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.

Block Cipher Verification (cont.)

- “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.

Block Cipher Verification (cont.)

- 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:  3509433608839194 20 nodes  (5647 unique nodes)`
  - `64-bit Blowfish:  ~2^17235  nodes (220817 unique nodes)`

64-bit Blowfish node count =

1024822625182104036460593754615016929511412475316201129628525747322369022664200327591542384423978026717371878568804620768217701833707573786933653
2106535156287280001223626379523619501717314565687530140520651527754688114063052039764239364615052791240295806546265852352209872975744469698125 4
9631589069188700607691552374320753991862969126651817843443792762362999531330095753015742721295320438456175092161158993941217791696544189173947570
5292559447772527333970487858121522203582370886714914804576046511644237941882132623600901453358142553087404053859865072472707928727331791863882980
9301225230516263546240015407182968159578081142381712922346340034294454538616104589499059462648630945309387328119586972131993598918467398284313 3
9472582698469281533962521536020902572943891617749096818097640782216783744437611375496721849526695814348367598343012705498448406142826197464298830
8100786984722453975622045706888185308758389591566295962723948166452609771352075057924583798075103412060992485868654376870829687093373045548722912
4106592189132782062983934733543328003431273225452294923469412276425960504924523118383787113608081247290559783921648516752581121589681792205615360
7143998835542822787325937307160393891122342381464836546759565734440641933393921384183273725614857097336045293812315057382405258689551968544761766
1485811117765690302843290093284716857176462882920173083702409194119506369853267503976406402067936376469566827213290491351998814619894160094739172
6397761332549276223377091412621401843800398510815931630853603148239458918160551652737737815061848634023092138295480997139730897434302993417395328
8761872375145257534911166407462873655027745863529707025163079304201209181093204500794818520900507416055126143044290274770785214052661447151046869 5
6320513681607754376483901112440334650376364164288339347933125522603932765968474162611035196019342961208909844527334138700233542087243522926053418
5575894169915931177144003540816224763153411351775317851976865606703054659269769825703248957426995134700929075624438126522423519464541889001691883 7
5312914719670730326247439202875552778808920625991905526932018870120325452522711543257921339856299278640243039227292916028122157768238389687030 12
1905986323767689612441224405266107949029362502041777463196904973965104275884398547239647060291829037896648828476663767032114746105911377690020276
6613531334372449683346287170196709585655194149757353940823009657988581400996722896502064444951501384179637197888714573538655316716809410273351786
1135330928041242068924618901203805764726457206195509628670387082670180957012761203031174523258600014854047799948575803516209373889047505044679787
4137509438104506884848252542585096839278604318396137470704092836431348849099358953105534443982622671419637494683683954704218500647904606002688 76
5178117711080455992650321334631578941538657060714464316059197205679787455027800339422522285863093993154155796195967950816238114930243591773379132
1946007403921240885154920537552306330522523949543369707623623492616668904500718011272138409479846223527283833147374954810148101200595611902182668
9352119594107802720157089577658666931336833456506638671481978326163485188192795593575613635212520708249304362450562094227696465863588863276674 16
4579998065412725507273272424714610100295724566460939162792666764538246041638148368755057815550359122509031211458727243817097684403486894532993261
9202267641843209019426161542909207219106729971264121895075500924755200255656152091103759176423926065390130477656303544039978870395336886294568 91
9299999680244835791980784498426979484184026320071045084862862143004203970467067902433421114915071876095796028140347358030143059724008818710055 68
3146515817568096388458738573357072511589973880821163747240849859985531792082670497108507411833298959307232441500620641431987450954258347700417628 4
6441140277943018862240726068250908904889989229681983084685583717547804000446161504171969918202836898154759360839597000021526701277612618110050000
6427362582924744232030292756911413347405470376526524149278208297916363806042609347061143325203633356456515204684667556970059805596991854925037 92
4407916109082660862219867848343810741122729656747438275818365782023271443703812367206595811799659116918407061030246468449232144245308955453738045
1273345004941425221116435678642654405515836663817450682242514464628554393249793230230404056226437779540066470907072658516618784712155440112571 78
9012085127644642505289965304964784861335814756564612936652598436937659158730653242693223669775175128254398041648111000488450509842165206414424551
3672290550804438539591133548732147854138592412503563310840215461543081121671674326361837950048198824579327355327145163958077589874389790477370723
7881343204471848152728312253414328478673724659447020068816861158887223728775750304876808353429148089072641051103672494064572279518176630667354522
7148192226066461700551729853174906005581525856373321371353708396176679774661433731921797009609783638190409021682760730965442688408430103600984083
2619307084840910656318192634697235696370766330398897806049589569822165083141597876977642884116174175422890336160180073208550510348317357910260540
1737551586200973130150700649844129367805429429235571695824374622862798505488678987305869608124859277201272118662409129421 65

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
  - Stobj 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).