Continuous Verification of Critical Software

Aaron Tomb

IEEE SecDev — September 30, 2018

SAW Basics (Session A: 1:30 – 3:00)

- Installation and configuration
- How to think about verification
- Basic overview of SAW
- More general use of SAW

Advanced Use and CI (Session B: 3:30 – 5:00)

- Compositional verification
- Advanced proof techniques
- Build integration and CI
- Examples from s2n

- Option 1: manual installation
    - LLVM + Clang: http://releases.llvm.org/download.html (most versions work, including those from Xcode)
    - Yices: http://yices.csl.sri.com/ (most tested with v2.6.0)
    - Z3: https://github.com/Z3Prover/z3/releases/tag/z3-4.7.1
    - SAW: https://saw.galois.com/builds/nightly/ (tested with 2018-08-26)
- Option 2: Docker container
    - `docker pull atomb/secdev18-saw`
    - `docker run --rm -it atomb/secdev18-saw`
- Also, check out examples and slides
    - git clone https://github.com/atomb/secdev18-saw

galois

What is SAW?

- A tool to construct models of program behavior
  - Works with **C (LLVM)**, Java (JVM), and others in progress
  - Supports high-level specifications in the **Cryptol** language
- Models can then be proved to have certain properties
  - Equivalence with specifications
  - Guarantees to return certain values
- Proofs generally done using automated reasoning tools
  - Uses symbolic execution plus SAT/SMT
  - Similar level of effort to testing
  - Automatically repeatable once configured; great for CI

Thinking About Verification

- Rather than testing individual cases, state general properties
- For example, this function should always return a non-zero value:

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

- Then can test those properties on specific values
  - Manually selected
  - Randomly generated
- The QuickCheck approach is a common implementation of this paradigm

- 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

(See `xor-swap.c`.)

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);
}
```

(See `direct-swap.c` and `swap-correct.c`.)

Manual Swap Testing

```
int main() {
  assert(swap_correct(0, 0));
  assert(swap_correct(0, 1));
  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
  - Hard to get high coverage of a large input space

Random Swap Testing

```
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 inputs that are easy to identify by hand
  - Non-deterministic: different runs may have different results

Translating Programs to Pure Functions

- Pure function: equal arguments go to equal results
- $\lambda x.\ x + 1$ takes an argument x, and returns $x + 1$
- 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
- `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 $x \oplus x \equiv 0$, $x \oplus 0 \equiv x$, and $x \oplus y \equiv y \oplus x$

galois

SAT and SMT Solvers

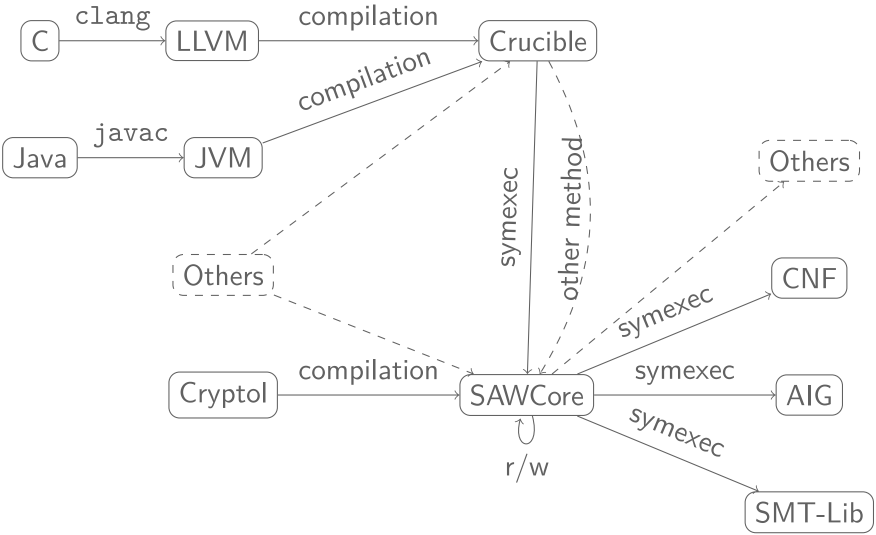
- Automated provers for mathematical theorems
  - Such as: $\forall x, y.\ (x \oplus y \oplus x \oplus y \oplus y, x \oplus y \oplus y) \equiv (y, x)$
- SAT = Boolean SATisfiability. Can encode:
  - Fixed-size bit vectors (even multiplication, but slowly)
  - Arrays of fixed sizes
  - Conditionals
- SMT = Satisfiability Modulo Theories. Adds things like:
  - Linear arithmetic on integers
  - 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
  - Practical for a smaller class of programs than testing
  - Sometimes much slower than testing

Basics of SAW

The Components of SAW



Generating LLVM Bitcode for SAW

- SAW supports C through LLVM
- The Clang compiler translates C source to LLVM "bitcode"
- Basic use, to produce `file.bc`:

  ```
  clang -c -emit-llvm file.c
  ```

- Use `llvm-link` to combine bitcode files
- For large, complex projects, `wllvm` is convenient. For example:

  ```
  CC=wllvm CXX=wllvm++ ./configure && make
  ```

- Get `wllvm` here:
  https://github.com/travitch/whole-program-llvm

- Language used to control behavior of SAW
- Not a general-purpose language
  - Basic mechanism for combining built-in functions
- Statically typed with Haskell-style `do` notation
- The `Term` type represents SAWCore models
- Uses special quoting, between `{{` and `}}`, for Cryptol expressions
- Any value in scope with type `Term` is visible in Cryptol

Verifying Swap Correctness

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

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

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

(See `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;
}
```

(See `ffs.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;
}
```

- Even exhaustive testing possible in this case
  - But not for 64-bit inputs

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

imp_correct <- llvm_extract m "ffs_imp_correct" llvm_pure;
bug_correct <- llvm_extract m "ffs_bug_correct" llvm_pure;

set_base 16;
print "Proving ffs_imp_correct always returns true...";
prove_print abc {{ \x -> imp_correct x == 1 }};
print "Failing to prove ffs_bug_correct returns true...";
prove_print abc {{ \x -> bug_correct x == 1 }};
```

(See `ffs_harness.saw`)

```
m <- llvm_load_module "ffs.bc";
print "Extracting functional models...";
ref <- llvm_extract m "ffs_ref" llvm_pure;
imp <- llvm_extract m "ffs_imp" llvm_pure;
print "Comparing reference and implementation...";
// The === operator compares functions directly
r <- time (prove abc {{ ref === imp }});
print r;
```

(See `ffs_eq.saw`)

1. Port the FFS code to use `uint64_t`
   - Translate both reference and implementation
   - Try to prove equivalence (and don't worry if you fail)
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`, uses SAW for testing)
     - 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

More Complex Verification

General Verification Structure

- Verifications in SAW consist of three phases
  - Initialize a (symbolic!) starting state
  - Run the target code (symbolically) in that state
  - Check that the final state is correct (using automated provers)
- For LLVM, encapsulated in `crucible_llvm_verify`
  - The `llvm_extract` command just simplifies a common case
- When running the target code, we can sometimes use previously-proven facts about code it calls (or assumed facts about external code)

galois

Using crucible_llvm_verify

```
m <- llvm_load_module "foo.bc";
let foo_spec = do {
    // Set up the initial state

    crucible_execute_func [/* some argument */];

    // Check the final state
};
crucible_llvm_verify m "foo" [] true foo_spec abc;
```

- Introduce symbolic variables with `crucible_fresh_var`
- Specify heap layout with `crucible_alloc`, `crucible_points_to`
- All work before or after `crucible_execute_func`, with different meanings

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;
```

(See `swap.saw`, comparing all versions)

Simplifying the XOR Swap specification

```
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);
};
crucible_llvm_verify m "swap_xor" [] true swap_spec abc;
```

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`

Composition

Review: SAW Basics

- SAW executes functions symbolically to translate them into pure functions
- Symbolic state must be configured before execution
  - Similar to writing a test harness
  - Need to allocate memory, indicate symbolic values
- After execution, SAT and SMT solvers can prove properties of result state
- Use `crucible_fresh_var` to create symbolic values
  - The name is just for debugging
- Use `crucible_execute_func` to specify arguments, division between pre and post state

- Use `crucible_alloc` to specify that allocated memory exists
- Use `crucible_points_to` to specify where values are stored in the heap

```
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);
};
```

```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: Specifying 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 }});
};
```

Composition: Verifying Salsa20 (SAW code)

```
m <- llvm_load_module "salsa20.bc";
let verify f ovs spec =
  crucible_llvm_verify m f ovs true spec abc;
qr <- verify "s20_quarterround" []   quarterround_setup;
rr <- verify "s20_rowround"     [qr] rowround_setup;
```

- Pass results of prior verification into later verification
- Can have multiple previously-verified facts about one function
    - For example, different array sizes
    - Used only for the top level of Salsa20

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 times
- 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 SAW versions are likely to relax these restrictions

galois

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

Additional SAW Details

Built-in Prover Tradeoffs

- SAW supports many back-end provers
  - `abc`: good for very bit-level things (and linked in)
  - `yices`: good for compositional bit vector problems
  - `z3`: good for integer problems
- Others are available, but less frequently useful
  - I usually use `yices`

- It's also possible to write theorems to files and prove later
  - `write_cnf` and `offline_cnf` support standard SAT solvers
  - `write_aig` and `offline_aig` support AIG-based solvers (like ABC as an external program)
  - `write_smtlib2` and `offline_smtlib2` support SMT solvers (like Yices and Z3 as external programs)
- The tactic versions (`offline_*`) always succeed, with a warning
- See `write_cnf.saw`

```
sawscript> let {{ f x y = (x : [8]) + y }}
sawscript> {{ f }}
f

sawscript> let t = unfold_term ["f"] {{ f }}
sawscript> t
let { x@1 = Prelude.Vec 8 Prelude.Bool }
 in \(x :: x@1) -> \(y :: x@1) ->
    Cryptol.ecPlus x@1
      (Cryptol.PArithSeqBool (Cryptol.TCNum 8)) x   y

sawscript> rewrite (cryptol_ss ()) t
let { x@1 = Prelude.Vec 8 Prelude.Bool }
 in \(x :: x@1) -> \(y :: x@1) -> Prelude.bvAdd 8 x y
```

(See `unfold.saw`.)

```
let {{ f x y = (x : [8]) + y }};
let {{ g x y = 2 * (f x y) }};
let {{ h x y = (f x y) + (f x y) }};
let {{ prop x y = g x y == h x y }};
let {{ prop2 x y = h x y == 2*x + 2*y }};

print "Proving prop fully unfolded:";
prove_print yices {{ prop }};
print "Proving prop with f uninterpreted:";
prove_print (unint_yices ["f"]) {{ prop }};
print "Proving prop2 fully unfolded:";
prove_print yices {{ prop2 }};
print "Proving prop2 with f uninterpreted (should fail):";
prove_print (unint_yices ["f"]) {{ prop2 }};
```

(See `unint.saw`.)

```
let {{
    f x y = (x : [8]) + y
    g x y = (y : [8]) + x
    h x y = (f x y) + (g x y)
}};
f_eq_g <- prove_print abc {{ \x y -> f x y == g x y }};
print f_eq_g;

t1 <- unfold_term ["h"] {{ \x y -> h x y == 2*(f x y) }};
print_term t1;

t2 <- rewrite (addsimp f_eq_g empty_ss) t1;
print_term t2;

prove_print (unint_yices ["g"]) t2;
```

(See `rewrite.saw`.)

Proof Scripts

- Commands like `prove` and `crucible_llvm_verify` use proof scripts
- Instead of just a prover, can have tactics in a `do` block
- Show current goal with `print_goal`
- Term manipulation
  - `unfold_term` becomes `unfolding`
  - `rewrite` becomes `simplify`
- Finishing proofs
  - Skip proof with `assume_unsat`
  - Match `True` with `trivial`
  - Invoke prover (`abc`, `yices`, `z3`, etc.)

galois

Debugging Symbolic Execution Failure

- Invalid memory reads/writes
  - Usually a result of failing to declare an input (argument, field, global)
  - Compile with `-g` to see where the read is happening
- Failure to terminate
  - Intrinsic non-termination in the target program?
  - Try branch satisfiability checking
  - Complex memory operations
- Check whether it's in symbolic execution or proof
  - With prover other than `abc`, check `saw` CPU use

Continuous Integration

Onward to Continuous Integration

- SAW verification configurations are manual
- But executing a script can be automated
- Scripts need to change only in a few cases
  - If the expected behavior of the code being checked changes
  - If the inputs or outputs of the code change
- So CI systems can automatically re-check many code changes
  - Changes to other parts of code
  - Changes that preserve functionality

Continuous Integration with Travis

1. Download SAW binaries
   - From https://saw.galois.com/builds/nightly
2. Download prover binaries
   - Yices from http://yices.csl.sri.com
   - Z3 from https://github.com/Z3Prover/z3/releases
3. Unpack everything and put binaries in the `PATH`
4. Build your code (maybe using `wllvm`)
5. Run `saw` on a script file
   - Caching binaries (e.g., on S3) can improve reliability

galois

Tips and Tricks for Travis

- General procedure for binary tarballs (SAW, Yices, Z3)

```
- curl -L --retry 3 URL --output /tmp/foo.tar.gz
- tar -xzvf /tmp/foo.tar.gz
- export PATH=$PATH:$(pwd)/foo-0.1
```

- See `.travis.yml` in the repository for this talk for details
- To experiment with changes, use branches (and tell Travis to build them)
- Travis output: https://travis-ci.org/atomb/secdev18-saw

Exercises: Travis

1. Create a new repository running SAW under Travis:
   - Try the `ffs` example, or `xor-swap`
   - Get it to pass
2. Experiment with changes to the code
   - Create a branch for a change (so you can delete a series of messy commits)
   - Make a mistake, push, observe the output
   - Travis example: https://travis-ci.org/atomb/secdev18-saw/builds/435026975

galois

Proof Maintenance

- Unchanged code easily reproved
- Modified code with identical functionality automatically reproved
- Some changes require script changes
  - Changes in types (e.g., `struct` definitions) used by function
  - Changes in number or shape of inputs or outputs
- We're working on ways to automate some changes, when possible

NIST Document

$$HMAC(k, m) = H((k_0 \oplus opad) \| H((k_0 \oplus ipad) \| m))$$

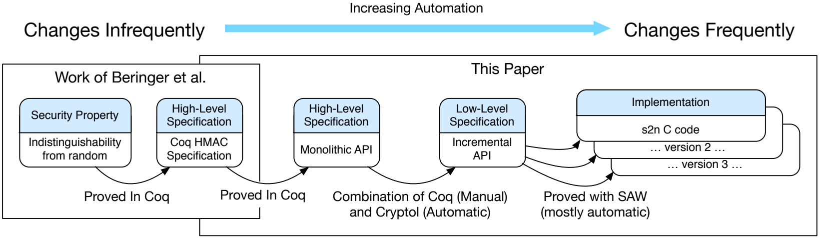
Cryptol

```
hmac k m = H (opad # split (H (ipad # m)))
  where
    k0 = kinit H k
    opad = [kb ^ 0x5C | kb <- k0]
    ipad = [kb ^ 0x36 | kb <- k0]
```

C and Verification

- ~200 lines of code, multiple functions
- Low-level spec corresponding to each function
- Proof between Cryptol specs in SAW (one lemma in Coq)
- One proof for each of several hash algorithms, message sizes

- Abstract Cryptol spec derived from RFC or NIST document
- Concrete spec matching what s2n implements
  - Incremental HMAC, subset of handshake protocol
- Proof of refinement between two Cryptol specs
- Proof of equivalence between low-level Cryptol and C



galois

- One of many testing and analysis tasks for s2n
- Run on every commit
- Completes in about the same time as the concrete tests
- See: https://travis-ci.org/awslabs/s2n

✓	# 2206.3	Xcode: xcode8 C	S2N_LIBCRYPTO=openssl-1.0.2-fips BUILD_S2N=true TESTS=inte...	9 min 8 sec
✓	# 2206.4	Xcode: xcode8 C	S2N_LIBCRYPTO=libressl BUILD_S2N=true TESTS=integration GC...	9 min 34 sec
✓	# 2206.5	Xcode: xcode8 C	S2N_LIBCRYPTO=openssl-1.1.0 OPENSSL_ia32cap="~0x2000002...	9 min 50 sec
✓	# 2206.6	Xcode: xcode8 C	S2N_LIBCRYPTO=openssl-1.1.0 LATEST_CLANG=true TESTS=fuzz ...	17 min 25 sec
✓	# 2206.12	Xcode: xcode8 C	TESTS=sawHMAC SAW_HMAC_TEST=sha256 SAW=true GCC6_RE...	7 min 27 sec
✓	# 2206.13	Xcode: xcode8 C	TESTS=sawHMAC SAW_HMAC_TEST=sha384 SAW=true GCC6_RE...	8 min 33 sec
✓	# 2206.14	Xcode: xcode8 C	TESTS=sawHMAC SAW_HMAC_TEST=sha512 SAW=true GCC6_RE...	9 min 10 sec
✓	# 2206.15	Xcode: xcode8 C	TESTS=tls SAW=true GCC6_REQUIRED=false	12 min 48 sec
✓	# 2206.16	Xcode: xcode8 C	TESTS=sawHMACFailure SAW=true	7 min 4 sec

galois

Wrapping Up

More Complex Examples

- Proof of equivalence between Cryptol and Java versions of ECDSA
  - https://github.com/GaloisInc/saw-script/tree/master/examples/ecdsa
- Proof of the absence of undefined behavior or assertion failures
  - https://github.com/GaloisInc/saw-script/tree/master/examples/sv-comp
- Proof of HMAC, DRBG, and the TLS handshake in s2n
  - GitHub: https://github.com/awslabs/s2n/tree/master/tests/saw
  - CAV Paper: https://link.springer.com/chapter/10.1007/978-3-319-96142-2_26

galois

Future of SAW

- Better support for unbounded programs
  - Data: variable-size and variable-shape heap structures
  - Control: unbounded iteration
- More flexible/powerful scripting language
  - May expose SAWScript functions as an API for other languages (Python?)
- Analysis of more languages
  - Partial support for Rust, Go, various forms of machine code

Contributors

Aaron Tomb, Adam Foltzer, Adam Wick, Andrei Stefanescu, Andrey Chudnov, Andy Gill, Benjamin Barenblat, Ben Jones, Bill Hallahan, Brian Huffman, Brian Ledger, Daniel Wagner, David Christiansen, David Lazar, Dylan McNamee, Edward Yang, Edwin Westbrook, Eric Mertens, Eric Mullen, Fergus Henderson, Guannan Wei, Iavor Diatchki, Jeff Lewis, Jim Teisher, Joe Hendrix, Joe Hurd, Joe Kiniry, Joel Stanley, Joey Dodds, John Launchbury, John Matthews, Jonathan Daugherty, Kenneth Foner, Keshav Kini, Kevin Quick, Kyle Carter, Langston Barrett, Ledah Casburn, Lee Pike, Levent Erkök, Luke Maurer, Magnus Carlsson, Mark Shields, Mark Tullsen, Matt Sottile, Max Orhai, Nathan Collins, Philip Weaver, Robert Dockins, Robby, Sally Browning, Sam Anklesaria, Sigbjørn Finne, Stephanie Weirich, Thomas Nordin, Trevor Elliott, Tristan Ravitch

Continuing with SAW

- 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
  - These examples and slides: https://github.com/atomb/secdev18-saw
- If this sort of thing interests you, Galois is hiring!