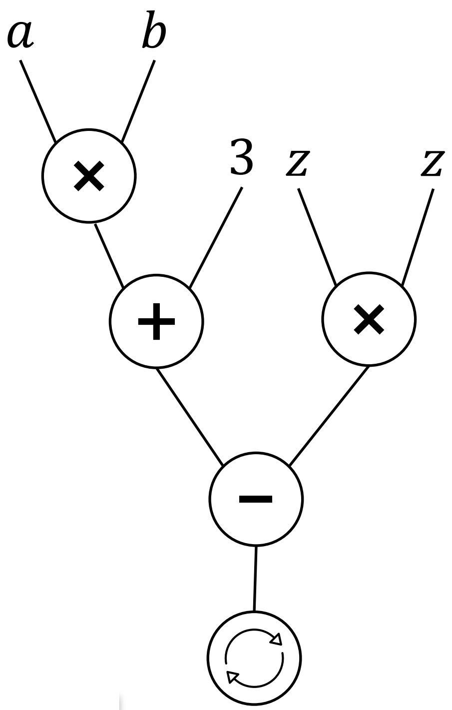
Evolution of FHE Development



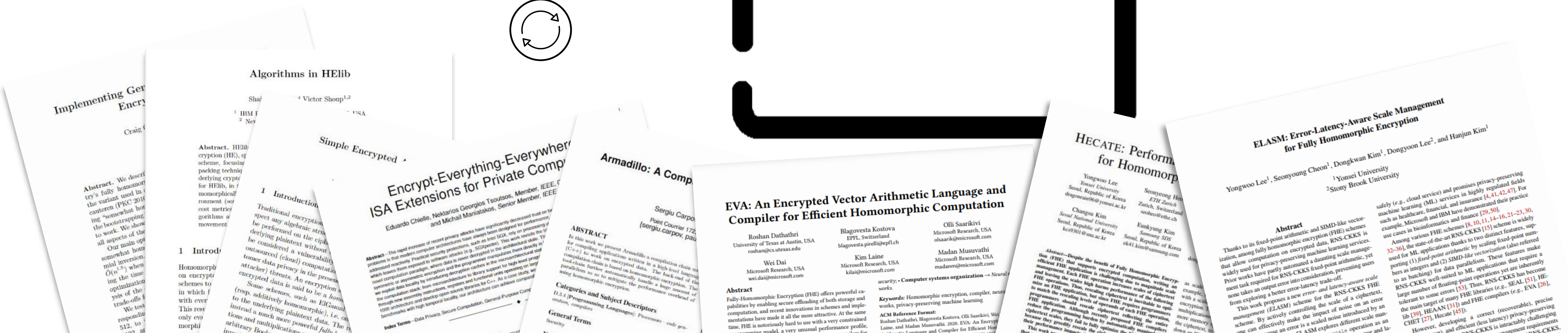




Evolution of FHE Development




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

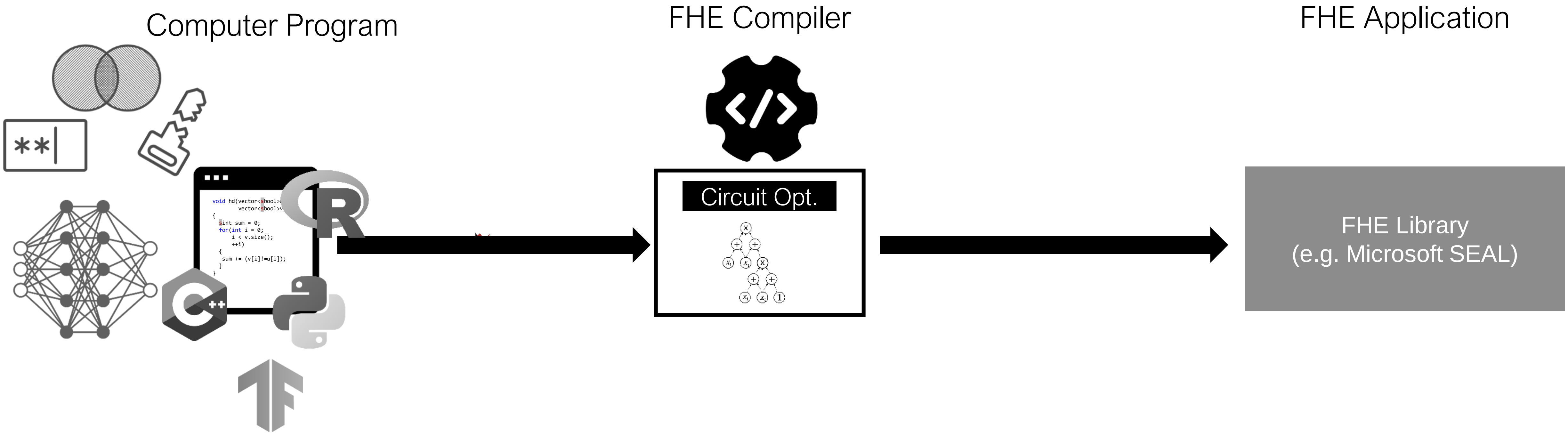


Contributions

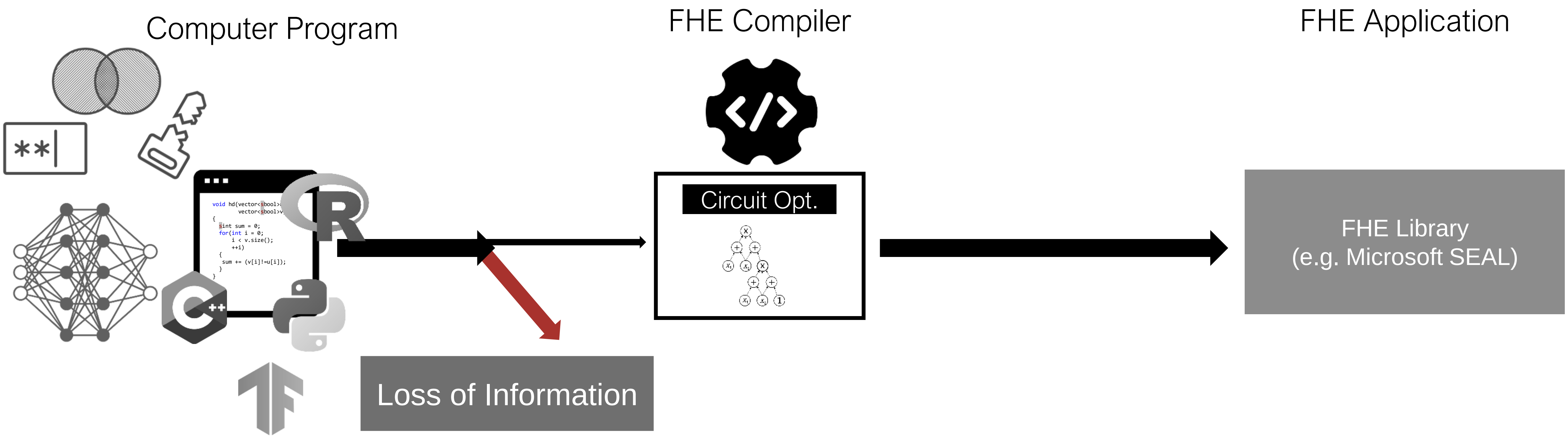
Architecture	Address FHE development as an end-to-end problem
Optimizations	Order-of-magnitude speedups via high-level transformations

End-to-End FHE Toolchain Architecture

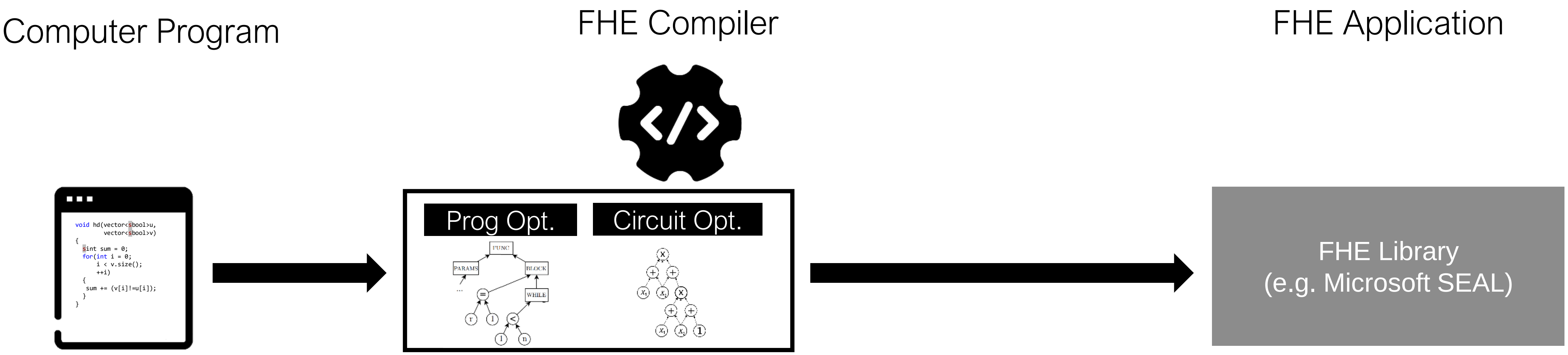
End-to-End FHE Toolchain



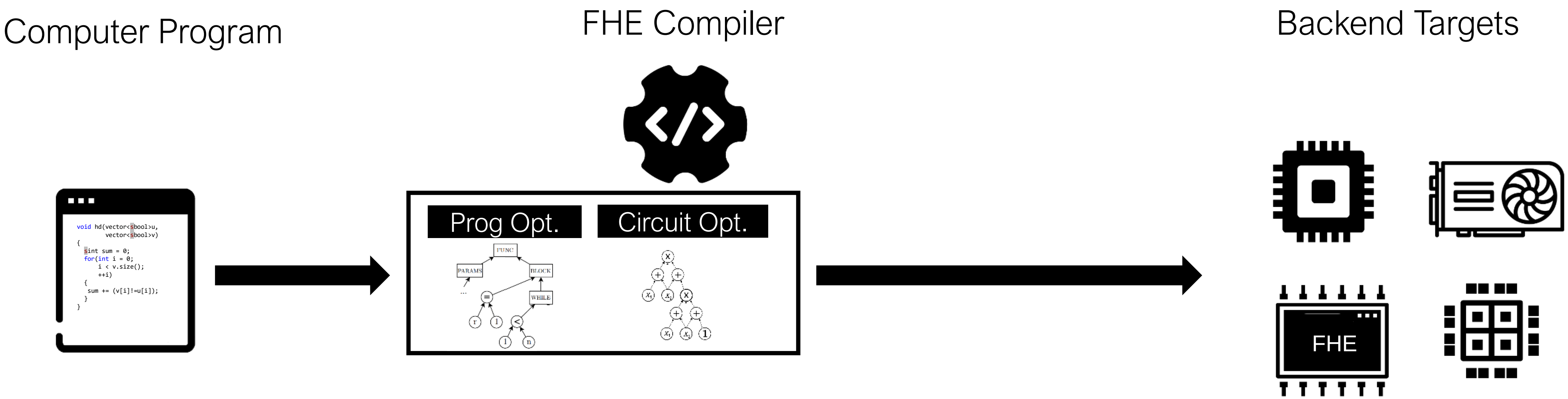
End-to-End FHE Toolchain



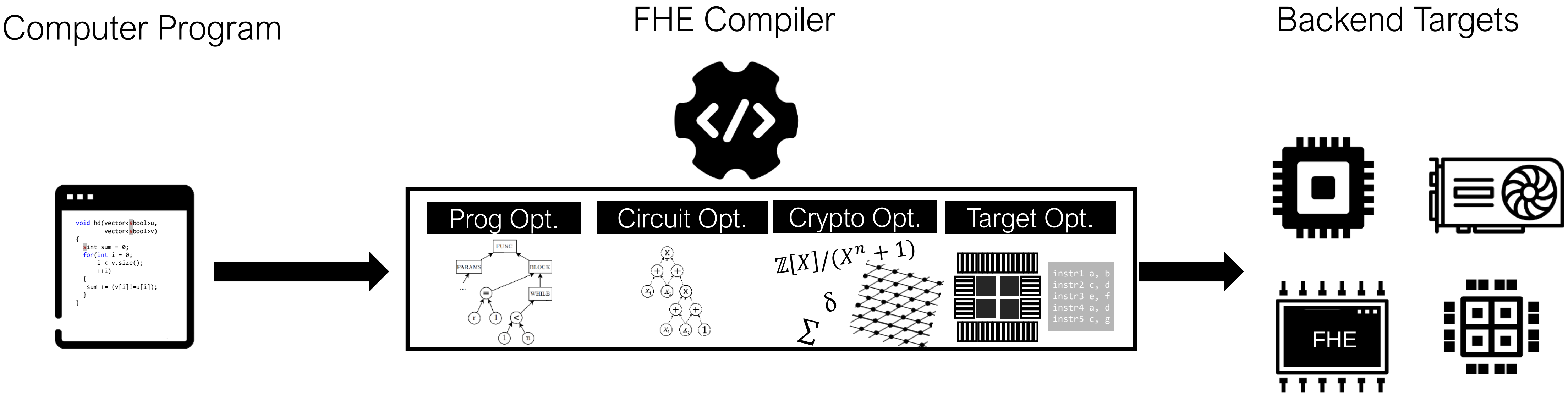
End-to-End FHE Toolchain



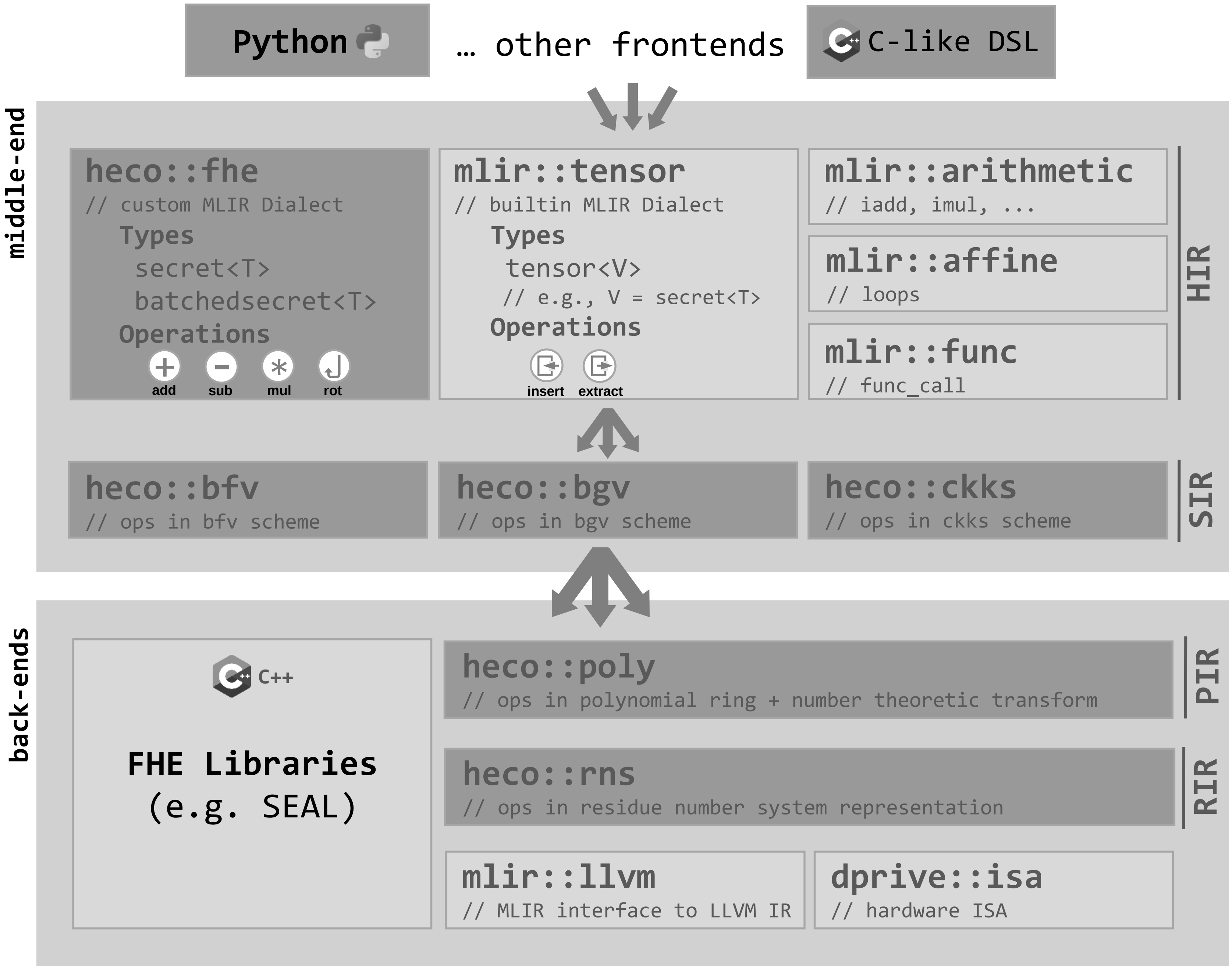
End-to-End FHE Toolchain



End-to-End FHE Toolchain



HECO Architecture



SIMD Batching Optimization

SIMD-like Parallelism

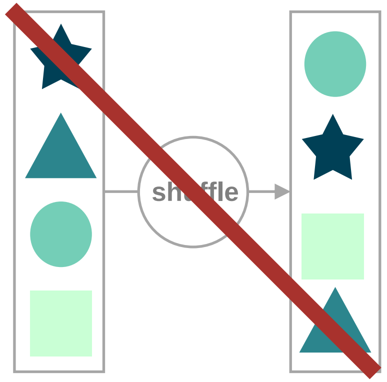
Standard C++

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

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

}
```

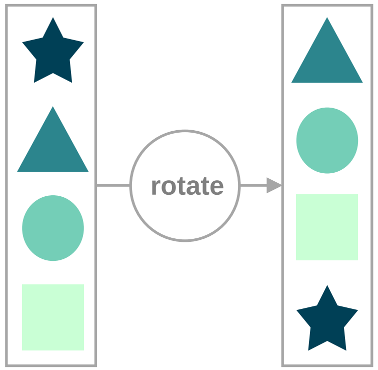
Batched FHE

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

   return a * b;

}
```

SIMD Batching



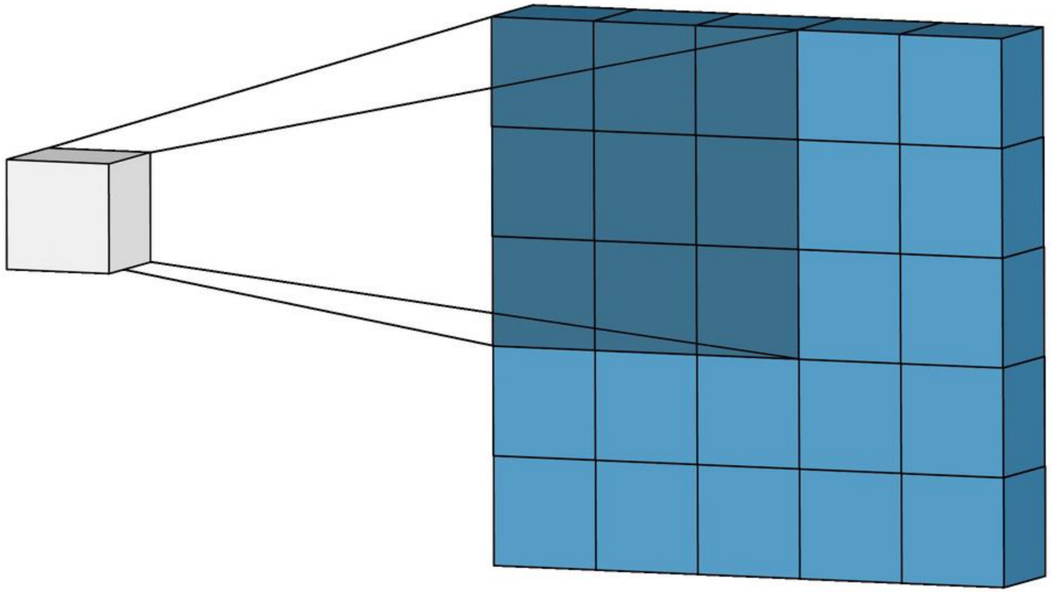
No efficient free permutation or scatter/gather



Only cyclical rotations

Example: Simple Image Processing

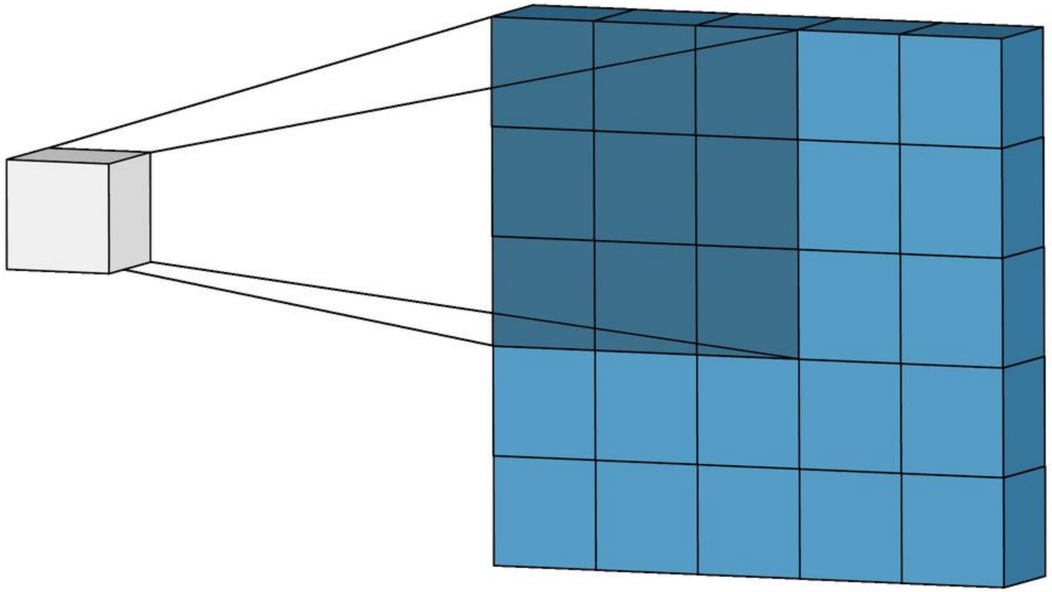
```
@template(N)
def LaplaceSharpening(img: Tensor[N, Secret[f64]]):
 img_out = img.copy()
 w = [[1, 1, 1], [1, -8, 1], [1, 1, 1]]
 for x in range(N): # loop over pixels
  for y in range(N): t = 0
   for j in range(-1, 2): # apply kernel
    for i in range(-1, 2):
     t += w[i+1][j+1] * img[((x+i)*N+(y+j))%N]
     img_out[(x*N+y)%N] = 2*img[(x*N+y)%N] - t
 return img_out
```



$9N^2$ Homomorphic Multiplications

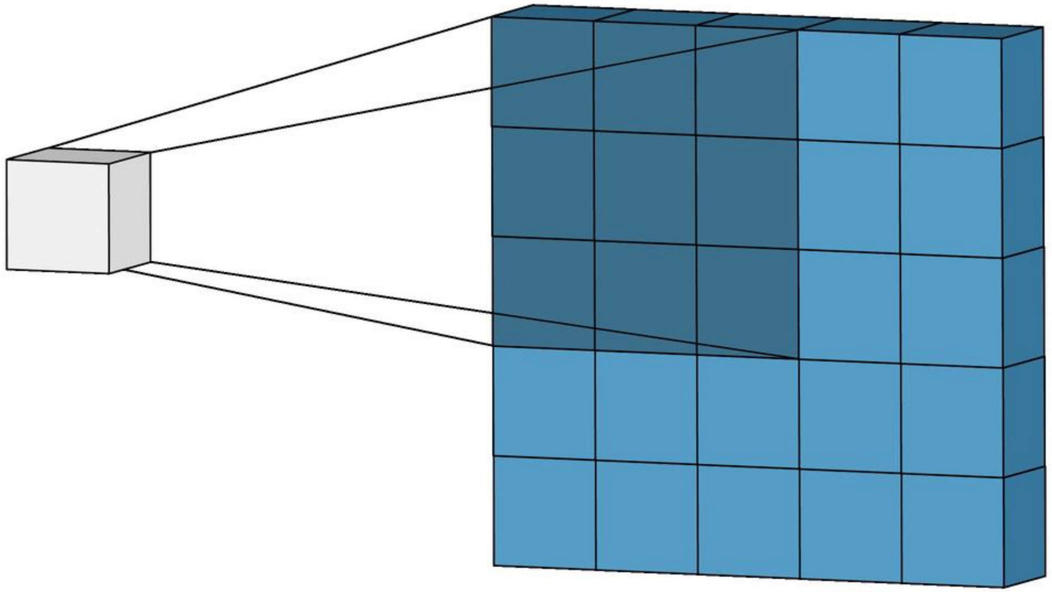
Example: Simple Image Processing

```
@template(N)
def LaplaceSharpening(img: Tensor[N, Secret[f64]]):
 img_out = img.copy()
 w = [[1, 1, 1], [1, -8, 1], [1, 1, 1]]
 for x in range(N): # loop over pixels
  for y in range(N): t = 0
    t += w[0][0] * img[((x-1)*N+(y-1))%N]
    t += w[1][0] * img[(x*N+(y-1))%N]
    t += w[2][0] * img[((x+1)*N+(y-1))%N]
    t += w[0][1] * img[((x-1)*N+y)%N]
    t += w[1][1] * img[(x*N+y)%N]
    t += w[2][1] * img[((x+1)*N+y)%N]
    t += w[0][2] * img[((x-1)*N+(y+1))%N]
    t += w[1][2] * img[(x*N+(y+1))%N]
    t += w[2][2] * img[((x+1)*N+(y+1))%N]
    img_out[(x*N+y)%N] = 2*img[(x*N+y)%N] - t
 return img_out
```



Example: Simple Image Processing

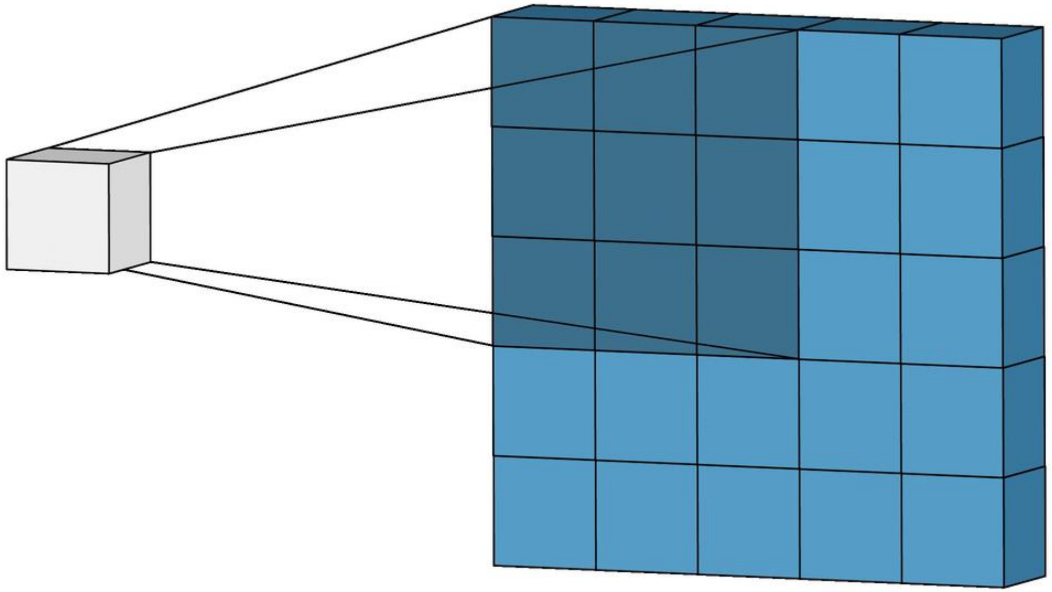
```
@template(N)
def LaplaceSharpening(img: Tensor[N, Secret[f64]]):
 img_out = img.copy()
 w = [[1, 1, 1], [1, -8, 1], [1, 1, 1]]
 for x in range(N): # loop over pixels
  for y in range(N): t = 0
    t += w * temp
    img_out[(x*N+y)%N] = 2*img[(x*N+y)%N] - t
 return img_out
```



N^2 Batched Homomorphic Multiplications?

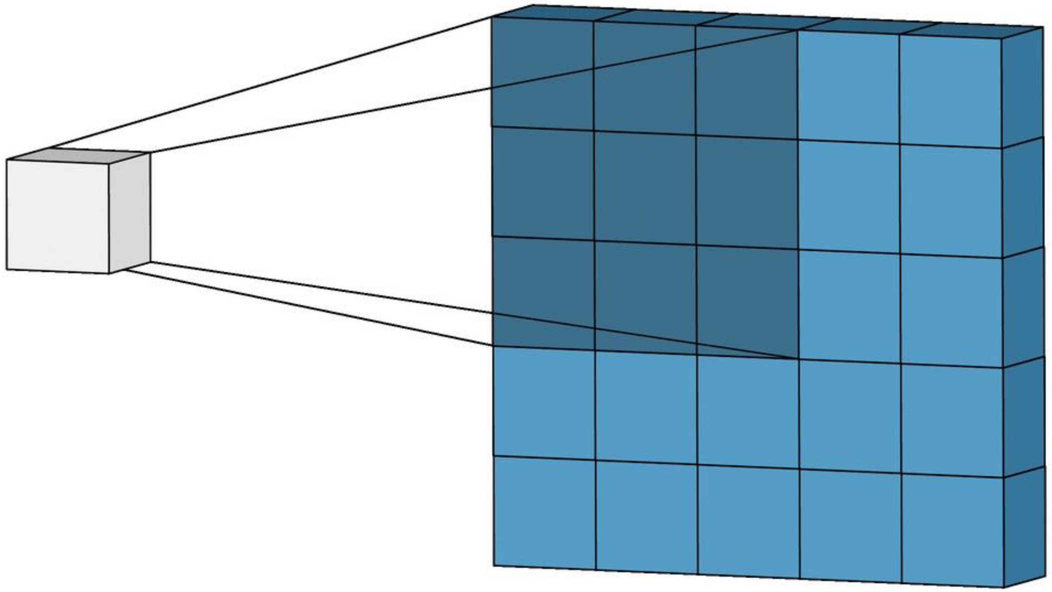
Example: Simple Image Processing

```
@template(N)
def LaplaceSharpening(img: Tensor[N, Secret[f64]]):
 img_out = img.copy()
 w = [[1, 1, 1], [1, -8, 1], [1, 1, 1]]
 for x in range(N): # loop over pixels
  for y in range(N): t = 0








    t += w * temp
    img_out[(x*N+y)%N] = 2*img[(x*N+y)%N] - t
 return img_out
```



Example: Simple Image Processing

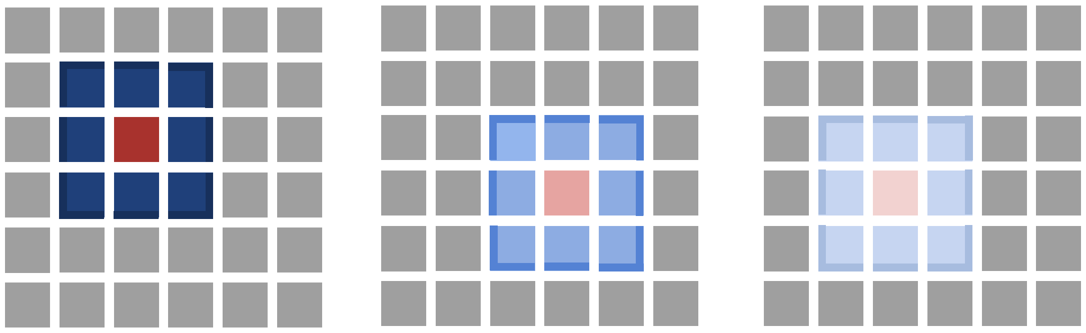
```
@template(N)
def LaplaceSharpening(img: Tensor[N, Secret[f64]]):
 img_out = img.copy()
 w = [[1, 1, 1], [1, -8, 1], [1, 1, 1]]
 for x in range(N): # loop over pixels
  for y in range(N): t = 0
    temp[0] = img[((x-1)*N+(y-1))%N]
    temp[1] = img[(x*N+(y-1))%N]
    temp[2] = img[((x+1)*N+(y-1))%N]
    temp[3] = img[((x-1)*N+y)%N]
    temp[4] = img[(x*N+y)%N]
    temp[5] = img[((x+1)*N+y)%N]
    temp[6] = img[((x-1)*N+(y+1))%N]
    temp[7] = img[(x*N+(y+1))%N]
    temp[8] = img[((x+1)*N+(y+1))%N]
    t += w * temp
    img_out[(x*N+y)%N] = 2*img[(x*N+y)%N] - t
 return img_out
```

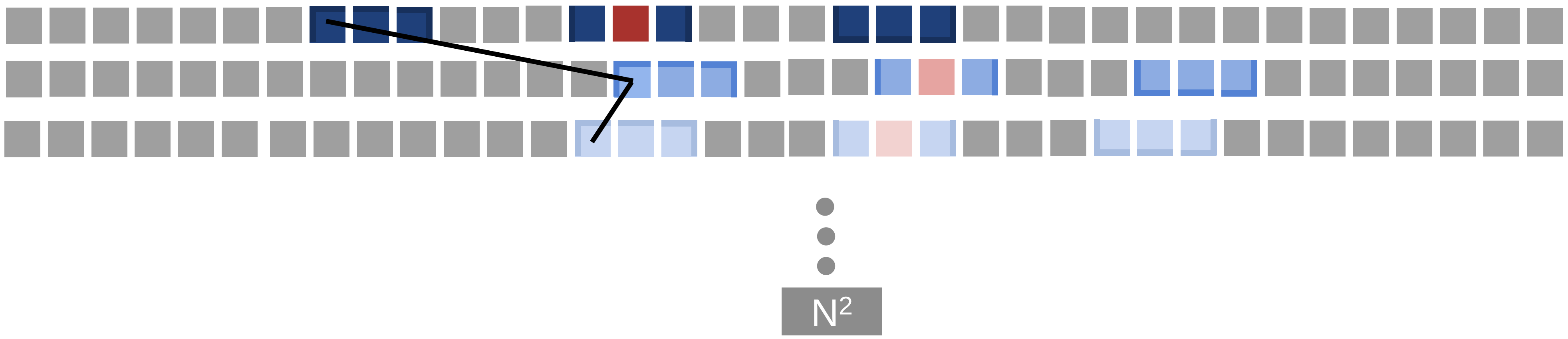


N^2 Homomorphic Multiplications + $9N^2$ Homomorphic Rotations

Example: Simple Image Processing

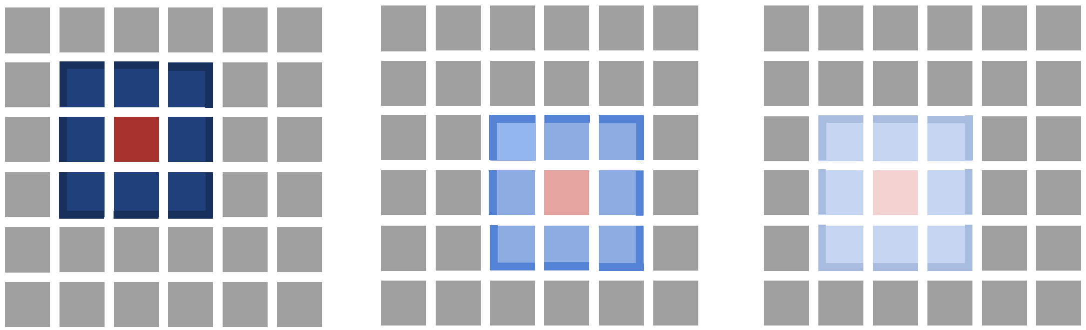
```
@template(N)
def LaplaceSharpening(img: Tensor[N, Secret[f64]]):
 img_out = img.copy()
 w = [[1, 1, 1], [1, -8, 1], [1, 1, 1]]
 for x in range(N): # loop over pixels
  for y in range(N): t = 0
   for j in range(-1, 2): # apply kernel
    for i in range(-1, 2):
     t += w[i+1][j+1] * img[((x+i)*n+(y+j))%N]
     img_out[(x*n+y)%N] = 2*img[(x*n+y)%N] - t
 return img_out
```





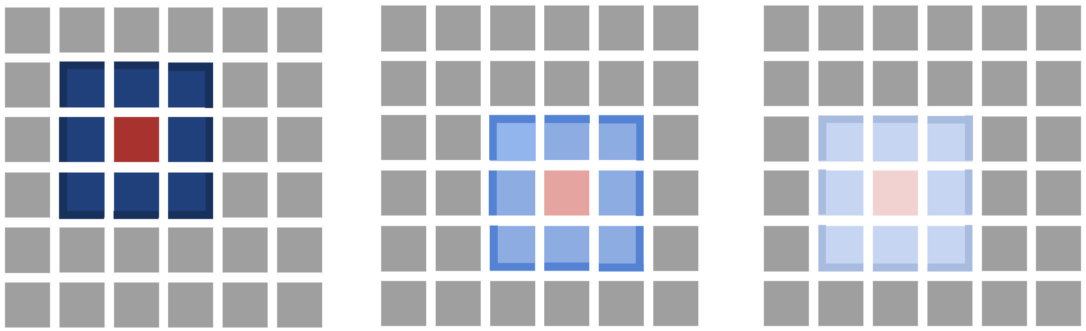
Example: Simple Image Processing

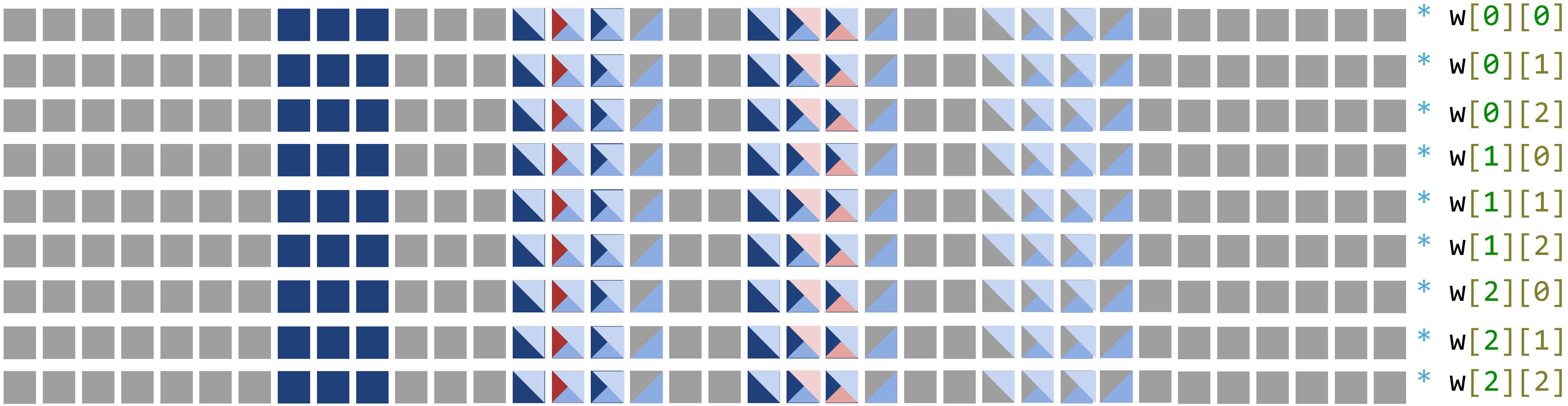
```python
@template(N)
def LaplaceSharpening(img: Tensor[N, Secret[f64]]):
 img_out = img.copy()
 w = [[1, 1, 1], [1, -8, 1], [1, 1, 1]]
 for x in range(N): # loop over pixels
  for y in range(N): t = 0
   for j in range(-1, 2): # apply kernel
    for i in range(-1, 2):
     t += w[i+1][j+1] * img[((x+i)*n+(y+j))%N]
     img_out[(x*n+y)%N] = 2*img[(x*n+y)%N] - t
 return img_out
```



Example: Simple Image Processing

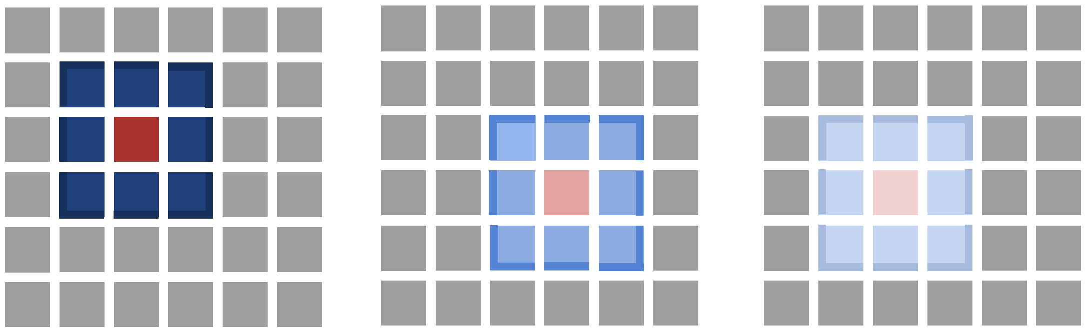
```
@template(N)
def LaplaceSharpening(img: Tensor[N, Secret[f64]]):
 img_out = img.copy()
 w = [[1, 1, 1], [1, -8, 1], [1, 1, 1]]
 for x in range(N): # loop over pixels
  for y in range(N): t = 0
   for j in range(-1, 2): # apply kernel
    for i in range(-1, 2):
     t += w[i+1][j+1] * img[((x+i)*n+(y+j))%N]
     img_out[(x*n+y)%N] = 2*img[(x*n+y)%N] - t
 return img_out
```

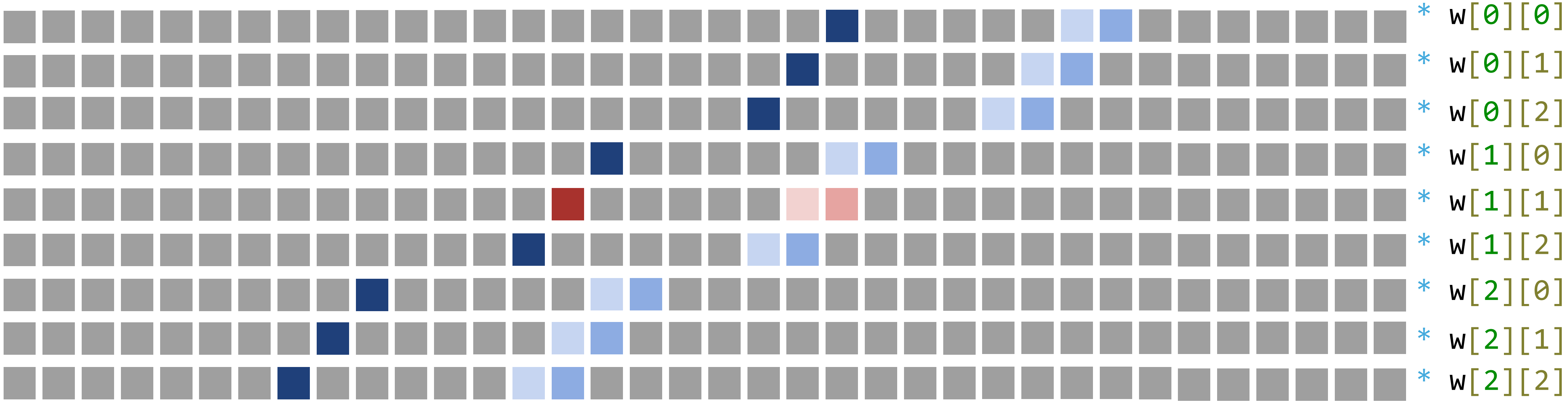




Example: Simple Image Processing

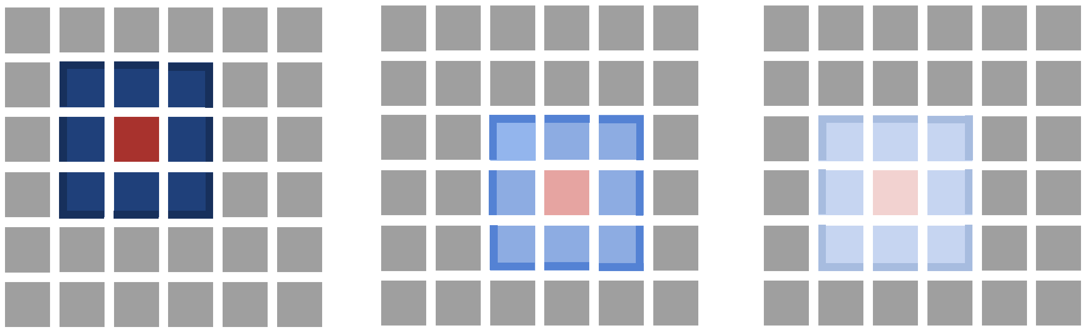
```
@template(N)
def LaplaceSharpening(img: Tensor[N, Secret[f64]]):
 img_out = img.copy()
 w = [[1, 1, 1], [1, -8, 1], [1, 1, 1]]
 for x in range(N): # loop over pixels
  for y in range(N): t = 0
   for j in range(-1, 2): # apply kernel
    for i in range(-1, 2):
     t += w[i+1][j+1] * img[((x+i)*n+(y+j))%N]
     img_out[(x*n+y)%N] = 2*img[(x*n+y)%N] - t
 return img_out
```

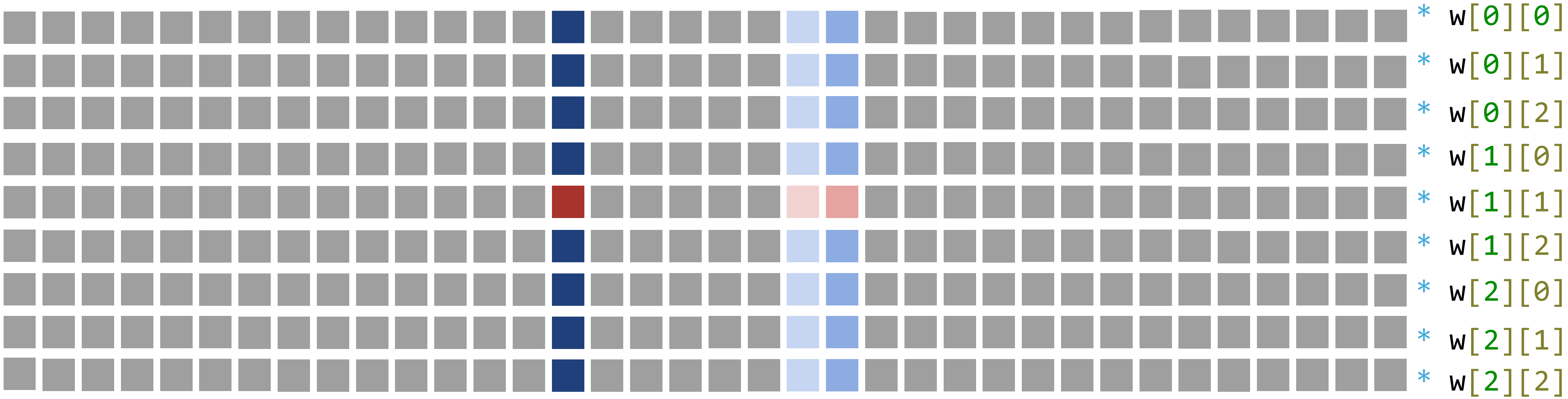




Example: Simple Image Processing

```
@template(N)
def LaplaceSharpening(img: Tensor[N, Secret[f64]]):
 img_out = img.copy()
 w = [[1, 1, 1], [1, -8, 1], [1, 1, 1]]
 for x in range(N): # loop over pixels
  for y in range(N): t = 0
   for j in range(-1, 2): # apply kernel
    for i in range(-1, 2):
     t += w[i+1][j+1] * img[((x+i)*n+(y+j))%N]
     img_out[(x*n+y)%N] = 2*img[(x*n+y)%N] - t
 return img_out
```





Generalizing Batching

Algorithm 3 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]:
```

Generalizing Batching

```
        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 ⊥
procedure OperandConversion(op, ts, V, E)
    foreach v ∈ V ∧ (v, op) ∈ E ∧ type(v) = fhe.secr
        switch v:
            case fhe.extract(x, i):
                u ← fhe.rotate(x, i − ts)
                Replace(v, u, V, E)
            case fhe.ptxt[p]:
                p' ← Repeat(p)
```

Generalizing Batching

procedure OPERANDCONVERSION$(op, rs, \mathcal{V}, \mathcal{E})$

  foreach $v \in \mathcal{V} \land (v, op) \in \mathcal{E} \land \text{type}(v) \equiv$ fhe.secr

    switch v:

      case fhe.extract(x, i):

        $u \leftarrow$ fhe.rotate$(x, i - rs)$

        REPLACE$(v, u, \mathcal{V}, \mathcal{E})$

      case fhe.ptxt$[p]$:

        $p' \leftarrow$ REPEAT(p)

        $u \leftarrow$ fhe.ptxt(p')

        REPLACE$(v, u, \mathcal{V}, \mathcal{E})$

procedure REPLACE$(v, u, \mathcal{V}, \mathcal{E})$

  $\mathcal{V} \leftarrow (\mathcal{V} \setminus \{v\}) \bigcup \{u\}$

  foreach $w \in \mathcal{V} \land (v, w) \in \mathcal{E}$:

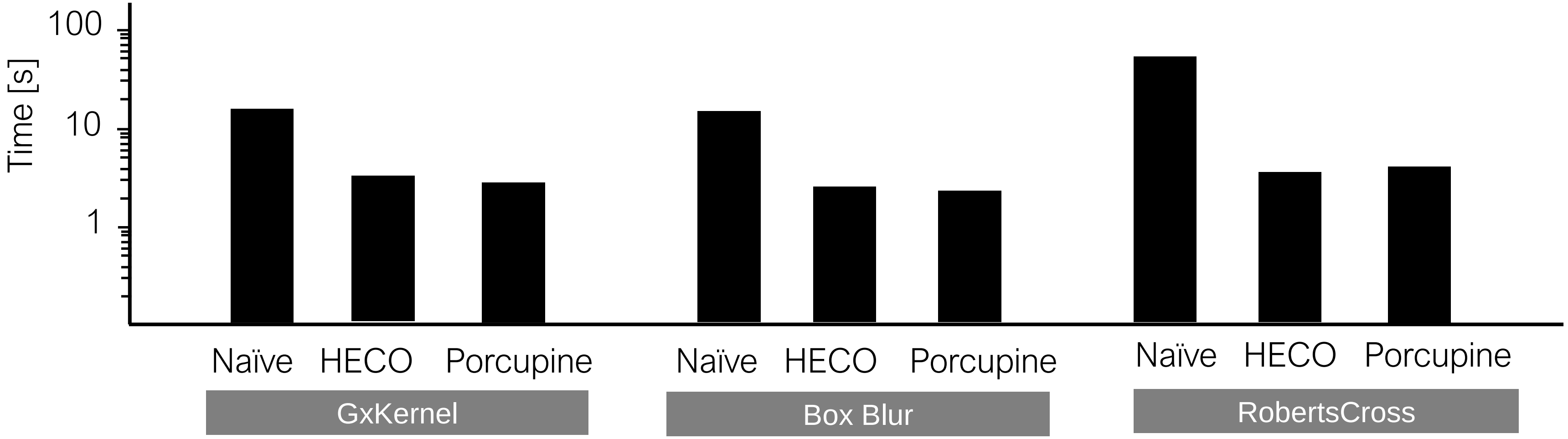
    $\mathcal{E} \leftarrow (\mathcal{E} \setminus \{(v, w)\}) \bigcup \{(u, w)\}$

  foreach $w \in \mathcal{V} \land (w, v) \in \mathcal{E}$:

    $\mathcal{E} \leftarrow (\mathcal{E} \setminus \{(w, v)\}) \bigcup \{(w, u)\}$

Evaluation: Effect of Batching

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.

ETH zürich

PPS

This work:

- End-to-End FHE Toolchain Architecture
- High-Level Batching Optimization

Ongoing FHE Abstraction Standardization Effort:

intel, Google, ZAMA, Duality, HEAAN CRYPTO LAB

marbleHE/HECO

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE | ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL | ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

arxiv.org/abs/2202.01649


HECO: Fully Homomorphic Encryption Compiler

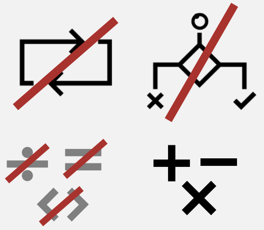
Alexander Viand, Patrick Jattke, Miro Haller, Anwar Hithnawi

ETH Zurich

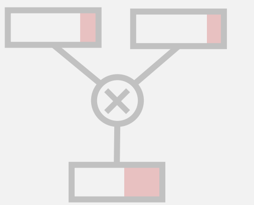
Abstract

1 Introduction

Future work:



Limited Expressiveness



Noise Management

SIMD Batching