A Formal Y86 Simulator with CHERI Features

FMCAD 2025

Carl Kwan, Yutong Xin, Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: October 6, 2025 at 12:34

Overview

Our objective: Explore formal proofs about CHERI-style capabilities in an x86-style ISA (Y86).

Our process:

- Introduce CHERI-style capabilities into an existing formal ISA model
- Explore memory protection and security features of CHERI within this context
- Develop techniques and tools for proving properties of capability-protected programs
- Prove the correctness of the CHERI capability compression scheme (CHERI Concentrate)
- Do everything within ACL2

CHERI

CHERI (Capability Hardware Enhanced RISC Instructions) is a capability architecture:

- University of Cambridge, SRI International, others
- Designed to add capabilities to existing ISAs: MIPS, RISC V, ARM
- Goal is to enable strong security protections, code containment
- An x86 version is under development

Microsoft described CHERI as a "building block for higher-level security abstractions."

Capabilities

- Unforgeable tokens, containing an address range and access permissions for that range
- Memory accesses are mediated via capabilities
- Holding a capability is sufficient to grant access to the resource
- Provides a number of security benefits

ACL2

ACL2 (A Computational Logic for Applicative Common Lisp) is a software system consisting of:

- a programming language (applicative subset of Common Lisp)
- an extensible theory in a first-order logic
- an automated theorem prover
- very efficient bit-twiddling features (BDDs and AIGs)

ACL2 is widely used for software and hardware verification.

Some industrial users: AMD, Arm, Centaur Technology, IBM, Intel, Oracle, and Collins Aerospace.

ACL2 ISA Modeling

ACL2 is a programming language; can build executable formal models / simulators for digital systems.

- AMD K5 floating point unit
- Java Virtual Machine
- Motorola CAP digital processor
- Rockwell Collins AAMP7 processor
- many others

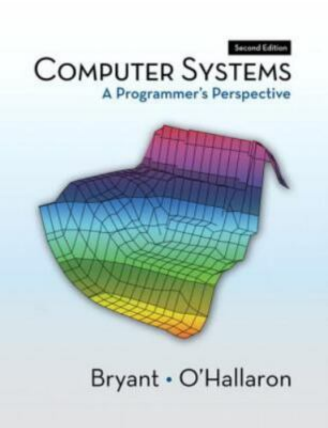
Most relevant for us, the x86 ISA (Shilpi Goel, et al).

Shilpi Goel's ACL2 x86 Model

- A very complete formal simulator of the x86 ISA (400+ instructions)
- Includes realistic memory management (segmentation and paging)
- Covers both user and system modes
- Validated via co-simulation with a physical x86 processor
- Provides the semantics for verifying x86 machine code
- Supports symbolic execution of x86 programs
- Fast: simulates 3M+ instructions/second on application programs
- Has been used to verify many x86 programs
- **Not a toy:** has been used to boot Alpine Linux

The Y86

- Very simple ISA developed by Randal Bryant and David O'Hallaron
- Loosely based on the x86 (32-bit and 64-bit versions)
- Widely used in teaching computer architecture
- Available assembler, simulator, other tools



Warren Hunt wrote a Y86 assembler and simulator in ACL2, our starting point for this work.

Why use the Y86, rather than x86?

Ours was a relatively small project aimed at *exploring* proofs of CHERI-style capabilities in a x86-style ISA.

> **Subtask 1.1(a)** *Prototype user-level (compiler) portions of CHERI-x86 ISA on a reduced Y86 model for early proof.*

Complete CHERI-x86 specs are not yet available.

The x86 is incredibly complex, and so is Shilpi's ACL2 x86 model.

So is CHERI.

CHERI-Y86 serves as a proof-of-concept for integrating CHERI into similar ISAs, especially the x86.

Y86 Model

The Y86 ISA model is an interpreter-style operational semantics:

- recursively-defined interpreter
- given instructions/programs stored in the processor state.
- each instruction is defined by a semantic function
- simulates the execution of machine code

Y86 Model

Four main aspects to any such model:

- **State:** data structure representing the current processor state (memory, registers, flags, etc.)
- **Instruction Semantic Functions:** for a specific instruction, take a processor state as input and return a modified state as output.
- **Step Function:** fetches, decodes and executes a machine instruction by calling an appropriate semantic function.
- **Run Function:** execute n steps on an initial state and return a modified state.

Can use ACL2 both to simulate concrete program runs and/or reason about symbolic program runs.

CHERI-Y86 State

Name	Field	Description
Registers	rgf	15 general-purpose 128-bit registers
RIP	rip	Program counter
ZF	zf	Zero flag
SF	sf	Sign flag
OF	of	Overflow flag
Memory	mem	2^{64} bytes modelled with 2^{24} addresses
MS	ms	Model state, indicates model errors

Registers are extended to 128 bits to accommodate CHERI capabilities.

If MS is anything other then `nil`, execution halts.

Y86 Step and Run Functions

```
(defun y86-step (y86-64)
  (b* ((pc (rip y86-64))
       (byte-at-pc (rm08 pc y86-64))
       (nibble-1 (logand byte-at-pc #xF0)))
      (case nibble-1
        ;; halt:  Stop the machine
        (#x00
         (case byte-at-pc
           (#x00 (y86-halt y86-64))
           (t    (y86-illegal-opcode y86-64))))
        ;; nop:  No-operation
        (#x10
         (case byte-at-pc
           (#x10 (y86-nop y86-64))
           (t    (y86-illegal-opcode y86-64))))

        ....
    )))

(defun y86 (y86-64 n)
  (if (or (zp n) (ms y86-64))
      y86-64
    (y86 (y86-step y86-64) (1- n))))))
```

Updates to Y86

To modify the Y86 with CHERI-like capabilities we needed to:

- Modify the state, extending registers from 64 to 128 bits
- Add Y86 versions of CHERI instructions that create and manipulate capabilities
- Modify existing Y86 instructions that touch memory to be capability aware (incomplete)

CHERI Instructions

CHERI capability instructions added to CHERI-Y86:

Capability inspection	**Capability modification**
cPerm-to-rB	cSeal
cType-to-rB	cAndPerm
cBase-to-rB	cSetOffset
cLen-to-rB	cSetAddr
cTag-to-rB	cIncOffset
gcOff	cIncOffsetImm
gcFlags	cSetBounds
gcHi	cSetBoundsExact
gcLim	cSetBoundsImm
	cClearTag
	cBuildCap
	cCopyType
	cCSeal
	cSealEntry

Example CHERI-Y85 Capability Instruction

An example CHERI instruction (pseudocode): Get permissions from the capability in rA, store in rB.

```
procedure cPerm-to-rB(y86-64)
   pc <- RIP(y86-64)
   if pc < 2^54 - 4 then
      return !MS(y86-64, "PC too large")
   rArB <- ReadMem(pc+2, y86-64)
   rB <- rArB & 2^4
   rA <- ShiftRight(rArB, 4) & 2^4 - 1
   if rA = 15 or rB = 15 then
      return !MS(y86-64, "Prohibited register")
   cs1 <- GetCapability(y86-64, rA)
   cPerms <- GetPerm(cs1)
   y86-64 <- !rgfi(rb, cPerms, y86-64)
   y86-64 <- !rip(pc+3, y86-64)
   return y86-64
```

Example Property: Monotonicity

An important property of capabilities is *monotonicity*: a new capability created from an existing capability doesn't expand the permissions.

Example theorem: if a capability-modification instruction is given a capability c_1 that results in a new capability c_2, then permissions$(c_2) \subset$ permissions(c_1).

This might apply to `cSetBounds`, which takes a capability c_1 and register r to create a new capability c_2 with bounds based on the address of c_1 and the value in r. (All fields but the bounds are identical to c_1.)

Example Property: Monotonicity of setBounds

```
(defthmd setBounds-cap-bounded
  (b* ((cs1 (get-reg-capability 0 y86-64))
       (rs2 (rgfi 1 y86-64))
       (addr (+ (acap->base cs1)
                (acap->offset cs1)))
       (cs1 (change-acap cs1 :base addr))
       (cs1 (change-acap cs1 :len rs2))
       (new-cap (get-reg-capability 2
                      (y86-cap-step y86-64))))
      (implies
           (and (y86-64p y86-64)
                (y86-mem-program-bytes-loadedp
                      *mod-instr-code-1* y86-64)
                (equal (rip y86-64) 131)
                (equal (rgfi 0 y86-64) (rm128  0 y86-64))
                (equal (rgfi 1 y86-64) (rm128 16 y86-64)))
           (and (<= (cGetBase cs1) (cGetBase new-cap))
                (<= (cGetTop new-cap) (cGetTop cs1))))))
```

Raw (Architectural) Capabilities in CHERI

A CHERI capability encodes:

- 64-bit base address (base)
- 64-bit length (length)
- 64-bit offset
- 16-bit permissions (perms)
- 19-bit metadata

This requires 256 bits to store, which is expensive, and would require 256-bit registers.

CHERI Concentrate

The CHERI developers designed a compressed format they call *CHERI Concentrate*.

- 27-bit compressed bounds
- 64-bit current address (base + offset)
- 16-bit permissions (perms)
- 19-bit metadata

This compressed form fits into 128 bit memory / registers.

Reduces L2 cache misses by 50–70% compared to uncompressed capabilities.

CHERI Capabilities

64							0
p		otype	I	$t_{[11:3]}$	t_E	$b_{[13:33]}$	b_E
a							

CHERI capabilities contain:

- a: 64-bit address
- p: 16 permission bits
- otype: type of object designated / sealed
- I: 1 bit indicating whether bounds are in exponent form
- $t_{[11:3]}\&t_E$: 12 bits for computing "top" 14-bit memory address
- $b_{[13:3]}\&b_E$: 14 bits for computing "base" 14-bit memory address

Properties of CHERI Concentrate

We model the CHERI Concentrate encode (compression) and decode algorithms.

Naturally, compressing 227 bits of information into 128 bits loses accuracy, *except in certain circumstances*.

Properties of CHERI Concentrate

Let b_0, l_0 and t_0 be base, length and top of the raw capability. Let b_1, l_1 and t_1 be base, length and top after encoding and then decoding the result.

We verify these properties:

1. $b_0 \geq b_1$, for any b_0, t_0, and address;
2. $b_0 - b_1 \leq 2^{E+3}$, for any b_0, t_0, and address;
3. $t_0 \leq t_1$, for any b_0, t_0, and address;
4. $t_1 - t_0 \leq 2^{E+3}$, for any b_0, t_0, and address;
5. $b_0 = b_1$ and $t_0 = t_1$ when the lower $E + 3$ bits of b_0 and t_0 are zero;
6. $b_0 = b_1$ and $t_0 = t_1$ when $l_0 < 2^{12}$

Proving CHERI Concentrate

These properties of CHERI Concentrate are bit-twiddling properties over a finite domain.

We used ACL2's symbolic simulation framework GL, which supports model checking with binary decision diagrams (BDDs).

This is property 6.

```
(def-gl-thm decode-encode-equal-small-seg
    :hyp   (and (valid-addr-p addr base len)
                (valid-b-l-p base len)
                (< len (expt 2 12)))
    :concl (equal (decode-compression
                     (encode-compression len base) addr)
                  (bounds (+ len base) base))
    :g-bindings `((base ,(gl::g-int 0 3 65))
                  (len ,(gl::g-int 1 3 66))
                  (addr ,(gl::g-int 2 3 65))))
```

What We Accomplished

1. Modeled CHERI capabilities in ACL2.
2. Modified our Y86 model to be capability-aware (CHERI-Y86), adding all CHERI capability instructions.
3. Proved security properties of some simple programs
4. Proved some security properties of CHERI instructions (e.g. monotonicity)
5. Proved CHERI Concentrate conversions between *memory-resident* capabilities and *raw capabilities*

More to be Done

- Add pre- and post-processing to allow reading compiled binaries
- Port capability features from Y86 to x86
- Prove properties of significant capability-aware applications