Multi-Language and Multi-Prover Verification with SAWScript

Aaron Tomb
Galois, Inc.

High Confidence Software and Systems (HCSS'15)
May 5, 2015

galois

Overview

- SAWScript is a special-purpose scripting language
  - User interface to the Software Analysis Workbench (SAW)
  - Provides ability to to construct, manipulate, and query mathematical models of software semantics
  - Support for LLVM, JVM, Cryptol
  - Proof of properties using various automated provers
  - Various internal transformation and composition techniques
- Verification infrastructure that was originally built-in to Cryptol [2]
- We'll walk through SAWScript with a number of specific examples
  - Using C, Java, Cryptol, sometimes together
  - Focusing on cryptography, but not exclusively

© 2015 Galois, Inc. All Rights Reserved.

Key Concepts

- SAWScript focused on manipulating values of `Term` type
  - Internally, a `Term` is a term in a dependently-typed λ-calculus
  - Staged type checking: each `Term` checked when constructed at runtime
- Ties together lots of infrastructure
  - Language parsing and translation
  - Theorem proving (mostly automated now, but potentially manual)
- Interactive REPL or batch scripts
  - Both composed mostly of command sequences

© 2015 Galois, Inc. All Rights Reserved.

Simple Examples

- Beginnings

```
sawscript> print "Hello, world!"
Hello, world!
```

- Cryptol expressions (`{{_}} : Cryptol -> Term`)

```
sawscript> print {{ [0x01, 0x02] + [0x03, 0x04] }}
[4, 6]
```

- Iteration, construction of new `Terms`

```
sawscript> for [ {{1}}, {{2}}, {{3}} ] (\x -> print {{ x == 3 }})
False
False
True
```

© 2015 Galois, Inc. All Rights Reserved.

Simple Proofs

- Satisfiability and validity checking 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 {{ thm (x:[4096]) = x+x+x+x == x*4 }}
sawscript> time (prove abc {{ thm }})
```

© 2015 Galois, Inc. All Rights Reserved.

Simple Proofs

- Satisfiability and validity checking 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 {{ thm (x:[4096]) = x+x+x+x == x*4 }}
sawscript> time (prove abc {{ thm }})
Time:  3.433s
Valid
```

© 2015 Galois, Inc. All Rights Reserved.

Simple Proofs

- Satisfiability and validity checking 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 {{ thm (x:[4096]) = x+x+x+x == x*4 }}
sawscript> time (prove abc {{ thm }})
Time:  3.433s
Valid
sawscript> time (prove z3 {{ thm }})
```

© 2015 Galois, Inc. All Rights Reserved.

Simple Proofs

- Satisfiability and validity checking 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 {{ thm (x:[4096]) = x+x+x+x == x*4 }}
sawscript> time (prove abc {{ thm }})
Time: 3.433s
Valid
sawscript> time (prove z3 {{ thm }})
Time: 0.006s
Valid
```

© 2015 Galois, Inc. All Rights Reserved.

Satisfiability Checking

- You've got 30 coins that add up to $1.09 - what are they?

```
coinPuzzle : [10] -> [10] -> [10] -> [10] -> Bit
coinPuzzle a b c d = (coinCount a b c d 30) && (coinSum a b c d 109)

coinSum : [10] -> [10] -> [10] -> [10] -> [10] -> Bit
coinSum   a b c d s = (a + 5 * b + 10 * c + 25 * d) == s

coinCount : [10] -> [10] -> [10] -> [10] -> [10] -> Bit
coinCount a b c d s =
  (((a + b + c + d) == s) && // the coin count adds up
  (a <= s && b <= s && c <= s && d <= s)) // and we don't wrap
```

```
sawscript> cp <- cryptol_load "Coins.cry"
sawscript> time (sat abc {{ cp::coinPuzzle }})
```

© 2015 Galois, Inc. All Rights Reserved.

Satisfiability Checking

- You've got 30 coins that add up to $1.09 - what are they?

```
coinPuzzle : [10] -> [10] -> [10] -> [10] -> Bit
coinPuzzle a b c d = (coinCount a b c d 30) && (coinSum a b c d 109)

coinSum : [10] -> [10] -> [10] -> [10] -> [10] -> Bit
coinSum   a b c d s = (a + 5 * b + 10 * c + 25 * d) == s

coinCount : [10] -> [10] -> [10] -> [10] -> [10] -> Bit
coinCount a b c d s =
  (((a + b + c + d) == s) && // the coin count adds up
  (a <= s && b <= s && c <= s && d <= s)) // and we don't wrap
```

```
sawscript> cp <- cryptol_load "Coins.cry"
sawscript> time (sat abc {{ cp::coinPuzzle }})
Time:  0.073s
Sat:  [("a",19),("b",4),("c",7),("d",0)]
```

© 2015 Galois, Inc. All Rights Reserved.

Satisfiability Checking

- You've got 30 coins that add up to $1.09 - what are they?

```
coinPuzzle : [10] -> [10] -> [10] -> [10] -> Bit
coinPuzzle a b c d = (coinCount a b c d 30) && (coinSum a b c d 109)

coinSum : [10] -> [10] -> [10] -> [10] -> [10] -> Bit
coinSum   a b c d s = (a + 5 * b + 10 * c + 25 * d) == s

coinCount : [10] -> [10] -> [10] -> [10] -> [10] -> Bit
coinCount a b c d s =
  (((a + b + c + d) == s) && // the coin count adds up
  (a <= s && b <= s && c <= s && d <= s)) // and we don't wrap
```

```
sawscript> cp <- cryptol_load "Coins.cry"
sawscript> time (sat abc {{ cp::coinPuzzle }})
Time:  0.073s
Sat:  [("a",19),("b",4),("c",7),("d",0)]
sawscript> time (sat z3 {{ cp::coinPuzzle }})
```

© 2015 Galois, Inc. All Rights Reserved.

Satisfiability Checking

- You've got 30 coins that add up to $1.09 - what are they?

```
coinPuzzle : [10] -> [10] -> [10] -> [10] -> Bit
coinPuzzle a b c d = (coinCount a b c d 30) && (coinSum a b c d 109)

coinSum : [10] -> [10] -> [10] -> [10] -> [10] -> Bit
coinSum   a b c d s = (a + 5 * b + 10 * c + 25 * d) == s

coinCount : [10] -> [10] -> [10] -> [10] -> [10] -> Bit
coinCount a b c d s =
  (((a + b + c + d) == s) && // the coin count adds up
  (a <= s && b <= s && c <= s && d <= s)) // and we don't wrap
```

```
sawscript> cp <- cryptol_load "Coins.cry"
sawscript> time (sat abc {{ cp::coinPuzzle }})
Time:  0.073s
Sat:  [("a",19),("b",4),("c",7),("d",0)]
sawscript> time (sat z3 {{ cp::coinPuzzle }})
Time:  0.009s
Sat:  [("a",19),("b",7),("c",3),("d",1)]
```

© 2015 Galois, Inc. All Rights Reserved.

Java Reference vs. Implementation

```
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
```

© 2015 Galois, Inc. All Rights Reserved.

Cross-Language Comparison

```
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;
}
```

```
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;
}
```

```
...
ffs_bc <- llvm_load_module "ffs.bc";
ffs_imp <- llvm_extract ffs_bc "ffs_imp" llvm_pure;
prove abc {{ \x -> ffs_ref x == ffs_imp x }}; // Valid: 0.013s
```

© 2015 Galois, Inc. All Rights Reserved.

Larger Verification: DES

```
DES = { encrypt key pt = des pt (expandKey key)
      , decrypt key ct = des ct (reverse (expandKey key)) }
des pt keys = (swap (split last)) @@ FPz
  where pt' = pt @@ IPz
        iv  = [ round (k, split lr)
              | k  <- keys
              | lr <- [pt'] # iv ]
        last = iv @ (width keys - 1)
round (k, [l, r]) = r # (l ^ f (r, k))
f (r, k) = (SBox(k ^ (r @@ EPz))) @@ PPz
swap [a, b] = b # a
```

```
sawscript> m <- cryptol_load "DES.cry"
sawscript> let {{ enc = m::DES.encrypt }}
sawscript> let {{ dec = m::DES.decrypt }}
sawscript> time (prove abc {{ \k m -> dec k (enc k m) == m }})
```

© 2015 Galois, Inc. All Rights Reserved.

Larger Verification: DES

```
DES = { encrypt key pt = des pt (expandKey key)
      , decrypt key ct = des ct (reverse (expandKey key)) }
des pt keys = (swap (split last)) @@ FPz
  where pt' = pt @@ IPz
        iv  = [ round (k, split lr)
              | k  <- keys
              | lr <- [pt'] # iv ]
        last = iv @ (width keys - 1)
round (k, [l, r]) = r # (l ^ f (r, k))
f (r, k) = (SBox(k ^ (r @@ EPz))) @@ PPz
swap [a, b] = b # a
```

```
sawscript> m <- cryptol_load "DES.cry"
sawscript> let {{ enc = m::DES.encrypt }}
sawscript> let {{ dec = m::DES.decrypt }}
sawscript> time (prove abc {{ \k m -> dec k (enc k m) == m }})
Valid
Time: 4.521s
```

© 2015 Galois, Inc. All Rights Reserved.

Larger Verification: DES

```
DES = { encrypt key pt = des pt (expandKey key)
      , decrypt key ct = des ct (reverse (expandKey key)) }
des pt keys = (swap (split last)) @@ FPz
  where pt' = pt @@ IPz
        iv  = [ round (k, split lr)
              | k  <- keys
              | lr <- [pt'] # iv ]
        last = iv @ (width keys - 1)
round (k, [l, r]) = r # (l ^ f (r, k))
f (r, k) = (SBox(k ^ (r @@ EPz))) @@ PPz
swap [a, b] = b # a
```

```
sawscript> m <- cryptol_load "DES.cry"
sawscript> let {{ enc = m::DES.encrypt }}
sawscript> let {{ dec = m::DES.decrypt }}
sawscript> time (prove abc {{ \k m -> dec k (enc k m) == m }})
Valid
Time: 4.521s
sawscript> time (prove z3 {{ m -> dec k (enc k m) == m }})
// Times out
```

© 2015 Galois, Inc. All Rights Reserved.

Compositional Proof: 3DES

```
m <- cryptol_load "DES.cry";
enc <- define "enc" {{ m::DES.encrypt }};
dec <- define "dec" {{ m::DES.decrypt }};
dec_enc <- time (prove abc {{ \k m -> dec k (enc k m) == m }});
enc_dec <- time (prove abc {{ \k m -> enc k (dec k m) == m }});
let ss = simpset [dec_enc, enc_dec];
let {{
  enc3 k1 k2 k3 msg = enc k3 (dec k2 (enc k1 msg))
  dec3 k1 k2 k3 msg = dec k1 (enc k2 (dec k3 msg))
  dec3_enc3 k1 k2 k3 msg = dec3 k1 k2 k3 (enc3 k1 k2 k3 msg) == msg
}};
time (prove do { simplify ss; abc; } {{ dec3_enc3 }});
```

© 2015 Galois, Inc. All Rights Reserved.

Compositional Proof: 3DES

```
m <- cryptol_load "DES.cry";
enc <- define "enc" {{ m::DES.encrypt }};
dec <- define "dec" {{ m::DES.decrypt }};
dec_enc <- time (prove abc {{ \k m -> dec k (enc k m) == m }});
enc_dec <- time (prove abc {{ \k m -> enc k (dec k m) == m }});
let ss = simpset [dec_enc, enc_dec];
let {{
  enc3 k1 k2 k3 msg = enc k3 (dec k2 (enc k1 msg))
  dec3 k1 k2 k3 msg = dec k1 (enc k2 (dec k3 msg))
  dec3_enc3 k1 k2 k3 msg = dec3 k1 k2 k3 (enc3 k1 k2 k3 msg) == msg
}};
time (prove do { simplify ss; abc; } {{ dec3_enc3 }});
```

© 2015 Galois, Inc. All Rights Reserved.

Compositional Proof: 3DES

```
m <- cryptol_load "DES.cry";
enc <- define "enc" {{ m::DES.encrypt }};
dec <- define "dec" {{ m::DES.decrypt }};
dec_enc <- time (prove abc {{ \k m -> dec k (enc k m) == m }});
enc_dec <- time (prove abc {{ \k m -> enc k (dec k m) == m }});
let ss = simpset [dec_enc, enc_dec];
let {{
  enc3 k1 k2 k3 msg = enc k3 (dec k2 (enc k1 msg))
  dec3 k1 k2 k3 msg = dec k1 (enc k2 (dec k3 msg))
  dec3_enc3 k1 k2 k3 msg = dec3 k1 k2 k3 (enc3 k1 k2 k3 msg) == msg
}};
time (prove do { simplify ss; abc; } {{ dec3_enc3 }});
```

```
Valid
Time: 4.694s
Valid
Time: 4.718s
Valid
Time: 0.003s
```

© 2015 Galois, Inc. All Rights Reserved.

The ZUC Stream Cipher

- LFSR-based stream cipher used in GSM protocols [1]
- Version 1.4 of ZUC has weakness due to non-injectivity of initialization function [3]
  - Version 1.5 fixed it
  - Vulnerability in v1.4, and fix in v1.5, have been shown on Cryptol versions of the specification
- What about the C code included in the specification?
  - We can prove a Cryptol reference equivalent to the C code
  - Or we can prove properties directly on the C code

© 2015 Galois, Inc. All Rights Reserved.

C Implementation of ZUC v1.5: No Weakness

```
zucbc <- llvm_load_module "zuc15.bc";
k   <- fresh_symbolic "k"   {| [16][8] |};
iv1 <- fresh_symbolic "iv1" {| [16][8] |};
iv2 <- fresh_symbolic "iv2" {| [16][8] |};
let results = [ ("LFSR_S0", 1), ("LFSR_S1", 1), ... ];
init1 <- llvm_symexec zucbc "InitializationOne"
  [ ("k", 16),             ("iv", 16) ]
  [ ("*k", k, 16),         ("*iv", iv1, 16)
  , ("F_R1", {{ 0 }}, 1), ("F_R2", {{ 0 }}, 1)
  ] results;
init2 <- llvm_symexec zucbc "InitializationOne"
  [ ("k", 16),             ("iv", 16) ]
  [ ("*k", k, 16),         ("*iv", iv2, 16)
  , ("F_R1", {{ 0 }}, 1), ("F_R2", {{ 0 }}, 1)
  ] results;
time (prove abc {{ iv1 == iv2 || init1 != init2 }});
```

```
$ saw zuc15.saw
Time:     7.293s
Valid
```

© 2015 Galois, Inc. All Rights Reserved.

C Implementation of ZUC v1.5: No Weakness

```
zucbc <- llvm_load_module "zuc15.bc";
k   <- fresh_symbolic "k"   {| [16][8] |};
iv1 <- fresh_symbolic "iv1" {| [16][8] |};
iv2 <- fresh_symbolic "iv2" {| [16][8] |};
let results = [ ("LFSR_S0", 1), ("LFSR_S1", 1), ... ];
init1 <- llvm_symexec zucbc "InitializationOne"
  [ ("k", 16),              ("iv", 16) ]
  [ ("*k", k, 16),          ("*iv", iv1, 16)
  , ("F_R1", {{ 0 }}, 1), ("F_R2", {{ 0 }}, 1)
  ] results;
init2 <- llvm_symexec zucbc "InitializationOne"
  [ ("k", 16),              ("iv", 16) ]
  [ ("*k", k, 16),          ("*iv", iv2, 16)
  , ("F_R1", {{ 0 }}, 1), ("F_R2", {{ 0 }}, 1)
  ] results;
time (prove abc {{ iv1 == iv2 || init1 != init2 }});
```

```
$ saw zuc15.saw
Time:     7.293s
Valid
```

© 2015 Galois, Inc. All Rights Reserved.

C Implementation of ZUC v1.4: Weak IV Witness

```
zucbc <- llvm_load_module "zuc14.bc";
... // Same script otherwise
// Code to pretty-print results
```

```
$ saw zuc14.saw
Time:     6.796s
k =   [ 30, 255, 255, 255,  65, 201, 255, 255,
       255, 112, 253, 255, 255, 235, 255, 107]
iv1 = [136,   0, 143,   0, 190,   0,   0, 181,
         0,   0, 247, 251,   0, 127,  12,   0]
iv2 = [  4,   0, 143,   0, 190,   0,   0, 181,
         0,   0, 247, 251,   0, 127,  12,   0]
init1 = [2141633536, 2145545103, 2140364288,  550996414,
         1689641472, 2146514176, 2139729845, 2144172032,
          942609152, 2129380599, 2140860923, 2145265152,
         1975274879, 2146998796,  902278144, 2147483647]
init2 = [2141633536, 2145545103, 2140364288,  550996414,
         1689641472, 2146514176, 2139729845, 2144172032,
          942609152, 2129380599, 2140860923, 2145265152,
         1975274879, 2146998796,  902278144, 2147483647]
```

© 2015 Galois, Inc. All Rights Reserved.

Current Limitations

- Limited memory models for JVM/LLVM
  - Current system: fixed layout of finite size
- Symbolic termination
  - Current system: all loops must have branch conditions that eventually become constant under symbolic evaluation

© 2015 Galois, Inc. All Rights Reserved.

Current Limitations

- Limited memory models for JVM/LLVM
  - Current system: fixed layout of finite size
  - Long-term plan: explicit pointers and heap passing
- Symbolic termination
  - Current system: all loops must have branch conditions that eventually become constant under symbolic evaluation
  - Long-term plan: explicit encoding of iteration with fixpoint operators
- Both enhancements will rely on powerful provers
  - Both existing and new
  - The combination of Z3 and Lean is an appealing possibility

© 2015 Galois, Inc. All Rights Reserved.

Conclusion

- SAWScript allows a flexible combination of analysis and verification techniques
  - Semantic models of Cryptol, JVM, and LLVM programs
  - Direct connections to SAT and SMT solvers
  - Support for compositional analysis
- We've used it to analyze a variety of cryptographic implementations
  - Equivalence checking
  - Property-driven input discovery
  - Other property checking
  - Within and across multiple languages

© 2015 Galois, Inc. All Rights Reserved.

[1] GSM Association.
Specification of the 3GPP confidentiality and integrity algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC specification, June 2011.

[2] Levent Erkök and John Matthews.
High assurance programming in cryptol.
In *Fifth Cyber Security and Information Intelligence Research Workshop, CSIIRW '09, Knoxville, TN, USA, April 13-15, 2009*, page 60. ACM, 2009.

[3] Hongjun Wu, Tao Huang, PhuongHa Nguyen, Huaxiong Wang, and San Ling.
Differential attacks against stream cipher ZUC.
In *Advances in Cryptology – ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 262–277. Springer Berlin Heidelberg, 2012.

© 2015 Galois, Inc. All Rights Reserved.

|galois|