Axe: An Automated Formal Equivalence Checking Tool for Programs

Eric W. Smith

Advised by David Dill
Computer Science Department
Stanford University

Motivation

- Cryptography is used in many important applications:
  - Privacy, E-commerce, National Security, Military, Voting
- Relies on algorithmic "building blocks"
  - Block ciphers, stream ciphers, hash functions
  - Errors could compromise the security of larger systems.
- Is worth getting right.
  - No accidental or deliberate errors.

Crypto. Validation in Practice

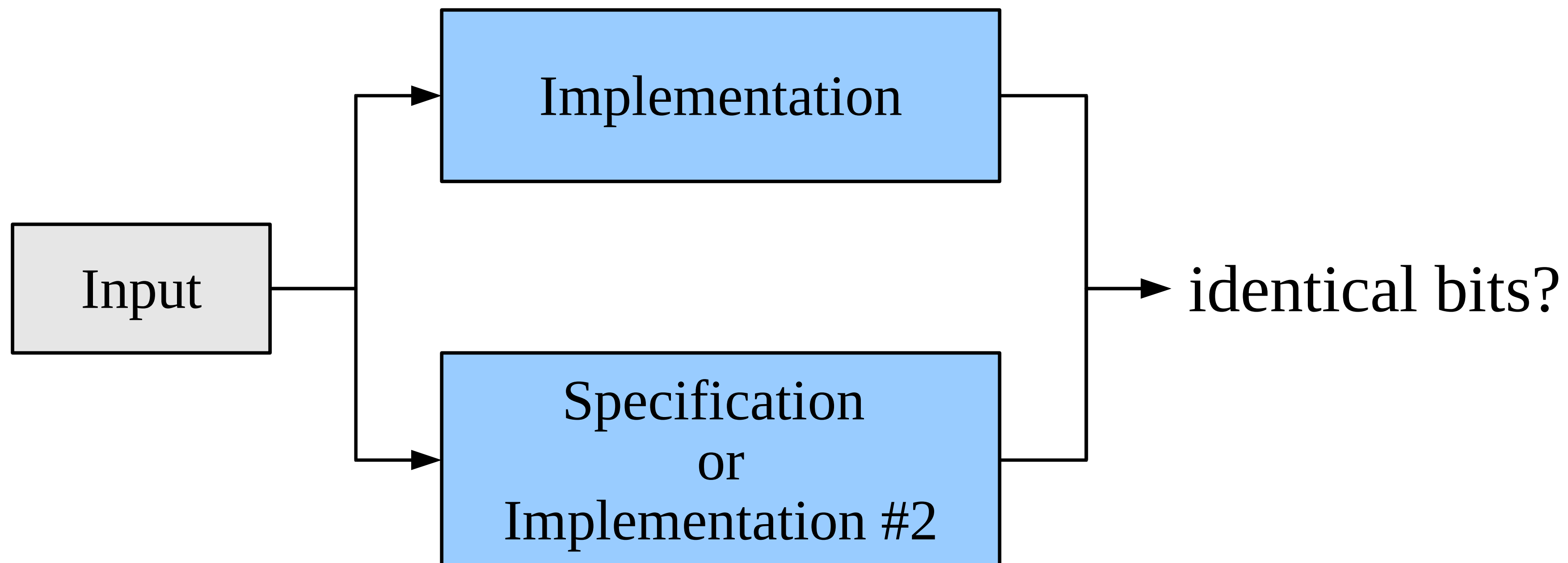
- Testing:
  - NIST certifies AES implementations.
  - Run lots of tests.
  - They admit that certification is not a proof.
- Too many inputs to test them all
  - at least 2^{256} for AES
- We would like a *proof*.

Formal Proofs of Programs

- Formals verification has been a grand challenge of computer science for many years (Turing, Dijkstra, Floyd, Hoare, Pnueli, Manna, Clarke, Emerson, Boyer, Moore).
  - many properties are undecidable
- Not going to solve it in this talk!
- We restrict the domain to crypto.
  - Show how to automate the proofs.

What Axe Proves

- Other techniques can prove generic properties of huge programs (memory issues, threading).
- We want *full functional correctness:*



Results

- Prior to now, very few real crypto. implementations verified.
- We verified AES, DES, 3DES, Blowfish, RC2, RC4, RC6, Skipjack, MD5, SHA-1
  - real Java code
  - code used by many people
  - code written by others (essentially unmodified)
  - non-prohibitive cost to verify

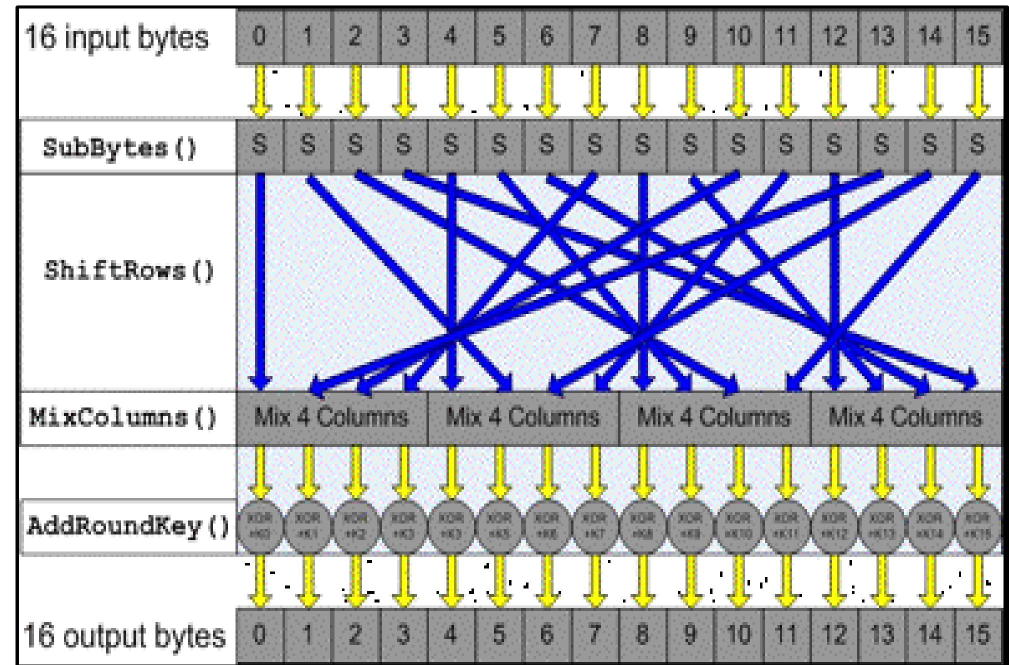
Challenges of Crypto. Verification

- Algorithms designed to be intractable (impossible to invert).
- Lots of mixing (leads to huge terms).
- Optimized implementations.

Block ciphers

- Encrypt and decrypt.
- Shared, secret key.
- Very widely used.
- Fixed size inputs.
  - too many to test!

One round of AES



- Fixed number of loop iterations
  - can "unroll" loops

Excerpt of AES Encryption

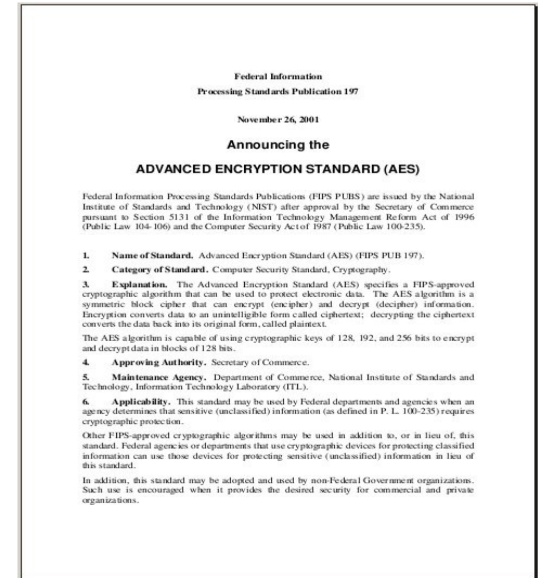
```
private void encryptBlock(int[][] KW) {
  int r, r0, r1, r2, r3;
  C0 ^= KW[0][0];
  C1 ^= KW[0][1];
  C2 ^= KW[0][2];
  C3 ^= KW[0][3];

  for (r = 1; r < ROUNDS - 1;)
  {
     r0 = mcol((S[C0&255]&255) ^ ((S[(C1>>8)&255]&255)<<8) ^ ((S[(C2>>16)&255]&255)<<16) ^ (S[(C3>>24)&255]<<24)) ^ KW[r][0];
     r1 = mcol((S[C1&255]&255) ^ ((S[(C2>>8)&255]&255)<<8) ^ ((S[(C3>>16)&255]&255)<<16) ^ (S[(C0>>24)&255]<<24)) ^ KW[r][1];
     r2 = mcol((S[C2&255]&255) ^ ((S[(C3>>8)&255]&255)<<8) ^ ((S[(C0>>16)&255]&255)<<16) ^ (S[(C1>>24)&255]<<24)) ^ KW[r][2];
     r3 = mcol((S[C3&255]&255) ^ ((S[(C0>>8)&255]&255)<<8) ^ ((S[(C1>>16)&255]&255)<<16) ^ (S[(C2>>24)&255]<<24)) ^ KW[r++][3];
     C0 = mcol((S[r0&255]&255) ^ ((S[(r1>>8)&255]&255)<<8) ^ ((S[(r2>>16)&255]&255)<<16) ^ (S[(r3>>24)&255]<<24)) ^ KW[r][0];
     C1 = mcol((S[r1&255]&255) ^ ((S[(r2>>8)&255]&255)<<8) ^ ((S[(r3>>16)&255]&255)<<16) ^ (S[(r0>>24)&255]<<24)) ^ KW[r][1];
     C2 = mcol((S[r2&255]&255) ^ ((S[(r3>>8)&255]&255)<<8) ^ ((S[(r0>>16)&255]&255)<<16) ^ (S[(r1>>24)&255]<<24)) ^ KW[r][2];
     C3 = mcol((S[r3&255]&255) ^ ((S[(r0>>8)&255]&255)<<8) ^ ((S[(r1>>16)&255]&255)<<16) ^ (S[(r2>>24)&255]<<24)) ^ KW[r++][3];
  }
  r0 = mcol((S[C0&255]&255) ^ ((S[(C1>>8)&255]&255)<<8) ^ ((S[(C2>>16)&255]&255)<<16) ^ (S[(C3>>24)&255]<<24)) ^ KW[r][0];
  r1 = mcol((S[C1&255]&255) ^ ((S[(C2>>8)&255]&255)<<8) ^ ((S[(C3>>16)&255]&255)<<16) ^ (S[(C0>>24)&255]<<24)) ^ KW[r][1];
  r2 = mcol((S[C2&255]&255) ^ ((S[(C3>>8)&255]&255)<<8) ^ ((S[(C0>>16)&255]&255)<<16) ^ (S[(C1>>24)&255]<<24)) ^ KW[r][2];
  r3 = mcol((S[C3&255]&255) ^ ((S[(C0>>8)&255]&255)<<8) ^ ((S[(C1>>16)&255]&255)<<16) ^ (S[(C2>>24)&255]<<24)) ^ KW[r++][3];

  // the final round is a simple function of S

  C0 = (S[r0&255]&255) ^ ((S[(r1>>8)&255]&255)<<8) ^ ((S[(r2>>16)&255]&255)<<16) ^ (S[(r3>>24)&255]<<24) ^ KW[r][0];
  C1 = (S[r1&255]&255) ^ ((S[(r2>>8)&255]&255)<<8) ^ ((S[(r3>>16)&255]&255)<<16) ^ (S[(r0>>24)&255]<<24) ^ KW[r][1];
  C2 = (S[r2&255]&255) ^ ((S[(r3>>8)&255]&255)<<8) ^ ((S[(r0>>16)&255]&255)<<16) ^ (S[(r1>>24)&255]<<24) ^ KW[r][2];
  C3 = (S[r3&255]&255) ^ ((S[(r0>>8)&255]&255)<<8) ^ ((S[(r1>>16)&255]&255)<<16) ^ (S[(r2>>24)&255]<<24) ^ KW[r][3];
  }
```

Lookup Table for Optimized AES

```
private static final int[] T0 =
{
 0xa56363c6, 0x847c7cf8, 0x997777ee, 0x8d7b7bf6, 0x0df2f2ff, 0xbd6b6bd6, 0xb16f6fde, 0x54c5c591, 0x50303060, 0x03010102,
 0xa96767ce, 0x7d2b2b56, 0x19fefee7, 0x62d7d7b5, 0xe6abab4d, 0x9a7676ec, 0x45caca8f, 0x9d82821f, 0x40c9c989, 0x877d7dfa,
 0x15fafaef, 0xeb5959b2, 0xc947478e, 0x0bf0f0fb, 0xecadad41, 0x67d4d4b3, 0xfda2a25f, 0xeaafaf45, 0xbf9c9c23, 0xf7a4a453,
 0x967272e4, 0x5bc0c09b, 0xc2b7b775, 0x1cfdfde1, 0xae93933d, 0x6a26264c, 0x5a36366c, 0x413f3f7e, 0x02f7f7f5, 0x4fcccc83,
 0x5c343468, 0xf4a5a551, 0x34e5e5d1, 0x08f1f1f9, 0x937171e2, 0x73d8d8ab, 0x53313162, 0x3f15152a, 0x0c040408, 0x52c7c795,
 0x65232346, 0x5ec3c39d, 0x28181830, 0xa1969637, 0x0f05050a, 0xb59a9a2f, 0x0907070e, 0x36121224, 0x9b80801b, 0x3de2e2df,
 0x26ebebcd, 0x6927274e, 0xcdb2b27f, 0x9f7575ea, 0x1b090912, 0x9e83831d, 0x742c2c58, 0x2e1a1a34, 0x2d1b1b36, 0xb26e6edc,
 0xee5a5ab4, 0xfba0a05b, 0xf65252a4, 0x4d3b3b76, 0x61d6d6b7, 0xceb3b37d, 0x7b292952, 0x3ee3e3dd, 0x712f2f5e, 0x97848413,
 0xf55353a6, 0x68d1d1b9, 0x00000000, 0x2cededc1, 0x60202040, 0x1ffcfce3, 0xc8b1b179, 0xed5b5bb6, 0xbe6a6ad4, 0x46cbcb8d,
 0xd9bebe67, 0x4b393972, 0xde4a4a94, 0xd44c4c98, 0xe85858b0, 0x4acfcf85, 0x6bd0d0bb, 0x2aefefc5, 0xe5aaaa4f, 0x16fbfbed,
 0xc5434386, 0xd74d4d9a, 0x55333366, 0x94858511, 0xcf45458a, 0x10f9f9e9, 0x06020204, 0x817f7ffe, 0xf05050a0, 0x443c3c78,
 0xba9f9f25, 0xe3a8a84b, 0xf35151a2, 0xfea3a35d, 0xc0404080, 0x8a8f8f05, 0xad92923f, 0xbc9d9d21, 0x48383870, 0x04f5f5f1,
 0xdfbcbc63, 0xc1b6b677, 0x75dadaaf, 0x63212142, 0x30101020, 0x1affffe5, 0x0ef3f3fd, 0x6dd2d2bf, 0x4ccdcd81, 0x140c0c18,
 0x35131326, 0x2fececc3, 0xe15f5fbe, 0xa2979735, 0xcc444488, 0x3917172e, 0x57c4c493, 0xf2a7a755, 0x827e7efc, 0x473d3d7a,
 0xac6464c8, 0xe75d5dba, 0x2b191932, 0x957373e6, 0xa06060c0, 0x98818119, 0xd14f4f9e, 0x7fdcdca3, 0x66222244, 0x7e2a2a54,
 0xab90903b, 0x8388880b, 0xca46468c, 0x29eeeec7, 0xd3b8b86b, 0x3c141428, 0x79dedea7, 0xe25e5ebc, 0x1d0b0b16, 0x76dbdbad,
 0x3be0e0db, 0x56323264, 0x4e3a3a74, 0x1e0a0a14, 0xdb494992, 0x0a06060c, 0x6c242448, 0xe45c5cb8, 0x5dc2c29f, 0x6ed3d3bd,
 0xefacac43, 0xa66262c4, 0xa8919139, 0xa4959531, 0x37e4e4d3, 0x8b7979f2, 0x32e7e7d5, 0x43c8c88b, 0x5937376e, 0xb76d6dda,
 0x8c8d8d01, 0x64d5d5b1, 0xd24e4e9c, 0xe0a9a949, 0xb46c6cd8, 0xfa5656ac, 0x07f4f4f3, 0x25eaeacf, 0xaf6565ca, 0x8e7a7af4,
 0xe9aeae47, 0x18080810, 0xd5baba6f, 0x887878f0, 0x6f25254a, 0x722e2e5c, 0x241c1c38, 0xf1a6a657, 0xc7b4b473, 0x51c6c697,
 0x23e8e8cb, 0x7cdddda1, 0x9c7474e8, 0x211f1f3e, 0xdd4b4b96, 0xdcbdbd61, 0x868b8b0d, 0x858a8a0f, 0x907070e0, 0x423e3e7c,
 0xc4b5b571, 0xaa6666cc, 0xd8484890, 0x05030306, 0x01f6f6f7, 0x120e0e1c, 0xa36161c2, 0x5f35356a, 0xf95757ae, 0xd0b9b969,
 0x91868617, 0x58c1c199, 0x271d1d3a, 0xb99e9e27, 0x38e1e1d9, 0x13f8f8eb, 0xb398982b, 0x33111122, 0xbb6969d2, 0x70d9d9a9,
 0x898e8e07, 0xa7949433, 0xb69b9b2d, 0x221e1e3c, 0x92878715, 0x20e9e9c9, 0x49cece87, 0xff5555aa, 0x78282850, 0x7adfdfa5,
 0x8f8c8c03, 0xf8a1a159, 0x80898909, 0x170d0d1a, 0xdabfbf65, 0x31e6e6d7, 0xc6424284, 0xb86868d0, 0xc3414182, 0xb0999929,
 0x772d2d5a, 0x110f0f1e, 0xcbb0b07b, 0xfc5454a8, 0xd6bbbb6d, 0x3a16162c};
```

Good News

- Most software poorly specified...
- But crypto. specifications are precise.
  - FIPS-197 describes AES.
  - Gives the exact value of each output bit.



Federal Information

Processing Standards Publication 197

November 26, 2001

Announcing the

ADVANCED ENCRYPTION STANDARD (AES)

Federal Information Processing Standards Publications (FIPS PUBS) are issued by the National Institute of Standards and Technology (NIST) after approval by the Secretary of Commerce pursuant to Section 5131 of the Information Technology Management Reform Act of 1996 (Public Law 104-106) and the Computer Security Act of 1987 (Public Law 100-235).

1. **Name of Standard.** Advanced Encryption Standard (AES) (FIPS PUB 197).

2. **Category of Standard.** Computer Security Standard, Cryptography.

3. **Explanation.** The Advanced Encryption Standard (AES) specifies a FIPS-approved cryptographic algorithm that can be used to protect electronic data. The AES algorithm is a symmetric block cipher that can encrypt (encipher) and decrypt (decipher) information. Encryption converts data to an unintelligible form called ciphertext; decrypting the ciphertext converts the data back into its original form, called plaintext.

The AES algorithm is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits.

4. **Approving Authority.** Secretary of Commerce.

5. **Maintenance Agency.** Department of Commerce, National Institute of Standards and Technology, Information Technology Laboratory (ITL).

6. **Applicability.** This standard may be used by Federal departments and agencies when an agency determines that sensitive (unclassified) information (as defined in P. L. 100-235) requires cryptographic protection.

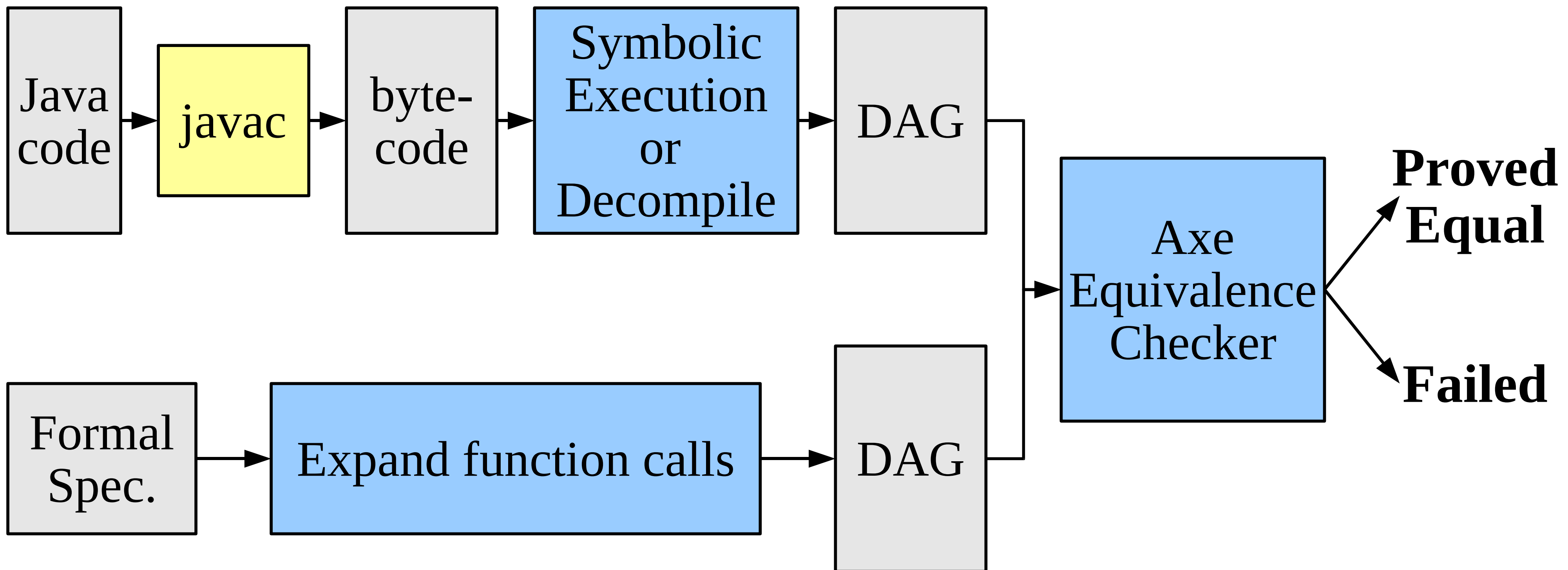
Other FIPS-approved cryptographic algorithms may be used in addition to, or in lieu of, this standard. Federal agencies or departments that use cryptographic devices for protecting classified information can use those devices for protecting sensitive (unclassified) information in lieu of this standard.

In addition, this standard may be adopted and used by non-Federal Government organizations. Such use is encouraged when it provides the desired security for commercial and private organizations.

Formalizing the Specifications

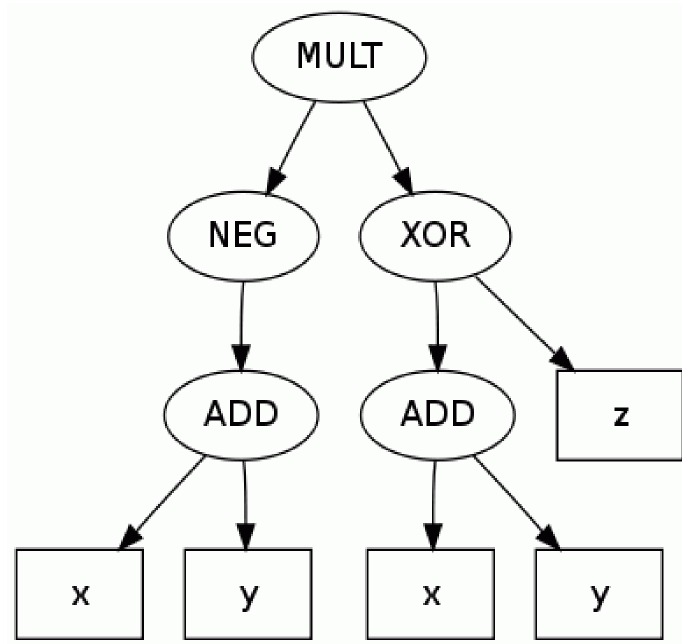
- We formalized the English descriptions:
  - in a precise, mathematical language (ACL2)
  - closely match the official descriptions (unoptimized)
- Validated by running them on test cases.
- Reusable.

Using Axe

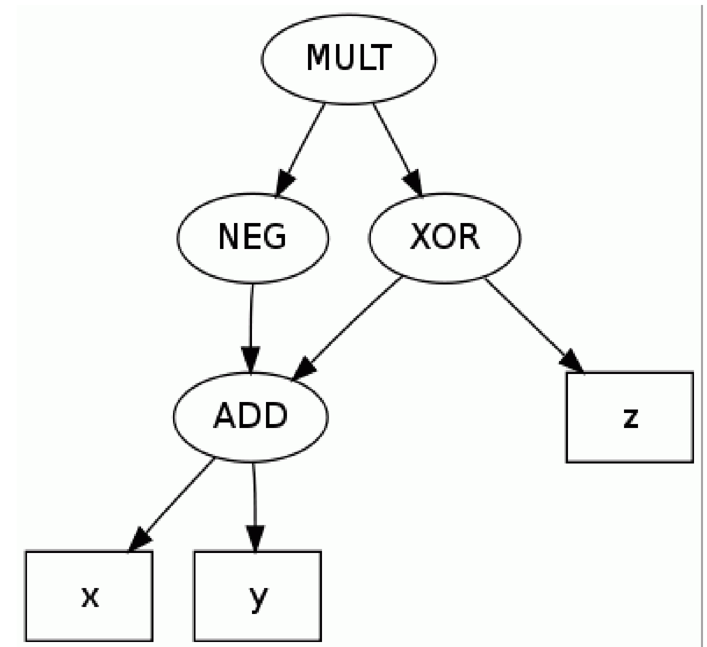


DAG Representation (Directed Acyclic Graph)

- Mathematical terms
- Operator trees with structure sharing
- Tree:



DAG:



- Compact:
  - Blowfish tree: ~6.9 x 10^{5186} nodes
  - Blowfish DAG: 220,639 nodes

Dealing with Loops

- Loop are one of the hardest aspects of formal verification.
- Induction proofs
  - require invariants
- Two approaches:
  - unroll loops
  - deal with loops inductively

Other Key Ideas

- Use test-cases to discover what to prove.
- Simplify DAGs whenever possible.
- Work at the word level (32-bit operations).

Complete Loop Unrolling by Symbolic Execution

- Extracts a DAG representing the Java code.
  - output bits in terms of input bits
- Step through execution on symbolic inputs.
  - Uses a formal model of the JVM, Axe Rewriter.
  - Follows approach of Moore et al.

- Amounts to unrolling all loops and inlining all method calls.
  - Many crypto. loops unrollable (10 rounds of AES-128).

Complete Loop Unrolling (continued)

- Also unroll formal spec.
  - Open function definitions

- Advantage: Loops are gone.
  - No program annotations needed.
- Disadvantage: Resulting DAGs are large

  $\sim 10^3$ to $\sim 10^5$ nodes

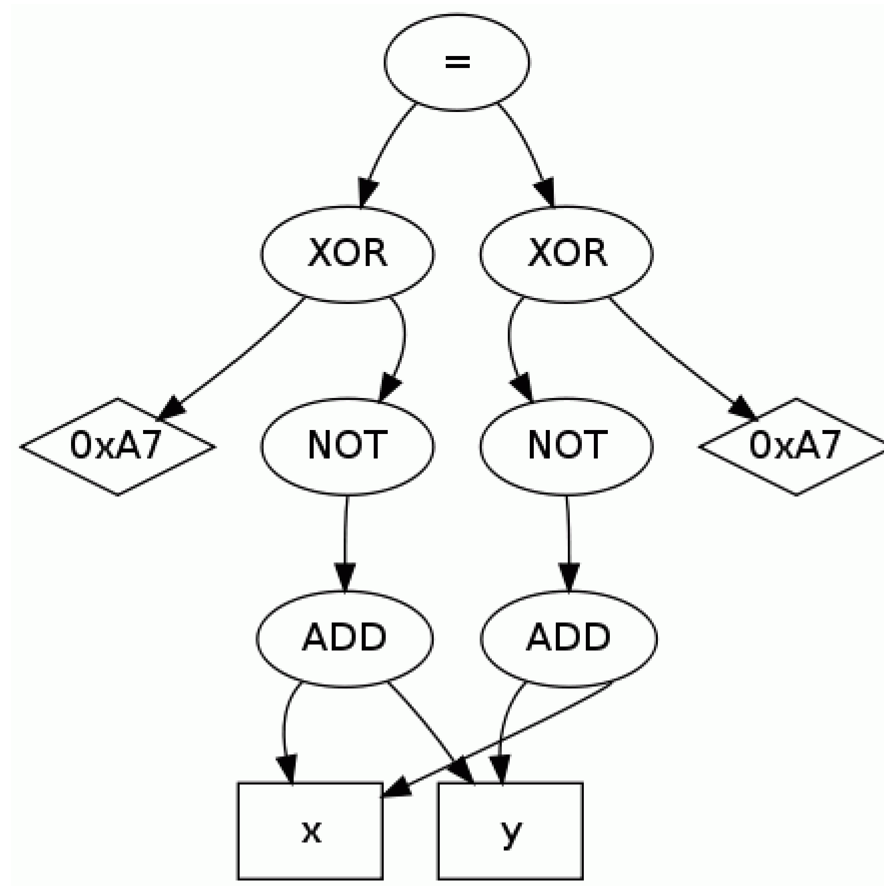
Equivalence Checking

- Form equality of two DAGs (“miter circuit”).
- Prove true for all inputs.
- Works well for hardware (bit level).
- We tried ABC (Berkeley) and STP (Stanford), and they were slow or ran for days and crashed.

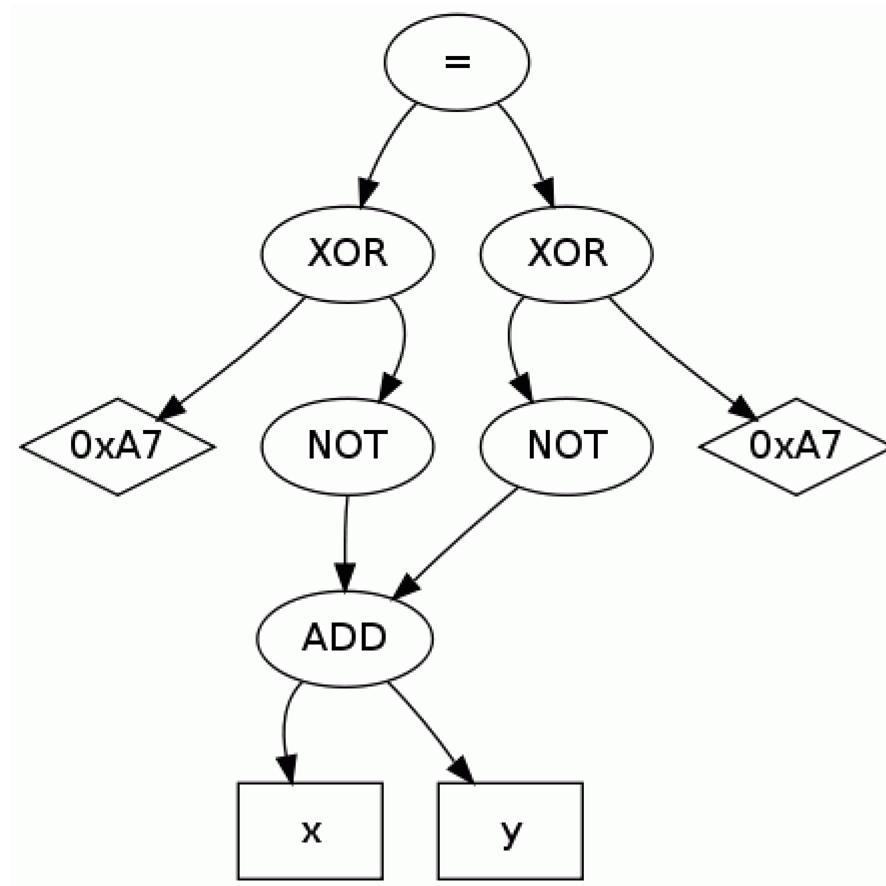
Axe Equivalence Checker

- Based on "sweeping and merging":
  1. Run test cases (assign values to nodes).
  2. Find "probably-equal" pairs of nodes.
  3. Sweep from bottom to top, proving and merging pairs.

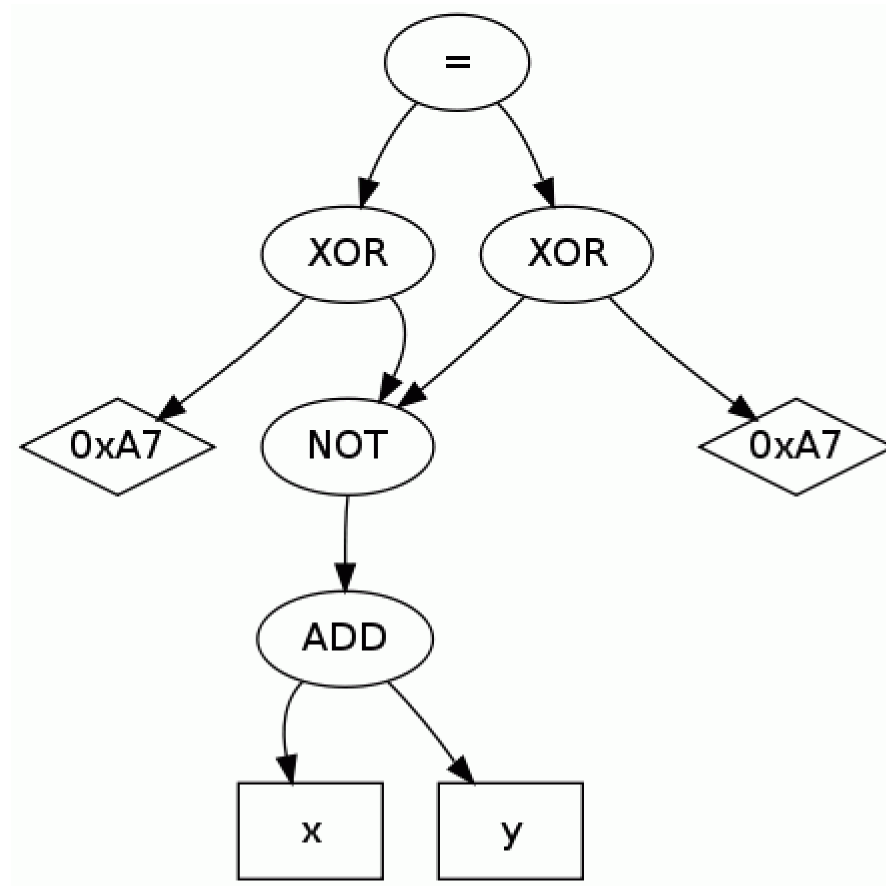
Sweeping and Merging Step 1



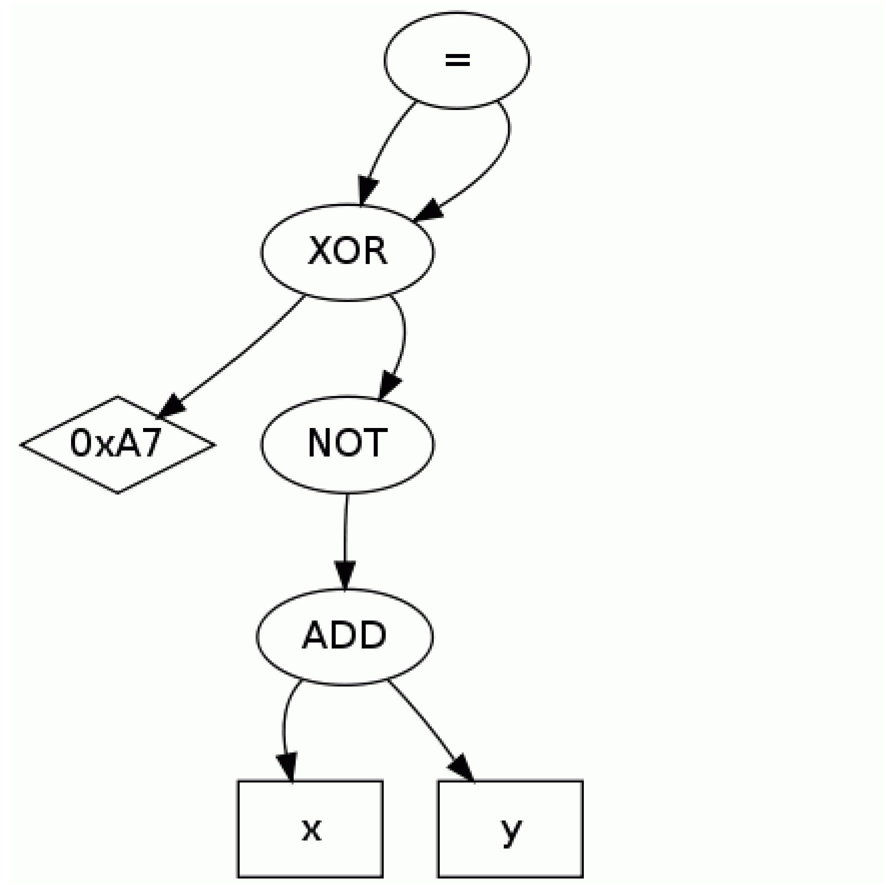
Sweeping and Merging Step 2



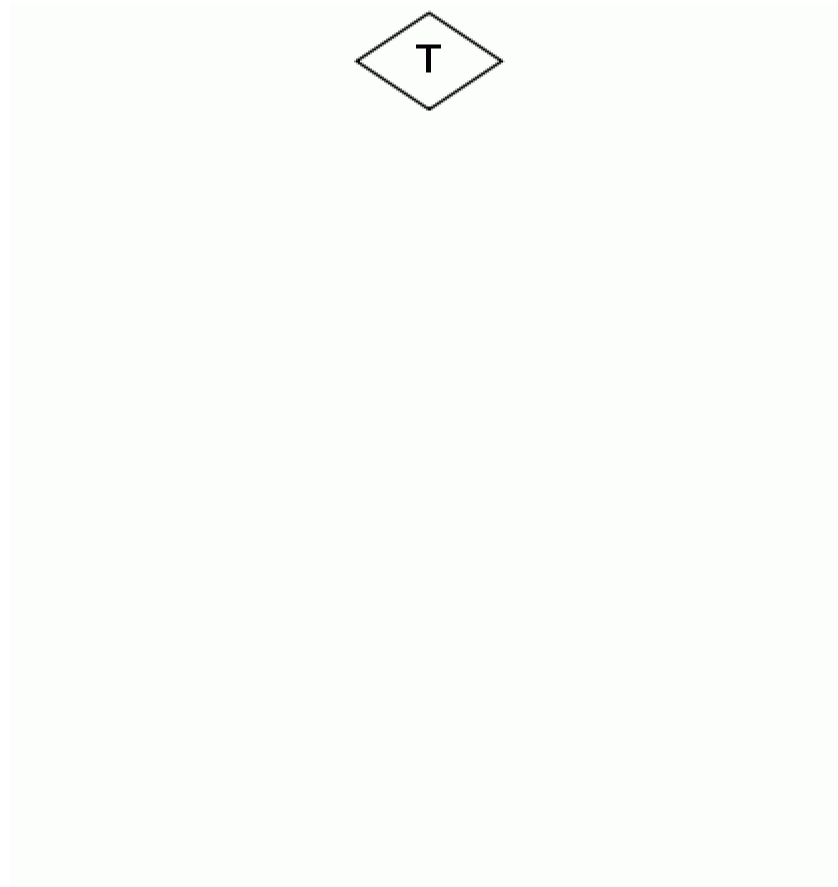
Sweeping and Merging Step 3



Sweeping and Merging Step 4



Sweeping and Merging Step 5



Sweeping and Merging

- Works well for crypto.
  - Implementations match up between “rounds.”
  - optimizations occur within rounds.

- To do the proof at each step, call STP (Dill and Ganesh)
  - SAT-based decision procedure for bit-vectors and arrays

Heuristic Goal Cutting

- STP times out on huge goals.
  - Sometimes can prove a smaller, more general goal.
  - Heuristics to find a good cut (incremental, binary search).
- Goal cutting works well for crypto.
  - Only the nodes for the current round are relevant.

Rewriting

- RC6 contains 32-bit multipliers
  - bad for SAT solvers
- Apply Axe Rewriter before sweeping and merging
  - Applies local simplification rules to DAGs.
  - Rules are proved ACL2 theorems.
  - Helps make equivalent terms more similar.
  - Easy to experiment with new rules, turn rules on and off.
  - Crucial for some examples.

Axe Rewriter Features

- Conditional rewrite rules
- Free variable binding
- Syntactic control
- Rewrite objectives
- Outside-in rewriting
- Memoization
- Use of contextual information
- Phases

Word-level Simplification

- Dozens of general-purpose simplification rules.
- We identified several crucial domain-specific simplifications for crypto. verification:
  - Concatenation
  - Rotation (needed for RC6)
  - Table Lookups

Bit-Vector Rotations

- No JVM bytecode for rotation.
- Common idiom: shift, shift, and combine:

```
  1010101000001111   (want to rotate right by 3)


  0001010101000001   (shifted right by 3)
+ 1110000000000000   (shifted left by 16-3=13)
  ----------------
  1111010101000001   (rotated right by 3)
```

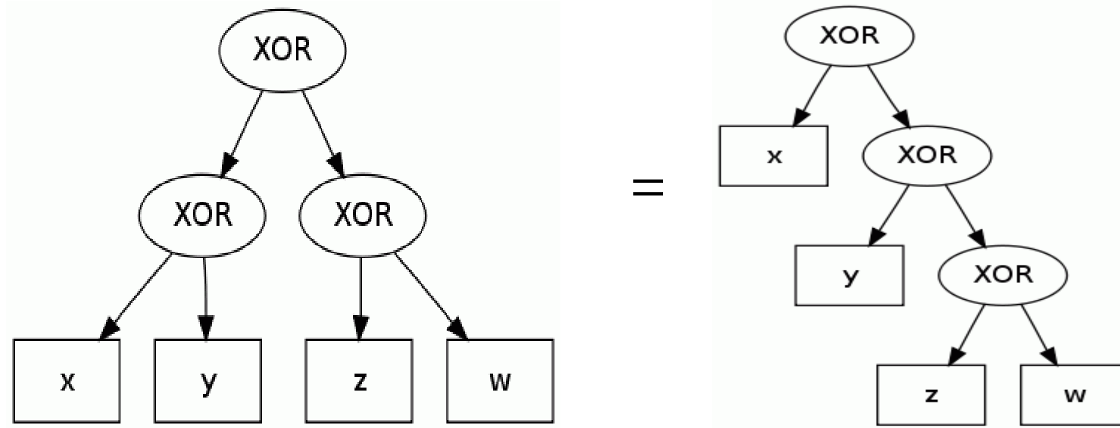
- Different idioms: could also combine with OR or XOR.

Variable Rotation Amounts

- Rotation amount depends on inputs (RC6).
- Cannot directly send to STP, or SAT.
- Normalize rotation operations:
  - rewrite to introduce LEFTROTATE operator
  - helps simplify RC6 equality DAG to TRUE.

XORs

- XOR is associative and commutative.
- For a given set of values, there are many equivalent nested XOR trees.



- Also: XOR(y,y) = 0 XOR(y,0) = y NOT(y) = XOR(y,1)
- XORs are very common in crypto. code.
- The Axe Rewriter contains custom code to normalize XORs.

Bit-blasting

- Convert multi-bit operations into concatenations of single-bit operations.
- Delay until after word-level simplification.
- Bit-blasting can obscure word-level properties.
  - Example: Bit-blasting (ADD 32 x y) into ripple-carry adder
  - Commutativity no longer clear!

Java Code Verified by Unrolling

- Rewriting alone:
  - bouncycastle.org: AES(unoptimized), RC2, RC6, Blowfish, Skipjack
  - Sun: RC2, Blowfish
- Rewriting and Sweeping and Merging:
  - bouncycastle.org: AES(Fast), AES(Medium), DES, 3DES, RC2
  - Sun: AES, DES, 3DES

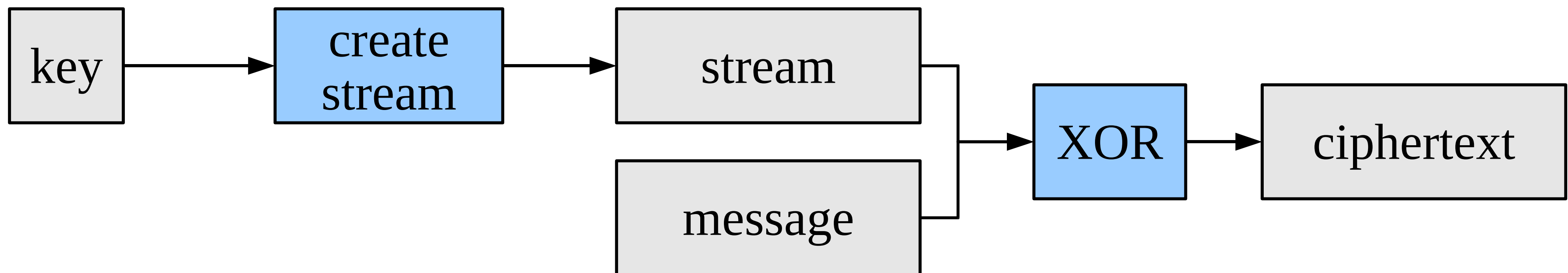
- No bugs found.
- Proof time: 30 seconds to a few hours.

Example: Skipjack

- Early examples were done in parallel with tool development.
  - Hard to estimate effort.
- Skipjack took less than three hours, including:
  - writing and debugging the formal spec
  - doing the equivalence proof

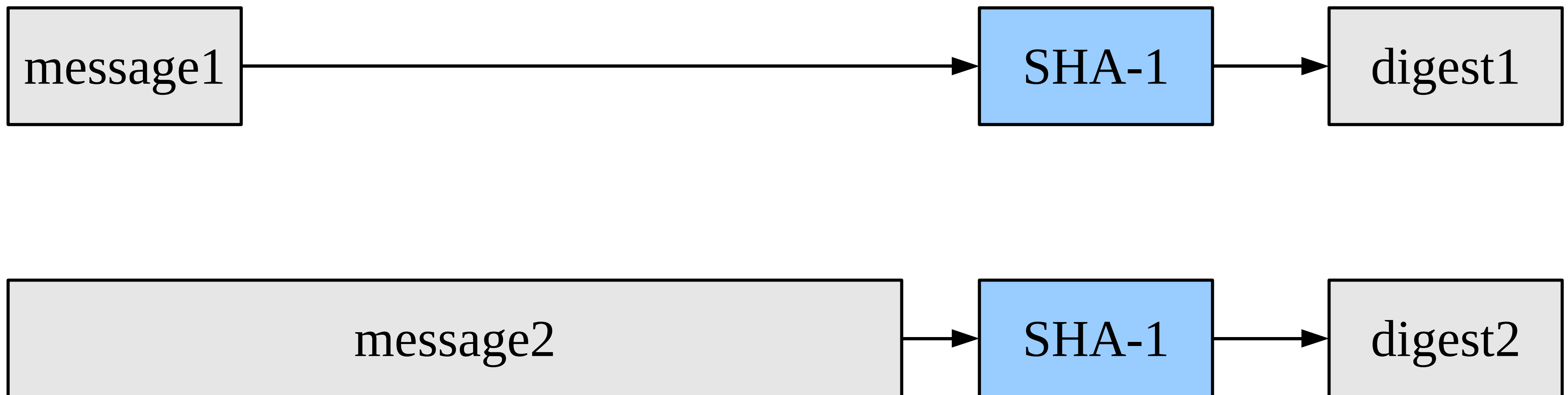
Stream Ciphers

- Encrypt and decrypt messages of any length.
- Example: RC4 (used in SSL/TLS, WEP, WPA)



Cryptographic Hash Functions

- Take a message of any length.
- Compute a fixed-size digest (16 or 20 bytes).
- Tiny change to message leads to totally different digest.
- Examples: SHA family (e.g., SHA-1), MD5



Restricted Input Size

- Hash functions and stream ciphers not unrollable:
  - Number of loop iterations depends on input.
  - Choose an input size and unroll.
- Results:
  - SHA-1 hash function on messages of 32, 512, or 4096 bits.
  - MD5 hash function on messages of 32, 512, or 4096 bits.
  - RC4 stream cipher with 64-bit key and 64-bit message, with 2048-bit key and 4096-bit message

Non-unrollable Loops

- Want a proof for all input sizes:

1. Decompile Java bytecode
   - Some DAG operators will be “loop functions” (recursive functions that model loops)
2. Call the Equivalence Checker.
   - Loop reasoning needed to merge nodes

Loop Properties

- Trying to merge two probably-equal nodes.
- They are the outputs of “equivalent” loops.
- Detect loop properties from traces.
  - connections between loops
  - also, loop invariants
- Prove by induction.

Connections

- Loop iterations counts must agree
  - (if not, transform one loop)
- Connections
  - Formulas over loop parameters.
  - Always true on corresponding iterations.
- Equalities, linear relationships, mod, etc.
- Detected from traces.
- Need to prove them!

Loop Property Detection

- Want to find candidate properties that are
  - true
  - provable
  - useful
- Key properties not clear from static analysis
  - x (in loop 1) always equal to y (in loop 2)
  - true for a deep reason
- Analyze execution traces.

Connections using Sums and Differences

- To connect x and y, check whether their sum is unchanged-per-trace. Then try to explain that value. Example:

```
x:    ( 0  1  2  3  4  5)

      ( 0  1  2)

      ( 0  1  2  3  4  5  6  7  8  9 10 11)


y:    ( 5  4  3  2  1  0)

      ( 2  1  0)

      (11 10  9  8  7  6  5  4  3  2  1  0)
```

Connections using Sums and Differences

- To connect x and y, check whether their sum is unchanged-per-trace. Then try to explain that value. Example:

```
x:   ( 0  1  2  3  4  5)

     ( 0  1  2)

     ( 0  1  2  3  4  5  6  7  8  9 10 11)


y:   ( 5  4  3  2  1  0)

     ( 2  1  0)

     (11 10  9  8  7  6  5  4  3  2  1  0)


x+y: ( 5  5  5  5  5  5)

     ( 2  2  2)

     (11 11 11 11 11 11 11 11 11 11 11 11)
```

Connections using Sums and Differences

- To connect x and y, check whether their sum is unchanged-per-trace. Then try to explain that value. Example:

```
x:    ( 0  1  2  3  4  5)
      ( 0  1  2)
      ( 0  1  2  3  4  5  6  7  8  9 10 11)

y:    ( 5  4  3  2  1  0)
      ( 2  1  0)
      (11 10  9  8  7  6  5  4  3  2  1  0)

x+y:  ( 5  5  5  5  5  5)
      ( 2  2  2)
      (11 11 11 11 11 11 11 11 11 11 11 11)
```

- x + y = oldy
- x = oldy - y

Properties of Single Loops

- Constant values: x=256
- Unchanged values: x=oldx
- Type facts (bit-vector, non-negative, array length)
- Mod facts

Numeric Bounds

- Often want to know bounds on loop variables.
  - to show array indices in bounds
- Other tools (Ernst's Daikon tool) might take the largest and smallest observed values.
  - May not capture all behaviors.
  - Axe looks at which values are in which traces.
- Axe checks whether every trace is the same.
  - If so, generate aggressive bounds.
  - If every trace for x is (0 4 8 12 16), assert x is in [0, 16].

Numeric bounds

- Three traces for i:

```
(0 1 2 3 4 5)
(0 1 2)
(0 1 2 3 4 5 6 7 8 9 10 11)
```

- Lower bound is clearly 0. What is the upper bound?

Numeric bounds

- Three traces for i:

```
(0 1 2 3 4 5)
(0 1 2)
(0 1 2 3 4 5 6 7 8 9 10 11)
```

- Lower bound is clearly 0. What is the upper bound?

```
Upper bounds for i:
                   5
                   2
                  11
```

Numeric bounds

- Three traces for i:

```
(0 1 2 3 4 5)
(0 1 2)
(0 1 2 3 4 5 6 7 8 9 10 11)
```

- Lower bound is clearly 0. What is the upper bound?

Upper bounds for i:	len(input):
5	21
2	18
11	27

Numeric bounds

- Three traces for i:

```
(0 1 2 3 4 5)

(0 1 2)

(0 1 2 3 4 5 6 7 8 9 10 11)
```

- Lower bound is clearly 0. What is the upper bound?

```
Upper bounds for i:        len(input):
                  5        -        21        =
                  2        -        18        =
                 11        -        27        =
```

Numeric bounds

- Three traces for i:

```
(0 1 2 3 4 5)

(0 1 2)

(0 1 2 3 4 5 6 7 8 9 10 11)
```

- Lower bound is clearly 0. What is the upper bound?

```
Upper bounds for i:        len(input):
                   5     -      21     =    -16
                   2     -      18     =    -16
                  11     -      27     =    -16
```

Numeric bounds

- Three traces for i:

```
(0 1 2 3 4 5)

(0 1 2)

(0 1 2 3 4 5 6 7 8 9 10 11)
```

- Lower bound is clearly 0. What is the upper bound?

```
Upper bounds for i:        len(input):
                5      -      21     =    -16
                2      -      18     =    -16
               11      -      27     =    -16
```

- Aha! i <= len(input) - 16

Proving Connections

- First, prove individual loop invariants.
- Assume:
  - the connections hold before the loop bodies
  - neither loop exits
  - both loop invariants hold
- Prove: connections hold after loop bodies.

Proving a Connection (continued)

- Proof is done with Equivalence Checker
  - Rewriting, sweeping and merging, etc.
  - Nested loops handled (often unrolled)
- Some failures tolerated
  - Find maximal provable subset of candidates.

Connecting Two Loops

- Prove that exit tests agree
  - assume the connection and invariants
  - use Axe Prover

- Characterize final loop values
  - connection: first n elements of array1 and array2 agree
  - the loops exit when n=len(array1)=len(array2)
  - array1=array2 on exit

Main Theorem About Two Loops

- If the values passed in satisfy the connection and both functions' loop invariants ...

  ... then the final claims hold after the loops.
- Proved by induction.
  - “Old variables” handled appropriately.
- Expressed as a rewrite rule.
- Used by Axe Prover to merge the nodes supported by the loops.
  - and possibly for other merges.

Loop transformations

- Help synchronize the loops of two programs.
  - also can simplify individual loops
- Typical operation:
  - Mechanically generate a new loop function.
  - Prove equivalence.
  - Express equivalence as a rewrite rule.

Loop Transformations

- Constant-factor unrolling.
- Loop fission.
- Combine producer and consumer (“streamify”).
- Completely unroll for bounded iterations (not constant).
- Loops that exit immediately.
- Put loops into “simple form.”
- Loops that walk down a list.
- Drop “redundant” loop parameters.

Inductive Equivalence Checking Results

- Verified RC4 stream cipher
  - Any length message (must fit into a Java array).
  - Any length key (except 0).
  - No user annotations required.
  - All loops handled inductively.
- Verified SHA-1 and MD5 hash functions
  - Any length message.
  - Some annotations required (could automate)
  - Hybrid approach: unroll inner loops

Conclusion

- Axe can greatly reduce the effort of crypto. verification.
- Strong correctness results (bit for bit equivalence).
- Applied to real code.
- Formal proofs may now be a viable alternative to testing for this domain.

Contributions

- Word-level equivalence checking.
  - unroll, rewrite, bit-blast, rewrite again, sweep and merge
  - crypto-specific simplifications.
- Inductive equivalence checking of programs with loops.
  - Integrated with unrolling techniques.
- Test-based identification of properties to prove.
- Axe system (Equivalence Checker, Rewriter, Prover)
  - Large library of proved simplification rules
- (Infrastructure may be useful to others: JVM model, bytecode parser and decompiler, general purpose rewriter, simplification rules, prover, combinational equivalence checker.)

Inversion Proofs

- Functional Correctness Proofs of Encryption Algorithms (Duan, Hurd, Li, Owens, Slind, Zhang)
  - Proved cipher inversion
    - weaker property
    - satisfied by trivial insecure cipher.
    - ignores key expansion.
  - Manual effort to guide the prover.
  - Does not verify pre-existing implementations.
    - Implementation in higher order logic.

Echo Tool

- Formal Verification by Reverse Synthesis (Yin, Knight, Nguyen, Weimer)
  - Verified an AES implementation.
  - Transforms the code by undoing optimizations.
  - User must specify some of the transformations:
    - Must find instances of work packing.
    - Must specify the patterns encoded in lookup tables.

Combinational equivalence checking

- Use random test cases to find equalities (Berman and Trevillyan, 1989)
- Prove equivalences bottom-up (Kuehlmann)
- SAT-sweeping / fraiging
- BDDs
  - give equivalent computations the same representation
  - but may take exponential space and are sensitive to variable ordering
- AIGs ("and-inverter graphs")
  - low-level, big, but good tool support
  - don't exploit word-level properties

Galois Connections and NSA

- Equivalence checking of similar VHDL designs
  - Represent as AIGs
  - SAT-Sweeping.
  - ABC or jaig equivalence checker (Berkeley, Galois)
- Verified an AES implementation.
  - AES is relatively easy.
- Verified Skein hash function on 256-bit inputs (17.5 hours).
- MD5 takes 10 minutes on 2-bit input (!)
- Doesn't work for RC6 (32-bit multiplications)

Galois Connections and NSA (continued)

- Cryptol language
  - Can be compiled down to an implementation using verified compiler transformations.
  - Correct by construction framework.

Translation Validation

- Check each individual step of a synthesis or compilation process.
- Uses equivalence checking.
- Designs compared are often similar.
- Tool can track the transformations it performs.
  - indicates how things match up.

Inductive Assertions, Cutpoints, Hoare Logic

- Label key program points with assertions
  - Must cut all the loops.
- Check that all paths between cutpoints preserve assertions.
- Verification conditions generated using weakest preconditions (or strongest postconditions).
- JML (Java Modeling Language)

ACL2

- Axe is built on top of ACL2 theorem prover (Boyer, Moore, Kaufmann)
- ACL2 terms are trees, not DAGs.
- ACL2 proofs generally require manual guidance.
- JVM model and symbolic simulation based on M5 model (Porter and Moore)

Static Analysis

- Abstract interpretation (Cousot and Cousot, Karr):
  - Numeric properties: intervals, polyhedra, etc.
  - Over-approximation (due to choice of domain, widening)
  - Terminates (due to widening)
- Check generic properties on big programs.
  - Coverity, Saturn
- Axe uses dynamic analysis
  - Some things hard to detect statically but obvious from traces.

Dynamic Invariant Detection

- Daikon
  - detects properties of Java programs
  - focuses on procedure entry and exit (not loop invariants)
  - guess and check (try all possible instantiations of a template)
  - mostly treats samples as a flat set

Future Work

- Extend Axe to other languages (C, hardware)
  - just extract computation as a DAG
  - or compile to JVM bytecode
- Do more examples (Threefish, Skein, block cipher modes of operation)
- Automate detection of when to apply loop transformations (unrolling, splitting):
  - use test cases
- Detect more kinds of loop invariants.

- Please ask questions!

References

[1] Federal Information Processing Standard Publication 197. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[2] C. L. Berman and L. H. Trevillyan. Functional comparison of logic designs for VLSI circuits. In Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design, Nov. 1989.

[3] Galois Connections. Cryptol. http://www.cryptol.net/.

[4] J. Duan, J. Hurd, G. Li, S. Owens, K. Slind, and J. Zhang. Functional correctness proofs of encryption algorithms. In G. Sutcliffe and A. Voronkov, editors, Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005), volume 3835 of Lecture Notes in Artificial Intelligence. Springer-Verlag, December 2005.

Refences (cont.)

[5] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In Computer Aided Verification (CAV ’07), Berlin, Germany, July 2007. Springer-Verlag.

[6] Matt Kaufmann and J Moore. ACL2 Version 3.2. http://www.cs.utexas.edu/users/moore/acl2/.

[7] Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In Design Automation Conference, pages 263–268, 1997.

[8] J Moore. Proving Theorems about Java and the JVM with ACL2. http://www.cs.utexas.edu/users/moore/publications/marktoberdorf-02/index.html.

[9] N.I.S.T. and Lawrence E. Bassham III. The Advanced Encryption Standard Algorithm Validation Suite (AESAVS). http://csrc.nist.gov/groups/STM/cavp/documents/aes/AESAVS.pdf.

References (cont.)

[10] Lee Pike, Mark Shields, and John Matthews. A verifying core for a cryptographic language compiler. In ACL2 ’06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications, pages 1–10, New York, NY, USA, 2006. ACM.

[11] Ronald L. Rivest, M.J.B. Robshaw, R. Sidney, and Y.L. Yin. The RC6 Block Cipher. http://theory.lcs.mit.edu/∼ rivest/rc6.pdf.

[12] Diana Toma and Dominique Borrione. SHA formalization. In Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 ’03), July 2003.

[13] Sean Weaver. Equivalence checking. http://gauss.ececs.uc.edu/Courses/C702/Weaver/ec.01.08.07.ppt.

[14] Xiang Yin, John C. Knight, Elisabeth A. Nguyen, and Westley Weimer. Formal verification by reverse synthesis, to appear in SAFECOMP 2008.

Related work (cont.)

- Toma and Borrione used the ACL2 theorem prover to verify a hardware implementation of SHA-1
  - Seemed to require manual effort to guide the prover.

JVM Bytecode Decompiler

- Produces a DAG representing the result computed by the code
  - e.g., ciphertext as function of plaintext and key
- Uses Axe Rewriter and JVM model.
- Loops are modeled as stylized recursive functions

```
(defun rc4-loop-1 (params)
  (if (rc4-loop-1-exit-test params)
      params
      (rc4-loop-1 (rc4-loop-1-update params))))
```

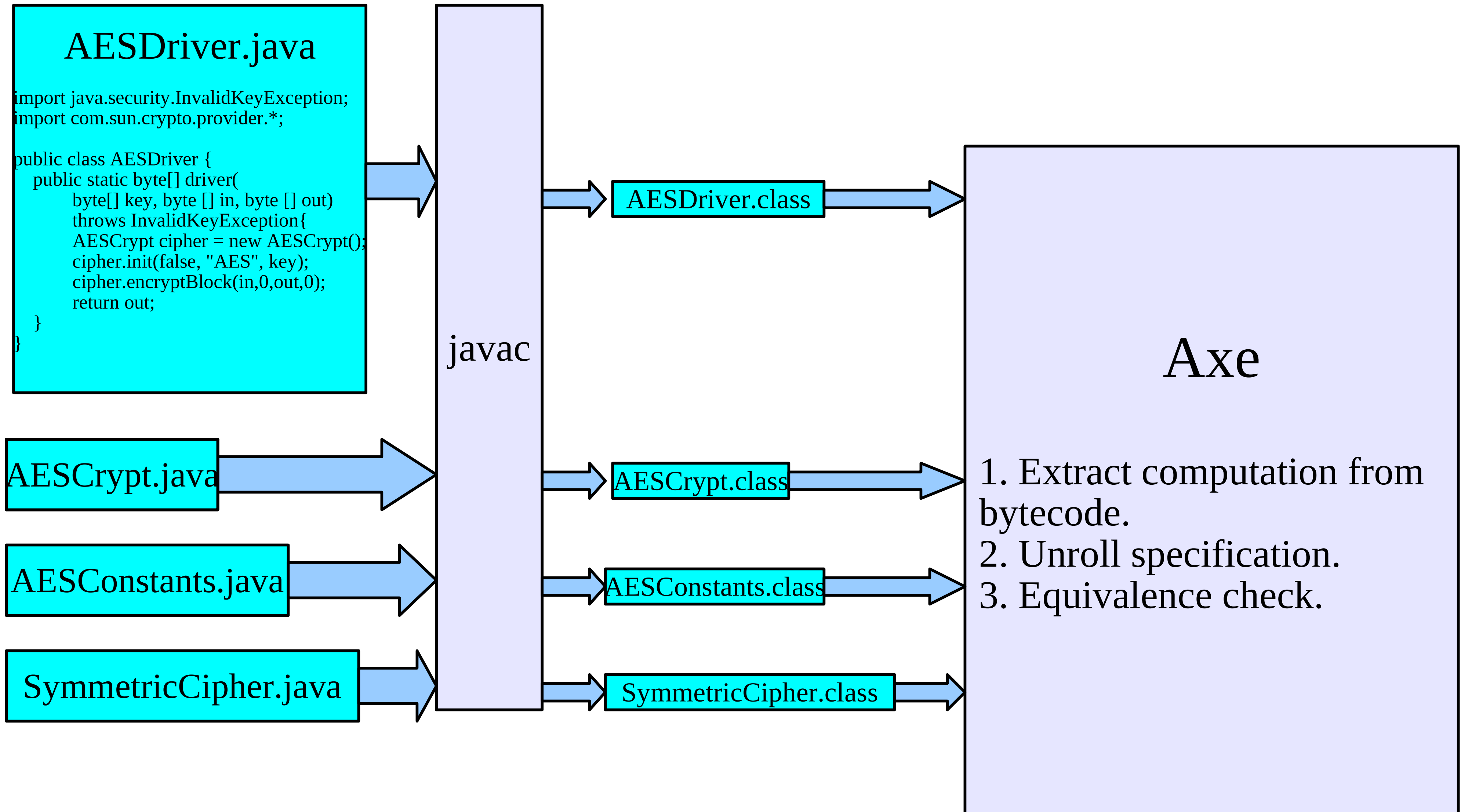
Decompiler Details

- Input: Java classes, input assumptions
- Output: symbolic JVM state after running driver
  - Extract the state component you care about (output buffer)
- Symbolically executes segments of code (loop bodies).
- Builds recursive functions.
- Parameters of the loop function
  - include each loop variable (local variable or object field)
  - can be entire arrays
  - do not include a heap (cannot decompile all programs)

Decompiler Details (continued)

- May need to generate and check invariants
  - the classes of methods on which objects are called
  - facts about pointers not changing during loops
- Conditional branches
  - state is an if-then-else nest
  - loop exit test combines all branches that leave the loop
  - loop body combines all branches that return to the loop top
- Class initialization
- Exceptions

Using Axe



Lookup tables

- Replace sequences of logical operations, for speed.
- Appear as array terms with constant elements.
- Lookups can be turned back into logic to match the specs.
  - Usually the logic will involve XORs.
- Our approach:
  - Blast the tables to handle each bit position of the elements separately.
  - Look for index bits that are irrelevant or XORed in.

Lookup table example

- Based on a real block cipher operation:
- Consider a three-bit quantity: $x = x_2 x_1 x_0$
- Want to compute:
  - $(x_2 \oplus x_1) @ (x_2 \oplus x_0) @ (x_1 \oplus x_0)$
- XORing two of the bits would require several operations: shift, XOR, mask, shift result into position.

Lookup table example (cont.)

Could simply compute $(x_2 \oplus x_1)$ @ $(x_2 \oplus x_0)$ @ $(x_1 \oplus x_0)$
from $x_2x_1x_0$ using the table:
T[000] = 00000000
T[001] = 00000011
T[010] = 00000101
T[011] = 00000110
T[100] = 00000110
T[101] = 00000101
T[110] = 00000011
T[111] = 00000000

Lookup table example (cont.)

T[000] = 00000000
T[001] = 00000011
T[010] = 00000101
T[011] = 00000110
T[100] = 00000110
T[101] = 00000101
T[110] = 00000011
T[111] = 00000000

- Want to turn the table back into logic
- Bit-blast the table into single-bit tables
  - One table per column.
  - A lookup in T is now a concatenation of 8 lookups in the 1-bit tables.
- Recognize tables where the data values are all the same:
  - First 5 columns of T contain only 0s.
  - Lookup into a table of 0's returns 0.

Lookup table example (cont.)

T0[000] = 0
T0[001] = 1
T0[010] = 1
T0[011] = 0
T0[100] = 0
T0[101] = 1
T0[110] = 1
T0[111] = 0

- Recognize when tables have irrelevant index bits
  - T0 does not depend on x_2
  - First and second halves of the table are the same.
- Recognize when table values have index bits XORed in.
  - T0 has x_0 XORed in
  - When x_0 goes from 0 to 1, the table value always flips.
- The value of T0[$x_2x_1x_0$] is $(x_1 \oplus x_0)$.

Concatenation Example

Concatenation is used to pack bytes into machine words.

Ex: To concatenate:

```
        10101010
        11110000
```

shift one operand and OR the results:

```
1010101000000000
0000000011110000
----------------
1010101011110000
```

The shift introduces zeros. We never OR two ones together.
So we could also use XOR or ADD instead.

Concatenation Example

- Three different idioms (combine using OR, XOR, ADD)
- Rewrite all three to use a concatenation operator
  - Unique representation.
  - Reflects what's really going on.
- Rules are a bit tricky
  - Require the presence of zeros so that we never combine two ones
  - Trickier when more than two values are being concatenated.
- Could always just bit-blast these operations away, but better to work at the word level.

Results (cont.)

- 4 AES implementations:
  - org.bouncycastle.crypto.engines.AESEngine
  - org.bouncycastle.crypto.engines.AESLightEngine
  - org.bouncycastle.crypto.engines.AESFastEngine
  - com.sun.crypto.provider.AESCrypt
- 2 operations (encrypt and decrypt) for each
- 3 key lengths (128, 192, and 256 bits) for each
- 24 AES proofs

Normalizing XOR nests

- We normalize XOR nests to have the following properties:
  - All XOR operations are binary and associated to the right.
  - Values being XORed are sorted (by node number, with constants at the front)
  - Pairs of the same value are removed.
  - Multiple constants are XORed together, and a constant of 0 is dropped.
  - Negations of values being XORed are turned into XORs with ones. (The ones are pulled to the front and combined with other constants.)
- Result: Equivalent XOR nests are made syntactically equal.

Facts About Mod

- Handles index increment other than 1.
- Example: Every trace is (0 4 8 12 16)
- Yields the invariant x mod 4 = 0.
- If modulus is a power of 2, a bit slice is involved.

Handling a Single Loop

- Prove a theorem about the values after the loop
  - Used as a rewrite rule.
  - Helps justify merging nodes:
    - gives type information
    - gives bound information (helpful for array indices)
    - possible "closed form" expression
      - increment i by 1 for j iterations: i = oldi + oldj
    - some variables unchanged by the loop
  - Also used when comparing two loops.

Loop Invariants

- Formulas over loop parameters and “old” variables.
- Hold on each recursive call
  - each time the “loop top” is reached
- Run the loop and analyze execution traces.
- Result is a set of candidate invariants.
- Must prove them!

Proving a Loop Invariant

- Show loop body preserves the invariant.
- Assume all loop invariants hold before the body.
- Can assume loop does not exit.
- Proof done with the Axe Equivalence Checker
  - handles nested loops (unroll or prove loop properties)
  - test cases come from loop traces

Maximal Provable Subset of Invariant Candidates

- Seek an "inductive" subset of loop invariants.
- Try to prove every candidate in the set.
- If any fail to prove, discard them and repeat.
- Failures tolerated (but waste time).
- The inductive invariant is the conjunction of surviving candidates.

Invariant Improvement

- Gets rid of weaker invariants.
  - were included because the strong ones might have failed.
- Pushes back mentions of unchanged variables.
- Puts in constants when possible.
  - Ex: x >= oldx and oldx = 0   give   x >= 0
- Simplifies phrasing
  - (sbvlt 32 x y) to (bvlt 31 x y) when x and y non-negative
- Returns a nicer but equivalent invariant.

Final Claims

- Characterize values returned by the loop.
- Stronger than the loop invariant.
  - loop exit test is also true
  - invariant: i<=j,   loop exits when i>=j,   so i=j upon exit
- Detect from traces and then prove (using Axe Prover)
  - Must me implies by loop invariant and exit test.

The Main Theorem for a Loop

- If invariant holds on the initial values ...

  ... then final claims hold after the loop.
- Proved by induction.
  - Axe does the hard parts (inductive step, base case)
  - Calls ACL2 to assemble the pieces
- “old variables” handled appropriately.
- Used as a rewrite rule for all future merges.

Proving A Merge Involving Loops

- Prove a theorem about each loop.
- Call the Axe Prover
  - Rewriting (using loop theorems and other rules)
  - Substitution
  - “Tuple” Elimination
  - Case splitting
  - Calls STP (cuts out operations STP can't handle)
- Proof can use “context” information.
- If the Prover fails, we may need a theorem connecting two “equivalent” loops.

Final Claims

- Similar to the final claims about a single loop.
- Connection must hold on every iteration.
- Exit tests hold on final iteration: Can make stronger claims.
- Detect from traces
- Prove with Axe Prover (assume connection, invariants, and loop exit tests)
- Example:
  - Loops are filling in array elements (same array length).
  - Connection: the first n elements of the arrays agree.
  - The loops exit when n is the array length.
  - On the final iteration, the entire arrays are equal.

Handling a Single Loop

- Prove a theorem about the values after the loop
  - Used as a rewrite rule.
  - Helps justify merging nodes:
    - gives type information
    - gives bound information (helpful for array indices)
    - possible “closed form” expression
      - increment i by 1 for j iterations: i = oldi + oldj
    - some variables unchanged by the loop
  - Also used when comparing two loops.

Loop Invariants

- Formulas over loop parameters and “old” variables.
- Hold on each recursive call
  - each time the “loop top” is reached
- Run the loop and analyze execution traces.
- Result is a set of candidate invariants.
- Must prove them!

Proving a Loop Invariant

- Show loop body preserves the invariant.
- Assume all loop invariants hold before the body.
- Can assume loop does not exit.
- Proof done with the Axe Equivalence Checker
  - handles nested loops (unroll or prove loop properties)
  - test cases come from loop traces

Maximal Provable Subset of Invariant Candidates

- Seek an “inductive” subset of loop invariants.
- Try to prove every candidate in the set.
- If any fail to prove, discard them and repeat.
- Failures tolerated (but waste time).
- The inductive invariant is the conjunction of surviving candidates.

Invariant Improvement

- Gets rid of weaker invariants.
  - were included because the strong ones might have failed.
- Pushes back mentions of unchanged variables.
- Puts in constants when possible.
  - Ex: x >= oldx and oldx = 0   give   x >= 0
- Simplifies phrasing
  - (sbvlt 32 x y) to (bvlt 31 x y) when x and y non-negative
- Returns a nicer but equivalent invariant.

Final Claims

- Characterize values returned by the loop.
- Stronger than the loop invariant.
  - loop exit test is also true
  - invariant: i<=j,   loop exits when i>=j,   so i=j upon exit
- Detect from traces and then prove (using Axe Prover)
  - Must me implies by loop invariant and exit test.

The Main Theorem for a Loop

- If invariant holds on the initial values ...

  ... then final claims hold after the loop.
- Proved by induction.
  - Axe does the hard parts (inductive step, base case)
  - Calls ACL2 to assemble the pieces
- “old variables” handled appropriately.
- Used as a rewrite rule for all future merges.

Proving A Merge Involving Loops

- Prove a theorem about each loop.
- Call the Axe Prover
  - Rewriting (using loop theorems and other rules)
  - Substitution
  - “Tuple” Elimination
  - Case splitting
  - Calls STP (cuts out operations STP can't handle)
- Proof can use “context” information.
- If the Prover fails, we may need a theorem connecting two “equivalent” loops.

Explanations

- Try to explain one value using other values:
- To connect variables of two loops.
- To find “redundant” loop parameters.
- (Axe avoids circular explanations.)
- Constants: x=3
- Equalities: x=y,   x = len(thearray)
- Mod facts: x = y mod z
- Linear relationships: x=2*y
- Sums and differences.

Complete Unrolling

- Requires a bound on the number of loop iterations.
  - Not necessarily constant! Example: 0, 1, or 2 iterations.
  - Tricky to detect (finite test suite: everything looks bounded)
- Does not apply to all loops.
- Done when user specifies it, or when there is no other choice (comparing a loop to something that contains no loop).
- Force the loop to unroll k times.
- Then prove k iterations is sufficient.

Loops that Exit Immediately

- Detected from traces.
- Loop always just returns.
- Common for the “residual” computation after complete unrolling.
- Unroll one more step
  - Exposes exit test to the Equivalence Checker.
  - Exit test will be “probably true.”

Simple Loop Form

- Putting a loop into “simple loop form”
  - Making a list builder tail-recursive.
  - Peeling off a non-trivial base case.
  - Combining several base cases.
- Sometimes needed after other transformations (dropping, constant factor unrolling).

“Un-cdr-ing”

- Specification functions often process lists by repeatedly calling cdr on them
  - Lists get shorter each iteration.
- Doesn't match up well with Java code
  - Array lengths don't change.
- Mechanically generate a new function:
  - Passes the list unchanged to the recursive call.
  - Keeps a counter, n.
  - cdrs the list n times right before using it in the body.

Dropping Redundant Loop Parameters

- Some parameters explainable in terms of others:
- x = 16 drop x from the loop and hard code 16
- x = y drop x from the loop (just use y)
- x = y mod len(z) drop x (calculate from y and z as needed)
- x = oldy - y replace x with oldy (unchanged, so it's a win)
- Axe mechanically generates a new, simpler function and proves equivalence.
- Helps when proving two loops equivalent
  - issues of redundancy already dealt with
  - fewer possible relationships

Constant Factor Unrolling

- Loops are equivalent, but one loop does more work per iteration.
  - Common optimization to eliminate jumps.
- For equivalence proof, loops must be in sync.
- Unroll one loop by a constant factor (e.g., 16).
  - Mechanically generate a new function.
  - Prove equivalence (yields a rewrite rule).
  - Tricky part: extra iterations (not a multiple of 16)
- Used for SHA-1, MD5 proofs.

Loop Fission

- One loop does more iterations than the other.
  - Example: handling pad blocks in the loop or separately
- Split the loop that does more iterations.
  - Split amount may be symbolic.
  - Example: number of complete blocks = len(message) div 16
- Mechanically generate a new, limited version of the loop.
- Call the limited loop first, then the original loop.
- Prove equivalence, generates a rewrite rule.
- One of the pieces may be completely unrollable.
- Used in SHA-1 and MD5 proofs.

Streamifying

- One loop produces a sequence (elements don't depend on previous elements). Example: RC4 pseudorandom stream
- Another loop consumes the sequence (never accesses elements out of order). Example: XOR stream and message
- Could instead produce the elements “just in time”.
  - Need not actually make the entire sequence.
  - Single loop that produces and consumes.
- Mechanically generate new function, make a rewrite rule.
- Used in proof of RC4.