From Testing to Proof using Symbolic Execution

Aaron Tomb
Galois, Inc.

StrangeLoop
September 28, 2017

Agenda

- Installation (~15m)
- Basic overview (~15m)
- Exercises (~20m)
- More flexible verification (~15m)
- Exercises (~20m)
- Composition and more (~15m)
- Exercises (~20m)

Installing SAW

- SAW 2017-09-06 from https://saw.galois.com/builds/nightly/
- Z3 4.5.0 from https://github.com/Z3Prover/z3/releases
- SAW builds available for:
  - CentOS 6 (32-bit and 64-bit) (anything similar to older RedHat)
  - CentOS 7 (64-bit) (anything similar to newer RedHat)
  - macOS (64-bit)
  - Ubuntu 14.04 (64-bit) (anything similar to recent-ish Debian)
  - Windows (64-bit)
- VirtualBox VM image and local(ish) tarballs:
  - http://10.129.176.174:8000/
    - SAW Workshop (Debian).vdi
    - saw/* (files for various platforms)
    - z3/* (files for various platforms)
  - Login: root/saw-workshop, saw/saw-workshop

galois

What is SAW?

- A tool to construct models of program behavior
  - Works with C (LLVM), Java (JVM), and others in progress
  - Also supports specifications written in Cryptol
- Models can then be proved to have certain properties
  - Equivalence with specifications
  - Guarantees to return certain values
- Proofs generally done using automated reasoning tools
  - So similar level of effort to testing
  - Uses a technique called symbolic execution, plus SAT/SMT

|galois|

- Rather than testing individual cases, state general properties
- Then can test those properties on specific values
  - Manually selected
  - Randomly generated
- For example, this function should always return a non-zero value:

```c
int add_commutes(uint32_t x, uint32_t y) {
    return x + y == y + x;
}
```

- The QuickCheck approach is a common implementation of this paradigm

|galois|

- Say we're using the XOR-based trick for swapping values:

```
void swap_xor(uint32_t *x, uint32_t *y) {
  *x = *x ^ *y;
  *y = *x ^ *y;
  *x = *x ^ *y;
}
```

- Focus on values, since that's where the tricky parts are
  - Pointers used just so it can be a separate function

A Specification for Swapping

```c
void swap_direct(uint32_t *x, uint32_t *y) {
  uint32_t tmp;
  tmp = *y;
  *y = *x;
  *x = tmp;
}

int swap_correct(uint32_t x, uint32_t y) {
  uint32_t x1 = x, x2 = x, y1 = y, y2 = y;
  swap_xor(&x1, &y1);
  swap_direct(&x2, &y2);
  return (x1 == x2 && y1 == y2);
}
```

Manual Swap Testing

```c
int main() {
  assert(swap_correct(0, 0));
  assert(swap_correct(0, 1));
  assert(swap_correct(1, 0));
  assert(swap_correct(32, 76));
  assert(swap_correct(0, 0xFFFFFFFF));
  assert(swap_correct(0xFFFFFFFF, 0xFFFFFFFF));
  return 0;
}
```

- Advantages
  - Ensures that you will always test important values
  - Carefully chosen tests can cover many important cases quickly
- Disadvantages
  - May miss classes of inputs that you didn't think of
  - Non-deterministics: different runs may have different results

galois

```
int main() {
  for(int idx = 0; i < 100; i++) {
    uint32_t x = rand();
    uint32_t y = rand();
    assert(swap_correct(x, y));
  }
  return 0;
}
```

- Advantages
  - Better theoretical coverage of input space
  - Number of tests limited only by available processing power
- Disadvantages
  - May miss important classes of inputs that are easy to identify by hand

galois

Translating Programs to Pure Functions

- $\lambda x.\ x + 1$ is a function
  - takes an argument x, and returns $x + 1$
- `swap_direct`: $\lambda(x, y).\ (y, x)$
- `swap_xor`: $\lambda(x, y).\ (x \oplus y \oplus x \oplus y \oplus y, x \oplus y \oplus y)$
  - but $x \oplus x \equiv 0$ and $x \oplus 0 \equiv x$
- Translation achieved in SAW using a technique called *symbolic execution*
  - Think: an interpreter with expressions in place of values
  - Every variable's value at the end is an expression representing *all possible values* it might take

SAT and SMT Solvers

- Automated provers for mathematical theorems
  - Such as: $\forall x, y.\ (x \oplus y \oplus x \oplus y \oplus y, x \oplus y \oplus y) \equiv (y, x)$
- SAT = Boolean SATisfiability
- SMT = Satisfiability Modulo Theories
- Almost magic for what they can do. SAT can encode:
  - Fixed-size bit vectors (even multiplication, but slowly)
  - Bit manipulation operations (and, or, xor, shifts)
  - Arrays of fixed sizes
  - Conditionals
- SMT adds things like:
  - Linear arithmetic on integers (addition, subtraction, multiplication by constants)
  - Arrays of arbitrary size
  - Uninterpreted functions

Automated Verification vs. Testing

- Advantages
  - Ensures that you will test all possible input values
  - Sometimes faster than testing
- Disadvantages
  - Applicable to a smaller class of programs than testing
  - Sometimes much slower than testing

Verifying Swap Correctness

```
// Load the bitcode file generated by Clang
swapmod <- llvm_load_module "swap.bc";

// Extract a formal model of `swap_correct`
harness <- llvm_extract swapmod "swap_correct" llvm_pure;

// Use ABC prover to show it always returns non-zero
prove_print abc {{ \x y -> harness x y != 0 }};
```

(In `swap_harness.saw`)

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

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

```c
int ffs_imp_correct(uint32_t x) {
  return ffs_imp(x) == ffs_ref(x);
}

int main() {
  assert(ffs_imp_correct(0x00000000));
  assert(ffs_imp_correct(0x00000001));
  assert(ffs_imp_correct(0x80000000));
  assert(ffs_imp_correct(0x80000001));
  assert(ffs_imp_correct(0xF0000000));
  assert(ffs_imp_correct(0x0000000F));
  assert(ffs_imp_correct(0xFFFFFFFF));
  return 0;
}
```

- Same pros and cons as for the swap example

```
int main() {
  for(int idx = 0; i < 100; i++) {
    uint32_t x = rand();
    assert(ffs_imp_correct(x));
  }
  return 0;
}
```

```
m <- llvm_load_module "ffs.bc";

correct <- llvm_extract m "ffs_imp_correct" llvm_pure;

print "Proving ffs_imp_correct always returns true...";
prove_print abc {{ \x -> correct x == 1 }};
```

(In `ffs_harness.saw`)

```
m <- llvm_load_module "ffs.bc";

ref <- llvm_extract m "ffs_ref" llvm_pure;
imp <- llvm_extract m "ffs_imp" llvm_pure;

// Following equivalent to \x -> ref x == imp x
prove_print abc {{ ref === imp }};
```

(In `ffs_eq.saw`)

Exercises: FFS

0. Run the equivalence proofs in `ffs_harness.saw` and `ffs_eq.saw`
1. Port the FFS code to use `uint64_t`

- Translate both reference and implementation
- Which one is wrong?

2. Try to break the FFS code, in obvious and subtle ways

- Can you make it do the wrong thing and not be caught?

3. Try to discover the “haystack” bug in `ffs_bug`

- Use random testing (`ffs_bug_fail.saw`)
  - Increase the number of tests and see how long it takes
  - Try a similar case with `uint64_t`
- Use `ffs_bug.saw` to find it with a SAT solver

Pointers: Verifying XOR Swap Without Wrapper

```
m <- llvm_load_module "xor-swap.bc";
// void swap_xor(uint32_t *x, uint32_t *y);
let swap_spec = do {
    x <- crucible_fresh_var "x" (llvm_int 32);
    y <- crucible_fresh_var "y" (llvm_int 32);
    xp <- crucible_alloc (llvm_int 32);
    yp <- crucible_alloc (llvm_int 32);
    crucible_points_to xp (crucible_term x);
    crucible_points_to yp (crucible_term y);
    crucible_execute_func [xp, yp];
    crucible_points_to xp (crucible_term y);
    crucible_points_to yp (crucible_term x);
};
crucible_llvm_verify m "swap_xor" [] true swap_spec abc;
```

(In `swap.saw`)

galois

```
m <- llvm_load_module "xor-swap.bc";

let ptr_to_fresh nm ty = do {
    x <- crucible_fresh_var nm ty;
    p <- crucible_alloc ty;
    crucible_points_to p (crucible_term x);
    return (x, p);
};

let swap_spec = do {
    (x, xp) <- ptr_to_fresh "x" (llvm_int 32);
    (y, yp) <- ptr_to_fresh "y" (llvm_int 32);
    crucible_execute_func [xp, yp];
    crucible_points_to xp (crucible_term y);
    crucible_points_to yp (crucible_term x);
};
```

Exercises: Code with Pointers

1. Try to break the XOR-based swapping in some way and run the proof
   - Use `swap.saw` or `swap_harness.saw`
2. Write a buggy version and use SAW to find inputs for which it's correct

- These would be bad test cases!

3. Write a script to prove the FFS test harness using `crucible_llvm_verify`

- You'll need `crucible_return {{ 1 : [32] }}` and `crucible_term`
- You won't need `crucible_alloc` or `crucible_points_to`

More Complex Verifications, In General

- Verifications in SAW consist of three phases
  - Initialize a starting state
  - Run the target code in that state
  - Check that the final state is correct
- Commands like `llvm_extract` just simplify a common case
- When running the target code, we can sometimes use previously-proven facts about code it calls

galois

Composition: Verifying Salsa20 (C code)

```c
uint32_t rotl(uint32_t value, int shift) {
  return (value << shift) | (value >> (32 - shift));
}

void s20_quarterround(uint32_t *y0, uint32_t *y1,
                      uint32_t *y2, uint32_t *y3) {
  *y1 = *y1 ^ rotl(*y0 + *y3, 7);
  // ... and three more
}

void s20_rowround(uint32_t y[static 16]) {
  s20_quarterround(&y[0], &y[1], &y[2], &y[3]);
  // ... and three more
}
```

Composition: Verifying Salsa20 (SAW code)

```
let quarterround_setup : CrucibleSetup () = do {
  (p0, y0) <- ptr_to_fresh "y0" i32;
  // ... and three more
  crucible_execute_func [p0, p1, p2, p3];
  let zs = {{ quarterround [y0,y1,y2,y3] }};
  crucible_points_to p0 (crucible_term {{ zs@0 }});
    // ... and three more
};

let rowround_setup = do {
  (y, p) <- ptr_to_fresh "y" (llvm_array 16 i32);
  crucible_execute_func [p];
  crucible_points_to p (crucible_term {{ rowround y }});
};
```

Sidebar: Array Sizes and Looping

- With the current version of SAW, programs must be finite
  - SAT-based proofs need to know how many bits are involved
  - Inputs need to have fixed sizes
  - All pointers must point to data of known size
  - All loops need to execute a bounded number of types
- But Salsa20 can operate on any input size
  - So we prove it correct separately for several possible sizes
  - Our original version had a bug because of this!
- Future versions are likely to relax these restrictions

galois

Exercises: Composition

1. Run the monolithic and compositional proofs
   - `salsa.saw` and `salsa-compositional.saw`
2. Compare the timing of the two
   - When checking multiple sizes, how does it compare?
   - How many sizes before it becomes better?
3. Try to break the code and see what happens
   - First try a leaf function
   - Then try the top-level function
4. Can you break it so that one size succeeds but another fails?

galois

Sidebar: Fuzzing for Property Based Tests

- Klee is another LLVM symbolic execution system
- Doesn't aim for complete coverage, but very powerful
  - Better at finding bugs than SAW
  - Not generally usable for verification
- Associated fuzz testing system, `libfuzzer`
  - Includes a `main` function that calls fuzzing harness

```
int LLVMFuzzerTestOneInput(const uint8_t *Data,
                           size_t Size) {
  DoSomethingInterestingWithMyAPI(Data, Size);
  return 0;
}
```

```
let fuzzer_spec n = do {
  let ty = llvm_array n i8;
  (pdata, data) <- ptr_to_fresh "data" ty;
  crucible_execute_func
    [ pdata
    , crucible_term {{ `n : [64] }}
    ];
  crucible_return (crucible_term {{ 0 : [32] }});
};

m <- llvm_load_module "fuzztarget.bc";
for [1, 10, 20, 100] (\sz ->
  crucible_llvm_verify m "LLVMFuzzerTestOneInput"
    [] true (fuzzer_spec n) abc);
```

Other Things Available in SAW

- Support for various languages
  - Others that compile to LLVM (simple C++ and Rust have been tested, others YMMV)
  - Languages that compile to JVM (only Java known to work well, but others might)
  - Coming soon: Rust, Go, some degree of machine code
- Some interactive proof tactics
  - Mostly rewriting with user-defined rules
  - Coming soon: bindings to external interactive provers, including Lean and Coq

Final Points

- Resources
  - SAW web site: https://saw.galois.com
  - Cryptol web site: https://cryptol.net
  - SAW documentation
    - Tutorial: https://saw.galois.com/tutorial.html
    - Manual: https://saw.galois.com/manual.html
  - Cryptol documentation: https://cryptol.net/documentation.html
- I'll be around all day, and happy to talk more.
- And if this sort of thing interests you, Galois is hiring!