Analysis of x86 Application and System Programs via Machine-Code Verification

Shilpi Goel, Warren A. Hunt, Jr., and Matt Kaufmann

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

hunt@cs.utexas.edu
TEL: +1 512 471 9748
FAX: +1 512 471 8885

April, 2016
Introduction

Motivation: Increase the reliability of programs used in the industry

Approach: Machine-code verification for x86 platforms
- We are developing a formal x86 model in ACL2 for code analysis.
- We are vetting our tools on commercial-sized problems.

Today, we talk about our ongoing work in formal verification of software, and present our plans to verify supervisor-level code in the immediate future.

Objective: Emulate an operating system, like FreeBSD, along with the programs running on it, and prove properties about kernel code
Ecosystem

Our group has significant collaboration with the government and industry.

Our own research includes:
- Development of core technologies
- Application of these technologies in different domains
- Validation of commercial processor designs at Centaur and Oracle (10+ developers, 30+ users)
Project Overview

Goal: Build robust tools to increase software reliability
   ‣ Verify critical properties of application and system programs
   ‣ Correctness with respect to behavior, security, & resource usage

Plan of Action:
1. Build a formal, executable x86 ISA model using ACL2
2. Develop a machine-code analysis framework based on this model
3. Employ this framework to verify application and system programs
Contributions

A **new tool**: General-purpose analysis framework for x86 machine-code
- Accurate x86 ISA reference

*Program verification taking memory management into account:*
- Properties of x86 memory-management data structures
- Analysis of programs, including low-level system & ISA features

*Reasoning strategies*: Insight into low-level code verification in general
- Build effective lemma libraries

*Foundation for future research:*
- Target for verified/verifying compilers
- Resource usage guarantees
- Information-flow analysis
- Ensuring process isolation
Outline

- Motivation
- Project Description
  - [1] Developing an x86 ISA Model
  - [3] Verifying Application and System Programs
- Future Work & Conclusion
- Accessing Source Code + Documentation
Model Development

Obtaining the x86 ISA Specification
Model Development

Obtaining the x86 ISA Specification

Intel® 64 and IA-32 Architectures Software Developer’s Manual

Combined Volumes:
1, 2A, 2B, 2C, 3A, 3B and 3C

~3400 pages
Model Development

Obtaining the x86 ISA Specification

Intel® 64 and IA-32 Architectures
Software Developer’s Manual

Combined Volumes:
1, 2A, 2B, 2C, 3A, 3B and 3C


~3400 pages

AMD64 Technology

AMD64 Architecture
Programmer’s Manual
Volume 3:
General-Purpose and
System Instructions

All AMD manuals: ~3000 pages
Model Development

Obtaining the x86 ISA Specification

Intel® 64 and IA-32 Architectures
Software Developer’s Manual
Combined Volumes:
1, 2A, 2B, 2C, 3A, 3B and 3C

AMD64 Technology

AMD64 Architecture
Programmer’s Manual

```
__asm__ volatile
("stc\n\t"
"mov $0, %eax\n\t"
"mov $0, %ebx\n\t"
"mov $0, %ecx\n\t"
"mov $0, %edx\n\t"
"mov %4, %ecx\n\t"
"mov %3, %edx\n\t"
"mov %2, %eax\n\t"
"rcl %cl, %al\n\t"
"cmovb %edx, %ebx\n\t"
"mov %%eax, %0\n\t"
"mov %%ebx, %1\n\t"
"mov %eax, %0\n\t"
"mov %ebx, %1\n\t"

: "=g"(res), "=g"(cf)
: "g"(num), "g"(old_cf), "g"(rotate_by)
: "rax", "rbx", "rcx", "rdx";
```

Running tests on x86 machines
Model Development

Focus: 64-bit sub-mode of Intel’s IA-32e mode
Focus: 64-bit sub-mode of Intel’s IA-32e mode

Figure 3-2. 64-Bit Mode Execution Environment

Source: Intel Manuals
Focus: 64-bit sub-mode

BASIC EXECUTION ENVIRONMENT

- **Debug registers** - Debug registers expand to 64 bits. See Chapter 17, “Debug, Branch Profile, TSC, and Quality of Service,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

- **Descriptor table registers** - The global descriptor table register (GDTR) and interrupt descriptor table register (IDTR) expand to 10 bytes so that they can hold a full 64-bit base address. The local descriptor table register (LDTR) and the task register (TR) also expand to hold a full 64-bit base address.

3.3 MEMORY ORGANIZATION

The memory that the processor addresses on its bus is called physical memory. Physical memory is organized as a sequence of 8-bit bytes. Each byte is assigned a unique address, called a physical address. The physical address space ranges from zero to a maximum of $2^{36} - 1$ (64 GBytes) if the processor does not support Intel 64-bit mode.

Figure 3-2. 64-Bit Mode Execution Environment

Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode

Source: Intel Manuals
64-bit sub-mode

**asic Program Execution Registers**

<table>
<thead>
<tr>
<th>Registers</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>General-Purpose Registers</td>
<td>16-bit Registers</td>
</tr>
<tr>
<td>Segment Registers</td>
<td>64-bits</td>
</tr>
<tr>
<td>RFLAGS Register</td>
<td>64-bits</td>
</tr>
<tr>
<td>RIP (Instruction Pointer Register)</td>
<td>64-bits</td>
</tr>
<tr>
<td>Eight 80-bit Registers</td>
<td>16 bits, 16 bits, 16 bits</td>
</tr>
<tr>
<td>Floating-Point Data Registers</td>
<td>16 bits, 16 bits, 16 bits</td>
</tr>
<tr>
<td>Control Register</td>
<td>16 bits</td>
</tr>
<tr>
<td>Status Register</td>
<td>16 bits</td>
</tr>
<tr>
<td>Tag Register</td>
<td>16 bits</td>
</tr>
<tr>
<td>Opcode Register</td>
<td>16 bits</td>
</tr>
<tr>
<td>FPU Instruction</td>
<td>16 bits</td>
</tr>
<tr>
<td>FPU Data (Operand)</td>
<td>16 bits</td>
</tr>
<tr>
<td>MMX Registers</td>
<td>16 bits</td>
</tr>
<tr>
<td>MXCSR Register</td>
<td>32-bits</td>
</tr>
</tbody>
</table>

**In 128-bit Registers**

<table>
<thead>
<tr>
<th>Registers</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handler's Stack</td>
<td>SS, ESP, EFLAGS, CS, ESP, Error Code</td>
</tr>
<tr>
<td>ESP Before Transfer to Handler</td>
<td>16 bits</td>
</tr>
<tr>
<td>ESP After Transfer to Handler</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

**Interrupted Procedure's Stack**

<table>
<thead>
<tr>
<th>Registers</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFLAGS, CS, ESP, Error Code</td>
<td>16 bits</td>
</tr>
<tr>
<td>ESP Before Transfer to Handler</td>
<td>16 bits</td>
</tr>
<tr>
<td>ESP After Transfer to Handler</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

**Stack Usage with No Privilege-Level Change**

<table>
<thead>
<tr>
<th>Registers</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Register</td>
<td>16 bits</td>
</tr>
<tr>
<td>Status Register</td>
<td>16 bits</td>
</tr>
<tr>
<td>Tag Register</td>
<td>16 bits</td>
</tr>
<tr>
<td>Opcode Register</td>
<td>16 bits</td>
</tr>
<tr>
<td>FPU Instruction</td>
<td>16 bits</td>
</tr>
<tr>
<td>FPU Data (Operand)</td>
<td>16 bits</td>
</tr>
<tr>
<td>MMX Registers</td>
<td>16 bits</td>
</tr>
<tr>
<td>MXCSR Register</td>
<td>32-bits</td>
</tr>
</tbody>
</table>

**In 128-bit Registers**

<table>
<thead>
<tr>
<th>Registers</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handler's Stack</td>
<td>SS, ESP, EFLAGS, CS, ESP, Error Code</td>
</tr>
<tr>
<td>ESP Before Transfer to Handler</td>
<td>16 bits</td>
</tr>
<tr>
<td>ESP After Transfer to Handler</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

**Interrupted Procedure's Stack**

<table>
<thead>
<tr>
<th>Registers</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFLAGS, CS, ESP, Error Code</td>
<td>16 bits</td>
</tr>
<tr>
<td>ESP Before Transfer to Handler</td>
<td>16 bits</td>
</tr>
<tr>
<td>ESP After Transfer to Handler</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Model Development

64-bit sub-mode

asic Program Execution Registers

Sixteen 64-bit Registers

Segment Registers

General-Purpose Registers

Figure B-1. General Machine Instruction Format

NOTE:
* The Reg Field may be used as an opcode extension field (TTT) and as a way to encode diagnostic registers (see).

Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

Interrupted Procedure’s and Handler’s Stack

Interrupted Procedure’s Stack

Figure 5-3. Protection Rings

Protection Rings

Operating System Kernel

Operating System Services

Applications

Level 0

Level 1

Level 2

Level 3

Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode

Page

Page Table

Page Dir.

Page Directory Table

Offset

Physical Addr.

Page

Page Table Entry

Physical Address

TSS Desc.

Seg. Desc.

Task State

Segment Selector

Register

Interrupt Vector

TR

Current TSS

NULL

Code

Stack

Interrupt Handler

Exception Handler

Protected Procedure

Code

Stack

NULL

Interrupt, Handler

NULL

Code

Stack

NULL

Code

Stack

Interrupt Handler

Figure 3-2. 64-Bit Mode Execution Environment

Figure 2-1. General Machine Instruction Format

Legacy Prefixes

REX Prefixes

Grp 1, Grp 2, (optional)

Grp 3, Grp 4

1, 2, or 3 Byte Opcodes (T = Opcode)

7-6 5-3 2-0 7-6 5-3 2-0

Mod Reg* R/M Scale Index Base d32 | 16 | 8 | None
d32 | 16 | 8 | None

Address Displacement

Immediate Data

(4, 2, 1 Bytes or None)

(4, 2, 1 Bytes or None)

NOTE:
* The Reg Field may be used as an opcode extension field (TTT) and as a way to encode diagnostic registers (see).

Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

Interrupted Procedure’s and Handler’s Stack

Interrupted Procedure’s Stack

Figure 5-3. Protection Rings

Protection Rings

Operating System Kernel

Operating System Services

Applications

Level 0

Level 1

Level 2

Level 3

Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode

Page

Page Table

Page Dir.

Page Directory Table

Offset

Physical Addr.

Page

Page Table Entry

Physical Address

TSS Desc.

Seg. Desc.

Task State

Segment Selector

Register

Interrupt Vector

TR

Current TSS

NULL

Code

Stack

Interrupt Handler

Exception Handler

Protected Procedure

Code

Stack

NULL

Interrupt, Handler

NULL

Code

Stack

NULL

Code

Stack

Interrupt Handler

Source: Intel Manuals

Model Development
Model Development

Under active development: an x86 ISA model in ACL2

- **x86 State**: specifies the components of the ISA (registers, flags, memory)

- **Instruction Semantic Functions**: specify the effect of each instruction

- **Step Function**: fetches, decodes, and executes one instruction

Layered modeling approach mitigates the trade-off between reasoning and execution efficiency [ACL2’13]
How can we know that our model faithfully represents the x86 ISA?

Validate the model to increase trust in the applicability of formal analysis.
Current Status: x86 ISA Model

• The x86 ISA model supports 400+ instructions, including some floating-point and supervisor-mode instructions
  ‣ Can execute almost all user-level programs emitted by GCC/LLVM
  ‣ Successfully co-simulated a contemporary SAT solver on our model
  ‣ Successfully simulated a supervisor-mode zero-copy program

• IA-32e paging for all page configurations (4K, 2M, 1G)

• Segment-based addressing

• Lines of Code: ~85,000 (not including blank lines)

• Simulation speed*:
  ‣ ~3.3 million instructions/second (paging disabled)
  ‣ ~330,000 instructions/second (with 1G pages)

Model Development: Current Status

* Simulation speed measured on an Intel Xeon E31280 CPU @ 3.50GHz
Outline

- Motivation
- Project Description
  - [1] Developing an x86 ISA Model
  - [3] Verifying Application and System Programs
- Future Work & Conclusion
- Accessing Source Code + Documentation
Building a Lemma Database

- Semantics of the program is given by the effect it has on the machine state.

1. read instruction from mem
2. read flags
3. write new value to pc

The database should include lemmas about reads from and writes to the machine state, along with the interactions between these operations.
Building a Lemma Database

• System data structures, like the paging structures, are extremely complicated.

• Correct operation of a system heavily depends upon such structures.

• We need to prove lemmas that can aid in proving the following kinds of critical properties:
  - Processes are isolated from each other.
  - Page tables, including access rights, are set up correctly.
Address Translations
### Address Translations

<table>
<thead>
<tr>
<th>Logical Address</th>
<th>Segment Selector</th>
<th>Offset</th>
</tr>
</thead>
</table>

**SEGMENTATION**
Address Translations

Logical Address

Segment Selector

Offset

Descriptor Table(s)

Segment Descriptor

Linear Memory

Global or Local Descriptor Table Register

SEGMENTATION
Address Translations

Segment Selector

Descriptor Table(s)

Segment Descriptor

Linear Memory

Global or Local Descriptor Table Register

Logical Address

Offset

Segment

Linear Addr.

SEGMENTATION
Address Translations

Logical Address

Segment Selector

Offset

Descriptor Table(s)

Segment Descriptor

Segment

Linear Addr.

Linear Memory

Global or Local Descriptor Table Register

Control Register has the base address of these structures.

PML4E

PML4

Dir. Ptr.

Dir.

Table

Offset

IA-32e PAGING (4K page)
Address Translations

Logical Address

Segment Selector

Offset

Descriptor Table(s)

Segment Selector

Segment Descriptor

Linear Addr.

Segment

Linear Memory

Global or Local Descriptor Table Register

IA-32e PAGING (4K page)

Control Register has the base address of these structures.

PML4E

PML4

Dir. Table

Dir. Ptr.

Offset

Physical Memory

PDPT

CR3

SEGMENTATION
SEGMENTATION

Address Translations

IA-32e PAGING (4K page)
Address Translations

Logical Address

Segment Selector

Offset

Descriptor Table(s)

Segment Descriptor

Linear Address

Segment

Linear Memory

Global or Local Descriptor Table Register

Control Register has the base address of these structures.

Physical Memory

PML4E

PML4 Dir. Ptr.

Dir.

Table

Offset

PTE

PDPTE

PDE

PML4
Address Translations

SEGMENTATION

Linear Address

Segment Selector

Offset

Descriptor Table(s)

Segment Descriptor

Linear Address

4K Page

Segment

Linear Memory

Global or Local Descriptor Table Register

Control Register has the base address of these structures.

IA-32e PAGING (4K page)

CR3

PML4E

PDPTE

PDE

PTE

Physical Addr.

4K Page

Physical Memory

Linear Address

PML4

Dir. Ptr.

Dir.

Table

Offset

Logical Address

Segment Selector

Offset

Descriptor Table(s)

Segment Descriptor

Linear Address

4K Page

Segment

Linear Memory

Global or Local Descriptor Table Register
Address Translations

Logical Address

Segment Selector

Offset

Descriptor Table(s)

Segment Descriptor

Linear Addr.

4K Page

Segment

Linear Memory

Global or Local Descriptor Table Register

Control Register has the base address of these structures.

PML4

Dir. Ptr.

Dir.

Table

Offset

 Linear Address

PML4E

PDE

PDPTLE

PTE

Physical Addr.

4K Page

Physical Memory

Control Register has the base address of these structures.

CR3

4K Page

IA-32e PAGING (4K page)

Segmentation

accessed flag
Address Translations

Segment Selector  Offset

Descriptor Table(s)

Segment Descriptor

Linear Addr.

4K Page

Segment

Linear Memory

Global or Local Descriptor Table Register

SegmenSelector

Offset

Logical Address

Linear Address

PML4

Dir. Ptr.

Dir.

Table

Offset

PDE

PDPTE

PML4E

PTE

Physical Addr.

4K Page

Physical Memory

CR3

Control Register has the base address of these structures.

Accessed flag

Dirty flag

IA-32e PAGING (4K page)
Current Status: Analysis Framework

- Automatically generate and prove many lemmas about reads and writes
- Libraries to reason about (non-)interference of memory regions
- Proved general lemmas about paging data structure traversals
Outline

- Motivation
- Project Description
  - [1] Developing an x86 ISA Model
  - [3] Verifying Application and System Programs
- Future Work & Conclusion
- Accessing Source Code + Documentation
Application Program #1: *popcount*

Automatically verify snippets of straight-line machine code using **bit-blasting** [VSTTE’13]

```assembly
55 push %rbp
48 89 e5 mov %rsp,%rbp
89 7d fc mov %edi,-0x4(%rbp)
8b 7d fc mov -0x4(%rbp),%edi
8b 45 fc mov -0x4(%rbp),%eax
c1 e8 01 shr $0x1,%eax
25 55 55 55 55 and $0x55555555,%eax
29 c7 sub %eax,%edi
89 7d fc mov %edi,-0x4(%rbp)
8b 45 fc mov -0x4(%rbp),%eax
25 33 33 33 33 and $0x33333333,%eax
8b 7d fc mov -0x4(%rbp),%edi
c1 ef 02 shr $0x2,%edi
81 e7 33 33 33 33 and $0x33333333,%edi
01 f8 add %edi,%eax
89 45 fc mov %eax,-0x4(%rbp)
8b 45 fc mov -0x4(%rbp),%eax
8b 7d fc mov -0x4(%rbp),%edi
c1 ef 04 shr $0x4,%edi
01 f8 add %edi,%eax
25 0f 0f 0f 0f and $0xf0f0f0f0f,%eax
69 c0 01 01 01 01 imul $0x101010101,%eax,%eax
c1 e8 18 shr $0x18,%eax
89 45 fc mov %eax,-0x4(%rbp)
8b 45 fc mov -0x4(%rbp),%eax
5d pop %rbp
c3 retq
```
Application Program #1: *popcount*

Automatically verify snippets of straight-line machine code using **bit-blasting** [VSTTE’13]

```c
int popcount_32 (unsigned int v)
{
  // From Sean Anderson’s Bit-Twiddling Hacks
  v = v - ((v >> 1) & 0x55555555);
  v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
  v = ((v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >> 24;
  return(v);
}
```

| 8b 45 fc | mov  -0x4(%rbp),%eax |
| 25 33 33 33 33 | and  $0x33333333,%eax |
| 8b 7d fc | mov  -0x4(%rbp),%edi |
| c1 ef 02 | shr  $0x2,%edi |
| 81 e7 33 33 33 33 | and  $0x33333333,%edi |
| 01 f8 | add  %edi,%eax |
| 89 45 fc | mov  %eax,-0x4(%rbp) |
| 8b 45 fc | mov  -0x4(%rbp),%eax |
| 8b 7d fc | mov  -0x4(%rbp),%edi |
| c1 ef 04 | shr  $0x4,%edi |
| 01 f8 | add  %edi,%eax |
| 25 0f 0f 0f 0f | and  $0xf0f0f0f,%eax |
| 69 c0 01 01 01 01 | imul  $0x1010101,%eax,%eax |
| c1 e8 18 | shr  $0x18,%eax |
| 89 45 fc | mov  %eax,-0x4(%rbp) |
| 8b 45 fc | mov  -0x4(%rbp),%eax |
| 5d | pop  %rbp |
| c3 | retq |

Program Verification
Automatically verify snippets of straight-line machine code using **bit-blasting** [VSTTE’13]

```c
int popcount_32 (unsigned int v)
{
    // From Sean Anderson's Bit-Twiddling Hacks
    v = v - ((v >> 1) & 0x55555555);
    v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
    v = ((v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >> 24;
    return(v);
}
```

```assembly
8b 45 fc  mov  -0x4(%rbp),%eax
25 33 33 33 33  and  $0x33333333,%eax
8b 7d fc  mov  -0x4(%rbp),%edi
c1 ef 02  shr  $0x2,%edi
81 e7 33 33 33 33  and  $0x33333333,%edi
01 f8  add  %edi,%eax
89 45 fc  mov  %eax,-0x4(%rbp)
8b 45 fc  mov  -0x4(%rbp),%eax
8b 7d fc  mov  -0x4(%rbp),%edi
c1 ef 04  shr  $0x4,%edi
01 f8  add  %edi,%eax
25 0f 0f 0f 0f  and  $0xF0F0F0F,%eax
69 c0 01 01 01 01  imul  $0x1010101,%eax,%eax
c1 e8 18  shr  $0x18,%eax
89 45 fc  mov  %eax,-0x4(%rbp)
8b 45 fc  mov  -0x4(%rbp),%eax
5d  pop  %rbp
c3  retq
```

**specification function**

```plaintext
RAX = popcount(input)
```

**popcount(x):**

```plaintext
if (x <= 0) then
    return 0
else
    lsb := x & 1
    x := x >> 1
    return (lsb + popcount(x))
endif
```
Application Program #2: word-count

- Proved the functional correctness of a word-count program that reads input from the user using \texttt{read} system calls. System calls are \textit{non-deterministic} for application programs. [FMCAD’14]

```
55                      push   %rbp
48 89 e5                mov    %rsp,%rbp
48 83 ec 20             sub    $0x20,%rsp
C7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
C7 45 e8 00 00 00 00    movl   $0x0,-0x18(%rbp)
C7 45 ec 00 00 00 00    movl   $0x0,-0x14(%rbp)
C7 45 f0 00 00 00 00    movl   $0x0,-0x10(%rbp)
C7 45 f4 00 00 00 00    movl   $0x0,-0xc(%rbp)
E8 90 ff ff ff          callq  <_gc>
...                      ...
05 01 00 00 00          add    $0x1,%eax
89 45 f0                mov    %eax,-0x10(%rbp)
e9 00 00 00 00          jmpq   <_main+0xb8>
e9 6e ff ff ff          jmpq   <_main+0x2b>
31 c0                   xor    %eax,%eax
48 83 c4 20             add    $0x20,%rsp
5d                      pop    %rbp
c3                      retq
```
Application Program #2: word-count

- Proved the functional correctness of a word-count program that reads input from the user using \texttt{read} system calls. System calls are \textit{non-deterministic} for application programs. [FMCAD’14]

Specification for counting the # of characters in \texttt{str}:

\begin{verbatim}
ncSpec(offset, str, count):
    if (well-formed(str) && offset < len(str)) then
        c := str[offset]
        if (c == EOF) then
            return count
        else
            count := (count + 1) mod 2^32
            ncSpec(1 + offset, str, count)
        endif
    endif
\end{verbatim}

\textbf{Functional Correctness Theorem:} Values computed by specification functions on standard input are found in the expected memory locations of the final x86 state.
Other properties verified using our machine-code framework:

- **Resource Usage:**
  - Program and its stack are disjoint for all inputs.
  - Irrespective of the input, program uses a fixed amount of memory.

- **Security:**
  - Program does not modify unintended regions of memory.
System Program: zero-copy

Specification:
Copy data $x$ from virtual memory location $l0$ to disjoint linear memory location $l1$.

Verification Objective:
After a successful copy, $l0$ and $l1$ contain $x$.

Implementation:
Include the copy-on-write technique: $l0$ and $l1$ can be mapped to the same physical memory location $p$.
  - Modifications to address mapping
  - Access control management
Proved that the **implementation of a zero-copy program** meets the **specification of a simple copy operation**.

For simplicity, marking of paging structures during their traversal was turned off, i.e., no accessed and dirty bit updates were allowed for this proof.

We are currently porting this proof over to a more accurate x86 model, which characterizes updates to accessed and dirty bits as well.
Outline

- Motivation

- Project Description
  - [1] Developing an x86 ISA Model
  - [3] Verifying Application and System Programs

- Future Work & Conclusion

- Accessing Source Code + Documentation
Future Work

- **Run a 64-bit FreeBSD kernel on our x86 ISA model**
  - This involves identifying and implementing relevant instructions, call gates, supporting task management, etc.

- **Develop lemma libraries to reason about kernel code**
  - This involves developing automated reasoning infrastructure for page table walks, access rights, etc.

- **Identify and prove critical invariants in kernel code**
  - This includes proving the correctness of context switches, privilege escalations, etc.

We look forward to collaborating with the FreeBSD® community!
It is essential to state and prove properties related to behavior, security, and resource usage; bug-hunting can only take us so far. This task is **within the scope of mechanized theorem proving**, as is evidenced by its use by our collaborators in the government and the industry to prove complex properties about complex systems.

Although full verification of all software is the ultimate goal, the focus for the coming years is to create *islands of trust*, i.e., parts of the system for which complex properties have been formally verified.
The x86isa project is available under **BSD 3-Clause license** as a part of the **ACL2 Community Books** project.

Go to [https://github.com/acl2/acl2/](https://github.com/acl2/acl2/) and see books/projects/x86isa/README for details.

Also, documentation and user’s manual is available online at [www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2_____X86ISA](http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2_____X86ISA)
Some Publications

- **Shilpi Goel, Warren A. Hunt, Jr., and Matt Kaufmann.**
  Abstract Stobjs and Their Application to ISA Modeling
  In ACL2 Workshop, 2013

- **Shilpi Goel and Warren A. Hunt, Jr.**
  Automated Code Proofs on a Formal Model of the x86
  In Verified Software: Theories, Tools, Experiments (VSTTE), 2013

- **Shilpi Goel, Warren A. Hunt, Jr., Matt Kaufmann, and Soumava Ghosh.**
  Simulation and Formal Verification of x86 Machine-Code Programs That Make System Calls
  In Formal Methods in Computer-Aided Design (FMCAD), 2014

- **Shilpi Goel, Warren A. Hunt, Jr., and Matt Kaufmann.**
  Engineering a Formal, Executable x86 ISA Simulator for Software Verification
  To appear in Provably Correct Software (ProCoS), 2015
Extra Slides
<table>
<thead>
<tr>
<th>Programmer-level Mode</th>
<th>System-level Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verification of <em>application</em> programs</td>
<td>Verification of <em>system</em> programs</td>
</tr>
<tr>
<td><em>Linear</em> memory address space (2⁶⁴ bytes)</td>
<td><em>Physical</em> memory address space (2⁵² bytes)</td>
</tr>
<tr>
<td><em>Assumptions</em> about correctness of OS operations</td>
<td><em>No assumptions</em> about OS operations</td>
</tr>
</tbody>
</table>
Motivation: x86 Machine-Code Verification

- **Why not high-level code verification?**
  - ✗ High-level verification frameworks do not address compiler bugs
    - ✓ Verified/verifying compilers can help
    - ✗ But these compilers typically generate inefficient code
  - ✗ Need to build verification frameworks for many high-level languages
  - ✗ Sometimes, high-level code is unavailable

- **Why x86?**
  - ✓ x86 is in widespread use — our approach will have immediate practical application
Building a Lemma Database

Three kinds of theorems:
- Read-over-Write Theorems
- Write-over-Write Theorems
- Preservation Theorems
Read-over-Write Theorem: #1

non-interference

Program Order

memory

i

j

y
Read-over-Write Theorem: #1

non-interference

Program Order

i

j

x

y

W_i(x)

memory
Read-over-Write Theorem: #1

Program Order

\[
\begin{array}{c}
\text{non-interference} \\
\hline
\text{x} & \text{y} \\
\end{array}
\]

\[
W_i(x) \quad R_j: y
\]

memory
Read-over-Write Theorem: #2

Program Order

overlap

memory

Task 2 | Machine-Code Analysis Framework | Lemma Database
Read-over-Write Theorem: #2

Program Order

overlap

memory

Program Order

\[ W_i(x) \]

Task 2 | Machine-Code Analysis Framework | Lemma Database
Read-over-Write Theorem: #2

Program Order

\[ W_i(x) \]

\[ R_i: x \]

\[ \text{overlap} \]

\[ x \]

\[ i \]

memory
Write-over-Write Theorem: #1

Program Order

independent writes commute safely

memory
Write-over-Write Theorem: #1

Program Order

independent writes commute safely

\[ W_i(x) \]

memory

Task 2 | Machine-Code Analysis Framework | Lemma Database
Write-over-Write Theorem: #1

Program Order

independent writes commute safely

\[ W_i(x) \]  \[ W_j(y) \]

memory
Write-over-Write Theorem: #1

independent writes commute safely

Program Order

\[ W_i(x) \]

Program Order

\[ W_j(y) \]

Program Order

\[ = \]

Program Order

\[ = \]

Program Order

\[ = \]
Write-over-Write Theorem: #1

Program Order

independent writes commute safely

Program Order
Write-over-Write Theorem: #1

Program Order

independent writes commute safely

Program Order

memory

\[ W_i(x) \]
\[ W_j(y) \]

memory

\[ W_j(y) \]
\[ W_i(x) \]

Task 2 | Machine-Code Analysis Framework | Lemma Database
Write-over-Write Theorem: #2

Program Order

visibility of writes

memory
Write-over-Write Theorem: #2

Program Order

visibility of writes

\( W_i(x) \)

memory
Write-over-Write Theorem: #2

Program Order

visibility of writes

memory

Program Order

$W_i(x)$

$W_i(y)$
Write-over-Write Theorem: #2

Program Order

visibility of writes

Program Order

$W_i(x)$

$W_i(y)$

$=$

Program Order
Write-over-Write Theorem: #2

Program Order

Program Order

visibility of writes

\[ W_i(x) \]

\[ W_i(y) \]

\[ = \]

\[ W_i(y) \]
Preservation Theorems

reading from a valid x86 state

valid-address-p(i) \land
valid-x86-p(x86)
⇒
valid-value-p(R_{i}:x) \land
valid-x86-p(x86)
Preservation Theorems

**reading from a valid x86 state**

\[
\begin{align*}
&\text{valid-address-p}(i) \land \\
&\text{valid-x86-p}(x86) \\
\Rightarrow &\quad \text{valid-value-p}(R_i_x) \land \\
&\text{valid-x86-p}(x86)
\end{align*}
\]

**writing to a valid x86 state**

\[
\begin{align*}
&\text{valid-address-p}(i) \land \\
&\text{valid-value-p}(x) \land \\
&\text{valid-x86-p}(x86) \\
\Rightarrow &\quad \text{valid-x86-p}(W_i(x))
\end{align*}
\]
Verification Effort vs. Verification Utility

System-level Mode

User Space (Ring 3)
- MOV %rax, 3
- SYSCALL
- MOV %rbx, %rax
- ...

Kernel Space (Ring 0)
- ...
- SYSRET
- ...

Programmer-level Mode

FreeBSD read system call semantics

save user state
restore user state
Programmer-level Mode: Model Validation

Task A: Validate the logical mode against the execution mode

Task B: Validate the execution mode against the processor + system call service provided by the OS
Programmer-level Mode: Execution Mode

A Common Lisp Distribution

ACL2

x86 ISA Model

Operating System

SYSCALL

Return
Programmer-level Mode: Execution and Reasoning

Execution Mode

$x_0$ \arrow{x_0}{x_1}

ENV \arrow{ENV}{ENV'}

Logical Mode

$x_0$ \arrow{x_0}{x_1}

env \arrow{env}{env'}
Verification Tools:

- **Automatic**
  - Limited analysis capabilities
  - Static & dynamic analyzers
  - State explosion problem
  - Model checkers
  - SAT & SMT solvers

- **Interactive**
  - High degree of manual effort
  - Can be applied to large systems
  - Interactive theorem provers coupled with automatic tools
  - Interactive theorem provers