The Software Analysis Workbench as a Platform for Verification Research

Aaron Tomb
Galois, Inc.

Workshop on Democratizing Software Verification
July 14, 2019

galois

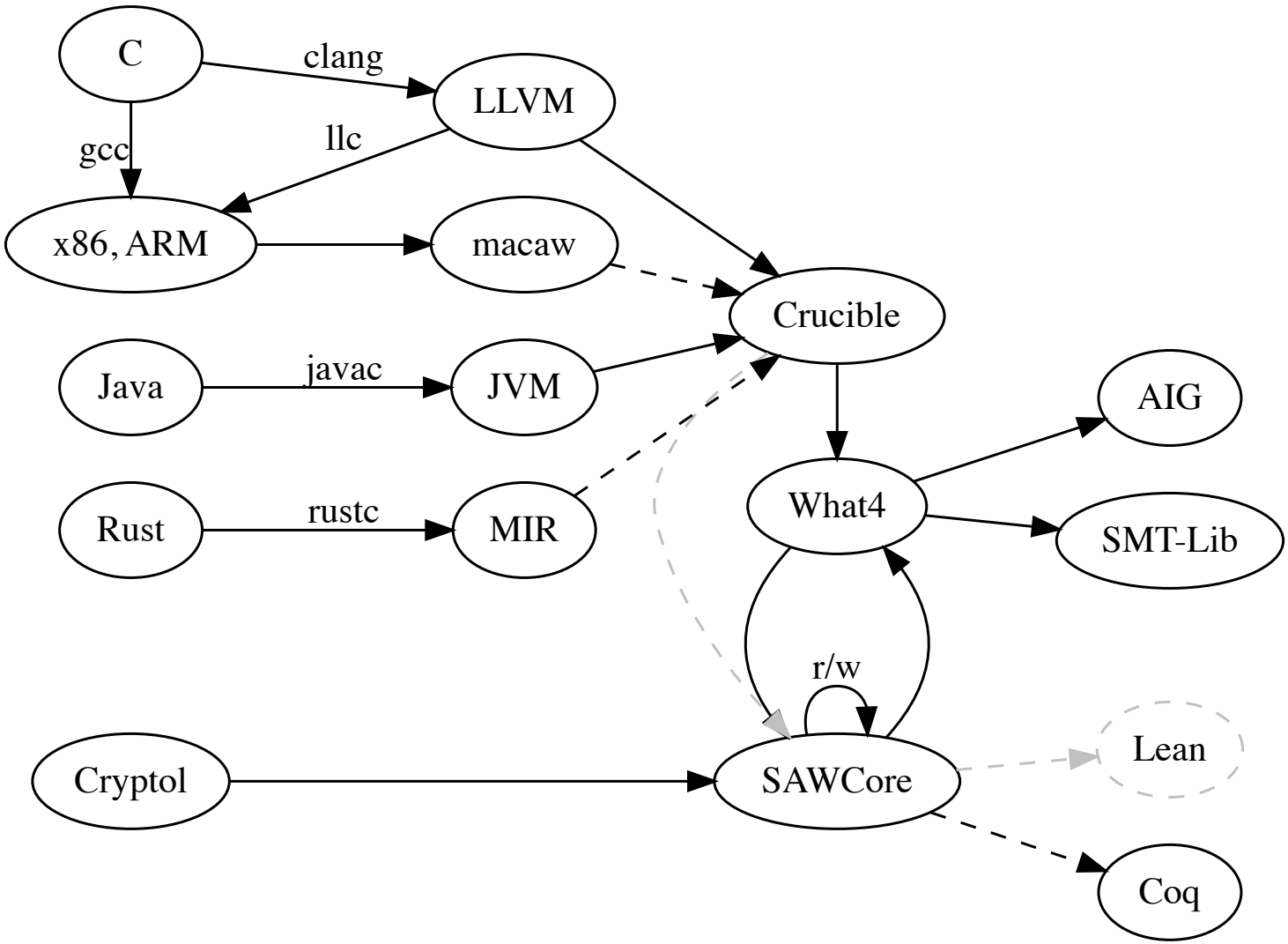
The importance of verification infrastructure

- Program analysis requires lots of infrastructure
  - Front end: parsers, type checkers, file location, ...
  - Verification: encoding of semantics, connection to solvers, ...
- Real languages are complicated!
  - Using same definitions as other developer tools is ideal
  - Front ends starting to go this way (e.g., `clang`)
  - Semantics should be re-usable, too!
- Two topics
  - Approach provided by the Software Analysis Workbench
  - Ideas about what an ideal platform might look like

The Software Analysis Workbench (SAW)

- SAW = Software Analysis Workbench
  - Software: many languages
  - Analysis: many types of analysis, focused on functionality
  - Workbench: flexible interface, supporting many goals
- What separates it from other systems?
  - One view: compiler from imperative code to functional code
  - Captures all functional behavior, simplifying later if necessary
  - Uses efficient representations tuned to equivalence checking
  - Strong bit vector reasoning support
  - Focus on practicality over novelty
- Open source (BSD3), implemented in Haskell, available now

SAW architecture (language translation view)



galois

Scripting

- Interaction of SAW components scriptable
- Currently, a custom language
  - Simple typed functional language
  - Primitives to construct, transform, prove SAWCore terms
- Working on migrating to Python
  - Communicates with server using JSON-RPC
  - Incremental computation, to avoid redundant work
  - Full power of Python for flexible scripting

Example: FFS

```
uint32_t ffs_ref(uint32_t w) {
  int c, i = 0;
  if(!w) return 0;
  for(c = 0; c < 32; c++)
    if((1 << i++) & w) return i;
  return 0;
}

uint32_t ffs_imp(uint32_t w) {
  uint32_t r, n = 1;
  if(!(w & 0xffff)) { n += 16; w >>= 16; }
  if(!(w & 0x00ff)) { n += 8;  w >>= 8; }
  if(!(w & 0x000f)) { n += 4;  w >>= 4; }
  if(!(w & 0x0003)) { n += 2;  w >>= 2; }
  return (w) ? (n+((w+1) & 0x01)) : 0;
}
```


galois

Example: Running verification

Script

```
m <- llvm_load_module "ffs.bc";
ref <- crucible_llvm_extract m "ffs_ref";
imp <- crucible_llvm_extract m "ffs_imp";
time (prove_print yices {{ \x -> ref x == imp x }});
```

Output

```
# saw ffs_llvm.saw
Loading file "ffs_llvm.saw"
Valid
Time: 0.030429s
```

Language front ends and capabilities

- LLVM (C, C++)
  - Direct bitcode reader, supports LLVM 3.5 - 7.0 (8 soon!)
- JVM (Java)
  - All bytecode except `invokedynamic`
- Others experimental, without SAWScript bindings for now
  - MIR (Rust)
  - Machine code (x86, ARM, PPC)

Crucible language

- Language semantics definable by translation to common IR
  - Similar in goals to Boogie, Why3
- Imperative, with unstructured control flow
  - SSA representation available
  - Merge point identification for symbolic execution
  - External s-expression syntax for non-Haskell front ends
- Extensive expression language
  - Strongly-typed (including for library clients)
- Some missing features
  - Concurrency
  - Exceptional control flow

Crucible example

```
(defun @fact ((n Integer)) Integer
  (start at-the-beginning:
    (let donep (< n 1))
    (branch donep recur: done:))
  (defblock recur:
    (let next (funcall @fact (- n 1)))
    (let val (the Integer (* next n)))
    (return val))
  (defblock done:
    (let init (the Integer 1))
    (return init)))
```

SAWCore language

- Dependently-typed functional language, essentially the calculus of constructions
  - Built-in rewriter, optional hash-consing
  - Some simple proof tactics
- Many primitives, for smooth interaction with ensure external provers
- Two forms of external syntax: human-optimized and machine-optimized

SAWCore example

Direct function definitions

```
implies : Bool -> Bool -> Bool;
implies = \(a:Bool) (b:Bool) -> or (not a) b;
```

Function specifications by axiomatic equality

```
axiom not_eq :
  (b:Bool) -> Eq Bool (not b) (ite Bool b False True);
```

Machine-optimized syntax for identity function on `Bool`

```
SAWCoreTerm 3
1 Global Prelude.Bool
2 Var 0
3 Lam x 1 2
```

What4 language

- First-order formula representation
- Similar in sprit to SMT-Lib
  - More primitives
  - Not just predicates
- Aggressive simplification
- Back ends for SMT-Lib, AIG, CNF
  - Plus some solver-specific SMT formats
- Strongly-typed internally
  - Haskell clients can't construct ill-typed terms
- No external syntax so far

Symbolic execution in Crucible

- Agressively merges paths at post-dominators
  - Recently-added option for individual path exploration
- Uses efficient What4 formulas internally
- Fully unrolls loops
  - Doesn't terminate with some programs
  - Option to impose fixed bound, leading to BMC-like analysis
  - Complete if it terminates normally
- Accomplishes translation from imperative to functional language

Memory model

- For well-structured languages, references similar to ML
- For LLVM, machine code, a custom memory model
  - Pointer = allocation ID + offset
    - Integers are pointers with ID 0
  - Heap = "list of writes"
    - Writes append, reads search
    - Maps to optimize some common cases
  - Compound objects stored whole
  - Conversions between large types and bytes

Other analyses

- Abstract interpretation
  - Can be instantiated with arbitrary domains, transfer functions
  - Lightly used: information flow and optimizing symbolic execution
- Weakest preconditions
  - Simpler memory model, experimental
- Monadic functional program generation
  - Very experimental, but promising
- Horn clause generation
  - Very experimental, but promising

Use in other projects

Pros

- Open source
- Composed of modular translations
- Various subsets of IRs can be re-used independently
- Strong types help avoid mistakes
- Server available for flexible scripting from other programs

Cons

- Haskell code unfamiliar to many developers, and uses many language extensions
- May be hard to link to other tools
- Some IRs lack external syntax (so far)

Towards an ideal verification platform

Ideal platform desiderata

Current reality

- Program verification tools, in reality, either:
  - require huge investment in the complex details of real language syntax, semantics, or
  - don't work with real programs.

What I want in a platform

- Easy development of new analyses
- Ability to apply new analyses to complex programs
- Insulation from complexities of real languages
- Flexibility and possibility of precision
- Clarity about soundness, relation to semantics
- Trustworthy implementation

Semantics-based approach

- Include a description of the full language semantics
  - Separates the effort of defining a verification technique from defining semantics
  - Allows new techniques to experiment on real code
  - Ensures that different verification techniques agree on semantics
- Machine-readable semantics can help get compilers and other tools right
- What's the right way to describe semantics?
  - Operational? Denotational? Axiomatic?
  - I think denotational semantics have been underexplored

Compatible front ends

- Real code will use all the features a compiler supports[1]
- So it's key to use standard compiler front end:
  - at least for parsing and probably type checking,
  - better: simplified IR (GHC Core, JVM, LLVM, MIR, SIL)
- Ideally, compiler or interpreter would be based on mechanized description of language semantics
  - Verifiers could build on the same description

[1] See comments about Coverity buying old compilers off eBay in *A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World*

One possibility for trustworthy foundations

- DSL embedded in proof assistant for several translations
  - Surface syntax $\rightarrow$ AST (BNF?)
  - AST $\rightarrow$ IR (rewrite system?)
  - IR $\rightarrow$ denotation (coinduction over a free monad?)
  - System logic for expressions, predicates
- Lean looks like an appealing tentative option
  - Tactics in the logic
  - Strong reflection capabilities
  - Focus on efficient code generation

Speculative opportunities

- Symbolic execution as abstract interpretation?
  - No abstraction
  - No guarantee of termination
  - But it fits the shape
- Everything as abstract interpretation?
  - Theoretically, almost everything is
  - Engineering effort advantage?
  - Efficiency advantage? Disadvantage?
- Strong duals between WP and SP (essentially symbolic execution)
  - Can they share implementation?

Summary

SAW

- A flexible, open source analysis platform
- Modular, with many IRs and strong types to help guide design and avoid mistakes
- On GitHub: https://github.com/GaloisInc/saw-script

The platform of the future

- Let's make "verification toolkits" that gives for semantics what parser generators have given for syntax
- Ideally, language designers would define semantics in a way compatible with such a toolkit