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

Shilpi Goel

The University of Texas at Austin
Ph.D. Dissertation Proposal

29th April, 2015
Outline

- Motivation
- State of the Art
- Proposed Dissertation Project
  - [Task 1] Developing an x86 ISA Model
  - [Task 3] Verifying Application and System Programs
- Future Work
- Expected Contributions
Motivation

• Software systems are ubiquitous.

• Cost of incorrect software is extremely high.

• *Formal verification* can increase software quality.

• Approach: **Machine-code verification for x86 platforms**
Motivation

• 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
## State of the Art: x86 Machine-Code Analysis

<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Morrisett et al.</td>
<td>Formal x86 ISA specification</td>
<td>Security policy violation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>checks in browsers</td>
</tr>
<tr>
<td>Reps et al.</td>
<td>Control &amp; data flow analysis</td>
<td>Point tools; approximation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>of program behavior</td>
</tr>
<tr>
<td>Myreen</td>
<td>Decompilation-in-Logic</td>
<td>Analysis of application</td>
</tr>
<tr>
<td></td>
<td>Codewalker</td>
<td>programs</td>
</tr>
<tr>
<td>Klein et al.</td>
<td>SeL4</td>
<td>Clean-slate development of</td>
</tr>
<tr>
<td>Shao et al.</td>
<td>Certified OS kernels</td>
<td>verified software</td>
</tr>
<tr>
<td>We need more!</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>General analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precisely capture program behavior</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analyze application and system programs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Verify existing software</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Security policy violation checks in browsers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point tools; approximation of program behavior</td>
</tr>
<tr>
<td>Analysis of application programs</td>
</tr>
<tr>
<td>Clean-slate development of verified software</td>
</tr>
</tbody>
</table>
Example: Analysis of a Data-Copy Program

Specification:
Copy data $x$ from linear (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 $p$.
- System calls
- Modifications to address mapping
- Access control management
Proposed Dissertation Project

- **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, and
  3. Employ this framework to verify application and system programs.
Expected Contributions

Briefly:

- **A new tool:**
  general-purpose analysis framework for x86 machine-code

- **Program verification taking memory management into account:**
  analysis of programs, including low-level system & ISA features

- **Reasoning strategies:**
  insight into low-level code verification in general

- **Foundation for future research:**
  target for verified/verifying compilers
Outline

Motivation

State of the Art

Proposed Dissertation Project

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

Future Work

Expected Contributions
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

AMD64 Technology

AMD64 Architecture Programmer’s Manual

Volume 3: General-Purpose and System Instructions

All AMD manuals: ~3000 pages

~3400 pages

Task 1 | x86 ISA Model | Model Development

Running tests on x86 machines
Model Development

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

- **Basic Program Execution Registers**
  - Sixteen 64-bit Registers
  - General-Purpose Registers
  - Six 16-bit Registers
  - Segment Registers
  - RFLAGS Register
  - RIP (Instruction Pointer Register)

- **FPU Registers**
  - Eight 80-bit Registers
  - Floating-Point Data Registers
  - Control Register
  - Status Register
  - Tag Register
  - Opcode Register (11-bits)
  - FPU Instruction Pointer Register
  - FPU Data (Operand) Pointer Register

- **MMX Registers**
  - Eight 64-bit Registers
  - MMX Registers

- **XMM Registers**
  - Sixteen 128-bit Registers
  - XMM Registers
  - 32-bits
  - MXCSR Register

---

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

**Basic Program Execution Registers**
- Sixteen 64-bit Registers
- General-Purpose Registers
- Six 16-bit Registers
- 64-bits
- Segment Registers
- RFLAGS Register
- RIP (Instruction Pointer Register)

**FPU Registers**
- Eight 80-bit Registers
- Floating-Point Data Registers
- 16 bits
- Control Register
- Status Register
- Tag Register
- Opcode Register
- FPU Instruction Register
- FPU Data (Operand Register)
- 64 bits
- 64 bits

**MMX Registers**
- Eight 64-bit Registers
- MMX Registers

**XMM Registers**
- Sixteen 128-bit Registers
- 32-bits
- MXCSR Register

**Model Development**

**Task 1 | x86 ISA Model | Model Development**

Source: Intel Manuals
Model Development

64-bit sub-mode

Basic Program Execution Registers

- Sixteen 64-bit Registers
- General-Purpose Registers
- Sixteen 64-bit Registers
- Segment Registers
- 64-bits
- RIP (Instruction Pointer Register)
- 64-bits
- RFLAGS Register
- 64-bits
- Privilege Check

PU Registers

- Eight 80-bit Registers
- Floating-Point Data Registers
- 16 bits
- Control Register
- 16 bits
- Status Register
- 16 bits
- Tag Register
- 64 bits
- Opcode Register
- 64 bits
- FPU Instruction
- 64 bits
- FPU Data (Oper

MMX Registers

- Eight 64-bit Registers

XMM Registers

- Sixteen 128-bit Registers

MXCSR Register

Figure 3-2. 64-Bit Mode Execution Environment

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

Task 1 | x86 ISA Model | Model Development

Source: Intel Manuals
The procedure in code segment C is not able to access register is not loaded. The processor loads the segment selector into a segment register, it performs a privilege check (see Figure 5.8.4, "Accessing a Code Segment Through a Call Procedure") for the segment. See Section 5.10.4, "Checking Caller Privilege Levels" to determine if access to a segment is allowed. Even if the program or task requesting access to a segment has privilege levels that are both numerically lower than (more privileged) than the DPL of the segment, a general-protection fault is generated and the segment selector for the data segment must be loaded into the data-segment register (IDTR) expand to 10 bytes so that they can hold the segment selector for the data segment. For example, if the DPL of a conforming code segment and the RPL of segment selector E1 are equal to the DPL of data segment E, the processor checks the DPL of the interrupt or trap gate. When the currently executing code segment is different in the following ways:

- The DPL is the privilege level of a segment or gate.
- The DPL indicates the numerically highest privilege level that the currently executing program or task can have to be allowed to access a program or task.
- The constraint prevents application programs or procedures from overwriting segment selectors. It is stored in bits 0 and 1 of the segment register (CS and SS).
- The processor uses privilege levels to prevent a program or procedure from accessing data in a data segment that is different in the following ways:
  - The DPL indicates the numerically lowest privilege level that the program or task requesting access to a segment has. Privilege-level protection for exception- and interrupt-handling procedures is different in the following ways:
    - The DPL indicates the numerically highest privilege level that the processor ignores the DPL of interrupt and trap gates.
    - The DPL indicates the numerically lowest privilege level that a program or task can have to be allowed to access a program or task.

To access operands in a data segment, the processor uses privilege levels to prevent a program or procedure from accessing data in a data segment that is different in the following ways:

- The DPL indicates the numerically highest privilege level that the currently executing program or task has to be allowed to access a program or task.
- The DPL indicates the numerically lowest privilege level that the currently executing program or task has to be allowed to access a program or task.

- The DPL is not checked on implicit calls to exception and interrupts.
- The DPL is not checked on implicit calls to exception and interrupts.

The privilege level of a segment or gate is determined from the DPL and RPL and is stored in bits 0 and 1 of the segment register (CS and SS). For example, if the DPL of a conforming code segment and the RPL of segment selector E1 are both numerically lower than (more privileged) than the DPL of code segment A and the RPL of segment selector E2 to data segment E using segment selector E3 (dotted line), because the CPL is not changed when the processor accesses a segment or gate, the DPL of the segment or gate is used when the currently executing code segment attempts to access a segment or gate. When the currently executing code segment attempts to access a segment or gate, the DPL of the segment or gate is used when the currently executing code segment attempts to access a segment or gate.

Task 1 | x86 ISA Model | Model Development

Source: Intel Manuals
64-bit sub-mode

asic Program Execution Registers

Sixteen 64-bit Registers

General-Purpose Registers

Segment Registers

RFLAGS Register

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)

Interrupt Vector

Task-State Segment (TSS)

Stack

Code

NULL

Interrupt Handler

Exception Handler

Protected Procedure

Segment Sel.

Interrupt

Page

Physical Address

Segment Selector

Register

Task Register

Global Descriptor Table (GDT)
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 100+ instructions (~220 opcodes)
  - Can execute almost all user-level programs emitted by GCC/LLVM
  - Successfully co-simulated a contemporary SAT solver on our model
- IA-32e paging for all page configurations (4K, 2M, 1G)
- Segment-based addressing
- Privileged instructions and system state
- Simulation speed*: 
  - ~3.3 million instructions/second (paging disabled)
  - ~330,000 instructions/second (with 1G pages)

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

- Motivation
- State of the Art
- Proposed Dissertation Project
  - [Task 1] Developing an x86 ISA Model
  - [Task 3] Verifying Application and System Programs
- Future Work
- Expected Contributions
Semantics of the program is given by the effect it has on the machine state.

Need to reason about:
- Reads from machine state
- Writes to machine state

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

non-interference

Program Order

| W_i(x) | R_j: y |

memory
Read-over-Write Theorem: #2

Program Order

overlap

\[ \text{memory} \]

\[ \text{W}_i(x) \]

\[ \text{R}_i: x \]
Write-over-Write Theorem: #1

Program Order

independent writes commute safely

Program Order
Write-over-Write Theorem: #2

Program Order

visibility of writes

Program Order

memory

| W_i(x) |
| W_i(y) |

= i

memory
## Preservation Theorems

### Reading from a valid x86 state

\[
\text{valid-address-p}(i) \land \
\text{valid-x86-p}(x86) \implies \
\text{valid-value-p}(\textcolor{green}{R_i:x}) \land \
\text{valid-x86-p}(x86)
\]

### Writing to a valid x86 state

\[
\text{valid-address-p}(i) \land \
\text{valid-value-p}(x) \land \
\text{valid-x86-p}(x86) \implies \
\text{valid-x86-p}(\textcolor{red}{W_i(x)})
\]
Address translations for \( R_i: x \) and \( W_i(x) \)

**Segmentation**

- Logical Address
- Segment Selector
- Offset
- Descriptor Table(s)
- Segment Descriptor
- Linear Address
- 4K Page
- Segment
- Linear Memory
- Global or Local Descriptor Table Register
- Control Register

**IA-32e Paging (4K page)**

- Control Register has the base address of these structures.
- \( a \): accessed flag
- \( d \): dirty flag
- PML4
- PML4E
- PDPTE
- PDE
- PTE
- Physical Addr.
- 4K Page
Current Status: Analysis Framework

- Automatically generate and prove:
  - Read-over-Write theorems
  - Write-over-Write theorems
  - Preservation theorems
- Libraries to reason about (non-)interference of memory regions
- Predicates that recognize valid paging structure entries
- Proved some properties about paging data structure traversals
Outline

๏ Motivation
๏ State of the Art
๏ Proposed Dissertation Project
  ‣ [Task 1] Developing an x86 ISA Model
  ‣ [Task 3] Verifying Application and System Programs
๏ Future Work
๏ Expected Contributions
Verification Effort vs. Verification Utility

**Programmer-level Mode**

- Verification of *application* programs
- *Linear* memory address space ($2^{64}$ bytes)
- *Assumptions* about correctness of OS operations

**System-level Mode**

- Verification of *system* programs
- *Physical* memory address space ($2^{52}$ bytes)
- *No assumptions* about OS operations
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
Application Program #1: \textit{popcount}

Automatically verify snippets of straight-line machine code using symbolic simulation \[\text{VSTTE'13}\]

\begin{verbatim}
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 0f    and $0x0f0f0f0f,%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
\end{verbatim}

RAX = popcount(input)

\begin{verbatim}
unsigned int popcount(x):
    if (x <= 0) then
        return 0
    else
        lsb := x & 1
        x := x >> 1
        return (lsb + popcount(x))
    endif
\end{verbatim}
Application Program #2: word-count

- Proved the functional correctness of a word-count program that reads input from the user using `read` system calls [FMCAD’14]

- Interesting; system calls are **non-deterministic** for application programs

Specification for counting the characters in str:

```plaintext
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
```

**Functional Correctness Theorem:** Values computed by specification functions on standard input are found in the expected memory locations of the final x86 state.
Application Program #2: word-count

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.
Future Work

[Task 3]: System Program Verification:
The optimized data-copy program that accesses and manipulates x86 memory-management data structures

- In support of [Task 3], we will continue to:
  - [Task 1]: Model additional features of x86 ISA
  - [Task 2]: Reason about reads from, writes to, and non-interference of x86 memory-management data structures
Overview: Current Status

<table>
<thead>
<tr>
<th>Tasks</th>
<th>% Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Task 1] x86 ISA Model</td>
<td>90%</td>
</tr>
<tr>
<td>[Task 3] Program Verification</td>
<td>40%</td>
</tr>
</tbody>
</table>

Comprehensive documentation and user’s manual

---

**x86isa**

/Users/shigoel/Desktop/x86ISA/top.lisp

x86 ISA model and machine-code analysis framework developed at UT Austin

**Subtopics**

- **Dev-philosophy**
  Notes on the development style of the x86ISA model

- **Globally-disabled-events**
  A ruleset containing all the events supposed to be mostly globally disabled in our books

- **Utils**
  The books in this directory provide some supporting events for the rest of the books in x86ISA.

- **Machine**
  The books in this directory define the core elements of the x86ISA, like the x86 state, decoder function, etc. Also included are proofs about the specification.

- **Proof-utilities**
  Basic utilities for x86 machine-code proofs

- **Execution**
  Setting up the x86 ISA model for a program run
Expected 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
Publications

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


Thanks!
Extra Slides
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
Programmer-level Mode: Execution and Reasoning

Execution Mode

$x_0$ \(\rightarrow\) $x_1$

ENV \(\rightarrow\) ENV'

Logical Mode

$x_0$ env \(\rightarrow\) $x_1$ env'

ENV \(\rightarrow\) ENV'
Verification Landscape

Verification Tools:

- **Static & dynamic analyzers**
  - limited analysis capabilities

- **Model checkers**
  - state explosion problem

- **SAT & SMT solvers**

- **Interactive theorem provers coupled with automatic tools**

- **Interactive theorem provers**
  - high degree of manual effort
  - can be applied to large systems