|galois|

Techniques for Scalable Symbolic Simulation

Aaron Tomb (Galois)
Sean Weaver (DoD)

HCSS | May 2013

The team, past and present:

Sally Browning, Kyle Carter, Ledah Casburn, Iavor Diatchki, Trevor Elliot, Levent Erkok, Sigbjorn Finne, Adam Foltzer, Andy Gill, Fergus Henderson, Joe Hendrix, Brian Huffman, Joe Hurd, John Launchbury, Brian Ledger, Jeff Lewis, Lee Pike, John Matthews, Thomas Nordin, Mark Shields, Joel Stanley, Frank Seaton Taylor, Jim Teisher, Aaron Tomb, Mark Tullsen, Philip Weaver, Adam Wick, Edward Yang

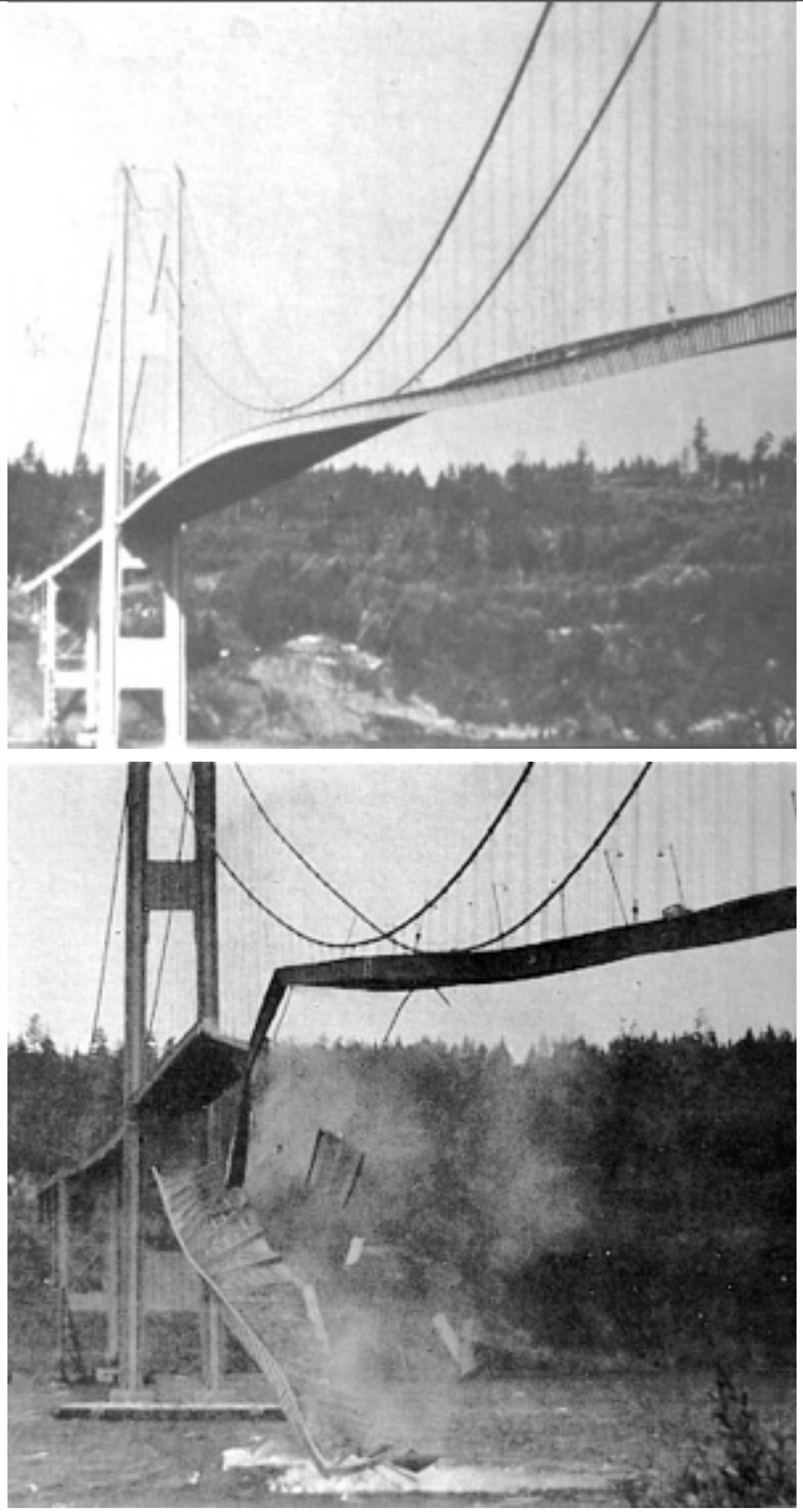
|galois|

Statistics from the testing laboratories show that 48 percent of the cryptographic modules and 27 percent of the cryptographic algorithms brought in for voluntary testing had security flaws that were corrected during testing.

Without this program, the federal government would have had only a 50-50 chance of buying correctly implemented cryptography.

NIST Computer Security Division, 2008 Annual report

Software is a digital artifact — potential for much greater confidence in the correctness of our software than in the correctness of our bridges.



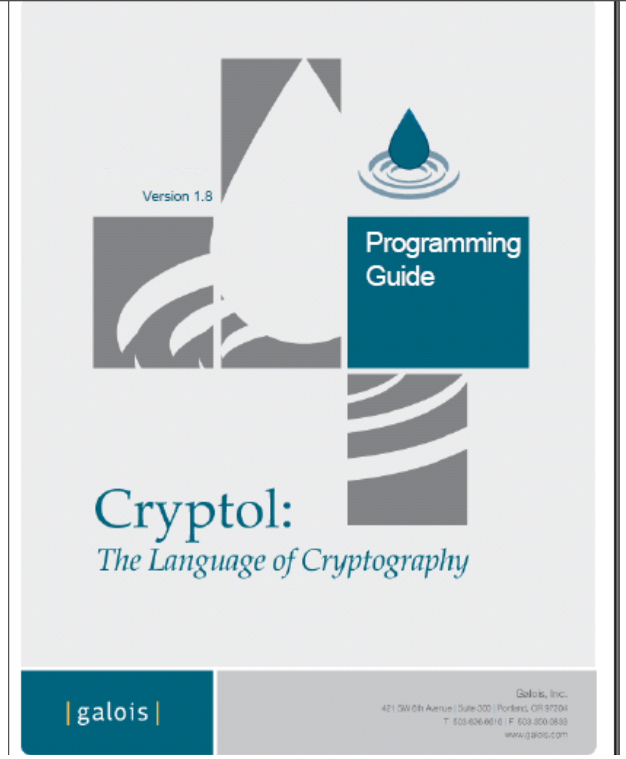
Bugs Are Prevalent

Symbolic Simulation

Galois has developed tools for showing that **different** algorithm implementations compute the **same** values for all possible keys and inputs.

Tools use formal verification techniques including symbolic simulation, rewriting, and third-party SAT and SMT-solvers.

This talk: making symbolic simulation feasible for non-trivial programs.



ABC

Yices

Example


```
int x, y;
...
if (x > y) {
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
} else {
  ;
}
```

Example

$x = X \wedge y = Y$

```
int x, y;
...
if (x > y) {
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
} else {
  ;
}
```

Example

$x = X \land y = Y$

$x = X \land y = Y \land tmp = X \land X > Y$

```
int x, y;
...
if (x > y) {
   int tmp;
   tmp = x;
   x = y;
   y = tmp;
} else {
   ;
}
```

Example

```
int x, y;
...
if (x > y) {
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
} else {
  ;
}
```

x = X ∧ y = Y

x = X ∧ y = Y ∧ tmp = X ∧ X > Y

x = Y ∧ y = Y ∧ tmp = X ∧ X > Y

Example

```
int x, y;
...
if (x > y) {
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
} else {
  ;
}
```

$x = X \wedge y = Y$

$x = X \wedge y = Y \wedge tmp = X \wedge X > Y$

$x = Y \wedge y = Y \wedge tmp = X \wedge X > Y$

$x = Y \wedge y = X \wedge tmp = X \wedge X > Y$

Example

```
int x, y;
...
if (x > y) {
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
} else {
  ;
}
```

$x = X \land y = Y$

$x = X \land y = Y \land tmp = X \land X > Y$

$x = Y \land y = Y \land tmp = X \land X > Y$

$x = Y \land y = X \land tmp = X \land X > Y$

$x = X \land y = Y \land X \leq Y$

Example


```
int x, y;
...
if (x > y) {
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
} else {
  ;
}
```

$x = X \wedge y = Y$

$x = X \wedge y = Y \wedge tmp = X \wedge X > Y$

$x = Y \wedge y = Y \wedge tmp = X \wedge X > Y$

$x = Y \wedge y = X \wedge tmp = X \wedge X > Y$

$x = X \wedge y = Y \wedge X \leq Y$

$(x = Y \wedge y = X \wedge X > Y) \vee$
$(x = X \wedge y = Y \wedge X \leq Y)$

Handling Errors

- Some operations have undefined behavior
  - Memory access, division, etc.
- Symbolic result has two parts
  - Value: in undefined cases, takes default value
  - Error flag: satisfiable for undefined cases

Structural Hashing of Terms

- Symbolic simulation yields repetitive terms
  - Sharing repeated sub-terms is critical
  - Want a DAG instead of a tree
- Can do this at any level of abstraction
  - AIGs at the bit level (low-level but can be very compact)
  - Our own term data structure at the word level

A Case Study: SHA Message Digest

SHA-384 is one variation of the standard SHA-2 message digest algorithm, part of Suite B.

Widely used for integrity verification, and part of the FIPS 180-2 standard.

A challenging target for verification, due to extensive bit-level operations, and the need to process arbitrarily long messages.

OpenSSL
Cryptography and SSL/TLS Toolkit

Structure of SHA-384

- Iterative application of a block digest function
  - Results of previous iterations feed into current
- Block function involves many applications of a few primitives
  - Bitwise and, xor, inversion
  - Word rotation, addition
- Bit-precise reasoning is critical

Implementations

- We will work with two implementations
  - Reference specification (Cryptol, 178 lines)
  - Bouncy Castle (Java, 591 lines)
- And two levels of models
  - Bit-level And-Inverter Graphs, with SAT solvers
  - Word-level terms, with SMT solvers

Path Merging

Goal: more efficient symbolic simulation

Program-level Merging

```
1: int ffs(int i) {
2:   byte n = 0;
3:   if ((i & 0xffff) == 0) {
4:     n += 16; i >>= 16;
5:   }
6:   if ((i & 0x00ff) == 0) {
7:     n += 8;  i >>= 8;
8:   }
9:   if ((i & 0x000f) == 0) {
10:     n += 4;  i >>= 4;
11:   }
12:   if ((i & 0x0003) == 0) {
13:     n += 2;  i >>= 2;
14:   }
15:   if (i != 0) {
16:     return (n+((i+1) & 0x01));
17:   }
18:   return 0;
19: }
20: ffs(x);
21: ffs(y);
```

- Approach taken by simple symbolic simulators: only merge at end of program, if at all
- Merge 1024 independent states at line 21
- Total of 1023 merge operations

Method-level Merging

```
1: int ffs(int i) {
2:   byte n = 0;
3:   if ((i & 0xffff) == 0) {
4:     n += 16; i >>= 16;
5:   }
6:   if ((i & 0x00ff) == 0) {
7:     n += 8;  i >>= 8;
8:   }
9:   if ((i & 0x000f) == 0) {
10:    n += 4;  i >>= 4;
11:  }
12:  if ((i & 0x0003) == 0) {
13:    n += 2;  i >>= 2;
14:  }
15:  if (i != 0) {
16:    return (n+((i+1) & 0x01));
17:  }
18:  return 0;
19: }
20: ffs(x);
21: ffs(y);
```

- Our first approach: merge before returning from a method
- Merge 32 independent states at lines 20 and 21
- Total of 62 merge operations

Post-dominator Merging

```
1: int ffs(int i) {
2:   byte n = 0;
3:   if ((i & 0xffff) == 0) {
4:     n += 16; i >>= 16;
5:   }
6:   if ((i & 0x00ff) == 0) {
7:     n += 8;  i >>= 8;
8:   }
9:   if ((i & 0x000f) == 0) {
10:    n += 4;  i >>= 4;
11:  }
12:  if ((i & 0x0003) == 0) {
13:    n += 2;  i >>= 2;
14:  }
15:  if (i != 0) {
16:    return (n+((i+1) & 0x01));
17:  }
18:  return 0;
19: }
20: ffs(x);
21: ffs(y);
```

- Our current approach: similar to join points in dataflow analysis, abstract interpretation
- Merge at every point in CFG that post-dominates more than one other point
- Merge 2 independent states at lines 5, 8, 11, 14, 19, twice each
- Total of 10 merge operations

Symbolic Instruction Set


```
                          %entry.0  setCurrentBlock %0.0
0: Iload 1                %0.0      0: Iload 1
1: Istore 3                         1: Istore 3
2: Iload 2                          2: Iload 2
3: Istore 4                         3: Istore 4
5: Iload 3                          5: Iload 3
6: Iload 4                          6: Iload 4
8: If_icmple 21                     pushPending %0.1 [S[0] <= S[1]] [merge at %11.0]
                                    [setCurrentBlock %11.0]
                          %0.1      setCurrentBlock %21.0
11: Iload 3               %11.0     11: Iload 3
12: Istore 5                        12: Istore 5
14: Iload 4                         14: Iload 4
16: Istore 3                        16: Istore 3
17: Iload 5                         17: Iload 5
19: Istore 4                        19: Istore 4
                                    setCurrentBlock %21.0
21: Iload 3               %21.0     21: Iload 3
22: Ireturn                         returnVal
```

Path Merging Comparison

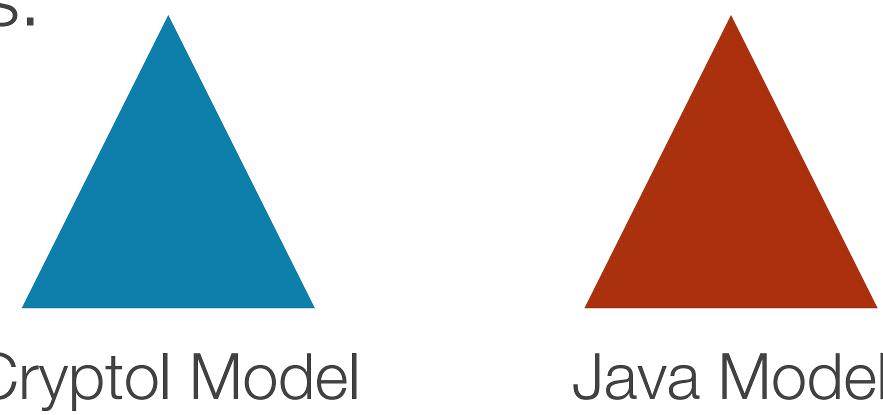
Approach	FFS AIG Nodes	FFS Time	SHA384 AIG Nodes	SHA384 Time
Program-level	24018	2.75s		
Method-level	2311	0.41s	555942	10.2s
Post-dominator	1022	0.18s	555942	4.8s

Compositional Reasoning

Goal: more efficient symbolic simulation and proof

Inlining Equivalent Subterms

1. Use forward symbolic simulation to unroll implementations, and generate terms that precisely describe results.

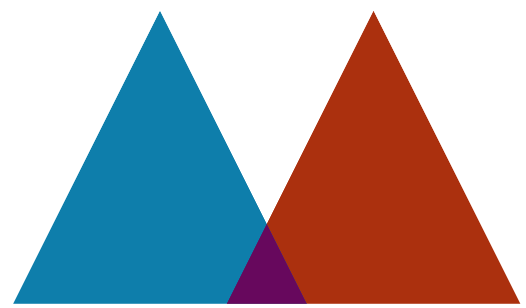


2. Show equivalence of two complete terms through rewriting, and off-the-shelf theorem provers, including abc or Yices.

ABC Yices Rewriting Cryptol

Inlining Equivalent Subterms

1. Use forward symbolic simulation to unroll implementations, and generate terms that precisely describe results.



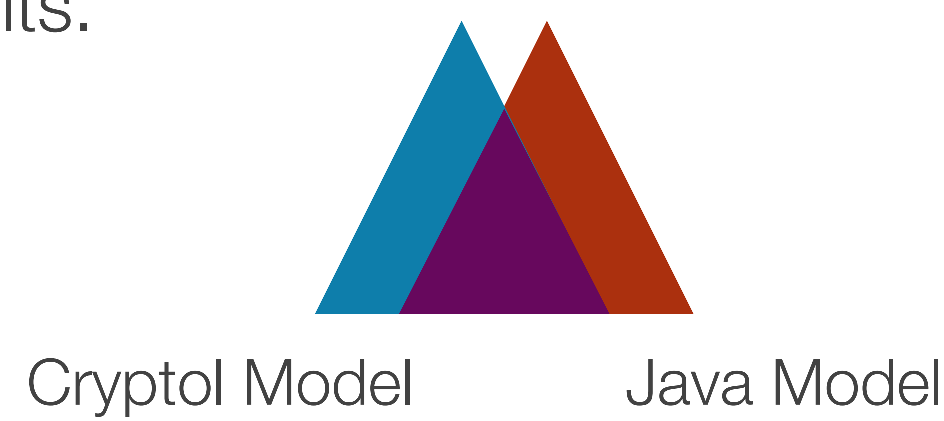
Cryptol Model Java Model

2. Show equivalence of two complete terms through rewriting, and off-the-shelf theorem provers, including abc or Yices.

ABC Yices Rewriting Cryptol

Inlining Equivalent Subterms

1. Use forward symbolic simulation to unroll implementations, and generate terms that precisely describe results.

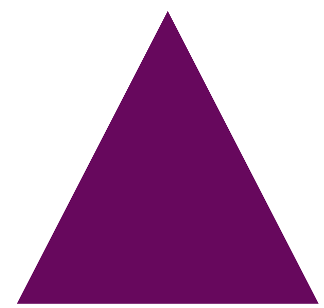


2. Show equivalence of two complete terms through rewriting, and off-the-shelf theorem provers, including abc or Yices.

ABC Yices Rewriting Cryptol

Inlining Equivalent Subterms

1. Use forward symbolic simulation to unroll implementations, and generate terms that precisely describe results.



Cryptol Model Java Model

2. Show equivalence of two complete terms through rewriting, and off-the-shelf theorem provers, including abc or Yices.

ABC Yices Rewriting Cryptol

Abstracting Equivalent Terms

1. Use forward symbolic simulation to unroll implementations, and generate terms that abstractly describe results.

Cryptol Model   Java Model

2. Show equivalence of two terms with uninterpreted functions through rewriting, and off-the-shelf theorem provers, including abc or Yices.

ABC Yices Rewriting Cryptol

Abstracting Equivalent Terms

1. Use forward symbolic simulation to unroll implementations, and generate terms that abstractly describe results.

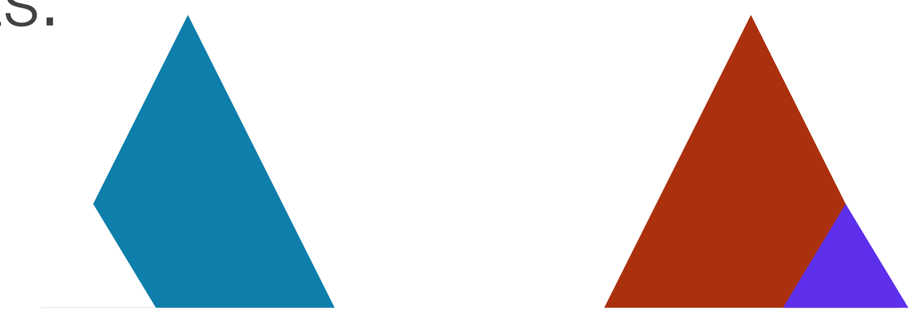
Cryptol Model Java Model

2. Show equivalence of two terms with uninterpreted functions through rewriting, and off-the-shelf theorem provers, including abc or Yices.

ABC Yices Rewriting Cryptol

Abstracting Equivalent Terms

1. Use forward symbolic simulation to unroll implementations, and generate terms that abstractly describe results.



Cryptol Model   Java Model

2. Show equivalence of two terms with uninterpreted functions through rewriting, and off-the-shelf theorem provers, including abc or Yices.

ABC   Yices   Rewriting   Cryptol

Abstracting Equivalent Terms

1. Use forward symbolic simulation to unroll implementations, and generate terms that abstractly describe results.

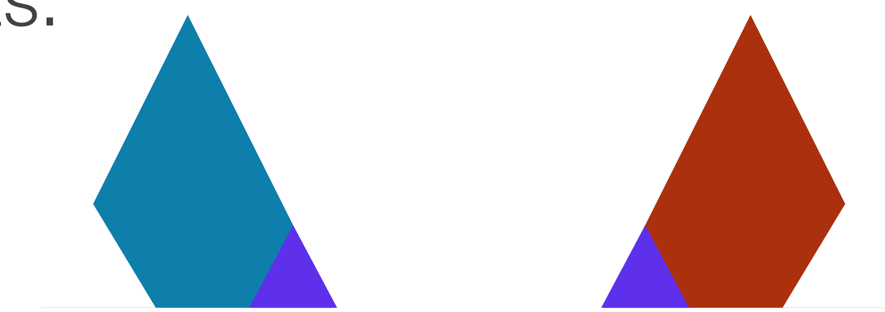
Cryptol Model   Java Model

2. Show equivalence of two terms with uninterpreted functions through rewriting, and off-the-shelf theorem provers, including abc or Yices.

ABC   Yices   Rewriting   Cryptol

Abstracting Equivalent Terms

1. Use forward symbolic simulation to unroll implementations, and generate terms that abstractly describe results.



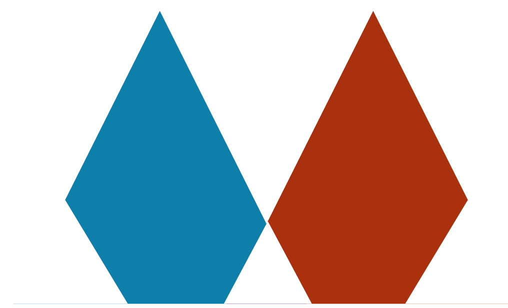
Cryptol Model   Java Model

2. Show equivalence of two terms with uninterpreted functions through rewriting, and off-the-shelf theorem provers, including abc or Yices.

ABC   Yices   Rewriting   Cryptol

Abstracting Equivalent Terms

1. Use forward symbolic simulation to unroll implementations, and generate terms that abstractly describe results.



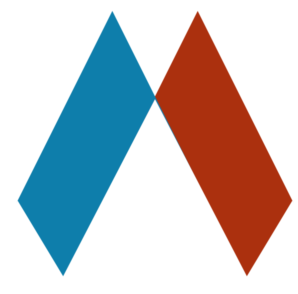
Cryptol Model   Java Model

2. Show equivalence of two terms with uninterpreted functions through rewriting, and off-the-shelf theorem provers, including abc or Yices.

ABC   Yices   Rewriting   Cryptol

Abstracting Equivalent Terms

1. Use forward symbolic simulation to unroll implementations, and generate terms that abstractly describe results.



Cryptol Model Java Model

2. Show equivalence of two terms with uninterpreted functions through rewriting, and off-the-shelf theorem provers, including abc or Yices.

ABC Yices Rewriting Cryptol

Abstracting Equivalent Terms

1. Use forward symbolic simulation to unroll implementations, and generate terms that abstractly describe results.

Cryptol Model　　Java Model

2. Show equivalence of two terms with uninterpreted functions through rewriting, and off-the-shelf theorem provers, including abc or Yices.

ABC　Yices　Rewriting　Cryptol

SHA-384 Block Loop Iteration Decomposition Helpful

Approach	Eq Model Nodes	Decomposition Steps Required	Verification Time
No Composition (Bit)	13,008	None needed Automatic (ABC)	1.13s
Inlined (Bit)		Ten manual steps Proved using ABC	0.27s
No Composition (Word)	41,316	None needed Automatic (Yices)	2.01s
Inlined (Word)	64,254	Ten manual steps Proved using Yices	2.43s
Abstracted (Word)	10,579	Ten manual steps Proved using Yices	0.26s

Full SHA-384 Block Decomposition Necessary

Approach	Eq Model Nodes	Decomposition Steps Required	Verification Time
No Composition (Bit)	1,212,993	None needed Automatic (ABC)	>30m
Inlined (Bit)		Ten manual steps Proved using ABC	26.5s
No Composition (Word)	5,298,656	None needed Automatic (Yices)	>30m
Inlined (Word)	9,986,526	Ten manual steps Proved using Yices	>30m
Abstracted (Word)	336,172	Ten manual steps Proved using Yices	>30m

SAWScript:
Language for Compositional Verification

Goal: convenient and flexible access to simulator capabilities

SAWScript 2.0 Currently under development

SAWScript Goals

- Allow flexible coordination of software analysis
  - Somewhat like interactive theorem provers, but tailored to software verification
- Strong emphasis on compositional reasoning
- Enable the application of a wide variety of proof tools to programs written in numerous languages

SAWScript Capabilities

- Allows precise reasoning about behavior of both imperative and functional programs, including recursion, side effects
- Method specifications are used in two ways:
  - As statements to be proven
  - As lemmas to help verify later methods
- SAWScript has a simple tactic language for user control over verification steps

Method Specification Requirements

- Consistent types for target program variables, including lengths for arrays
- Assumptions on inputs
- Which imperative references can alias other references
- Expected results when function or method terminates
- Optionally, postconditions at intermediate breakpoints within functions/methods
- Tactics for performing verification on resulting term

Example:
Ch Verification


```
ref_Ch : ([64], [64], [64]) -> [64];
ref_Ch <- extractCryptol “SHA384.cry” “Ch”;

ch_result <- verifyJava “org.bouncycastle.crypto.digests.SHA384Digest.Ch” (do {
  x <- var “x” long;
  y <- var “y” long;
  z <- var “z” long;
  return ref_Ch(x, y, z);
  verify abc;
})

java_Ch : ([16][64], [8][64]) -> [8][64];
java_Ch <- extractJava “org.bouncycastle.crypto.digests.SHA384Digest.Ch” pure;

verify (do {
  goal (\(a, b) -> java_Ch (a, b) == ref_Ch (a, b));
  abc;
})
```

Example: Ch Verification

Multiple languages

```
ref_Ch : ([64], [64], [64]) -> [64];
ref_Ch <- extractCryptol "SHA384.cry" "Ch";

ch_result <- verifyJava "org.bouncycastle.crypto.digests.SHA384Digest.Ch" (do {
  x <- var "x" long;
  y <- var "y" long;
  z <- var "z" long;
  return ref_Ch(x, y, z);
  verify abc;
})

java_Ch : ([16][64], [8][64]) -> [8][64];
java_Ch <- extractJava "org.bouncycastle.crypto.digests.SHA384Digest.Ch" pure;

verify (do {
  goal (\(a, b) -> java_Ch (a, b) == ref_Ch (a, b));
  abc;
})
```

Example: Ch Verification

Multiple languages

Pre-conditions

```
ref_Ch : ([64], [64], [64]) -> [64];
ref_Ch <- extractCryptol “SHA384.cry” “Ch”;

ch_result <- verifyJava “org.bouncycastle.crypto.digests.SHA384Digest.Ch” (do {
  x <- var “x” long;
  y <- var “y” long;
  z <- var “z” long;
  return ref_Ch(x, y, z);
  verify abc;
})

java_Ch : ([16][64], [8][64]) -> [8][64];
java_Ch <- extractJava “org.bouncycastle.crypto.digests.SHA384Digest.Ch” pure;

verify (do {
  goal (\(a, b) -> java_Ch (a, b) == ref_Ch (a, b));
  abc;
})
```

Example: Ch Verification

Multiple languages

```
ref_Ch : ([64], [64], [64]) -> [64];
ref_Ch <- extractCryptol “SHA384.cry” “Ch”;

ch_result <- verifyJava “org.bouncycastle.crypto.digests.SHA384Digest.Ch” (do {
  x <- var “x” long;
  y <- var “y” long;
  z <- var “z” long;
  return ref_Ch(x, y, z);
  verify abc;
})

java_Ch : ([16][64], [8][64]) -> [8][64];
java_Ch <- extractJava “org.bouncycastle.crypto.digests.SHA384Digest.Ch” pure;

verify (do {
  goal (\(a, b) -> java_Ch (a, b) == ref_Ch (a, b));
  abc;
})
```

Pre-conditions

Simplifying defaults

Example: Ch Verification

Multiple languages

```
ref_Ch : ([64], [64], [64]) -> [64];
ref_Ch <- extractCryptol “SHA384.cry” “Ch”;

ch_result <- verifyJava “org.bouncycastle.crypto.digests.SHA384Digest.Ch” (do {
  x <- var “x” long;
  y <- var “y” long;
  z <- var “z” long;
  return ref_Ch(x, y, z);
  verify abc;
})

java_Ch : ([16][64], [8][64]) -> [8][64];
java_Ch <- extractJava “org.bouncycastle.crypto.digests.SHA384Digest.Ch” pure;

verify (do {
  goal (\(a, b) -> java_Ch (a, b) == ref_Ch (a, b));
  abc;
})
```

Pre-conditions

Simplifying defaults

Proof tactics

Example: `processBlock` Verification


```
ref_Block : ([8][64], [16][64]) -> [8][64];
ref_Block <- extractCryptol “SHA384.cry” “block512”;

blockMeth = “org.bouncycastle.crypto.digests.SHA384Digest.processBlock”;

block_spec <- verifyJava blockMeth (do {
  this <- var “this” (class “org.bouncycastle.crypto.digests.SHA384Digest”);
  H1 <- field this “H1” long;
  ...
  H8 <- field this “H8” long;
  W <- field this “W” (array 80 long);
  override_uninterpreted [ ch_result, maj_result,
                           usig0_result, lsig0_result,
                           usig1_result, lsig1_result,
                         ];
  let H = [H1 H2 H3 H4 H5 H8 H7 H8];
  let H’ = ref_Block(H, W);
  updateField this “H1” (H’ @ 0);
  ...
  updateField this “H8” (H’ @ 7);
  verify yices;
})
```

Example:
processBlock Verification

```
ref_Block : ([8][64], [16][64]) -> [8][64];
ref_Block <- extractCryptol “SHA384.cry” “block512”;

blockMeth = “org.bouncycastle.crypto.digests.SHA384Digest.processBlock”;

block_spec <- verifyJava blockMeth (do {
  this <- var “this” (class “org.bouncycastle.crypto.digests.SHA384Digest”);
  H1 <- field this “H1” long;
  ...
  H8 <- field this “H8” long;
  W <- field this “W” (array 80 long);
  override_uninterpreted [ ch_result, maj_result,
                           usig0_result, lsig0_result,
                           usig1_result, lsig1_result,
                         ];
  let H = [H1 H2 H3 H4 H5 H8 H7 H8];
  let H’ = ref_Block(H, W);
  updateField this “H1” (H’ @ 0);
  ...
  updateField this “H8” (H’ @ 7);
  verify yices;
})
```

Composition

Example: `processBlock` Verification

```
ref_Block : ([8][64], [16][64]) -> [8][64];
ref_Block <- extractCryptol “SHA384.cry” “block512”;

blockMeth = “org.bouncycastle.crypto.digests.SHA384Digest.processBlock”;

block_spec <- verifyJava blockMeth (do {
  this <- var “this” (class “org.bouncycastle.crypto.digests.SHA384Digest”);
  H1 <- field this “H1” long;
  ...
  H8 <- field this “H8” long;
  W <- field this “W” (array 80 long);
  override_uninterpreted [ ch_result, maj_result,
                           usig0_result, lsig0_result,
                           usig1_result, lsig1_result,
                         ];
  let H = [H1 H2 H3 H4 H5 H8 H7 H8];
  let H’ = ref_Block(H, W);
  updateField this “H1” (H’ @ 0);
  ...
  updateField this “H8” (H’ @ 7);
  verify yices;
})
```

Composition

Post-conditions

Summary

- Symbolic simulation made practical:
  - Represent states efficiently but precisely
  - Merge paths whenever possible
  - Abstract over calls
- With these techniques, equivalence checking scales to programs of thousands of lines, likely larger

Thanks!