Simulation and Formal Verification of x86 Machine-Code Programs that Make System Calls

Shilpi Goel
Warren A. Hunt, Jr.
Matt Kaufmann
Soumava Ghosh

The University of Texas at Austin

22nd October, 2014
OUTLINE

1. INTRODUCTION

2. SIMULATION AND REASONING FRAMEWORK
   - x86 ISA Model
   - System Calls Model

3. CODE PROOFS

4. CONCLUSION AND FUTURE WORK
OUTLINE

1. INTRODUCTION

2. SIMULATION AND REASONING FRAMEWORK
   - x86 ISA Model
   - System Calls Model

3. CODE PROOFS

4. CONCLUSION AND FUTURE WORK
Motivation

Bug-hunting tools, like static analyzers, have matured remarkably.

- Regularly used in the software development industry
- Strengths: easy to use; largely automatic
- Weaknesses: cannot prove complex invariants; cannot prove the absence of bugs
**Motivation**

Bug-hunting tools, like static analyzers, have matured remarkably.

- Regularly used in the software development industry
- Strengths: easy to use; largely automatic
- Weaknesses: cannot prove complex invariants; cannot prove the absence of bugs

We want to formally verify properties of (x86 machine-code) programs that cannot be established in the foreseeable future by automatic tools.
Our Approach

*Focus*: Mechanical verification of **user-level x86 machine-code programs** that request services from an operating system via **system calls**
Our Approach

Focus: Mechanical verification of user-level x86 machine-code programs that request services from an operating system via system calls

- Specify the x86 ISA and Linux/FreeBSD system calls in ACL2 programming/proof environment
Our Approach

Focus: Mechanical verification of user-level x86 machine-code programs that request services from an operating system via system calls

- **Specify** the x86 ISA and Linux/FreeBSD system calls in ACL2 programming/proof environment
- **Validate** the above specification against real hardware and software
Our Approach

Focus: Mechanical verification of user-level x86 machine-code programs that request services from an operating system via system calls

- **Specify** the x86 ISA and Linux/FreeBSD system calls in ACL2 programming/proof environment
- **Validate** the above specification against real hardware and software
- **Reason** about x86 machine-code programs using this specification
WHAT’S SPECIAL ABOUT SYSTEM CALLS?

▶ From the point of view of a programmer, system calls are non-deterministic; different runs can yield different results on the same machine.
WHAT’S SPECIAL ABOUT SYSTEM CALLS?

▶ From the point of view of a programmer, system calls are non-deterministic; different runs can yield different results on the same machine.

▶ This makes it non-trivial to reason about user-level programs that make system calls.
What’s special about System Calls?

- From the point of view of a programmer, system calls are **non-deterministic**; different runs can yield different results on the same machine.

- This makes it non-trivial to reason about user-level programs that make system calls.

Proved **functional correctness** of a word count program
Correctness of the Word Count Program

Assembly Program Snippet

```assembly
... push %rbx lea -0x9(%rbp),%rax mov %rax,-0x20(%rbp) mov $0x0,%rax xor %rdi,%rdi mov -0x20(%rbp),%rsi mov $0x1,%rdx syscall mov %eax,%ebx mov %ebx,-0x10(%rbp) movzbl -0x9(%rbp),%eax movzbl %al,%eax...```

Pseudo-code: Specification Function

```python
ncSpec(offset, str, count):
if (EOF-TERMINATED(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
```

Theorem

```
preconditions(rip, x86) ∧ x86f = x86-run(clk(x86i), x86i) ⇒
getNc(x86f) = ncSpec(Offset(x86i), Str(x86i), 0)
```
**Correctness of the Word Count Program**

**Assembly Program Snippet**

```
... push %rbx lea -0x9(%rbp),%rax mov %rax,-0x20(%rbp) mov $0x0,%rax xor %rdi,%rdi mov -0x20(%rbp),%rsi mov $0x1,%rdx syscall mov %eax,%ebx mov %ebx,-0x10(%rbp) movzbl -0x9(%rbp),%eax movzbl %al,%eax ...
```

**Pseudo-code: Specification Function**

```
def ncSpec(offset, str, count):
    if (EOF-TERMINATED(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
```

**Theorem**

\[
\text{preconditions}(\text{rip}_i, x86_i) \land x86_f = x86-\text{run}(\text{clk}(x86_i), x86_i) \implies \text{getNc}(x86_f) = \text{ncSpec}(\text{Offset}(x86_i), \text{Str}(x86_i), 0)
\]

---
OUTLINE

1. INTRODUCTION

2. SIMULATION AND REASONING FRAMEWORK
   - x86 ISA Model
   - System Calls Model

3. CODE PROOFS

4. CONCLUSION AND FUTURE WORK
x86 ISA + System Calls Specification

- Formalization of the x86 ISA, with syscall extended by a specification of Linux and FreeBSD system calls

- Formal and executable specification

- Memory model: 64-bit linear address space
OUTLINE

1 INTRODUCTION

2 SIMULATION AND REASONING FRAMEWORK
   ● x86 ISA Model
   ● System Calls Model

3 CODE PROOFS

4 CONCLUSION AND FUTURE WORK
x86 ISA Model in ACL2

- Interpreter-style operational semantics
- Semantics of a program is given by the effect it has on the state of the machine.
- State-transition function is characterized by a recursively defined interpreter. We call this state transition function x86-run.
## Formalization: x86 State

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers</td>
<td>general-purpose, segment, debug, control, floating point, MMX, model-specific</td>
</tr>
<tr>
<td>rip</td>
<td>instruction pointer</td>
</tr>
<tr>
<td>flg</td>
<td>flags register</td>
</tr>
<tr>
<td>env</td>
<td>environment field</td>
</tr>
<tr>
<td>mem</td>
<td>memory</td>
</tr>
</tbody>
</table>
FORMALIZATION: STATE TRANSITION FUNCTION

- State transition function: **fetch, decode & execute**
- Each instruction has its own semantic function
FACTSHEET: x86 ISA Model

- 64-bit mode of Intel’s IA-32e mode
- 221 general and 96 SSE/SSE2 opcodes
- Implementation of all addressing modes
- Lines of Code: ~40,000
- Execution speed: up to 3.3 million instructions/second

Machine used: 3.50GHz Intel Xeon E31280 CPU
Assessing the Accuracy of the ISA Model

Co-simulations

State-by-State Diff

ACL2 printing functions
GDB scripts, Pin

x86 ISA model in ACL2
Instruction Semantic Functions
Fetch, Decode, and Execute Function
x86 state

x86

Binary Program Loader in ACL2

Program Opcodes
Implemented?
No → Implement missing opcodes
Yes

GCC/LLVM

C

Introduction
Simulation and Reasoning Framework
Code Proofs
Conclusion and Future Work
OUTLINE

1 INTRODUCTION

2 SIMULATION AND REASONING FRAMEWORK
   - x86 ISA Model
   - System Calls Model

3 CODE PROOFS

4 CONCLUSION AND FUTURE WORK
System Calls Model: Extending syscall

System calls in the real world

User Space (Ring 3)

5: MOV %rax, 0
6: SYSCALL
7: MOV %rbx, %rax

Kernel Space (Ring 0)

rcx ← rip
rip ← ia32_lstar
RPL ← 0

rip ← rcx
RPL ← 3
SYSRET
System Calls Model: Extending syscall

System calls in the real world

User Space (Ring 3)

5: MOV %rax, 0
6: SYSCALL
7: MOV %rbx, %rax

Kernel Space (Ring 0)

rcx ← rip
rip ← ia32_lstar
RPL ← 0

... SYSRET...

System calls in our x86 model

5: MOV %rax, 0
6: SYSCALL
7: MOV %rbx, %rax

...
Benefits of the System Call Model

- Useful for verifying **application programs** while assuming that services like I/O operations are provided reliably by the OS

  We check such assumptions during co-simulations.
**BLENTS OF THE SYSTEM CALL MODEL**

- **Useful for verifying application programs** while assuming that services like I/O operations are provided reliably by the OS
  
  We check such assumptions during co-simulations.

- **Removes** the complexity of low-level interactions between the OS and the processor
  
  - Faster simulation
  - Simpler reasoning
Benefits of the System Call Model

- Useful for verifying application programs while assuming that services like I/O operations are provided reliably by the OS
  
  We check such assumptions during co-simulations.

- Removes the complexity of low-level interactions between the OS and the processor
  - Faster simulation
  - Simpler reasoning

- Provides the same abstraction for reasoning as is provided by an OS for programming
Recall: system calls are non-deterministic from the point of view of a programmer

We need to be able to:

1. **Efficiently execute** runs of a program with system calls on concrete data, and
2. **Formally reason** about such a program given symbolic data
In execution mode, the model interacts directly with the OS.
**System Calls: Execution Mode**

- In **execution mode**, the model interacts directly with the OS.
- System call service is provided by *raw Lisp* functions to obtain "real" results from the OS.
**System Calls: Execution Mode**

- In **execution mode**, the model interacts directly with the OS.
- System call service is provided by *raw Lisp* functions to obtain "real" results from the OS.
- Simulation of all instructions other than *syscall* happens within ACL2 (and hence, Lisp).
System Calls: Execution Mode

- These raw Lisp functions should not be used for reasoning since they are impure.
System Calls: Execution Mode

- These raw Lisp functions should not be used for reasoning since they are impure.

- It is critical for our framework to prohibit proofs of theorems that unconditionally state that some system call returns a specific value.
**System Calls: Logical Mode**

- The **logical mode** incorporates an environment `env` field into the x86 state.
**System Calls: Logical Mode**

- The **logical mode** incorporates an environment `env` field into the x86 state.

- `env` represents the part of the external world that affects or is affected by system calls.
**System Calls: Logical Mode**

- The **logical mode** incorporates an environment `env` field into the x86 state.

- `env` represents the part of the external world that affects or is affected by system calls.

- Kind of theorems about system calls that can be proved:
  
  Given a **particular characterization of the environment**, a system call returns some specific value.
RELATIONSHIP: EXECUTION & LOGICAL MODE

- **Identical** for all instructions except syscall:
  
  All other instructions have the same definitions in both these modes.
Relationship: Execution & Logical Mode

- **Identical** for all instructions except syscall:
  
  All other instructions have the same definitions in both these modes.

- **Correspond** in the case of syscall instruction if:
  
  The env field in the logical mode is an accurate characterization of the real environment.
**RELATIONSHIP: EXECUTION & LOGICAL MODE**

- **Identical** for all instructions except **syscall**:
  
  All other instructions have the same definitions in both these modes.

- **Correspond** in the case of **syscall** instruction if:

  The `env` field in the logical mode is an **accurate characterization of the real environment**.

  Then, the execution of system calls produces the **same results** in the logical mode as in the execution mode.
**Task A:** Validate the logical mode against the execution mode
**System Calls Model Validation**

**Task A:** Validate the logical mode against the execution mode

- Extensive code reviews

- Comparing program runs in the execution mode to corresponding runs in the logical mode
**System Calls Model Validation**

Task B: Validate the **execution mode** against the processor + system call service provided by the OS.
System Calls Model Validation

Task B: Validate the execution mode against the processor + system call service provided by the OS

- Validating the functions that marshal the input arguments and return values from the raw Lisp functions
OUTLINE

1. INTRODUCTION

2. SIMULATION AND REASONING FRAMEWORK
   - x86 ISA Model
   - System Calls Model

3. CODE PROOFS

4. CONCLUSION AND FUTURE WORK
x86 Machine-Code Proofs using \texttt{env}

Word Count Program

\begin{equation*}
\text{preconditions}(\text{rip}_i, \text{x86}_i) \land \text{x86}_f = \text{x86-run}(\text{clk}(\text{x86}_i), \text{x86}_i) \implies \text{getNc}(\text{x86}_f) = \texttt{ncSpec}(\text{Offset}(\text{x86}_i), \text{Str}(\text{x86}_i), 0)
\end{equation*}
x86 Machine-Code Proofs using *env*

**Word Count Program**

\[
\text{preconditions}(\text{rip}_i, \text{x86}_i) \land \text{x86}_f = \text{x86}-\text{run}(\text{clk}(\text{x86}_i), \text{x86}_i) \\
\implies \text{getNc}(\text{x86}_f) = \text{ncSpec}(\text{Offset}(\text{x86}_i), \text{Str}(\text{x86}_i), 0)
\]

**Preconditions:** *env* specifies a subset of the file system.

1. File descriptor is valid.
2. File contents are terminated by a valid EOF character.
3. File is open in a mode that allows reading.
4. Initial file offset points to a location within the file contents.
AUTOMATION OF X86 MACHINE-CODE PROOFS

- Developed lemma libraries to **automate reasoning** about user-level code
- Example of a useful theorem that was proved automatically:
  
  The program does not modify **unintended** regions of memory.
OUTLINE

1. INTRODUCTION

2. SIMULATION AND REASONING FRAMEWORK
   - x86 ISA Model
   - System Calls Model

3. CODE PROOFS

4. CONCLUSION AND FUTURE WORK
CONCLUSION AND FUTURE WORK

- Mechanical verification of **user-level x86 machine-code programs** with our **evolving x86 ISA model**
CONCLUSION AND FUTURE WORK

- Mechanical verification of user-level x86 machine-code programs with our evolving x86 ISA model

- Formal analysis of user-level programs exhibiting non-determinism demonstrated to be tractable
  - SYSCALL, RDRAND instructions

- Plans for the immediate future:
  - Improve/add to our lemma libraries
  - Support SYSCALL and SYSRET on the ISA level
  - Simulate and then reason about kernel code
CONCLUSION AND FUTURE WORK

- Mechanical verification of user-level x86 machine-code programs with our evolving x86 ISA model

- Formal analysis of user-level programs exhibiting non-determinism demonstrated to be tractable
  - SYSCALL, RDRAND instructions

- Led to the development of ACL2 lemma libraries that help automate machine-code verification
CONCLUSION AND FUTURE WORK

- Mechanical verification of user-level x86 machine-code programs with our evolving x86 ISA model
- Formal analysis of user-level programs exhibiting non-determinism demonstrated to be tractable
  - SYSCALL, RDRAND instructions
- Led to the development of ACL2 lemma libraries that help automate machine-code verification
- Plans for the immediate future:
  - Improve/add to our lemma libraries
  - Support SYSCALL and SYSRET on the ISA level
  - Simulate and then reason about kernel code
SIMULATION AND FORMAL VERIFICATION OF x86 MACHINE-CODE PROGRAMS THAT MAKE SYSTEM CALLS

Shilpi Goel
Warren A. Hunt, Jr.
Matt Kaufmann
Soumava Ghosh

The University of Texas at Austin

THANK YOU!