Axe: An Automated Formal Equivalence Checking Tool for Programs

Eric W. Smith
Kestrel Institute

joint work with David Dill (Stanford)

STANFORD
COMPUTER SCIENCE

Crypto is Worth Getting Right

- Important applications
  - Privacy, E-commerce, National Security, Military, Voting
- Want to ensure no accidental or deliberate errors.
- Focus on algorithmic “building blocks”
  - Block ciphers, stream ciphers, hash functions
  - Errors could compromise the security of larger systems

Crypto Validation in Practice

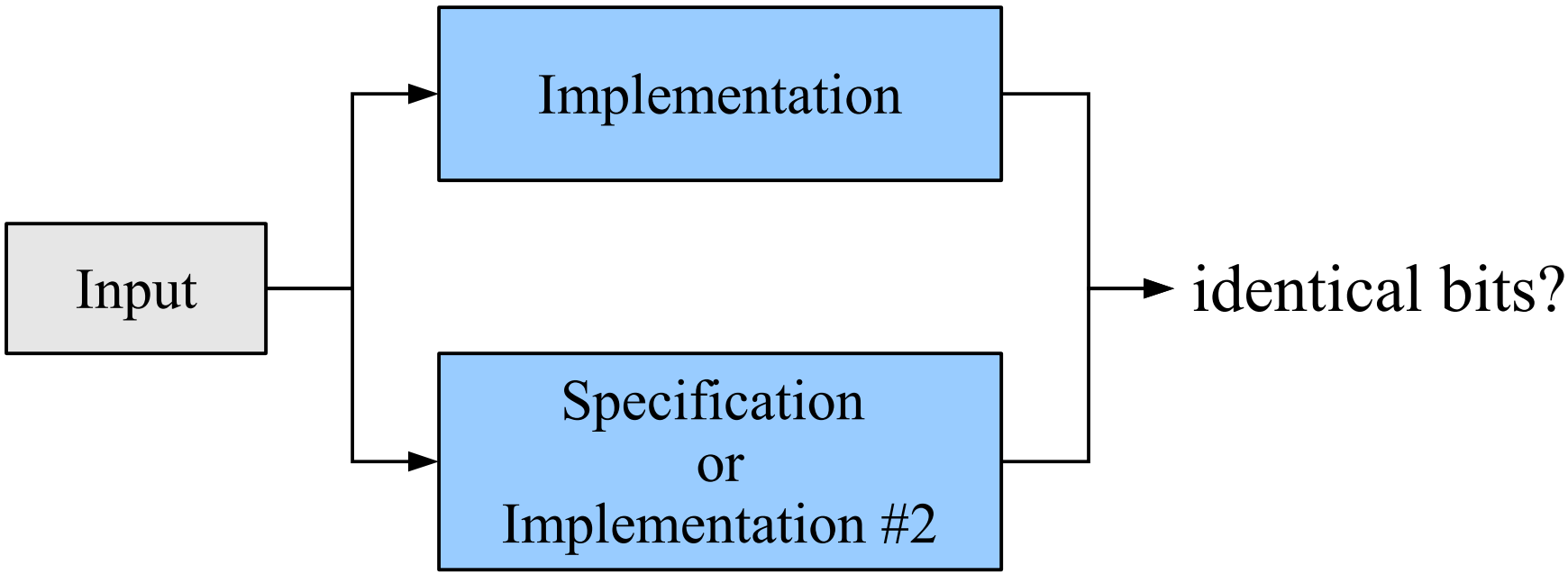
- Lots of tests
  - NIST tests AES implementations.
  - They admit that certification is not a proof.
- Too many inputs to test them all
  - at least 2^{256} for AES
- We want a *proof*.

Formal Proofs of Programs

- A grand challenge for many years
  - Turing, Dijkstra, Floyd, Hoare, Pnueli, Manna, Clarke, Emerson, Boyer, Moore
  - most properties are undecidable
- Not going to solve it in this talk!
- Restrict the domain to crypto.
  - Can automate the proofs.
- Other techniques can prove generic properties of huge programs (memory issues, threading, divide-by-0).
  - Axe proves full correctness of smaller programs.

What Axe Proves

- *Functional correctness:*



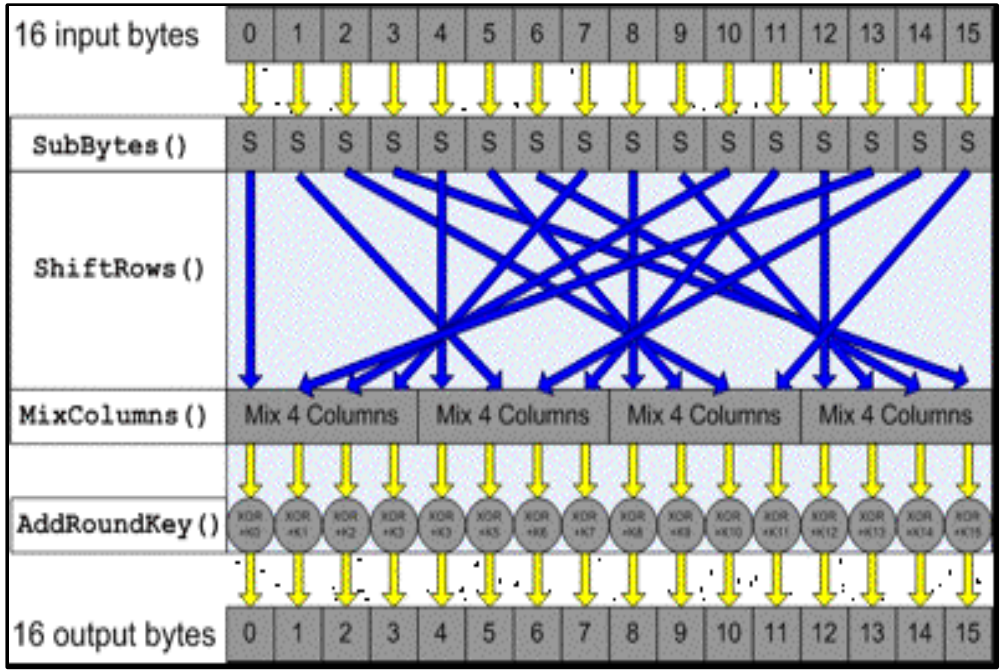
Axe Has Verified

- 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)

- Shows feasibility of proving this kind of code.

Challenges of Crypto Verification

- Algorithms designed to be intractable.
- Lots of mixing (leads to huge terms).



- Optimized implementations.

Excerpt of AES Block Cipher

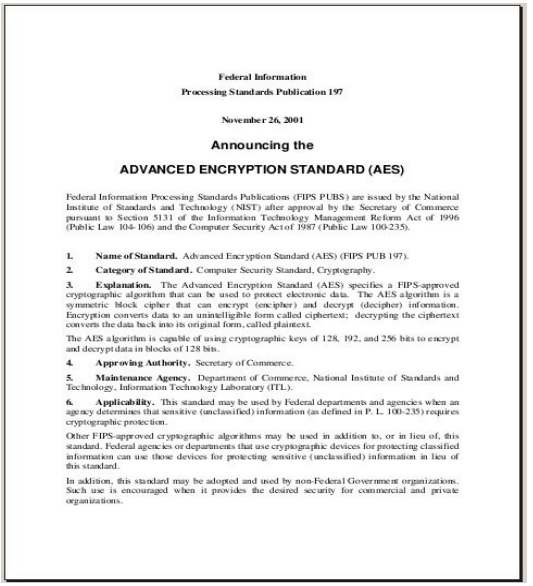
```
private void encryptBlock(int[][] KW) {
  int r, r0, r1, r2, r3;
  C0 ^= KW[0][0];
  C1 ^= KW[0][1];
  C2 ^= KW[0][2];
  C3 ^= KW[0][3];

  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 from Optimized AES

```
private static final int[] T0 =
{
 0xa56363c6, 0x847c7cf8, 0x997777ee, 0x8d7b7bf6, 0x0df2f2ff, 0xbd6b6bd6, 0xb16f6fde, 0x54c5c591, 0x50303060, 0x03010102,
 0xa96767ce, 0x7d2b2b56, 0x19fefee7, 0x62d7d7b5, 0xe6ababd4d, 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

- Crypto. specifications are precise.
  - FIPS-197 describes AES.
  - Specifies the exact value of each output bit.
- We formalized the English descriptions:
  - in a precise, mathematical language (ACL2)
  - closely match the official descriptions (unoptimized)
  - validated by running test cases.
  - reusable.



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.

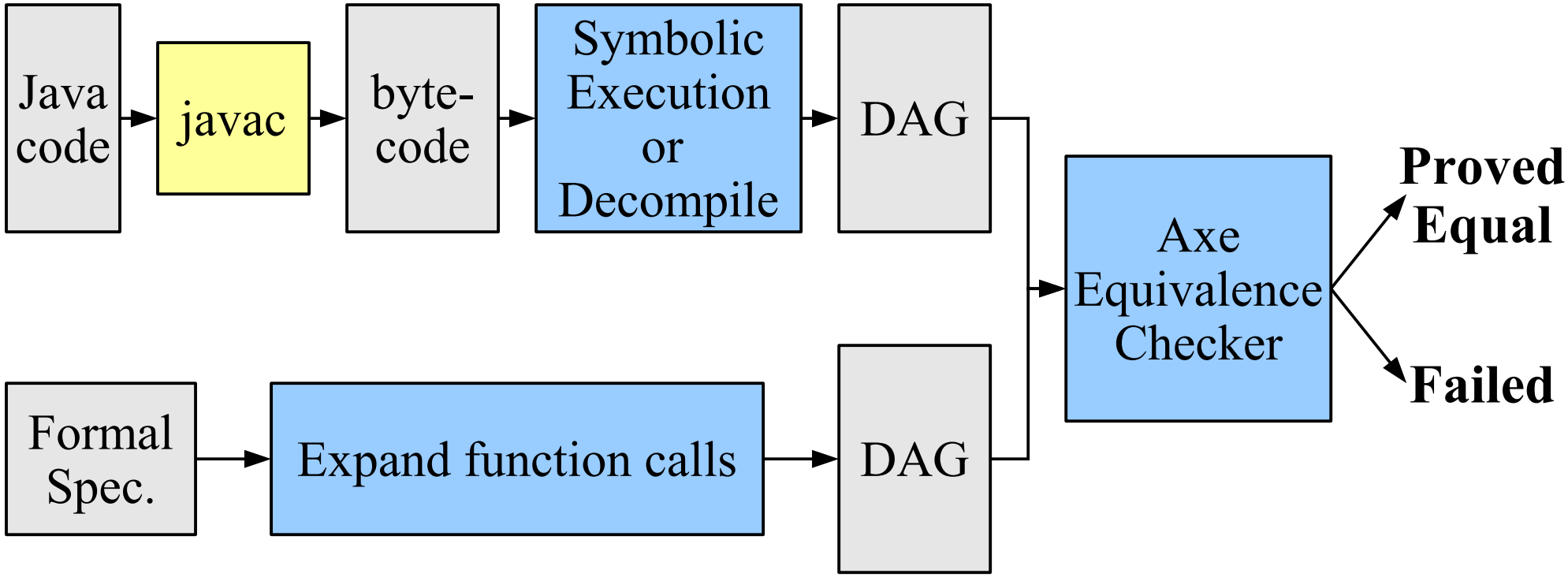
The AES algorithm is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits.

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.

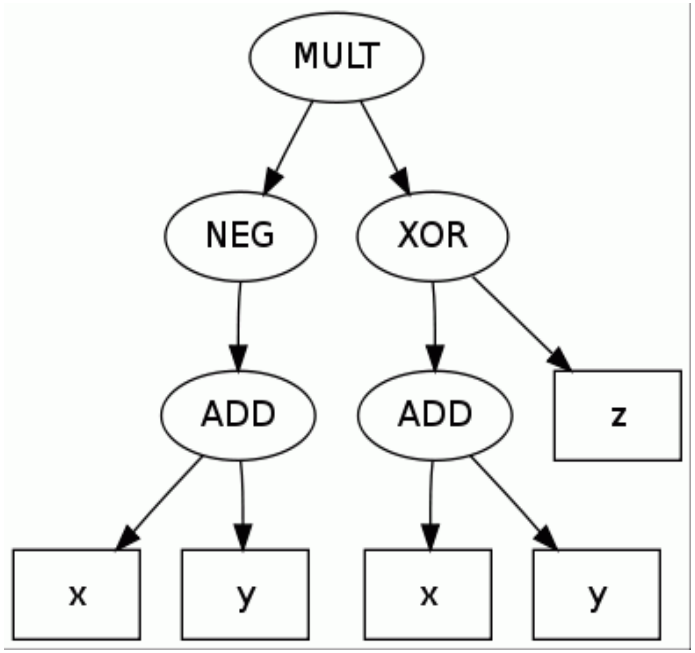
In addition, this standard may be adopted and used by non-Federal Government organizations. Such use is encouraged when it provides the desired security for commercial and private organizations.

Using Axe

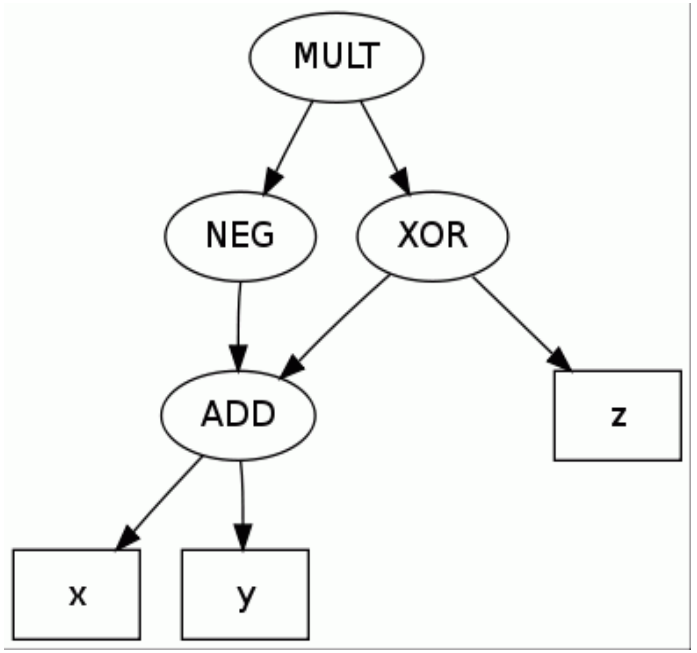

Java code
javac
byte-code
Symbolic Execution or Decompile
DAG
Formal Spec.
Expand function calls
DAG
Axe Equivalence Checker
Proved Equal
Failed

DAG Representation (Directed Acyclic Graph)

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



DAG:



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

Key Ideas

- Two approaches to loops:
  - Completely unroll
  - Do induction proofs

- 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
  - Steps through bytecode program
  - Operate on symbolic inputs.
  - Uses a formal model of the JVM (Moore et al)
  - Uses Axe Rewriter
- Result is a DAG
  - expresses output bits in terms of input bits
  - no loops
  - no method calls
- Works for many crypto. loops (ex: 10 rounds of AES-128).

DAG Equivalence Checking

- Advantage: Loops are gone.
  - No program annotations, no induction.
- Disadvantage: Resulting DAGs are large

  $\sim 10^3$ to $\sim 10^5$ nodes, but no exponential blowup

- Also unroll formal spec or second implementation.
- Then prove equivalence of the two DAGs.
  - Rewrite the equality
  - "Sweeping and merging"

Rewriting

- Repeatedly applies local simplification rules to the DAG.
  - Rules are proved as ACL2 theorems.
  - Goal is to normalize
    - Make equivalent terms more similar.
  - Uses general purpose Axe Rewriter
    - Easy to experiment with new rules.
    - Easy to turn rules on and off.

Axe Rewriter

- Conditional rewrite rules
- Free variable binding
- Syntactic control
- Rewrite objectives (strengthen or weaken)
- Outside-in and inside-out rewriting
- Memoization
- Use of contextual information
- Phases (these rules, then those rules)

Rewrite Rules

- General-purpose
  - Dozens of rules (mostly about bit vectors).
- Domain-specific, for common crypto idioms
  - Concatenation
  - Rotation (including “variable rotation”)
  - Table Lookups
  - Special handling of XORs
- Goal: Normalize different ways of doing the same thing.

Bit-blasting

- Convert multi-bit operations into concatenations of single-bit operations.
  - Ex: Blast 32-bit addition into 1-bit ANDs, ORs, XORs
- Delay as long as possible!
  - Can obscure word-level properties.
    - Commutativity / associativity of addition.
  - Makes the DAGs much bigger.

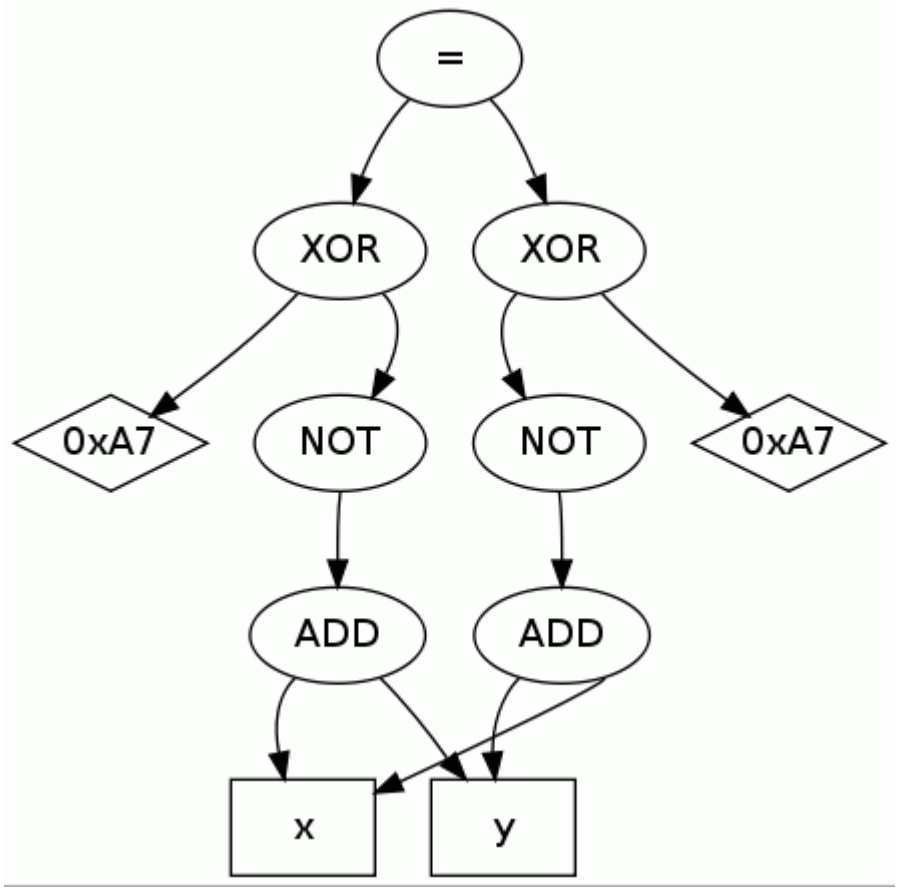
Java Code Verified by Rewriting

- bouncycastle.org: AES(unoptimized), RC2, RC6, Blowfish, Skipjack
- Sun: RC2, Blowfish.

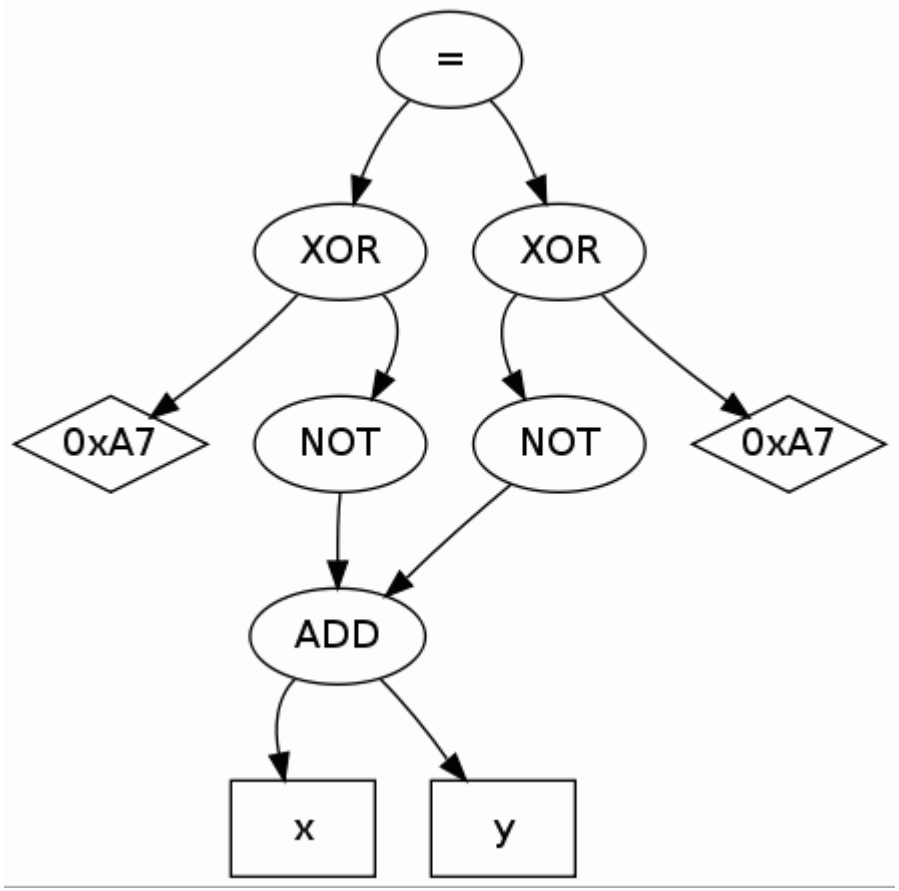
Sweeping and Merging

- Run test cases (assign values to each node).
- Find “probably-equal” pairs of nodes.
- Sweep from bottom to top, proving and merging node pairs.
- For each step, call STP (Dill and Ganesh)
  - SAT-based decision procedure for bit-vectors and arrays

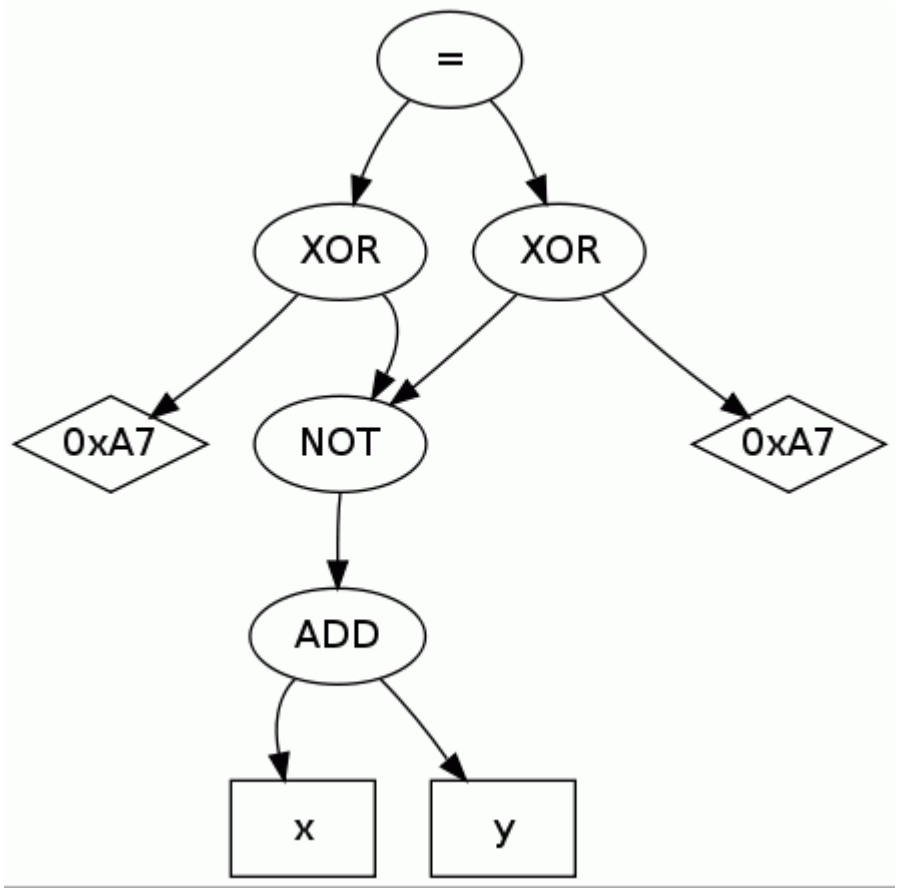
Sweeping and Merging Step 1



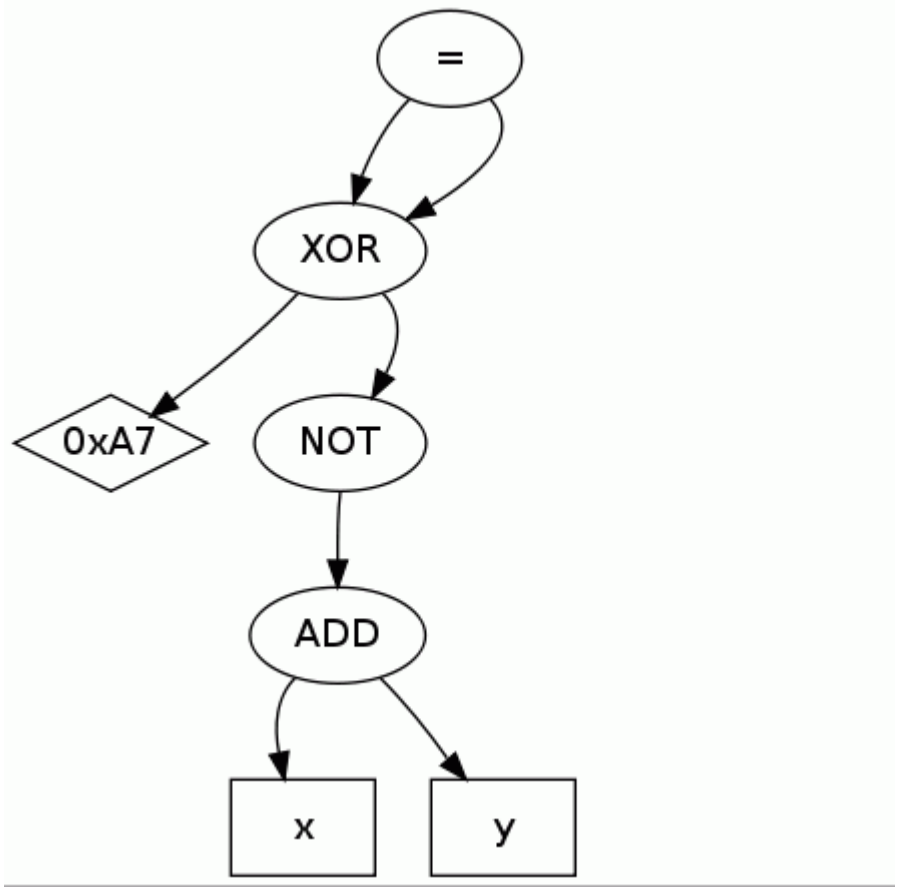
Sweeping and Merging Step 2



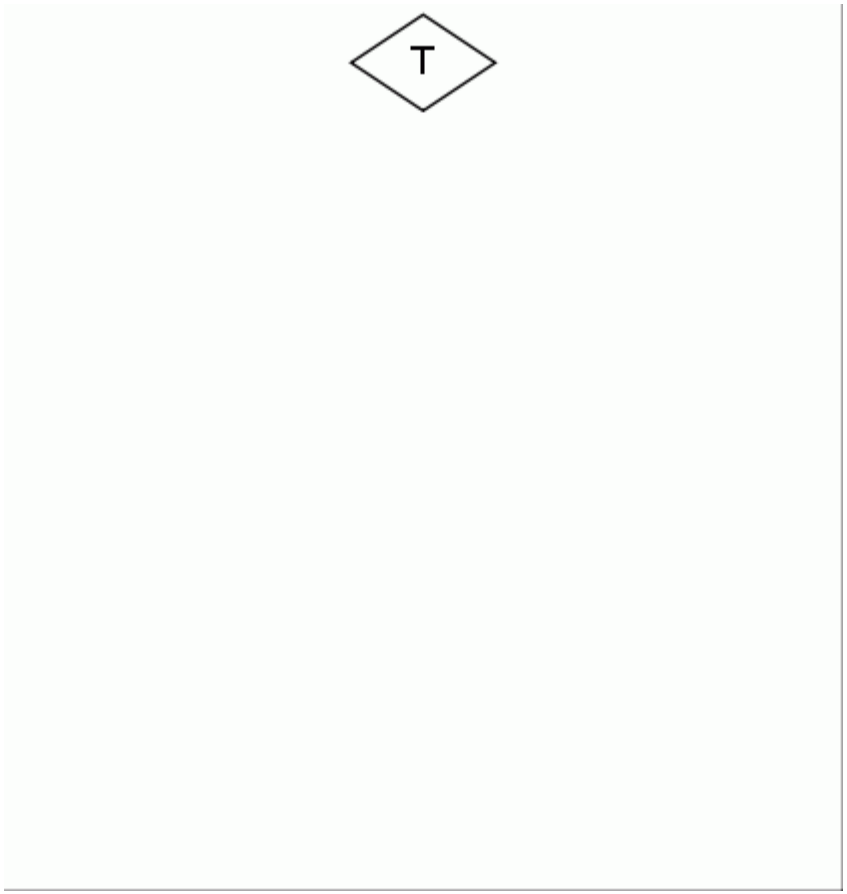
Sweeping and Merging Step 3



Sweeping and Merging Step 4



Sweeping and Merging Step 5



Sweeping and Merging

- Works well for crypto.
  - Intermediate values match up between “rounds.”

- Heuristic goal cutting
  - STP times out on huge goals.
  - So try a smaller, more general goal.
  - Heuristics to find a good cut (incremental, binary search).
  - Works well: Only the nodes for the current round are relevant.

Results for Sweeping and Merging

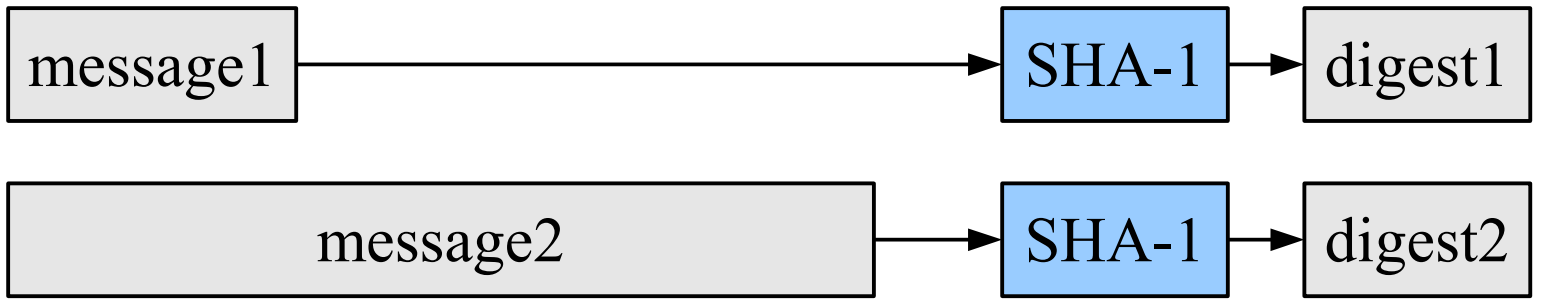
- bouncycastle.org: AES(Fast), AES(Medium), DES, 3DES
- Sun: AES, DES, 3DES

How Much Work?

- Early examples were done in parallel with tool development.
  - Hard to estimate effort.
- Skipjack took less than 3 hours, including:
  - writing and debugging the formal spec
  - doing the equivalence proof

Non-Unrollable Algorithms

- Unbounded input size
- Ex: Hash functions:



- Also, stream ciphers and block cipher “modes of operation”

Non-Unrollable Algorithms

- Hack: Fix the input size and unroll
  - 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
- Want a proof for all input sizes!

Non-unrollable Loops

- Decompile loops into recursive functions.
  - These “loop functions” appear as DAG node operators.
- Rewrite, sweep, and merge as usual
  - Loop reasoning needed to merge loop nodes.

To Merge Equivalent Loops

- Find loop connections:
  - Formulas over parameters of the two loops.
  - Always true on corresponding iterations.
  - Equalities, linear relationships, mod facts, etc.
- Detect connections from execution traces.
- Prove by induction.

- Also find invariants of individual loops
  - Bounds, types facts, mod facts, constant values, unchanged values

Loop Property Detection

- Want to find candidate properties that are
  - true
  - provable
  - useful
- Key properties not clear from static analysis
  - x1= y1 because both loops are computing AES
- So we analyze execution traces.

Example: Numeric Bounds

- Needed to show array accesses are in-bounds.
- Don't just take the largest and smallest values seen.
- Axe looks at which values are in which traces.

trace 1: i = 0, 4, 8, 12
trace 2: i = 0, 4, 8, 12
trace 3: i = 0, 4, 8, 12
...

Clearly, i is in [0, 12].

trace 1: i = 0, 1, 2, 3, 4, 5
trace 2: i = 0, 1, 2
trace 3: i = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

What is the upper bound?

If the loop always stops when i is
16 less than the input length...

...then i is in [0, len(input)-16]

Proving a Connection

- (First, prove individual loop invariants.)

- Assume:
  - the connections hold before the loop bodies
  - both loop invariants hold (already proven)
  - neither loop exits
- And prove: connections hold after loop bodies.
- By induction, the loop connections always hold.
  - Induction checked by ACL2.

Proving a Connection

- Uses the Equivalence Checker
  - Rewriting, sweeping and merging, etc.
  - Nested loops handled recursively (often unrolled)
- Some failures tolerated
  - Finds maximal provable subset of candidates.

Many Details

- Prove synchronization.
- “Old” variables
- The last iteration is special. Ex:
  - Invariant: len(array1)=len(array2) and first n elements agree
  - The loops exit when n=len(array1)
  - So, array1=array2 on exit

Main Theorem About Two Loops

- Equivalence of loops' outputs.
- Expressed as a rewrite rule.
- Used by Axe Prover to merge the loop nodes.
- Then sweeping continues.

Loop transformations

- Help synchronize the loops of two programs.
  - also can simplify individual loops
- Axe can perform and verify transformations.
- Typical operation:
  - Mechanically generate a new loop function.
  - Prove equivalence as a rewrite rule.
  - Replace old function.

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
  - For all valid inputs.
  - 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 (bit-precise) correctness results.
- Works for 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
- Other infrastructure may be useful to others
  - Also: JVM model, bytecode parser, decompiler

Future Work

- Extend Axe to other languages (C, hardware)
  - just extract computation as a DAG
  - or compile to JVM bytecode
- 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.
- Integrate with SAW, Specware.
- Apply to Android (Dalvik VM).