Formal Verification of x86 Machine-Code Programs

Computer Architecture and Program Analysis

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
Software Formal Verification: proving or disproving that the implementation of a program meets its specification using mathematical techniques
Formal Verification of Software: Example 1

**Software Formal Verification:** proving or disproving that the *implementation* of a program meets its *specification* using mathematical techniques.

Suppose you needed to count the number of 1s in the binary representation of a natural number (*population count*).

**Specification:**

```
popcountSpec(v): [v: natural number]
if v <= 0 then
  return 0
else
  lsb = v & 1
  v   = v >> 1
  return (lsb + popcountSpec(v))
endif
```
Formal Verification of Software: Example 1

Specification:

```
popcountSpec(v): [v: natural number]
if v <= 0 then
    return 0
else
    lsb = v & 1
    v   = v >> 1
    return (lsb + popcountSpec(v))
endif
```
Formal Verification of Software: Example 1

Specification:

\[
\text{popcountSpec}(v): [v: \text{natural number}]
\]

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

Implementation:

```c
int popcount_32 (unsigned int v) {
    v = v - ((v >> 1) & 0x55555555);
    v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
    v = ((v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >> 24;
    return(v);
}
```

Source: Sean Anderson’s Bit-Twiddling Hacks
Formal Verification of Software: Example 1

Specification:

\[
\text{popcountSpec}(v): [v: \text{natural number}] \\
\text{if } v \leq 0 \text{ then} \\
\quad \text{return } 0 \\
\text{else} \\
\quad \text{lsb} = v \& 1 \\
\quad v = v >> 1 \\
\quad \text{return } (\text{lsb} + \text{popcountSpec}(v)) \\
\text{endif}
\]

Implementation:

```c
int popcount_32 (unsigned int v) {
    v = v - ((v >> 1) & 0x55555555);
    v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
    v = ((v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >> 24;
    return(v);
}
```

Do the specification and implementation behave the same way for all inputs?

Source: Sean Anderson’s Bit-Twiddling Hacks
Suppose you needed to check if a given natural number is a power of 2.

**Specification:**

```plaintext
isPowerOfTwoSpec(x): [x: natural number]
if x == 0 then
    return 0
else
    if x == 1 then
        return 1
    else
        if remainder(x,2) == 0 then
            return isPowerOfTwoSpec(x/2)
        else
            return 0
        endif
    endif
endif
```
Formal Verification of Software: Example 2

Can you trust your specification?

Source: Sean Anderson’s Bit-Twiddling Hacks
Formal Verification of Software: Example 2

Can you trust your specification?

**Correctness of `isPower0fTwoSpec`:**

1. If `isPower0fTwoSpec(v)` returns 1, then there exists a natural number \( n \) such that \( v = 2^n \).
2. If \( v = 2^n \), where \( n \) is a natural number, then `isPower0fTwoSpec(v)` returns 1.

*Source: Sean Anderson’s Bit-Twiddling Hacks*
Can you trust your specification?

**Correctness of isPower0fTwoSpec:**

1. If isPower0fTwoSpec(v) returns 1, then there exists a natural number $n$ such that $v = 2^n$.
2. If $v = 2^n$, where $n$ is a natural number, then isPower0fTwoSpec(v) returns 1.

**Implementation:**

```c
bool power0fTwo (long unsigned int v) {
    bool f;
    f = v && !(v & (v - 1));
    return f;
}
```

*Source: Sean Anderson’s Bit-Twiddling Hacks*
Formal Verification of Software: Example 2

Can you trust your specification?

*Correctness of `isPowerOfTwoSpec`:*

1. If `isPowerOfTwoSpec(v)` returns 1, then there exists a natural number \( n \) such that \( v = 2^n \).
2. If \( v = 2^n \), where \( n \) is a natural number, then `isPowerOfTwoSpec(v)` returns 1.

**Implementation:**

```cpp
bool powerOfTwo (long unsigned int v) {
    bool f;
    f = v && !(v & (v - 1));
    return f;
}
```

*Do the specification and implementation behave the same way for all inputs?*

Source: Sean Anderson’s Bit-Twiddling Hacks
Inspection of a Program’s Behavior

- **Testing:**
  - × Exhaustive analysis is infeasible

- **Formal Verification:**
  - ✓ Wide variety of techniques
    - Lightweight: e.g., checking if array indices are within bounds
    - Heavyweight: e.g., proving functional correctness
Example: Pop-Count Program

### Functional Correctness:

RAX = popcountSpec(v)

specification function

### popcountSpec(v):

[v: unsigned int]

```plaintext
if v <= 0 then
  return 0
else
  lsb = v & 1
  v   = v >> 1
  return (lsb + popcountSpec(v))
endif
```

### popcount_64:

```
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    %edx,%eax
  shr    $0x8,%eax
  and    $0x0f0f0f0f,%edx
  imul   $0x1010101,%edx,%edx
  imul   $0x1010101,%eax,%eax
  shr    $0x18,%edx
  shr    $0x18,%eax
  add    %edx,%eax
  retq
```
Case Study: Pop-Count Program

(defthm x86-popcount-64-symbolic-simulation
  (implies
   (and (x86p x86)
        (equal (model-related-error x86) nil)
        (unsigned-byte-p 64 n)
        (equal n (read 'register *rdi* x86))
        (equal *popcount-64-program*
               (read 'memory
                    (address-range
                     (read 'pc x86)
                     (len *popcount-64-program* x86))))
        (equal (read 'register *rax* (x86-run *num-of-steps* x86))
               (popcountSpec n))))
Heavyweight Formal Verification
Heavyweight Formal Verification

- Build a mathematical or *formal model* of programs
- Prove *theorems* about this model in order to establish program properties
Heavyweight Formal Verification

- Build a mathematical or **formal model** of programs
- Prove **theorems** about this model in order to establish program properties
Heavyweight Formal Verification

- Build a mathematical or \textit{formal model} of programs
- Prove \textit{theorems} about this model in order to establish program properties

\textbf{Instruction Set Architecture:} interface between hardware and software
- Defines the machine language
- Specification of state (registers, memory), machine instructions, instruction encodings, etc.
Heavyweight Formal Verification

- Build a mathematical or **formal model** of programs
- Prove **theorems** about this model in order to establish program properties

**Instruction Set Architecture:** interface between hardware and software
- Defines the machine language
- Specification of state (registers, memory), machine instructions, instruction encodings, etc.

- An ISA model specifies the behavior of each machine instruction in terms of *effects made to the processor state*. 
Heavyweight Formal Verification

- Build a mathematical or **formal model** of programs
- Prove **theorems** about this model in order to establish program properties

**Instruction Set Architecture:** interface between hardware and software
- Defines the machine language
- Specification of state (registers, memory), machine instructions, instruction encodings, etc.

- An ISA model specifies the behavior of each machine instruction in terms of *effects made to the processor state*.
- All high-level programs compile down to machine-code programs.
  - A program is just a sequence of machine instructions.
Heavyweight Formal Verification

- Build a mathematical or **formal model** of programs
- Prove **theorems** about this model in order to establish program properties

**Instruction Set Architecture:** interface between hardware and software
- Defines the machine language
- Specification of state (registers, memory), machine instructions, instruction encodings, etc.

- An ISA model specifies the behavior of each machine instruction in terms of **effects made to the processor state**.
- All high-level programs compile down to machine-code programs.
  - A program is just a sequence of machine instructions.
- We can reason about a program by **inspecting the cumulative effects of its constituent instructions on the machine state**.
Why Not Use Abstract Machine Models?

Abstract vs. Concrete Machine Models

**Machine Models**

![Diagram of Machine Models]

**Data**
1) char  
2) int, float  
3) double  
4) struct, array  
5) pointer

**Control**
1) loops  
2) conditionals  
3) switch  
4) proc. call  
5) proc. return

**Assembly**

![Diagram of Assembly]

1) byte  
2) 2-byte word  
3) 4-byte long word  
4) contiguous byte allocation  
5) address of initial byte  
1) branch/jump  
2) call  
3) ret
Why Not Use Abstract Machine Models?

Abstract vs. Concrete Machine Models

Machine Models

- C
- Memory
- Processor

Data
1) char
2) int, float
3) double
4) struct, array
5) pointer

Control
1) loops
2) conditionals
3) switch
4) proc. call
5) proc. return

Assembly

- Memory
- Stack
- Regs
- Processor
- ALU

- 1) byte
- 2) 2-byte word
- 3) 4-byte long word
- 4) contiguous byte allocation
- 5) address of initial byte

1) branch/jump
2) call
3) ret
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 programs
- E.g., correctness w.r.t. behavior, security, resource usage, etc.
Overview

**Goal:** Specify and verify properties of x86 programs
- E.g., correctness w.r.t. behavior, security, resource usage, etc.

- **Program property:** statement about a program’s behavior
  - One state, set of states, relationship between a set of final & initial states
Overview

**Goal:** Specify and verify properties of x86 programs
- E.g., correctness w.r.t. behavior, security, resource usage, etc.

- **Program property:** statement about a program’s behavior
  - One state, set of states, relationship between a set of final & initial states

- **Program’s computation:** how the execution of each instruction transforms one state to another

*specify:* in terms of states of computation
Overview

**Goal:** Specify and verify properties of x86 programs
- E.g., correctness w.r.t. behavior, security, resource usage, etc.

- **Program property:** statement about a program’s behavior
  - One state, set of states, relationship between a set of final & initial states

- **Program’s computation:** how the execution of each instruction transforms one state to another

- **Symbolic Executions:** a final (or next) x86 state is described in terms of *symbolic updates* made to the initial x86 state
  - Allows consideration of many, if not all, possible executions at once
Formal Tool Used: ACL2 Theorem-Proving System

- ACL²: A Computational Logic for Applicative Common Lisp
  - Programming Language
  - Mathematical Logic
  - Mechanical Theorem Prover

- See ACL2 Home Page for more details.
  - Extensive documentation!
  - ACL2 Research Group located at GDC 7S
x86 ISA Model
**Interpreted-Style Operational Semantics**: The x86 ISA model is a machine-code interpreter written in ACL2's formal logic.

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

Recursively defined interpreter that specifies the x86 model

\texttt{run(n, x86):}
\texttt{if n == 0 then}
\texttt{	return x86}
\texttt{else}
\texttt{	if model-related error encountered then}
\texttt{		return x86}
\texttt{	else}
\texttt{		run(n - 1, step(x86))}
\texttt{end if}
\texttt{end if}
State-transition function that corresponds to the execution of a single x86 instruction

```python
step(x86):
pc = rip(x86)
[prefixes, opcode, ... , imm] = Fetch-and-Decode(pc, x86)
case opcode:
    #x00 -> add-semantic-fn(prefixes, ... , imm, x86)
    ....
    #xFF -> inc-semantic-fn(prefixes, ... , imm, x86)
```
**Instruction Semantic Functions**

- A semantic function describes the effects of executing an instruction.
  - Input: x86 state and decoded parts of the instruction
  - Output: next x86 state
- Every instruction has its own semantic function.

```python
def add_semantic_fn(prefixes, ..., imm, x86):
    operand1 = getOperand1(prefixes, ..., imm, x86)
    operand2 = getOperand2(prefixes, ..., imm, x86)
    resultSum = fix(operand1 + operand2, ...)
    resultFlags = computeFlags(operand1, operand2, result, x86)
    x86 = updateState(resultSum, dst, resultFlags)
    return x86
```
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

running tests on x86 machines
x86 State

Focus: Intel’s 64-bit mode

Figure 3-2. 64-Bit Mode Execution Environment

Source: Intel Manuals
**x86 State**

**Focus: Intel’s 64-bit mode**

---

**Basic Program Execution Registers**

- **Sixteen 64-bit Registers**
- **General-Purpose Registers**
- **Six 16-bit Registers**
- **Segment Registers**
- **FPU Registers**
  - **Eight 80-bit Registers**
  - **Floating-Point Data Registers**
- **MMX Registers**
  - **Eight 64-bit Registers**
- **XMM Registers**
  - **Sixteen 128-bit Registers**
  - **32-bits**
  - **MXCSR Register**

**Address Space**

- $2^{64}$

---

**Source:** Intel Manuals
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.
Symbolic Execution
Supporting Symbolic Execution

Rules (theorems) describing interactions between these reads and writes to the x86 state enable *symbolic execution* of programs.
Read-over-Write Theorem #1

non-interference

Program Order

memory
Read-over-Write Theorem #1

non-interference

Program Order

memory

$W_i(x)$
Read-over-Write Theorem #1

non-interference

Program Order

\[ W_i(x) \quad R_j: y \]

memory
Read-over-Write Theorem #2

Overlap

Program Order

memory
Read-over-Write Theorem #2

Program Order

$W_i(x)$

Overlap

memory
Read-over-Write Theorem #2

Program Order

\[ W_i(x) \]

\[ R_i: x \]

overlap

memory
Write-over-Write Theorem #1

independent writes commute safely
Write-over-Write Theorem #1

independent writes commute safely

Program Order

memory

$W_i(x)$
Write-over-Write Theorem #1

independent writes commute safely

Program Order

memory

$W_i(x)$

$W_j(y)$
Write-over-Write Theorem #1

independent writes commute safely

Program Order

$W_i(x)$

Program Order

$W_j(y)$
Write-over-Write Theorem #1

independent writes commute safely

Program Order

\[ W_i(x) \]

memory

= 

Program Order

\[ W_j(y) \]

memory
Write-over-Write Theorem #1

independent writes commute safely

Program Order

\[ W_i(x) \]

\[ W_j(y) \]

memory

Program Order

\[ W_i(x) \]

\[ W_j(y) \]

memory
Write-over-Write Theorem #2

visibility of writes

Program Order

memory
Write-over-Write Theorem #2

visibility of writes

Program Order

memory

$W_i(x)$
Write-over-Write Theorem #2

visibility of writes

Program Order

memory

$W_i(x)$

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

visibility of writes

Program Order

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

Program Order

memory

memory
Write-over-Write Theorem #2

visibility of writes

Program Order

memory

Program Order

memory
Symbolic Execution

These read-over-write and write-over-write lemmas operate on symbolic expressions that describe the program’s behavior.

(implies
  (preconditions loc val x86)
  (let ((old-rbx (read 'register *rbx* x86))
        (old-pc (read 'pc x86)))
   (equal
    (x86-run (clk) x86)
    (write 'register *rax* old-rbx
      (write 'pc (+ 18 old-pc)
        (write 'memory loc val x86))))))

Also, we can project out relevant parts of the resulting state.
Conclusions
What I Haven’t Talked About Today...

1. **How to prove theorems using a mechanical theorem prover**
   - Useful to reason about both hardware and software
   - Fall’16 Grad-level Course: *Programming Languages*
   - Spring’17 Grad-level Course: *Recursion and Induction*

2. **Supervisor-mode features of the x86 ISA**
   - Useful for developing and analyzing kernel programs
   - An advanced architecture class
   - An OS class
## 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>
Resources

We have exciting research and engineering projects in this area! Please feel free to email if you want to know more.

- See ACL2 Home Page
- Talk to people on GDC 7S
- See some publications
Publications


Shilpi Goel. *Formal Verification of Application and System Programs Based on a Validated x86 ISA Model*. Ph.D. Dissertation, The University of Texas at Austin, 2016
[Source Code]

Github

[Documentation]

x86isa in the ACL2+Community Books Manual