Kestrel Institute

Kestrel Technology

# Formal Verification and Synthesis with Axe and APT

Eric W. Smith

Kestrel Institute
Kestrel Technology LLC

SRI - January 17, 2018

# Speaker: Eric Smith

- Grew up in Texas
- Went to UT-Austin and got hooked on theorem proving
  - Learned the ACL2 theorem prover
- Stanford Ph.D. in 2011
  - Developed Axe toolkit for verifying crypto code
  - Advised by David Dill
- Researcher at Kestrel Institute / Kestrel Technology since 2011
  - Leading various DARPA projects
  - Continuing to improve Axe
  - Co-developing APT toolkit for software synthesis
  - Current projects involve blockchain, assured autonomy, x86 analysis

## Kestrel Institute (KI)

- Non-profit research institute in Palo Alto
- Founded in 1981
- 12 researchers
- Basic and applied research
- Mostly government funding (DARPA, Air Force Research Lab)
- Also have a grant from the Ethereum Foundation

Research focus:

- correct-by-construction software
- formal methods and proofs
- security and assurance

Palo Alto, CA

www.kestrel.edu

# Kestrel Technology (KT)

- Small business spun out of Kestrel Institute in 2000
- Mission: to make software safer through static analysis
  - Emphasis on sound analysis
  - Based on abstract interpretation
- 5 researchers
- CodeHawk static analyzer for C, Java, and x86 binaries.

Palo Alto, CA

www.kestreltechnology.com

# Background

- Our DARPA MUSE project focused on high-assurance re-use of code found in large online repositories.
    1. Find code that seems to do what you want.
    2. Verify that it is correct.
- This talk will focus on Step 2.

- Note: Kestrel's main focus remains software synthesis.

## Talk Outline

- ACL2 theorem prover
- Formal JVM model in ACL2
- Axe toolkit
  - Axe Rewriter
  - Axe Lifter ("lifting" code into logic)
  - Axe Equivalence Checker
- APT Toolkit and Software Synthesis
- Examples
- Formal x86 model and lifter

## ACL2 Theorem Prover (github.com/acl2)

- Automated, industrial-strength theorem prover
- Similar to Coq, Isabelle, HOL
- Emphasis on scale (industry use) and efficiency
- Large community libraries (thousands of files of definitions, proofs)

## Formal JVM Model (Java Virtual Machine)

- Goal is to verify Java / JVM code
  - Compile Java code to JVM bytecode
  - Or verify JVM found in the wild (possibly obfuscated)
  - Could also handle Kotlin, Scala
- Bytecode verification means you don't have to trust the compiler
- We parse the .class file into an ACL2 constant
- Formal JVM model
  - assigns semantics to JVM bytecode
  - defined in ACL2
  - can reason about the model
  - can reason about code executing on the model
  - (also can just run code)

```
(MYDEFCONST
 |*fact_iter_down-CLASS-INFO*|
 '((:ACCESS-FLAGS :ACC_SUPER)
   (:METHODS (("main" . "([Ljava/lang/String;)V")
              (:LINE-NUMBER-TABLE (0 63)
                                  (7 64)
                                  (35 65))
              (:LOCAL-VARIABLE-TABLE (0 0 35 "args" (:ARRAY "java.lang.String"))
                                     (1 7 35 "i" :INT))
              (:MAX-LOCALS . 2)
              (:RETURN-TYPE . :VOID)
              (:PARAMETER-TYPES (:ARRAY "java.lang.String"))
              (:PROGRAM (0 :ALOAD_0)
                        (1 :ICONST_0)
                        (2 :AALOAD)
                        (3 :INVOKESTATIC "java.lang.Integer"
                         "parseInt" "(Ljava/lang/String;)I" 1)
                        (6 :ISTORE_1)
                        (7 :GETSTATIC "java.lang.System"
                         ("out" . "java.io.PrintStream")
                         NIL)
                        (10 :NEW "java.lang.StringBuilder")
                        (13 :DUP)
                        (14 :INVOKESPECIAL "java.lang.StringBuilder"
                         "<init>" "()V" 0)
                        (17 :LDC "Result: ")
                        (19 :INVOKEVIRTUAL
                         "java.lang.StringBuilder" "append"
                         "(Ljava/lang/String;)Ljava/lang/StringBuilder;"
                         1)
                        (22 :ILOAD_1)
                        (23 :INVOKESTATIC "fact_iter_down"
                         "factorialIterative" "(I)I" 1)
                        (26 :INVOKEVIRTUAL
                         "java.lang.StringBuilder" "append"
                         "(I)Ljava/lang/StringBuilder;" 1)
                        (29 :INVOKEVIRTUAL "java.lang.StringBuilder"
                         "toString" "()Ljava/lang/String;" 0)
                        (32 :INVOKEVIRTUAL "java.io.PrintStream"
                         "println" "(Ljava/lang/String;)V" 1)
                        (35 :RETURN))
              (:STATIC-FLAG . T)
              (:PUBLIC . T))
             (("factorialIterative" . "(I)I")
              (:LINE-NUMBER-TABLE (0 56)
                                  (2 57)
                                  (8 58)
                                  (12 57)
                                  (18 59))
              (:LOCAL-VARIABLE-TABLE (0 0 19 "n" :INT)
                                     (1 2 19 "acc" :INT)
                                     (2 4 17 "i" :INT))
              (:MAX-LOCALS . 3)
              (:RETURN-TYPE . :INT)
              (:PARAMETER-TYPES :INT)
              (:PROGRAM (0 :ICONST_1)
                        (1 :ISTORE_1)
                        (2 :ILOAD_0)
                                 ...
```

# Formal JVM Model

- Formalizes the JVM state:
  - heap
  - storage for static fields
  - thread info, including call stacks
    - call stack frame = {program counter, local variables, operand stack, ...}
  - code for loaded classes
  - monitor table (for synchronization)
  - table of interned strings
  - map from class names to java.lang.Class objects
- Formalizes each instruction
  - Shows the effect of each instruction on the JVM state
  - Models almost every JVM bytecode instruction (~200)

# Example (IADD instruction)

```
(defun execute-IADD (th s)
 (modify th s
        :pc (+ 1 (pc (top-frame th s)))
        :stack (push (bvplus 32 (top (stack (top-frame th s)))
                                (top (pop (stack (top-frame th s))))
                     (pop (pop (stack (top-frame th s)))))))
```

- Similar models of loads, stores, stack operations (push, pop, swap, push constant), array ops, casts, conversions, comparisons, jumps, branches, calls, returns, exceptions, synchronization, etc.
- Some instructions much more complicated (e.g., INVOKEVIRTUAL).

# Formal JVM Model

- Step function: Fetch next instruction and dispatch on the opcode (big case split)
- Run functions: Repeatedly step
  - "run n steps"
  - "run until return"
  - "run until exit code region"

- Many details covered: class initialization, string interning, etc.
- Greatly extends the M5 model (in ACL2 community libraries)
- Does not yet fully support:
  - native methods, class loading, Unicode, multi-threaded memory accesses, floating point, invokedynamic

## Lifting Into Logic
## (aka Decompilation Into Logic)
## (aka Specification Extraction)

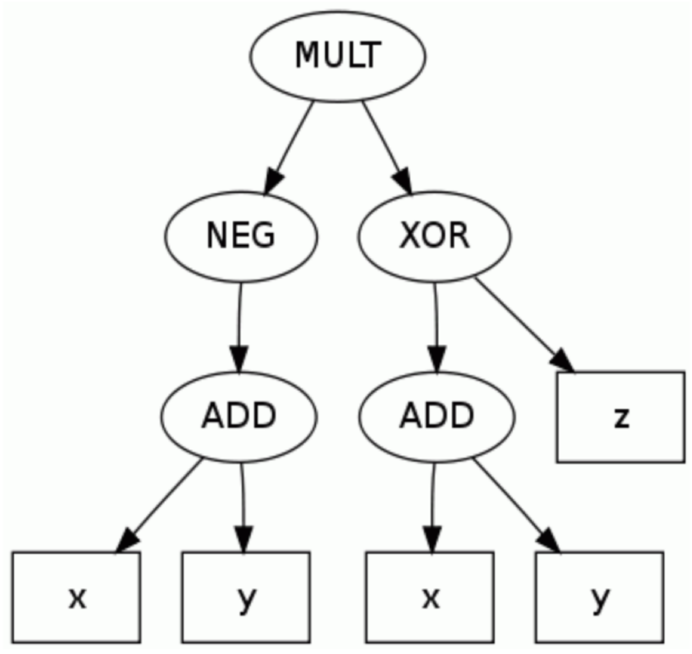
# Lifting Into Logic

- Given the code and the model,
- Obtain a function in the ACL2 logic that represents the code
  - Result should not mention the machine model
- Lifting puts the code into a clear, logical form we can manipulate.
- Related work:
  - Magnus Myreen (implemented in HOL prover)
  - CodeWalker (UT-Austin)
- Key issue is how to handle loops:
  - Easy approach (when possible): unroll loops
  - Harder approach (but more general): turn loops into recursive functions
- Approach: Use a rewriter to perform symbolic execution
  - Standard approach in ACL2 community but Axe Rewriter is novel

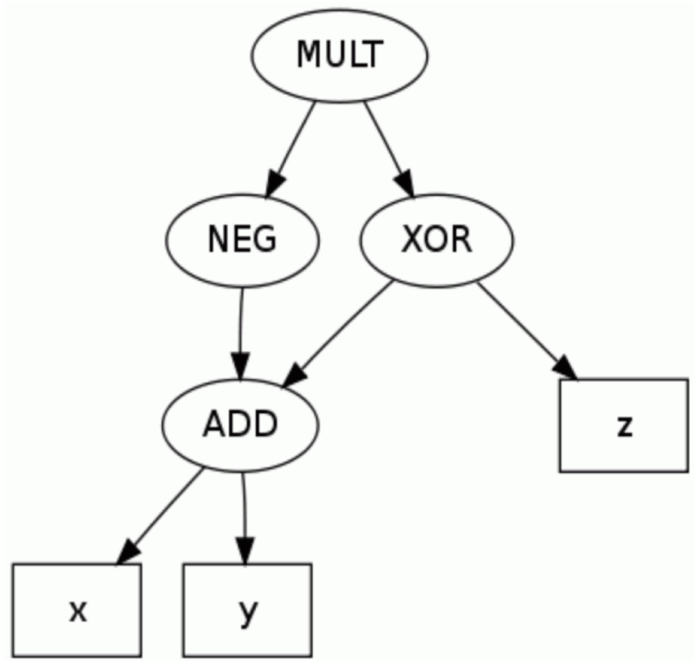
# Axe Rewriter

- Represents terms as DAGs (not trees). with structure sharing
  - no two nodes represent the same term
  - critical for crypto (tree representation blows up)

Tree:



DAG:



## Axe Rewriter

- Applies ACL2 theorems (equalities) as simplification rules
- High performance: ~600,000 rule attempts per second
- Supports:
  - conditional rewriting
  - free variable binding using assumptions
  - syntactic control (like ACL2's syntaxp and bind-free features)
  - rewrite objectives
  - memoization
  - inside-out rewriting and some support for outside-in-rewriting
  - use of contextual information (tricky)
  - phased rewriting

# Lifting Code into Logic
# Easier Version: Unrollable Loops

- Use the Axe Rewriter to perform symbolic execution:
  1. Start with a symbolic state
  2. Add assumptions
     - About the code and program counter
     - About the inputs
  3. Form a "run-until-return" expression
  4. Apply rewrite rules and unfold definitions to repeatedly step and simplify the expression
     - *run* becomes *step* followed by *run*
     - expand *step* (dispatch based on current instruction)
     - simplify aggressively using hundreds of rules
- Result is a symbolic representation of the final state as a function of the initial state
  - return value
  - final state of the heap
  - final values of static variables
- Handle conditional branches

## Example: AES Encryption (128-bit key)

```
(unroll-java-code *aes-128-encrypt-light*
                  "AESEncryptLightDriver.driver([B[B[B)[B"
                  :output (:array-local 2)
                  :array-length-alist `((key . 16)
                                        (in . 16)
                                        (out . 16))
                  ...)
```

- Result has only bit-vector and array operations
  - All JVM details gone (states, steps, stacks, jumps, etc.)
- All loops have been unrolled (e.g., 10 rounds in main loop)
- All subroutines "inlined"
  - Though Axe now supports *compositional lifting*
- Lifted result has 47,898,065,689,733,522 nodes !
- but only 3,646 unique nodes in DAG form

# Verifying the lifted code

- Prove equivalent to a formal specification
  - We have ACL2 specs for many ciphers, hash functions, etc.
  - Use Axe Rewriter to unroll all recursion in spec and inline all non-built-in functions
  - Then call the Axe Equivalence Checker
- Or prove equivalent to a golden implementation
  - Unroll both
  - Call Axe Equivalence Checker
- We prove functional (bit-for-bit) equivalence
  - For all plaintexts and keys, the two versions produce the same ciphertext
  - (You could never test this exhaustively)

## Axe Equivalence Checker (my Ph.D. thesis)

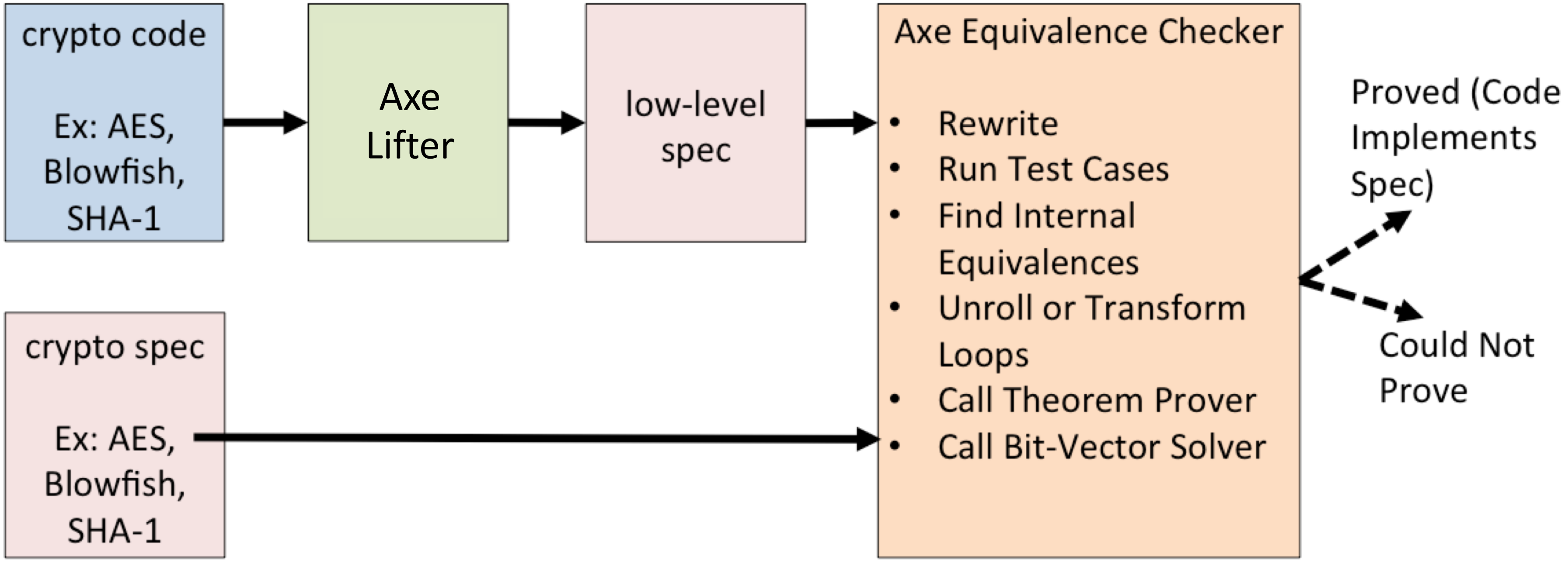
- Performs a highly-automated comparison of two logical terms (DAGs)
  - e.g., unrolled spec and unrolled code

1. Form the equality of the two DAGs
2. Call the Axe Rewriter to simplify/normalize
   - Library of hundreds of general simplification rules
   - Special-purpose rules to handle lookup tables, bit vector rotations
   - Special handling of XOR nests
   - Rules for bit-blasting (but first simplify at the word level)
3. Break down problem into smaller problems ("Sweeping and merging")
   1. Run random tests
   2. identify likely equivalent subexpressions
   3. prove and merge them one by one, bottom up
   - Uses SMT solver (STP) for bit vectors and arrays
   - Heuristically cuts SMT goals for speed

- New tactic-based checker: pruning, case splitting, etc.

# Verification with Axe

- Can automatically prove equivalence of crypto code and specs.



# Crypto Implementations Verified
## (from bouncycastle, Sun, GNU, etc.)

Block ciphers:

- AES (128/192/256 bit keys, several levels of optimization)
- DES and Triple DES
- Blowfish (64-bit and 448-bit keys)
- Skipjack
- RC6 (128/192/256 bit keys)
- RC2 (64-bit key)
- TEA

Stream ciphers:

- RC4 stream cipher (2048-bit key, 4096-bit message)

Cryptographic hash functions:

- SHA-1 (32/512/4096-bit messages)
- SHA-224, SHA-256, SHA-384, SHA-512 (various message sizes)
- MD5 (32/512/4096-bit messages)
- RIPEMD-160 (32/256-bit messages)

Caveats: Assumed static data not tampered with, assumed hash functions are given all data at once

# Lifting Code into Logic
# Harder Version: Non-unrollable Loops

- The Axe Loop Lifter can lift loops into ACL2 functions
- Basic idea: Symbolically execute the loop body and wrap it up into a tail-recursive function
- Repeatedly step and simplify the loop body.
  - Use formal JVM model and Axe rewriter, as above
- Result is an *if-then-else* nest of states.
  - Branches that exit the loop tell us the *exit test*.
  - Branches that return to the loop top tell us the *update function*.
  - Loop becomes a recursive function:

```
(defun loopXXX (params)
   (if (exit-test params)
       params
       (loopXXX (update params))))
```

- The params are the state components (scalars and arrays) touched by the loop: local variables, instance fields, static fields.
  - Cannot yet lift loops that touch an unbounded amount of state

# Simple Example (Sum of Squares)

```
((0 :ICONST_0)
 (1 :ISTORE_1)
 (2 :ILOAD_0)
 (3 :ISTORE_2)
 (4 :ILOAD_0)
 (5 :ILOAD_0)
 (6 :IMUL)
 (7 :ISTORE_3)
 (8 :ILOAD_2)
 (9 :IFLE 21)
 (12 :ILOAD_1)
 (13 :ILOAD_3)
 (14 :IADD)
 (15 :ISTORE_1)
 (16 :ILOAD_3)
 (17 :ICONST_1)
 (18 :ILOAD_2)
 (19 :ISUB)
 (20 :ILOAD_2)
 (21 :ISUB)
 (22 :IADD)
 (23 :ISTORE_3)
 (24 :IINC 2 -1 3)
 (27 :GOTO -19)
 (30 :ILOAD_1)
 (31 :IRETURN))
```

bytecode (in our parsed form)

$\xrightarrow{\text{lift}}$

```
(DEFUN SUMSQUARES-JAVA-LOOP-1$1 (SQI I ACC N)
  (IF (BOOLOR (SBVLT 32 N I)
              (NOT (SBVLT 32 0 I)))
      (CONS SQI (CONS I (CONS ACC (CONS N NIL))))
      (SUMSQUARES-JAVA-LOOP-1$1
       (BVPLUS 32 1 (BVPLUS 32 I (BVPLUS 32 I SQI)))
       (BVPLUS 32 1 I)
       (BVPLUS 32 ACC SQI)
      N)))
```

# Loop lifting

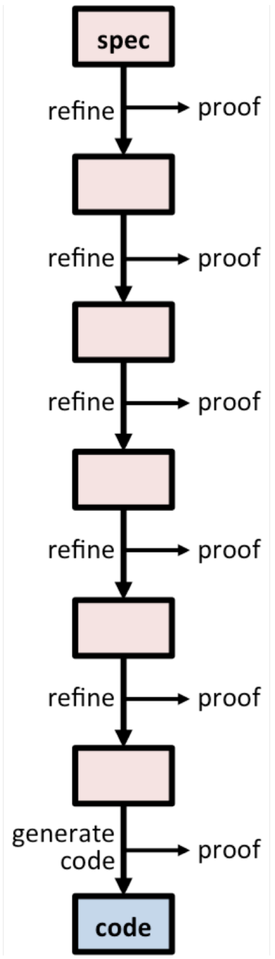
- May need to generate and check loop invariants:
  - need to know the classes of objects on which methods are called
  - need to show that exceptions do not occur
- Invariants:
  - classes of objects
  - facts about pointers not changing
  - numeric invariants
- Axe's approach:
  1. Guess invariants.
  2. Lift the loop body, assuming the invariants.
  3. Try to prove that the invariants still hold.
  4. Discard any failed invariants and repeat.
  5. (Nested loops handled inside-out.)
- Help from the user is often needed
  - Invariants and loop properties to prove inductively
  - Invariants found by static analysis can help.
- Termination: Can be proved or assumed.

# Verifying the Lifted Code

- Use APT transformations to create a *derivation* that connects lifted code and spec
  - APT = Automated Program Transformations
  - APT is a major Kestrel project
- Or (older approach): Use Axe Equivalence Checker
  - Contains transformations to synchronize loops (based on patterns observed in testing)
  - Can prove loops equivalent by induction
  - Has been used to verify RC4, MD5, SHA-1
  - Requires user hints, can be very slow

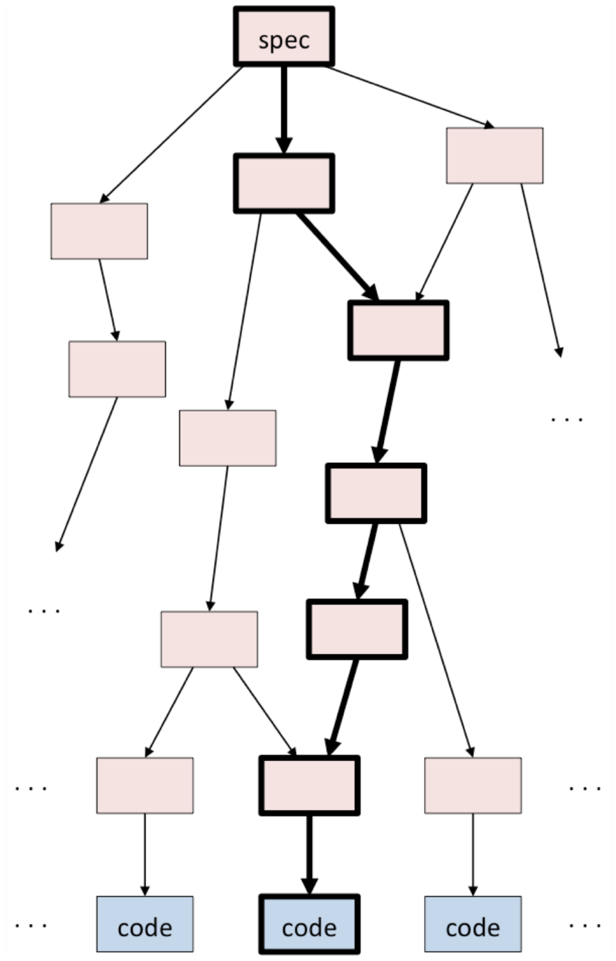
# Background: Software Synthesis by Stepwise Refinement

- Specification ("spec"):
  - written in logic
  - formalizes requirements
  - does not constrain *how* to do it
- Each refinement:
  - makes a single design choice (algorithm, optimization)
  - narrows down space of possible implementations.
  - may be carried out by an automated transformation.
- Each refinement step is proved.
  - code is correct-by-construction
- Finally, generate code from the lowest-level spec.



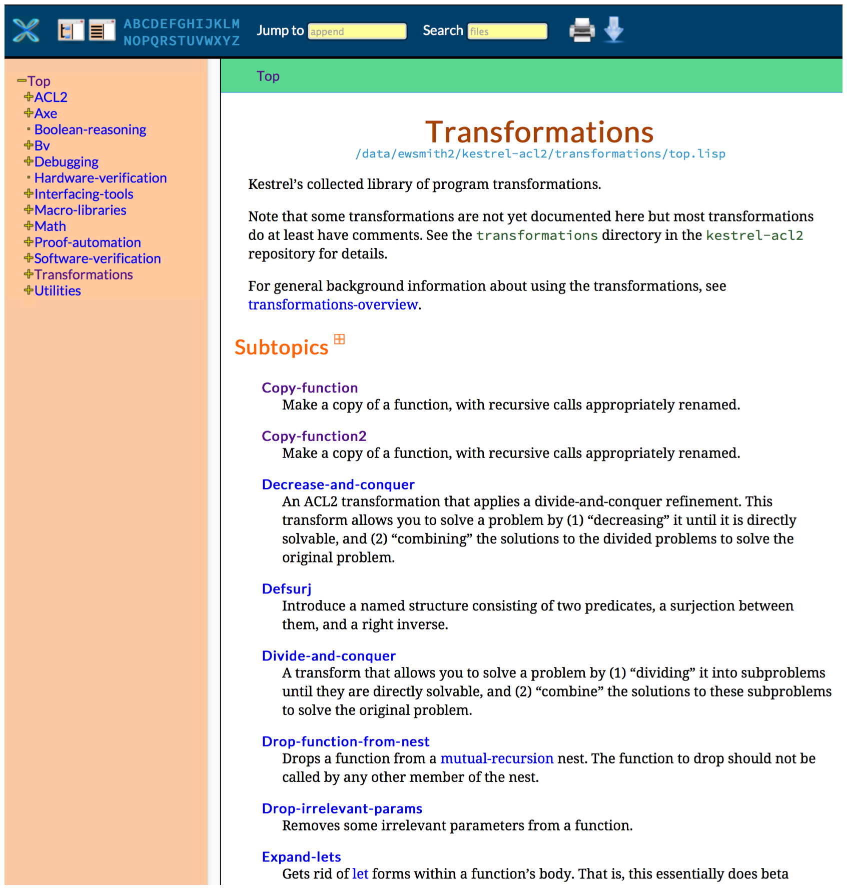
# Alternate Derivations

- A spec has many implementations:
  - different algorithms
  - different data structures
  - different optimizations
  - different behaviors when underspecified
- Refinement tree
  - (actually a DAG)
- A derivation is one path in the tree.



# APT Toolkit

- APT = "**A**utomated **P**rogram **T**ransformations"
- Software synthesis system built on the ACL2 theorem prover
- Includes automated transformations that perform refinement steps
- Each synthesis step produces a proof checked by the prover



## Transformations

/data/ewsmith2/kestrel-acl2/transformations/top.lisp

Kestrel's collected library of program transformations.

Note that some transformations are not yet documented here but most transformations do at least have comments. See the `transformations` directory in the `kestrel-acl2` repository for details.

For general background information about using the transformations, see transformations-overview.

### Subtopics

**Copy-function**
Make a copy of a function, with recursive calls appropriately renamed.

**Copy-function2**
Make a copy of a function, with recursive calls appropriately renamed.

**Decrease-and-conquer**
An ACL2 transformation that applies a divide-and-conquer refinement. This transform allows you to solve a problem by (1) "decreasing" it until it is directly solvable, and (2) "combining" the solutions to the divided problems to solve the original problem.

**Defsurj**
Introduce a named structure consisting of two predicates, a surjection between them, and a right inverse.

**Divide-and-conquer**
A transform that allows you to solve a problem by (1) "dividing" it into subproblems until they are directly solvable, and (2) "combine" the solutions to these subproblems to solve the original problem.

**Drop-function-from-nest**
Drops a function from a mutual-recursion nest. The function to drop should not be called by any other member of the nest.

**Drop-irrelevant-params**
Removes some irrelevant parameters from a function.

**Expand-lets**
Gets rid of let forms within a function's body. That is, this essentially does beta

# APT Transformations – All Proof-Producing

- **make-tail-rec**
  - assoc
  - monoid
  - nats counting down
  - bit vectors counting down
  - list builder
- **finite-difference**
- remove-cdring
- expand-lets
- letify
- simplify-body
- **simplify**
  - function
  - second-order function
  - quantified function
- rewrite-body
- drop-function-from-nest
- drop-irrelevant-params
- flatten-params
- homogenize-tail-rec
- flip-if
- rename-params
- reorder-params
- restructure-elseif
- make-do-while
- lift-condition
- weaken
- strengthen
- wrap-output
- wrap-input
- undo-finite-difference
- undo-make-tail-rec
- define-op
- copy-spec
- spec substitution
- decrease-and-conquer
- **divide-and-conquer**
- predicate narrowing
- **isomorphic data transformation**
- extract output
- restore multiple values
- expand data representation (in progress)

# Proof-Producing Transformations

- Each transformation generates a compound ACL2 event
  - New function or new spec
  - Proof (equivalence, refinement / implication, etc.)

- Proof is very tightly controlled
  - Split out lemmas and explicitly instantiate
  - Explicit induction scheme
  - Disable prover automation (generalize, destructor elim, other inductions)
  - Turn off all extraneous rules

# A tiny derivation: Sum of squares

```
;; Specification (executable but inefficient) for
;; unbounded integers:
(defun sum-squares (n)
  (if (zp n)
      0
    (+ (* n n) (sum-squares (- n 1)))))

;; Make the function tail-recursive:
(make-tail-rec sum-squares :domain natp)

;; Apply finite-differencing to incrementally
;; compute the square:
(finite-difference sum-squares$1 (* n n) *math-rules*)
```

# Make-tail-rec Transformation

```
(defun sum-squares (n)
  (if (zp n)
      0
    (+ (* n n) (sum-squares (- n 1)))))
```

```
(make-tail-rec sum-squares :domain natp)
```

```
(DEFUN SUM-SQUARES$1 (N ACC)
  (IF (ZP N)
      ACC
    (SUM-SQUARES$1 (- N 1) (+ ACC (* N N)))))

(DEFTHM SUM-SQUARES-BECOMES-SUM-SQUARES$1
   (EQUAL (SUM-SQUARES N)
          (SUM-SQUARES$1 N 0)))
```

```
(DEFTHM SUM-SQUARES-BECOMES-SUM-SQUARES$1
   (EQUAL (SUM-SQUARES N)
          (SUM-SQUARES$1 N 0)))
```

- APT generates:
  - new function
  - theorem
  - ~120 lines of proof, 14 lemmas
    - includes 2 critical inductive lemmas
  - takes 0.01 seconds

- ACL2 checks:
  - New function well-formed and equivalent to old
  - takes 0.07 seconds

# Finite-differencing Transformation

```
(DEFUN SUM-SQUARES$1 (N ACC)
  (IF (ZP N)
      ACC
    (SUM-SQUARES$1 (- N 1) (+ ACC (* N N)))))
```

```
(finite-difference sum-squares$1 (* n n) *math-rules*)
```

```
(DEFUN SUM-SQUARES$2 (N ACC NSQUARED)
   (IF (ZP N)
       ACC
     (SUM-SQUARES$2 (- N 1)
                    (+ ACC NSQUARED)
                    (+ NSQUARED (- N) (- N) 1))))

(DEFTHM SUM-SQUARES$1-BECOMES-SUM-SQUARES$2
  (EQUAL (SUM-SQUARES$1 N ACC)
         (SUM-SQUARES$2 N ACC (* N N))))
```

```
Maintain invariant for nsquared:
   (* (- n 1) (- n 1))
 = (+ (* n n) (- n) (- n) (* -1 -1))
 = (+ (* n n) (- n) (- n) 1)
 = (+ nsquared (- n) (- n) 1)
```

# Simplification Transformation

- Apply simplification rules to a function.
- More precisely, apply the ACL2 simplifier to the body of a function, and generate an equivalent function with the resulting body.
  - Support patterns to indicate that only certain parts of the body must be simplified.
- This transformation is useful to:
  - Optimize functions.
  - Carry out rewriting transformations via specific sets of rules, e.g. turn unbounded integer operations into two's complement integer operations, under suitable no-wrap-around conditions.
  - Propagate refinements from callees to callers.
  - Chain sequences of equivalence refinements.
- See the paper "A Versatile, Sound Tool for Simplifying Definitions" by A. Coglio, M. Kaufmann, and E. Smith in the upcoming ACL2-2017 Workshop. (Available at http://www.kestrel.edu/~coglio.)

# Algorithm Schemes

- Formalize abstract algorithms
  - Problem spec: Collection of functions with constraints
  - Solver spec: Provably solve the abstract problem, using the constraints
- Later, instantiate to solve particular problem
  - Map concrete problem to abstract problem
  - Prove the concrete constraints
  - Get the concrete solver (and proof) ‘for free’.
- Example: Divide and Conquer

```
(spec divide-and-conquer-problem
 (op problemp (x))
 (op solutionp (solution problem))
 (op directly-solvablep (problem))
 (op solve-directly (problem))
 (op divide (problem) :output (mv * *))
 (op combine (solution1 solution2))

 ...

 (axiom solve-directly-correct
   (implies (and (problemp p)
                 (directly-solvablep p))
            (solutionp (solve-directly p) p)))

 (axiom combine-correct
   (implies (and (problemp p)
                 (not (directly-solvablep p))
                 (solutionp solution0 (mv-nth '0 (divide p)))
                 (solutionp solution1 (mv-nth '1 (divide p))))
            (solutionp (combine solution0 solution1) p)))
  ...)
```

# Divide and Conquer

| Abstract Scheme | MergeSort | QuickSort |
|---|---|---|
| directly-solvable? | len < 2 | len < 2 |
| solve-directly | (do nothing) | (do nothing) |
| divide | **split list** | **partition** |
| combine | **merge** | **append** |
| measure | len | len |

- Other Schemes:
  - Decrease and conquer
  - Global search
  - Local search
  - Dynamic Programming
  - Fixpoint, ...

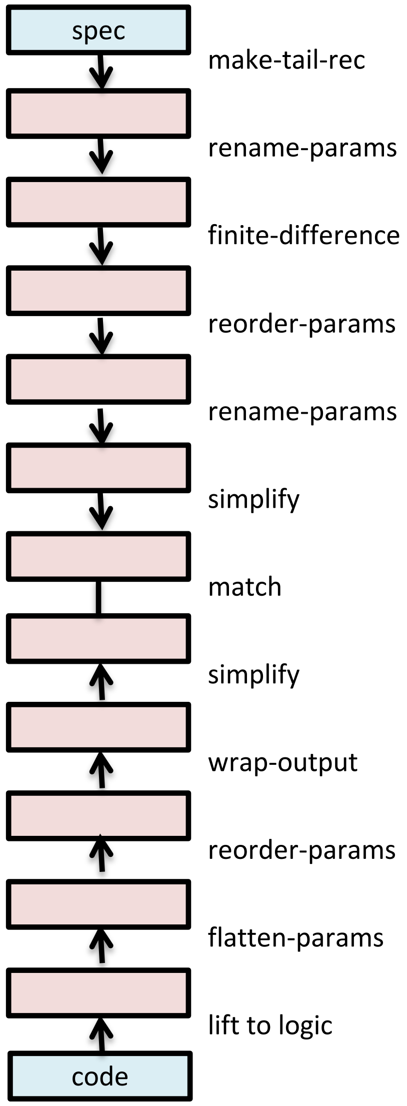
## How Can We Use Synthesis Tools to Verify Existing Code?

## Reconstructing A Derivation

- Our hypothesis: If the code is correct, there exists a derivation from the spec to the code.
  - even if the programmer did not use software synthesis tools !
- To fully understand the code, reconstruct its derivation
  - Code will then be proved correct, because each refinement step is proved.
- **How can we reconstruct a derivation?**

# Analysis-By-Synthesis: Find a Derivation

- The derivation:
  - connects existing code to a spec
  - is a sequence of provably correct refinement steps
  - contains top-down and bottom-up steps
  - represents a synthesis process that *could have been done* from the spec to the code, exposing the implementation decisions

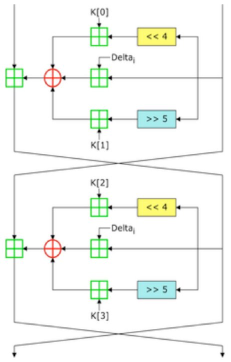


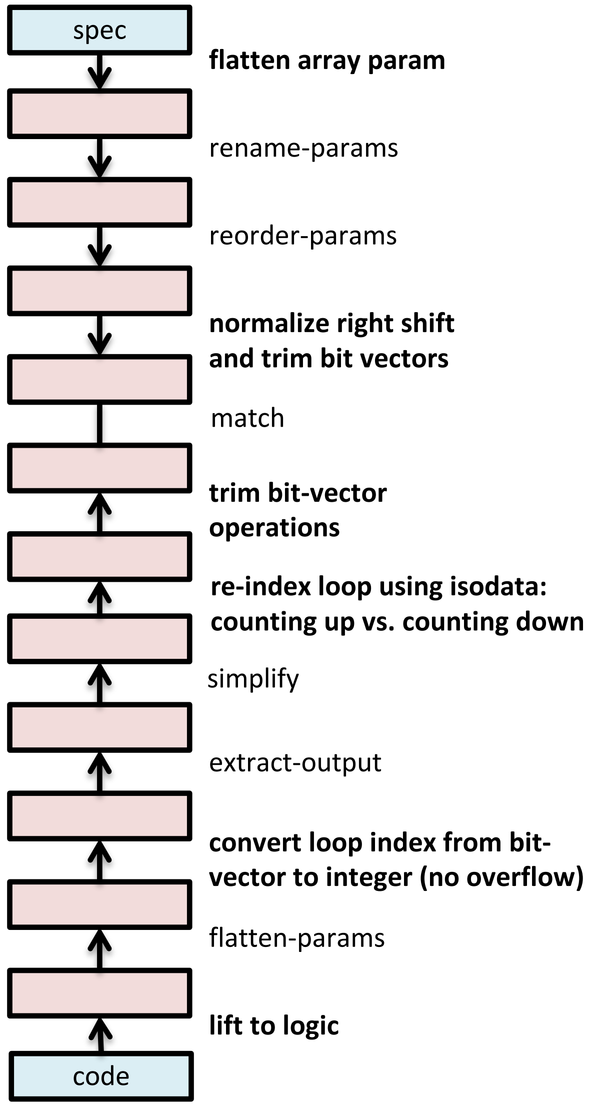
## Anatomy of an Analysis-by-Synthesis Derivation

- Write / obtain spec
- Obtain likely implementation (e.g., from a Big Code corpus)
- Lift code into logic
- Boilerplate bottom-up steps:
  - flatten params
  - extract outputs of interest
- Top-down steps:
  - apply algorithm theory (if spec is not executable)
  - (data type refinements)
  - make spec tail recursive (often)
  - (partial evaluation)
- Derivation-specific steps:
  - finite-difference, undo-finite-difference, restructure conditionals, handle optimizations, generalize, re-index loops, apply data isomorphisms, add necessary condition pre-filter, rename / reorder params
- Simplify (throughout)
- Convert bit-vector operations to integer (often) or vice versa
- Convert array operations to list operations (often) or vice versa

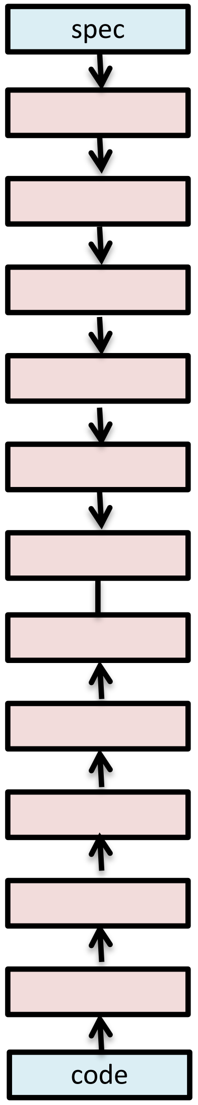
## Example: The TEA Block Cipher

- Tiny Encryption Algorithm (TEA)
- Found an implementation (javabeanz)
- Lifted it into logic
- Created a **derivation** of proven steps linking the spec and the code
  - 11 proof-producing steps
  - shows that the code and spec have the same output bits, for all inputs




spec
flatten array param
rename-params
reorder-params
normalize right shift and trim bit vectors
match
trim bit-vector operations
re-index loop using isodata: counting up vs. counting down
simplify
extract-output
convert loop index from bit-vector to integer (no overflow)
flatten-params
lift to logic
code

# Example: Legacy MAVLink CRC Checksum (x.25)



Major steps (informed by the corpus code):

- Un-Distribute polynomial remainder over XOR

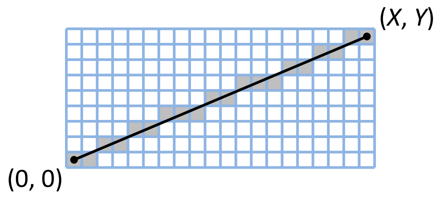
  rem(x, poly) xor rem(y, poly) → rem(x xor y, poly)
- Thread through 16-bit CRC accumulator
  - Calculate the update of the CRC
- Use 0-padding to simplify base case
- Distribute remainder over concatenate

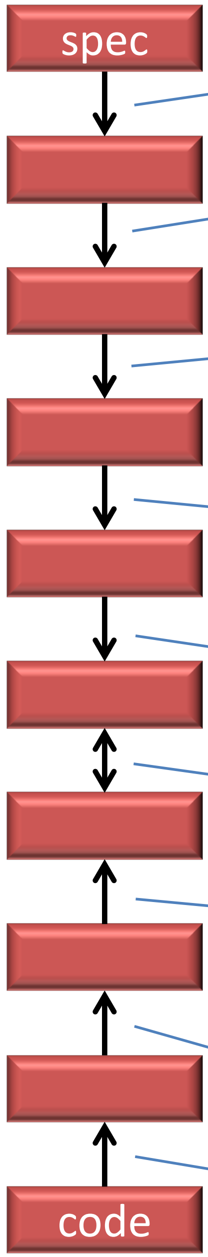
  rem'(x concat y, crc) = rem'(y, rem'(x, crc))
- Move complement (xor with FFFF) into initial CRC
- Unroll to process 8 bits at a time (and simplify!)
- Introduce bit vectors (isodata)
- Reverse the CRC to match the code (isodata)
- Normalize bit-vector operations

} Generate and verify 21 transformed versions (steps)

MAVLINK
MICRO AIR VEHICLE COMMUNICATION PROTOCOL

# Example: Bresenham's Algorithm

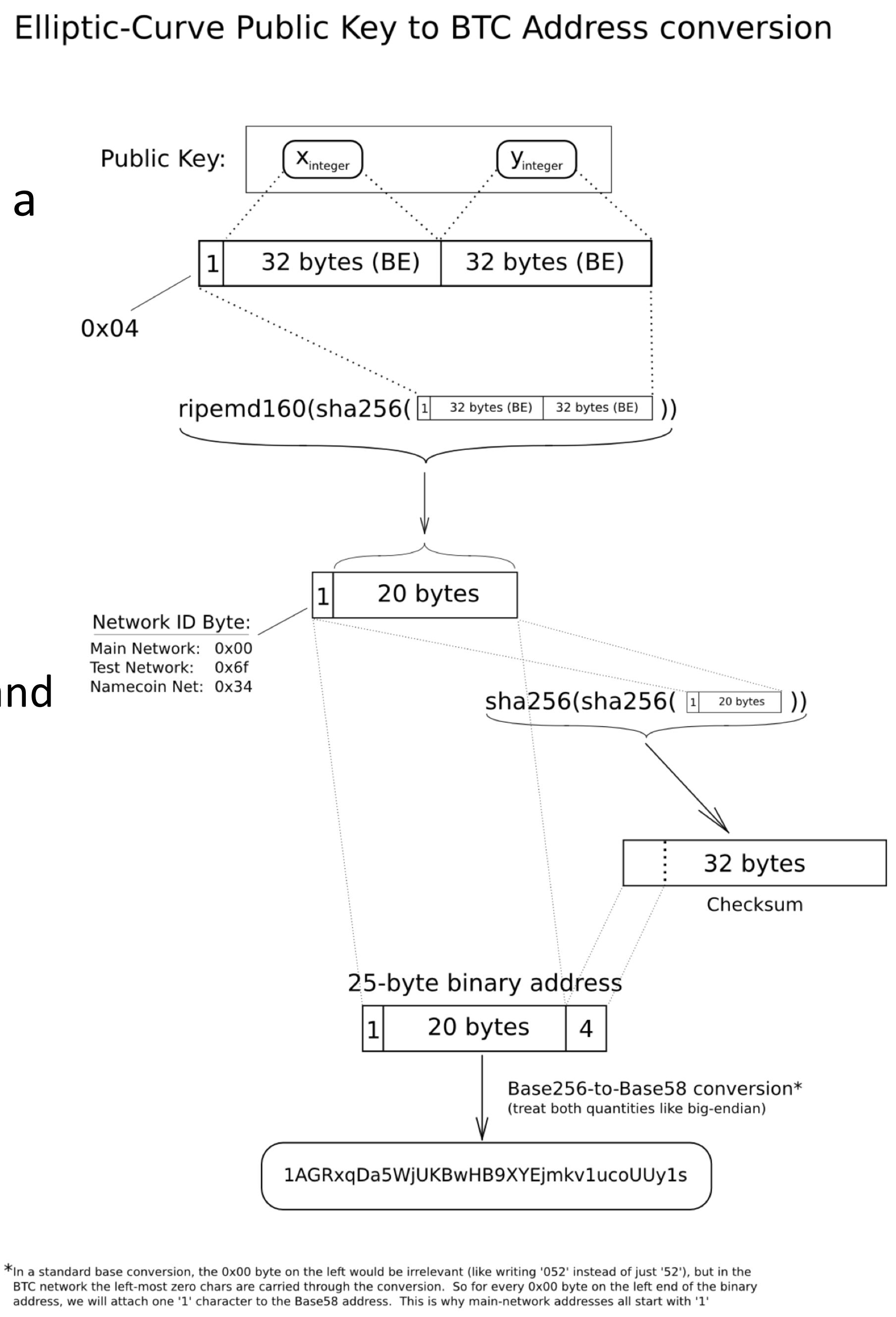




- choose a specific rounding function
- turn bounded quantification into executable recursion
- cache and use previous ordinate to compute the next one incrementally, with only integer operations and without rounding function
- cache and use previous decision expression to compute the next one incrementally, without using the point’s coordinates
- align the termination test with the one from the bottom-up steps
- join top-down sequence with bottom-up sequence
- change data representation from n-bit two’s complement integers to arbitrary-size integers
- turn Java’s modular arithmetic operations into exact arithmetic operations
- lift code to logic

## Bitcoin Public-Key-To-Address

- Goal: Obtain verified code that converts a bitcoin public key to an address
- Includes
  - 3 calls to SHA-256
  - 1 call to RIPEMD-160
  - Base58 encoding
- We specified it
  - Using specs for SHA-256, RIPEMD-160, and Base58 encoding
- We created an implementation using
  - SHA-256 and RIPEMD-160 from bouncycastle
  - Base58 encoding from bitcoinj
- We verified it
  - Axe to verify the crypto by unrolling
  - APT derivation to verify Base58
  - Axe to combine the proofs of the pieces



Elliptic-Curve Public Key to BTC Address conversion
Public Key:
x integer
y integer
1
32 bytes (BE)
32 bytes (BE)
0x04
ripemd160(sha256( 1 32 bytes (BE) 32 bytes (BE) ))
1
20 bytes
Network ID Byte:
Main Network: 0x00
Test Network: 0x6f
Namecoin Net: 0x34
sha256(sha256( 1 20 bytes ))
32 bytes
Checksum
25-byte binary address
1
20 bytes
4
Base256-to-Base58 conversion*
(treat both quantities like big-endian)
1AGRxqDa5WjUKBwHB9XYEjmkv1ucoUUy1s

*In a standard base conversion, the 0x00 byte on the left would be irrelevant (like writing '052' instead of just '52'), but in the BTC network the left-most zero chars are carried through the conversion. So for every 0x00 byte on the left end of the binary address, we will attach one '1' character to the Base58 address. This is why main-network addresses all start with '1'

etotheipi@gmail.com / 1Gffm7LKXcNFPrtxy6yF4JBoe5rVka4sn1

## Java/JVM Code That We Have Verified Using Axe and/or APT

- Bresenham’s Algorithm
- MAVLink checksum
- MAVLink message creation
- MAVLink message parsing
- MAVLink-over-HTTP message creation
- MAVLink-over-HTTP message parsing
- Bitcoin Base58 encoding
- Bitcoin public-key-to-address
- factorial
- Fibonacci
- sum of squares
- insertion sort
- membership
- find index
- max element
- negate bytes
- prime sieve
- reverse array
- lots of crypto code (previous slide)

## Axe for x86

- Can now apply Axe to lift and verify x86 code as we do for JVM code
  - including code with loops
- x86 tools are less well-developed than JVM versions

# Axe Code Query Tool

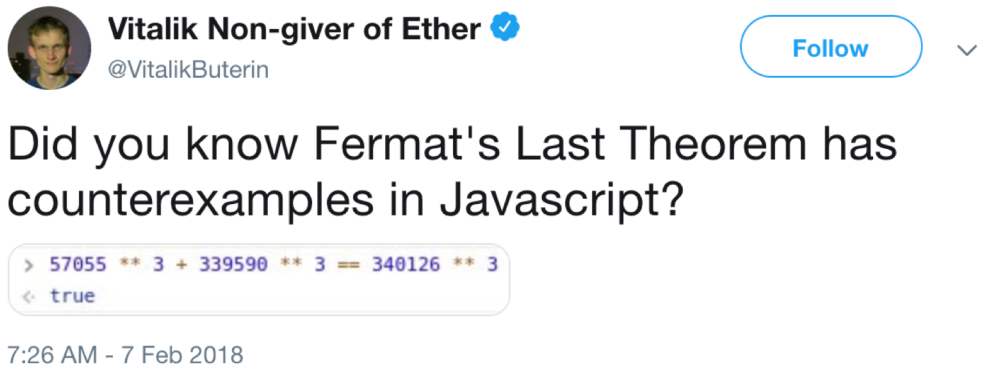
- Axe can answer questions like "Is there any input that causes this behavior?"
- Example:

```
class Fermat {
    // Fermat's equation when n = 3.  Test whether x^3 + y^3 = z^3.
    static boolean fermatEquation (int x, int y, int z) {
        return x * x * x + y * y * y == z * z * z;
    }
}
```

- Are there values of x, y, and z that cause this to return true (note that the math ops wrap around)?
- Axe finds:

```
Satisfying assignment: ((X . 53605105) (Y . 64151086) (Z . 71794969)).
```

- Vitalik noticed something similar for Javascript:


Vitalik Non-giver of Ether
@VitalikButerin

Did you know Fermat's Last Theorem has counterexamples in Javascript?

```
> 57055 ** 3 + 339590 ** 3 == 340126 ** 3
< true
```

7:26 AM - 7 Feb 2018

# Future Work

- Improve and apply Axe
  - make loop lifting more automatic
  - improve x86 support
  - perhaps support additional machines (LLVM, EVM, eWASM, ARM, ...)
  - look into emitting checkable ACL2 proofs
- Apply APT to bigger examples
  - currently synthesizing runtime monitors for AI systems
  - currently synthesizing a resolution theorem prover
  - new transformations and algorithm schemes will be needed
  - implement proof-emitting code generators
  - implement more automated derivation finding

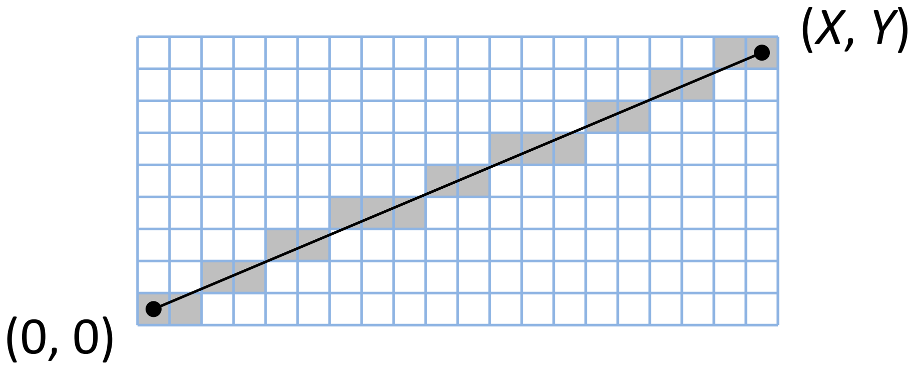
- **Let me know if you see opportunities to collaborate.**

## Questions?

# Extra Slides

# Bresenham Specification

compute a discrete best-fit line
from (0, 0) to $(X, Y) \in \mathbb{Z} \times \mathbb{Z}$, where $0 < Y \leq X$

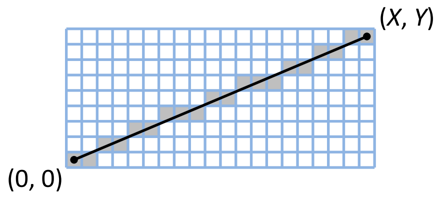


from the formal specification:

```
(forall i (implies (and (natp i)
                        (<= i X))
                   (roundingp (nth i ords-out)
                              (* i (/ Y X)))))
```

each $i$-th ordinate computed (where $0 \leq i \leq X$)
is a rounding of the exact ordinate $i \cdot (Y/X)$

# Bresenham Code



```
static void draw(int X, int Y, int[] ords) {
        int x = 0;
        int y = 0;
        int d = 2 * Y - X;
        while (x <= X) {
            ords[x] = y;
            x++;
            if (d >= 0) {
                y++;
                d += 2 * (Y - X);
            } else {
                d += 2 * Y;
            }
        }
    }
```

only integer operations
(Bresenham's algorithm!)

"Consider Bresenham's line drawing algorithm .... For efficiency, the algorithm only uses linear updates, which are non-trivial to verify [107] **or even understand** (let alone discover from scratch)."*

*Srivastava, Saurabh. "Satisfiability-based program reasoning and program synthesis." (2010).