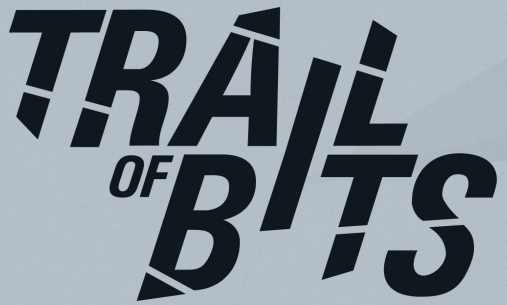


MLIR is the future of program analysis

Qualcomm Product Security Summit

May 18th, 2023

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

I have returned

Peter Goodman

- **Staff engineer at Trail of Bits**
  - Email: peter@trailofbits.com
  - Twitter: @peter_a_goodman
  - Mastodon: https://infosec.exchange/@pag
- **Talk to me about:**
  - **Static or dynamic binary translation**
    - Remill, Anvill, McSema, GRR, microx, Granary, DynamoRIO, etc.
  - **Static or dynamic program analysis**
    - PASTA, Magnifier, DeepState, KLEE-native, Datalog compilers
  - **LLVM, MLIR**
    - Rellic, VAST
- **Last time at QPSS (2019):**
  - **PowerFL: Fuzzing VxWorks embedded systems (slides)**

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

The last 15 years of program analysis focused on LLVM; the next 15 will not

We are where we are today because of LLVM

- **2003: LLVM created by Chris Lattner**
  - LLVM's intermediate representation (IR) makes low-level transformations and optimizations easy
- **2007: Clang frontend**
  - Makes LLVM IR *relevant*: can get it from C, C++
  - GCC-compatible command-line options
  - **Beginning of 15 years of continuous innovation**
- **2014-2018: Windows support**
  - Chrome compiles on Windows with `clang-cl`
- **2023: Clang and LLVM are everywhere**
  - Primary compiler used by Apple, Meta, Google
  - Primary IR used for academic research

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

Clang / LLVM made the important things easy

- **Clang makes it easy to get LLVM IR from C and C++ code**
  - **Primary analysis substrate of source code was LLVM IR by proxy**
  - Can generally handle GNU- and MSVC-specific extensions
  - Easy to extract LLVM IR from Clang (option `-emit-llvm`)
- **LLVM's APIs make analyzing and transforming LLVM IR *really* easy**
  - `IRBuilder` for convenient construction and injection of new instructions
  - `val->replaceAllUsesWith(other_val)`
  - `val->uses()` to find something is easy
    - `Use::get()` to get back `val`
    - `Use::set(new_val)` to change `val` in the user
    - `Use::getUser()` to get the user of `val`
- **Key takeaways: LLVM is a productivity multiplier for transforming code**

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

Eking out better results will require access to more information

Bug-finding has reached “peak LLVM”

- **LLVM has been *the* driving force in compiler-based bug-finding tools**
  - **Runtime sanitizers**: Address, Memory, Thread, Undefined, Data flow
  - **Fuzzing**: libFuzzer, AFL++
  - **Symbolic execution**: KLEE, S^2E, Sys
  - **Model checking**: DIVINE, LLBMC
  - **Static analysis**: Pagai, cclyzer / cclyzer++, PhASAR
  - **Translation validation**: Alive, Alive2
  - **Binary lifting**: McSema, Remill, Anvill, Rev.ng, RetDec, *etc.*
- **LLVM designed for optimization, *not bug-finding***
  - LLVM-based bug-finding tools are stuck in a **local maximum**

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

Intent of source code is lost in translation

C source code	LLVM IR	Machine code
`int, unsigned, time_t, pid_t`	`i32`	DWARF debug info, if lucky
`struct point { int x, y; }` as a function parameter	`i64` (x86-64, ARMv8) `i32, i32` (x86, MIPS) `[2 x i32]` (ARMv7)	DWARF debug info, if lucky
Implicit downcast, explicit downcast	`trunc i64 %val to i32`	Read low order bytes, sub-register, or mask
	`and i64 %val, i64 0xffffffff`	
Local variables	`alloca` (unspecified ordering)	Stack memory (hardcoded ordering)
	SSA values (e.g. `%foo.scev.sroa.1.1.3`)	Registers
		Stack memory (spill/fill slots)
		Return address, callee-saved registers

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

LLVM bug-finding suffers the streetlight effect

Intent of source code is lost in translation

C source code	LLVM IR	Machine code
`int, unsigned, time_t, pid_t`	`i32`	DWARF debug info, if lucky
`struct point { int x, y; }` as a function parameter		
Implicit downcast, explicit downcast	`and i64 %val, i64 0xffffffff`	ub-register, or mask
Local variables	`alloca` (unspecified ordering)	Stack memory (hardcoded ordering)
	SSA values (e.g. `%foo.scev.sroa.1.1.3`)	Registers
		Stack memory (spill/fill slots)
		Return address, callee-saved registers

Type names and signedness in high-level code can be load-bearing, and have *implied* semantics. This is lost in translation to LLVM.

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

LLVM bug-finding suffers the streetlight effect

Intent of source code is lost in translation

C source code	LLVM IR	Machine code
`int, unsigned, time_t, pid_t`	`i32`	DWARF debug info, if lucky
`struct point { int x, y; }` as a function parameter	`i64` (x86-64, ARMv8) `i32, i32` (x86, MIPS) `[2 x i32]` (ARMv7)	DWARF debug info, if lucky
Implicit downcast, explicit downcast	`t … i64 %val to i32`	Read low order bytes, sub-register, or mask
		Stack memory (hardcoded ordering)
		Registers
		Stack memory (spill/fill slots)
		Return address, callee-saved registers

Hard to report bugs related to types and values that cannot be easily related back to source code.

Ideally, want consistent representations so that analyses generalize to different architectures.

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

Intent of source code is lost in translation

C source code	LLVM IR	Machine code
`int, unsigned, time_t, pid_t`	`i32`	DWARF debug info, if lucky
`struct point { int x, y; }` as a function parameter	`i64` (x86-64, ARMv8) `i32, i32` (x86, MIPS) `[2 x i32]` (ARMv7)	DWARF debug info, if lucky
Implicit downcast, explicit downcast	`trunc i64 %val to i32` `and i64 %val, i64 0xffffffff`	Read low order bytes, sub-register, or mask
Local variables	ified ordering)	Stack memory (hardcoded ordering) slots) saved registers

Un/signed conversions are bread and butter of buffer overflows. Can't distinguish implicit vs. explicit conversion, or signed to unsigned conversions in LLVM

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

Intent of source code is lost in translation

Knowing how data is stored on the stack would help diagnose the severity of stack-based buffer overflows.

Need full-stack visibility to distinguish intra- from inter-structure overflows.

		Machine code
		DWARF debug info, if lucky
	ARMv8) PS)	DWARF debug info, if lucky
Implicit downcast, explicit downcast	and i6 l, i64 0xffffffff	Read low order bytes, sub-register, or mask
Local variables	`alloca` (unspecified ordering)	Stack memory (hardcoded ordering)
	SSA values (e.g. `%foo.scev.sroa.1.1.3`)	Registers
		Stack memory (spill/fill slots)
		Return address, callee-saved registers

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

Bugs span the semantic gap, bug-finding should too

Different IRs are good for different things

Level	Pros	Cons
High	• Close to bug-finder domain • Explicit abstractions (data structures) • Explicit intra-object boundaries	• Verbose, not efficiently analyzable • Missing implicit behaviors (e.g. C++ destructor calls)
Medium	• Explicit control-flow, data-flow	• Semantic types (e.g. `time_t`) elided
Low	• Efficiently analyzable • Explicit inter-object boundaries	• Tenuous connection back to source code • ABI-specific data representation, inlining, folding, propagation, has destroyed data
Binary	• Bug-exploiter domain • Blurred object boundaries (easier to evaluate buffer overflows) • Succinct	• Blurred object boundaries (hard to analyze) • Unreliability of debug info, symbols • Tight coupling of control-flow, type, variable recovery

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

Ideally, we want the efficiency of LLVM IR and expressivity of source

LLVM IR: A blessing and a curse

Blessings

- **Trust**
  - **Analyze the same representation used by the compiler to generate machine code**
- **Broad compatibility**
  - GNU and MSVC option and extension support
- **Permissive open-source license**
  - Academic and industry *momentum*
- **Easy and scalable to analyze**
  - Not that many kinds of instructions
  - Close-ish to C
- **Debug information**
  - Points back to source code
  - DWARF-like types

Curses

- **Many unspecified LLVM dialects**
  - -O0 *vs.* -O1 *vs.* -O2 *vs.* -O3
  - ABI-specific intrinsics, ABI lowering of types
- **Very low level**
  - **Inlined mechanics of abstractions** (e.g. C++ standard library containers)
  - **Optimized for target, not for analyzer**
- **LLVM values are meaningless**
  - **Not related to bug-finder's domain: source**
  - `%foo.1.scev.sroa.1.1.3` 🤣
- **API evolution causes tool churn**
  - Many tools stuck on LLVM 3.x, 4.x, 5.x, *etc.*
  - Many tools will never work with opaque `ptrs`
- **Debug information is unreliable**

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

Ideally, we want the efficiency of LLVM IR and expressivity of source

LLVM IR: A blessing and a curse

Blessings

- **Trust**
  - **Analyze the same representation used by the compiler to generate machine code**
- **Broad compatibility**
  - GNU and MSVC option and extension support
- **Permissive open-source license**
  - Academic and industry *momentum*
- **Easy and scalable to analyze**
  - Not that many kinds of instructions
  - Close-ish to C
- **Debug information**
  - Points back to source code
  - DWARF-like types

Curses

- Many unspecified LLVM dialects
- Debug information is unreliable

It turns out that if you pray for ~~forgiveness~~ a better IR then Chris Lattner will deliver one.

MLIR, the multi-level intermediate representation, looks like the future of domain-specific compiler technology, and is well-suited for full-stack bug-finding.

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

One ~~ring~~ IR to ~~rule~~ represent them all and in the ~~darkness~~ analysis bind them

Represent everything with MLIR

Multi-Level Intermediate Representation (MLIR)

- **Define new IRs, called *dialects***
  - Invent your own domain-specific language
  - Tree-structured, SSA-based
- **New dialects come batteries-included**
  - **Mutation operations, just like with LLVM**
  - Validation logic, e.g. for type checking
  - Serialization and printing logic for persistence
- **Key feature: Composition**
  - **Multiple dialects can be simultaneously used within a single module**

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

MLIR looks like it will thrive in the next 15 years

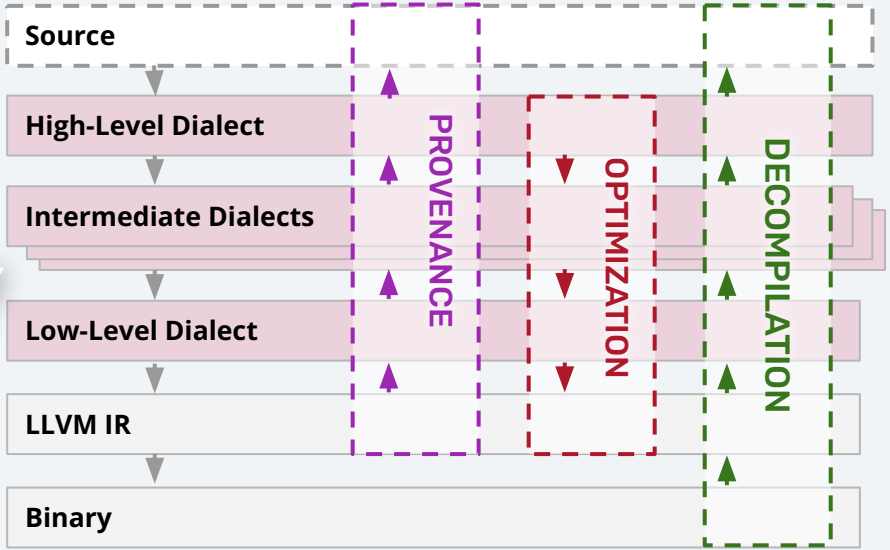
- **Momentum**
  - **Captured industry and academic mindshare** (TensorFlow, OpenXLA, Flang, *etc.*)
    - Driven by unique needs of ML and hardware compilers
    - Database query optimizers / compilers hot on their heels
    - Laggards: program analysis!
- **MLIR comes with an LLVM dialect built-in**
  - Don't throw the baby out with the bathwater
  - Learns from Swift's SIL and Rust's MIR
    - Some language-specific optimizations are best applied in a higher-level IR
    - Lower to LLVM IR to benefit from battery of pre-existing optimizations
- **Open-source, permissively licensed, enthusiastically developed**
  - MLIR is a first-class LLVM subproject

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

Our vision: a Tower-of-IRs

MLIR dialects can bridge the semantic gap by representing the *same* program at *different* levels of abstraction.

Provenance: higher-level dialects can be the debug information for lower-level dialects.



MLIR

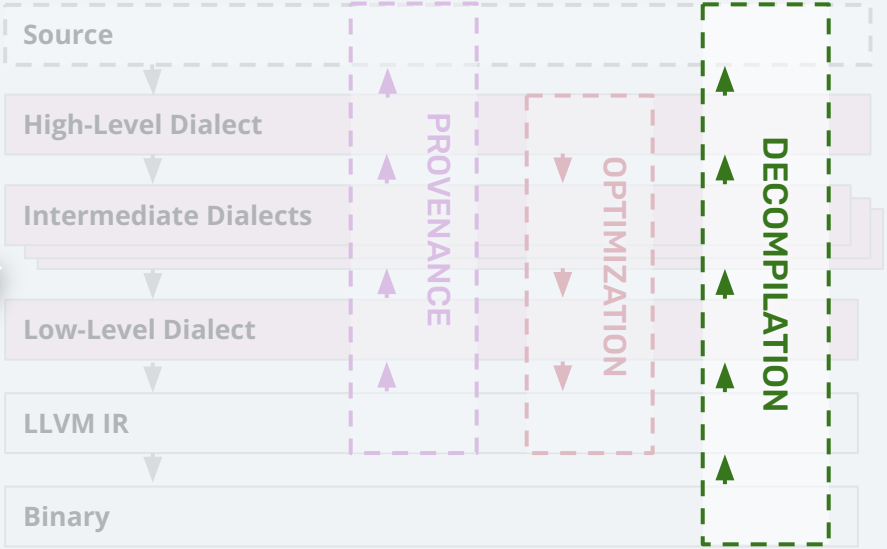
This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.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

Tomorrow's vision, today

Building the tower, one brick at a time

rev.ng is building an interactive decompiler. Their MLIR dialect, `clift`, is used in their backend to accurately represent high-level C types and constructs.

They also work with Qualcomm on qemu-hexagon!



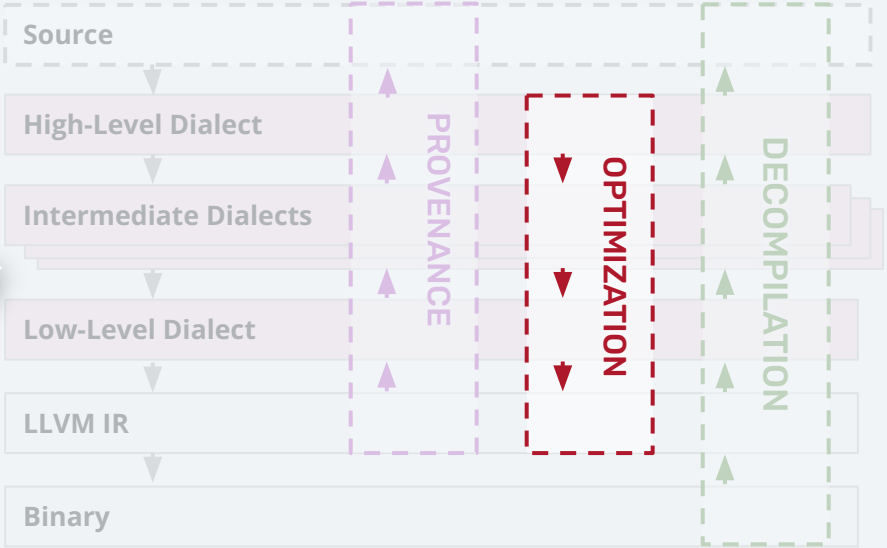
This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.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

Building the tower, one brick at a time

Meta is building Clang IR (CIR) to find C++ coroutine lifetime bugs, and better optimize C++ code using coroutines.

CIR is a mid-level dialect: it mixes high-level control-flow with low-level, LLVM-like data representations.

Meta

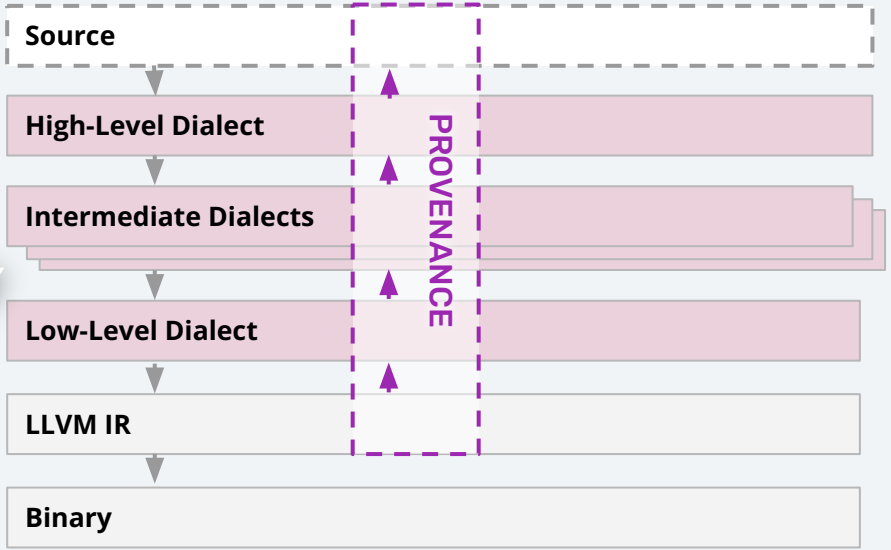


This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.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

Building the tower, one brick at a time

Trail of Bits is building the *whole damn tower* with github.com/trailofbits/vast.

VAST converts Clang ASTs into a high-level dialect, then progressively lowers it down to Clang-compatible LLVM IR. We will target CIR for C++ support.



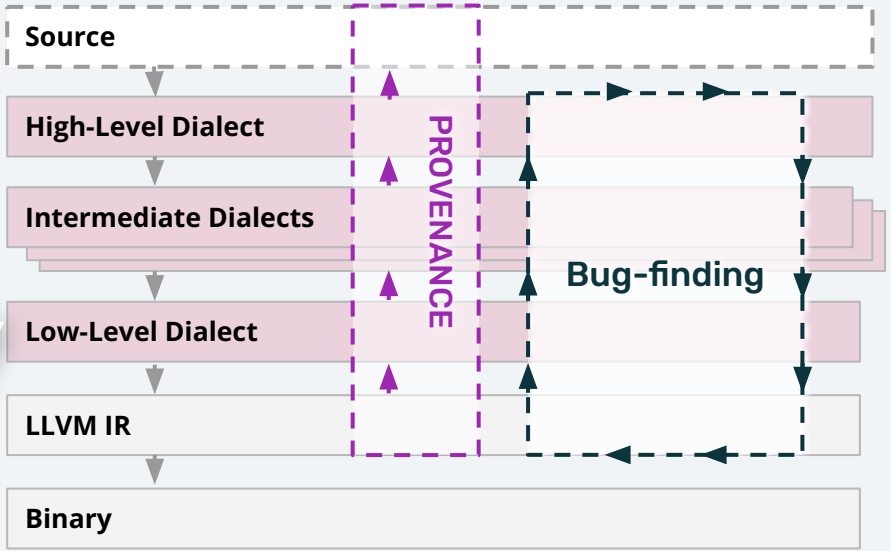
This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.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

Full-stack bug-finding

A tower-of-IRs gives your analysis *visibility*, from high to low.

MLIR lets you *specialize* the tower to your analysis needs: bring your own dialects!

Stay *productive* and skip format shifting: your analyses can live in one address space.



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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

State of MLIR

- **MLIR *usage* is fractured**
  - Mix of in-tree and out-of-tree, forked versions, etc.
  - Hard to compose independent projects, as they're often on slightly different MLIR versions
    - Fast pace of MLIR core development means high API churn, similar to early LLVM
- **Almost (but not yet) easy to use**
  - Steep learning curve
  - Defining a new dialect requires using the evil TableGen system
  - There's hope: IRDL and xDSL will make dialect definition/extension substantially simpler
- **High level of effort**
  - A new code generator doesn't materialize overnight
  - Lot's still to do to support C end-to-end

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

Actually finding bugs with MLIR

- **CVE-2021-33909, *aka* Sequoia, is a Linux kernel privilege escalation bug**
  - Implicit downcast of a `size_t` to an `int`
  - Integral types (e.g. `i64`, `i32`) in LLVM IR don't know their signedness
  - `trunc` operations in LLVM IR don't know if they were generated from explicit or implicit casts
- **Marek Surovič created vast-checker for finding Sequoia**
  - VAST's high-level MLIR dialect distinguishes integer signedness, implicit and explicit casts
  - Blog post coming soon!
- **Lessons learned**
  - Need ability to connect across translation units, e.g. integrate with Clang Static Analyzer

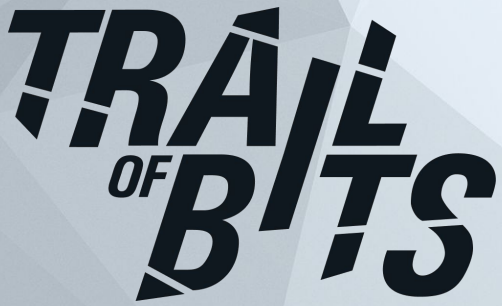
This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.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

I'm looking forward to the future

MLIR is the future of program analysis

- **The last 15 years of program analysis focused on LLVM**
  - Clang made LLVM relevant, LLVM made transformation easy
  - Eking out better results will require access to more information
  - Intent of source code is lost in translation
- **Bugs span the semantic gap, bug-finding should too**
  - Represent everything with MLIR, using a ***tower-of-IRs***
  - We want the efficiency of LLVM IR and expressivity of source
  - Avoid format shifting
- **State of MLIR**
  - High momentum, high investment
  - High churn, steep learning curve
- **Want MLIR for C *today*? Try VAST (github.com/trailofbits/vast)**
  - vast-checker is our first experiment with MLIR-based bug-finding

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited



Backup slide

Multiple IRs isn't a new idea

Binary Ninja is an interactive disassembler for reverse engineering

- **Represents code using *interconnected* IRs**
  - **LLIL**: Low-level machine code instruction semantics
  - **MLIL**: Stack slots, local variables, value analysis
  - **HLIL**: High-level types, structured control-flow
- **Pros:**
  - **Start your analysis at the best-fit IR for your goal**
  - Interactive and batch mode, killer for visualization
- **Cons:**
  - **Can't extend built-in IRs**
  - Mutation via re-creation or patching machine code bytes
  - Reverse engineering focused, so not ideal if you have source

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

Backup slide

Isn’t this kind of like CodeQL?

CodeQL is a semantic code analysis platform for vulnerability discovery across a broad set of languages

- **Squint and CodeQL classes kind of look like dialects**
- **Pros**:
  - **CodeQL is awesome, you should use it**
  - Sophisticated built-in analyses, e.g. taint tracking
- **Cons**:
  - **Hard to debug: what / where are the false-negatives?**
  - Re-implements compiler logic like IR generation, but is it feature- or bug-compatible with any *real* compiler?
  - Can’t go “all the way down” to executable code

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

Backup slide

Analysis workflow / architecture

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

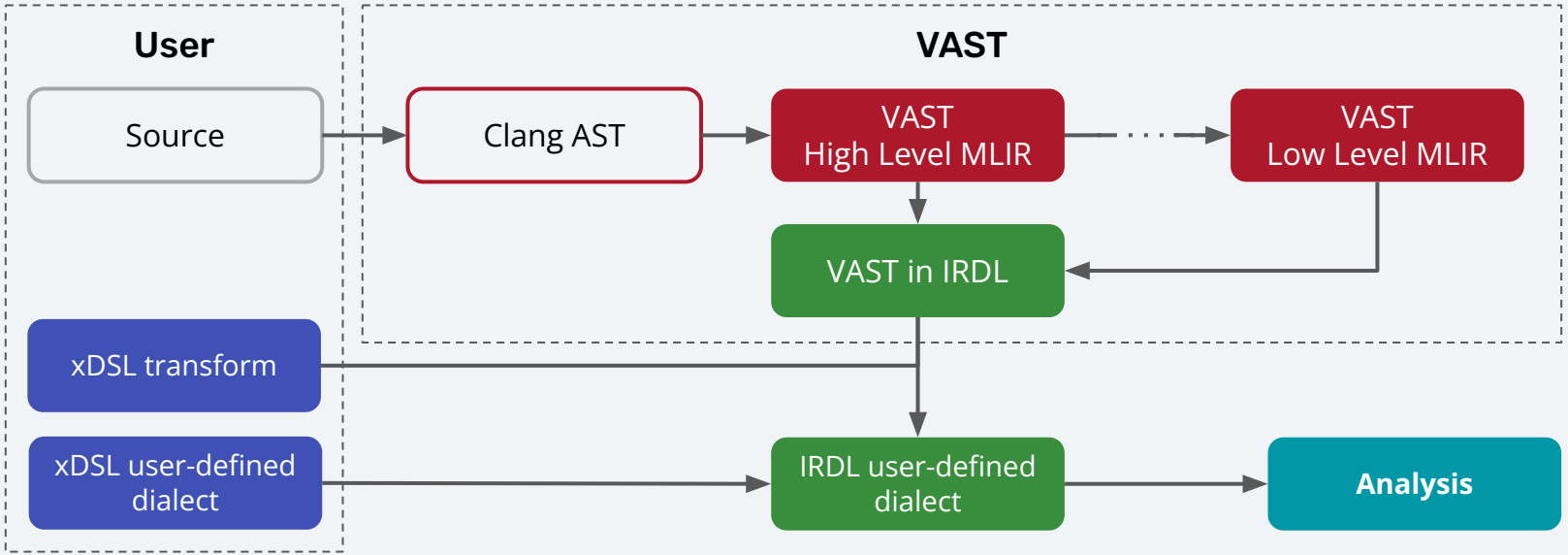
VAST represents the same program at multiple abstraction levels

MLIR “dialects” as intermediate representations

High-level dialect	Close to AST, includes control-flow
Middle-level / core dialect	Propagated aliases, tuples, lazy blocks
ABI dialect	Lowers high-level typed values
Built-in MLIR-provided dialects	Standard types, arithmetic, control-flow
LLVM dialect	Executable semantics
Metadata dialect	Connects everything to Multiplier, source

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

Dialects-as-abstractions workflow



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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

Backup slide

Extending MLIR from Python

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Should be easy to extend MLIR

- **How? Using the IRDL dialect definition language**
  - IRDL is a meta-dialect for defining MLIR dialects
  - Enables runtime-definable dialects

- **xDSL makes IRDL usable and extensible from Python**
  - Dialects defined using Python classes
  - Extension points defined with methods

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

First-class entities can be extended with xDSL

Example: Extending operations with methods…

```
@irdl_op_definition
class IfOp(Operation):
  name = "hl.if"
  cond = OperandDef(IntegerType.from_width(1))
  true_region  = RegionDef()
  false_region = RegionDef()
  def formula(self):  # Returns an SMT representation of an if statement
    return Ite(Equals(self.cond.op.symbol(), BV(1, 1)),
               region_formula(self.true_region),
               region_formula(self.false_region))
```


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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

The composition of these extension methods enables convenient analysis

… lets us synthesize SMT formulas

```
int x;
if (1) {
    int a;
    x = a + 1;
} else {
    int b;
    x = b + 2;
}
```



```
((FV0 = 1_32) &
 (FV1 = 2_32) &
 (FV3 = x) &
 (FV5 = 1_1)
     ? ((FV6 = a) &
        (FV7 = (FV6 + FV0)) &
        (FV3 = FV7))

     : ((FV8 = b) &
        (FV9 = (FV8 + FV1)) &
        (FV3 = FV9))
)
```


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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

Backup slide

Modelling libc functions with MLIR

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

Example: Modelling libc library functions as operations in a new dialect

Should be easy to define custom abstractions

- **User-defined dialects will branch off of VAST's tower of IRs like a tree**
  - **Problem**: How to introduce new abstractions, tailored to analysis needs?
  - **Solution**: Transform one of VAST's core dialects into analysis-specific dialect
- **How? Using the IRDL dialect definition language**
  - IRDL is a meta-dialect for defining MLIR dialects
  - Enables runtime-definable dialects
- **xDSL makes IRDL usable and extensible from Python**
  - Dialects defined using Python classes
  - Extension points defined with methods

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Using IRDL to specify `strcmp`

```
Dialect libc {
  Alias !string = !hl.array<!hl.char<const>>

  Operation strcmp {
    Operands (lhs: !string, rhs: !string)
    Results  (res: !hl.int)
    Format   "$lhs, $rhs"
    Summary  "Compares two null-terminated byte strings lexicographically."
  }
}
```


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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

Example: Modelling libc library functions as operations in a new dialect

Using xDSL to specify `strcmp`

```
@irdl_attr_definition
class StringType : HLArrayType(HLChar(const))
  name = "string"

@irdl_op_definition
class StrCmpOp(Operation):
  lhs    = OperandDef(StringType())
  rhs    = OperandDef(StringType())
  result = ResultDef(HLIntegerType())
```


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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited

Example: Modelling libc library functions as operations in a new dialect

Transforming code with xDSL

```
@dataclass
class StrCmpRewrite(RewritePattern):
  @op_type_rewrite_pattern
  def match_and_rewrite(self, op: CallOp, rewriter: PatternRewriter):
    if op.function.name() == "strcmp":
      rewriter.replace_matched_op(StrCmpOp(op.args))
```

- **String comparisons are now a first-class entity**
- **Converting generic operations into domain-specific operations enables *composable analyses* to be implemented in Python with xDSL!**

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A – Approved for Public Release, Distribution Unlimited