# UT Austin CRASH Project

## X86 Binary Code Analysis

**Shilpi Goel, Marijn Heule, Warren A. Hunt, Jr., Matt Kaufmann, J Strother Moore, Nathan Wetzler**

**August, 2013**

Computer Science Department
1 University Way, M/S C0500
University of Texas
Austin, TX 78712-0233

{shigoel, marijn, hunt, kaufmann
moore, nwetzler}@cs.utexas.edu
TEL: +1 512 471 9748
FAX: +1 512 471 8885

## Outline

## Introduction

To enable the modeling and analysis of industrial-sized systems:

- We are developing ISA models suitable for code analysis.
- We are extending our ACL2-based analysis toolsuite.
- We are vetting our tools on commercial-sized problems.

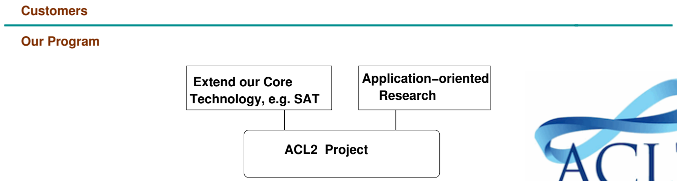
Our ACL2-based modeling and analysis toolsuite is in use by AMD, Centaur, IBM, Raytheon, Rockwell-Collins, and others.

Today, we present our analysis approach for modeling the X86 ISA and analyzing X86 binary-level programs.

These slides were previously presented at a DARPA CRASH PI Meeting in November, 2012. Some slides were elided to keep this presentation brief.

## Ecosystem

We have significant collaboration with the industry.

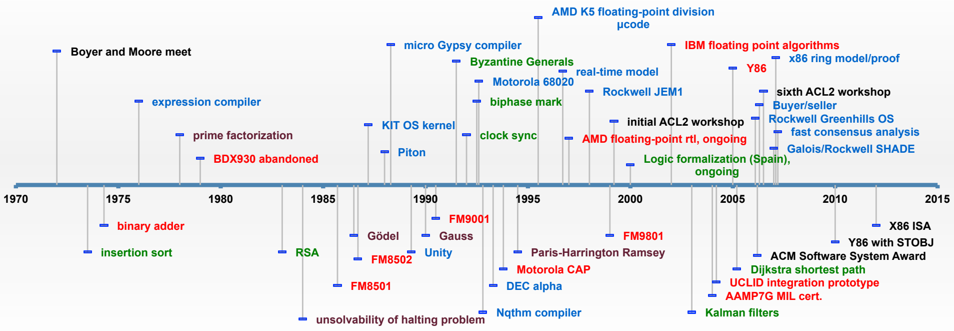

Customers
Our Program
Extend our Core Technology, e.g. SAT
Application–oriented Research
ACL2 Project

Our own research includes:

- **Development** of core technologies
- **Application** of these technologies on different verification domains
- **Commercial Driver** validation of Centaur's X86 design

## Timeline

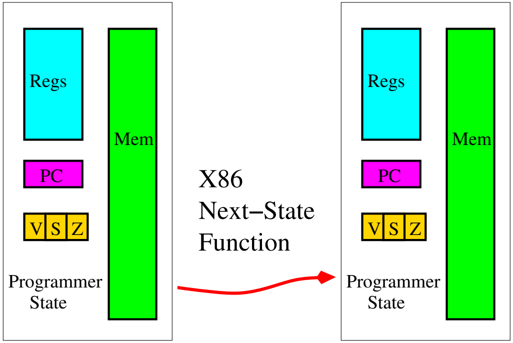
- Our group has been working on the development and deployment of reasoning systems for 40 years.



## Evolving X86 ISA Model

We are developing an ISA for the X86 architecture.

- X86 model implements almost all one- and two-byte instructions
- For all defined instructions, model implements all addressing modes
- It can emulate (almost all) X86 binary-level, integer programs emitted by GCC/LLVM



## X86 Top-Level Model

We extended **(now 118 instructions, 219 opcodes)** our X86 ISA model.

```
(defun x86-run (n x86)
 ; Returns x86 obtained by executing n instructions (or until halting).
  (cond ((ms x86) x86)

        ((zp n) x86)

        (t (let ((x86 (x86-fetch-decode-execute x86)))
             (x86-run (1- n) x86)))))
```

Our X86 ISA model is now about 40,000 lines in size.

- Without MM enabled: ∼3+ million instructions/sec
- With MM enabled: ∼580K instructions/sec

These numbers will improve. All memory accesses are currently performed as bytes, thus a 64-bit read requires 132+ memory accesses. If we use 32-bit values, a 64-bit read only requires 10+ memory accesses.

# X86 One-Byte OPCODE Map

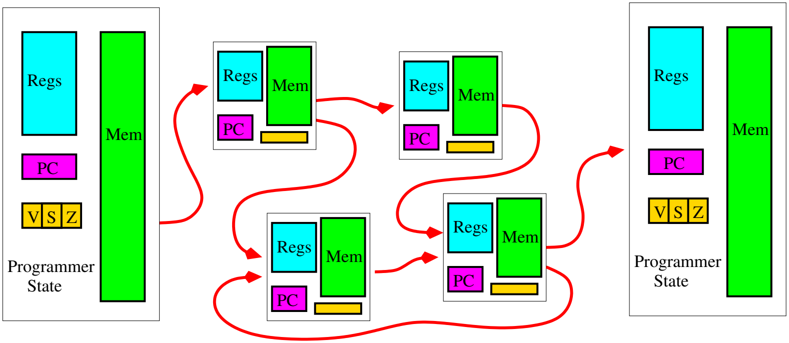
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | ADD | ADD | ADD | ADD | ADD | ADD | 🟨 | 🟨 | OR | OR | OR | OR | OR | OR | 🟨 | Two-byte escape |
| 1 | ADC | ADC | ADC | ADC | ADC | ADC | 🟨 | 🟨 | SBB | SBB | SBB | SBB | SBB | SBB | 🟨 | 🟨 |
| 2 | AND | AND | AND | AND | AND | AND | 🟩 | 🟨 | SUB | SUB | SUB | SUB | SUB | SUB | 🟩 | 🟨 |
| 3 | XOR | XOR | XOR | XOR | XOR | XOR | 🟩 | 🟨 | CMP | CMP | CMP | CMP | CMP | CMP | 🟩 | 🟨 |
| 4 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |
| 5 | PUSH | PUSH | PUSH | PUSH | PUSH | PUSH | PUSH | PUSH | POP | POP | POP | POP | POP | POP | POP | POP |
| 6 | 🟨 | 🟨 | 🟨 | MOVSXD | 🟩 | 🟩 | 🟩 | 🟩 | PUSH | IMUL | PUSH | IMUL | INS | INS | OUTS | OUTS |
| 7 | JO | JNO | JB | JNB | JZ | JNZ | JBE | JNBE | JS | JNS | JP | JNP | JL | JNL | JLE | JNLE |
| 8 | Imm Grp 1A | Imm Grp 1A | 🟨 | Imm Grp 1A | TEST | TEST | XCHG | XCHG | MOV | MOV | MOV | MOV | MOV | LEA | MOV | POP |
| 9 | NOP | XCHG | XCHG | XCHG | XCHG | XCHG | XCHG | XCHG | CBW | CWD | 🟨 | WAIT | PUSHF | POPF | SAHF | LAHF |
| A | MOV | MOV | MOV | MOV | MOVS | MOVS | CMPS | CMPS | TEST | TEST | STOS | STOS | LODS | LODS | SCAS | SCAS |
| B | MOV | MOV | MOV | MOV | MOV | MOV | MOV | MOV | MOV | MOV | MOV | MOV | MOV | MOV | MOV | MOV |
| C | Shift Grp 2 | Shift Grp 2 | RETN | RETN | 🟨 | 🟨 | MOV | MOV | ENTER | LEAVE | RETF | RETF | INT 3 | INT | 🟨 | IRET |
| D | Shift Grp 2 | Shift Grp 2 | Shift Grp 2 | Shift Grp 2 | 🟨 | 🟨 | 🟨 | XLAT | Escape to coprocessor instruction set | | | | | | | |
| E | LOOPNE | LOOPE | LOOP | JRCXZ | IN | IN | OUT | OUT | CALL Jz | NEAR Jz | 🟨 | Short Jb | IN | IN | OUT | OUT |
| F | 🟩 | 🟨 | 🟩 | 🟩 | HLT | CMC | Unary Grp 3 | Unary Grp 3 | CLC | STC | CLI | STI | CLD | STD | INC/DEC Grp4 | INC/DEC Grp4 |

| Legend: | |
|---|---|
| 🟩 | Implemented instructions |
| 🟩 | Implemented prefixes |
| 🟥 | Instructions not yet implemented |
| 🟥 | Instructions not supported by our model |
| 🟨 | Instructions not available in X86 64-bit mode |
| ⬜ | Escape to other opcode maps |

## Verifying X86 Programs

As an example, we use ACL2 to symbolically execute our X86 model.

- We compile C-code with GCC/LLVM.
- We *load* binary code into the *memory* of our X86 model.
- We initialize the registers, etc. with symbolic values.



## Popcount C-code

```
int popcount_32 (unsigned int v)
{
  v = v - ((v >> 1) & 0x55555555);   // by Sean Anderson
  v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
  v = ((v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >> 24;
  return (v);
}

int popcount_64 (long unsigned int v)
{
  long unsigned int v1, v2;
  v1 = (v & 0xFFFFFFFF);      // v1: lower 32 bits of v
  v2 = (v >> 32);             // v2: upper 32 bits of v
  return (popcount_32(v1) + popcount_32(v2));
}

int popcount_128 (long unsigned int v1, long unsigned int v2)
{
  if (v1 == 0)
    return (42);     // BUG introduced here!
  else
    return (popcount_64(v1) + popcount_64(v2));
}
```

## Popcount X86 Binary Code

```
(defconst *popcount/popcount-128-bug-binary*
  (list
   ;; Section: <popcount_32>:
   (cons #x4005c0 #x89) ;; mov %edi,%edx
   (cons #x4005c1 #xfa) ;;
   (cons #x4005c2 #x89) ;; mov %edi,%eax
   ...

   ;; Section: <popcount_64>:
   (cons #x400600 #x89) ;; mov %edi,%esi
   (cons #x400601 #xfe) ;;
   (cons #x400602 #x89) ;; mov %edi,%eax
   (cons #x400603 #xf8) ;;
   (cons #x400604 #xd1) ;; shr %esi
   (cons #x400605 #xee) ;;
   (cons #x400606 #x81) ;; and $0x55555555,%esi
   ...

   ;; Section: <popcount_128>:
   (cons #x400680 #x48) ;; mov %rbx,-0x10(%rsp)
   (cons #x400681 #x89) ;;
   (cons #x400682 #x5c) ;;
   (cons #x400683 #x24) ;;
   (cons #x400684 #xf0) ;;
   (cons #x400685 #x48) ;; mov %rbp,-0x8(%rsp)
   (cons #x400686 #x89) ;;
   (cons #x400687 #x6c) ;;
   (cons #x400688 #x24) ;;
   (cons #x400689 #xf8) ;;
   (cons #x40068a #x48) ;; sub $0x10,%rsp
   ... ))
```

## Not a Theorem

```
(def-gl-thm x86-popcount-correct
  :hyp (and (natp n)
            (< n (expt 2 128)))
  :concl
  (let* ((start-address #x400680)                           ; Program start
         (halt-address #x4006b9)                            ; Stop here
         (x86 (setup-for-popcount-run                       ; Initialize
               nil start-address halt-address nil 0         ;   Simulator
               *popcount/popcount-128-binary*))             ;   with code
         (x86 (!rgfi *rdi* (logand n *2^64-1*) x86))        ; RDI lower 64 bits of n
         (x86 (!rgfi *rsi* (ash n -64) x86))                ; RSI upper 64 bits of n
         (x86 (!rgfi *rsp* *2^45* x86))                     ; Stack pointer RSP set
         (count 300)                                        ; Maximum steps
         (x86 (x86-run count x86)))                         ; Simulate X86 model
    (and (equal (rgfi *rax* x86)                            ; Popcount by X86 code
                (logcount n))                               ;   equal to spec?
         (equal (rip x86)                                   ; Simulation stoped at
                (+ 1 halt-address))))                       ;   halt address?
  :g-bindings
  `((n   (:g-number ,(gl-int 0 1 129))))
  :rule-classes nil)
```

## Is a Theorem

```
(def-gl-thm x86-popcount-correct
  :hyp (and (natp n)                  ; Exclude inputs with lower 64-bits zero
            (< n (expt 2 128))
            (not (equal (logand n *2^64-1*) 0)))
  :concl
  (let* ((start-address #x400680)                          ; Program start
         (halt-address #x4006b9)                           ; Stop here
         (x86 (setup-for-popcount-run                      ; Initialize
               nil start-address halt-address nil 0        ;   Simulator
               *popcount/popcount-128-binary*))            ;   with code
         (x86 (!rgfi *rdi* (logand n *2^64-1*) x86))       ; RDI lower 64 bits of n
         (x86 (!rgfi *rsi* (ash n -64) x86))               ; RSI upper 64 bits of n
         (x86 (!rgfi *rsp* *2^45* x86))                    ; Stack pointer RSP set
         (count 300)                                       ; Maximum steps
         (x86 (x86-run count x86)))                        ; Simulate X86 model
    (and (equal (rgfi *rax* x86)                           ; Popcount by X86 code
                (logcount n))                              ;   equal to spec?
         (equal (rip x86)                                  ; Simulation stoped at
                (+ 1 halt-address))))                      ;   halt address?
  :g-bindings
  `((n   (:g-number ,(gl-int 0 1 129))))
  :rule-classes nil)
```

## Ongoing Work, Future Work

Extending the ACL2 system:

- Integrate SAT mechanisms into ACL2
- Extend our symbolic simulation techniques
- In general, improve the ACL2 system to support our X86 ISA

Extending our X86 ISA model:

- Continue integrating X86 memory management into our model
- Continue extending the number of instructions modeled
- Develop co-simulation environment for model validation

Check FreeBSD binary-level code properties?

- Interested in simple problems meaningful to FreeBSD community
- We can provide dynamic verification capabilities, flow analysis, ...

## Conclusion

We continue to expand our modeling and analysis capabilities.

- We have developed a 64-bit data and 52-bit address memory model
- We have specified most integer instructions with their addressing modes
- We are developing a co-simulation mechanism for model validation
- We have started verifying X86 binary programs
- Our model can be used as a built-to and a compile-to specification
- Our model can be used to safely explore all manner of malware

We perform all of our work in an environment where we can prove or disprove theorems about our models.