HEIR: A foundation for FHE compilers

Jeremy Kun
Google
2022-10-19

Agenda

01 One slide pitch

02 Why, why, why?

03 MLIR overview

04 What we're working on in HEIR

05 Resources

01

One slide pitch

We're building a compiler toolchain

with all the established ideas in FHE

to accelerate FHE research

and lower the barrier to production FHE

Google

Using the MLIR framework

We're building a compiler toolchain

with all the established ideas in FHE

to accelerate FHE research

and lower the barrier to production FHE

Using the MLIR framework

We're building a compiler toolchain

Every FHE scheme, fast polynomial math, optimizations, noise models, ...

with all the established ideas in FHE

to accelerate FHE research

and lower the barrier to production FHE

Using the MLIR framework

We're building a compiler toolchain

Every FHE scheme, fast polynomial math, optimizations, noise models, ...

with all the established ideas in FHE

Focus on your novel idea, not compiler infra

to accelerate FHE research

and lower the barrier to production FHE

Using the MLIR framework

We're building a compiler toolchain

Every FHE scheme, fast polynomial math, optimizations, noise models, ...

with all the established ideas in FHE

Focus on your novel idea, not compiler infra

to accelerate FHE research

Benchmarking, hardware targets, frontends (C++, Python, Tensorflow)

and lower the barrier to production FHE

Working group!

- github.com/google/heir
- Open design meeting every two weeks
- Participants from across industry and academia
  - Bigco: Google, Intel
  - Startups: Zama, Cryptolab
  - University: ETH Zurich, KU Leuven
  - Hardware startups: Optalysys, Niobium
  - **you?**

📆 Meeting calendar 📆

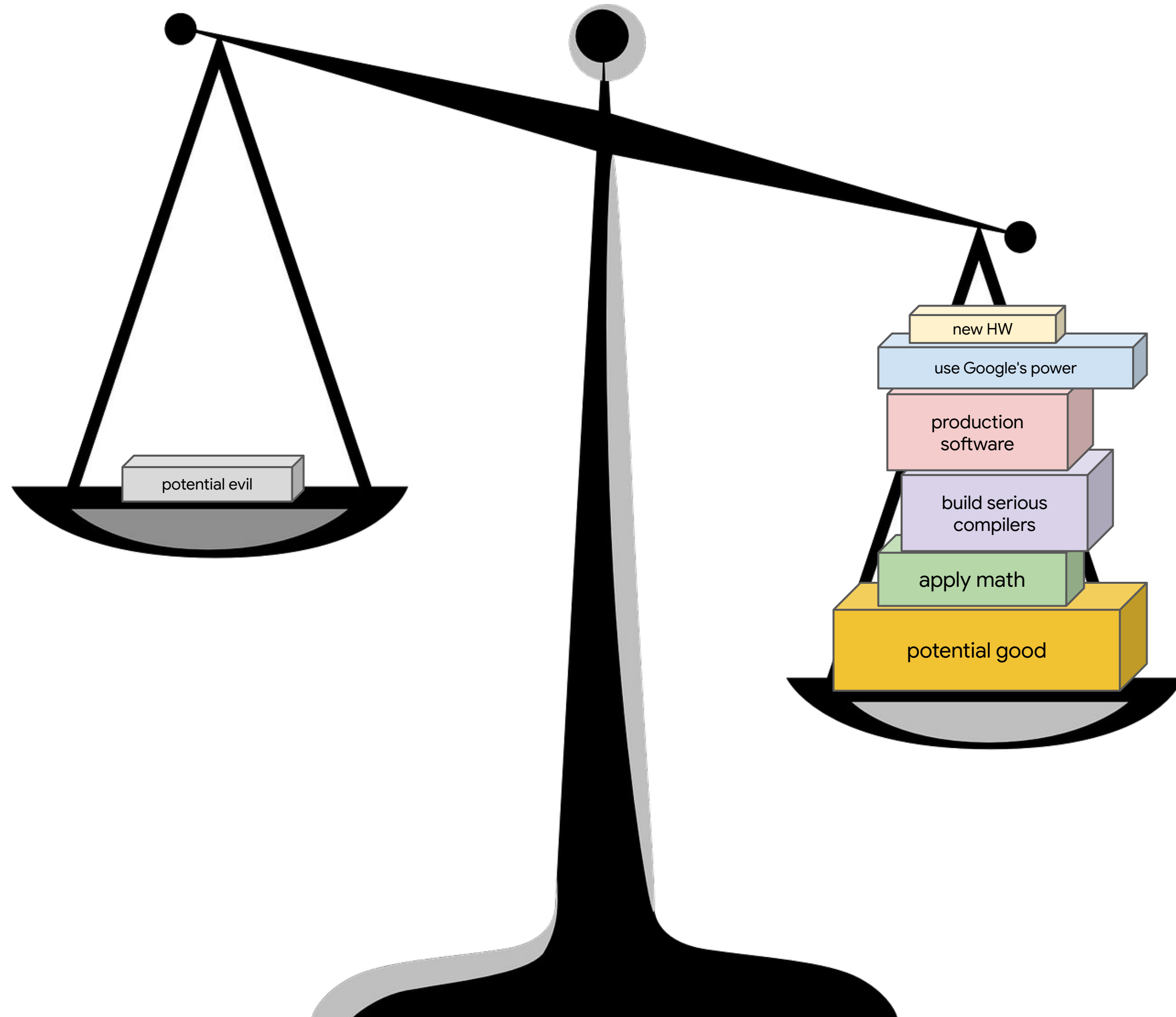
02

Why, why, why?

Why do I care about FHE?

- Undergrad in CS + math
- PhD in theoretical computer science
- Math blogging for 13 years (jeremykun.com)
- "Get paid to apply cool math to software"

- How do you choose a *meaningful* problem?
- How *best* to contribute?
- How to grow *and* leverage your strengths?
- When is the *right* time?



potential evil
new HW
use Google's power
production software
build serious compilers
apply math
potential good

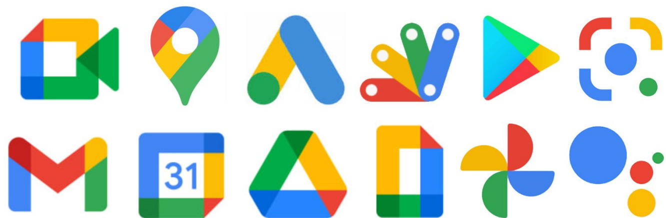
Why does Google care about FHE?

Increasing regulatory pressure

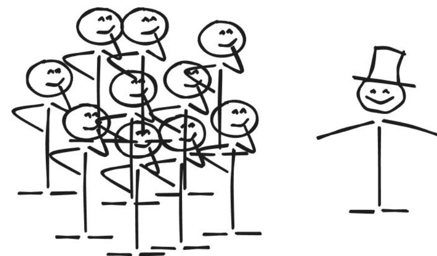
Many products could use it

Engineers want to do the right thing

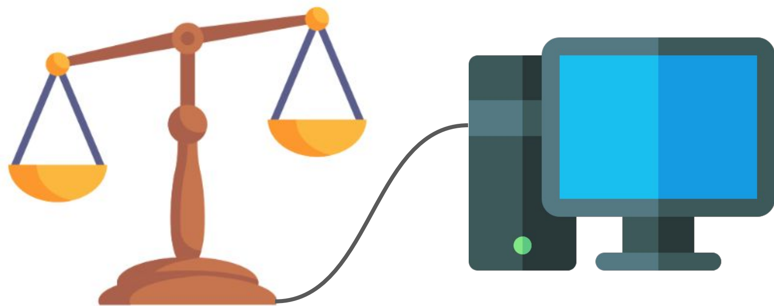
Why do we need FHE compilers?



Lots of legacy C++



Crypto expertise won't scale



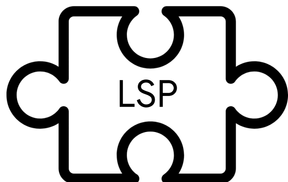
For general purpose, tradeoffs need to be machine-assisted or automated

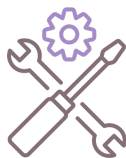
Need abstractions to manage {FHE schemes} x {perf tricks} × {HW targets}

Why MLIR?

Many production-strength compilers already built with it

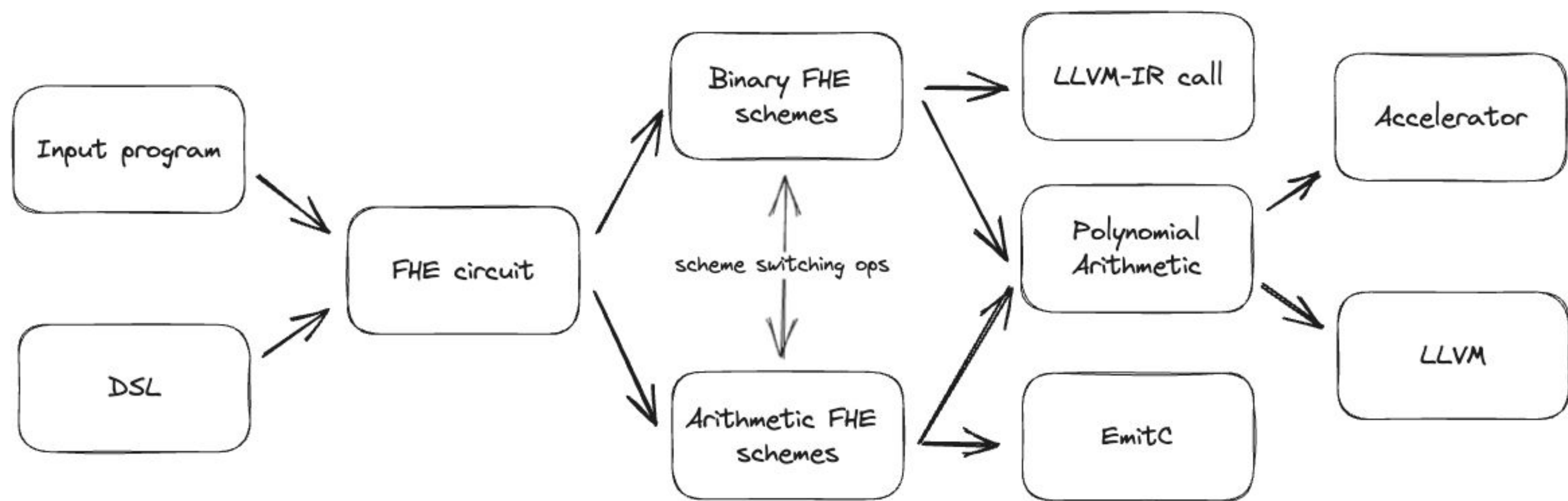
Backed by world class compiler engineers across the industry



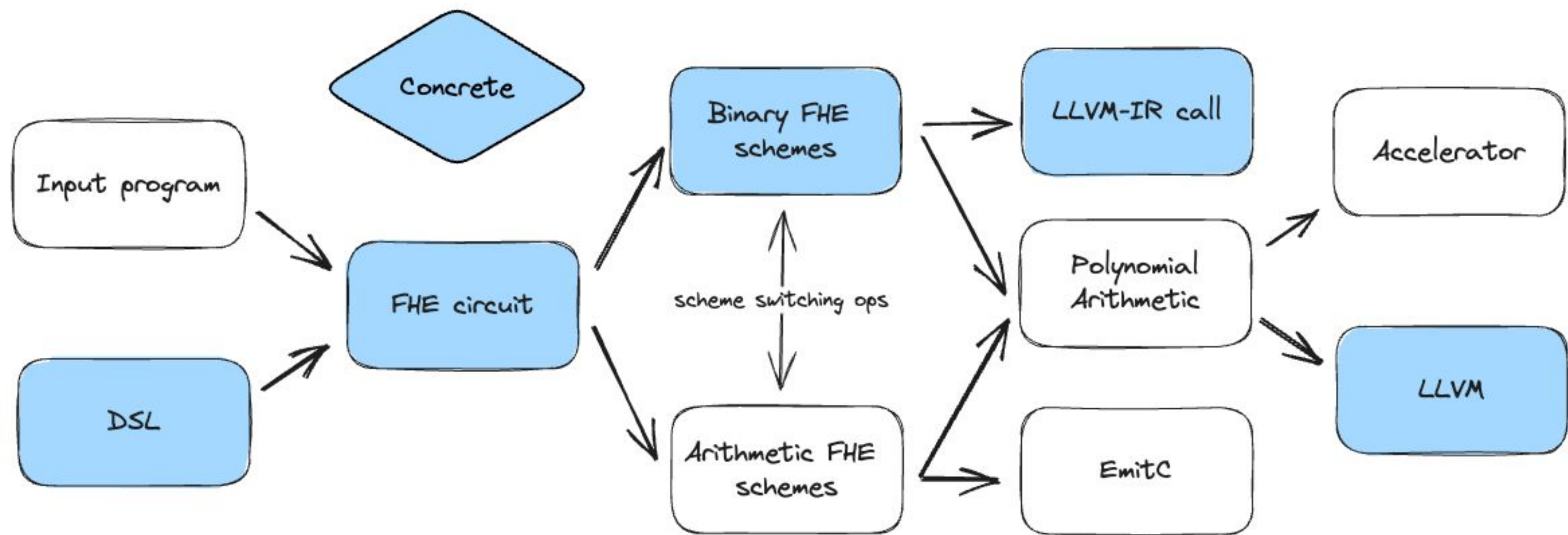


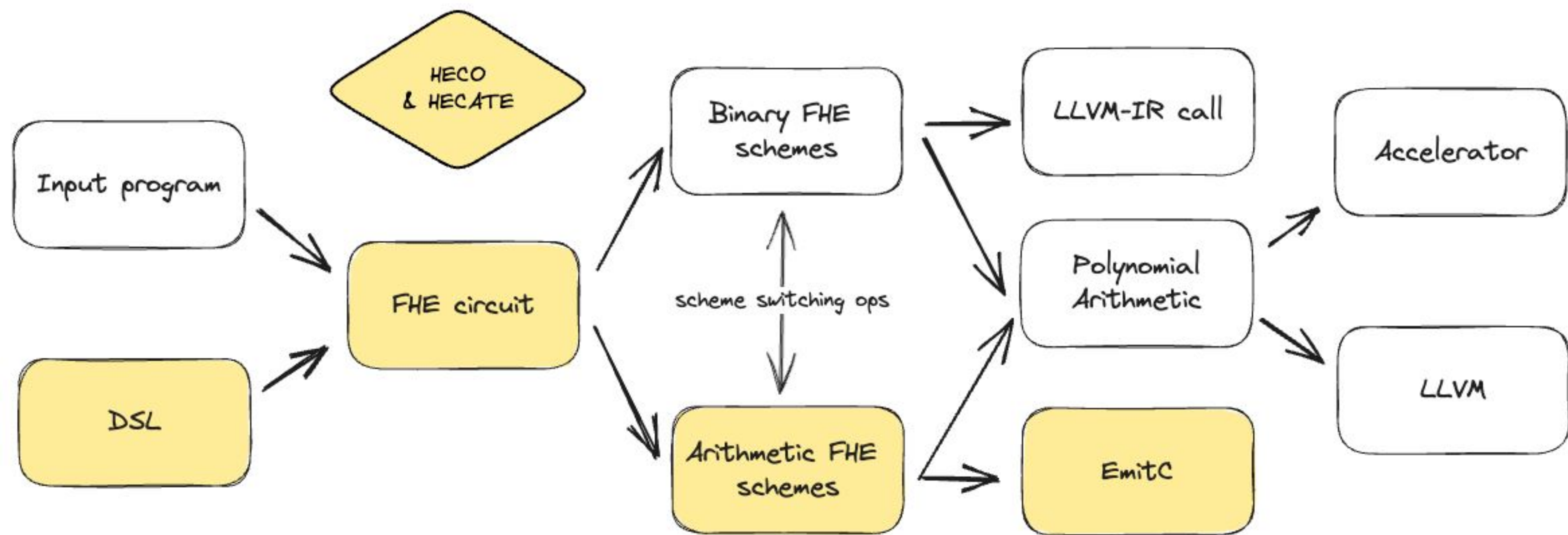
Pre-existing tooling

New languages are expensive, require rewrites

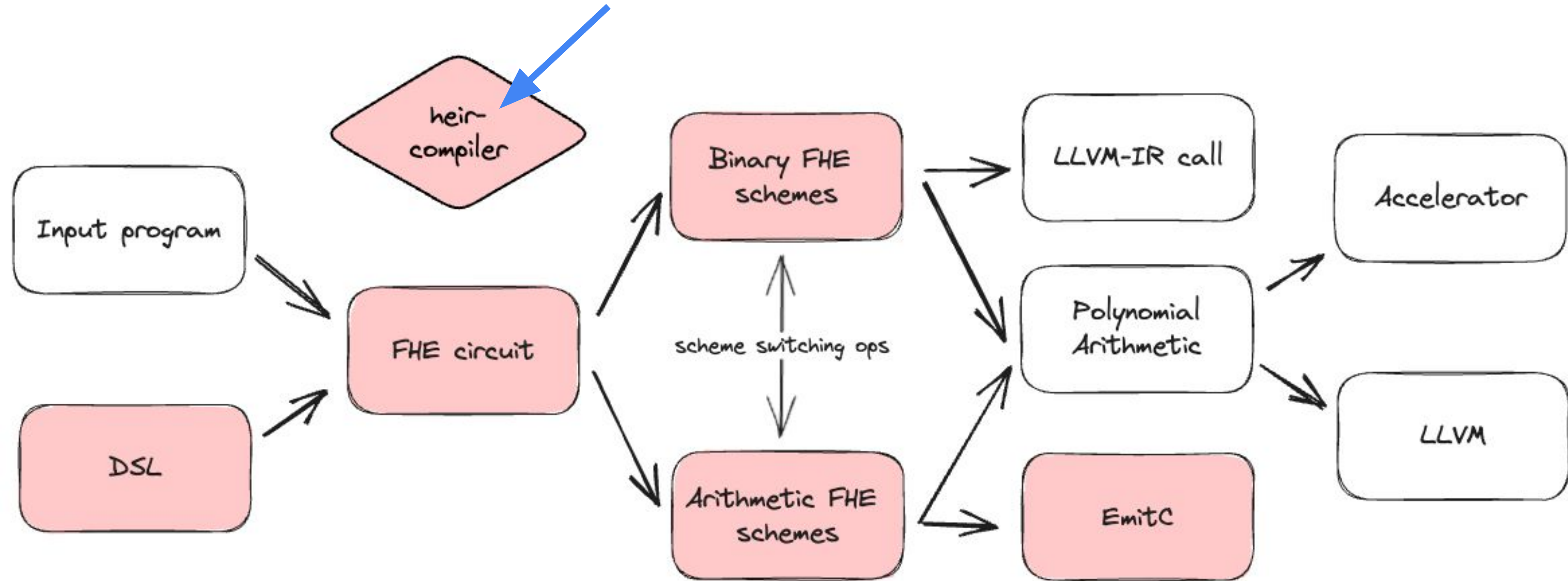


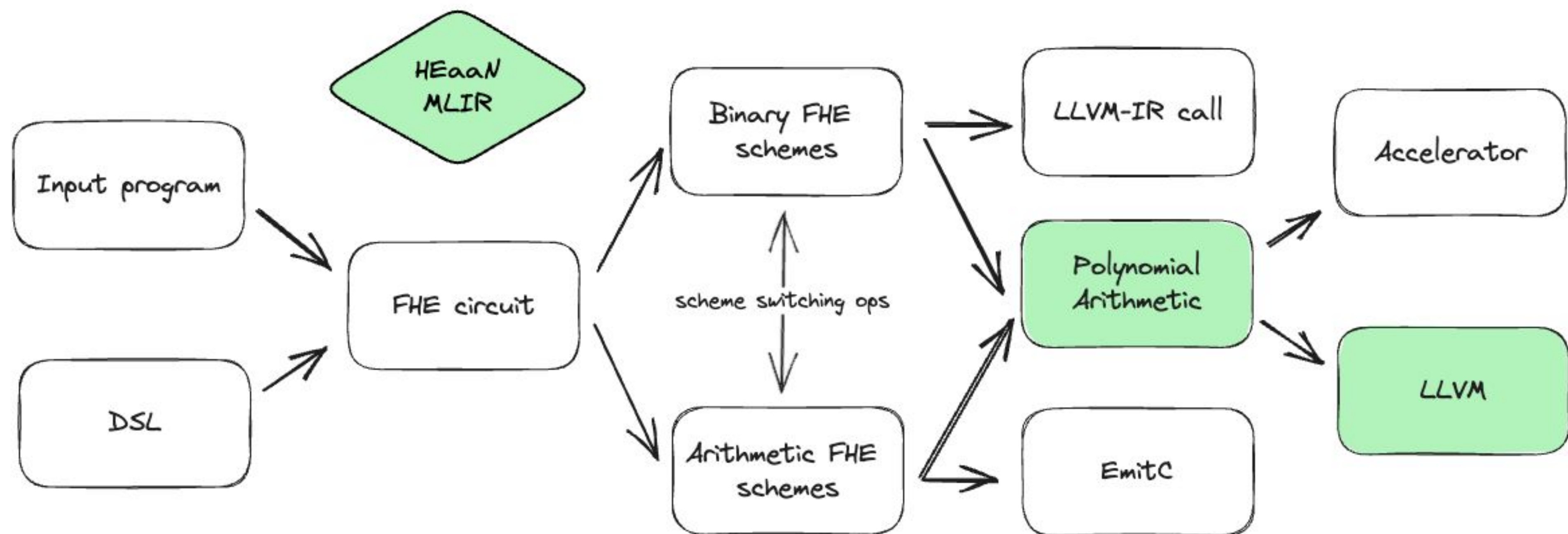
Input program
DSL
FHE circuit
Binary FHE schemes
scheme switching ops
Arithmetic FHE schemes
LLVM-IR call
Polynomial Arithmetic
EmitC
Accelerator
LLVM


Concrete
Input program
DSL
FHE circuit
Binary FHE schemes
scheme switching ops
Arithmetic FHE schemes
LLVM-IR call
Polynomial Arithmetic
EmitC
Accelerator
LLVM


HECO
& HECATE
Input program
DSL
FHE circuit
Binary FHE schemes
scheme switching ops
Arithmetic FHE schemes
LLVM-IR call
Polynomial Arithmetic
EmitC
Accelerator
LLVM

Unlucky name conflict!




HEaaN
MLIR
Input program
DSL
FHE circuit
Binary FHE
schemes
scheme switching ops
Arithmetic FHE
schemes
LLVM-IR call
Polynomial
Arithmetic
EmitC
Accelerator
LLVM

What's the catch?

- Steep learning curve
  - MLIR/LLVM is a sprawling C++ codebase
  - Compiler jargon unfamiliar to cryptographers
  - The MLIR documentation needs work
- Some parts of MLIR are unstable
  - Dialects being split into smaller pieces
  - We can hopefully protect you from the fallout
- MLIR community tends to implement just what they need
- Generality, production-strength means development will naturally be slower
  - The right abstractions will need to make sense for all of FHE

03

MLIR & Technical Details

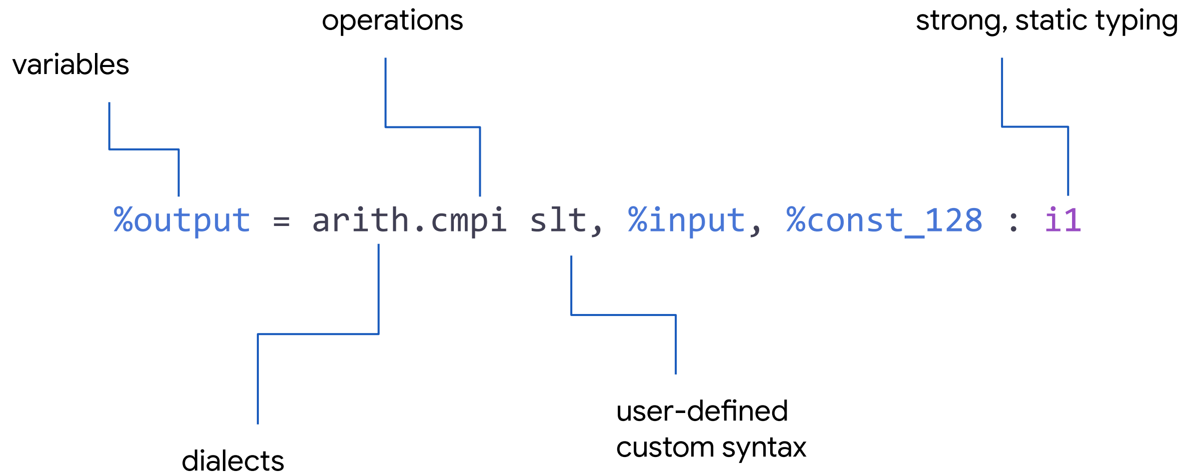
MLIR: Multi Level Intermediate Representation

History

- A software library for building compilers
- Unify TensorFlow ecosystem at Google
- "Batteries included" optimizations and backend targets
- Extensible: Fortran, CIRCT, Quantum compiler, Mojo

MLIR: Multi Level Intermediate Representation

Common Syntax, basic building blocks



MLIR: Multi Level Intermediate Representation

Higher level constructions built in

```
func.func bar(%A : memref<8x?xf32>) {
  %N = memref.dim %A, 1 : memref<8x?xf32>
  affine.for %i = 0 to 8 {
    affine.for %j = 0 to %N {
      // A[i,j] += 1
      %c1 = arith.constant 1.0 : f32
      %s1 = affine.load %A[%i, %j] : memref<8x?xf32>
      %s2 = add %s1, %c1
      affine.store %s2, %A[%i, %j] : memref<8x?xf32>
    }
  }
  return
}
```

MLIR: Multi Level Intermediate Representation

Passes and interfaces

```
func.func bar(%A : memref<8x?xf32>) {
  %N = memref.dim %A, 1 : memref<8x?xf32>
  affine.for %i = 0 to 8 {
    affine.for %j = 0 to %N {
      // A[i,j] += 1
      %c1 = arith.constant 1.0 : f32
      %s1 = affine.load %A[%i, %j] : memref<8x?xf32>
      %s2 = add %s1, %c1
      affine.store %s2, %A[%i, %j] : memref<8x?xf32>
    }
  }
  return
}
```

`loop-invariant-code-motion`

`AlwaysSpeculatable`
`MemoryEffectOpInterface`

MLIR: Multi Level Intermediate Representation

Easy to do many local rewrites

```
def HasOneUse: Constraint<CPred<"$_self.hasOneUse()">,
                                "has one use">;

// Rewrites (x^2 - y^2) as (x+y)(x-y)
def DifferenceOfSquares : Pattern<
  (SubOp (MulOp:$lhs $x, $x), (MulOp:$rhs $y, $y)),
  [
    (AddOp:$sum $x, $y),
    (SubOp:$diff $x, $y),
    (MulOp:$res $sum, $diff),
  ],
  [(HasOneUse:$lhs), (HasOneUse:$rhs)]
>;
```

MLIR: Multi Level Intermediate Representation

Global analysis & optimization (cost models)

```
affine-super-vectorize
scf-for-loop-specialization
scf-parallel-loop-specialization
sparse-vectorization

affine-loop-fusion
affine-loop-coalescing
affine-loop-tile
affine-loop-unroll-jam
affine-loop-parallelize
```

MLIR: Multi Level Intermediate Representation

Incremental lowering down to GPU, LLVM

- -arm-neon-2d-to-intr
- -convert-affine-for-to-gpu
- -convert-amdgpu-to-rocdl
- -convert-arith-to-amdgpu
- -convert-arith-to-llvm
- -convert-arith-to-spirv
- -convert-arm-sme-to-scf
- -convert-async-to-llvm
- -convert-bufferization-to-memref
- -convert-cf-to-llvm
- -convert-cf-to-spirv
- -convert-complex-to-libm
- -convert-complex-to-llvm
- -convert-complex-to-spirv
- -convert-complex-to-standard
- -convert-func-to-llvm
- -convert-func-to-spirv
- -convert-gpu-launch-to-vulkan-launch
- -convert-gpu-to-nvvm
- -convert-gpu-to-rocdl
- -convert-gpu-to-spirv
- -convert-index-to-llvm
- -convert-linalg-to-std
- -convert-math-to-funcs
- -convert-math-to-libm
- -convert-math-to-llvm
- -convert-math-to-spirv
- -convert-memref-to-spirv
- -convert-nvgpu-to-nvvm
- -convert-nvvm-to-llvm
- -convert-openacc-to-scf
- -convert-openmp-to-llvm
- -convert-parallel-loops-to-gpu
- -convert-scf-to-cf
- -convert-scf-to-emitc
- -convert-scf-to-openmp

…

MLIR: Multi Level Intermediate Representation

Incremental lowering down to GPU, LLVM

Payload

Structure

Tensor

Buffer

Dialect

New/Split/Previously Omitted Dialect

External Dialect

External Format

Sparse Tensor

AMDGPU

Quant

Builtin

Unplaced

EmitC

Output Utility

Transform

DLTI

ML Program

Work In Progress

PyTorch ATen

TFLite

TF

HLO

MHLO

Torch-MLIR

TOSA

Complex

Math

Linalg

LMHLO

Shape

Utility

Tensor

Affine

Func

Arith

Bufferization

Vector

ARM-Neon

?

ARM-SVE

Memref

NVGPU

SCF

Async

GPU

OpenMP

OpenACC

CF

AMX

SPIR-V

LLVM (CFG)

PDL

PDL-interp

System

LLVM (core)

AVX512 (x86vector)

ROCm (ROCDL)

NVVM

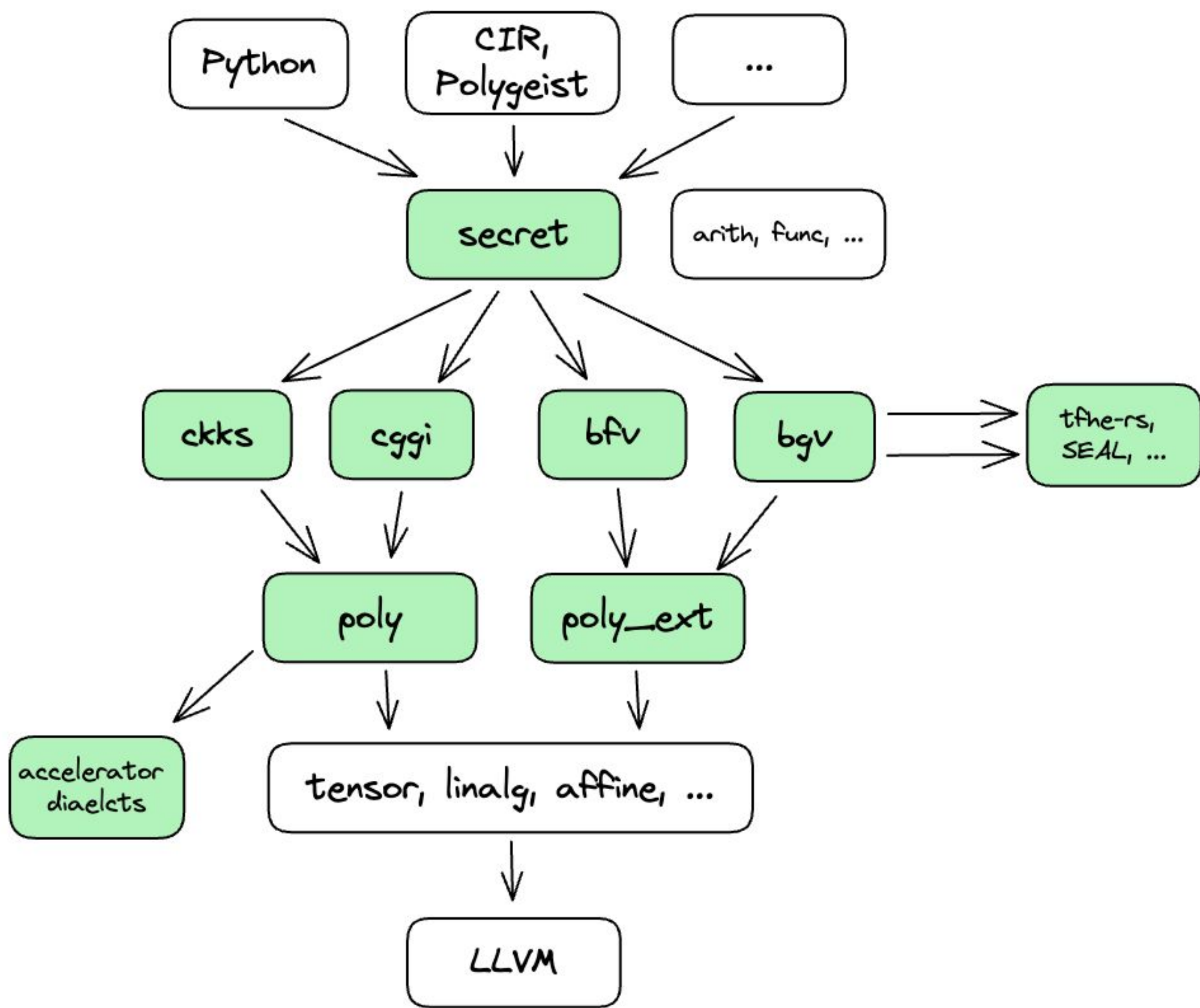
LLVM-IR

SPIR-V

04

What we're working on in HEIR

Warning: nothing is finished yet!



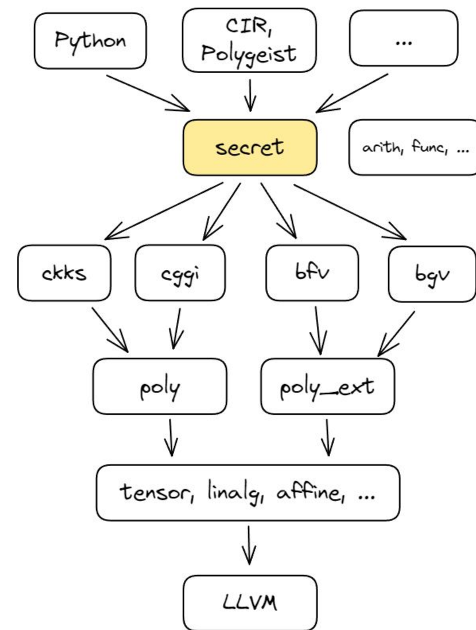
Google

Abstracted private computation

```
!ei32 = !secret.secret<i32>

// Problem:
%0 = ... : !ei32
%1 = ... : !ei32
%2 = ???.add : (!ei32, !ei32) -> !ei32
```

`arith.addi` can't be used

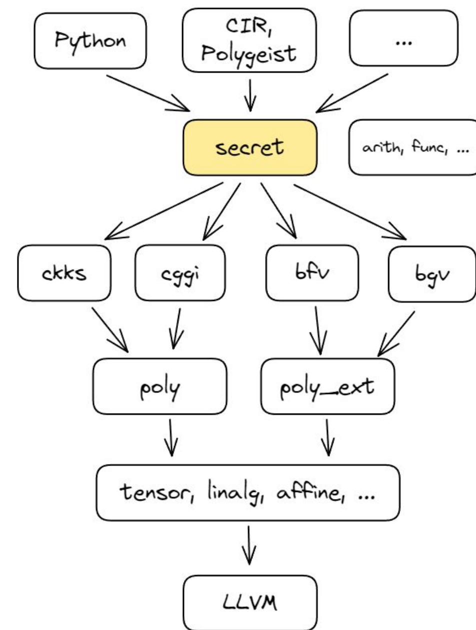
"Shadow" arith dialect => more work, harder to reuse existing MLIR passes



Abstracted private computation

```
%Z = secret.generic ins(%value, %cond : !secret.secret<i32>, i1) {
  ^bb0(%clear_value: i32, %clear_cond: i1):
    // computes (7x + 1)^2 if cond, else 0
    %c0 = arith.constant 0 : i32
    %c1 = arith.constant 1 : i32
    %c7 = arith.constant 7 : i32
    %0 = arith.muli %clear_value, %c7 : i32
    %1 = arith.addi %0, %c1 : i32
    %2 = arith.muli %1, %1 : i32
    %3 = arith.select %clear_cond, %2, %c0 : i32
    secret.yield %3 : i32
  } -> (!secret.secret<i32>)
  func.return %Z : !secret.secret<i32>
}
```

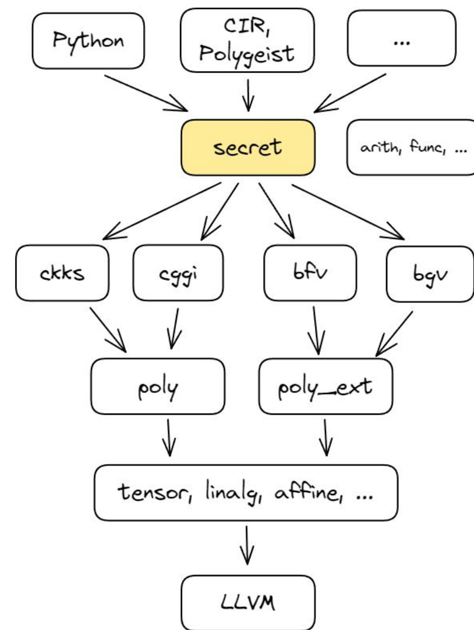
"wrap" a cleartext computation



Abstracted private computation

```
%c0_i32 = arith.constant 0 : i32
%c1_i32 = arith.constant 1 : i32
%c7_i32 = arith.constant 7 : i32
%0 = secret.generic ins(%arg0, %c7_i32 : !secret.secret<i32>, i32) {
^bb0(%arg2: i32, %arg3: i32):
  %4 = arith.muli %arg2, %arg3 : i32
  secret.yield %4 : i32
} -> !secret.secret<i32>
%1 = secret.generic ins(%0, %c1_i32 : !secret.secret<i32>, i32) {
^bb0(%arg2: i32, %arg3: i32):
  %4 = arith.addi %arg2, %arg3 : i32
  secret.yield %4 : i32
} -> !secret.secret<i32>
%2 = secret.generic ins(%1, %1 : !secret.secret<i32>, !secret.secret<i32>) {
^bb0(%arg2: i32, %arg3: i32):
  %4 = arith.muli %arg2, %arg3 : i32
  secret.yield %4 : i32
} -> !secret.secret<i32>
%3 = secret.generic ins(%arg1, %c0_i32, %2 : i1, i32, !secret.secret<i32>) {
^bb0(%arg2: i1, %arg3: i32, %arg4: i32):
  %4 = arith.select %arg2, %arg4, %arg3 : i32
  secret.yield %4 : i32
} -> !secret.secret<i32>
```

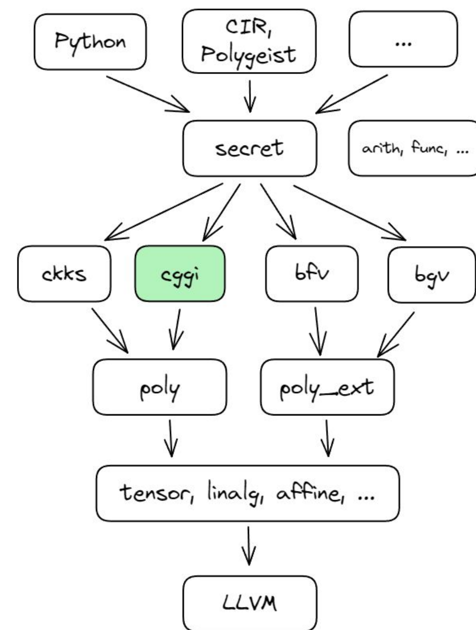
Reduce to minimal private instructions, extract cleartext computation.



Scheme dialects + building a circuit

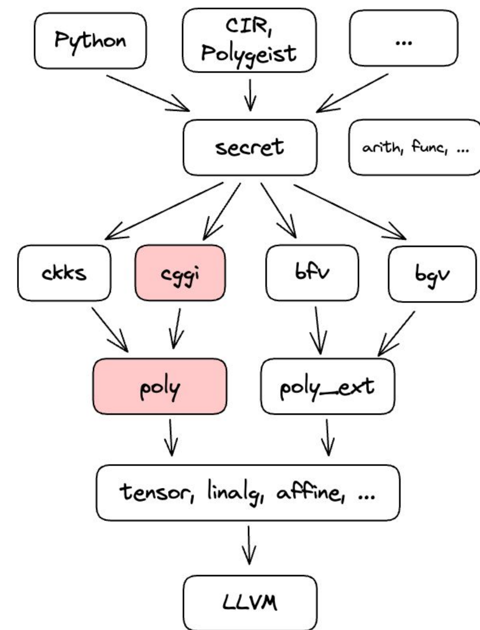
```
  %0 = ... : !cggi.ciphertext<i4>
  %1 = ... : !cggi.ciphertext<i4>

  // cheeky trick:
  // x*y = ((x+y)**2 - (x-y)**2) / 4
  %lut = cggi.lut <...> // a look-up table for x^2
* %sum = cggi.add %0, %1
* %diff = cggi.sub %0, %1
  %sum_sq = cggi.programmable_bootstrap %sum, %lut
  %diff_sq = cggi.programmable_bootstrap %diff, %lut
* %diff_diff = cggi.sub %sum_sq, %diff_sq
* %res = cggi.div_scalar %diff_diff, %c4
```

* adds noise



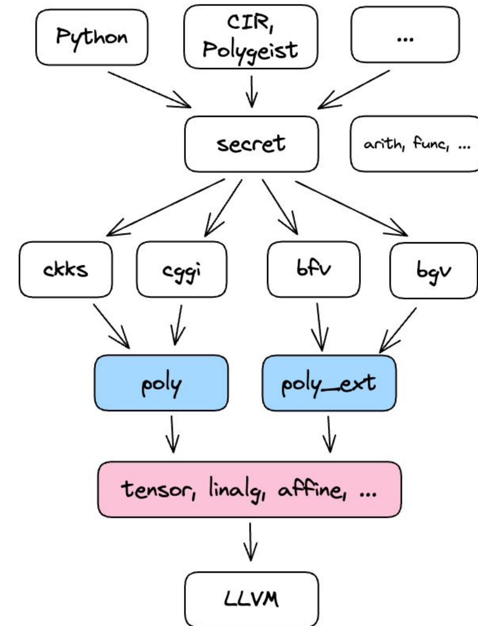
FHE scheme dialects

```
func.func @blind_rotate(
        %lut_index: !lwe.lwe_ciphertext<i32, 720>,
        %lut: !lwe.rlwe_ciphertext<i32, 2, !poly_ty>,
        %bsk: tensor<720x!cggi.rgsw_ciphertext<...>>) {
    %n = lwe.dim %lut_index : index
    %b = lwe.extract %lut_index[%n]
    %minus_b = arith.muli %c-1, %b
    %init_monomial = poly.monomial %c1, %minus_b
    %init_accum = poly.monomial_mul %init_monmial, %lut
    %res = scf.for %iv = %0 to %n step 1 iter_args(%sum_iter = %init_accum) {
        %init_monomial = poly.monomial %c1, %minus_b
        %a = lwe.extract %lut_index[%iv]
        %monomial = poly.monomial %c1, %a
        %opt1 = poly.monomial_mul %sum_iter, %monomial
        %bsk_item = tensor.extract[%iv]
        %sum_next = cggi.cmux %bsk_item, %sum_iter, %opt1
        scf.yield %sum_next
    }
}
```



Fast polynomial arithmetic

```
#poly = #poly.polynomial<1 + x**1024>
#ring = #poly.ring<cmod=2**32, ideal=#poly>
!poly_ty = !poly.poly<#ring>

%1 = poly.constant <-1 + x**1023> : !poly_ty
%2 = poly.mul %1, %1 : !poly_ty
%coeff, %deg = poly.leading_term %2: !poly_ty -> (index, i32)
%3 = poly.fft %2 : !poly_ty -> tensor<1024xi32>
```



Fast polynomial arithmetic

Security tradeoff

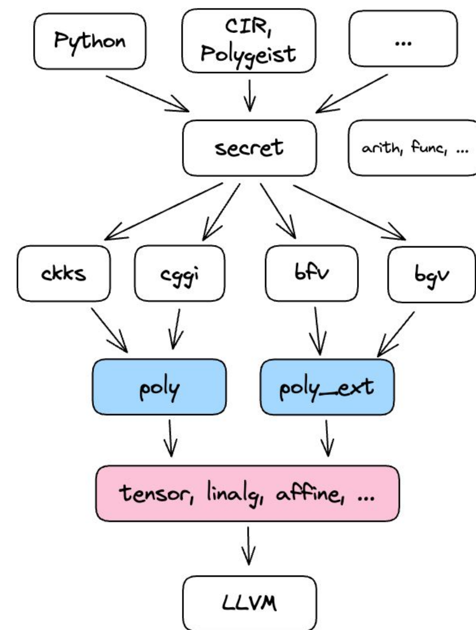
$$(\mathbb{Z}/n\mathbb{Z})[x]/(x^N + 1)$$

Machine word size vs large prime
"NTT friendly"
RNS/bigint decomp

Security critical

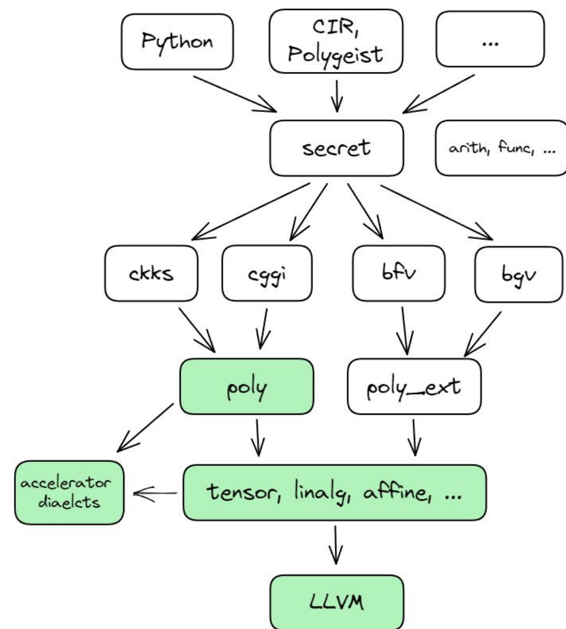
$$R[x]/(p(x))$$

Enough interest to upstream it to MLIR:
https://discourse.llvm.org/t/rfc-a-poly-dialect-for-polynomial-arithmetic/73891



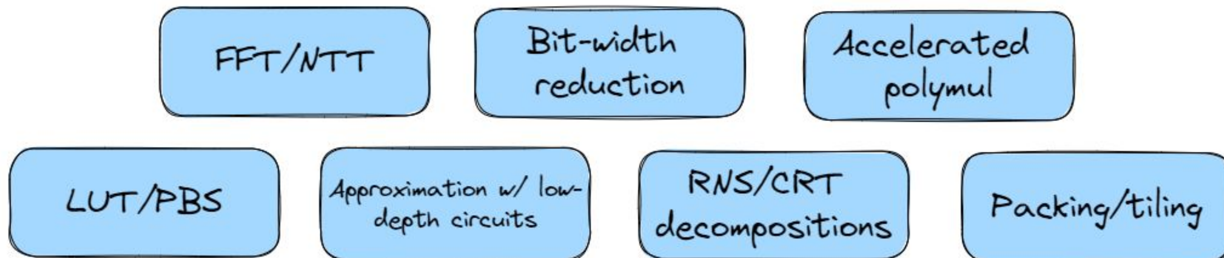
HW dialects

- MLIR: affine loop optimizations, vectorizer, x86 & GPU
- Already serious benefits on CPU just by going through LLVM
- Per-platform dialects:
  - Optalysys
  - FPGA (KU Leuven, Niobium, et al)
  - TPU (Google)
- Static scheduling
- Benchmarks!



Big questions

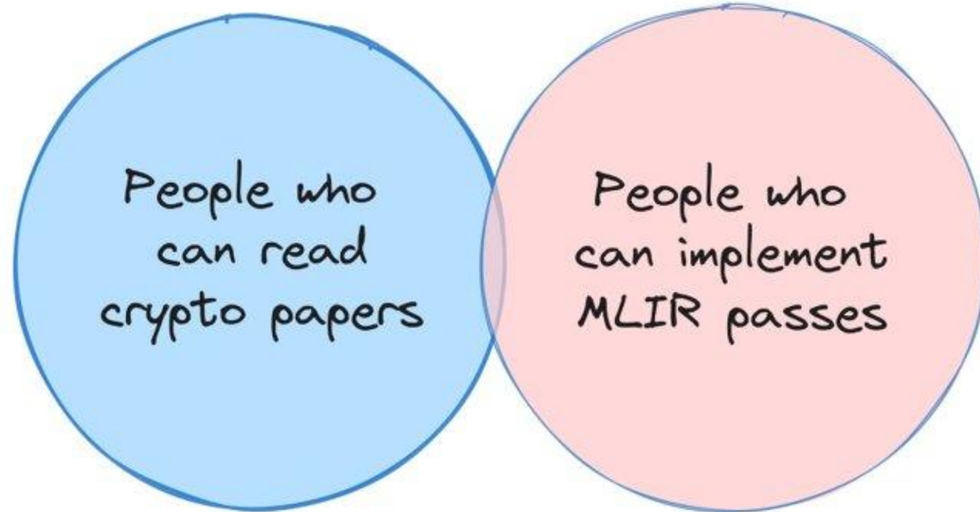
- When should the compiler forget it's crypto and just do polynomial math?
- How late can we commit to representational details like RNS/CRT encodings?
- What is the right approach to noise models?
  - How precise can the noise model be while still being efficient to solve?
  - Some program modifications will change semantics
  - Intertwined with security parameter selection
- How to do dataflow/noise analysis for hybrid schemes
- What sort of cost models will help navigate all these performance tricks?



05

Resources

“We tried using MLIR but found the learning curve too steep, so instead we made something custom.”



MLIR For Beginners (github.com/j2kun/mlir-tutorial)

A series of articles on the MLIR framework for building compilers.

1. Build System (Getting Started)
2. Running and Testing a Lowering
3. Writing Our First Pass
4. Using Tablegen for Passes
5. Defining a New Dialect
6. Using Traits
7. Folders and Constant Propagation
8. Verifiers
9. Canonicalizers and Declarative Rewrite Patterns
10. Dialect Conversion
11. Lowering to LLVM and Machine Code
12. Analysis Passes
13. (keep going until I basically write a book)

github.com/google/heir

google.github.io/heir/community/

Join us!