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

22^{nd} October, 2014

OUTLINE

OUTLINE

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

```
...
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

```
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
```

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

```
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

$$preconditions(rip_i,\ x86_i) \ \wedge\ x86_f = x86\text{-}run(clk(x86_i),\ x86_i)$$
$$\Longrightarrow$$
$$getNc(x86_f) = \mathbf{ncSpec}(Offset(x86_i),\ Str(x86_i),\ 0)$$

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

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

Component	Description
registers	general-purpose, segment, debug, control, floating point, MMX, model-specific
rip	instruction pointer
flg	flags register
env	environment field
mem	memory

FORMALIZATION: STATE TRANSITION FUNCTION

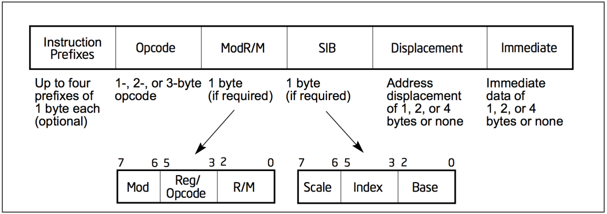


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

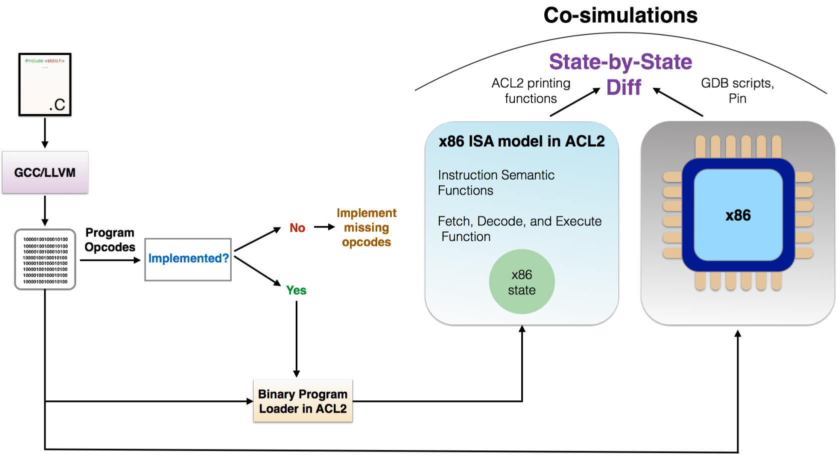
- State transition function: **fetch**, **decode** & **execute**
- Each instruction has its own semantic function

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

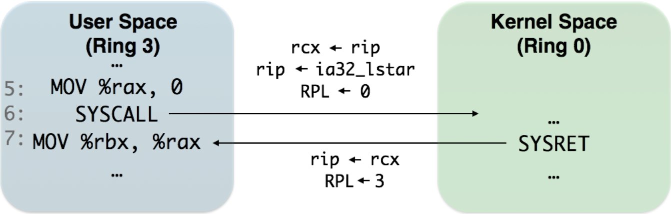
ASSESSING THE ACCURACY OF THE ISA MODEL



OUTLINE

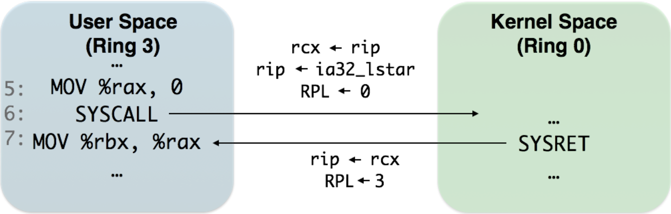
SYSTEM CALLS MODEL: EXTENDING SYSCALL

System calls in the real world

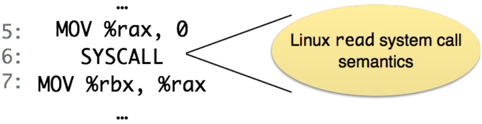


SYSTEM CALLS MODEL: EXTENDING SYSCALL

System calls in the real world



System calls in our x86 model



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.

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

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.

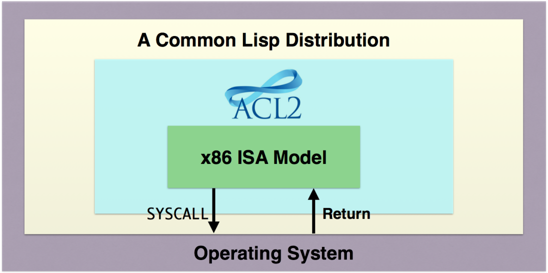
- **Removes** the complexity of **low-level interactions** between the OS and the processor
  - Faster simulation
  - Simpler reasoning

- Provides the same abstraction for reasoning as is provided by an OS for programming

Executing and Reasoning about System Calls

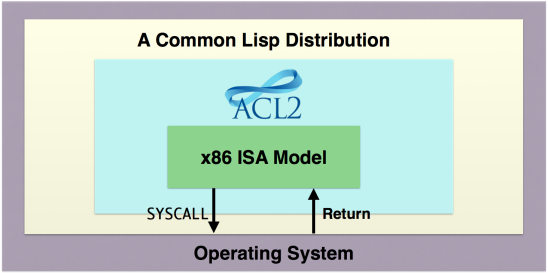
- Recall: system calls are non-deterministic from the point of view of a programmer
- We need to be able to:
  1. **Efficiently execute** runs of a program with system calls on concrete data, and
  2. **Formally reason** about such a program given symbolic data

System Calls: Execution Mode



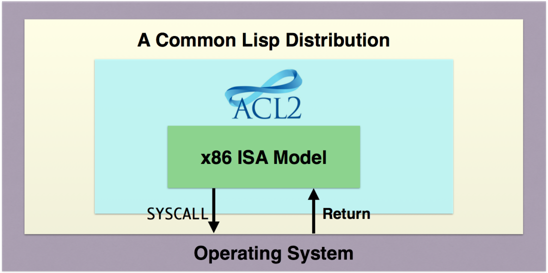
- In **execution mode**, the model **interacts directly with the OS**.

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`:

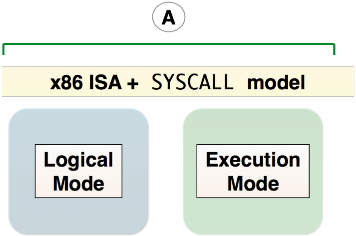
  All other instructions have the same definitions in both these modes.

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

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

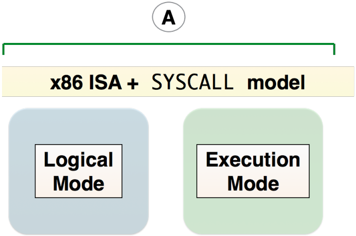
  Then, the execution of system calls produces the **same results** in the logical mode as in the execution mode.

SYSTEM CALLS MODEL VALIDATION



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

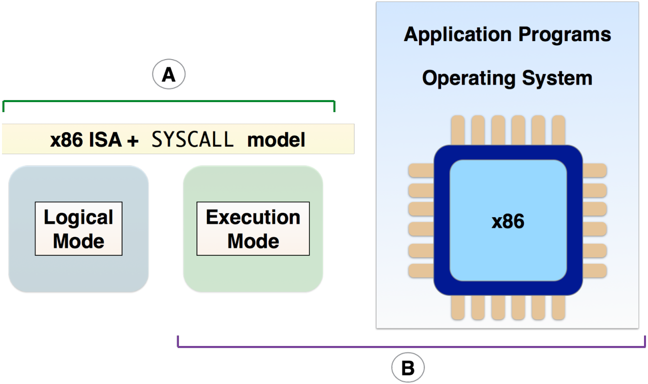
SYSTEM CALLS MODEL VALIDATION



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

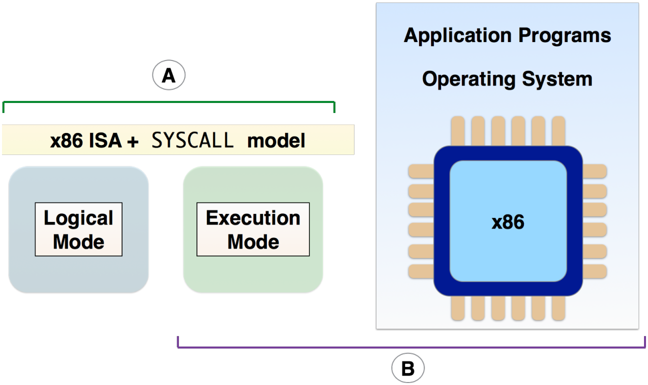
- Extensive code reviews
- Comparing program runs in the execution mode to **corresponding** runs in the logical mode

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

x86 Machine-Code Proofs using env

Word Count Program

Theorem

```
preconditions(rip_i, x86_i)  ∧  x86_f = x86-run(clk(x86_i), x86_i)
  ⟹
getNc(x86_f) = ncSpec(Offset(x86_i), Str(x86_i), 0)
```

x86 Machine-Code Proofs using `env`

Word Count Program

Theorem

$$\texttt{preconditions}(\texttt{rip}_i,\ \texttt{x86}_i)\ \wedge\ \texttt{x86}_f = \texttt{x86-run}(\texttt{clk}(\texttt{x86}_i),\ \texttt{x86}_i)$$
$$\Longrightarrow$$
$$\texttt{getNc}(\texttt{x86}_f) = \textbf{ncSpec}(\texttt{Offset}(\texttt{x86}_i),\ \texttt{Str}(\texttt{x86}_i),\ 0)$$

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

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

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!