Quentin Bourgerie
Samuel Tap

| 07.06.2023

Real life TFHE

Concrete Compiler to the rescue

ZAMA

Meet The Team

Quentin Bourgerie
Head of Concrete

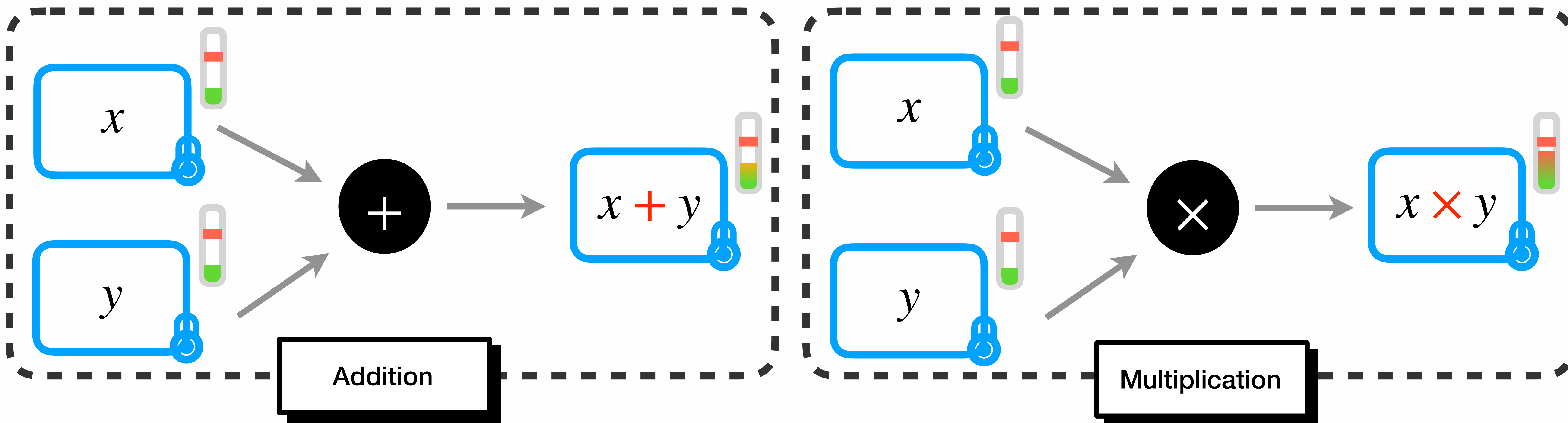
Samuel Tap
Scientific Advisor

Agenda

Introduction — 04
Concrete Python — 13
Concrete Compiler — 20
Concrete Optimizer — 27
Conclusion — 36

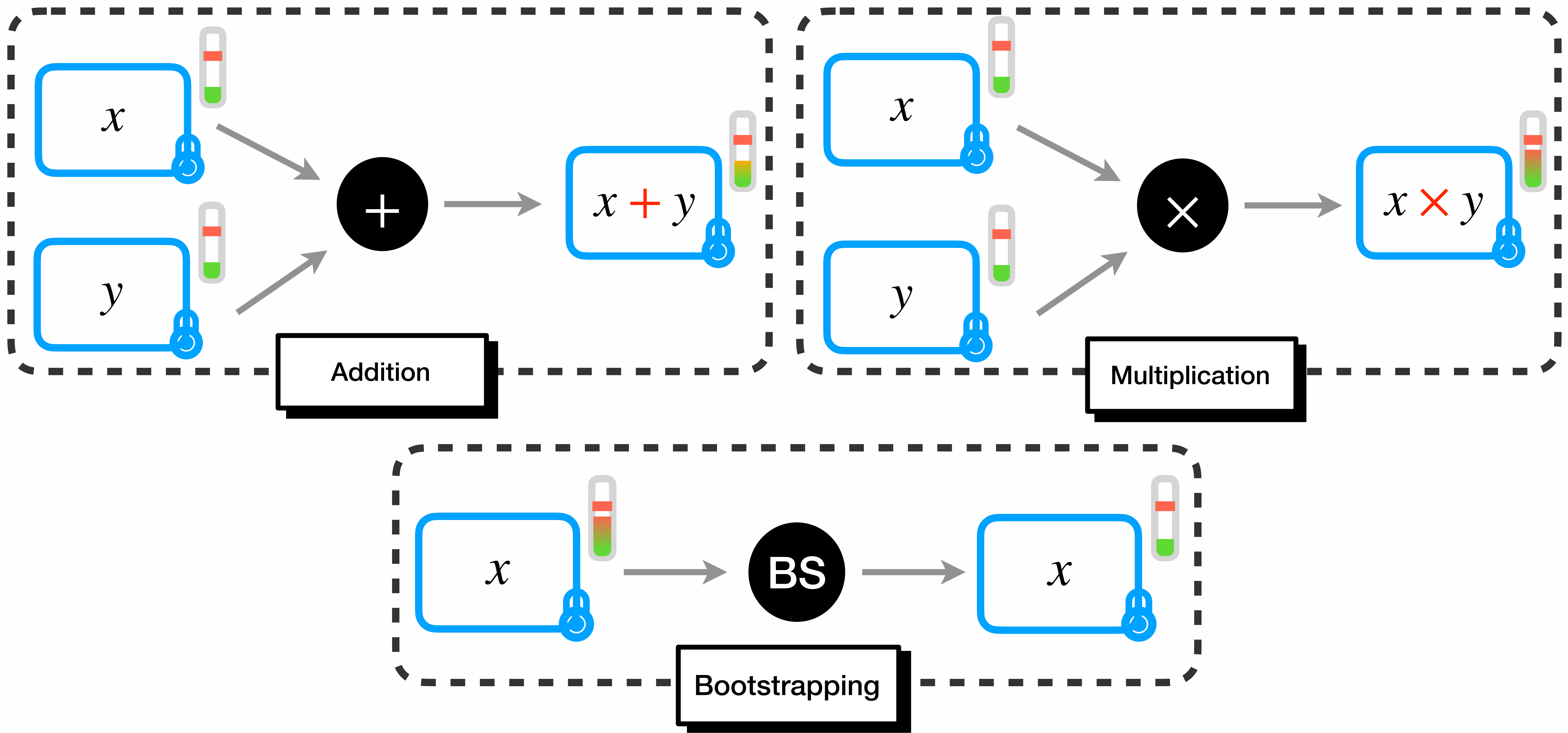
Introduction

FHE



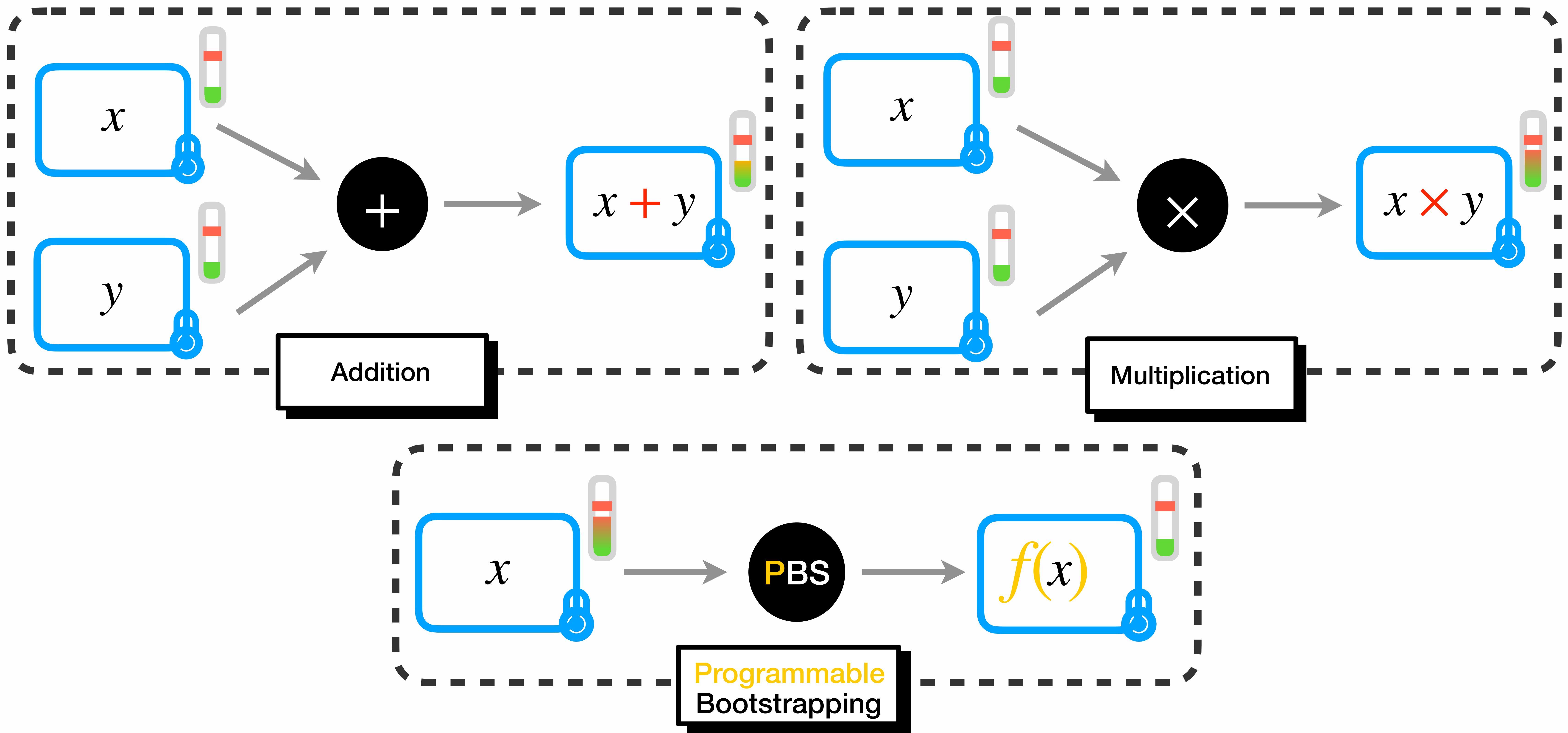
too much noise $\implies$ incorrect decryption 💀

FHE



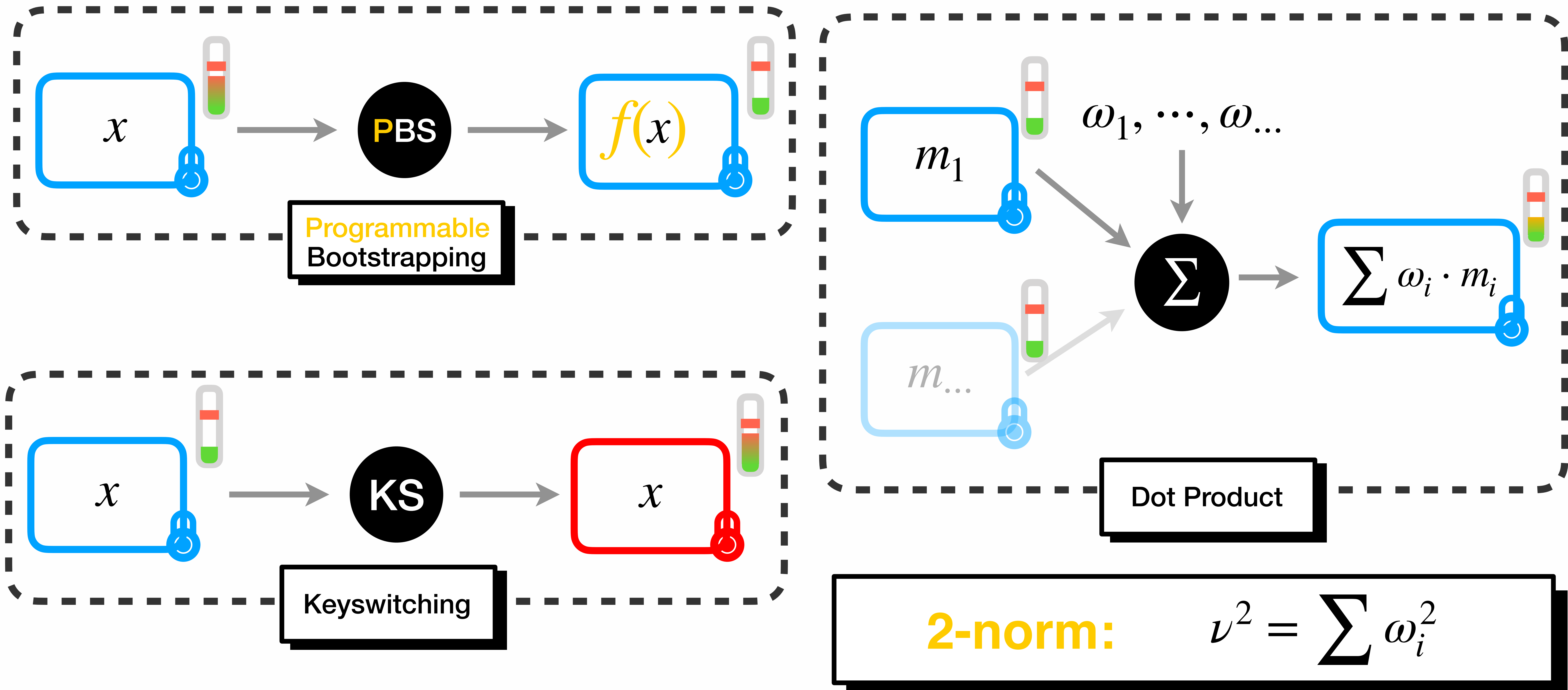
[CGGI20] I. Chillotti, N. Gama, M. Georgieva, M. Izabachène. TFHE: Fast Fully Homomorphic Encryption over the Torus. Journal of Cryptology 2020.

TFHE



[CGGI20] I. Chillotti, N. Gama, M. Georgieva, M. Izabachène. TFHE: Fast Fully Homomorphic Encryption over the Torus. Journal of Cryptology 2020.

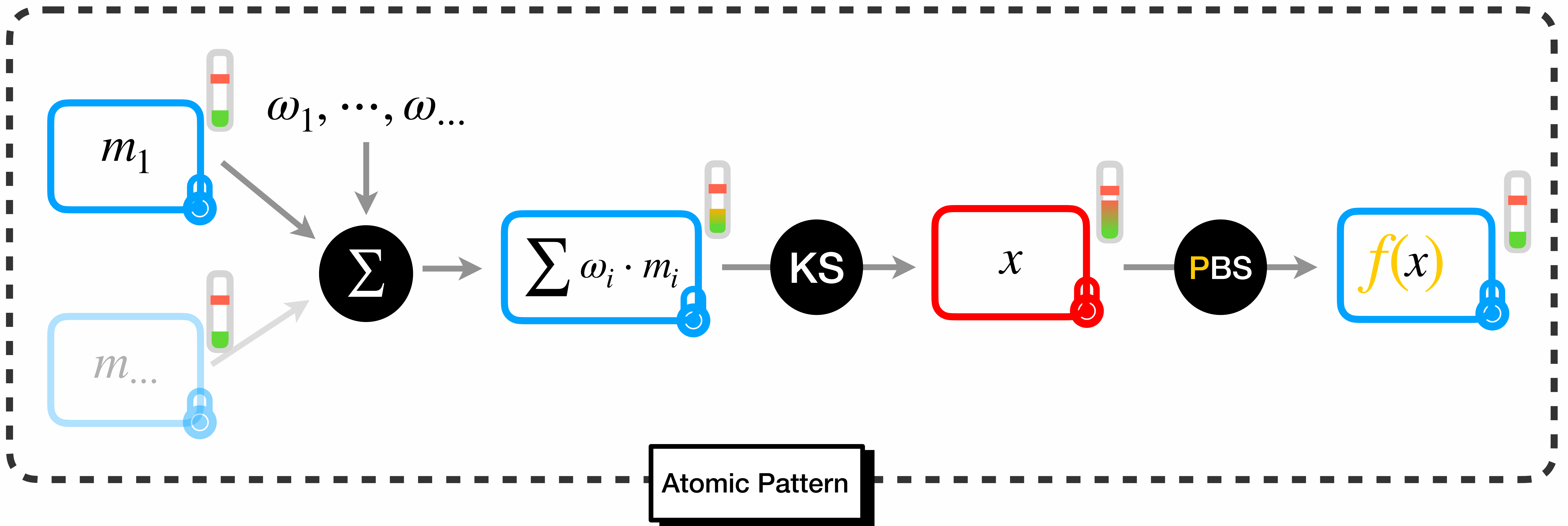
TFHE building blocks



2-norm: $\nu^2 = \sum \omega_i^2$

[CGGI20] I. Chillotti, N. Gama, M. Georgieva, M. Izabachène. TFHE: Fast Fully Homomorphic Encryption over the Torus. Journal of Cryptology 2020.

Optimal arrangement of TFHE building blocks



[CJP21] I. Chillotti, M. Joye, and P. Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In CSCML 2021
[BBB+22] L. Bergerat, A. Boudi, Q. Bourgerie, I. Chillotti, D. Ligier, J.-B. Orfila, S. Tap Parameter Optimization & Larger Precision for (T)FHE. Eprint

Enhancing TFHE

New paradigms

Short integers (1-10 bits) instead of boolean-only

One or several parameter set for a given DAG

Failure probability at DAG level

New algorithms

CRT/Radix encoding

WoP-PBS

Rounded-PBS

Approximate-PBS

Needs

Pick parameters

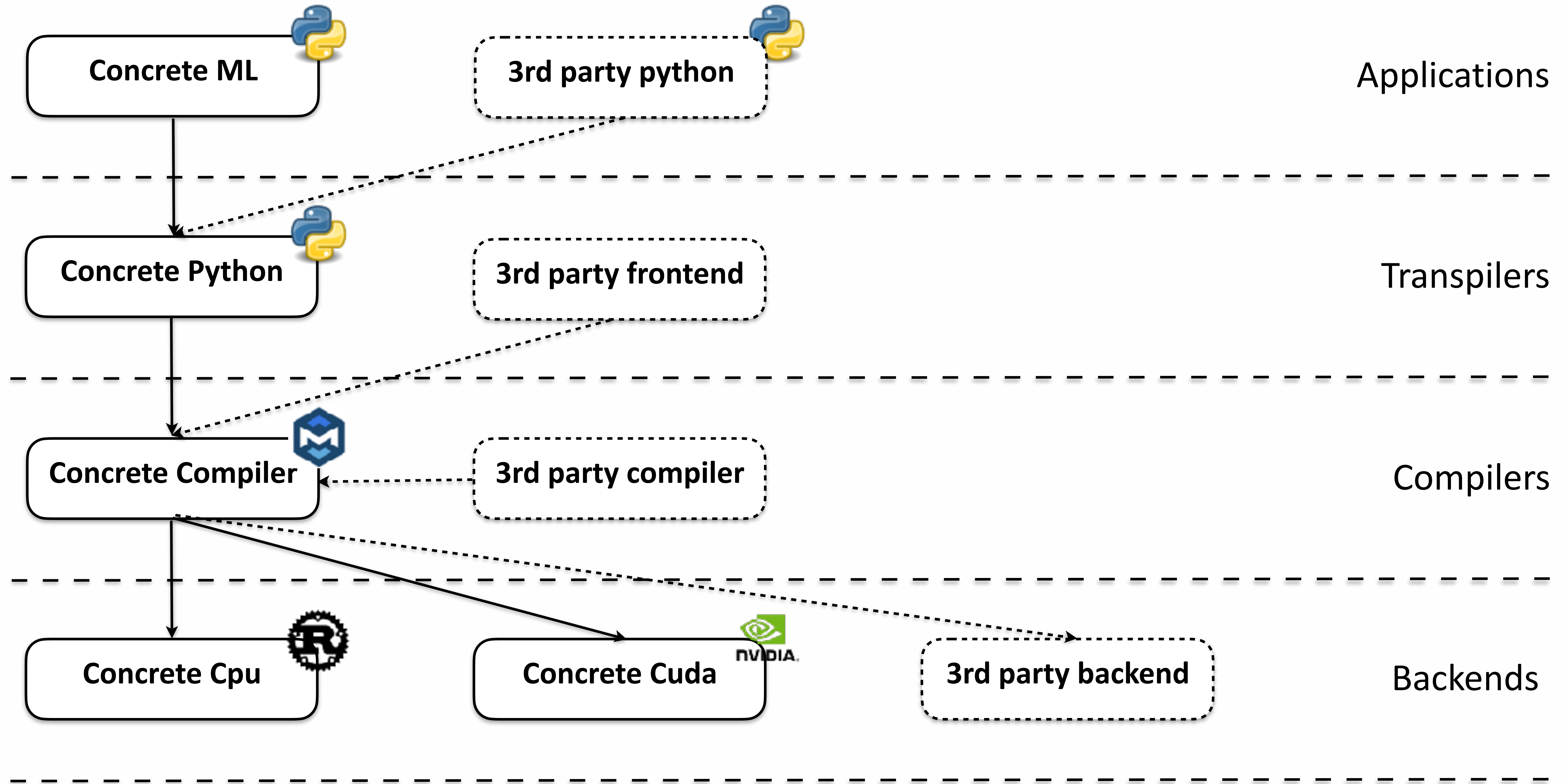
Optimal translation of a plain DAG into a TFHE DAG

Easy-to-use toolchain

Concrete

A modular framework for FHE applications

Concrete: a modular framework



Concrete Python

Transpiler for Concrete Compiler

Concrete Python

An easy to use frontend

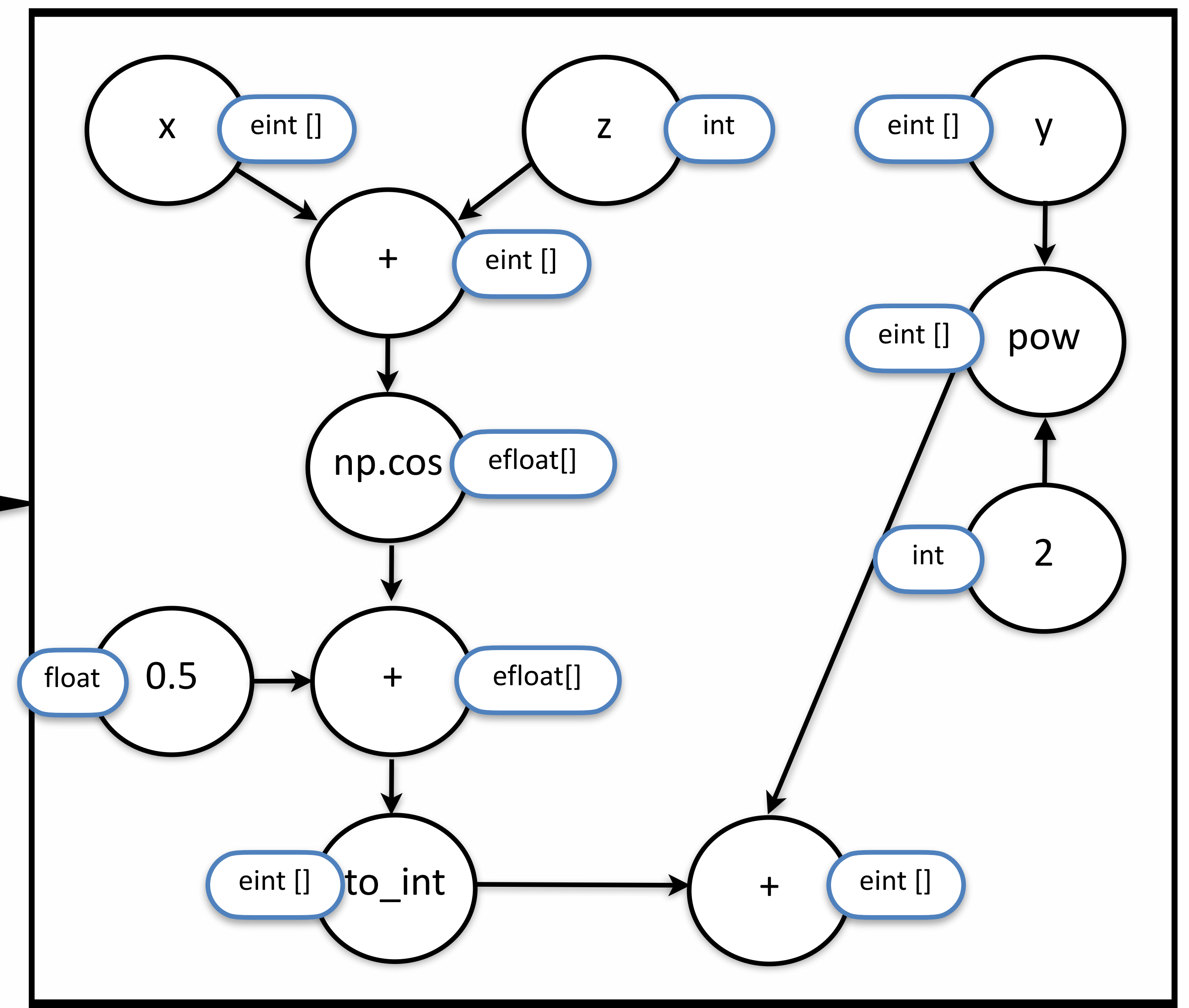
- Simple interface to **compile python programs**
- **Type inference** based on the dataset evaluation
- **Client and server API** to run FHE evaluation
- Seemless conversion of univariate **floating point and integer function** to table lookup
- **Extensive support of python** and numpy standard functions on scalar and tensor values

```
1   from concrete import fhe
2
3   import numpy as np
4
5   # Define your standard python function
6   @fhe.compiler({"x": "encrypted", "y": "encrypted"})
7   def f(x, y):
8       return (x + y) ** 2
9
10  # Compile with an input set
11  circuit = f.compile([(0, 2), (3, 4)], verbose=True)
12
13  # Encrypt data and export public evaluation material
14  encrypted_args = circuit.client.encrypt(1, 2)
15  eval_keys = circuit.client.evaluation_keys
16
17  # Evaluate on encrypted data
18  public_res = circuit.server.run(encrypted_args, eval_keys)
19
20  # Decrypt and assert that is equal to the clear evaluation
21  assert(f(1,2) == circuit.client.decrypt(public_res))
```

Concrete Python: The transpilation pipeline

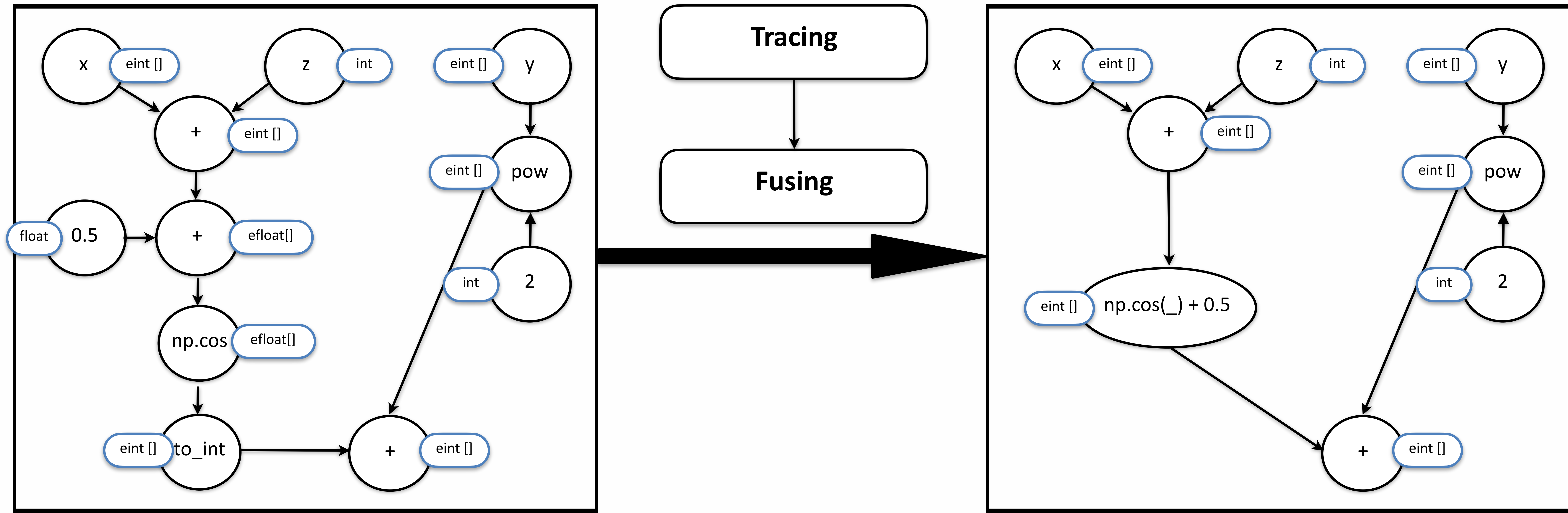
```
1   from concrete import fhe
2
3   import numpy as np
4
5   @fhe.compiler({"x": "encrypted",
6                  "y": "encrypted",
7                  "z": "clear"})
8   def f(x, y, z):
9       x = x + z
10      x = np.cos(x)
11      x = (x + 0.5).astype(np.int64)
12      y = y**2
13      return x + y
14
15  circuit = f.compile([([0,3], [2,4], 12)])
```

Tracing



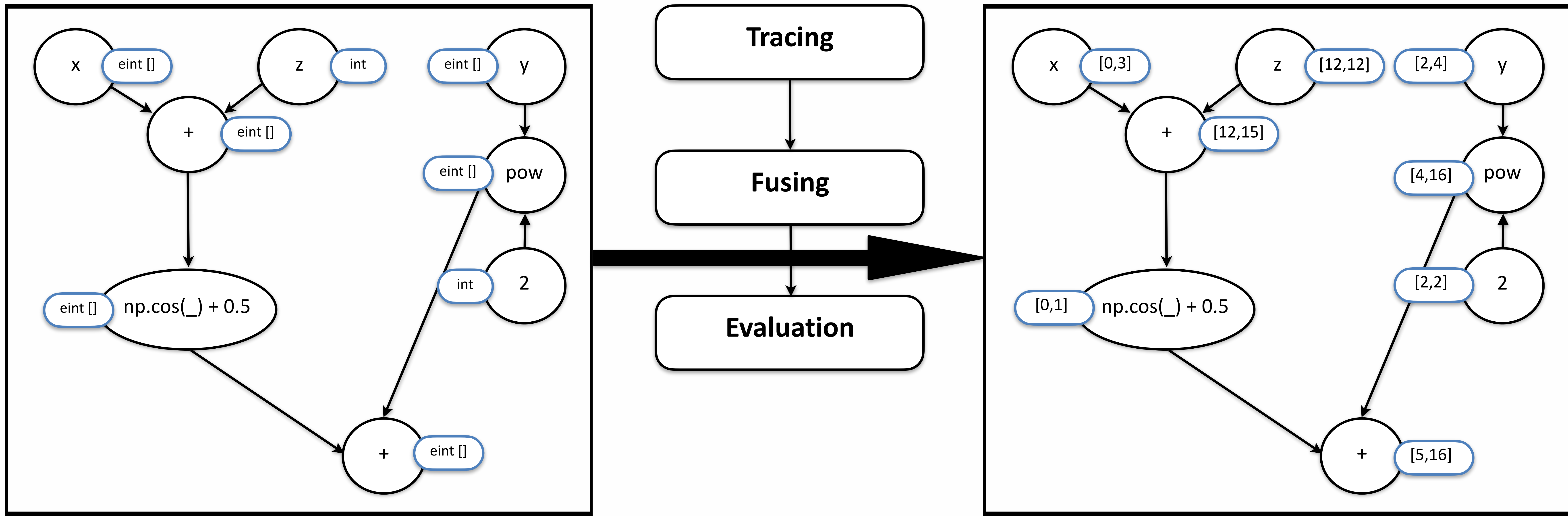
Trace the execution of the **python function to** build a **computation dag**

Concrete Python: The transpilation pipeline



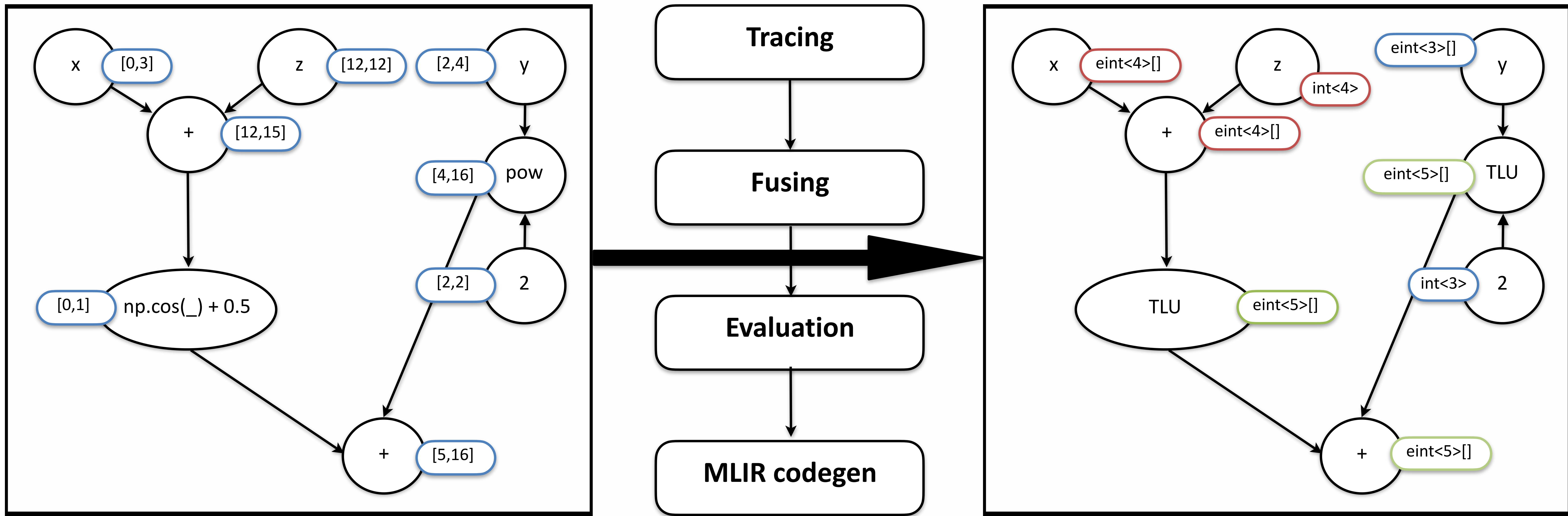
Fuse **floating** point subgraph **to** an **integer** node

Concrete Python: The transpilation pipeline



Evaluate the computation dag with the input set to **compute nodes bounds**

Concrete Python: The transpilation pipeline



Assign bitwidth to connected TLU free subgraph and **generate FHE MLIR code**

Concrete Python: MLIR generated code

```
1  func.func @main(%arg0: tensor<2x!FHE.eint<4>>, %arg1: tensor<2x!FHE.eint<3>>, %arg2: i5) -> tensor<2x!FHE.eint<5>> {
2    // Boiler plate code to transform scalar integer to one element tensor
3    %from_elements = tensor.from_elements %arg2 : tensor<1xi5>
4
5    // x + z
6    %0 = "FHELinalg.add_eint_int"(%arg0, %from_elements)
7        : (tensor<2x!FHE.eint<4>>, tensor<1xi5>) -> tensor<2x!FHE.eint<4>>
8
9    // np.cos(_) + 0.5
10   %cst = arith.constant dense<[1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]> : tensor<16xi64>
11   %1 = "FHELinalg.apply_lookup_table"(%0, %cst)
12       : (tensor<2x!FHE.eint<4>>, tensor<16xi64>) -> tensor<2x!FHE.eint<5>>
13
14   // pow(_, 2)
15   %cst_0 = arith.constant dense<[0, 1, 4, 9, 16, 25, 36, 49]> : tensor<8xi64>
16   %2 = "FHELinalg.apply_lookup_table"(%arg1, %cst_0)
17       : (tensor<2x!FHE.eint<3>>, tensor<8xi64>) -> tensor<2x!FHE.eint<5>>
18
19   // _ + _
20   %3 = "FHELinalg.add_eint"(%1, %2)
21       : (tensor<2x!FHE.eint<5>>, tensor<2x!FHE.eint<5>>) -> tensor<2x!FHE.eint<5>>
22   return %3 : tensor<2x!FHE.eint<5>>
23 }
```

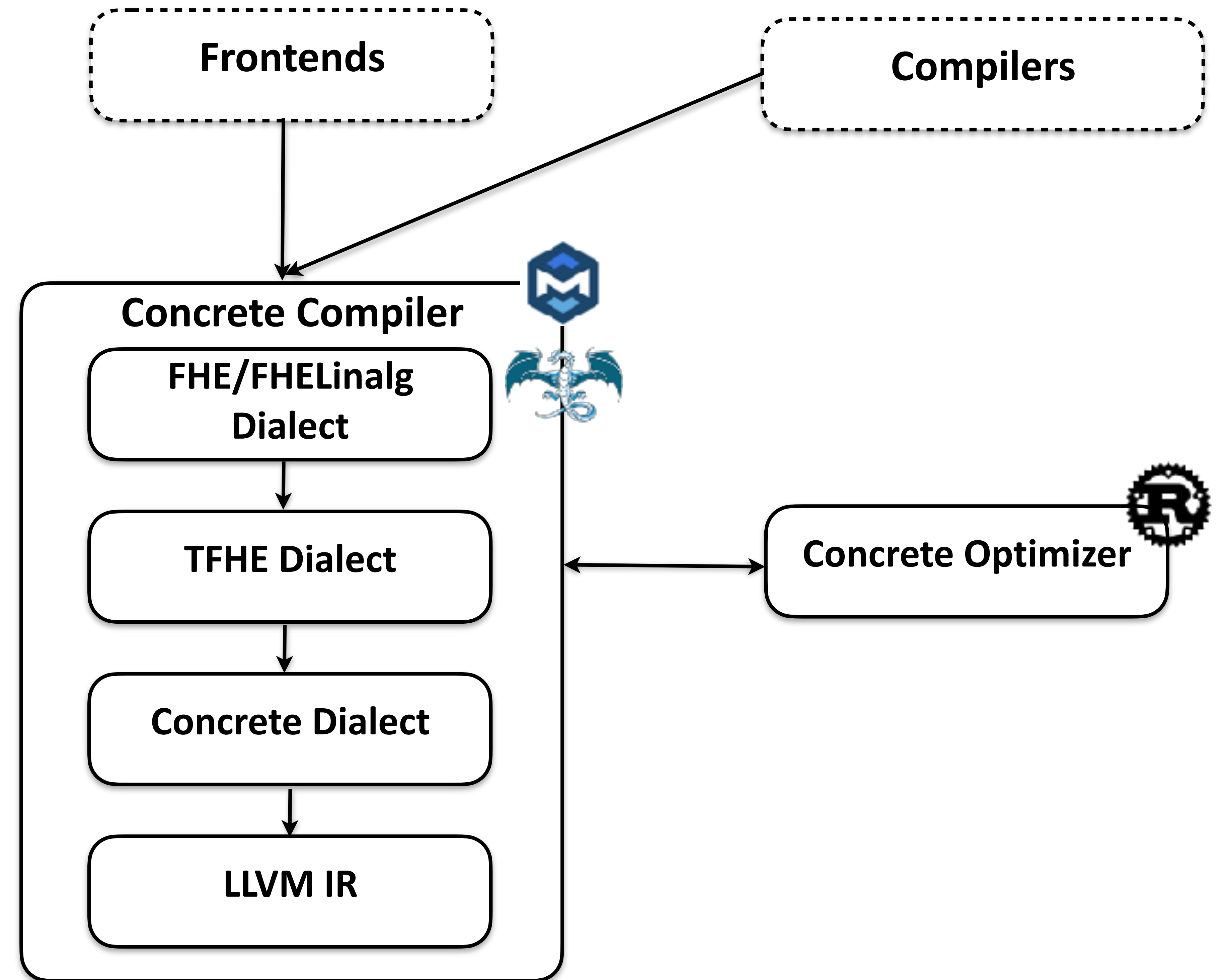
Concrete Compiler

A modular framework for FHE application

Concrete Compiler

High-Level Overview (1)

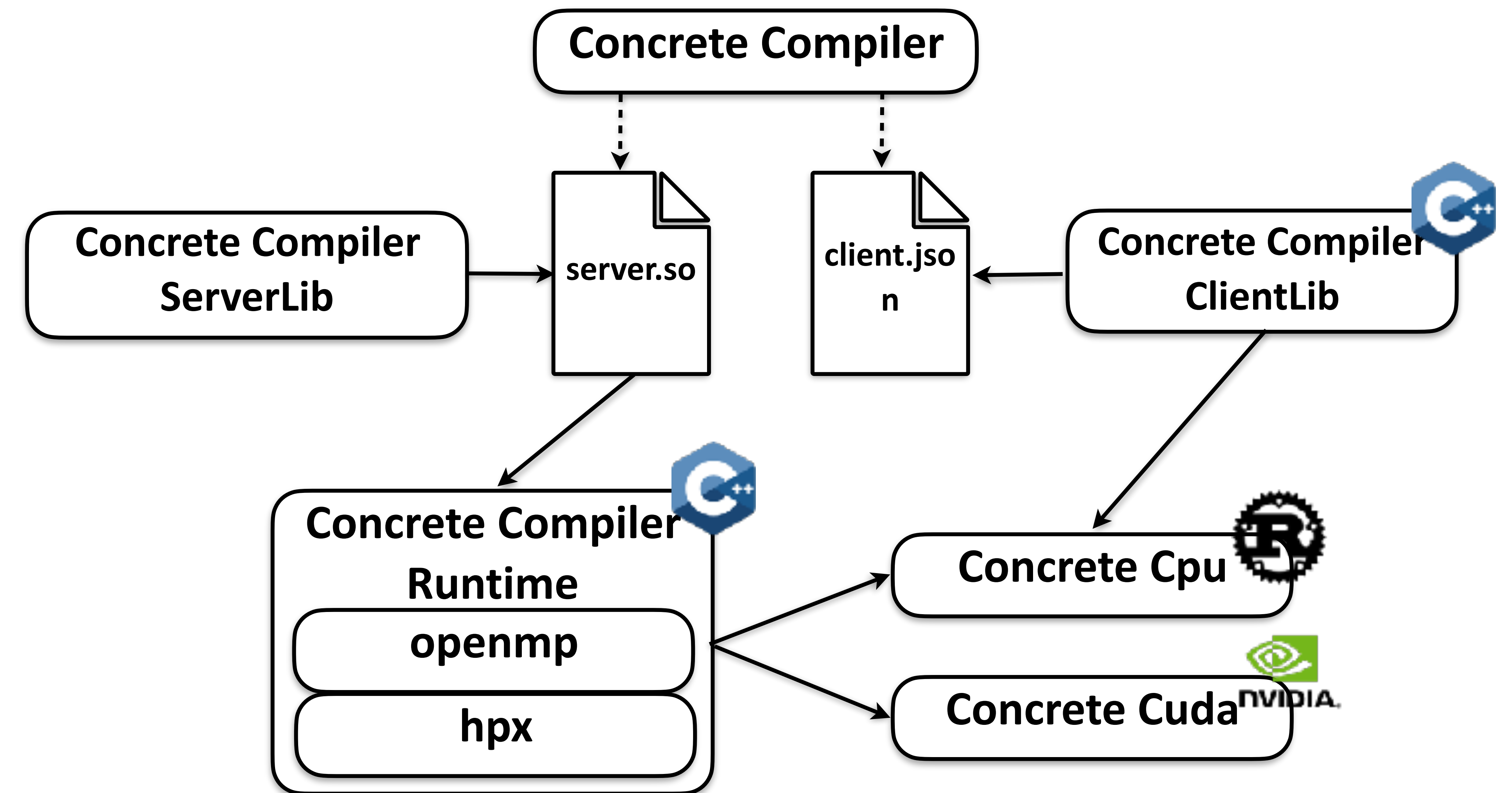
- **MLIR-**based compiler to be reusable and leverage community effort on common problems
- **Concrete Optimizer** to solve TFHE parametrization problems
- **LLVM Toolchain** to produce binary library



Concrete Compiler

High-Level Overview (2)

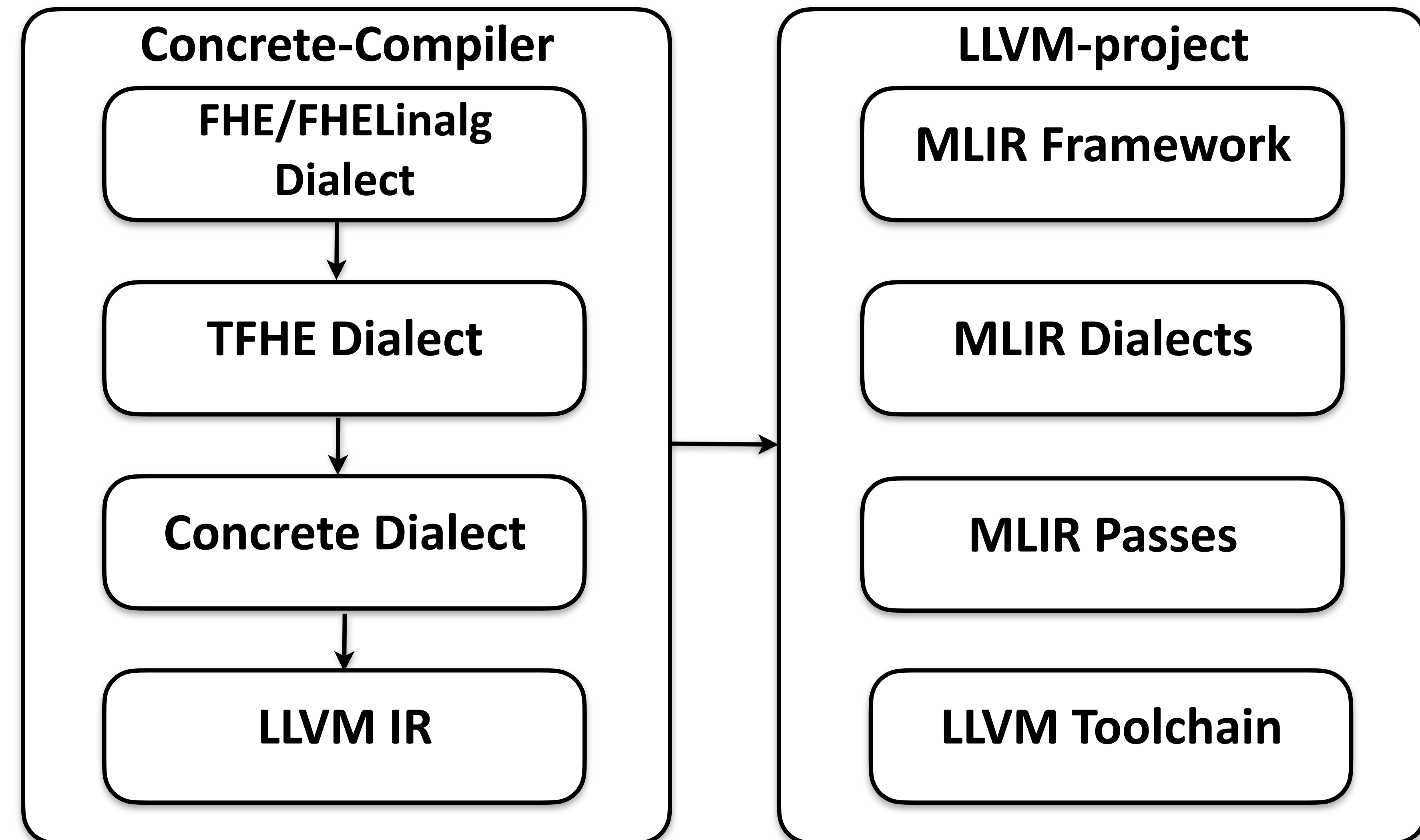
- Runtime linked with **OpenMP** and **HPX** libraries for loop parallelism and task scheduling
- Runtime linked to **Concrete CPU/GPU** backends to use the fastest hand optimized TFHE implementation
- **Client and Server toolkit** to use compilation artifacts



Concrete Compiler

Why MLIR?

- **MLIR infrastucture** allows to r**educe the cost** of building Concrete Compiler
- **Standard MLIR Dialects** to **model common** compiler **abstraction** (tensor, memref, linalg, scf, .omp, ..)
- **Standard MLIR Passes** to **solve common** compiler **problems** (canonicalization, linalg generalization, dead code elimination, bufferization...)
- Leverage **LLVM toolchain** to produce efficient binaries
- Allows for a **reusable definition of FHE-specific dialect** and optimization passes



Concrete Compiler

Specific Dialects

- **FHE Dialect** defines crypo-free FHE **types and scalar operators**
- **FHELinalg Dialect** defines very high level **tensor operators**
- **TFHE Dialect** introduces crypto-system dependent parameters and operators
- **Concrete Dialect** represents unabstracted implementation operators to prepare the codegen

```
1   %0 = "FHE.mul_eint"(%arg0, %arg1):
2   | (!FHE.eint<2>, !FHE.eint<2>) -> (!FHE.eint<2>)
```

```
1   %0 = "FHELinalg.matmul_eint_int"(%x, %y):
2   | (tensor<4x3x!FHE.eint<2>>, tensor<3x2xi3>) -> tensor<4x2x!FHE.eint<2>>
```

```
1   %2 = "TFHE.keyswitch_glwe"(%0) {key = #TFHE.ksk<sk<0,1,1280>, sk<1,1,677>, 3, 4>}
2   | : (!TFHE.glwe<sk<0,1,1280>>) -> !TFHE.glwe<sk<1,1,677>>
3   %3 = "TFHE.bootstrap_glwe"(%2, %1) {key = #TFHE.bsk<sk<1,1,677>, sk<0,1,1280>, 256, 5, 1, 15>}
4   | : (!TFHE.glwe<sk<1,1,677>>, tensor<256xi64>) -> !TFHE.glwe<sk<0,1,1280>>
```

```
1   %0 = "Concrete.add_lwe_tensor"(%arg0, %arg1)
2   | : (tensor<1281xi64>, tensor<1281xi64>) -> tensor<1281xi64>
```

Concrete Compiler

High Level Pipeline

- A set of **analysis passes** to **build** the FHE **constraint DAG**
- A set of **transformations' passes** to translate non natively supported TFHE operators
- A set of **conversion passes** to go from **FHE/FHELinalg** dialects **to** TFHE following Concrete Optimizer re-writing guidelines

```
1  %0 = "FHE.mul_eint"(%arg0, %arg1):
2    (!FHE.eint<2>, !FHE.eint<2>) -> (!FHE.eint<2>)
```

FHE Transformations

FHE Analysis

TFHE Optimization

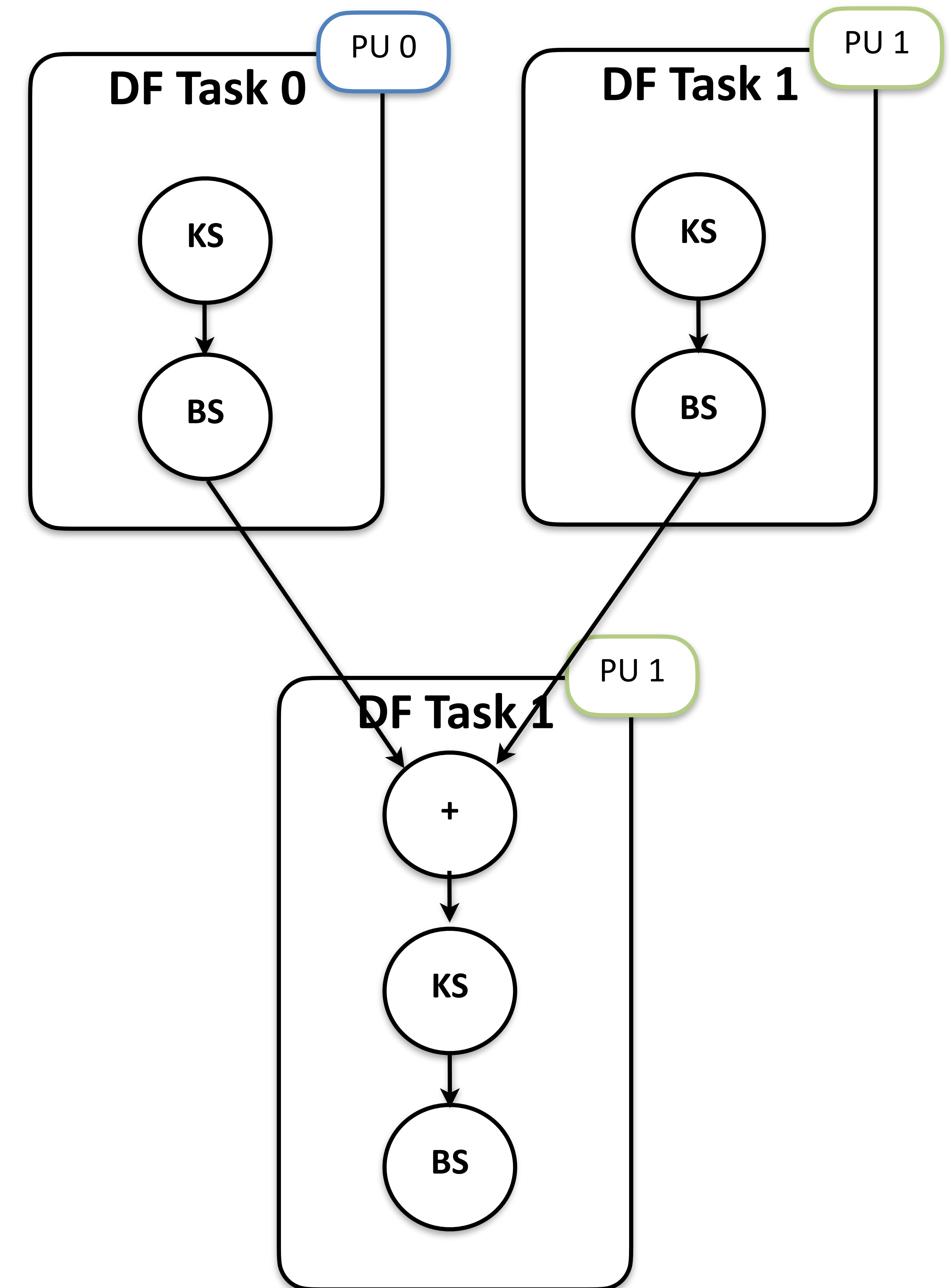
Conversion

```
1  %c4611686018427387904_i64 = arith.constant 4611686018427387904 : i64
2  %cst = arith.constant dense<[0, 0, 1, 0]> : tensor<4xi64>
3  %cst_0 = arith.constant dense<[0, 0, 1, 2]> : tensor<4xi64>
4  %0 = "TFHE.add_glwe"(%arg0, %arg1) : (!TFHE.glwe<sk<0,1,1280>>, !TFHE.glwe<sk<0,1,1280>>)
5  %1 = "TFHE.encode_expand_lut_for_bootstrap"(%cst_0) {isSigned = false, outputBits = 2 : i3
6  %2 = "TFHE.keyswitch_glwe"(%0) {key = #TFHE.ksk<sk<0,1,1280>, sk<1,1,677>, 3, 4>} : (!TFHE
7  %3 = "TFHE.bootstrap_glwe"(%2, %1) {key = #TFHE.bsk<sk<1,1,677>, sk<0,1,1280>, 256, 5, 1,
8  %4 = "TFHE.neg_glwe"(%arg1) : (!TFHE.glwe<sk<0,1,1280>>) -> !TFHE.glwe<sk<0,1,1280>>
9  %5 = "TFHE.add_glwe"(%arg0, %4) : (!TFHE.glwe<sk<0,1,1280>>, !TFHE.glwe<sk<0,1,1280>>) ->
10 %6 = "TFHE.encode_expand_lut_for_bootstrap"(%cst) {isSigned = true, outputBits = 2 : i32,
11 %7 = "TFHE.add_glwe_int"(%5, %c4611686018427387904_i64) : (!TFHE.glwe<sk<0,1,1280>>, i64)
12 %8 = "TFHE.keyswitch_glwe"(%7) {key = #TFHE.ksk<sk<0,1,1280>, sk<1,1,677>, 3, 4>} : (!TFHE
13 %9 = "TFHE.bootstrap_glwe"(%8, %6) {key = #TFHE.bsk<sk<1,1,677>, sk<0,1,1280>, 256, 5, 1,
14 %10 = "TFHE.neg_glwe"(%9) : (!TFHE.glwe<sk<0,1,1280>>) -> !TFHE.glwe<sk<0,1,1280>>
15 %11 = "TFHE.add_glwe"(%3, %10) : (!TFHE.glwe<sk<0,1,1280>>, !TFHE.glwe<sk<0,1,1280>>) -> !
```

Concrete Compiler

Parallelism and distribution

- Automatic **loop parallelism** using high level information to lower to openmp
- Automatic **dataflow parallelism** by generating a dataflow task graph using high level information, and **tiling**
- A **SDFG** Dialect (Static DataFlow Graph) instantiated with high level TFHE primitives to **offload** a whole **TFHE subgraph** (pipeline) for **hardware accelerators** (GPU, ...)
- A **dataflow runtime** based on HPX to implement dataflow **tasks' parallelism** and distributed computation
- A **GPU SDFG runtime** to schedule GPU kernels over multi-GPU hosts

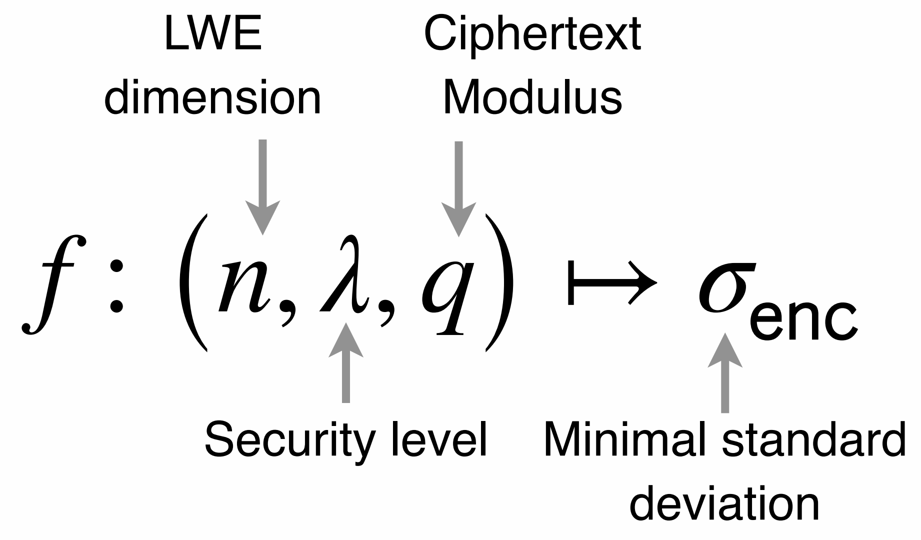


Concrete Optimizer

An optimizer for TFHE

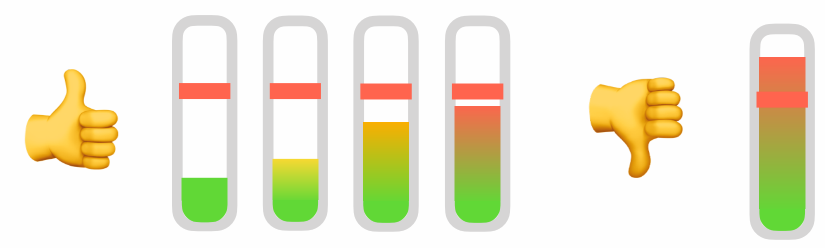
Goals

Security



→ Using the **lattice estimator**

Correctness

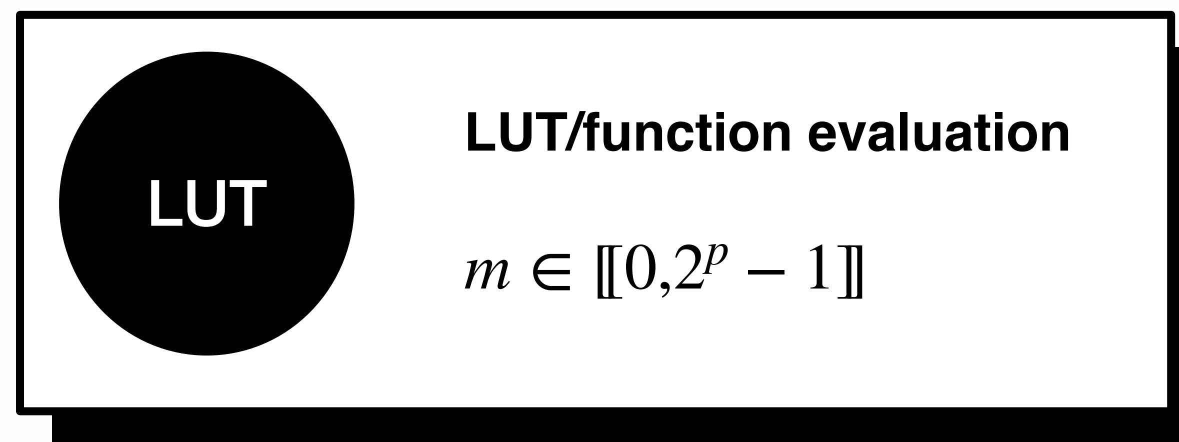


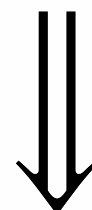
→ **Noise Model** to track the noise along the computation

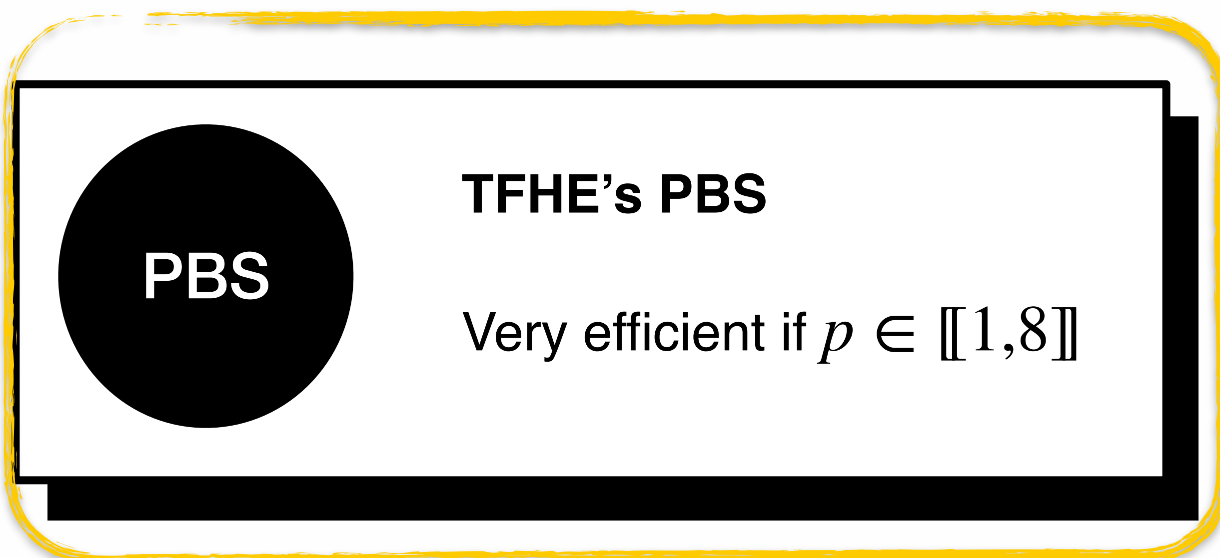
Efficiency

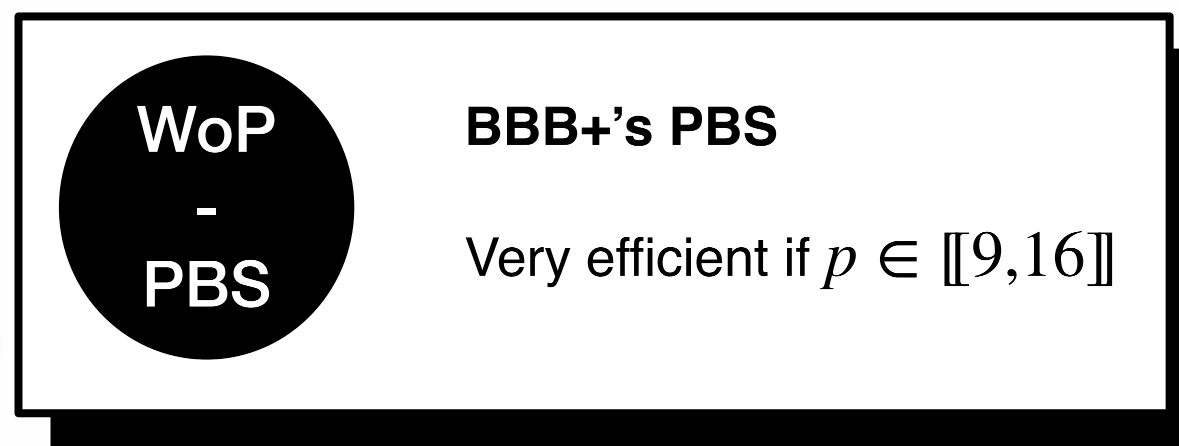
→ **Cost Model** as a surrogate of the execution time

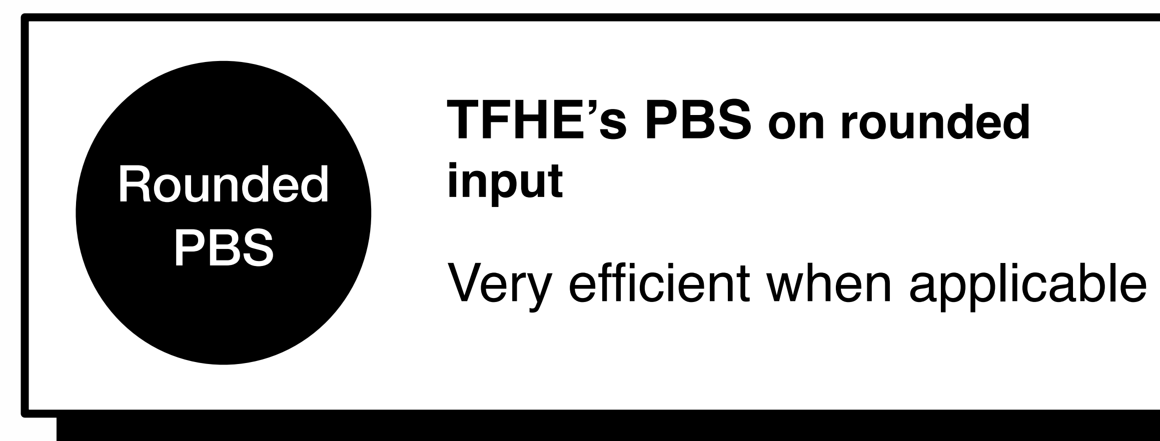
Choice of algorithm





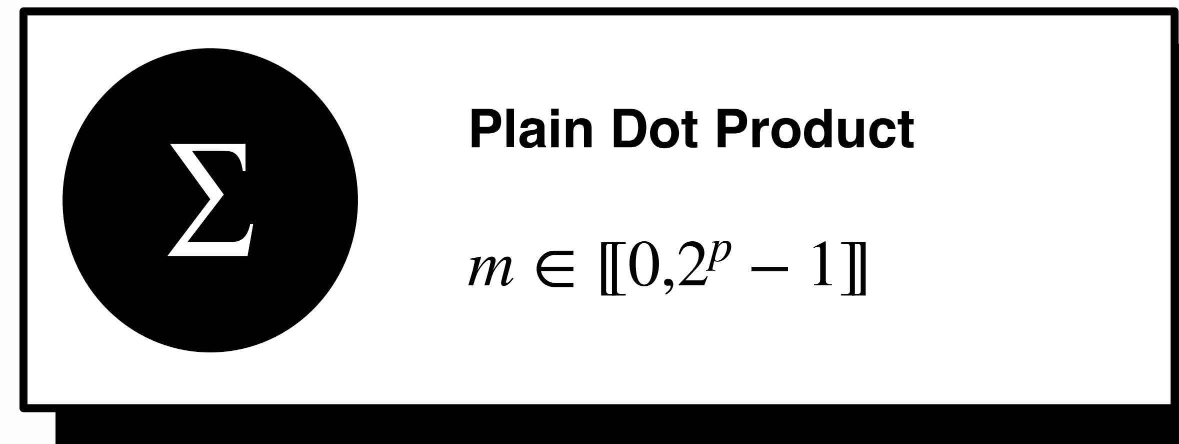


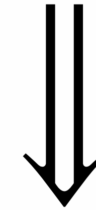


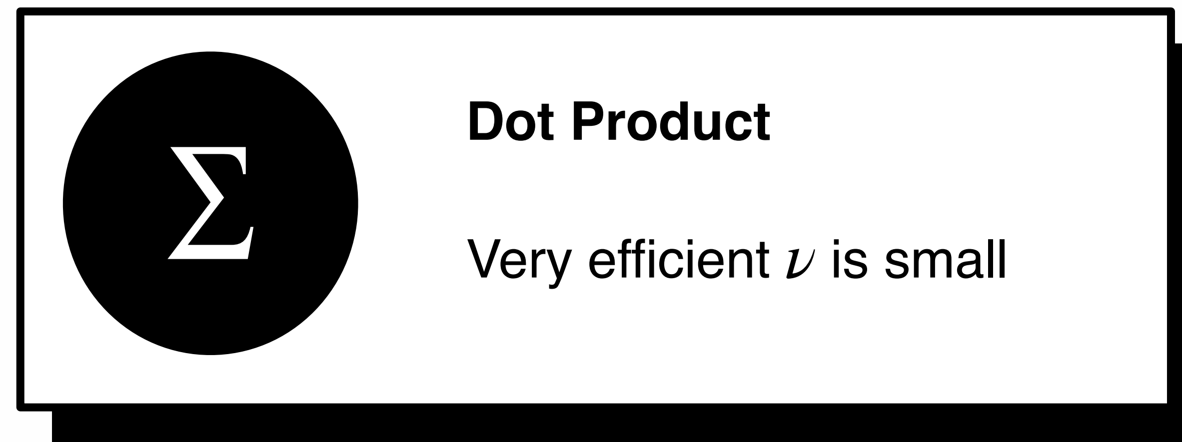


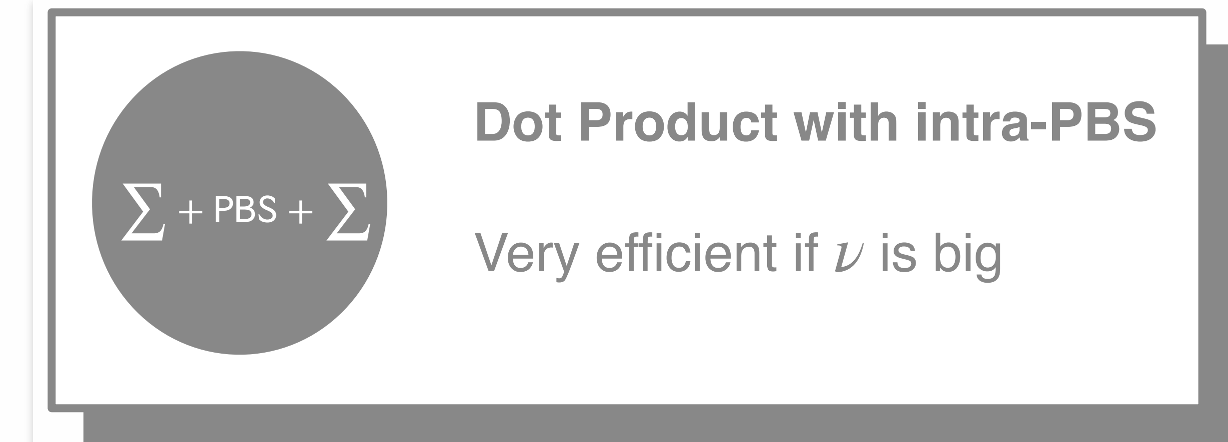
[CGGI20] I. Chillotti, N. Gama, M. Georgieva, M. Izabachène. TFHE: Fast Fully Homomorphic Encryption over the Torus. Journal of Cryptology 2020.
[BBB+22] L. Bergerat, A. Boudi, Q. Bourgerie, I. Chillotti, D. Ligier, J.-B. Orfila, S. Tap Parameter Optimization & Larger Precision for (T)FHE. Eprint

Choice of algorithm









Translation

Additions, substractions, multiplications

m_1

$\omega_1, \cdots, \omega_{\dots}$

$m_{\dots}$

$\sum$

$\sum \omega_i \cdot m_i$

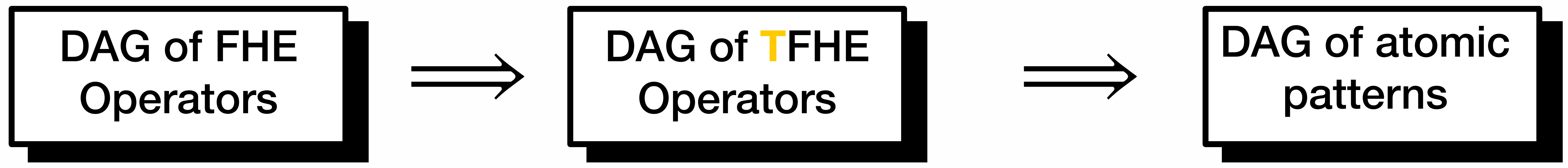
Lookup table + noise management

KS

x

PBS

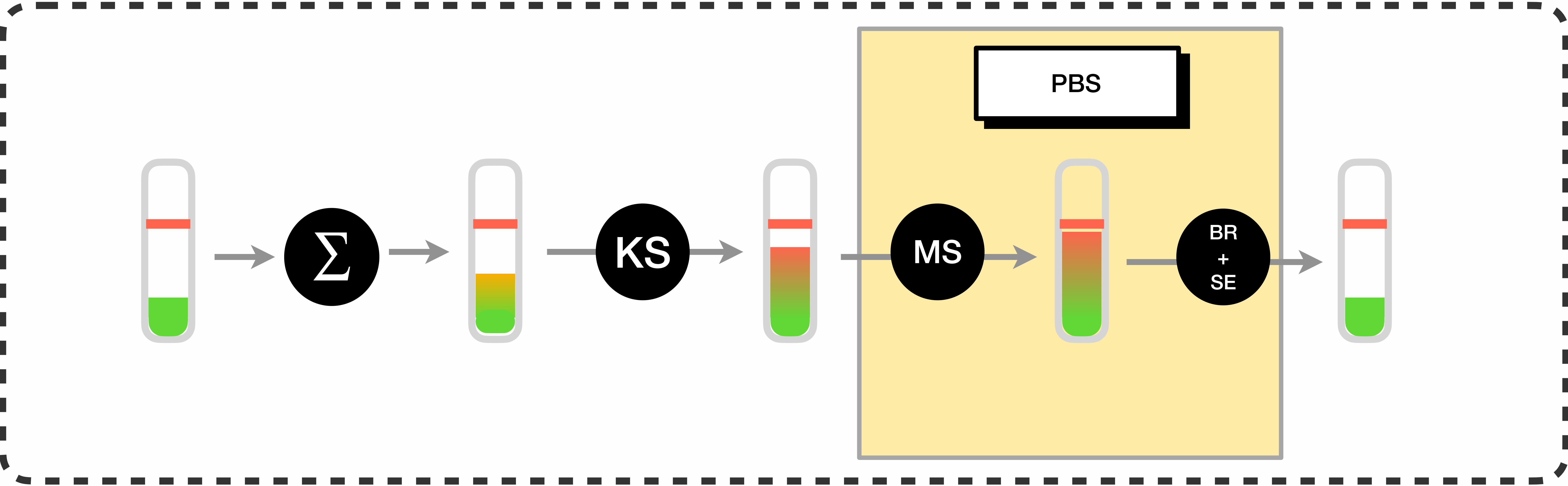
$f(x)$



[CJP21] I. Chillotti, M. Joye, and P. Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In CSCML 2021

[BBB+22] L. Bergerat, A. Boudi, Q. Bourgerie, I. Chillotti, D. Ligier, J.-B. Orfila, S. Tap Parameter Optimization & Larger Precision for (T)FHE. Eprint

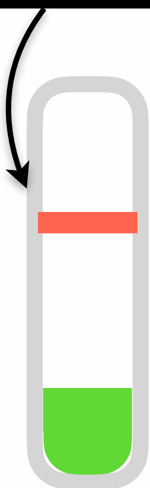
Noise analysis of an Atomic Pattern



Noise is increasing between two modulus switching

Noise Bound

Noise Bound is a function of



message precision

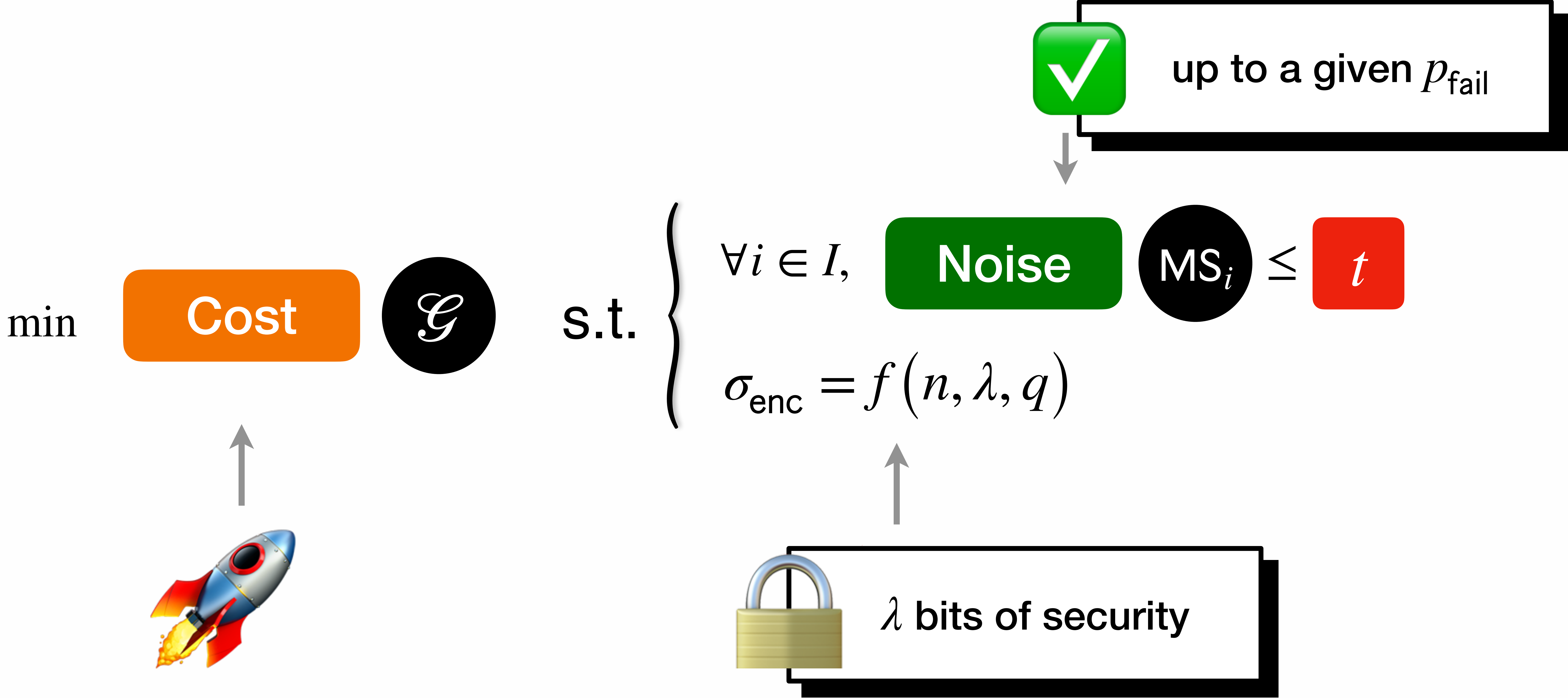
encoding

Natif, CRT, Radix

failure probability

DAG dependent

Optimization Problem



Reducing the search space

Constraint between parameters

For one atomic pattern

$|\text{search space}| \approx 2^{40} \Longrightarrow |\text{search space}| \approx 2^{30}$

Remove unnecessary noise constraints

For a DAG $\mathcal{G} = \{A_i\}_{i \in I}$ and P the set of precision in $\mathcal{G}$

$|\text{constraints}| \propto |I| \Longrightarrow |\text{constraints}| \propto |P|$

One vs several parameter set

For one parameter set

$|\text{search space}| \propto 2^{30 \cdot |I|} \Longrightarrow |\text{search space}| \approx 2^{30}$

For X parameter sets

$|\text{search space}| \propto 2^{30 \cdot |X|} \Longrightarrow |\text{search space}| \approx 2^{30} \cdot |X|$

Conclusion

Concrete

A growing community

821 githubs stars

39 contributors

2916 commits

A complete stack for FHE

An easy to use frontend

A reusable compiler infrastructure

Multi backend integrations

Built to be fast

TFHE native integer

A specific TFHE optimizer

A compiler pipeline and runtime to scale

Thank you.

ZAMA

Contact and Links

quentin.bourgerie@zama.ai
samuel.tap@zama.ai

zama.ai

github.com/zama-ai/concrete

community.zama.ai