Formal Verification of Application and System Programs Based on a Validated x86 ISA Model

Ph.D. Final Defense

Shilpi Goel
shigoel@cs.utexas.edu
Department of Computer Science
The University of Texas at Austin
Can we rely on our software systems?

Recent example of a serious bug:

**CVE-2016-5195 or “Dirty COW”**

- Privilege escalation vulnerability in Linux
- E.g.: allowed a user to write to files intended to be read only
- Copy-on-Write (COW) breakage of private read-only memory mappings
- Existed since around v2.6.22 (2007) and was **fixed on Oct 18, 2016**
# Tools for Formal Software Verification

**How do we increase software reliability?**

<table>
<thead>
<tr>
<th>Point Tools</th>
<th>General-Purpose Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Low overhead</td>
<td>- High overhead</td>
</tr>
<tr>
<td>- Limited scope</td>
<td>- Unrealistic models</td>
</tr>
<tr>
<td><strong>Restrictive</strong></td>
<td><strong>Misleading</strong></td>
</tr>
</tbody>
</table>

- **Point Tools**
  - Low overhead
  - Limited scope

- **General-Purpose Tools**
  - High overhead
  - Unrealistic models
Tools for Formal Software Verification

**How do we increase software reliability?**

**Point Tools**
- Low overhead
- Limited scope
  
  **Restrictive**

**General-Purpose Tools**
- Lower overhead
  - Accurate models
  - Reliable
## Tools for Formal Software Verification

### How do we increase software reliability?

<table>
<thead>
<tr>
<th>Point Tools</th>
<th>General-Purpose Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Low overhead</td>
<td>Lower overhead</td>
</tr>
<tr>
<td>- Limited scope</td>
<td>Accurate models</td>
</tr>
<tr>
<td>Restrictive</td>
<td>Reliable</td>
</tr>
</tbody>
</table>

- This research: *General-purpose tool for formal software verification based on an accurate model of the x86 ISA*

- Make formal verification of machine code a *practical* choice
Why x86 Machine-Code Verification?

• **Why not high-level code verification?**
  × Sometimes, high-level code is unavailable (e.g., malware)
  × 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

• **Why x86?**
  ✓ x86 is in widespread use
Overview

Goal

Specify and verify properties of x86 application and system programs
Overview

Goal

*Specify and verify* properties of x86 application and system programs

*specify:*
in terms of states of computation
Overview

Goal

*Specify and verify* properties of x86 application and system programs

**specify:** in terms of states of computation

**verify:** reason about symbolic executions
Overview

Goal

Specify and verify properties of x86 application and system programs

specify:
in terms of states of computation

verify:
reason about symbolic executions

Approach

1. Build a **formal, executable model of the x86 ISA** using ACL2
2. Develop a **machine-code analysis framework** based on this model that supports reasoning about
   (a) application programs, and (b) system programs
3. **Employ this framework** to verify
   (a) application programs, and (b) system programs
My Ph.D. proposal described:

1. x86 ISA model
2. (a) Libraries to reason about application programs
3. (a) Verification of two application programs
Review

My Ph.D. proposal described:
1. x86 ISA model
2. (a) Libraries to reason about application programs
3. (a) Verification of two application programs

Focus of this talk:
1. New features of the x86 ISA model
2. (b) Libraries to reason about system programs
3. (b) Verification of a system program — Zero-Copy
Review

My Ph.D. proposal described:
1. x86 ISA model
2. (a) Libraries to reason about application programs
3. (a) Verification of two application programs

Focus of this talk:
1. New features of the x86 ISA model
2. (b) Libraries to reason about system programs
3. (b) Verification of a system program — Zero-Copy

[Diss. Ch. 7] [Diss. Ch. 10] [Diss. Ch. 12]
Review

My Ph.D. proposal described:
1. x86 ISA model
2. (a) Libraries to reason about application programs
3. (a) Verification of two application programs

Focus of this talk:
1. New features of the x86 ISA model
2. (b) Libraries to reason about system programs
3. (b) Verification of a system program — Zero-Copy

<table>
<thead>
<tr>
<th></th>
<th>STATUS: THEN</th>
<th>STATUS: NOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86 ISA Model</td>
<td>220 Opcodes</td>
<td>413 Opcodes</td>
</tr>
<tr>
<td>Lemma Libraries</td>
<td>Support only for application programs</td>
<td>Support added for system programs</td>
</tr>
<tr>
<td>Case Studies</td>
<td>Application programs</td>
<td>Added system program (Zero-Copy)</td>
</tr>
<tr>
<td>Documentation</td>
<td>Largely developer-focused topics</td>
<td>Added user-focused topics, including a guide to debug failed proofs</td>
</tr>
</tbody>
</table>
Our Framework: Design Goals

**Accuracy**
Reliable program analysis
## Our Framework: Design Goals

<table>
<thead>
<tr>
<th><strong>Accuracy</strong></th>
<th>Reliability program analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Efficiency</strong></td>
<td>Aid in co-simulations and testing</td>
</tr>
</tbody>
</table>
Our Framework: Design Goals

- **Accuracy**
  Reliable program analysis

- **Execution Efficiency**
  Aid in co-simulations and testing

- **Reasoning Efficiency**
  Reduce user effort, e.g., support failed proofs’ debugging
Our Framework: Design Goals

**Accuracy**
Reliable program analysis

**Execution Efficiency**
Aid in co-simulations and testing

**Usability**
Balance verification effort and verification utility

**Reasoning Efficiency**
Reduce user effort, e.g., support failed proofs’ debugging
Our Framework: Design Goals

**Accuracy**
Reliable program analysis

**Execution Efficiency**
Aid in co-simulations and testing

**Usability**
Balance verification effort and verification utility

**Reasoning Efficiency**
Reduce user effort, e.g., support failed proofs’ debugging
Our Framework: Design Goals

**Accuracy**
Reliable program analysis

**Usability**
Balance verification effort and verification utility

**Execution Efficiency**
Aid in co-simulations and testing

**Reasoning Efficiency**
Reduce user effort, e.g., support failed proofs’ debugging

abstract stobjs
Our Framework: Design Goals

**Accuracy**
Reliable program analysis

**Execution Efficiency**
Aid in co-simulations and testing

**Usability**
Balance verification effort and verification utility

**Reasoning Efficiency**
Reduce user effort, e.g., support failed proofs’ debugging

*abstract stobjs*
Our Framework: Design Goals

**Accuracy**
Reliable program analysis

**Usability**
Balance verification effort and verification utility

**Execution Efficiency**
Aid in co-simulations and testing

**Reasoning Efficiency**
Reduce user effort, e.g., support failed proofs’ debugging

modes of operation

abstract stobjs
Outline

Overview

1. Formal Model of the x86 ISA
2. Lemma Libraries for Machine-Code Verification
3. Case Studies

Concluding Remarks and Future Work
Obtaining the x86 ISA Specification

~3400 pages

---

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

Running tests on x86 machines
Focus: Intel's 64-bit mode

Figure 3-2. 64-Bit Mode Execution Environment

Source: Intel Manuals
2.1.1 Global and Local Descriptor Tables

When operating in protected mode, all memory accesses pass through either the global descriptor table (GDT) or an optional local descriptor table (LDT) as shown in Figure 2-1. These tables contain entries called segment descriptors. Segment descriptors provide the base address of segments as well as access rights, type, and usage information.

Each segment descriptor has an associated segment selector. A segment selector provides the software that uses it with an index into the GDT or LDT (the offset of its associated segment descriptor), a global/local flag (determines whether the selector points to the GDT or the LDT), and access rights information.

---

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

- **Address Space:**
  - $2^{64}$
  - 0

- **Registers:**
  - **Control Register (CR0)**
  - **CR1**
  - **CR2**
  - **CR3**
  - **CR4**
  - **LDTR**
  - **GDTR**
  - **Task Register (TR)**
  - **Global Descriptor Table (GDT)**
  - **Segment Selector**
  - **Local Descriptor Table (LDT)**
  - **IST**
  - **Interrupt Gate**
  - **Trap Gate**
  - **Interrupt Descriptor Table (IDT)**
  - **Interrupt Vector**
  - **Task-State Segment (TSS)**
  - **Code, Data or Stack Segment (Base = 0)**

- **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.

---

**Source:** Intel Manuals

---

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

- **Sixteen 64-bit Registers**
- **General-Purpose Registers**
- **Segment Registers**
- **RFLAGS Register**
- **RIP (Instruction Pointer Register)**
- **Sixteen 128-bit Registers**
- **XMM Registers**
- **Six 16-bit Registers**
- **Segment Registers**
- **64-bits**
- **RFLAGS Register**
- **64-bits**
- **RIP (Instruction Pointer Register)**
- **Eight 80-bit Registers**
- **Floating-Point Data Registers**
- **16 bits**
- **Control Register**
- **Status Register**
- **16 bits**
- **Tag Register**
- **16 bits**
- **Opcode Register (11-bits)**
- **FPU Instruction Pointer Register**
- **FPU Data (Operand) Pointer Register**
- **Eight 64-bit Registers**
- **MMX Registers**
- **Eight 16-bit Registers**
- **Registers**
- **Sixteen 128-bit Registers**
- **XMM Registers**
- **32-bits**
- **MXCSR Register**

---

**Focus:** Intel’s 64-bit mode

---

**Source:** Intel Manuals

---

**Focus:** Intel's 64-bit mode

---

**Source:** Intel Manuals
## Modes of Operation of the x86 ISA Model

### User-level Mode
- Verification of *application* programs
- *Linear* memory address space ($2^{64}$ bytes)
- *Assumptions* about correctness of OS operations
  - Specification of system calls

### System-level Mode
- Verification of *system* programs
- *Physical* memory address space ($2^{52}$ bytes)
  - Specification of paging
- *No assumptions* about OS operations
Model Validation

*How can we know that our model faithfully represents the x86 ISA?*

Validate the model to increase trust in the applicability of formal analysis.
Outline

Overview

1. Formal Model of the x86 ISA

2. Lemma Libraries for Machine-Code Verification

3. Case Studies

Concluding Remarks and Future Work
Supporting Symbolic Execution

Rules (theorems) describing interactions between these reads and writes to the x86 state enable *symbolic execution* of programs.
Linear Memory Non-Interference Theorem

user-level mode

Program Order

linear memory
Linear Memory Non-Interference Theorem

user-level mode

Program Order

linear memory

\[ W(i, x) \]
Linear Memory Non-Interference Theorem

user-level mode

Program Order

W(i, x)

R(j) = y

linear memory
Linear Memory Non-Interference Theorem

(user-level mode)

\[ W(i, x) \]

\[ R(j) = y \]

Program Order

(defthm linear-mem-non-interference-user-level-mode
  (implies
    (and (disjoint-p las-1 las-2)
         (user-level-mode x86))
    (equal
      (read-mem las-1 r-x (write-mem las-2 bytes x86))
      (read-mem las-1 r-x x86))))

\textbf{las-1 las-2} – lists of linear addresses
Reasoning about Paging is Complicated #1

1. Complicated data structures — hierarchical, with two to four levels of indirection, depending on the page configuration
1. Complicated data structures — hierarchical, with two to four levels of indirection, depending on the page configuration.
Reasoning about Paging is Complicated #1

1. Complicated data structures — hierarchical, with two to four levels of indirection, depending on the page configuration
Reasoning about Paging is Complicated #1

1. Complicated data structures — hierarchical, with two to four levels of indirection, depending on the page configuration

<table>
<thead>
<tr>
<th>CR3</th>
<th>Source: Intel Manuals</th>
</tr>
</thead>
<tbody>
<tr>
<td>PML4E: present</td>
<td></td>
</tr>
<tr>
<td>PML4E: not present</td>
<td></td>
</tr>
<tr>
<td>PDPT: 1GB page</td>
<td></td>
</tr>
<tr>
<td>PDPT: page directory</td>
<td></td>
</tr>
<tr>
<td>PDPT: not present</td>
<td></td>
</tr>
<tr>
<td>PDE: 2MB page</td>
<td></td>
</tr>
<tr>
<td>PDE: page table</td>
<td></td>
</tr>
<tr>
<td>PDE: not present</td>
<td></td>
</tr>
<tr>
<td>PTE: 4KB page</td>
<td></td>
</tr>
<tr>
<td>PTE: not present</td>
<td></td>
</tr>
</tbody>
</table>

![Figure 4-11. Formats of CR3 and Paging-Structure Entries with IA-32e Paging](source:intel Manuals)
Reasoning about Paging is Complicated #1

1. Complicated data structures — hierarchical, with two to four levels of indirection, depending on the page configuration

<table>
<thead>
<tr>
<th>CR3</th>
<th>Reserved²</th>
<th>Address of PML4 table</th>
<th>Ignored</th>
<th>Ign.</th>
<th>Ign.</th>
<th>PML4E: present</th>
<th>PML4E: not present</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CR3</th>
<th>Reserved²</th>
<th>Address of page-directory-pointer table</th>
<th>Ignored</th>
<th>Ignored</th>
<th>PML4E: present</th>
<th>PML4E: not present</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Ignored</td>
<td>Rsvd.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CR3</th>
<th>Reserved²</th>
<th>Address of 1GB page frame</th>
<th>Reserved</th>
<th>Ignor.</th>
<th>G 1</th>
<th>PDPTE: 1GB page</th>
<th>PDPTE: page directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Ignored</td>
<td>Rsvd.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CR3</th>
<th>Reserved²</th>
<th>Address of 2MB page frame</th>
<th>Reserved</th>
<th>Ignor.</th>
<th>G 1</th>
<th>PDPTE: page directory</th>
<th>PDPTE: 2MB page</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Ignored</td>
<td>Rsvd.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CR3</th>
<th>Reserved²</th>
<th>Address of page table</th>
<th>Ignored</th>
<th>PDPTE: page table</th>
<th>PDPTE: not present</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Ignored</td>
<td>Rsvd.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CR3</th>
<th>Reserved²</th>
<th>Address of 4KB page frame</th>
<th>Ignored</th>
<th>PDPTE: 4KB page</th>
<th>PDPTE: not present</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Ignored</td>
<td>Rsvd.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4-11. Formats of CR3 and Paging-Structure Entries with IA-32e Paging

Source: Intel Manuals
Reasoning about Paging is Complicated #2

2. *Accessed and dirty flag* updates during paging structure traversals cause side-effect writes

Paging entries governing the translation of a linear address are *marked*. 
3. Paging data structures are located in the physical memory
   - Physical memory cannot be accessed directly in the 64-bit mode — not even by supervisor-mode programs.
   - In order to access a paging entry, the entry’s own linear address needs to be translated to a physical address first.
   - **Paging structures are mapped, too!**
Linear Memory Non-Interference Theorem

*When is a linear memory read operation unaffected by a linear memory write operation?*

**system-level mode**

When is a linear memory read operation unaffected by a linear memory write operation?
Linear Memory Non-Interference Theorem

When is a linear memory read operation unaffected by a linear memory write operation?

system-level mode

A and B are disjoint

A • B
Linear Memory Non-Interference Theorem

When is a linear memory read operation unaffected by a linear memory write operation?

**system-level mode**

A and B are disjoint

A and B are disjoint

A and B are disjoint

A and B are disjoint
Linear Memory Non-Interference Theorem

When is a linear memory read operation unaffected by a linear memory write operation?

**system-level mode**

A and B are disjoint

A ❌ B
Linear Memory Non-Interference Theorem

When is a linear memory read operation unaffected by a linear memory write operation?

**system-level mode**

A and B are disjoint

```
A ——— B
```
Linear Memory Non-Interference Theorem

(defun linear-mem-non-interference-system-level-mode
  (let* ((pas-1 (las-to-pas las-1 r-x (cpl x86) x86))
         (pas-2 (las-to-pas las-2 :w (cpl x86) x86)))
    (implies (and
              (disjoint-p pas-1 pas-2)
              (disjoint-p pas-2
                          (paging-entries-paddrs las-1 x86))
              (disjoint-p pas-1
                          (paging-entries-paddrs las-2 x86))
              (disjoint-p pas-1
                          (paging-entries-paddrs las-1 x86))
              (system-level-mode x86)
              ;; <other simple hypotheses elided here...>
             )
      (equal
       (read-mem las-1 r-x (write-mem las-2 bytes x86))
       (read-mem las-1 r-x x86))))

las-1 las-2 – lists of linear addresses
Linear Memory Non-Interference Theorem

(defthm linear-mem-non-interference-system-level-mode
  (let* ((pas-1 (las-to-pas las-1 r-x (cpl x86) x86))
         (pas-2 (las-to-pas las-2 :w (cpl x86) x86)))
    (implies (and (disjoint-p pas-1 pas-2)
                   (disjoint-p pas-2
                               (paging-entries-paddrs las-1 x86))
                   (disjoint-p pas-1
                               (paging-entries-paddrs las-2 x86))
                   (disjoint-p pas-1
                               (paging-entries-paddrs las-1 x86)))
     (equal (read-mem las-1 r-x
                       (write-mem las-2 bytes x86))
            (read-mem las-1 r-x x86))))

Complicates precondition discovery

A large number of hypotheses makes it challenging to discover interesting and/or non-obvious preconditions.

las-1 las-2 – lists of linear addresses
System-level Mode: Sub-modes of Operation

- *Common case*: reads to fetch the next instruction or obtain program’s data
  - A program and its data are usually disjoint from system data structures
  - Why pay the penalty of side-effect A/D flag updates for these reads?
System-level Mode: Sub-modes of Operation

- **Common case:** reads to fetch the next instruction or obtain program’s data
  - A program and its data are usually disjoint from system data structures
  - Why pay the penalty of side-effect A/D flag updates for these reads?
- **Optimization:** separate side-effect A/D flag updates from other updates
System-level Mode: Sub-modes of Operation

- **Common case**: reads to fetch the next instruction or obtain program’s data
  - A program and its data are usually disjoint from system data structures
  - Why pay the penalty of side-effect A/D flag updates for these reads?
- **Optimization**: separate side-effect A/D flag updates from other updates
- Two sub-modes of operation: *marking* and *non-marking* mode
  - **Marking Mode**: true specification of the x86 ISA
  - **Non-marking Mode**: side-effect updates to A/D flags suppressed
    - Simpler theorems, easier precondition discovery
System-level Mode: Sub-modes of Operation

- Non-marking mode: simpler theorems, easier precondition discovery

  *Modus Operandi*:
  - First verify a program in the non-marking mode and then port it over to the marking mode

- *Caveat*:
  - Works for programs that do not rely on side-effect A/D flag updates
  - Can always reason directly in the system-level marking mode
Linear Memory Non-Interference Theorem

*When is a linear memory read operation unaffected by a linear memory write operation?*

**system-level non-marking mode**

![Diagram showing memory hierarchy and disjointness of regions A and B]

A and B are disjoint

**A** ◼**B**
Linear Memory Non-Interference Theorem

When is a linear memory read operation unaffected by a linear memory write operation?

**system-level non-marking mode**

A and B are disjoint

A • B

**read_pa**

**write_pa**

**read_la**

**write_la**

PML4E

PML4E

CR3

PDPTE

1G Page
Linear Memory Non-Interference Theorem

(defun linear-mem-non-interference-system-level-non-marking-mode
  (let* ((pas-1 (las-to-pas las-1 r-x (cpl x86) x86))
         (pas-2 (las-to-pas las-2 :w (cpl x86) x86)))
    (implies (and (disjoint-p pas-1 pas-2)
                   (disjoint-p pas-2 (paging-entries-paddrs las-1 x86))
                   (system-level-non-marking-mode x86)
                   ;; <other simple hypotheses elided here...>
                   )
     (equal (read-mem las-1 r-x (write-mem las-2 bytes x86))
            (read-mem las-1 r-x x86))))

\text{las-1 las-2} \quad \text{– lists of linear addresses}
Reducing Reasoning Overhead in Marking Mode

In the **system-level marking mode** of operation:

- Memory reads disjoint from the paging data structures *automatically ignore* side-effect updates to A/D flags
  - Provided all the additional disjointness conditions are specified
Reducing Reasoning Overhead in Marking Mode

In the **system-level marking mode** of operation:

- Memory reads disjoint from the paging data structures *automatically ignore* side-effect updates to A/D flags
  - Provided all the additional disjointness conditions are specified

- **Conditional Congruence-based Rewriting:**
  - Rewrite `read-mem` to `read-mem-alt` if applicable; use congruence rules to allow `read-mem-alt` to ignore side-effect updates to A/D flags
Reducing Reasoning Overhead in Marking Mode

In the **system-level marking mode** of operation:

- Memory reads disjoint from the paging data structures *automatically ignore* side-effect updates to A/D flags
  - Provided all the additional disjointness conditions are specified

- **Conditional Congruence-based Rewriting:**
  - Rewrite `read-mem` to `read-mem-alt` if applicable; use congruence rules to allow `read-mem-alt` to ignore side-effect updates to A/D flags

- **Program Comprehension:**
  - Memory read operation in terms of `read-mem`: target is a paging entry
  - Memory read operation in terms of `read-mem-alt`: target is disjoint from paging structures
Outline

Overview

1. Formal Model of the x86 ISA
2. Lemma Libraries for Machine-Code Verification
3. Case Studies

Concluding Remarks and Future Work
Case Study: Zero-Copy Program

- **Copies data by modifying the paging structures** so that both the src and dst are mapped to the same physical memory location
  - Zero copies exist in reality
  - Can be used for implementing the Copy-on-Write (COW) technique
Case Study: Zero-Copy Program

- **Copies data by modifying the paging structures** so that both the src and dst are mapped to the same physical memory location
  - Zero copies exist in reality
  - Can be used for implementing the Copy-on-Write (COW) technique

- **Establishing this program’s correctness is critical:**
  - Linear memory is the only view of memory available to 64-bit x86 programs.
  - An incorrect setup of paging structures can cause security leaks and crashes in otherwise correct programs.
Case Study: Zero-Copy Program

Constraints:
- Data to be copied: 1GB
- Source and destination are 1GB-aligned
Case Study: Zero-Copy Program

Constraints:
- Data to be copied: 1GB
- Source and destination are 1GB-aligned
Case Study: Zero-Copy Program

**Constraints:**
- Data to be copied: 1GB
- Source and destination are 1GB-aligned

**Key Challenge:**
Discovering and specifying the conditions under which this program operates correctly
Proved Functional Correctness: implementation of a zero-copy program meets the specification of a simple copy operation.

1. [Copy Occurs] The 1GB of data at the destination’s linear addresses in the final x86 state is the same as the 1GB of data at the source’s linear addresses in the initial x86 state.

2. [Source is Unmodified] The 1GB of data at the source’s linear addresses in the final x86 state is the same as the 1GB of data at the source’s linear addresses in the initial x86 state.

3. [Program is Unmodified] The program in the final x86 state is the same as that in the initial x86 state.
Case Study: Zero-Copy Program

Proved Functional Correctness: implementation of a zero-copy program meets the specification of a simple copy operation.

1. [Copy Occurs] The 1GB of data at the destination’s linear addresses in the final x86 state is the same as the 1GB of data at the source’s linear addresses in the initial x86 state.

2. [Source is Unmodified] The 1GB of data at the source’s linear addresses in the final x86 state is the same as the 1GB of data at the source’s linear addresses in the initial x86 state.

3. [Program is Unmodified] The program in the final x86 state is the same as that in the initial x86 state.

Around 120 preconditions, mostly about the disjointness of different regions of the memory (e.g., program, data, stack, paging entries)
Case Study: Zero-Copy Program

Initial x86 state

Final x86 state

View of Linear Memory

two copies of data
modification to destination’s PDPT for linear address re-mapping

not drawn to scale
Outline

Overview

1. Formal Model of the x86 ISA
2. Lemma Libraries for Machine-Code Verification
3. Case Studies

Concluding Remarks and Future Work
Review

My Ph.D. proposal described:
1. x86 ISA model
2. (a) Libraries to reason about application programs
3. (a) Verification of two application programs

Focus of this talk: [Diss. Ch. 7]
1. New features of the x86 ISA model
2. (b) Libraries to reason about system programs [Diss. Ch. 10]
3. (b) Verification of a system program — Zero-Copy [Diss. Ch. 12]

<table>
<thead>
<tr>
<th></th>
<th>STATUS: THEN</th>
<th>STATUS: NOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86 ISA Model</td>
<td>220 Opcodes</td>
<td>413 Opcodes</td>
</tr>
<tr>
<td>Lemma Libraries</td>
<td>Support only for application programs</td>
<td>Support added for system programs</td>
</tr>
<tr>
<td>Case Studies</td>
<td>Application programs</td>
<td>Added system program (Zero-Copy)</td>
</tr>
<tr>
<td>Documentation</td>
<td>Largely developer-focused topics</td>
<td>Added user-focused topics, including a guide to debug failed proofs</td>
</tr>
</tbody>
</table>
Contributions

**Formal, executable specification of the x86 ISA (IA-32e mode)**
- Accurate reference of the x86 ISA
- Fastest formal simulator of its kind
- Tools that support its use as a practical instruction-set simulator

**Reasoning framework for x86 machine-code analysis**
- Automated symbolic simulation of x86 machine-code programs
- Supports reasoning about system data structures

**Verification strategies that can be adopted to verify a variety of machine-code programs**

**Documentation of engineering aspects of building a large-scale formal analysis framework**
## Opportunities for Future Research

<table>
<thead>
<tr>
<th>Operating System Verification</th>
<th>User-friendly Program Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>detect reliance on non-portable or undefined behaviors</td>
<td>automate the discovery of preconditions</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Multi-process/threaded Program Verification</th>
<th>Reasoning about the Memory System</th>
</tr>
</thead>
<tbody>
<tr>
<td>reason about concurrency-related issues</td>
<td>determine if caches are (mostly) transparent, as intended</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Firmware Verification</th>
<th>Micro-architecture Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>formally specify software/hardware interfaces</td>
<td>x86 ISA model serves as a build-to specification</td>
</tr>
</tbody>
</table>
Publications


[Source Code]
Github
[Documentation]
x86isa in the ACL2+Community Books Manual

Thanks!
Extra Slides
Case Study: Pop-Count Program

**Functional Correctness:**
RAX = popcount(v)

**specification function**

\[
\text{popcount}(v): [v: \text{unsigned int}]
\]

\[
\begin{align*}
\text{if} & \ (v \leq 0) \ \text{then} \\
& \ \text{return} \ 0 \\
\text{else} \\
& \ lsb := v \ & 1 \\
& \ v := v \gg 1 \\
& \ \text{return} \ (lsb + \ \text{popcount}(v)) \\
\end{align*}
\]

**Code Example:**

```
popcount_64:
   mov    %edi,%edx
   mov    %edx,%ecx
   shr    %ecx
   and    $0x55555555,%ecx
   sub    %ecx,%edx
   mov    %edx,%eax
   shr    $0x2,%eax
   and    $0x33333333,%eax
   add    %eax,%edx
   mov    %edx,%eax
   shr    $0x4,%eax
   and    $0x33333333,%eax
   add    %eax,%edx
   mov    %eax,%ecx
   shr    $0x2,%eax
   and    $0x33333333,%eax
   add    %edx,%eax
   mov    %eax,%ecx
   shr    $0x4,%ecx
   add    %ecx,%eax
   and    $0xf0f0f0f0f, %edx
   imul   $0x101010101,%edx,%edx
   imul   $0x101010101,%eax,%eax
   shr    $0x18,%edx
   shr    $0x18,%eax
   add    %edx,%eax
   retq
```
Case Study: Word-Count Program

- Program obtains input from the user via `read` system calls.
- System calls are **non-deterministic** for application programs.
Case Study: Word-Count Program

- Program obtains input from the user via read system calls.
- System calls are **non-deterministic** for application programs.

**Functional Correctness Theorem:**
Values computed by specification functions on standard input are found in the expected memory locations of the final x86 state.

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
```

```
Case Study: Word-Count Program

- Program obtains input from the user via read system calls.
- System calls are non-deterministic for application programs.

Functional Correctness Theorem:
Values computed by specification functions on standard input are found in the expected memory locations of the final x86 state.

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

```
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
```

Case Study: Word-Count Program

• Program obtains input from the user via `read` system calls.
• System calls are **non-deterministic** for application programs.

**Functional Correctness Theorem:**
Values computed by specification functions on standard input are found in the expected memory locations of the final x86 state.

```
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
```

**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.
Zero-Copy: View of Physical Memory

Initial x86 state

Final x86 state

Source
Program Stack
Paging entries pertaining to the stack and program
Source and destination PML4TEs and PDPTES
Map of Paging Data Structures

Physical Memory

Physical Memory

one (original) copy of data
modification to destination’s PDPTE for linear address re-mapping
Reasoning & Execution Efficiency: *Abstract Stobjs*

- Layered modeling approach mitigates the trade-off between reasoning and execution efficiency.

- Abstract stobjs were added to ACL2 in response to the needs of this research project.
Review: Timeline

Adhered to the timeline envisioned in the proposal:

“Spring 2015 – Summer 2015: Specifying more x86 instructions; modeling the system descriptor tables to support segmentation and interrupts; formulating and proving properties about paging data structure traversals and modifications

Fall 2015: Choosing and simulating system program(s), such as an optimized data-copy program; this would identify the x86 features that need to be modeled in order to support the program’s execution and verification

Spring 2016: Verification of the target program(s)—this includes discovering and specifying properties of interest; it may also involve re-visiting modeling choices made earlier

Summer 2016 – Fall 2016: Dissertation writing and final defense”

Data point: envisioning how long a verification effort will take is becoming predictable
Deliverables

**Formal Specification**
- A formal, executable x86 ISA model (64-bit mode)

**Instruction-Set Simulator**
- Executable file readers and loaders (ELF/Mach-O)
- A GDB-like mode for dynamic instrumentation of machine code
- Examples of program execution and debugging

**Code Proof Libraries**
- Helper libraries to reason about x86 machine code
- Proofs of various properties of some machine-code programs

**Manual**
- Documentation
x86 ISA Model

A Run of the x86 Interpreter that executes k instructions

Interpreter-Style Operational Semantics:

- **x86 State**: specifies the components of the ISA
- **Instruction Semantic Functions**: specifies instructions’ behavior
- **Step Function**: fetches, decodes, and executes one instruction
- **Run Function**: takes n steps or terminates early if an error occurs
Independence of Page Walks

- Proved using **congruence-based reasoning** in ACL2
  - Define an *equivalence relation* that states that two x86 states are equivalent if their paging structures are equal, modulo the A and D flags, and the rest of the memory is exactly equal.

  - Prove that the x86 state produced by a page walk is equivalent to the initial x86 state.

  - A page walk returns the same physical address for a linear address, given equivalent x86 states.
Successive Linear Memory Reads

System-level Marking Mode

\[(\text{mv-nth 1 (rb las-1 r-x-1})
   \quad (\text{mv-nth 2 (rb las-2 r-x-2 x86)}))\]

\[=\]

\[(\text{mv-nth 1 (rb las-1 r-x-1})
   \quad \text{<writes to A flags of las-2’s translation-governing entries>})\]

The above expression can be simplified to

\[(\text{mv-nth 1 (rb las-1 r-x-1 x86)})\]

only if physical addresses corresponding to las-1 are disjoint from the physical addresses of the translation-governing entries of las-2.
Successive Linear Memory Reads

System-level Non-marking Mode

\[(\text{mv-nth 1 (rb las-1 r-x-1)}
\quad \quad \text{\textbf{(mv-nth 2 (rb las-2 r-x-2 x86))))})\]
\[=\]
\[(\text{mv-nth 1 (rb las-1 r-x-1 x86))}\]

because, in the non-marking mode:

\[(\text{mv-nth 2 (rb las-2 r-x-2 x86))))\]
\[=\]
\[x86\]
Reasoning about Paging is Complicated #3

3. Paging data structures are located in the physical memory
   - Physical memory cannot be accessed directly in the 64-bit mode — not even by supervisor-mode programs.
   - In order to access a paging entry, the entry’s own linear address needs to be translated to a physical address first.
   - Paging structures are mapped, too!
Reasoning about Paging is Complicated #3

3. Paging data structures are located in the physical memory
   - Physical memory cannot be accessed directly in the 64-bit mode — not even by supervisor-mode programs.
   - In order to access a paging entry, the entry’s own linear address needs to be translated to a physical address first.
   - **Paging structures are mapped, too!**

**Specification:** Maximum number of memory accesses to translate one linear address with a 4K configuration
Reasoning about Paging is Complicated #3

3. Paging data structures are located in the physical memory
   - Physical memory cannot be accessed directly in the 64-bit mode — not even by supervisor-mode programs.
   - In order to access a paging entry, the entry’s own linear address needs to be translated to a physical address first.
   - **Paging structures are mapped, too!**

**Specification:** Maximum number of memory accesses to translate one linear address with a 4K configuration
3. Paging data structures are located in the physical memory
   - Physical memory cannot be accessed directly in the 64-bit mode — not even by supervisor-mode programs.
   - In order to access a paging entry, the entry’s own linear address needs to be translated to a physical address first.
   - **Paging structures are mapped, too!**

**Specification:** Maximum number of memory accesses to translate one linear address with a 4K configuration
Reasoning about Paging is Complicated #3

3. Paging data structures are located in the physical memory
   - Physical memory cannot be accessed directly in the 64-bit mode — not even by supervisor-mode programs.
   - In order to access a paging entry, the entry’s own linear address needs to be translated to a physical address first.
   - **Paging structures are mapped, too!**

**Specification:** Maximum number of memory accesses to translate one linear address with a 4K configuration
3. Paging data structures are located in the physical memory
   - Physical memory cannot be accessed directly in the 64-bit mode — not even by supervisor-mode programs.
   - In order to access a paging entry, the entry’s own linear address needs to be translated to a physical address first.
   - **Paging structures are mapped, too!**

**Specification:** Maximum number of memory accesses to translate one linear address with a 4K configuration

Total number of memory accesses: 40