VAST: MLIR for program analysis of C/C++

Henrich Lauko

November 8-9th, 2022

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Hi everyone

Henrich Lauko

Senior security engineer at Trail of Bits

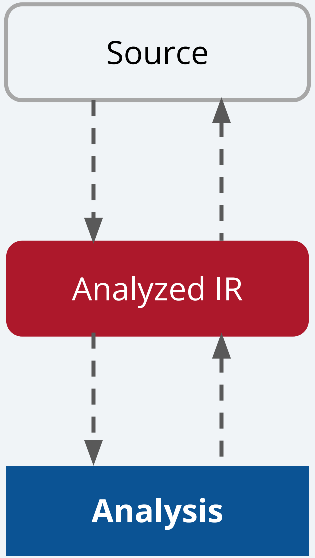
- ○ Email: henrich.lauko@trailofbits.com
- ○ Twitter: @HenrichLauko

VAST – MLIR library for program analysis

https://github.com/trailofbits/vast

Today’s IRs don’t meet analysis needs

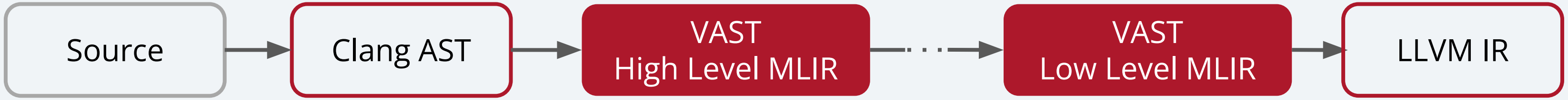
- **Program analysis:**
  - Static analysis and source base queries
  - Fuzzing, abstract interpretation, symbolic execution
  - Program models and instrumentation
- **An orthogonal problem to optimization:**
  - We need truthful information about program semantics
  - Optimizations are destructive transformations
  - Challenge is to relate results back to source



We need a program analysis-focused IR

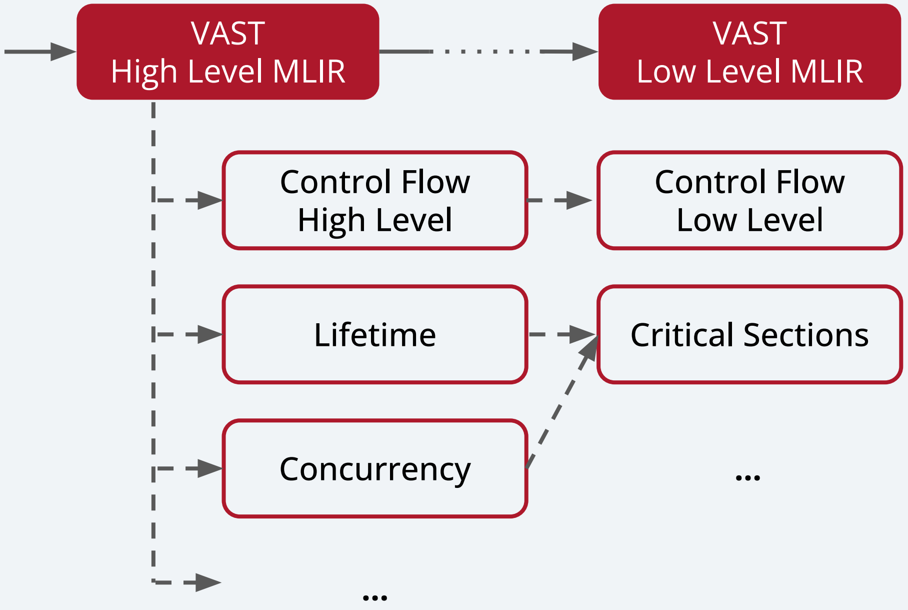
- **Analysis of source code level (semgrep, weggli):**
  - Lacks semantic awareness
- **Analysis at Clang AST level (ast-matcher):**
  - Too complex for more heavy-duty/interpretation based analysis
  - Not a complete source of truth
- **Analysis at LLVM IR level (sanitizers, KLEE):**
  - A collection of IR flavours/dialects – intrinsic-based dialects
  - Too low-level for some analyses
  - Hard to relate to source after optimization, e.g. ABI is already lowered

MLIR is the future of program analysis



- **VAST – MLIR library for program analysis:** https://github.com/trailofbits/vast
- **Views of the source code at the various stages of translation to LLVM**
- **Various stages are interesting for different analyses:**
  - A high-level control flow with a lowered types
  - Analysis of lifetimes of high-level code, in concurrent environments

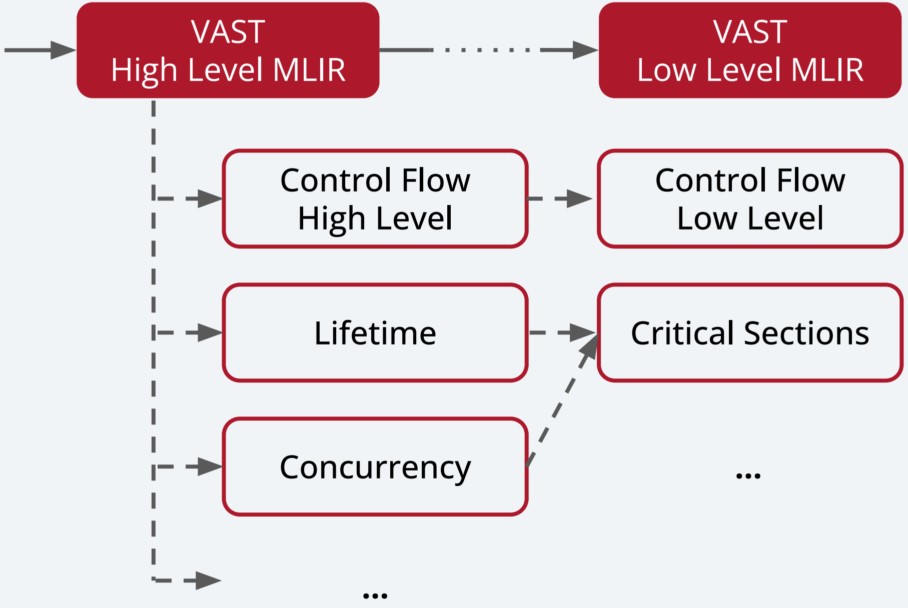
Semantic dialects tailored to analysis goals



```
hl.func external @loop_simple () -> !hl.void {
    %0 = hl.var "i" : !hl.lvalue = {
      %1 = hl.const #hl.integer<0> : !hl.int
      hl.value.yield %1 : !hl.int
    }
    hl.for {
      %1 = hl.ref %0 : !hl.lvalue
      %2 = hl.implicit_cast %1 LValueToRValue :
          !hl.lvalue -> !hl.int
      %3 = hl.const #hl.integer<100> : !hl.int
      %4 = hl.cmp slt %2, %3 : !hl.int, !hl.int
          -> !hl.int
      hl.cond.yield %4 : !hl.int
    } incr {
      %1 = hl.ref %0 : !hl.lvalue
      %2 = hl.post.inc %1 : !hl.lvalue -> !hl.int
    } do {
    }
    hl.return
}
```

Our vision for program analysis

Semantic dialects tailored to analysis goals



```
llvm.func @loop_simple() {
  %0 = llvm.mlir.constant(1 : index) : i64
  %1 = llvm.alloca %0 x i32 : (i64) -> !llvm.ptr<i32>
  %2 = llvm.mlir.constant(0 : i32) : i32
  llvm.store %2, %1 : !llvm.ptr<i32>
  hl.for {
    %3 = llvm.load %1 : !llvm.ptr<i32>
    %4 = llvm.mlir.constant(100 : i32) : i32
    %5 = llvm.icmp "slt" %3, %4 : i32
    hl.cond.yield %5 : si32
  } incr {
  } do {
  }
}
```

Provenance dialects

```
struct Point {
    int x, y, z;
};

Point add(Point a, Point b) {
    ...
}
```

LLVM

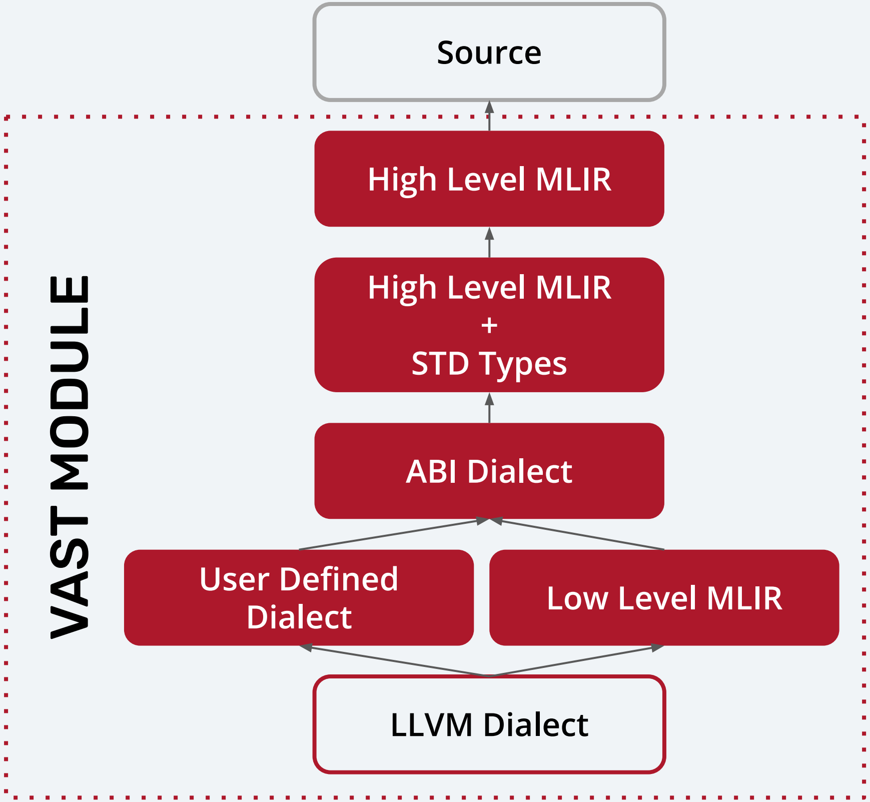
```
define {i64, i32} @add(i64 a1, i32 a2,
                       i64 b1, i32 b2)
```

VAST ABI Lowering

```
func add(Point a, Point b) -> Point {
  abi.entry { // prologue
    [a1: i64, a2: i32] = abi.lower(a)
    [b1: i64, b2: i32] = abi.lower(b)
  } body -> [i64, i32] {
    // use a1, a2, b1, b2
    // return {r1: i64, r2: i32}
  } abi.return { // epilogue
    return abi.lift(r1, r2): Point
  }
}
```

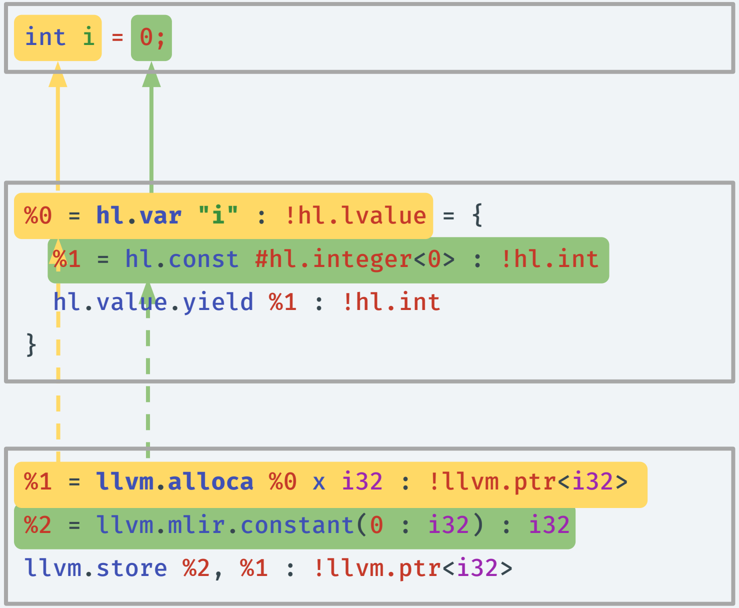
Our vision for program analysis

Tower of IRs



Our vision for program analysis

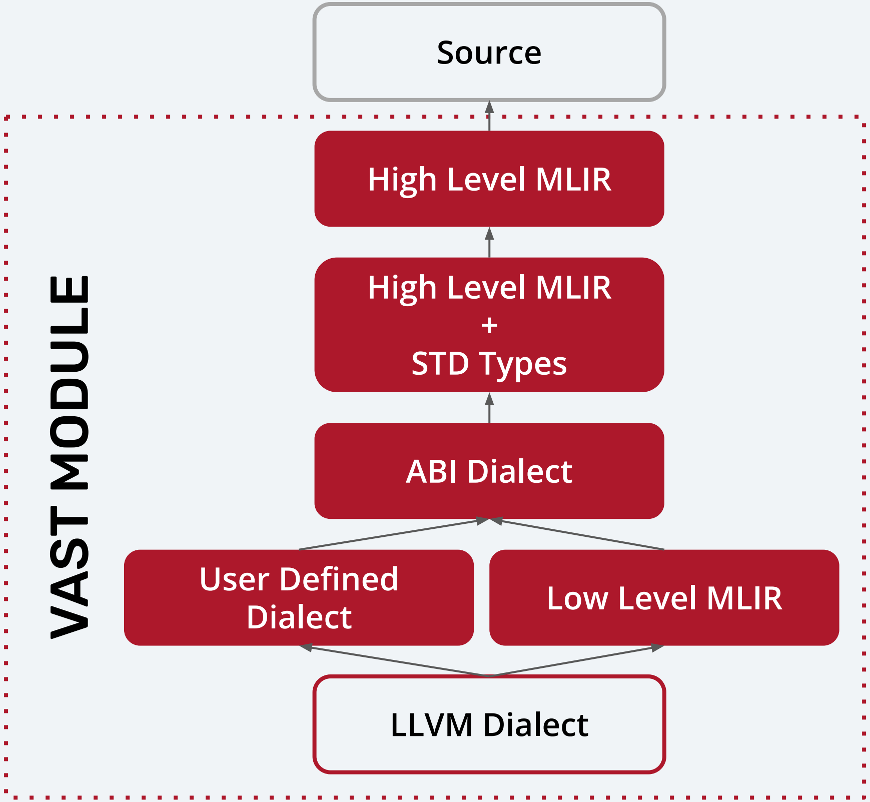
Tower of IRs



```
int i = 0;
```

```
%0 = hl.var "i" : !hl.lvalue = {
  %1 = hl.const #hl.integer<0> : !hl.int
  hl.value.yield %1 : !hl.int
}
```

```
%1 = llvm.alloca %0 x i32 : !llvm.ptr<i32>
%2 = llvm.mlir.constant(0 : i32) : i32
llvm.store %2, %1 : !llvm.ptr<i32>
```



VAST MODULE

Source

High Level MLIR

High Level MLIR + STD Types

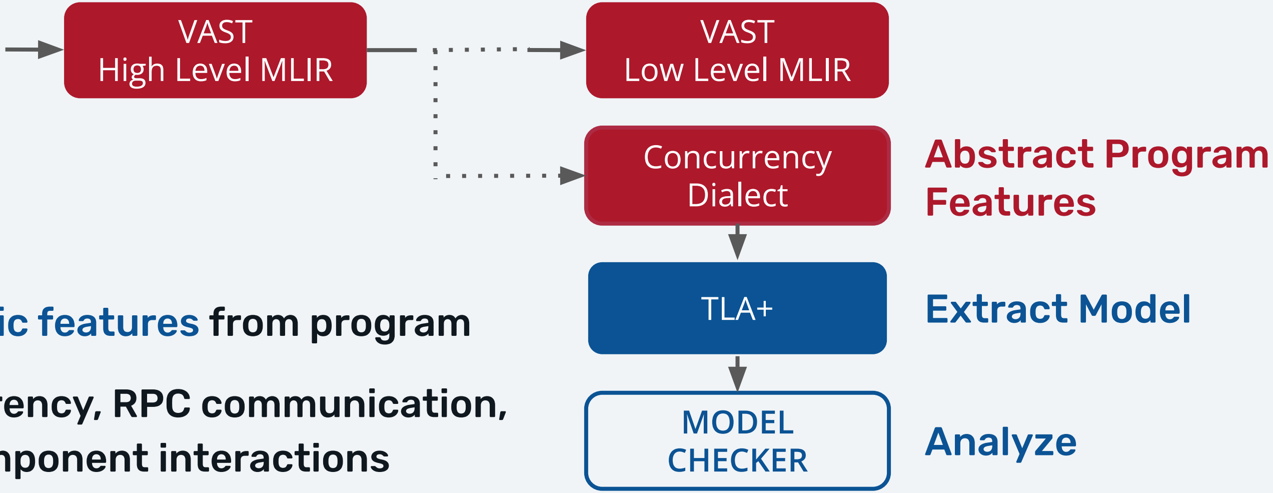
ABI Dialect

User Defined Dialect

Low Level MLIR

LLVM Dialect

Sometimes compilation isn't the goal

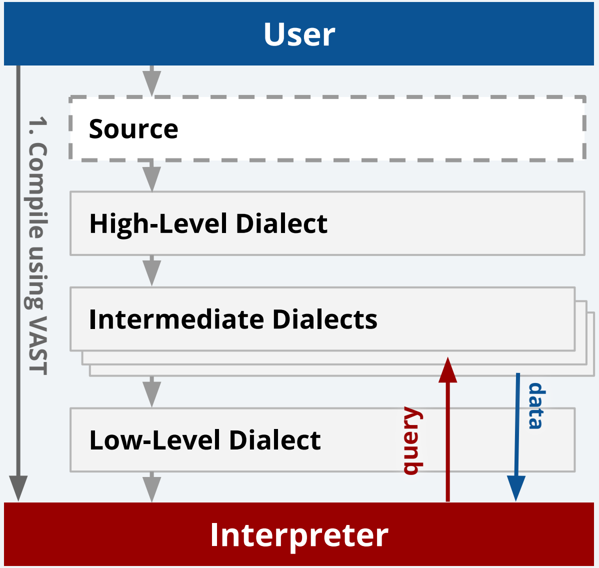


- **Extract specific features from program**
- **Model concurrency, RPC communication, protocols, component interactions**
- **Use tower to report analysis results**

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited)

How we want to analyze programs

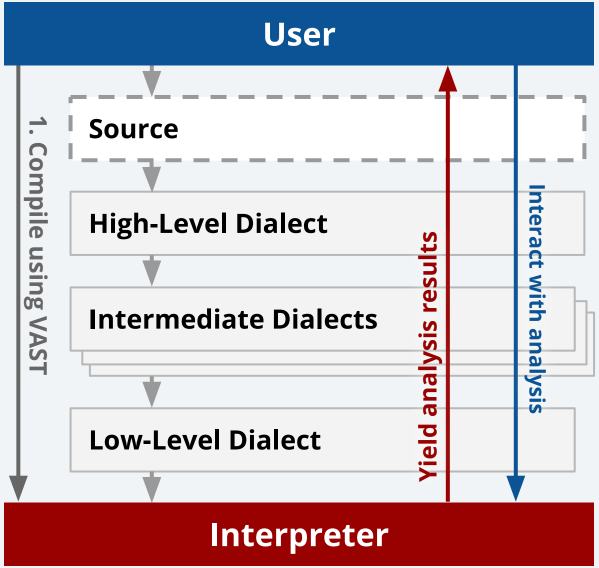
- Want efficiency of LLVM IR and expressivity of source
- Requires all representations
- Use tower of IRs to get high-level view



Distribution Statement "A" (Approved for Public Release, Distribution Unlimited)

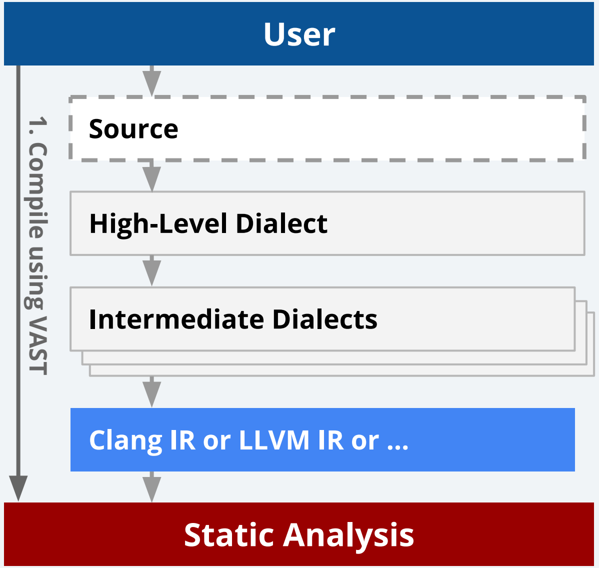
What about human-in-the-loop?

- Want efficiency of LLVM IR and expressivity of source
- Requires all representations
- Use tower of IRs to get high-level view
- Present the user what he recognizes

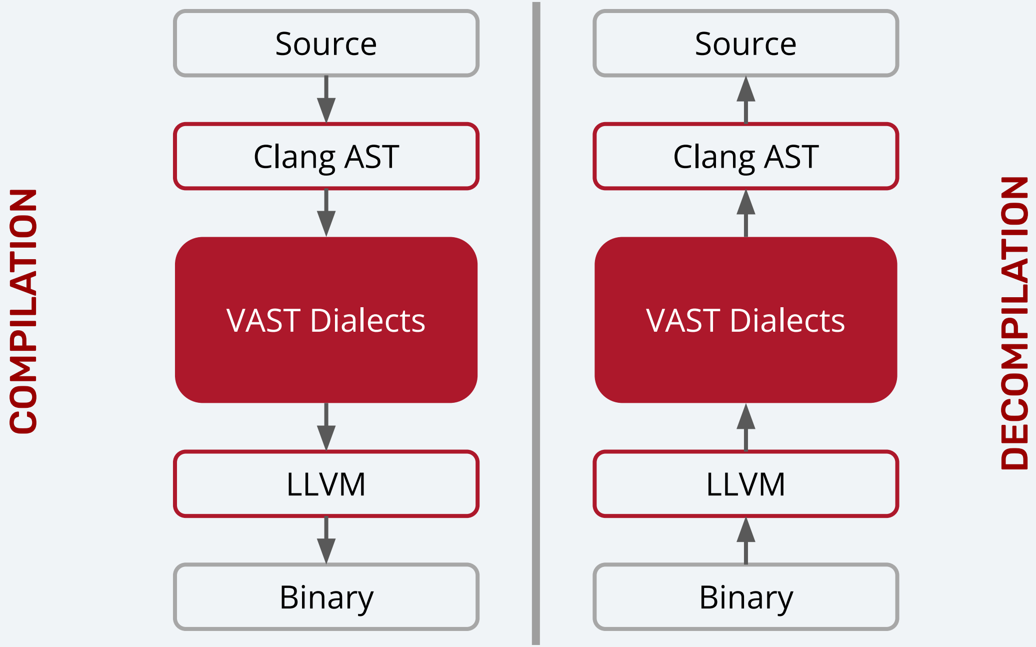


VAST gives you a tower, not a silo

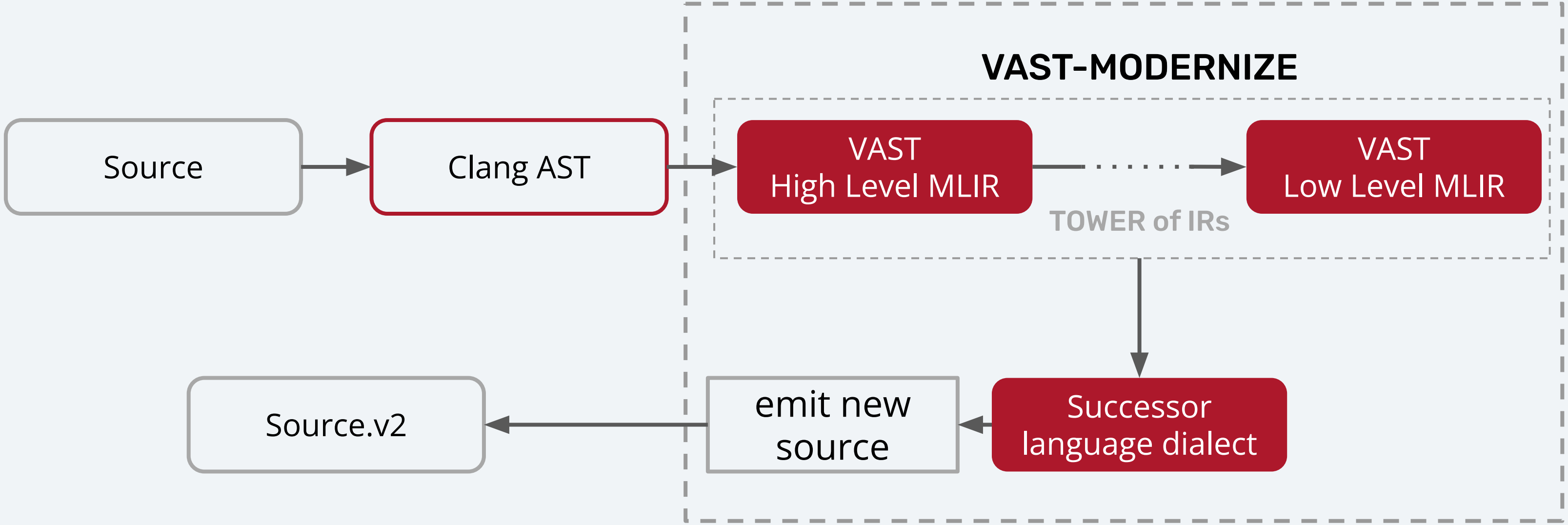
- **Information rich dialects**
- **Lower dialects to other tool's dialects**
- **For example Clang IR or LLVM IR**
- **Allows to leverage high-level MLIR for smoother instrumentation and easier program analysis**



Decompilation



Transpiling with VAST



Distribution Statement "A" (Approved for Public Release, Distribution Unlimited)

CPP to CPP2 declarative parameters

```
void f(const X& x) {
    g(x);
}
```

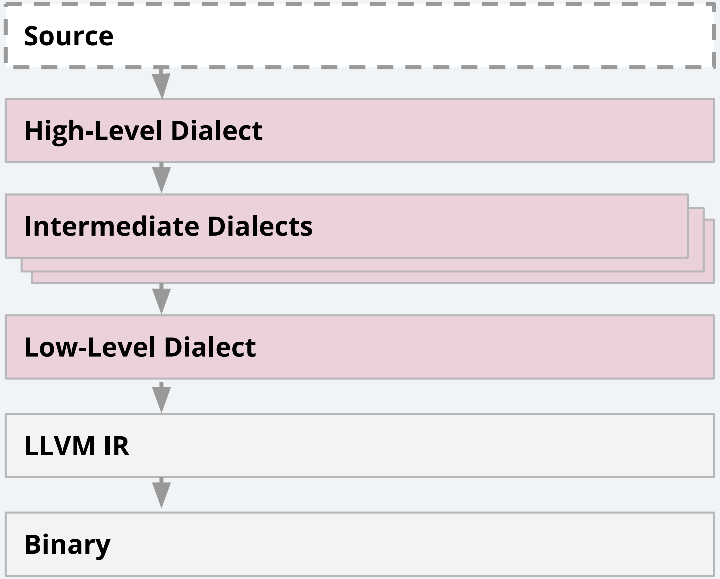
→

```
void f(in X x) {
    g(x);
}
```

```
hl.func @f(%x : !hl.ref<!hl.lvalue<!hl.struct<"X">>, const>) {
  %0 = hl.call @g(%x) : (!hl.ref<!hl.lvalue<!hl.struct<"X">>, const>)
     -> !hl.void
}
```

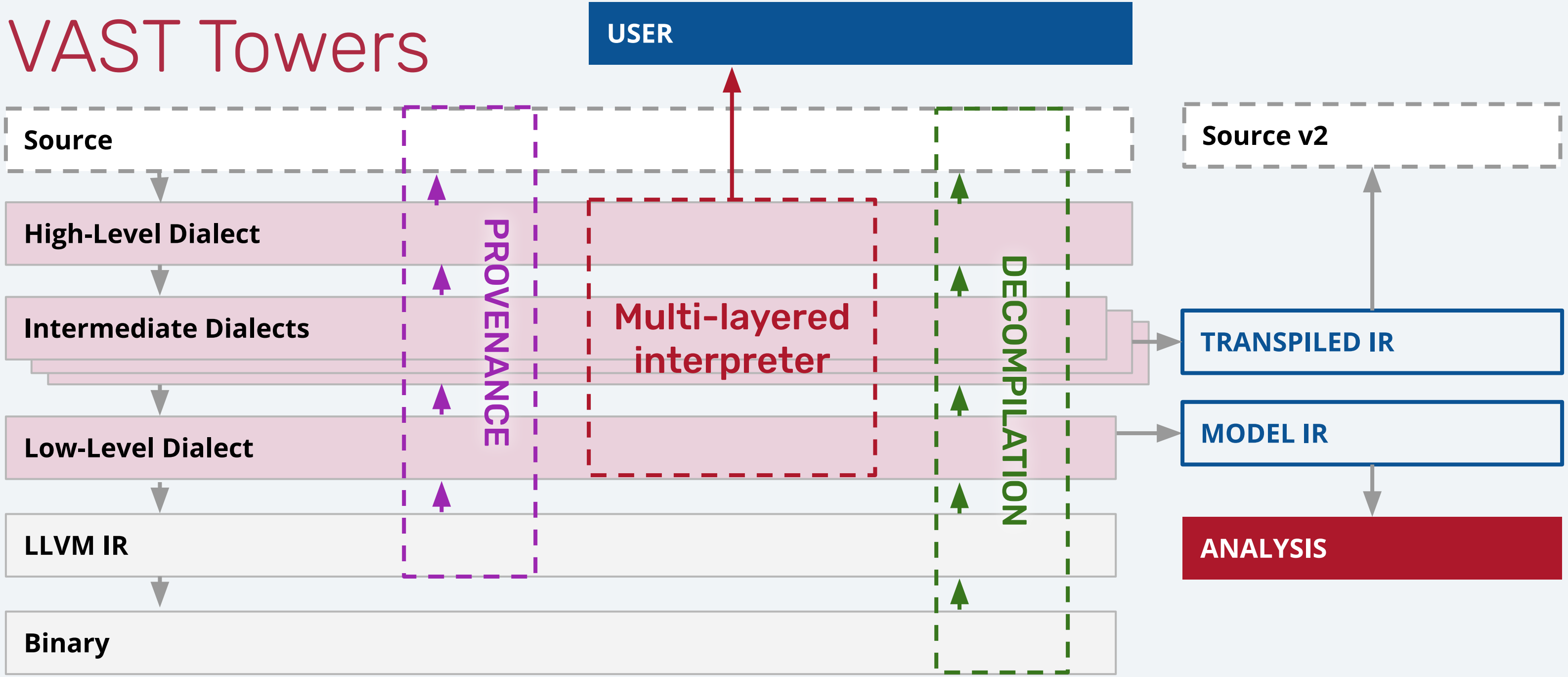
```
hl.func @f(%x : !par.in<!hl.lvalue<!hl.struct<"X">>>) {
  %0 = hl.call @g(%x) : (!par.in<!hl.lvalue<!hl.struct<"X">>>)
     -> !hl.void
}
```

VAST Tooling



- **Compilation description dialects**
- **Configurable codegen**
  - How to represent provenance
  - How to lower unsupported primitives
- **MLIR interactive editing tool (REPL)**
- **MLIR query tool**

VAST Towers



VAST open source at: https://github.com/trailofbits/vast