Can symbolic execution be a productivity multiplier for human bug-finders?

Peter Goodman

September 15-16th, 2022

Hi everyone

Peter Goodman

- **Staff engineer at Trail of Bits**
  - Email: peter@trailofbits.com
  - Twitter: @peter_a_goodman; the “a” is for “amazing”
- **Talk to me about:**
  - **Static or dynamic binary translation**
    - Remill, Anvill, VMill, McSema, GRR, microx, Granary, DynamoRIO, etc.
  - **Static or dynamic program analysis**
    - PASTA, Magnifier, Dr. Lojekyll Datalog compiler, DeepState, KLEE-native
  - **LLVM, MLIR**
    - Rellic, VAST

Humans are at the center of productivity

- **What makes a tool a productivity multiplier?**
  - How do today's tools measure up?
- **What KLEE can improve upon today to be ready for tomorrow**
  - LLVM's runtime is its biggest strength and missed opportunity
  - LLVM is KLEE's domain, but not the domain of bug-finders
- **The future is multi-level analyses with MLIR**
  - Bug-finding tools should communicate with bug-finders in their domain of interest
  - Bug-finding tools should operate on the best-fit domain(s) for their analyses
  - VAST is making this future happen with of MLIR dialects, from high level, down to LLVM

Three pillars of productivity tools

Bug-finding systems should be…

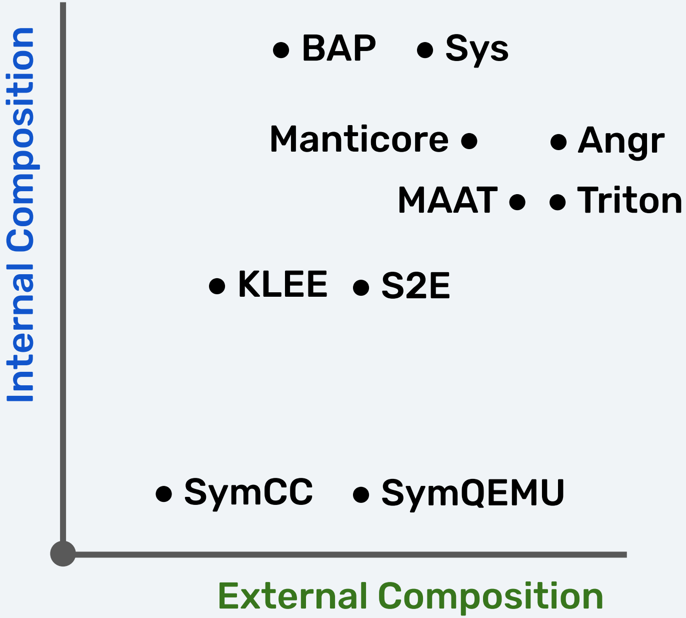
- **Composable**
  - **Internal**: combine existing results to derive new results
  - **External**: outputs can be used as inputs to other tools, tool fits into a larger workflow
- **Comprehensive**
  - Zero false-negatives at targeted bug class
  - Known presence of false-negatives can be mitigated by huge upside
- **Transparent**
  - Tool complexity often leads to unpredictable outcomes
  - Behavior should be *predictable*, results should be *explainable*, limitations *understandable*
  - Context is important: "this `free` is bad" is unsatisfactory

Bug-finding systems should be…

- **Composable**
  - **Internal**: combine existing results to derive new results
  - **External**: outputs can be used as inputs to other tools, tool fits into a larger workflow
- **Comprehensive**
  - Zero false-negatives at targeted bug class
  - Known presence of false-negatives can be mitigated by huge upside
- **Transparent**
  - Tool complexity often leads to unpredictable outcomes
  - Behavior should be *predictable*, results should be *explainable*, limitations *understandable*
  - Context is important: “this `free` is bad” is unsatisfactory

How symbolic executors measure up

- **Practitioner workflow matters**
  - Reverse engineers use IDA Pro, Binary Ninja, Ghidra – integration with these is important
  - Auditors might compile source, so integration with the compiler (e.g. LLVM) is a benefit
- **Requiring the user to do work to get work done doesn't lose points**
  - **Effective use of symbolic execution requires thinking *carefully* about where it best applies, and orchestrating its use**
  - High-value targets already well-tested, already require careful fuzzer harness development to find deep bugs
  - Comparing favorable to a fuzzer that starts at `main` is *not* inspiring

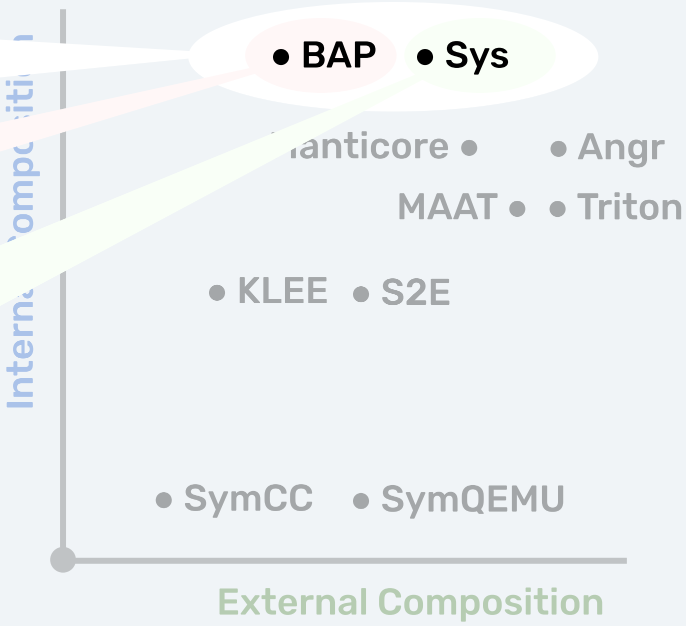


How symbolic executors measure up

- ± OCaml-based: great for program analysis, harder to integrate
- – In for a penny, in for a pound

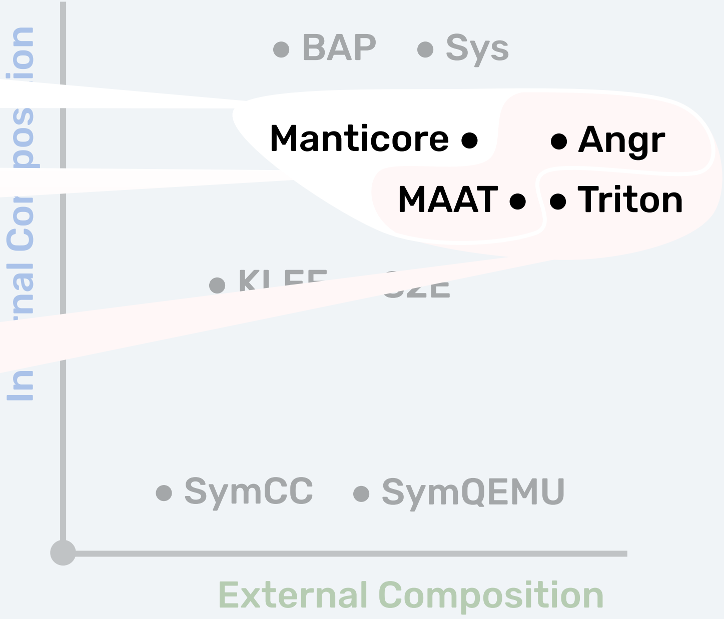
- + **Knowledge base**
  - ± Ensure monotonicity
  - + Mutual fixpoints

- + **Compositional variant analysis and underconstrained symexec**
- + LLVM bitcode as input



How symbolic executors measure up

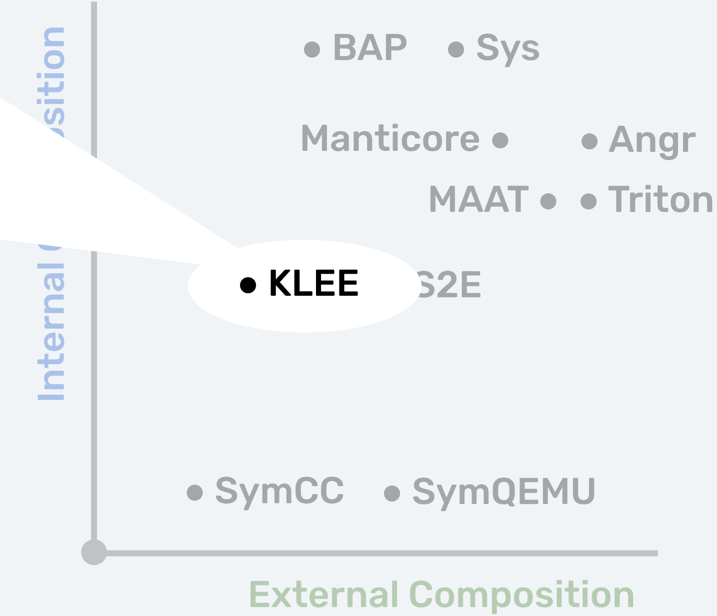
+ **Highly extensible**, due to Python-based implementation
+ **Easy-to-integrate** with domain tools, e.g. IDA Pro, Binary Ninja
+ **Actionable callbacks**: can interpose on memory/register reads or writes, and *alter* what data is read or written



What elephant?

How symbolic executors measure up

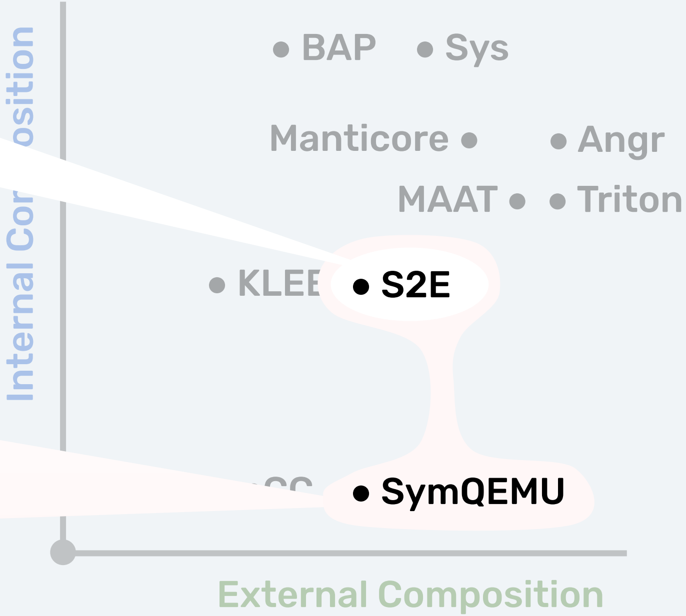
- \+ **The target and the supporting runtime are *both* subject to symbolic execution**
- ± Operates on LLVM bitcode
  - – Source code is user's domain
- – Not meta enough: special function handlers operate in wrong domain
- – Diffidence: KModule, KInstruction
- – Monolithic, focused on `main`: yields "complete" input models, but limits usefulness



How symbolic executors measure up

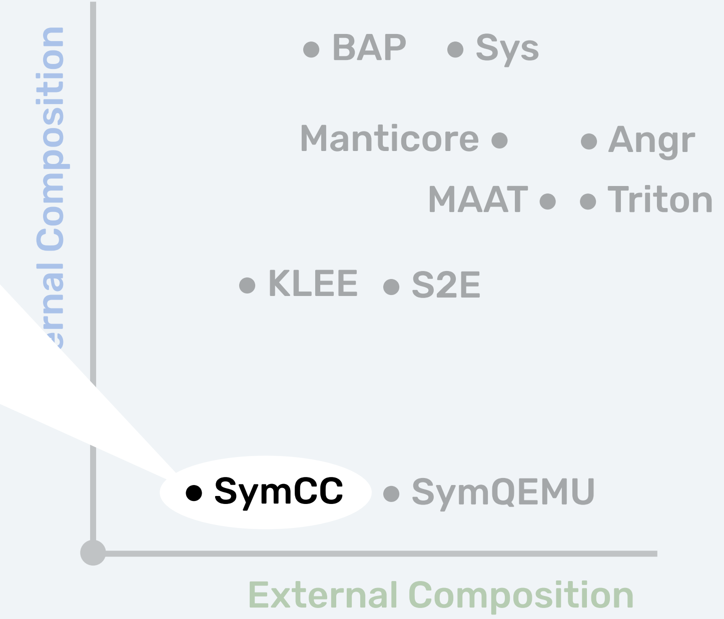
- \+ **Works on *everything*, e.g. Windows drivers, Linux userspace**
  - ± Runtime use requires manual intervention

- ± **Some of the benefits and all of the drawbacks of mashing three systems together: LLVM, QEMU, and KLEE/SymCC**
  - – Complex internal model ⇒ complex external model
  - \+ Works with LLVM



How symbolic executors measure up

- − Focused on running programs from the beginning, i.e. `main`; looks like a fancy fuzzer
- ± Has a similar "runtime" concept to KLEE (Really, Sym is the runtime and CC is a dfsan-like instrumentation)
    - ? Missed opportunity: be a symbolic property testing game-changer for languages targeting LLVM

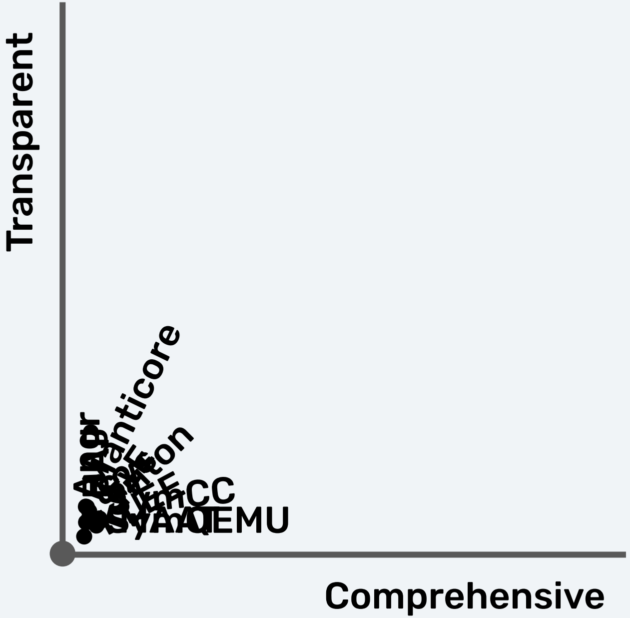


Bug-finding systems should be…

- **Composable**
  - Internal: combine existing results to derive new results
  - External: outputs can be used as inputs to other tools, tool fits into a larger workflow
- **Comprehensive**
  - Zero false-negatives at targeted bug class
  - Known presence of false-negatives can be mitigated by huge upside
- **Transparent**
  - Tool complexity often leads to unpredictable outcomes
  - Behavior should be *predictable*, results should be *explainable*, limitations *understandable*
  - Context is important: “this `free` is bad” is unsatisfactory

SE is neither transparent nor comprehensive

- **SSE appears comprehensive**
  - Explores all feasible paths 🤣
- **DSE appears transparent**
  - Symbolic pointers 🤔
- **Heuristics are *everywhere***
  - Address endemic problems (e.g. scalability, precision)
  - *Reduce* comprehensiveness (e.g. skip this state)
  - *Reduce* transparency (harder to predict, interaction of heuristics leads unexpected blind spots)



Heuristics: short-term gain, long-term pain

- **Developing SE tools begets "solving” SE problems**
  - **Fundamental**: Large state space
  - **Incidental**: Eagerly materializing states at forks causes state explosion
  - **Accidental**: Unpredictability from state scheduling, pruning
- **Heuristics reduce transparency, comprehensiveness**
  - Misaligned incentives: **Heuristics are often touted as novel contributions!**
- **Heuristics lead to accidental agency**
  - Takes decision-making power out of the hands of human bug-finders
  - Heuristics can interact in unexpected and unpredictable ways
  - Ideal: Externalize heuristics as much as possible

KLEE today and tomorrow

KLEE's superpower is its runtime

- **Written in *same* language as the target**
  - **Benefit**: Interface directly with target program entities in domain language

- **Subject to the *same* symbolic execution as the target**
  - Compiled to LLVM bitcode
    - Same approach is also used by SymCC, DIVINE
  - Can implement library code or system call models in runtime
  - **Benefit**: Fork *inside* of model implementation

- **Extensible with "special function handlers"**
  - Special runtime APIs for creating symbolic arrays, enabling heuristics, cancelling forks
  - **Drawback**: Impossible to create a symbolic value without involving memory objects 😠

KLEE's runtime hasn't reached its potential

- **Special function handlers are a cute hack**
  - **Theory**: Tell KLEE about program properties of interest to analyst, e.g. via `assert()`
  - **Illusion**: Trick KLEE into doing what you intend to do by indirectly having it fork down control-flow paths, and then cancel the uninteresting states
  - **Reality**: KLEE has *no way* of representing or using properties

- **KLEE has no persistent, runtime-accessible knowledge base**
  - **Properties as symbolic events, concrete facts in the knowledge base**
  - `input(Time, Data)`, `alloc(Time, Addr, Size)`, `free(Time, Addr)`, *etc.*

- **Manual knowledgebase implementations lack internal composition**
  - Why? KLEE has to *interpret the implementation mechanics*! Falls back on the fork hack
  - **Ideally, you want to be able to trivially query/compose properties**

KLEE's interpreter has taken on too much

- **KLEE codebase grew organically**
  - Scope creep caused by "paper-driven development"
  - Should have asked: could this feature have been implemented through composition?

- **Externalize implementations and policies to proxyable methods**
  - **Reentrant**: Can call other "top" policy methods (i.e. not just current/lower via `this` pointer)
  - **Effectful**: Policy methods *are* the implementations (e.g. symbolic memory can be a proxyable policy)
  - **Eventful**: A policy method that reads memory shouldn't return a value, instead it should schedule the next step(s) of the interpreter with the value(s) read
    - Symbolic execution is event stream processing
    - Policies are event sources/maps/filters/sorts
    - Properties are events in time

Bridging the interpreter/runtime gap

```
#define KLEE_ATTR(attr) __attribute__((…))

KLEE_ATTR(property) bool constant(void *);
KLEE_ATTR(property) bool changed_constant(void *);

KLEE_ATTR(wrapper:PyObject_CallMethod)
PyObject *call(PyObject *ob, …) {
  // Type confusion if object type has changed!
  assert(constant(&(ob->ob_type)));
  assert(!changed_constant(&(ob->ob_type)));
  return PyObject_CallMethod(ob, …);
}

KLEE_ATTR(wrapper:_PyObject_INIT)
PyObject *init(PyObject *ob, PyTypeObject *tp) {
  PyObject *ob = (PyObject *) _PyObject_INIT(ob, tp);
  constant(&(ob->ob_type));  // Add the property.
  return ob;
}
```

```
class ConstantAfterInitPolicy
    : public ProxyPolicy {
  KnowledgeBase<Expr> constant;
  KnowledgeBase<Expr> changed_constant;

  ConstantAfterInitPolicy(Policy *next_, Module *M)
    : ProxyPolicy(next_),
      constant(M, "constant"),
      changed_constant(M, "changed_constant") {}

  // Called by the interpreter to perform a store
  // to memory.
  void Store(Policy *P, State *S, StoreInst *I,
             Expr A, Expr V) override {

    // Derive a new property from an existing one.
    if (S->Match(constant(A)))
      S->Add(changed_constant(A));

    // Eventually, the next policy implements
    // memory storage.
    next->Store(P, S, I, A, V);
  }
};
```

The next frontier is domain integration

- **Interactive**
  - **Goal**: Mitigate performance problems due to fundamental problems, e.g. large state space
  - **Solution**: Ask a bug-finder what to do when an induction variable is symbolic
  - **Challenge**: Map values from analysis domain (LLVM) back to the bug-finder domain (source)
- **Flexible**
  - **Goal**: Drill down on specific paths of interest
  - **Solution**: Want under-constrained for some values, over-constrained or concrete for others
  - **Challenge**: Enable bug-finder to start in the middle of a function
- **Dynamic**
  - **Goal**: Operate very large codebases without llvm-linking all modules together
  - **Solution**: I have ideas that integrate nicely with build chain; come ask me!
  - **Challenge**: Extra indirection, initializers, etc.

MLIR is the future

LLVM IR: A blessing and a curse

Blessings

- **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**
- **Grad students can make papers out of LLVM passes**

Curses

- **Many unspecified LLVM dialects**
  - -O0 *vs.* -O1 *vs.* -O2 *vs.* -O3
  - ABI-specific intrinsics, ABI lowering of types
- **Debug information is unreliable**
- **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` 🤣
- **Legacy: API and grad student churn**
  - Many tools stuck on LLVM 3.x, 4.x, 5.x, *etc.*
  - Many tools will never work with opaque `ptrs`

Different IRs are good for different things

Level	Pros	Cons
High	• Close to bug-finder domain • Explicit abstractions, control-flow • Explicit intra-object boundaries	• Verbose, not efficiently analyzable • Missing implicit behaviors (e.g. C++ destructor calls)
Medium		• Doesn't really exist today?
Low	• Efficiently analyzable	• High-level abstractions, types, control-flow lost to optimization (inlining, hoisting/sinking, folding) • Loop and other program invariants less clear
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

The best fit for an analysis might be far from a bug-finder's domain

Different IRs are good for different things

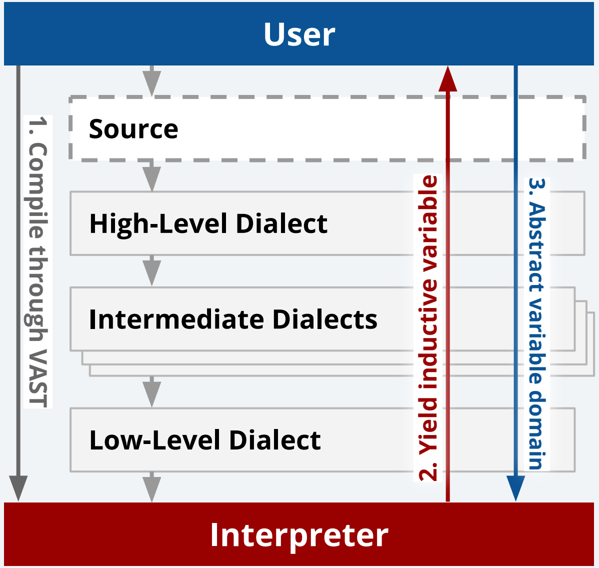
Level	Pros	Cons
High	• Close to bug-finder domain • Expl[illegible] abstracti[illegible], contro[illegible] w • Expl[illegible] ntra-obj[illegible] bounda[illegible]	• Verbose, not efficiently analyzable • M[illegible] impl[illegible] behaviors (e.g. C++ destructor calls)
Medium		• Doe[illegible] really [illegible] t today?
Low	• Effic[illegible] ly analyz[illegible] e	• High[illegible] vel abst[illegible] tions, types, control-flow lost to opti[illegible] ation (i[illegible] ing, hoisting/sinking, folding) • Loo[illegible] nd othe[illegible] ogram invariants less clear
Binary	• Bug-exploiter do[illegible] in • Blurred object b[illegible] ndaries (easier to evaluate buffer [illegible] rflows) • Succinct	• Blurred object boundaries (hard to analyze) • Unreliability of debug info, symbols • Tight coupling of control-flow, type, variable recovery

- Analysis efficiency ↓
- Execution faithfulness ↓
- Exploit domain ↓
- Abstraction loss ↓
- Logic bug domain ↑

Bug-finding needs full stack visibility

- **MLIR: Multi-level intermediate representation**
  - Like LLVM, but supports user-defined dialects
    - Dialects can be used to represent different abstraction levels
    - Can transform dialects, e.g. make a custom dialect for operations on a `std::vector`
    - Stop writing C parsers that produce custom IRs
    - Make a custom MLIR dialect instead
  - Mostly structured control-flow

- **VAST produces MLIR from Clang ASTs**
  - High-level dialect with high-level types, control-flow constructs
  - Medium-level dialect for type lowerings
  - Low-level dialect, MLIR embedding of `-O0` LLVM
  - Open source: https://github.com/trailofbits/vast



Parting thoughts from industry

Bug-finding tools are for bug-finders

- Bug-finders are skilled tool-users with an existing workflow
- Composition, especially with other tools in the workflow, dictates tool use
- Comprehensiveness and transparency dictate tool adoption
- Results should be presented in the domain of the bug-finder
- Today’s KLEE has the right capabilities but the wrong interfaces
- Tomorrow’s libKLEE should empower bug-finders by externalizing heuristics
- Bugs and their exploits cross abstraction levels, program analysis must follow
- VAST enables tailoring analysis domains to the tool and result domains to the bug-finder

Measurement crisis

- **Symbolic execution comparing favorably to fuzzing is *not* inspiring!**
  - Why invest time for possibility of marginal improvements?
  - **To overcome a tried-and-true process, the promised upside must be significant**
    - Adoption is an uphill battle because the status quo gets the job done
    - New approaches, especially sophisticated ones, *look risky*
    - Triton, SATURN have seen adoption as binary deobfuscation game-changers
- **Misaligned incentives**
  - "Novelty" appearing in the introduction of a paper begs the question
    - Everyone has their "redo SAGE phase", not everyone has an idle cluster of Intel Xeon's
  - What horrible hacks helped you achieve that novelty or maximize those metrics?
    - **Transparency**: Are your tool's outcomes predictable, explainable, understandable?
    - **Comprehensiveness**: Does your tool have blind spots?

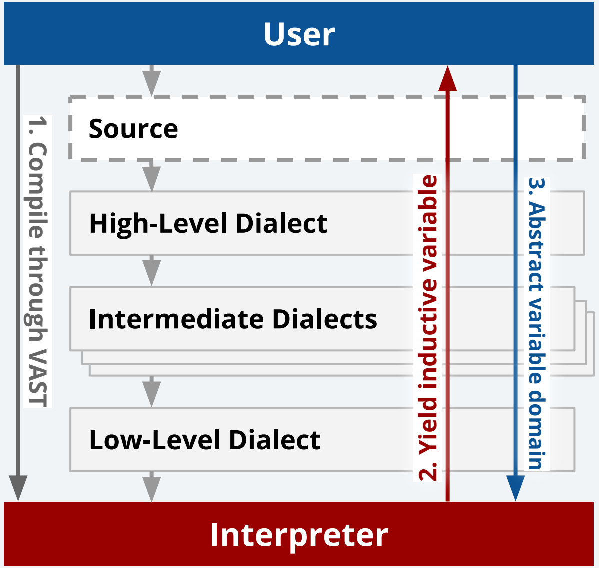
Bug-finding has to be human-centered

- **Adoption of SE in RE/VR proves monoliths are not desirable**
  - SE tools designed as extensible *libraries*, not push-button solutions
  - Easy to tailor to one's target, integrate into existing workflow (IDA Pro, Binary Ninja)
- **Target domain of tools (instructions) matches human-analyzed domain**
  - Externalize heuristics: require human analysts to make decisions / configure features
  - Actionable extension: observation of what's going on isn't sufficient
    - Context recovery challenge: optimized LLVM values are not meaningful, source code is
- **Bug-finders should set the objectives, not tools**
  - Code coverage maximization is not a universal objective
  - Sometimes want under-constrained, sometimes over-constrained, sometimes mixed
  - Enable bug-finders to actually express, record, and reason over properties of interest

VAST's dialects bridge the semantic gap

- **Next-generation symbolic execution needs to reliably integrate with bug-finders *in their domain***
  - Interpret at a low-level
  - Relate results and queries at a high level

- **Next-generation software verification should start with VAST dialects**
  - Stop writing C parsers that produce custom IRs
  - Make a custom MLIR dialect instead



Bug-finding productivity is a function of...

- **Skill and determination**
- **Focused effort**
  - Look at the code that matters
  - Understand the context and the critical paths through the code
- **Reliable tools**
  - Tools must be reliable, limitations must be minimized or well-understood
  - Helpful to have a mental model for predicting tool behavior
- **Leverage**
  - Compose tools or results to narrow focus, expand capabilities
  - Synergies abound
    - Tired: 99% false-positive rate
    - Wired: False-positives can be opportunities to improve code comprehension

Properties as first-class entities

- **Move logic out of the interpreter and into the runtime**
  - Properties should be implemented as a named, typed, possibly symbolic metadata store
  - Operate on target program data structures (e.g. named to track a target-specific property, or reference a target-specific memory location)
  - Expressed in domain of the bug-finder, i.e. C or C++, in the KLEE runtime

- **Bind uninterpreted SMT functions solver to runtime functions**
  - Tired: Attributed extern declarations mirrored into SMT solver as uninterpreted functions
  - Wired: Configure *when* properties are checked with special function handlers
  - Inspired: Deduce properties by defining property functions as compositions of other property functions (e.g. Datalog)

Properties as first-class entities: an example

```
#define KLEE_ATTR(...) \
  __attribute__((annotate("klee:" #__VA_ARGS__))

bool ref_count(void *) KLEE_ATTR(property);

void *malloc_wrapper(size_t sz) KLEE_ATTR(wrapper:PyObject_Malloc) {
  PyObject *ptr = (PyObject *) PyObject_Malloc(sz);
  ref_count(&(ptr->ref_cnt));
  return ptr;
}

bool store(void *ptr, size_t size) KLEE_ATTR(event);
bool load(void *ptr, size_t size) KLEE_ATTR(event);
bool store_to_ref_count(void *ptr, size_t sz) KLEE_ATTR(derived_event) {
  return store(ptr, sz) && ref_count(ptr);
}
```

KLEE's missing API

- **KLEE's internal and external composition story is uninspiring**
  - Special function handlers are extension points in name only
    - Require modifying and recompiling KLEE
    - Hard to tailor to the target program
  - Problematic focus on `main`
    - Reasonable for comparing against a fuzzer on coreutils, bad for big programs
    - **Exploring option/input parsing code should be a choice by the bug-finder**; it's a prerequisite now
- **Special function handlers should be a KLEE API**
  - Let bug-finders tell KLEE when to check properties
  - **Missed opportunity**: JIT-compile specially attributed runtime functions to native handlers
    - Then they can have access to target-specific data structures / functions
- **Today's KLEE should be an application of tomorrow's libKLEE**