The Software Analysis Workbench

Aaron Tomb
Galois, Inc.

2018 Dagstuhl Seminar on Program Equivalence
April 10, 2018

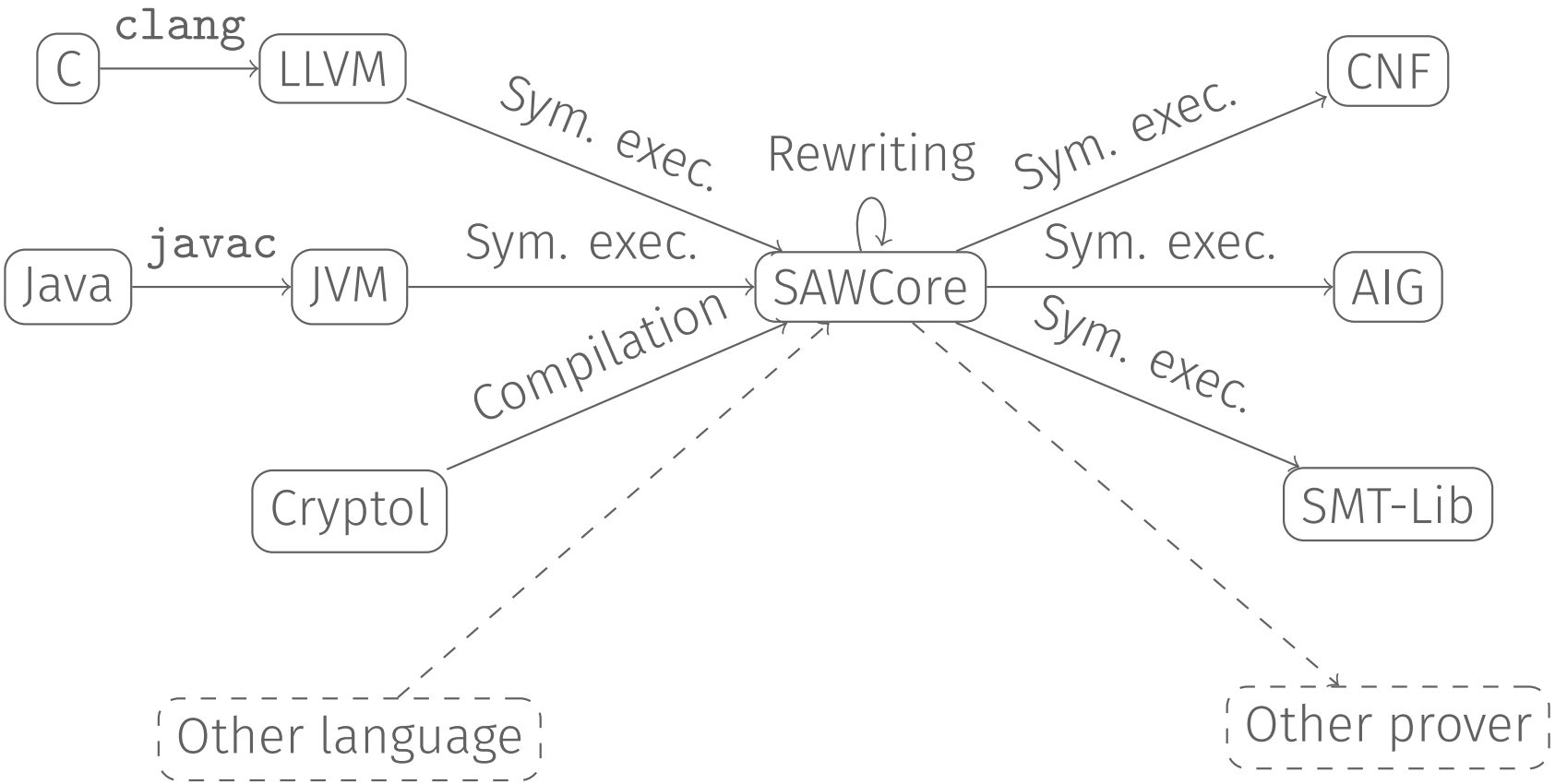
galois

What is SAW?

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

galois

What's Behind This?



galois

The Cryptol Language

A single, high-level specification for (cryptographic) algorithms

- Cryptol goals
  - Appropriate for cryptography
  - Natural
  - Concise
  - Similar to existing notation
  - Appropriate for execution and verification
- Language features
  - Statically-typed functional language
  - Sized bit vectors (type level naturals)
  - Stream comprehensions (stream diagrams)
- Convenient language for any program over finite data

Relationship Between Cryptol and SAW

- Cryptol is the expression language of SAW's scripting language, SAWScript
- Many functions that operate on SAWCore terms (type `Term`)
- Built-in support for Cryptol syntax
  - Translated automatically into `Term` objects with `{{ ... }}`
- Supports proofs purely on Cryptol
- Allows proofs comparing Cryptol to real-world implementations

Simple Proofs

- Proofs work on `Term` objects that have result type `Bit`
- Includes any Cryptol function with result type `Bit`, as well as terms coming from other sources
- The best-performing prover depends heavily on the problem

```
sawscript> let {{ p (x:[4096]) = x+x+x+x == x*4 }}
sawscript> time (prove abc {{ p }})
```

Simple Proofs

- Proofs work on `Term` objects that have result type `Bit`
- Includes any Cryptol function with result type `Bit`, as well as terms coming from other sources
- The best-performing prover depends heavily on the problem

```
sawscript> let {{ p (x:[4096]) = x+x+x+x == x*4 }}
sawscript> time (prove abc {{ p }})
Time:     3.433s
Valid
```

Simple Proofs

- Proofs work on `Term` objects that have result type `Bit`
- Includes any Cryptol function with result type `Bit`, as well as terms coming from other sources
- The best-performing prover depends heavily on the problem

```
sawscript> let {{ p (x:[4096]) = x+x+x+x == x*4 }}
sawscript> time (prove abc {{ p }})
Time:     3.433s
Valid
sawscript> time (prove z3  {{ p }})
```

Simple Proofs

- Proofs work on `Term` objects that have result type `Bit`
- Includes any Cryptol function with result type `Bit`, as well as terms coming from other sources
- The best-performing prover depends heavily on the problem

```
sawscript> let {{ p (x:[4096]) = x+x+x+x == x*4 }}
sawscript> time (prove abc {{ p }})
Time:     3.433s
Valid
sawscript> time (prove z3  {{ p }})
Time:     0.006s
Valid
```

Simple Equivalence Checking

```
static int ffs_ref(int word) {
    if(word == 0) return 0;
    for(int cnt = 0, i = 0; cnt < 32; cnt++)
        if(((1 << i++) & word) != 0) return i;
    return 0;
}
static int ffs_imp(int i) {
    byte n = 1;
    if((i & 0xffff) == 0) { n += 16; i >>= 16; }
    if((i & 0x00ff) == 0) { n +=  8; i >>=  8; }
    if((i & 0x000f) == 0) { n +=  4; i >>=  4; }
    if((i & 0x0003) == 0) { n +=  2; i >>=  2; }
    if(i != 0) { return (n+((i+1)&0x01)); } else { return 0; }
}
```

```
ffs_cls <- java_load_class "FFS";
ffs_ref <- java_extract ffs_cls "ffs_ref" java_pure;
ffs_imp <- java_extract ffs_cls "ffs_imp" java_pure;
prove abc {{ \x -> ffs_ref x == ffs_imp x }}; // Valid: 0.014s
```

galois

Equivalence Checking With Pointers

```
void swap_xor(uint8_t *x, uint8_t *y) {
    *x = *x ^ *y;
    *y = *x ^ *y;
    *x = *x ^ *y;
}
```

```
let swap_spec = do {
    x <- fresh_var "x" i32;
    y <- fresh_var "y" i32;
    xp <- alloc i32;
    yp <- alloc i32;
    points_to xp (term x);
    points_to yp (term y);
    execute_func [xp, yp];
    points_to xp (term y);
    points_to yp (term x);
};
llvm_verify "swap_xor" [] swap_spec abc;
```


galois

Constructing Models with Symbolic Execution

- Imperative $\longrightarrow$ functional via symbolic execution
- For straight-line code, symbolic value of any variable at end is a pure function of symbolic inputs
- Model memory ephemerally
- For branches, merge symbolic states at post-dominators
  - A nested application of the if-then-else function
- Unroll loops
  - So they're just a case of sequential branching
  - Can terminate more frequently by SAT-checking branch conditions
- Have also experimented with using fixpoint combinator
  - And plan to experiment with explicit heaps

The SAWCore Language

- Dependently typed core calculus
- Takes some inspiration from CiC, some from MLTT
- Represented efficiently with hash-consed DAGs
- Large number of primitives
  - Covering, e.g., the SMT `QF_AUFBV` theory
  - Even though these can be (and have been!) defined in SAWCore, too
- Two type checkers
  - One from surface syntax to explicitly type terms
  - One on explicitly typed terms (incomplete)
  - No guarantee that they agree!

Proofs in SAWCore

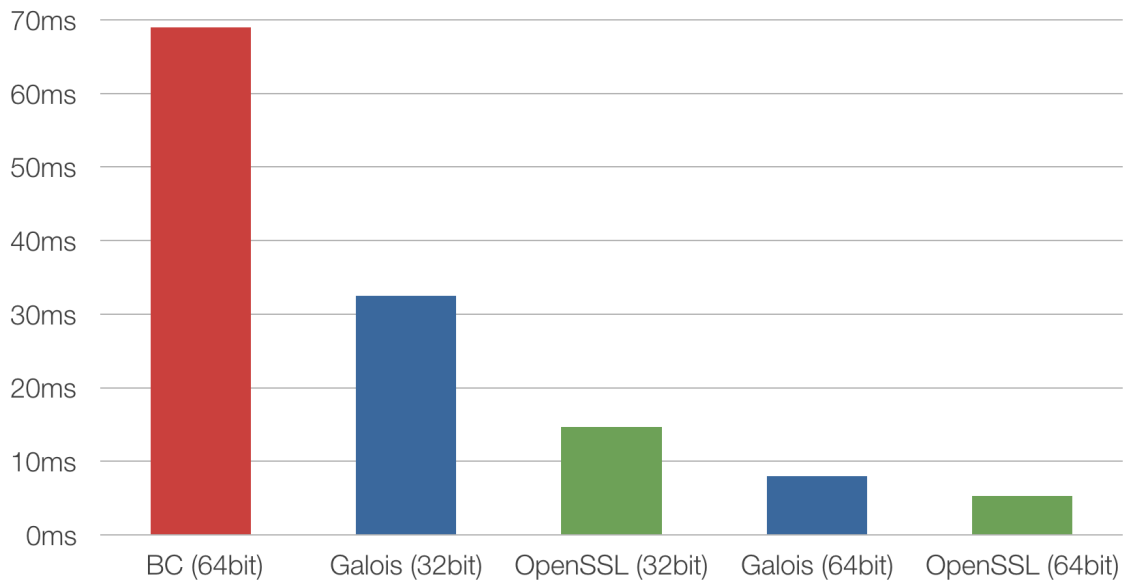
- As a type theory, two notions of proof in SAWCore
  - Showing inhabitant of equality type
  - Showing a Boolean term equivalent to `True`
- Proofs can be performed by SAT and SMT solvers
  - Several tactics for transformation in advance
- Hand-constructed proof objects are more powerful
  - But no tactics at this level
- Terms of type `Eq a b` are theorems
- Terms of structure `a == b` can be theorems
  - If shown valid by external prover

Rewriting in SAWCore

- Rewriting the main proof tactic available
- Both `Eq a b` and `a == b` can be used as rules
  - The latter normally proved before use, but optional
- Symbolic execution can be thought of as an instance of rewriting
- Some limitations:
  - No conditional rewriting (so far)
  - No auto-simplification for associativity, commutativity, etc.
- Other interactive provers are more flexible
  - Though in some cases less efficient (we routinely process multi-GB terms)
  - And not as integrated with automated provers or model extractors

Case Study: ECDSA

- Elliptic Curve Digital Signature Algorithm (ECDSA)
- In-house Java code, tuned for speed and verifiability
  - Available with SAW distribution
- ~ 2400 Java LOC
- ~ 1600 spec LOC
- ~ 1500 proof script LOC (largely plumbing)
- Proof completes in < 5m

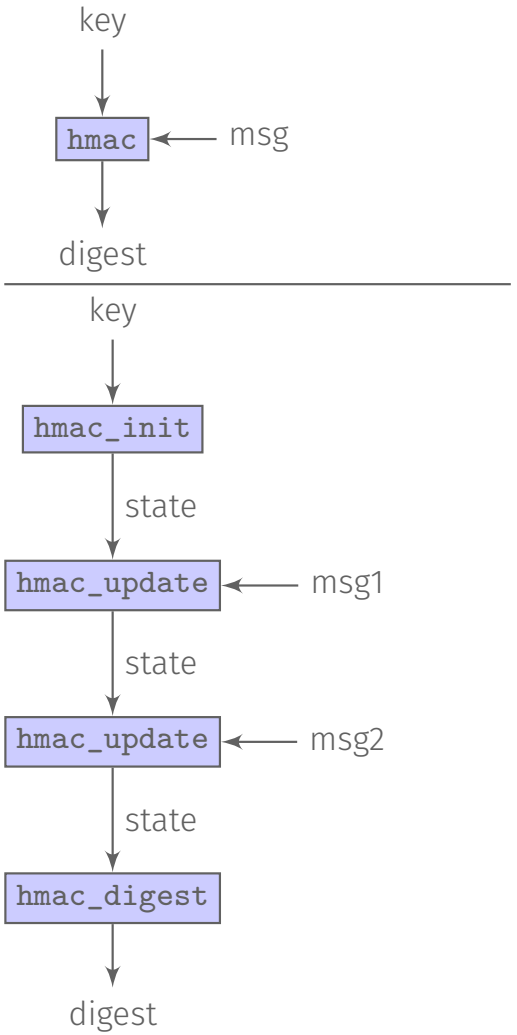


galois

Case Study: HMAC

s2n

- Amazon TLS implementation
- Code from official `s2n` repository
- ~ 15 (top-level) spec LOC (monolithic function)
- ~ 300 C LOC (iterative code)
- ~ 400 script LOC (all plumbing)
- Proofs for various fixed message sizes
  - <1m per proof



galois

SAW: efficient proofs about imperative programs via translation to functional programs + SAT/SMT

- Practical system, used to verify real-world code, such as:
  - AES from OpenSSL
  - HMAC, DRBG from s2n
  - ECDSA from Galois
  - Portions of several Curve25519 implementations
- Differences from other equivalence checking tools:
  - Built on complete symbolic execution (for now)
  - Focus on practicality, scalability
  - Flexible coordination via SAWScript

Contributors

Contributors include Aaron Tomb, Adam Foltzer, Adam Wick, Andrey Chudnov, Andy Gill, Benjamin Barenblat, Ben Jones, Brian Huffman, Brian Ledger, David Lazar, Dylan McNamee, Eddy Westbrook, Edward Yang, Eric Mertens, Eric Mullen, Fergus Henderson, Iavor Diatchki, Jeff Lewis, Jim Teisher, Joe Hendrix, Joe Hurd, Joe Kiniry, Joel Stanley, Joey Dodds, John Launchbury, John Matthews, Jonathan Daugherty, Kenneth Foner, Kyle Carter, Ledah Casburn, Lee Pike, Levent Erkök, Magnus Carlsson, Mark Shields, Mark Tullsen, Matt Sottile, Nathan Collins, Philip Weaver, Robert Dockins, Sally Browning, Sam Anklesaria, Sigbjørn Finne, Stephen Magill, Thomas Nordin, Trevor Elliott, and Tristan Ravitch.

Resources

- Cryptol
  - Web: `http://cryptol.net`
  - GitHub: `https://github.com/GaloisInc/cryptol`
- Software Analysis Workbench
  - Web: `https://saw.galois.com`
  - GitHub: `https://github.com/GaloisInc/saw-script`
- HMAC verification blog post:
  - `https://galois.com/blog/2016/09/verifying-s2n-hmac-with-saw/`