Quentin Bourgerie
Samuel Tap

15.06.2023

Concrete

Zama's FHE compiler

ZAMA

Meet The Team

Quentin Bourgerie
Head of Concrete

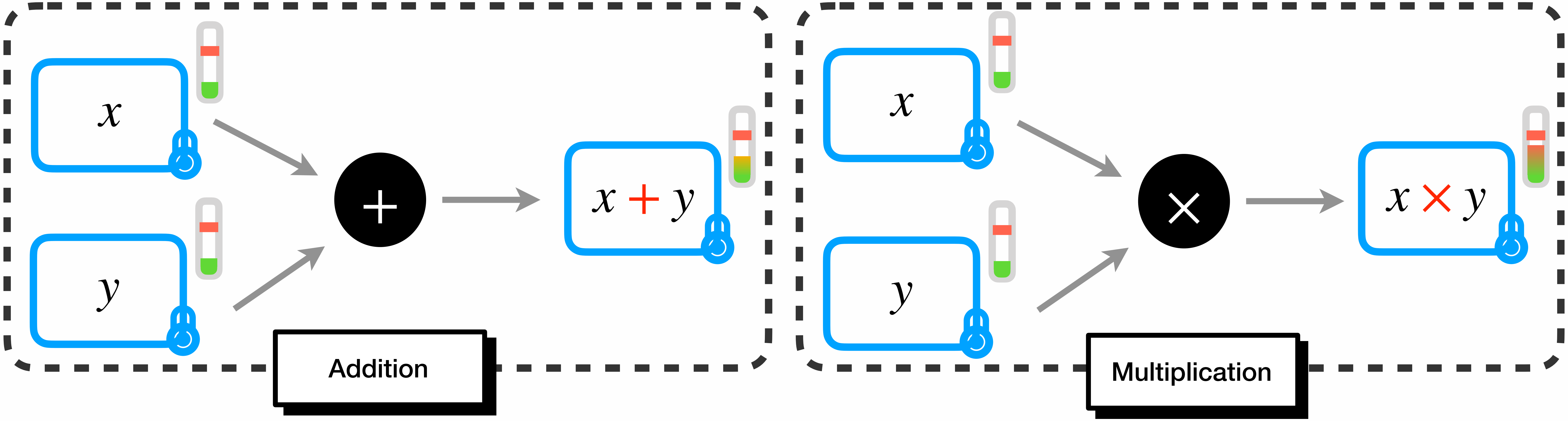
Samuel Tap
Scientific Advisor

Agenda

Introduction

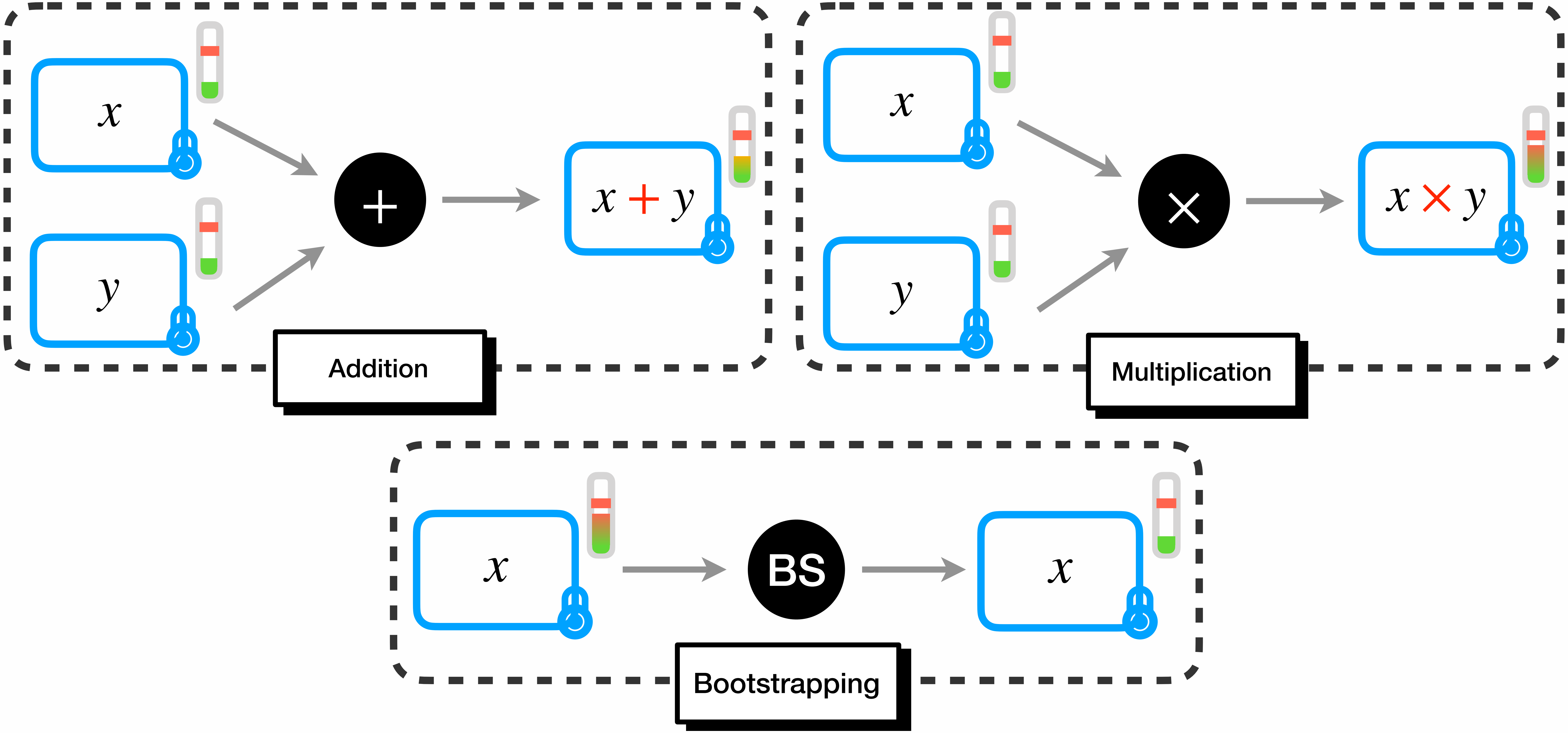
Why do we need a compiler for FHE ?

FHE



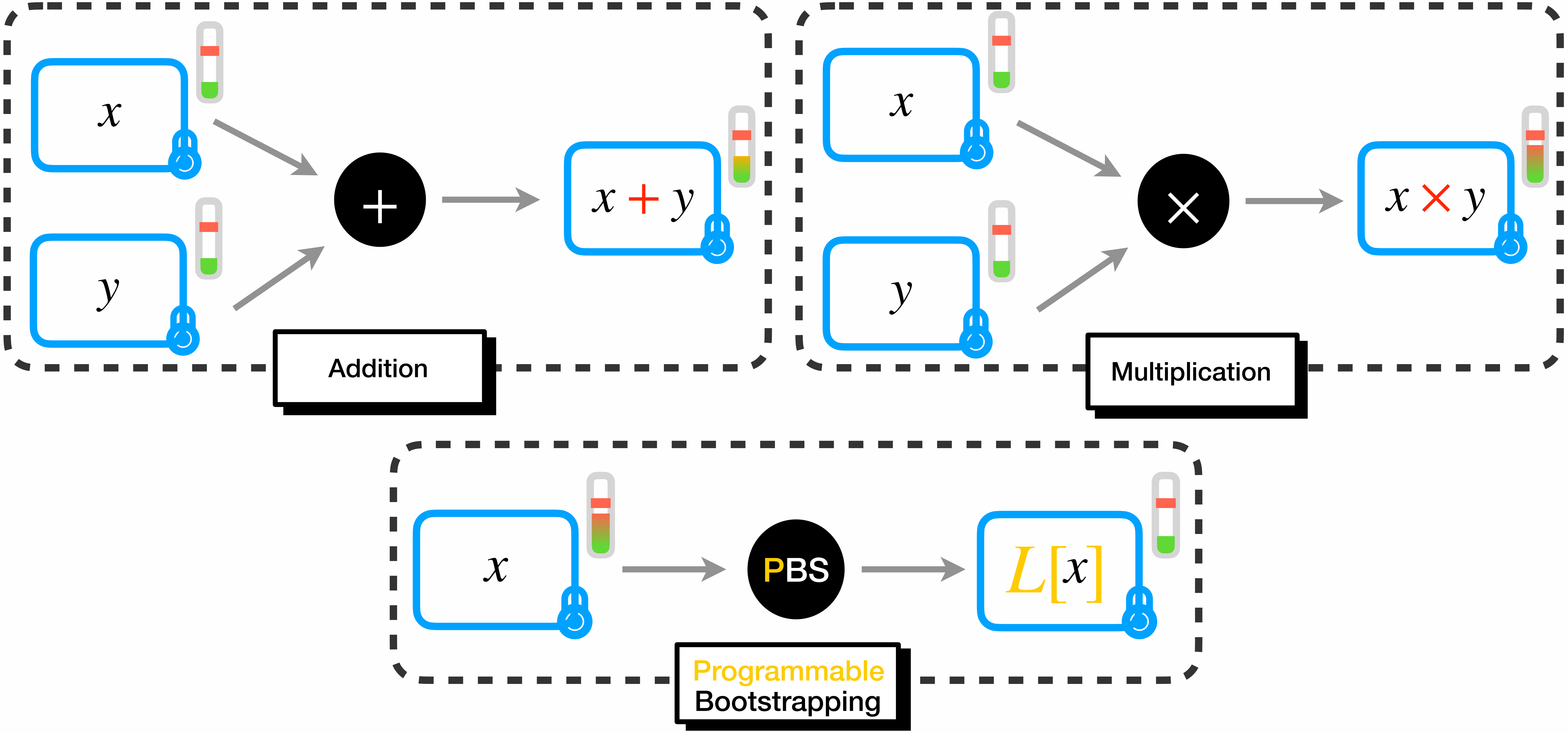
too much noise $\implies$ incorrect decryption 💀

FHE



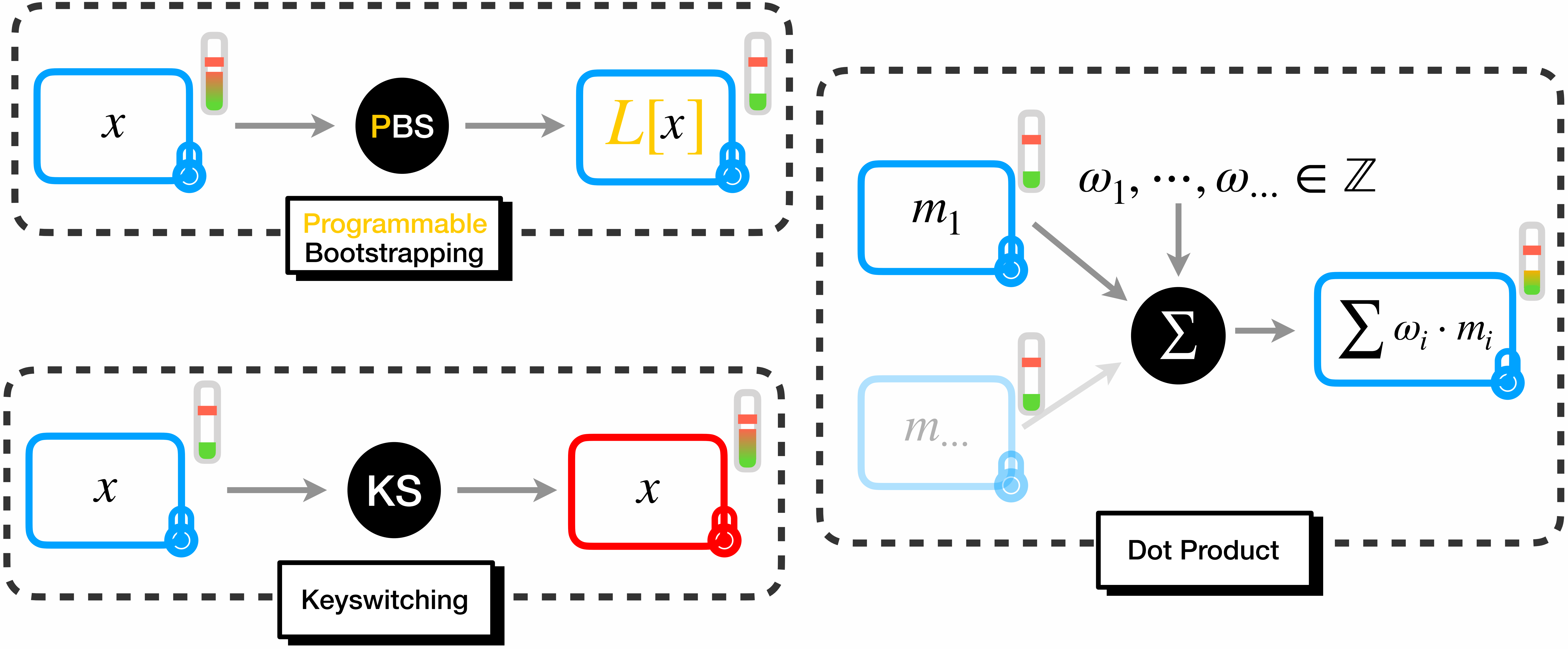
[CGGI20] I. Chillotti, N. Gama, M. Georgieva, M. Izabachène. TFHE: Fast Fully Homomorphic Encryption over the Torus. Journal of Cryptology 2020.

TFHE



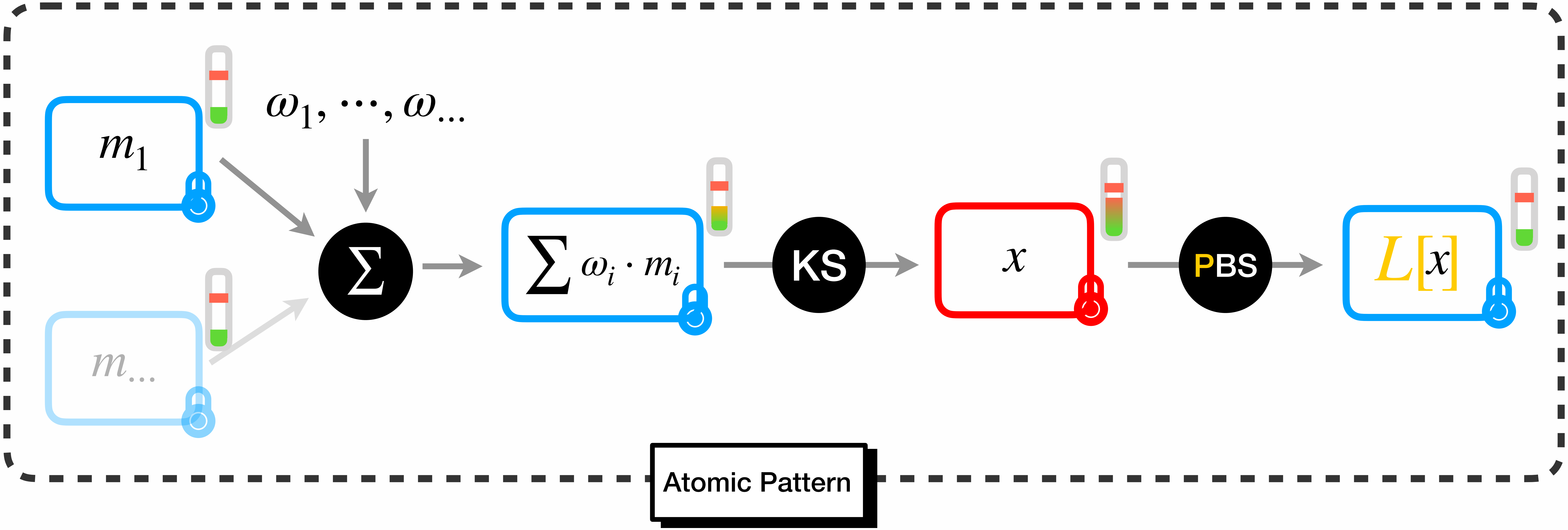
[CGGI20] I. Chillotti, N. Gama, M. Georgieva, M. Izabachène. TFHE: Fast Fully Homomorphic Encryption over the Torus. Journal of Cryptology 2020.

TFHE building blocks



[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. To appear in JoC

Enhancing TFHE

Boolean

Classical approach with TFHE

Independent steps: translation, boolean circuit optimization, parametrization

Leverage known techniques for boolean circuits

Support wide range of use cases

Support large precision

Fixed set of parameters

Integer

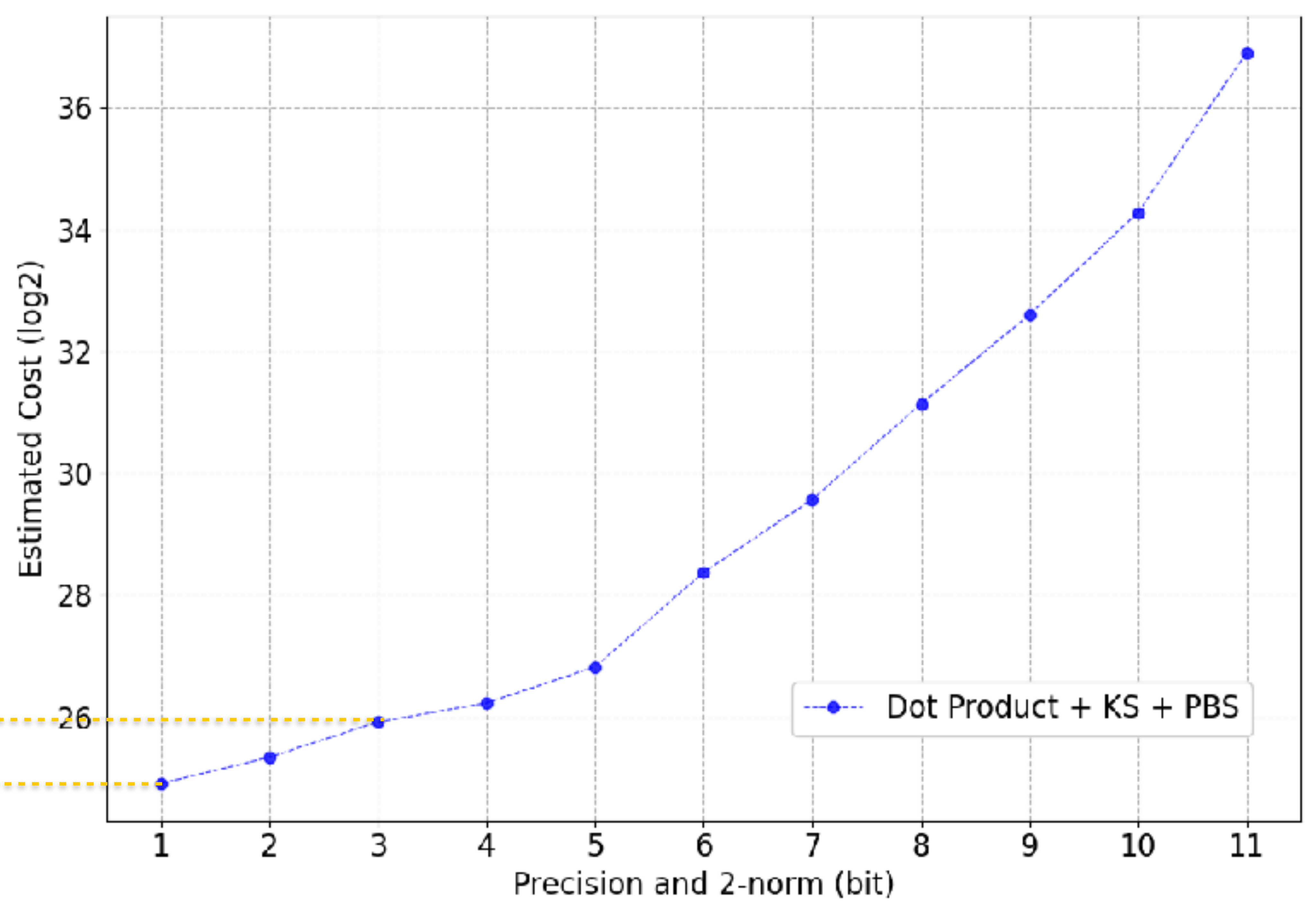
PBS-free leveled operations (additions, subtractions, …)

Generalization of the boolean approach

Perfect for small integer use-cases

Approximate paradigm

Enhancing TFHE



11.7 ms/bit ⟸ 35.1 ms

28.9 ms/bit ⟸ 28.9 ms

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

WoP-PBS

Rounded-PBS

$\Longrightarrow$

Needs

Optimal translation of a plain DAG into a TFHE DAG

Pick parameters

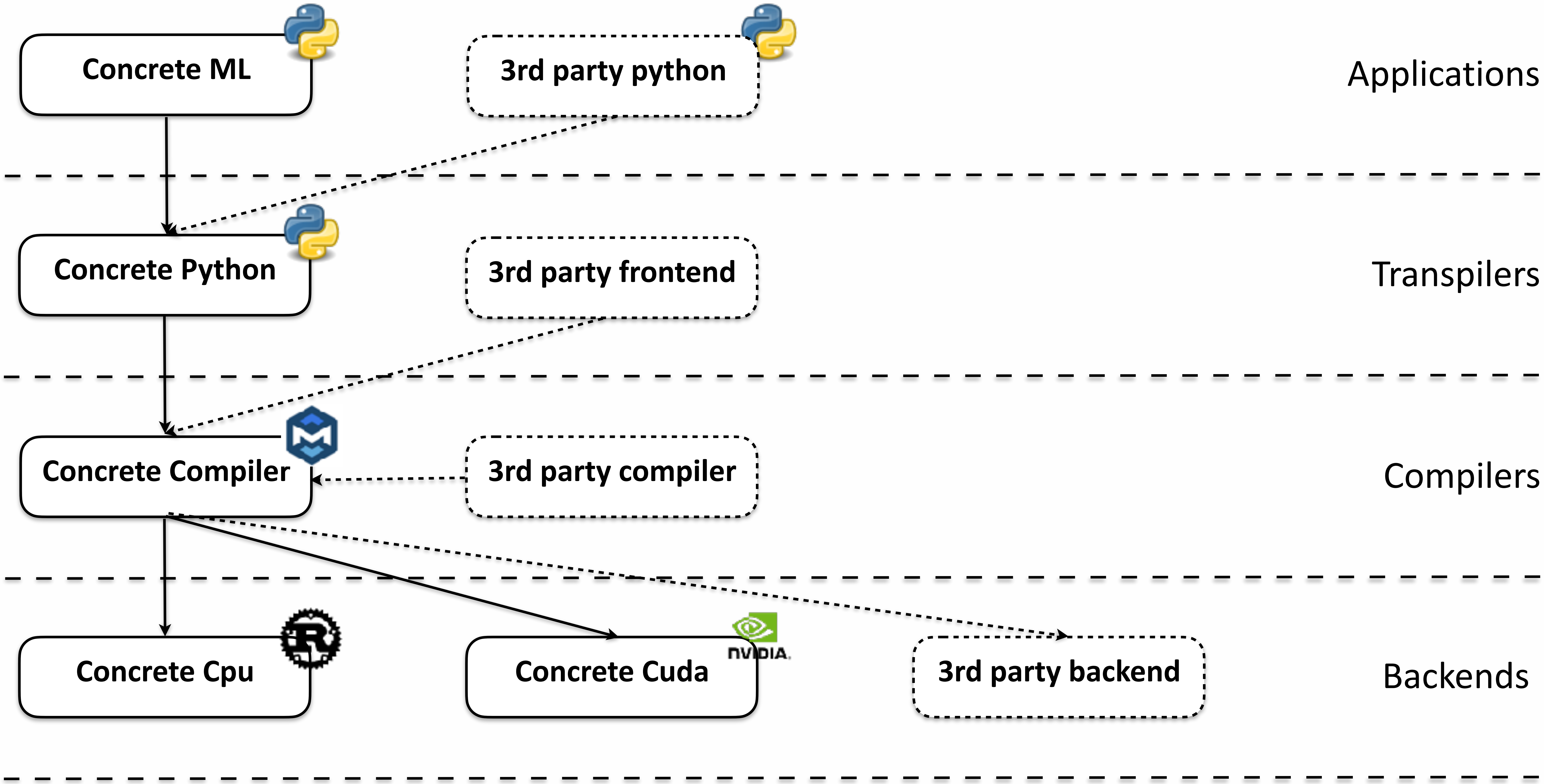
Easy-to-use toolchain

Target non-crypto users

Concrete

A modular framework for FHE applications

Concrete: a modular framework



Concrete Python

Transpiler for Concrete Compiler

Concrete Python

An easy to use frontend

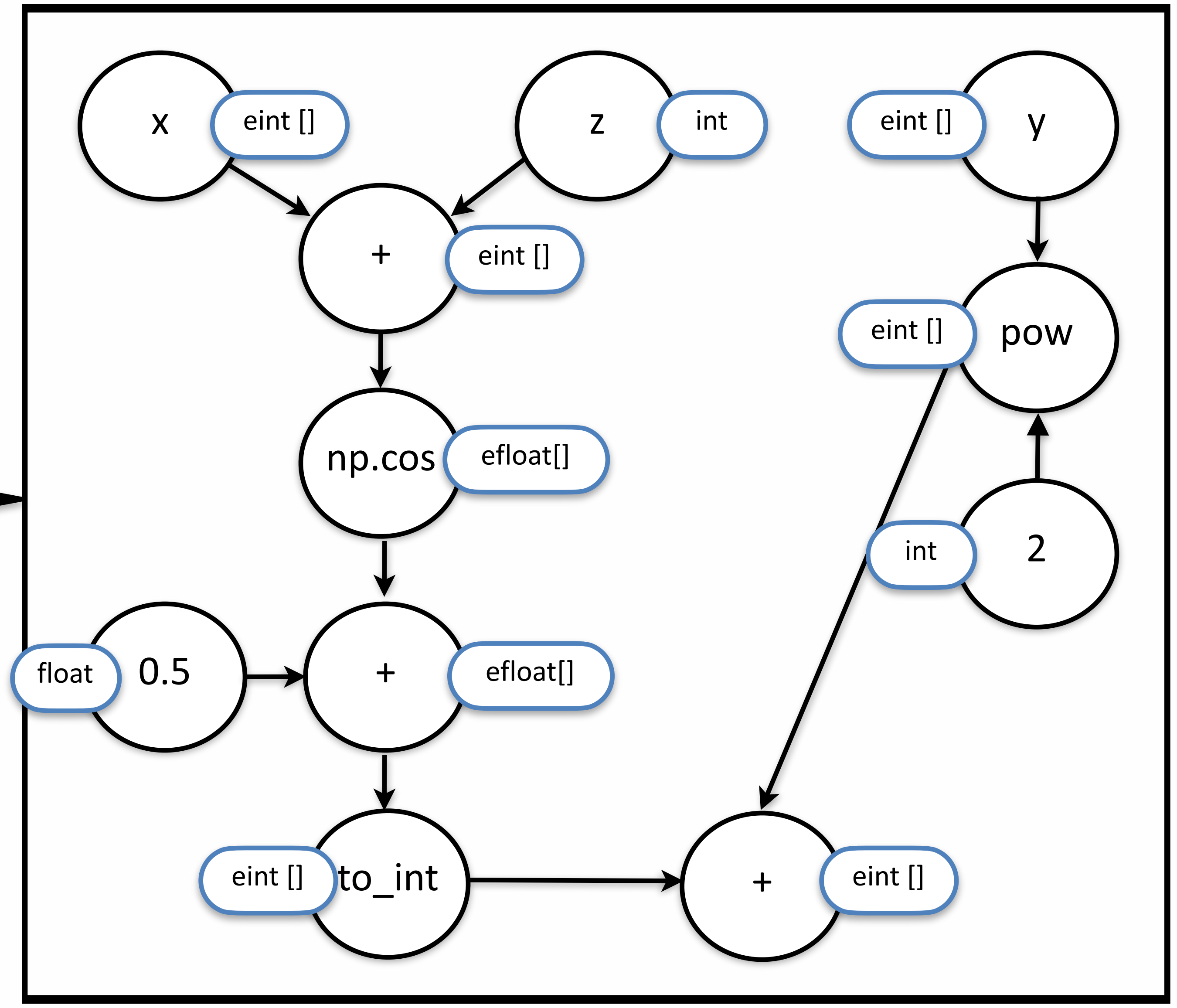
- Simple interface to **compile python programs**
- **Type inference** based on the dataset evaluation
- **Client and server API** to run FHE evaluation
- Seamless conversion of univariate **floating point and integer function** to table lookup
- **Extensive support of python** and numpy standard functions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

```python
from concrete import fhe

import numpy as np

# Define your standard python function
@fhe.compiler({"x": "encrypted", "y": "encrypted"})
def f(x, y):
    return (x + y) ** 2

# Compile with an input set
circuit = f.compile([(0, 2), (3, 4)], verbose=True)

# Encrypt data and export public evaluation material
encrypted_args = circuit.client.encrypt(1, 2)
eval_keys = circuit.client.evaluation_keys

# Evaluate on encrypted data
public_res = circuit.server.run(encrypted_args, eval_keys)

# Decrypt and assert that is equal to the clear evaluation
assert(f(1,2) == circuit.client.decrypt(public_res))
```

Concrete Python: The transpilation pipeline

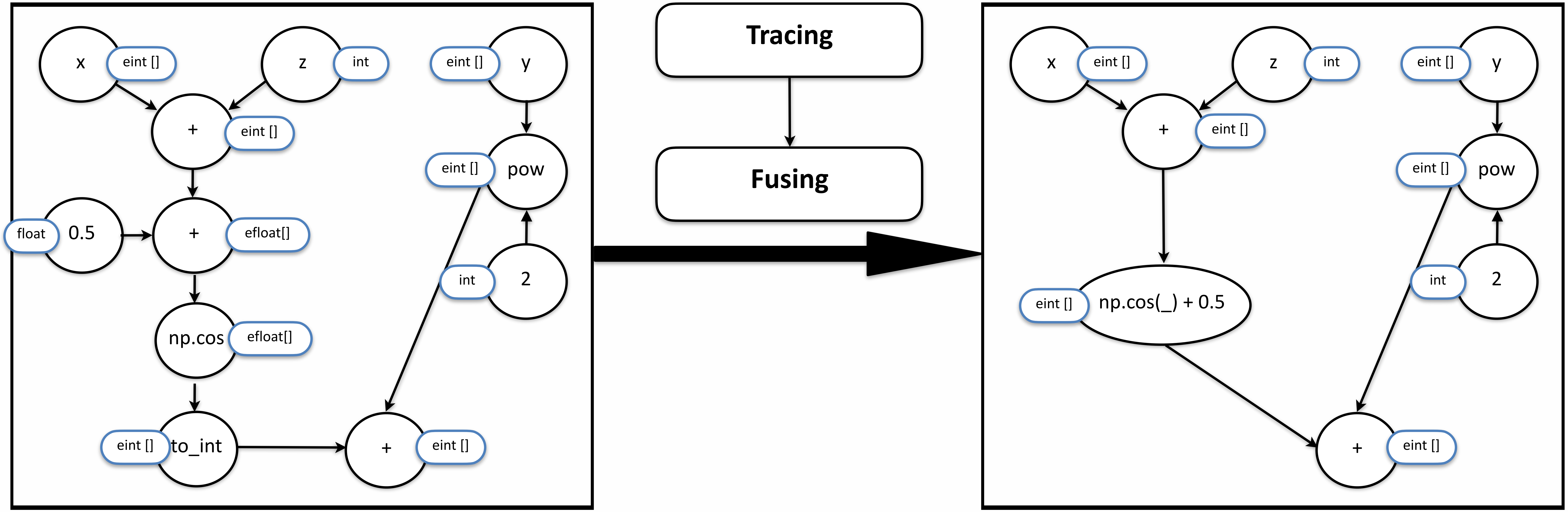
```python
from concrete import fhe

import numpy as np

@fhe.compiler({"x": "encrypted",
               "y": "encrypted",
               "z": "clear"})
def f(x, y, z):
    x = x + z
    x = np.cos(x)
    x = (x + 0.5).astype(np.int64)
    y = y**2
    return x + y

circuit = f.compile([([0,3], [2,4], 12)])
```

Tracing



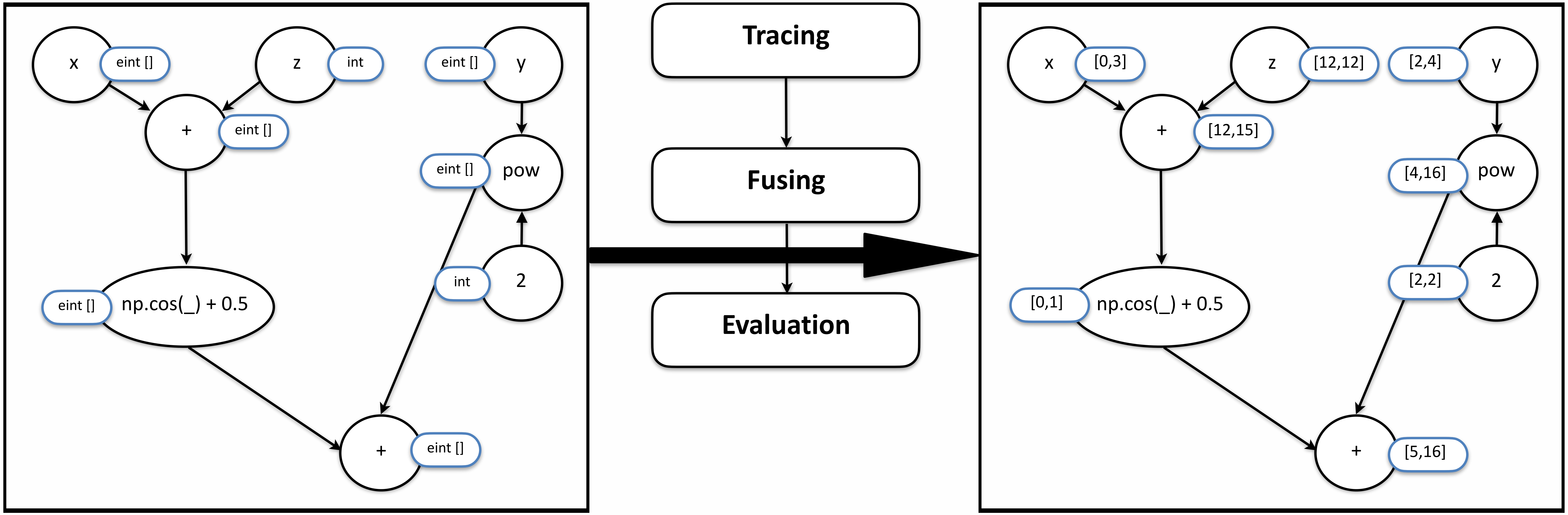
Trace the execution of the **python function to** build a **computation dag**

Concrete Python: The transpilation pipeline



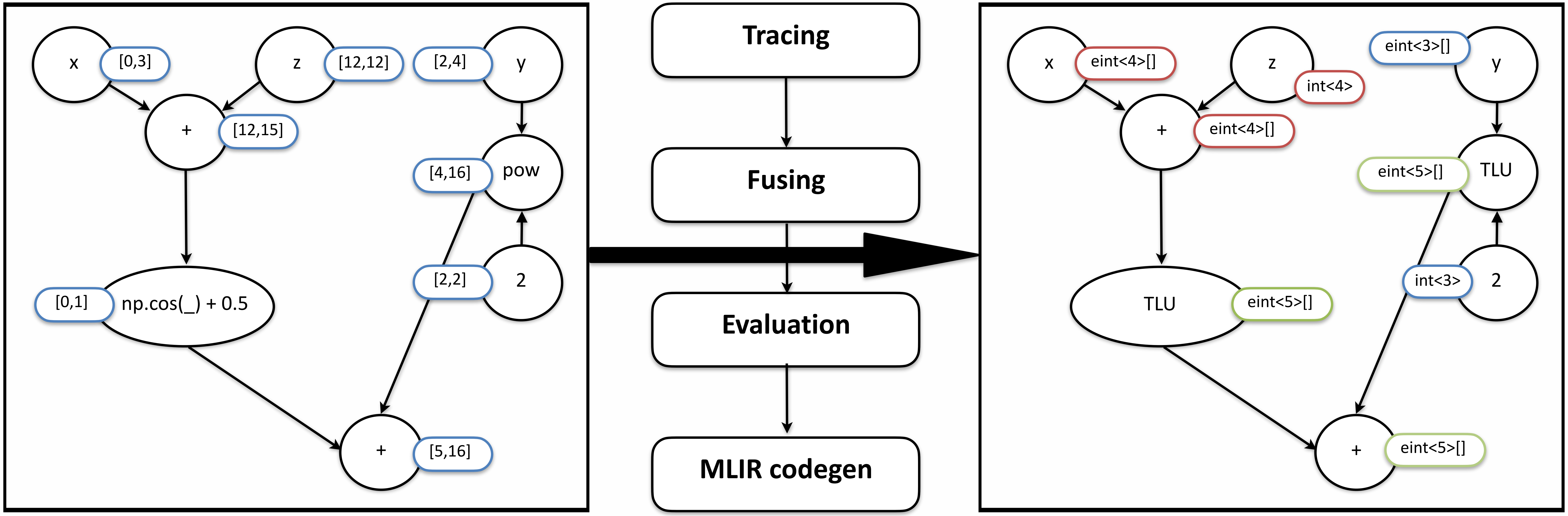
Fuse **floating** point subgraph **to** an **integer** node

Concrete Python: The transpilation pipeline



Evaluate the computation dag with the dataset 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

```
func.func @main(%arg0: tensor<2x!FHE.eint<4>>, %arg1: tensor<2x!FHE.eint<3>>, %arg2: i5) -> tensor<2x!FHE.eint<5>> {
  // Boiler plate code to transform scalar integer to one element tensor
  %from_elements = tensor.from_elements %arg2 : tensor<1xi5>

  // x + z
  %0 = "FHELinalg.add_eint_int"(%arg0, %from_elements)
      : (tensor<2x!FHE.eint<4>>, tensor<1xi5>) -> tensor<2x!FHE.eint<4>>

  // np.cos(_) + 0.5
  %cst = arith.constant dense<[1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]> : tensor<16xi64>
  %1 = "FHELinalg.apply_lookup_table"(%0, %cst)
      : (tensor<2x!FHE.eint<4>>, tensor<16xi64>) -> tensor<2x!FHE.eint<5>>

  // pow(_, 2)
  %cst_0 = arith.constant dense<[0, 1, 4, 9, 16, 25, 36, 49]> : tensor<8xi64>
  %2 = "FHELinalg.apply_lookup_table"(%arg1, %cst_0)
      : (tensor<2x!FHE.eint<3>>, tensor<8xi64>) -> tensor<2x!FHE.eint<5>>

  // _ + _
  %3 = "FHELinalg.add_eint"(%1, %2)
      : (tensor<2x!FHE.eint<5>>, tensor<2x!FHE.eint<5>>) -> tensor<2x!FHE.eint<5>>
  return %3 : tensor<2x!FHE.eint<5>>
}
```

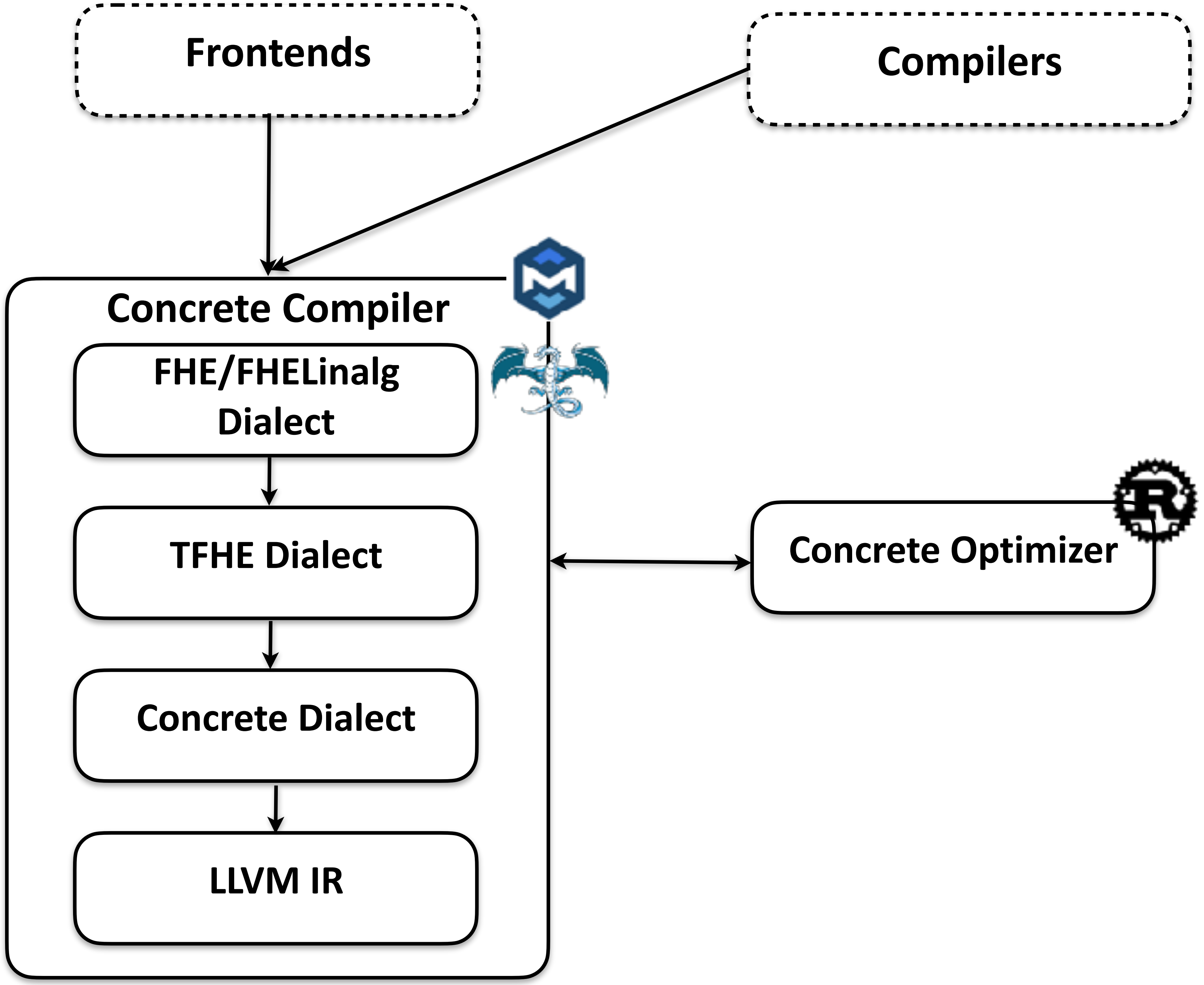
Concrete Compiler

From crypto-free representation to TFHE executable

Concrete Compiler

High-Level Overview (1)

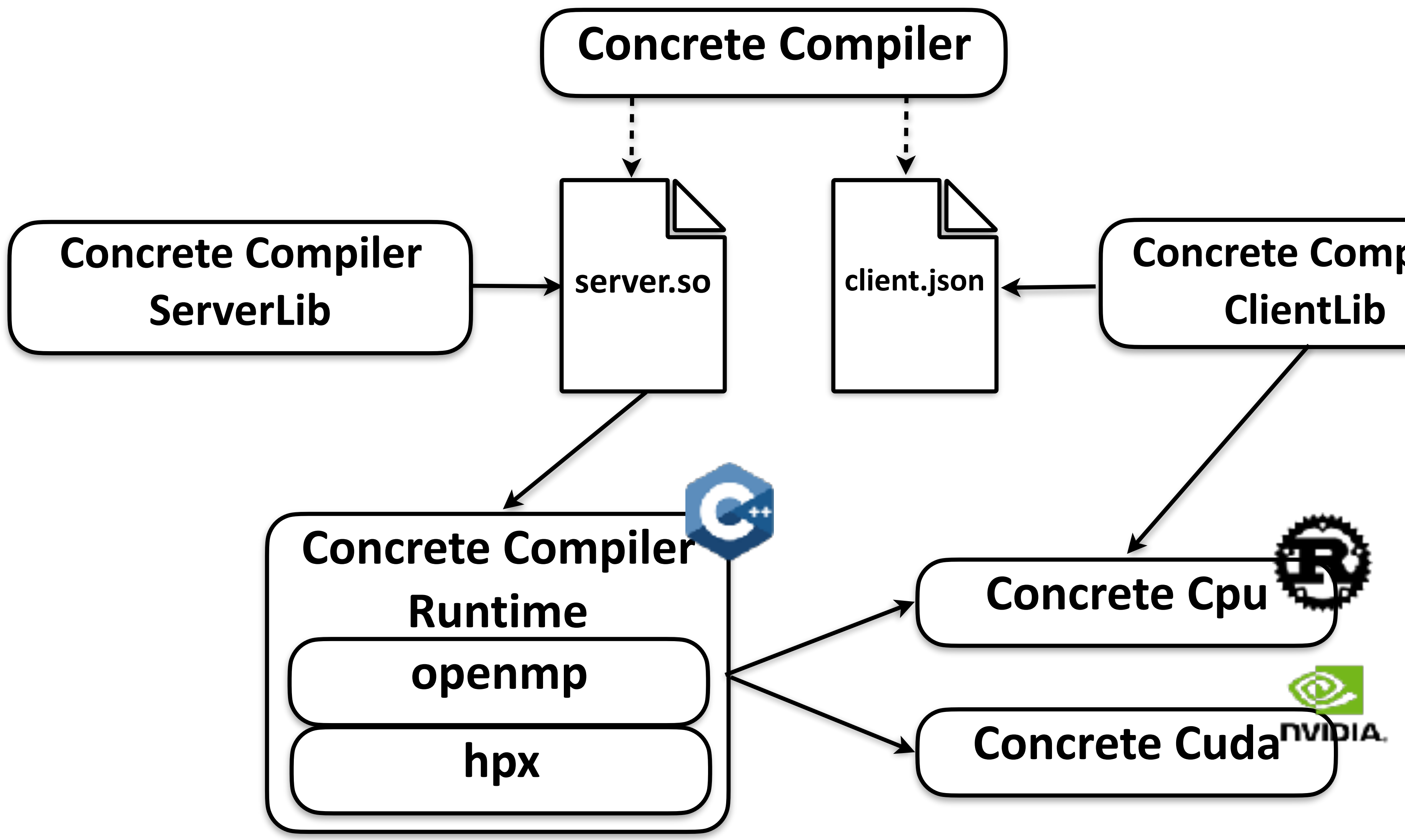
- **MLIR-**based compiler to be reusable and leverage community effort on common problems
- **Concrete Optimizer** to solve TFHE parametrization problems
- **LLVM Toolchain** to produce binary library



Concrete Compiler

High-Level Overview (2)

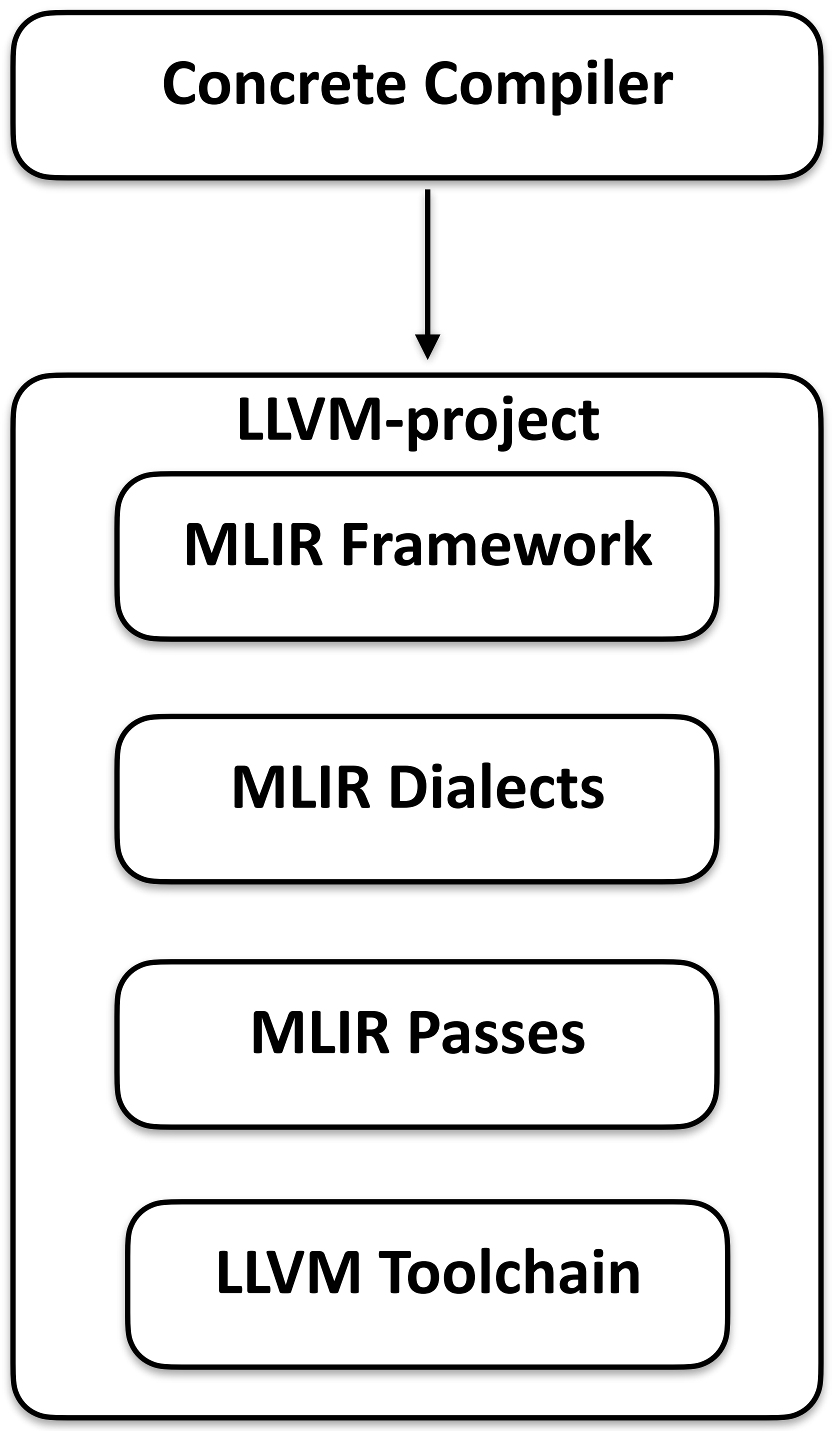
- Runtime linked with **OpenMP** and **HPX** libraries for loop parallelism and task scheduling
- Runtime linked to **Concrete CPU/GPU** backends to use the fastest hand optimized TFHE implementation
- **Client and Server toolkit** to use compilation artifacts



Concrete Compiler

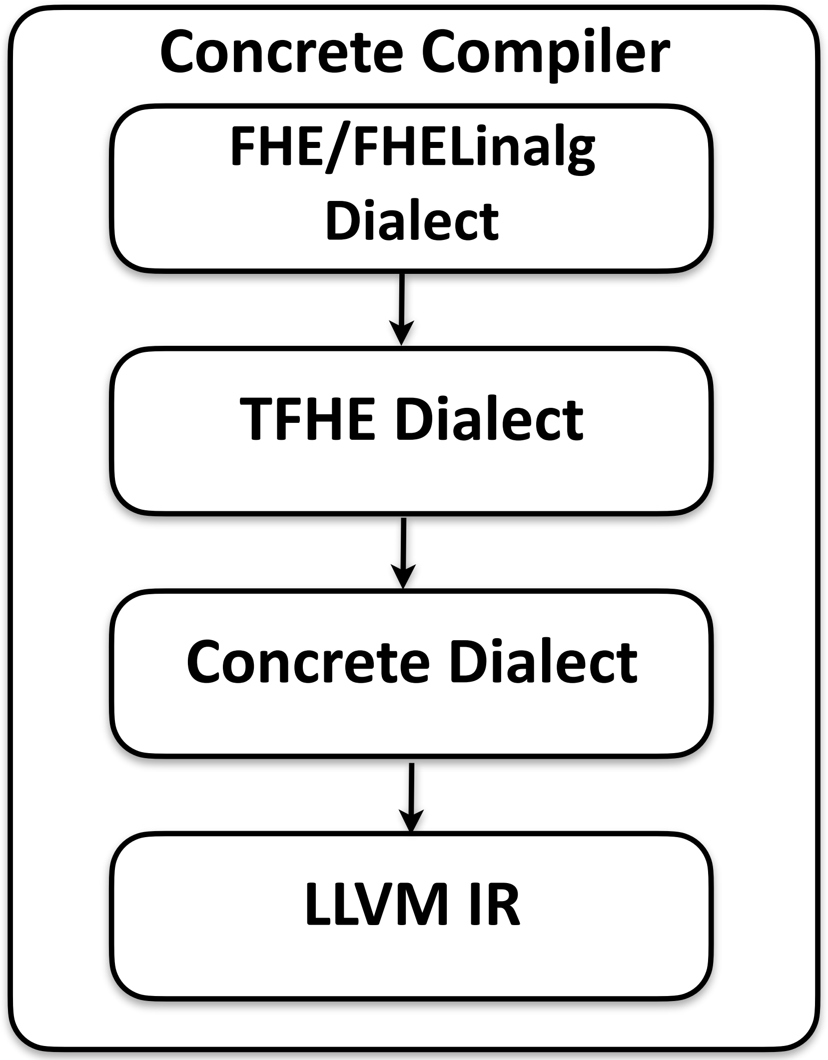
Why MLIR?

- **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




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

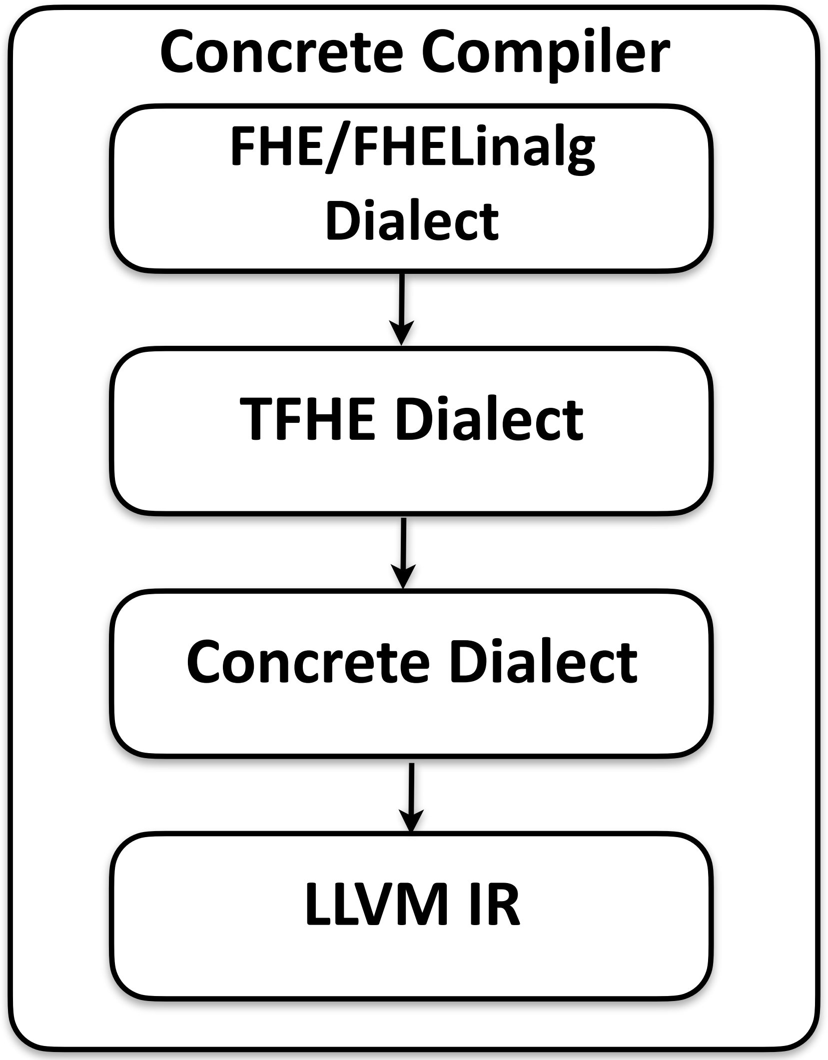
FHE Dialect defines crypto-free FHE **types and scalar operators**

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

FHELinalg Dialect defines very high level **tensor operators**

Concrete Compiler

Specific Dialects




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

TFHE Dialect introduces crypto-system dependent parameters and operators

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

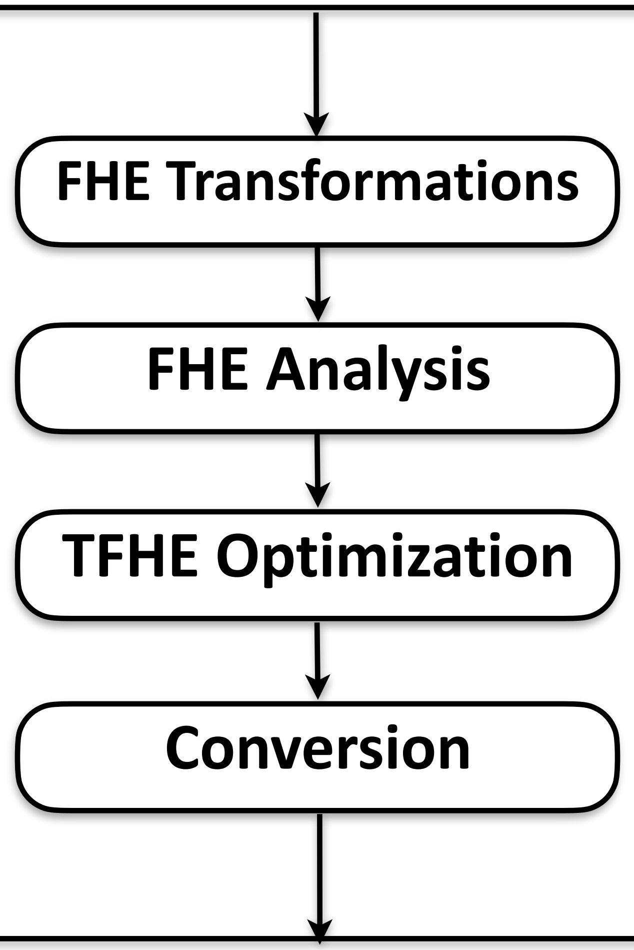
Concrete Dialect represents unabstracted implementation operators to prepare the codegen

Concrete Compiler

High Level Pipeline

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

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




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

Concrete Compiler

Automatic loop parallelism

High-level operators are lowered to **linalg.generic** with **parallel iterators**

```
%0 = "FHELinalg.apply_lookup_table"(%arg0, %arg1)
  : (tensor<2x3x4x!FHE.eint<2>>, tensor<4xi64>) -> tensor<2x3x4x!FHE.eint<2>>
```

```
%1 = linalg.generic {iterator_types = ["parallel", "parallel", "parallel"], ...} ... {
    ...
    %2 = "FHE.apply_lookup_table"(%in, %arg1) : (!FHE.eint<2>, tensor<4xi64>) -> !FHE.eint<2>
    linalg.yield %2 : !FHE.eint<2>
} -> tensor<2x3x4x!FHE.eint<2>>
```

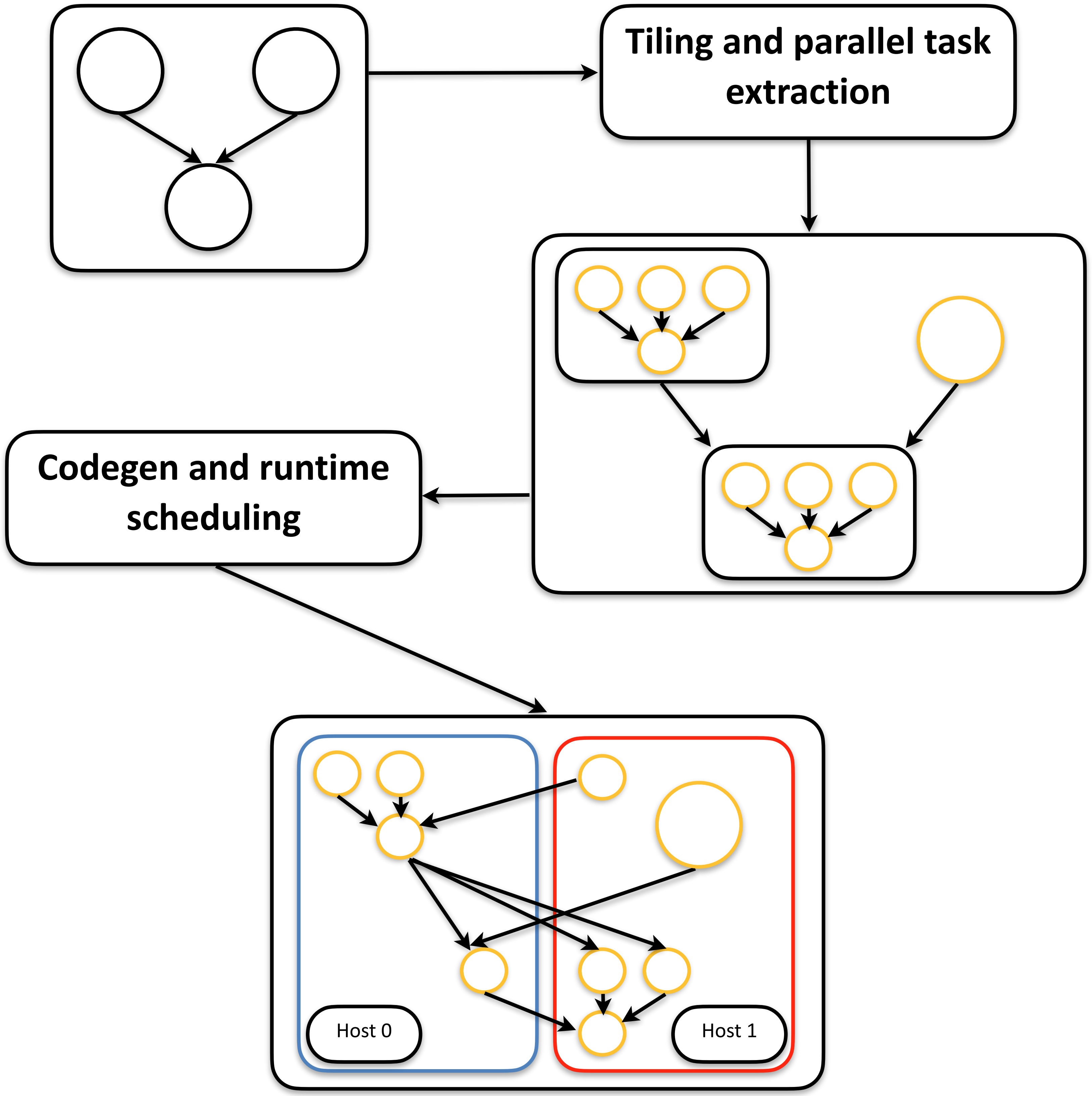
Rely on the **existing MLIR infrastructure** to generate **llvm-ir with OpenMP** annotations

```
...
omp_loop.body:                                    ; preds = %omp_loop.cond
  %9 = add i64 %omp_loop.iv, %5
  %10 = mul i64 %9, 1
  %11 = add i64 %10, 0
  br label %omp.wsloop.region

omp.wsloop.region:                                ; preds = %omp_loop.body
  %12 = call ptr @llvm.stacksave()
  br label %omp.wsloop.region3

omp.wsloop.region3:                               ; preds = %omp.wsloop.region
  %13 = srem i64 %11, 4
  %14 = sdiv i64 %11, 4
...
```

Concrete Compiler

Dataflow task parallelism

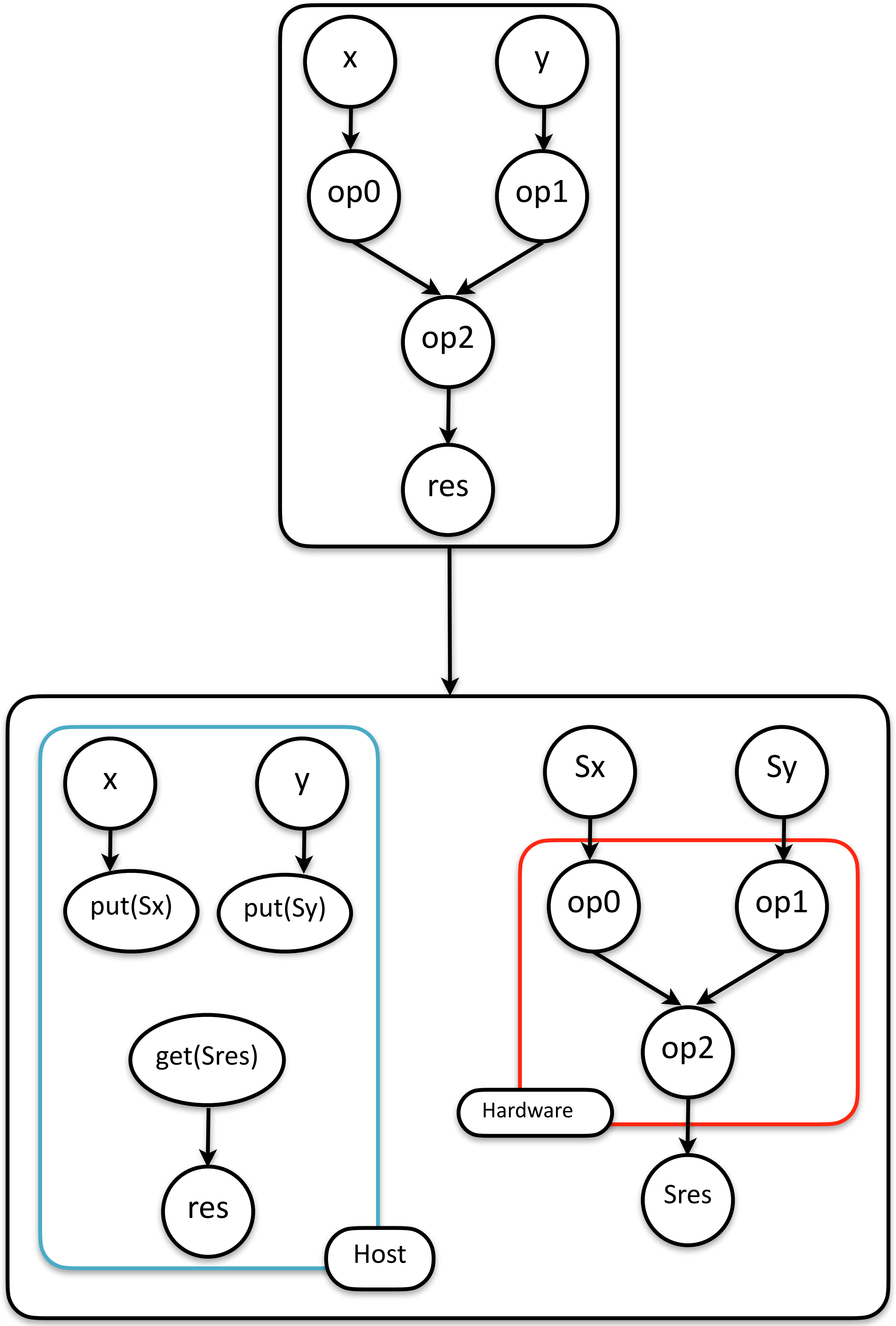
- <u>Motivation</u>: extract **parallel tasks** with dataflow analysis where loop parallelism is not available and execute tasks on **distributed systems**
- A dialect to express **high-level dataflow tasks** and **runtime abstractions**
- A set of passes **from building a dataflow task** graph **to generating code** for the dataflow runtime
- Reusing MLIR **tiling** infrastructure to expose **more parallelism** and control **granularity**
- A **HPX-based runtime** to schedule **dataflow** tasks within and across hosts



Concrete Compiler

SDFG: Static DataFlow Graph

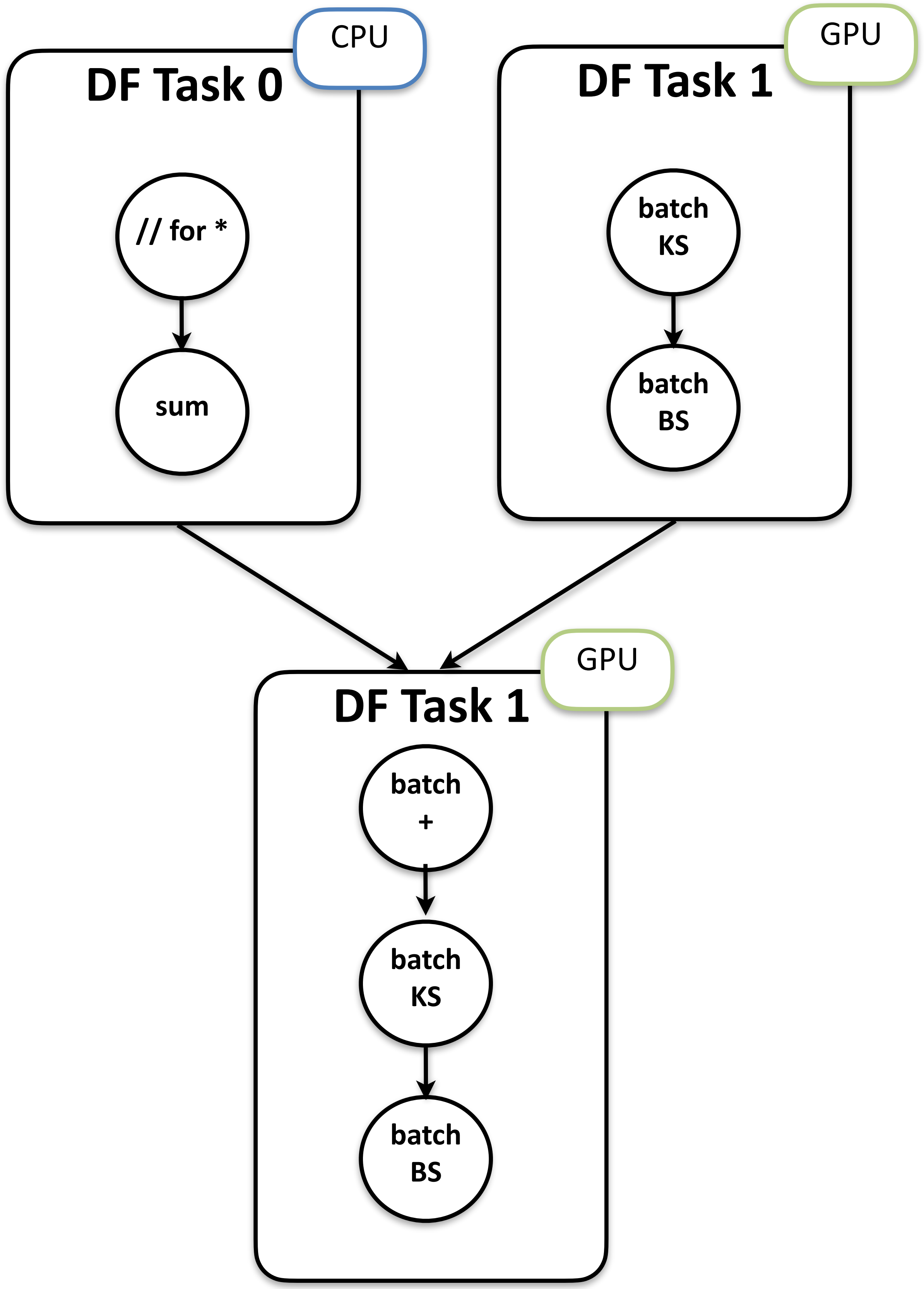
- Motivation: Efficiently **offload** a subgraph of tasks **on hardware accelerators** and maximizing data reuse on device (minimize back and forth data transfer from host to device)
- A dialect to express static dataflow graphs
- A GPU SDFG based runtime that schedules tasks on multi-GPU hosts



Concrete Compiler

Summary 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

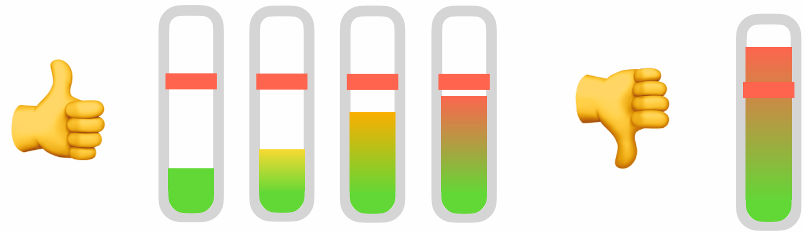
Security

$$f : (n, \lambda, q) \mapsto \sigma_{\text{enc}}$$

LWE dimension (n), Security level (λ), Ciphertext Modulus (q), Minimal standard deviation (σ_{enc})

→ Using the **lattice estimator**

Correctness



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

Efficiency

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

[APS] M. R. Albrecht, R. Player and S. Scott. On the concrete hardness of Learning with Errors. Journal of Mathematical Cryptology 2015

Choice of algorithm

LUT

LUT/function evaluation

$m \in [\![0, 2^p - 1]\!]$

$\Downarrow$

PBS

TFHE's PBS

Very efficient if $p \in [\![1,8]\!]$

WoP - PBS

BBB+'s PBS

Very efficient if $p \in [\![9,16]\!]$

Rounded PBS

TFHE's PBS on rounded input

Very efficient when applicable

[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

Σ

Plain Dot Product

$m \in [\![0, 2^p - 1]\!]$

$\Downarrow$

Σ

Dot Product

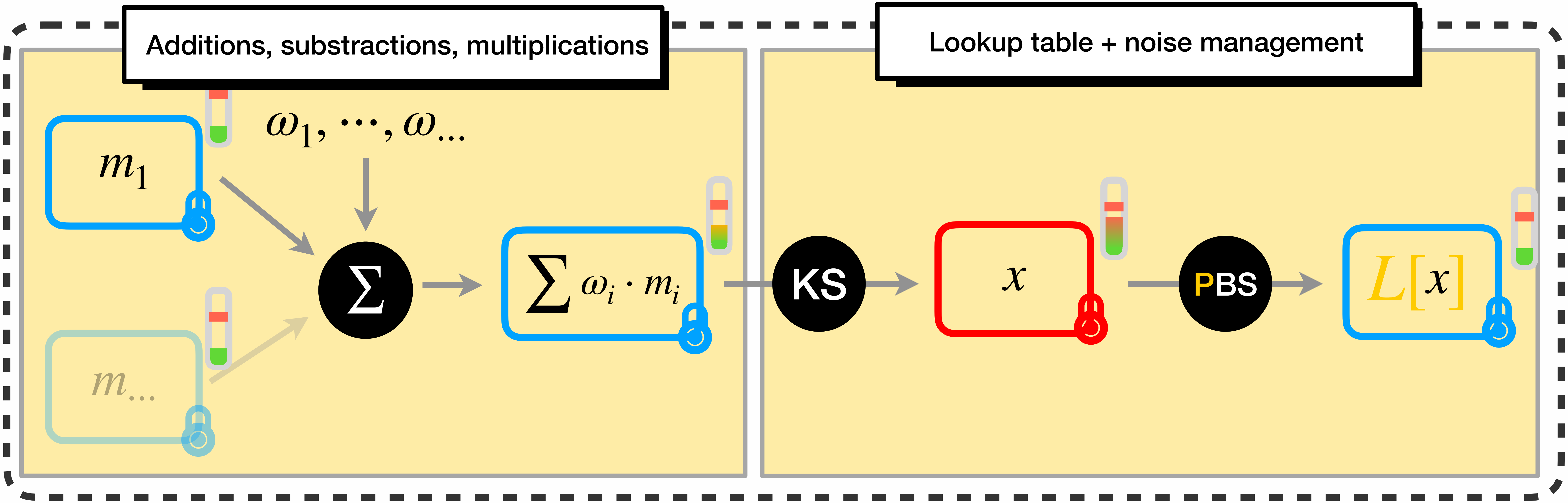
Very efficient ν is small

$\sum + \text{PBS} + \sum$

Dot Product with intra-PBS

Very efficient if ν is big

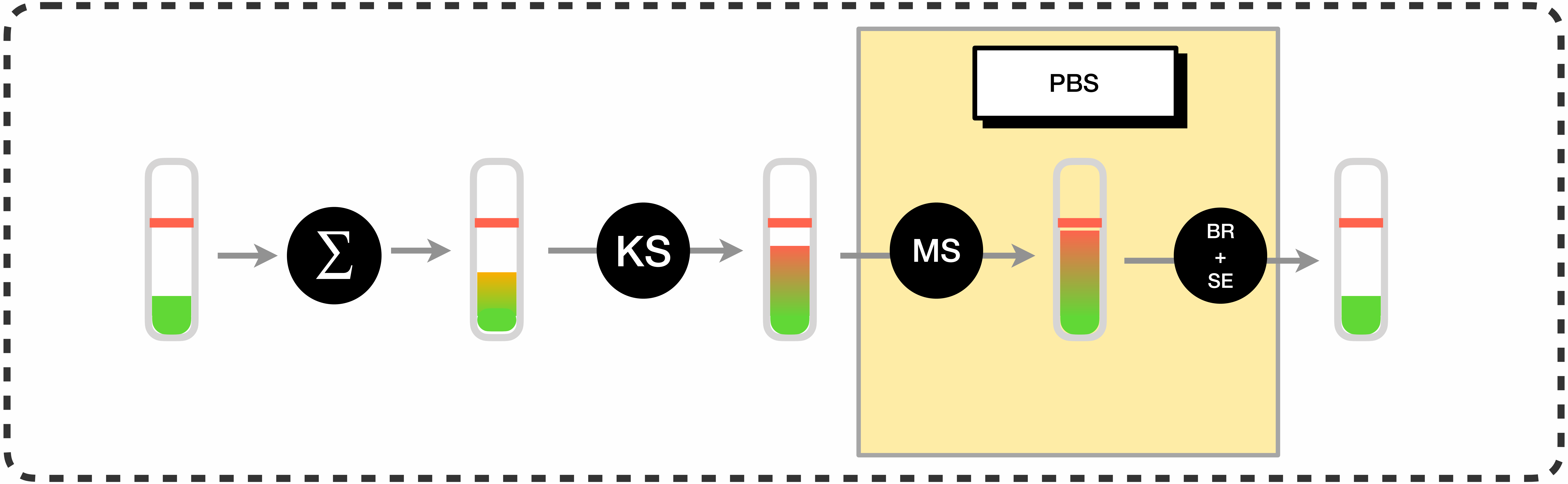
Translation



DAG of FHE Operators $\Longrightarrow$ DAG of TFHE Operators $\Longrightarrow$ DAG of atomic patterns

[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

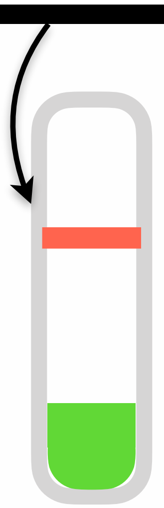
Noise analysis of an Atomic Pattern



Noise is increasing between two modulus switching

Noise Bound

Noise Bound t is a function of



message precision

encoding

Natif, CRT, Radix

failure probability

DAG dependent

Optimization Problem

up to a given p_{fail}

$$\min \; \text{Cost}\left(\mathcal{G}\right) \quad \text{s.t.} \begin{cases} \forall i \in I, \; \text{Noise}\left(\text{MS}_i\right) \leq t \\ \sigma_{\text{enc}} = f\left(n, \lambda, q\right) \end{cases}$$

λ bits of security

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 designed 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

ZAMA