Automating Formal Verification of Block Ciphers

Eric Smith
Stanford University
ewsmith@stanford.edu

joint work with:
David Dill
Stanford University
dill@cs.stanford.edu

Overview

- We verify Java implementations of block ciphers.
- We want as much automation as possible.
- We represent computations as DAGs, since ACL2 terms would be too big.
- We have tools to turn Java code and ACL2 code into DAGs.
- We have a tool to compare DAGs.
- DAG comparison uses STP, a decision procedure for bit vectors and arrays.

Block Ciphers

- Encrypt and decrypt using a shared, secret key.
- Operate on a small amount of data
  - ex: 128-bit input and 128-bit key
- Are the building blocks of larger systems.
- Include: AES, DES, Triple DES, RC6, Blowfish, Twofish, IDEA

What we verify

- We check that algorithms are implemented correctly.
- We don't check their cryptographic strength.
- We typically compare two implementations.
  - One “implementation” may be a formal spec.
  - We wrote formal ACL2 specs for several ciphers.
- Our proofs are stronger than “inversion” proofs.
- The job is complicated by aggressive optimizations:
  - Pre-computed tables
  - Packing into machine words
  - Optimized subroutines (ex: finite field multiplication)

Compositional Cutpoint Approach (not today's topic)

- We added annotations (esp. loop invariants) and checked them using symbolic simulation.
- We used our proof framework to combine the individual cutpoint proofs.
- We applied the approach to AES, DES, and RC6.
- Worked well but writing loop invariants took some effort.
- Had to account for data packing
  - input/output/internal formats for implementation/spec
  - Ex: packing AES state into 32-bit machine word by columns
  - Ex: DES left rotation by one bit
- Had to account for partial loop unrolling
  - Ex: Two rounds of AES per iteration

Increasing automation

- Loop invariants are sort of overkill for block ciphers.
- Most loops in block cipher code can be statically unrolled.
  - Ex: 10 rounds of encryption for 128-bit AES
- Can represent each computation in closed form (no loops or recursion).
- Doable for all the block ciphers we've studied.

Problem

- Closed form expressions for block ciphers are huge!
- Too big to express as ACL2 terms.
- Block ciphers have massive sharing of sub-terms.
  - The expression for round **`n`** of AES mentions the result of the first **`(n-1)`** rounds in several places.
- The tree for the 128-bit AES spec. would have 9,938,131,383,685,808,973 nodes (12,046 unique nodes)
- Blowfish example would have 10^17,000 nodes (110,693 unique nodes).

DAG representation

- Nodes are numbered.
  - Top node has the highest number.
- Each node is:
  - a variable, or
  - a quoted constant, or
  - a function applied to constants and lower nodes
- Essentially a huge nest of LETs.
  - (But real LETs would be expanded by ACL2.)
- Key property: A DAG can refer to a node many times without duplicating it.
- Can print DAGs and save in books, etc.

DAG for 128-bit AES encryption (formal ACL2 specification, bit-blasted)

```
((12045 ARRAYWRITE '8 '16 '0 11537 12044)
 (12044 ARRAYWRITE '8 '16 '1 11675 12043)
 (12043 ARRAYWRITE '8 '16 '2 11803 12042)

 ...537 more nodes...

 (11505 BITXOR 1150 10500)
 (11504 BIT '7 11488)
 (11503 ARRAYREAD '8 '8 11487 '(99 124 119 ...253 more
values...))

 ...11499 more nodes...

 (3 . IN3)
 (2 . IN2)
 (1 . IN1)
 (0 . IN0))
```

Proof Method

- To compare two implementations:
  - Generate a DAG for each one.
  - Compare the DAGs using our tool.

DAG rewriter

- Is similar to ACL2's rewriter but operates on DAGs.
- Only represents each term once.
- Does conditional rewriting using familiar rules from ACL2's logical world.
- Is written in ACL2.
- Uses ACL2 arrays and the “parent trick” for efficiency.

Unrolling an ACL2 Specification

- Use the DAG rewriter to open functions and unroll recursion.
- Ex: For AES, make a DAG (66 nodes) for:

```
(aes-encrypt (cons  in0 (cons  in1 ...)
                (cons key0 (cons key1 ...)))
```

- Apply the DAG rewriter to open aes-encrypt and its sub-functions.
- Result has only bit-vector and array operators.
- Result has 2,178 nodes.
- After bit-blasting (also done with DAG rewriter), the result has 12,046 nodes.

Symbolically simulating JVM bytecode

- Write a driver program in Java that calls the appropriate cipher methods (constructor, “init” method, “encrypt” method).
- Compile all the .class files and generate an “M5E” class table.
  - M5E is our version of the M5 JVM model.
- Construct an expression for an M5E state poised to execute the driver.
  - (Actually, must first execute static initializers.)
  - We can make this state fairly concrete if needed.
- Construct an expression for running the driver and extracting the output.
- Use the DAG rewriter to repeatedly simplify that expression.
- The result is a closed-form expression for the cipher's output as a function of the input.

Comparing DAGs

- We compare two DAGs with the same variables.
  - ex: The byte variables **`in0`** to **`in15`**, **`key0`** to **`key15`**.
- We prove that the top nodes of the DAGs agree, for all values of the variables.
- Easy to evaluate a DAG
  - Just go bottom-up.
- Too many cases to test
  - For AES-128: 2^128 inputs and 2^128 keys

Comparing DAGs (cont.)

- Can't just call a decision procedure.
  - STP runs a long time without finishing, even on easier problems.
- Must break up the problem.
- Find correspondences between internal nodes in the two implementations.
- Key insight: Block ciphers almost always match up at round boundaries.
  - Data layouts might differ, so we bit-blast if necessary.

DAG comparison tool

- Written in ACL2.
- Builds a DAG that equates the two implementations.
- Tries to transform it to “true”.

DAG comparison tool (cont.)

- Runs random test cases.
  - Records the value of each node for each test case.
  - Usually 40 or 80 test cases suffice for block ciphers.
- Finds “probably equal” nodes.
  - Nodes which agree each other in value for all test cases.
- Finds “probably constant” nodes.
- Top equality node should be “probably true”.
- Top nodes of the implementations should be “probably equal”.
- Nodes at round-boundaries in one implementation should be “probably equal” to analogous nodes in the other implementation.

DAG comparison tool (cont.)

- Goes bottom-up, proving the “probable” facts.
- Fixes up the DAG after a successful proof.
- Ex: If we think node 700 is always '0, prove it and then replace references to node 700 with the constant '0.
- Ex: If we think node 100 is always equal to node 200, prove it and then change all references to node 200 to point instead to node 100.

DAG comparison tool (cont.)

- The tool processes the DAG from the bottom up.
  - i.e., starting with the leaves (variables and constants)
- Eventually, the top nodes of the implementations get merged.
- So the top node becomes a trivial equality.

Doing the equality proofs

- We call STP, a decision procedure for bit-vectors and arrays.
- But sometimes large DAGs bog STP down.
  - Say we've merged the implementations up through the first 9 rounds of AES.
  - Now we want to show that the two implementations agree on the 10^{th} round.
  - We would send to STP an input which includes 9 rounds of AES
    - A lot of complicated, irrelevant computation!

Cutting the Proofs

- Replace big shared sub-DAGs with fresh variables.
- The resulting proof obligation is more general and smaller.
- If the proof goes through, great!
- If not, we move the cut down and try again.
  - Gives more and more structure to STP.
- In our experience, as soon as the cut moves below a round boundary, STP can do the proof.

Applications

- AES
  - org.bouncycastle.crypto.engines.AESLightEngine
  - org.bouncycastle.crypto.engines.AESEngine
  - org.bouncycastle.crypto.engines.AESFastEngine
  - com.sun.crypto.provider.AESCrypt
- DES (bouncycastle)
- RC6 (bouncycastle)
- Blowfish (bouncycastle)
- More to come...
- Automating the process (new build system).

Applications (cont.)

- Cryptographic hash functions:
  - MD5 (bouncycastle)
  - SHA1 (bouncycastle)
  - For these, we must fix the message length so that we can unroll the loops.
    - We did 32- and 512-bit messages.
    -
- No crypto. bugs found yet.
- Proofs and simulations take a few hours or less.

Future Work

- Finish present work!
  - Clean up and try to eliminate all manual steps.
  - Do a lot more examples.
- Apply to C code.
  - We are trying to get DAGs for a C version of AES.
- Think about how to handle iteration...
  - Use random simulation to find probable loop invariants?
- Think about how ACL2 might support DAGs.
- Study the literature
  - Some of these techniques are not new.
  - But we can work at the word level and handle array operations. How important are those things?

Related Work

- Lots of work on equivalence checking
  - BDD sweeping, SAT sweeping
  - Usually seems to work at a lower level (e.g. “and-inverter graphs”).
  - We stay at the word level when possible and apply ACL2 rewrite rules first.
  - We handle arrays (e.g., for Blowfish's key-dependent s-boxes).

Related Work (cont.)

- Standard ACL2 approach to JVM proofs (M5, M6, etc.)
  - We use a modified version of M5.
  - We use the DAG rewriter in place of ACL2's rewriter.

Related Work (cont.)

- Functional Correctness Proofs of Encryption Algorithms (Duan, Hurd, Li, Owens, Slind, Zhang)
  - Proves inversion of several block ciphers specified in higher order logic.
- Cryptol language from Galois Connections
  - Can be compiled down to an implementation using (mostly) verified compiler transformations.
  - Any other relevant work from Galois?

Related Work (cont.)

- Formal Verification of Specification Extraction (Yin, Knight, Strunk, Weimer)
  - Used a tool called Echo to verify AES.
  - Transforms the code by undoing optimizations.

Tools we have developed

- Java class file parser
  - works better than jvm2acl2
- New JVM model called “M5E”
  - based on UT's M5 model
- DAG rewriter.
- Library of bit vector functions and rules
  - Close to what STP supports.
  - Based on ihs library and similar to super-ihs.
- New build/dependency system.
- ACL2 specifications for several block ciphers.