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

Shilpi Goel

The University of Texas at Austin
Ph.D. Dissertation Proposal
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?**
  - X High-level verification frameworks do not address compiler bugs
  - ✓ Verified/verifying compilers can help
    - X But these compilers typically generate inefficient code
  - X Need to build verification frameworks for many high-level languages
  - X Sometimes, high-level code is unavailable
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
| [Morrisett et al.] Formal x86 ISA specification | Security policy violation checks in browsers |
## State of the Art: x86 Machine-Code Analysis

<table>
<thead>
<tr>
<th>[Morrisett et al.]</th>
<th>Formal x86 ISA specification</th>
<th>Security policy violation checks in browsers</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Reps et al.]</td>
<td>Control &amp; data flow analysis</td>
<td>Point tools; approximation of program behavior</td>
</tr>
<tr>
<td>Reference</td>
<td>Description</td>
<td>Application</td>
</tr>
<tr>
<td>----------------------------</td>
<td>--------------------------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>[Morrisett et al.]</td>
<td>Formal x86 ISA specification</td>
<td>Security policy violation checks in browsers</td>
</tr>
<tr>
<td>[Reps et al.]</td>
<td>Control &amp; data flow analysis</td>
<td>Point tools; approximation of program behavior</td>
</tr>
<tr>
<td>[Myreen]</td>
<td>Decompilation-in-Logic</td>
<td>Analysis of application programs</td>
</tr>
<tr>
<td>[Moore]</td>
<td>Codewalker</td>
<td></td>
</tr>
</tbody>
</table>
## State of the Art: x86 Machine-Code Analysis

<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Morrisett et al.</td>
<td>Formal x86 ISA specification</td>
<td>Security policy violation checks in browsers</td>
</tr>
<tr>
<td>Reps et al.</td>
<td>Control &amp; data flow analysis</td>
<td>Point tools; approximation of program behavior</td>
</tr>
<tr>
<td>Myreen</td>
<td>Decompilation-in-Logic</td>
<td>Analysis of application programs</td>
</tr>
<tr>
<td>Moore</td>
<td>Codewalker</td>
<td></td>
</tr>
<tr>
<td>Klein et al.</td>
<td>SeL4</td>
<td>Clean-slate development of verified software</td>
</tr>
<tr>
<td>Shao et al.</td>
<td>Certified OS kernels</td>
<td></td>
</tr>
</tbody>
</table>
State of the Art: x86 Machine-Code Analysis

We need more!

- General analysis
- Precisely capture program behavior
- Analyze application and system programs
- Verify existing software

<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
Example: Analysis of a Data-Copy Program

Specification:
Copy data $x$ from linear (virtual) memory location $l0$ to disjoint linear memory location $l1$. 
Example: Analysis of a Data-Copy Program

**Specification:**
Copy data \( x \) from linear (virtual) memory location \( l_0 \) to disjoint linear memory location \( l_1 \).
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 $x$. 
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
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 %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"

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

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

Figure 3-2. 64-Bit Mode Execution Environment
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
  - Control Register
  - Status Register
  - Tag Register
  - Opcode Register
  - FPU Instruction Data (Operand) Pointer Register
  - 64 bits
  - 64 bits

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

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

---

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

**Figure 3-2. 64-Bit Mode Execution Environment**

Source: Intel Manuals
64-bit sub-mode

Basic Program Execution Registers

- Sixteen 64-bit Registers
- General-Purpose Registers
  - Six 16-bit Registers
  - Segment Registers
  - 64-bits: RFLAGS Register
  - RIP (Instruction Pointer Register)

CPU Registers

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

MMX Registers

- Eight 64-bit Registers

XMM Registers

- Sixteen 128-bit Registers

Source: Intel Manuals
64-bit sub-mode

asic Program Execution Registers

<table>
<thead>
<tr>
<th>Sixteen 64-bit Registers</th>
<th>General-Purpose Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment Registers</td>
<td></td>
</tr>
<tr>
<td>64-bits</td>
<td></td>
</tr>
<tr>
<td>RIP (Instruction Pointer Register)</td>
<td></td>
</tr>
<tr>
<td>64-bits</td>
<td></td>
</tr>
</tbody>
</table>

IU Registers

<table>
<thead>
<tr>
<th>Eight 80-bit Registers</th>
</tr>
</thead>
</table>

Floating-Point Data Registers

<table>
<thead>
<tr>
<th>Control Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Register</td>
</tr>
<tr>
<td>Tag Register</td>
</tr>
<tr>
<td>Opcode Registe</td>
</tr>
<tr>
<td>FPU Instruction</td>
</tr>
<tr>
<td>FPU Data (Oper</td>
</tr>
<tr>
<td>16 bits</td>
</tr>
<tr>
<td>16 bits</td>
</tr>
<tr>
<td>16 bits</td>
</tr>
</tbody>
</table>

MMX Registers

| 16 bits |
| 16 bits |
| 8 bits |

XCR0 (XFEM) |

Protected Procedure |

Interrupt Vector |

Interrupt Handler |

Inter. Handler |

Exception Handler |

Global Descriptor Table (GDT) |

Interrupt Descriptor Table (IDT) |

Local Descriptor Table (LDT) |

Current TSS |

Call Gate |

Call-Gate Segment Selector |

Task Register |

Segment Selector |

Segment Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |

Seg. Desc. |
The procedure in code segment A is able to access data segment E using segment selector E1, because the CPL descriptor. The processor loads the segment selector into the data segment register if the DPL is numerically greater (less privileged) than the DPL of data segment E. Even if a code segment A procedure were to use segment selector E1, the processor checks the RPL along with the CPL to ensure that segment. See Section 5.10.4, "Checking Caller Privilege-Level Change.

When accessing conforming code segments, outer rings are used for less critical software. (Systems that usually the kernel of an operating system. The center (reserved for the most privileged code, data, and software, usually the kernel of an operating system. Outer rings are used for less critical software. (Systems that...}

A, B, C, and D), each ru...
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
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]

![Diagram showing layered modeling approach with x86 interpreter, Abstract Processor State, and Concrete Processor State. The diagram illustrates the correspondence and supports relationships between these states, optimized for reasoning and execution efficiency.]

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

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

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

```
add %edi, %eax
je 0x400304
```
Semantics of the program is given by the effect it has on the machine state.

```
add %edi, %eax
je 0x400304
```

1. read instruction from mem
2. read operands
3. write sum to eax
4. write new value to flags
5. write new value to pc
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

1. read instruction from mem
2. read operands
3. write sum to eax
4. write new value to flags
5. write new value to pc

```assembly
add %edi, %eax
je 0x400304
```
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

Program Order

non-interference

\[ i \quad y \quad j \]

memory
Read-over-Write Theorem: #1

non-interference

Program Order

\[ W_i(x) \]

\[ x \quad y \]

i \quad j

memory
Read-over-Write Theorem: #1

Program Order

\[ W_i(x) \]

\[ R_j: y \]

non-interference

memory
Read-over-Write Theorem: #2

Program Order

overlap

memory

i
Read-over-Write Theorem: #2

Program Order

overlap

\( W_i(x) \)

memory
Read-over-Write Theorem: #2

Program Order

overlap

i

x

$W_i(x)$

$R_i: x$

memory
Write-over-Write Theorem: #1

Program Order

independent writes commute safely

Program Order

memory

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

Program Order

independent writes commute safely

\[ W_i(x) \]

\[ W_j(x) \]
Write-over-Write Theorem: #1

Program Order

independent writes commute safely

\[ W_i(x) \]

\[ W_j(y) \]
Write-over-Write Theorem: #1

**Independent writes commute safely**

Program Order

\[ W_i(x) \]

\[ W_j(y) \]

Program Order

\[ = \]

\[ i \quad j \]

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

Program Order

independent writes commute safely

Program Order
Write-over-Write Theorem: #1

**Independent writes commute safely**

Program Order

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

Program Order

\[
W_i(x) \quad x \quad y \quad W_j(y)
\]
Write-over-Write Theorem: #2

Program Order

visibility of writes

memory

Program Order
Write-over-Write Theorem: #2

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |

Program Order

visibility of writes

| memory |
Write-over-Write Theorem: #2

Program Order

visibility of writes

Program Order

memory

W_i(x)

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

Program Order

visibility of writes

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

Program Order
Write-over-Write Theorem: #2

Program Order

visibility of writes

Program Order

Program Order

Program Order

Program Order
reading from a valid x86 state

\[
\text{valid-address-p}(i) \land \\
\text{valid-x86-p}(x86) \Rightarrow \\
\text{valid-value-p}(R_i : x) \land \\
\text{valid-x86-p}(x86)
\]
Preservation Theorems

**reading from a valid x86 state**

\[
\begin{align*}
\text{valid-address-p}(i) & \land \\
\text{valid-x86-p}(x86) & \\
\implies & \\
\text{valid-value-p}(R_i : x) & \land \\
\text{valid-x86-p}(x86) &
\end{align*}
\]

**writing to a valid x86 state**

\[
\begin{align*}
\text{valid-address-p}(i) & \land \\
\text{valid-value-p}(x) & \land \\
\text{valid-x86-p}(x86) & \\
\implies & \\
\text{valid-x86-p}(W_i(x)) &
\end{align*}
\]
Address translations for $R_i: x$ $W_i(x)$
Address translations for $R_i: x$, $W_i(x)$
Address translations for $R_i: x$ $W_i(x)$

SEGMENTATION
Address translations for $R_i: x \quad W_i(x)$

SEGMENTATION
Address translations for $R_i: x$ $W_i(x)$

SEGMENTATION

IA-32e PAGING (4K page)
Address translations for $R_i: x \text{ and } W_i(x)$

Logical Address → Segment Selector → Offset → Descriptor Table(s) → Segment Descriptor → Linear Addr. → Segment → Linear Memory

Control Register
- CR3: has the base address of these structures.
- CR4: other control information

Physical Memory
- PML4 E → Dir. Ptr. → Dir. → Table → Offset

Global or Local Descriptor Table Register

Segmentation

IA-32e Paging (4K page)
Address translations for \( R_i : x \) \( W_i(x) \)

Logical Address

Segment Selector

Offset

Descriptor Table(s)

Segment Descriptor

Linear Memory

Global or Local Descriptor Table Register

Linear Address

Segment

Linear Addr.

Control Register

has the base address
of these structures.

PML4E

PML4

Dir. Ptr.

Dir.

Table

Offset

Physical Memory

PDPTE

CR3

IA-32e PAGING (4K page)
Address translations for $R_i: x \ W_i(x)$

Logical Address

Segment Selector

Offset

Descriptor Table(s)

Segment Descriptor

Segment

Global or Local Descriptor Table Register

Linear Memory

Linear Address

PML4

Dir. Ptr.

Dir.

Table

Offset

PDE

PDPT

PML4

Dir. Ptr.

Dir.

Table

Offset

CR3

Physical Memory

Control Register has the base address of these structures.

SEGMENTATION
Address translations for $R_i: x \rightarrow W_i(x)$

Logical Address

Segment Selector

Offset

Descriptor Table(s)

Segment Descriptor

Linear Addr.

Segment

Linear Memory

Global or Local Descriptor Table Register

CR3

PML4E

PML4

Dir. Ptr.

Dir.

Table

Offset

PDE

PDPTE

PTE

Physical Memory

Control Register has the base address of these structures.

IA-32e PAGING (4K page)
Address translations for \( R_i: x \) and \( W_i(x) \)

- Logical Address
  - Segment Selector
  - Offset

- Descriptor Table(s)

- Global or Local Descriptor Table Register

- Linear Memory
  - Segment Descriptor
  - Linear Addr.
  - 4K Page
  - Segment

- Control Register has the base address of these structures.

### SEGMENTATION

- \( R_i: x \):
  - Linear Address
  - PML4
  - Dir. Ptr.
  - Dir. Table
  - Offset
  - PTE
  - Physical Addr.
  - 4K Page

- \( W_i(x) \):
  - Linear Memory
  - PML4E
  - PDPTE
  - PDE
  - Physical Memory
  - 4K Page

---

**IA-32e PAGING (4K page)**
Address translations for $R_i: x$ $W_i(x)$

- 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.

IA-32e PAGING (4K page)

- Linear Address
- PML4
- Dir. Ptr.
- Dir.
- Table
- Offset
- PDE
- PDPTE
- PML4E
- PTE
- Physical Addr.
- 4K Page

Accessed flag
Address translations for $R_i: x$ $W_i(x)$

Logical Address → Segment Selector → Offset → Descriptor Table(s) → Segment Descriptor → Linear Addr. → 4K Page → Segment → Linear Memory

Control Register has the base address of these structures.

Segment Selector

descriptor Table(s)

Segment Descriptor

Linear Memory

Global or Local Descriptor Table Register

Linear Address

PML4

Dir. Ptr.

Dir.

Table

Offset

PML4E

PDPTE

PDE

PTE

Physical Addr.

Physical Memory

4K Page

Control Register

accessed flag

dirty flag

IA-32e PAGING (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

<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 ($2^{64}$ bytes)</td>
<td><em>Physical</em> memory address space ($2^{52}$ 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>

Task 3 | Program Verification | Effort vs. Utility
Verification Effort vs. Verification Utility

**Task 3 | Program Verification | Effort vs. Utility**

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

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

**System-level Mode**

**FreeBSD read system call semantics**

**Programmer-level Mode**

- save user state
- restore user state
Application Program #1: \textit{popcount}

Automatically verify snippets of straight-line machine code using symbolic simulation [VSTTE’13]
Application Program #1: *popcount*

Automatically verify snippets of straight-line machine code using symbolic simulation [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
8b 3f 02            shr    $0x2,%edi
81 e7 33 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
01 f8               add    %edi,%eax
25 0f 0f 0f 0f      and    $0xf0f0f0f0f,%eax
69 c0 01 01 01 01   imul   $0x101010101,%eax,%eax
8b 1e 18            shr    $0x18,%eax
89 45 fc            mov    %eax,-0x4(%rbp)
8b 45 fc            mov    -0x4(%rbp),%eax
5d                 pop    %rbp
00 c3               retq
```
Application Program #1: popcount

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

```
55                 push   %rbp
48 89 e5           mov    %rsp,%rbp
89 7d fc           mov    %edi,-0x4(%rbp)
8b 7d fc           mov    -0x4(%rbp),%edi
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   $0x10101010,%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
```

RAX = popcount(input)

unsigned int

specification function
Application Program #1: popcount

Automatically verify snippets of straight-line machine code using symbolic simulation [VSTTE’13]

```
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
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
25 ef 02           shr    %eax,%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
01 ef 04           shr    %eax,%edi
01 f8              add    %edi,%eax
25 0f 0f 0f 0f     and    $0x0f0f0f0f,%eax
69 c0 01 01 01 01  imul   $0x1010101,%eax,%eax
01 e8 18           shr    %eax,%eax
89 45 fc           mov    %eax,-0x4(%rbp)
8b 45 fc           mov    -0x4(%rbp),%eax
5d                 pop    %rbp

RAX = popcount(input)
```

signature function

```
unsigned int

popcount(x):

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 `read` system calls [FMCAD’14]
- Interesting; system calls are *non-deterministic* for application programs
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
```
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.
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>
# 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
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
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
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 Stobjs 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

Logical Mode:
- $x_0 \leftarrow env \rightarrow x_1$
  - ENV'
Verification Landscape
Verification Landscape

Verification Tools:

Static & dynamic analyzers
Model checkers
SAT & SMT solvers
Interactive theorem provers

Automatic
Interactive
Verification Tools:

- Static & dynamic analyzers
- SAT & SMT solvers
- Model checkers
- Interactive theorem provers

Verification Landscape

limited analysis capabilities
Verification Tools:

- Static & dynamic analyzers
- SAT & SMT solvers
- Model checkers
- State explosion problem
- Interactive theorem provers

Limited analysis capabilities
Verification Tools:

Static & dynamic analyzers
- Model checkers
- SAT & SMT solvers

Automatic

State explosion problem
- Limited analysis capabilities

Interactive
- Interactive theorem provers
- Can be applied to large systems
Verification Landscape

Verification Tools:

- Static & dynamic analyzers
- Model checkers
- SAT & SMT solvers
- State explosion problem

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

- Limited analysis capabilities

Automatic to Interactive
Verification Tools:

- **Automatic**
  - Limited analysis capabilities
  - Static & dynamic analyzers
  - 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