Automatic Formal Verification of Block Cipher Implementations

FMCAD 2008 – Portland, Oregon
Nov. 18, 2008

Eric Whitman Smith and David Dill
Stanford University
{ewsmith, dill}@cs.stanford.edu

Overview

- Cryptography is important.
  - Worth verifying if we can do easily.
- We show that block ciphers can be verified nearly automatically.
- We handle real implementations in a widely-used language (Java).

Block ciphers

- Encrypt and decrypt using a shared, secret key.
- Are the building blocks of larger systems.
- Operate on a small amount of data.
  - too many inputs for exhaustive testing (at least 2^{256} for AES)
- Many important examples:
  - AES, DES, Triple DES, Blowfish, RC6, ...
- Very carefully described.

Block Ciphers (cont.)

- Are often structured in terms of rounds.
  - Loops can be completely unrolled (10 rounds for 128-bit AES)
- Are often heavily optimized:
  - data packed into machine words
  - loops partially unrolled
  - pre-computed partial results stored in lookup tables

Inner loop of “light” AES encrypt

```
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];
}
```

What Our Approach Proves

- We don't prove that the cipher is unbreakable.
- We show the implementation matches:
  - a formal specification
  - or another implementation
- Proves bit-for-bit equivalence.
- Complicated by aggressive optimizations and differences in programming idioms.

Inputs to the verification method

- Java implementation.
- Second Java implementation or formal specification.
- Indication of how the bits match up.
- Note: No program annotations!

Java code

- Class files that implement a block cipher
  - main cipher class
  - helper classes
  - ancestor classes and interfaces
- Driver program
  - Calls the cipher in the usual way

Formal Specifications

- Are written in the language of the ACL2 theorem prover
  - side-effect-free dialect of Common Lisp
  - simple, precise semantics
- Closely match the official cipher descriptions
  - clarity over efficiency
  - unoptimized
- Are executable and so can be validated on test cases.
- Take a few hours to write and debug.
- Can be reused for each implementation.

Two-step proof approach

1. Represent the computations as large mathematical terms.
   - Common language for describing computations.
2. Prove equivalence of the two terms.

Rest of the Talk

- Terms and the term simplifier
- How to get terms from ACL2 specifications.
- How to get terms from Java byetcode.
- **How to compare terms.**

Mathematical Terms (“DAGs”)

- Are essentially operator trees.
  - Leaves are input variables (plaintext, key) or constants.
  - Each internal node applies a function to its child nodes.
- **Represent shared subterms only once.**
- Are acyclic.
  - No loops (but operators can be recursive functions).
- Can be large.
  - 220,811 nodes for Blowfish after simplification

To simplify terms

- Could write code to manipulate terms directly.
- Instead, we use:

  1. General-purpose term simplifier
     - Similar to ACL2's rewriter but handles shared subterms
  2. Simplification rules
     - ACL2 theorems

- High confidence
- Easy to add / change simplifications and turn on/off

Normalization

- Equivalent terms should have the same syntactic form.
- Crucial to the verification effort.
- Often enables further simplifications.
- Normalization and bit-blasting suffice to very several ciphers:
  - Bouncy Castle “light” AES
  - Bouncy Castle RC2
  - Bouncy Castle RC6
  - Bouncy Castle Blowfish
  - Bouncy Castle Skipjack
  - Sun RC2
  - Sun Blowfish

From specifications to terms

- The term simplifier:
  - Opens and unrolls function calls.
  - Leaves only bit-vector and array operations.
- For a recursive function call, can usually tell whether it represents the base case or inductive case.

From Java bytecode to terms

- Java has lots of complicated concepts:
  - field and method resolution
  - allocation of new heap addresses
  - static initializers of classes
  - values from the runtime constant pool
  - string interning
  - exceptions
- Want to get rid of all this complexity.
- Want an expression for the output (ciphertext) in terms of the inputs (plaintext and key).

From Java bytecode to terms (cont.)

- Symbolically execute the driver (using a model of the JVM).
- Uses the term simplifier to repeatedly step and simplify.
  - Simplification helps discharge array bounds checks.
- Amounts to unrolling all loops and inlining all method calls.
- Can extract bit-accurate results of long JVM executions (tens of thousands of instructions)
- Based on the ACL2 approach of Moore et. al. but
  - handles shared subterms.
  - handles conditional branches smartly.

Proving equivalence of terms

- Given two terms with the same input variables:
- Build an equality term (similar to a miter circuit).
- Prove the equality is true for all inputs.
- Phases:
  - Apply word-level simplifications
  - Bit-blast and simplify again
  - Perform SAT-based equivalence checking
    - run tests to find internal equivalences
    - call STP to prove them

Word-level simplification

- Couldn't just give the miter to SAT-based equivalence checker.
  - We tried STP and ABC and they ran for days.
- We found that it's crucial to simplify first.
- One should simplify before bit-blasting
  - because bit-blasting can obscure interesting structures
- Ex: Associativity / commutativity of 32-bit addition
  - clear at the word level
  - not clear after additions have been blasted into ripple-carry adders!
- We identified several crucial word-level simplifications for block ciphers.

Concatenation Example

Concatenation helps pack bytes into machine words.

Ex: To concatenate:

```
10101010
11110000
```

shift one operand and OR the results:

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

The shifts introduce zeros. We never OR two ones together.
So we could also use XOR or addition 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.

Bit rotations

- Similar to shifts, but the bits “wrap around.”
- No JVM bytecode for rotation.
- Common idiom: two shifts followed by a combination (OR, XOR, or ADD).
- “Variable rotations” are especially hard.

Variable Rotations

- Rotation amount is not a constant but depends on inputs.
- Key feature of RC6 block cipher.
- Cannot directly bit-blast to send to SAT.
  - Would need to split into cases, one for each shift amount.
  - Didn't work well for RC6.
- Want to normalize.
- Solution: introduce LEFTROTATE operator
  - Rules to recognize the common idioms
  - RC6 miter equality simplifies to TRUE

Lookup tables

- Replace sequences of logical operations, for speed.
- Appear as array subterms with constant elements.
- Lookups should 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 $(x_1 \oplus x_0)$.

Handling XORs

- XOR is associative and commutative.
- For a given set of values, there are many equivalent nested XOR trees.
- Other XOR properties:
  - $y \oplus y = 0$
  - $y \oplus 0 = y$
  - $y \oplus \text{not}(y) = 1$ (equivalently, $\text{not}(y) = 1 \oplus y$)

Normalizing XORs (cont.)

- 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.

Equivalence checking phase

- Applied if simplifications do not reduce the miter equality to TRUE.
  - Simplifications will help this phase succeed.
- Terms to be proved equivalent are large (tens of thousands of nodes).
  - Usually cannot simply hand off to STP.

Finding internal correspondences

- Run random test cases.
- Nodes that agree on all test cases are considered to be “probably equal.”
- Sweep up the DAG, proving and merging probably equal nodes
  - Very similar to SAT-sweeping / fraiging
- Breaks down the large equivalence proof down into a sequence of smaller ones.
- (We also find “probably constant” nodes.)

Finding internal correspondences (cont.)

- Works well for block ciphers
  - Typically a series of rounds.
  - Computation of the rounds may differ.
  - But implementations typically match up between rounds.
- For block ciphers, a few dozen to a few hundred test cases suffice.

Proving two nodes equal

- Call STP
  - decision procedure for bit-vectors and arrays
  - developed by Prof. Dill and Vijay Ganesh
- We avoid sending huge goals to STP.
- Cut the proofs.
  - Heuristically replace large subterms with new variables (“primary inputs”).
  - Is sound because the resulting goal is more general.

Proving the equalities

- If the cut equivalence proof fails, the nodes might actually be equivalent (known problem: false negatives).
- We try less and less aggressive cuts
  - Until STP proves one of the goals or reports a counterexample on the full formula.
- Block ciphers don't lead to many false negatives
  - A false negative is an infeasible valuation for the variables along a cut.
  - But block cipher state nodes can usually assume any combination of values.

Results

- Sun's implementation of the Java Cryptography Extension:
  - package com.sun.crypto.provider
  - Verified all ciphers
    - AES, DES, Triple DES, Blowfish, RC2
- Open source Bouncy Castle project:
  - package org.bouncycastle.crypto
  - Verified AES (3 implementations), Blowfish, DES, Triple DES, RC2, RC6, Skipjack

Results (cont.)

- Each cipher proved equivalent to a formal mathematical spec., for all inputs and all keys of the given length.
- Some proofs performed between Sun and Bouncy Castle implementations of the same cipher.
  - no formal specification required
- Found no correctness bugs.
- Increased confidence in correctness.

Results (cont.)

- For AES,
  - 4 implementations
    - Sun
    - 3 from Bouncy Castle: “light,” “regular,” and “fast”
  - 2 operations
    - encrypt and decrypt
  - 3 key lengths
    - 128, 192, and 256 bits
  - 24 (4 x 2 x 3) total proofs

Results (cont.)

- Most proofs take a few minutes to a few hours.
- Terms have tens of thousands to hundreds of thousands of nodes.

Latest 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

Cryptographic hash functions

- Take a message of essentially any length and compute a fixed-size digest (hash).
- Ex: MD5 and SHA-1
- Not directly amenable to our methods
  - Input size not fixed.
  - Loop iterations counts unknown.
- Can use our method if we fix the message length.
- Verified MD5 and SHA-1 from Bouncy Castle for 32-bit and 512-bit messages.

Related Work

- Standard approach to block cipher validation is testing.
  - NIST provides a test suite.
  - Accredited labs certify putative AES implementations.
- But there are too many inputs to test
  - at least 2^{256} for AES

Related work (cont.)

- Functional Correctness Proofs of Encryption Algorithms (Duan, Hurd, Li, Owens, Slind, Zhang)
  - Used an interactive theorem prover to prove inversion of several block ciphers specified in higher order logic.
  - Seems to require significant manual effort to guide the prover.
  - Inversion property is weak
    - Satisfied by trivial insecure cipher
    - Ignores key expansion
  - Does not verify pre-existing implementations.
    - Implemenations written in the native language of the theorem prover

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.

Related work (cont.)

- Cryptol language from Galois Connections.
  - Can be compiled down to an implementation using verified compiler transformations.
    - (Same approach might apply to the ciphers of Duan, et. al.)
  - Requires the use of the correct by construction framework.
  - Doesn't check pre-existing implementations.

Related Work (cont.)

- Formal Verification by Reverse Synthesis (Yin, Knight, Nguyen, Weimer)
  - Used a tool called Echo to verify an AES implementation.
  - Transforms the code by undoing optimizations.
  - Seems less automatic than our approach.
  - User must specify some of the transformations:
    - Must find instances of work packing.
    - Must specify the patterns encoded in lookup tables.

Related Work (cont.)

- Sean Weaver has proposed a verification method similar to our equivalence checking phase.
  - Finds probable equivalences using test cases.
  - Calls a SAT solver.
- Not published (described in slides online)
- Verified an AES implementation.
- Doesn't seem to have tried other ciphers
  - AES was among the easiest of the ones we tried.

Related Work (cont.)

- Combinational equivalence checking
  - Use of random test cases to find equalities (Berman and Trevillyan, 1989)
  - Prove equivalences bottom-up (also done by Kuehlmann)
    - SAT-sweeping / fraiging
  - BDDs
    - give equivalent computations the same representation
    - but may take exponential space and are sensitive to variable ordering
  - Our word-level simplification:
    - isn't guaranteed to normalize
    - but works well in practice

Conclusion

- We've demonstrated the feasibility of highly automated proofs of block cipher implementations.
- Strong correctness results (bit for bit equivalence)
- Minimal effort.

Future Work

- Consider languages other than Java
  - C, hardware, ...
- Handle loops without unrolling:
  - Run test cases to find probable invariants.
  - Would let us verify the hash functions for all message lengths.

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.

The End!

Example: AES encryption

- Input:
  - 128 bits of plaintext
  - 128, 192, or 256 bit key
- Output: 128 bits of ciphertext
- Described in FIPS-197 (Federal Information Processing Standard).
  - Block ciphers are usually very well described.

Simplification rule examples

```
(defthm bitand-of-0-arg1
  (equal (bitand 0 x)
         0))
```

```
(defthm bvor-of-shl-and-shr-becomes-leftrotate32-1
  (implies (and (equal 0 (bvplus 5 amt amt2))
                (unsigned-byte-p 5 amt)
                (unsigned-byte-p 5 amt2))
           (equal (bvor 32 (shl 32 x amt)
                           (shr 32 x amt2))
                  (leftrotate32 amt x))))
```

Characteristics of block cipher code

-
- Bit rotations with non-constant rotation amounts
  - Can't just bit-blast and send to STP
- Constant arrays as lookup tables
  - sequences of logical operations are replaced with table lookups
- Lots of XORs
  - SAT-based tools often handle XOR poorly

Breaking down the equivalence proof

- Repeatedly select a pair of probably equal nodes
  - Try to prove them equal (using STP).
  - If the proof succeeds, “merge” the nodes:
    - Choose a representative.
    - Change all parents of the other node to use the representative.
  - If the proof fails (the nodes weren't equal), report the failure, don't merge, and continue.
- Sweep up the term, proving and merging from the leaves to the root.
- Eventually, the top nodes of the two implementations merge and the top equality becomes TRUE.