Formal Verification and Computer Architecture

A Validated Formal Model of the x86 ISA for Analyzing Computing Systems

Shilpi Goel
shilpi@centtech.com
Formal Verification Engineer
Centaur Technology, Inc.
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
Formal Verification:

Proving or disproving that the *implementation* of a program meets its *specification* using mathematical techniques
Formal Verification

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 $v$ (i.e., $v$’s *population count*).

E.g.,: Population count of 15 ($0b1111$) = 4
     Population count of  8 ($0b1000$) = 1

**Specification:**

```plaintext
popcountSpec(v): [v: natural number]
if v <= 0 then
    return 0
else
    lsb = v & 1
    v   = v >> 1
    return (lsb + popcountSpec(v))
endif
```
Pop-Count Computation

Specification:

```latex
popcountSpec(v): [v: natural number]
if v <= 0 then
  return 0
else
  lsb = v & 1
  v   = v >> 1
  return (lsb + popcountSpec(v))
endif
```

Source: Sean Anderson's Bit-Twiddling Hacks
Pop-Count Computation

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);
}
```

Source: Sean Anderson’s Bit-Twiddling Hacks
Pop-Count Computation

Specification:

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

Implementation:

```c
int popcount_32(unsigned int v) {
    v = v - ((v >> 1) & 0x55555555);
    v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
    v = ((v + (v >> 4) & 0xF0F0F0F0) * 0x1010101) >> 24;
    return v;
}
```

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

Source: Sean Anderson’s Bit-Twiddling Hacks
Specification and Implementation

Two very crucial points:
Specification and Implementation

Two very crucial points:

1. The specification should be **simple**!
   - Its correctness should be obvious.
Specification and Implementation

Two very crucial points:

1. The specification should be **simple**!
   - Its correctness should be obvious.

2. The specification and the implementation should **not** be the same!
   - Proving \( x = x \) isn’t useful.
Inspection of a Program’s Behavior

• Testing:
  ✗ Exhaustive testing is infeasible
    ‣ The pop-count program would require $4,294,967,296$ ($2^{32}$) tests!
    ‣ A binary function of two 32-bit numbers would require $18,446,744,073,709,551,616$ ($2^{64}$) tests!
• **Testing:**
  × Exhaustive testing is infeasible
    ‣ The pop-count program would require $4,294,967,296$ ($2^{32}$) tests!
    ‣ A binary function of two 32-bit numbers would require $18,446,744,073,709,551,616$ ($2^{64}$) tests!

• **Formal Verification:**
  ✓ Wide variety of techniques
    ‣ Lightweight: e.g., checking if array indices are within bounds
    ‣ Heavyweight: e.g., proving **functional correctness**
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);
}
## The Pop-Count Program: x86 Version

### `popcount_32`:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Machine Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>89 fa</td>
<td>mov %edi,%edx</td>
<td></td>
</tr>
<tr>
<td>89 d1</td>
<td>mov %edx,%ecx</td>
<td></td>
</tr>
<tr>
<td>d1 e9</td>
<td>shr %ecx</td>
<td></td>
</tr>
<tr>
<td>81 e1 55 55 55 55</td>
<td>and $0x55555555,%ecx</td>
<td></td>
</tr>
<tr>
<td>29 ca</td>
<td>sub %ecx,%edx</td>
<td></td>
</tr>
<tr>
<td>89 d0</td>
<td>mov %edx,%eax</td>
<td></td>
</tr>
<tr>
<td>c1 ea 02</td>
<td>shr $0x2,%edx</td>
<td></td>
</tr>
<tr>
<td>25 33 33 33 33</td>
<td>and $0x33333333,%eax</td>
<td></td>
</tr>
<tr>
<td>81 e2 33 33 33 33</td>
<td>and $0x33333333,%edx</td>
<td></td>
</tr>
<tr>
<td>01 c2</td>
<td>add %eax,%edx</td>
<td></td>
</tr>
<tr>
<td>89 d0</td>
<td>mov %edx,%eax</td>
<td></td>
</tr>
<tr>
<td>c1 e8 04</td>
<td>shr $0x4,%eax</td>
<td></td>
</tr>
<tr>
<td>01 c2</td>
<td>add %eax,%edx</td>
<td></td>
</tr>
<tr>
<td>48 89 f8</td>
<td>mov %rdi,%rax</td>
<td></td>
</tr>
<tr>
<td>48 c1 e8 20</td>
<td>shr $0x20,%rax</td>
<td></td>
</tr>
<tr>
<td>81 e2 0f 0f 0f 0f</td>
<td>and $0xf0f0f0f,%edx</td>
<td></td>
</tr>
<tr>
<td>89 c1</td>
<td>mov %eax,%ecx</td>
<td></td>
</tr>
<tr>
<td>d1 e9</td>
<td>shr %ecx</td>
<td></td>
</tr>
<tr>
<td>81 e1 55 55 55 55</td>
<td>and $0x55555555,%ecx</td>
<td></td>
</tr>
<tr>
<td>29 c8</td>
<td>sub %ecx,%eax</td>
<td></td>
</tr>
<tr>
<td>89 c1</td>
<td>mov %eax,%ecx</td>
<td></td>
</tr>
<tr>
<td>c1 e8 02</td>
<td>shr $0x2,%eax</td>
<td></td>
</tr>
<tr>
<td>81 e1 33 33 33 33</td>
<td>and $0x33333333,%ecx</td>
<td></td>
</tr>
<tr>
<td>25 33 33 33 33</td>
<td>and $0x33333333,%eax</td>
<td></td>
</tr>
<tr>
<td>01 c8</td>
<td>add %ecx,%eax</td>
<td></td>
</tr>
<tr>
<td>89 c1</td>
<td>mov %eax,%ecx</td>
<td></td>
</tr>
<tr>
<td>c1 e9 04</td>
<td>shr $0x4,%ecx</td>
<td></td>
</tr>
<tr>
<td>01 c8</td>
<td>add %ecx,%eax</td>
<td></td>
</tr>
<tr>
<td>25 0f 0f 0f 0f</td>
<td>and $0xf0f0f0f,%eax</td>
<td></td>
</tr>
<tr>
<td>69 d2 01 01 01 01</td>
<td>imul $0x1010101,%edx,%edx</td>
<td></td>
</tr>
<tr>
<td>69 c0 01 01 01 01</td>
<td>imul $0x1010101,%eax,%eax</td>
<td></td>
</tr>
<tr>
<td>c1 ea 18</td>
<td>shr $0x18,%edx</td>
<td></td>
</tr>
<tr>
<td>c1 e8 18</td>
<td>shr $0x18,%eax</td>
<td></td>
</tr>
<tr>
<td>01 d0</td>
<td>add %edx,%eax</td>
<td></td>
</tr>
<tr>
<td>c3</td>
<td>retq</td>
<td></td>
</tr>
</tbody>
</table>
The Pop-Count Program: x86 Version

**Functional Correctness:**
Final EAX = popcountSpec(Initial EDI)

**specification function**

```
popcountSpec(v):
  [v: unsigned int]
  if v <= 0 then
    return 0
  else
    lsb = v & 1
    v = v >> 1
    return (lsb + popcountSpec(v))
  endif
```
x86 Pop-Count: A Formal Statement of Correctness

Let:

1. \( x_{86_i} \) denote a well-formed initial x86 state;
2. \( EDI(x_{86_i}) = v \), where \( v \) is a 32-bit unsigned integer;
3. the entire pop-count program be located at a good memory location in \( x_{86_i} \);
4. \( PC(x_{86_i}) = \) the first instruction of this program.

Then:

Let \( x_{86_f} \) denote the final x86 state obtained after the pop-count program runs to completion.

\[ EAX(x_{86_f}) = \text{popcountSpec}(v) \]
x86 Pop-Count: A Formal Statement of Correctness

Let:

1. \( x_{86i} \) denote a well-formed initial x86 state;
2. \( \text{EDI}(x_{86i}) = v \), where \( v \) is a 32-bit unsigned integer;
3. the entire pop-count program be located at a good memory location in \( x_{86i} \);
4. \( \text{PC}(x_{86i}) = \) the first instruction of this program.

Then:
Let \( x_{86f} \) denote the final x86 state obtained after the pop-count program runs to completion.

\[
\text{EAX}(x_{86f}) = \text{popcountSpec}(v)
\]
x86 Pop-Count: A Formal Statement of Correctness

Let:

1. $x_{86_i}$ denote a well-formed initial x86 state;
2. $\text{EDI}(x_{86_i}) = v$, where $v$ is a 32-bit unsigned integer;
3. the entire pop-count program be located at a good memory location in $x_{86_i}$;
4. $\text{PC}(x_{86_i})$ == the first instruction of this program.

Then:
Let $x_{86_f}$ denote the final x86 state obtained after the pop-count program runs to completion.

$EAX(x_{86_f}) \equiv \text{popcountSpec}(v)$
What Else Can You Specify and Verify?

- What do you care about?
What Else Can You Specify and Verify?

- What do you care about?
- For example:
  - *Resource usage:*
    - How much memory is consumed during program execution? Is it a function of the inputs? [performance analysis]
What Else Can You Specify and Verify?

• What do you care about?
• For example:
  - **Resource usage:**
    ‣ How much memory is consumed during program execution? Is it a function of the inputs? [performance analysis]
  - **Program’s side-effects:**
    ‣ What values are left on the stack after the program terminates? Does the program “clean-up” after itself? [security analysis]
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
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 **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.
Behavior of an Instruction

• The ISA specifies the behavior of each machine instruction in terms of the effects made to the processor state.

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

1. **read** instruction from mem
2. **read** flags
3. **write** new value to pc

```
add %edi, %eax
```

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
Reasoning about Programs

• The ISA specifies the behavior of each machine instruction in terms of the 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.
x86 ISA Model
Why an x86 ISA Model?

• **Model-building** is an effective approach employed in many disciplines to perform various kinds of analysis. For example:
  - **Architectural models** are used to evaluate the strength of buildings, bridges, etc.
  - **Protein structural models** are used to predict the interaction among different kinds of proteins.
Why an x86 ISA Model?

- **Model-building** is an effective approach employed in many disciplines to perform various kinds of analysis. For example:
  - **Architectural models** are used to evaluate the strength of buildings, bridges, etc.
  - **Protein structural models** are used to predict the interaction among different kinds of proteins.
- Similarly, an x86 ISA model can be used to **analyze the behavior of x86 programs**.
Tool Used: ACL2 Theorem-Proving System

- ACL$^2$: 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

A Run of the x86 Interpreter that executes $k$ instructions

**Operational Semantics:** x86 ISA model is a machine-code interpreter written in ACL2’s formal logic

- **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
Run Function

Recursively defined interpreter that specifies the x86 model

\texttt{run(n, x86)}:
\begin{verbatim}
if n == 0 then
    return x86
else
    if model-related error encountered then
        return x86
    else
        run(n - 1, step(x86))
    end if
end if
\end{verbatim}
Step Function

State-transition function that corresponds to the execution of a single x86 instruction

\[
\text{step(x86):} \\
\text{pc} = \text{rip(x86)} \\
[\text{prefixes, opcode, ... , imm}] = \text{Fetch-and-Decode(pc, x86)} \\
\text{case opcode:} \\
\quad \#x00 \rightarrow \text{add-semantic-fn(prefixes, ... , imm, x86)} \\
\quad \ldots \ldots \\
\quad \#xFF \rightarrow \text{inc-semantic-fn(prefixes, ... , imm, x86)}
\]
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
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

NOTE: This document contains all seven volumes when evaluating your design needs.

---

__asm__ volatile

```
"stc\n\n"mov $0, %%eax\n\n"mov $0, %%ebx\n\n"mov $0, %%ecx\n\n"mov %4, %%ecx\n\n"mov %3, %%edx\n\n"mov %2, %%eax\n\n"rcl %\cl, %\al\n\n"cmoveb %\edx, %\ebx\n\n"mov %\eax, %\0\n\n"mov %\ebx, %\1\n```

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

Running tests on x86 machines
x86 State

**Focus:** Intel’s 64-bit mode

![Figure 3-2. 64-Bit Mode Execution Environment](image)

*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**

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

---

**Focus: Intel’s 64-bit mode**

**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
Reasoning about Programs: Symbolic Execution
What is Symbolic Execution?

- **Concrete Execution**: a usual program run, with concrete values
  - $\text{popcount}_32(15) = 4$
What is Symbolic Execution?

- **Concrete Execution**: a usual program run, with concrete values
  - \( \text{popcount}_32(15) = 4 \)

- **Symbolic Execution**: a final (or next) x86 state is described in terms of *symbolic updates* made to the initial x86 state
  - Consider many, if not all, possible concrete executions at once
What is Symbolic Execution?

- **Concrete Execution**: a usual program run, with concrete values
  - $\text{popcount}_32(15) = 4$

- **Symbolic Execution**: a final (or next) x86 state is described in terms of *symbolic updates* made to the initial x86 state
  - Consider many, if not all, possible concrete executions at once

It’s nothing but algebra really:

- The step, run, and instruction semantic functions are big *mathematical functions*.
- Values in the x86 state can be thought of as *unknowns*.
- Solve equations involving these functions and unknowns to *derive relationships among various unknowns*. 
Symbolic Execution: Example

- Consider this small 2-instruction position-independent x86 program.
  
  \[
  \begin{align*}
  \text{addr:} & \quad 0xf8 \quad // \text{CLC: clear the carry flag} \\
  \text{addr} + 1: & \quad 0xf9 \quad // \text{STC: set the carry flag}
  \end{align*}
  \]
Symbolic Execution: Example

• Consider this small 2-instruction position-independent x86 program.
  
  \[
  \text{addr: } \quad 0xf8 \quad \text{// CLC: clear the carry flag} \\
  \text{addr + 1: } \quad 0xf9 \quad \text{// STC: set the carry flag}
  \]

• **Usual pre-conditions:** well-formed initial x86 state \((\text{x86}_i)\); program located at a good memory location \texttt{addr}; PC points to \texttt{addr}. 
Symbolic Execution: Example

- Consider this small 2-instruction position-independent x86 program.
  
  addr: 0xf8  // CLC: clear the carry flag
  addr + 1: 0xf9  // STC: set the carry flag

- **Usual pre-conditions**: well-formed initial x86 state ($x_{86i}$); program located at a good memory location addr; PC points to addr.

- What is $x_{861}$ — the state after the execution of the first instruction?
  
  let $x_{861a} := \text{write}(\text{CF}, 0, x_{86i})$
  $x_{861b} := \text{write}(\text{PC}, \text{addr} + 1, x_{861a})$
  then $x_{861} == x_{861b}$
Symbolic Execution: Example

• Consider this small 2-instruction position-independent x86 program.
  \[
  \begin{align*}
  \text{addr:} & \quad 0xf8 \quad // \text{CLC: clear the carry flag} \\
  \text{addr + 1:} & \quad 0xf9 \quad // \text{STC: set the carry flag}
  \end{align*}
  \]

• **Usual pre-conditions:** well-formed initial x86 state (\(x86_1\)); program located at a good memory location \(\text{addr}\); PC points to \(\text{addr}\).

• What is \(x86_1\) — the state after the execution of the first instruction?
  \[
  \begin{align*}
  \text{let } x86_{1a} & := \text{write(CF, 0, } x86_1) \\
  x86_{1b} & := \text{write(PC, } \text{addr + 1, } x86_{1a}) \\
  \text{then } x86_1 & == x86_{1b}
  \end{align*}
  \]

• What is \(x86_f\) — the final state after the execution of the second instruction?
  \[
  \begin{align*}
  \text{let } x86_{2a} & := \text{write(CF, 1, } x86_1) \\
  x86_{2b} & := \text{write(PC, } \text{addr + 2, } x86_{2a}) \\
  \text{then } x86_f & == x86_{2b}
  \end{align*}
  \]
Symbolic Execution: Example

let $x_{86_1a} := \text{write}(\text{CF}, 0, x_{86_i})$
$x_{86_1b} := \text{write}(\text{PC}, \text{addr} + 1, x_{86_1a})$
then $x_{86_1} == x_{86_1b}$

let $x_{86_2a} := \text{write}(\text{CF}, 1, x_{86_1})$
$x_{86_2b} := \text{write}(\text{PC}, \text{addr} + 2, x_{86_2a})$
then $x_{86_f} == x_{86_{2b}}$
Symbolic Execution: Example

let x86₁a := write(CF, 0, x86ᵢ)
    x86₁b := write(PC, addr + 1, x86₁a)
then x86₁ == x86₁b

let x86₂a := write(CF, 1, x86₁)
    x86₂b := write(PC, addr + 2, x86₂a)
then x86ᵢ in terms of x86ᵢ:
    x86ᵢ == write[PC, addr + 2,
        write(CF, 1,
            write{PC, addr + 1,
                write(CF, 0, x86ᵢ)})]]
Symbolic Execution: Example

let $x_{861a} := \text{write}(\text{CF}, 0, x_{86i})$

$x_{861b} := \text{write}(\text{PC}, \text{addr} + 1, x_{861a})$

then $x_{861} == x_{861b}$

let $x_{862a} := \text{write}(\text{CF}, 1, x_{861})$

$x_{862b} := \text{write}(\text{PC}, \text{addr} + 2, x_{862a})$

then $x_{86f} == x_{862b}$

$x_{86f}$ in terms of $x_{86i}$:

$x_{86f} == \text{write}[\text{PC}, \text{addr} + 2, \text{write}(\text{CF}, 1, \text{write}\{\text{PC}, \text{addr} + 1, \text{write}(\text{CF}, 0, x_{86i})\})]$

After simplifying:

$x_{86f} == \text{write}[\text{PC}, \text{addr} + 2, \text{write}(\text{CF}, 1, x_{86i})]$
Component Projections

\[ x_{86f} \equiv \text{write}[\text{PC}, \text{addr} + 2, \text{write}(\text{CF}, 1, x_{86i})] \]

Can project out “interesting” components from the final state:

\[
\begin{align*}
\text{read}(\text{CF}, x_{86f}) & \equiv \text{read}(\text{CF}, \text{write}[\text{PC}, \text{addr} + 2, \\
& \quad \text{write}(\text{CF}, 1, x_{86i})]) \\
& \equiv \text{read}(\text{CF}, \text{write}(\text{CF}, 1, x_{86i})) \\
& \equiv 1
\end{align*}
\]
Efficient Symbolic Execution

- These symbolic expressions can get pretty large.
  - Large expressions are time-consuming to read and manipulate.

- **Rules** describing interactions between reads and writes with **symbolic values** to the x86 state enable **efficient** symbolic execution.
  - These rules curtail the size of the symbolic expressions.

- Two basic kinds of rules:
  1. Read-over-Write
  2. Write-over-Write
Read-over-Write Theorem #1

non-interference

Program Order

memory
Read-over-Write Theorem #1

non-interference

Program Order

$W_i(x)$

memory
Read-over-Write Theorem #1

non-interference

Program Order

memory

\[ W_i(x) \]

\[ R_j: y \]
Read-over-Write Theorem #2

overlap

Program Order

memory
Read-over-Write Theorem #2

Program Order

memory

overlap

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

Program Order

memory

\[ W_i(x) \]

\[ R_i: x \]

overlap
Write-over-Write Theorem #1

independent writes commute safely

Program Order

memory
Write-over-Write Theorem #1

independent writes commute safely

$W_i(x)$

Program Order

memory
Write-over-Write Theorem #1

independent writes commute safely

Program Order

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

memory
Write-over-Write Theorem #1

independent writes commute safely

Program Order

$W_i(x)$

$W_j(y)$

=  

Program Order
Write-over-Write Theorem #1

independent writes commute safely

Program Order

$W_i(x)$

$W_j(y)$

Program Order

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

independent writes commute safely

Program Order

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

Program Order

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

visibility of writes

Program Order

memory

i
Write-over-Write Theorem #2

visibility of writes

Program Order

\[ W_i(x) \]

memory
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

Program Order
Write-over-Write Theorem #2

visibility of writes

Program Order

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

memory

Program Order

\[ W_i(y) \]

memory
What I Haven’t Talked About Today

• **How to prove theorems using a mechanical theorem prover**
  - Useful to reason about computing systems, mathematical concepts, etc.
  - *Programming Languages*
  - *Recursion and Induction*

• **Supervisor-mode features of the x86 ISA**
  - Useful for developing and analyzing kernel programs
  - *Advanced Computer Architecture*
  - *Operating Systems*
Conclusions
Resources

There are exciting research and engineering projects in this area!

- See the [ACL2 Home Page](https://www.cs.utexas.edu/users/ai-resource-guide/)
- Talk to people on GDC 7S
- See some publications
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>
x86 Hardware Verification

What We Do at
Centaur Technology

- Centaur makes **low power, low cost, high performance x86 processors**.
- Founded as a start-up in Austin in 1995 to lower the cost of x86 processors
- 100 people; constant over the past 10 years
I work in the 4-person Formal Verification group at Centaur.

- Briefly: we verify that (parts of) Centaur’s micro-architecture correctly implement (parts of) the x86 instruction-set architecture.
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