How We Extended ACL2 to Verify Block Cipher Implementations

Eric Smith
(joint work with David Dill)
Stanford University

ACL2 2009 Rump Session
Boston, MA
May 12, 2009
Block Cipher Verification

- 2008 FMCAD paper: “Automatic Formal Verification of Block Cipher Implementations” by Eric Smith and David Dill
- Verifies JVM code for a block cipher (AES, DES, Blowfish).
- Proves equivalence to a formal ACL2 spec.
  - Bit-for-bit equivalence of outputs, for all inputs (too many to test).
- Nearly automatic
  - Verified Skipjack cipher from scratch in 3 hours.
- Couldn't use ACL2's term representation or rewriter.
  - Defined my own.
“Unroll” both computations completely.

ACL2 specification:
- Open all functions (except bit-vector and array primitives).
- Q: Why does this work for recursive functions?
- A: Because the depth of each recursion is fixed
  - e.g, 10 rounds for AES-128
- Result: A huge term with BV and array operators.

JVM code
- Symbolically simulate the whole program.
- Q: Why does this work for loops?
- A: Because the number of loop iterations is fixed
- Result: Another huge term with BV and array operators.
To prove equality of two huge terms:

- Rewrite terms using bit-vector library.
- Bit-blast and rewrite again.
  - Sufficient to verify several ciphers.
- Then use techniques from combinational equivalence checking.
  - Run test cases and find nodes that are “probably equal.”
  - Repeatedly prove two nodes equal and “merge” them.
  - Call STP (a SAT-based decision procedure for bit-vectors and arrays) to do the individual proofs.
Huge terms

- The terms being compared:
  - are completely unrolled (have only BV and array operators)
  - represent the ciphertext as a function of the inputs (plaintext and key).
  - are really complicated.
  - have massive sharing.
- Result of round $n$ appears many times in the expression for round $n+1$.
- Each round multiplies the term size.
  - Exponential explosion!
- Unrolled ACL2 specs:
  - 128-bit AES: 14033950307046 nodes (3948 unique nodes)
  - 192-bit AES: 3631268598854768 nodes (4683 unique nodes)
  - 256-bit AES: 350943360883919420 nodes (5647 unique nodes)
  - 64-bit Blowfish: ~$2^{17235}$ nodes (220817 unique nodes)
64-bit Blowfish node count =
Huge terms

- If you naively process a term, you die.
  - try to print it, evaluate it, rewrite it, get its size, get its variables, etc.
- ACL2's rewriter dies on:
  - Unrolling the spec.
  - Symbolically simulating the bytecode.
- My solution:
  - Define a representation of terms that shares subterms.
    - A subterm can have many parents.
    - These “terms” aren't trees.
    - They are directed, acyclic graphs (DAGs)
  - Define a rewriter on DAGs.
DAGs

- Nodes are numbered.
- Each node is:
  - a variable,
  - a function call (whose arguments are node numbers and/or quoted constants)
- DAGs are compact: No two nodes are identical.
- DAGS are convenient to print and read back in.

Term:

```
(foo (bar y '100) (bar y '100))
```

DAG:

```
((2 FOO 1 1)
 (1 BAR 0 '100)
 (0 . Y))
```
DAG Rewriter

- Takes a DAG and a set of rules.
- Sweeps up the DAG.
- Builds a simplified DAG.

- Can symbolically simulate JVM code.
- Can unroll ACL2 specifications.
- Can simplify the resulting DAGs.
- Useful in other tools as well.
  - (e.g., JVM bytecode decompiler)
DAG Rewriter (cont.)

- Similarities to ACL2's rewriter:
  - Applies standard ACL2 :rewrite and :definition rules.
  - Rewrites inside-out.
  - Relieves hypotheses recursively via rewriting.
  - Matches free variables from assumptions.
  - Has a version of syntaxp.
  - Has a version of bind-free.
  - Allows staged simplification
    - But implementation is different.
DAG Rewriter (cont.)

- Differences with ACL2:
  - Represents terms as DAGs.
  - No type-prescription, forward-chaining, or linear reasoning.
  - No splitting on ifs (!)
    - Either just one case or exponentially many.
    - Let SAT handle the cases.
  - User can change the order that rules are applied.
    - Better than ACL2's behavior?
  - Special purpose code for normalizing nests of XORs
    - Huge XOR nests arise in verifying block ciphers.
    - Sorting the arguments using rules is quadratic.
    - Could extend this to any associative/commutative function.
  - Easy to say “simplify this term.”
DAG Rewriter (cont.)

- Fairly efficient:
  - Uses ACL2 arrays under the hood.
  - Uses the “parent trick” to check whether a node already exists.
- Tail-recursive (when rewriting right hand sides)
  - Allows long chains of rewrites (e.g., in symbolic simulations)
- Memoizes rewrites.
- Performance:
  - Unrolls the spec. for AES-128 in ~7 seconds.
  - Symbolically executes JVM bytecode in ~14 seconds (~9900 bytecode instructions).
Bit vector rules and connection to STP

- ACL2 library of bit-vector and array operations
  - Many general-purpose rules.
    - (Some special rules for ciphers.)
  - Used in my modification of the M5 JVM model.
  - Operators mostly match STP's operators.
- Easy to translate from DAGs (with only these operators) to STP.
- Each DAG node becomes a LET.
  - (tricky to handle equality of arrays, constant arrays)
- STP uses DAGs internally.
Future Work

- Make DAG rewriter faster
  - Sobj arrays? Hashing? Honsing?
- Verify the DAG rewriter?
  - Would love to make it a verified clause processor.
  - How do we justify using rules from the ACL2 world?
  - Issues with evaluation of ground terms.
  - Do I have to fix the set of functions on which it operates?
- Modern tools seem to use DAGs (STP, ABC). Can we make ACL2 use DAGs?
  - Maybe just use hons and memoization?
  - Maybe just prevent LETs from expanding in proofs?
    - But we might still build the same term twice?
    - We'd need rewrite rules to fire despite intervening LETs.
  - Not sure how to handle a shared node that appears in many different IF contexts.
The End
Tools I've developed

- DAG rewriter.
- Supporting library of bit-vector and array functions.
- Java class file parser (written in ACL2)
- JVM model (improved version of M5)
- Formal specifications for many ciphers.
- Translator from DAGs to STP.
- JVM bytecode to ACL2 decompiler.
- ACL2 build system (written in ACL2).