A Formal Model of the X86 ISA for Binary Program Verification

Shilpi Goel

The University of Texas at Austin

April 2, 2013
OUTLINE

INTRODUCTION

RELATED WORK

X86 ISA MODEL
  X86 INSTRUCTION INTERPRETER
  EXECUTING PROGRAMS ON X86 MODEL

BINARY PROGRAM VERIFICATION
  CLOCK FUNCTION APPROACH
  SYMBOLIC EXECUTION

CONCLUSION AND FUTURE WORK
OUTLINE

INTRODUCTION

RELATED WORK

X86 ISA MODEL

X86 INSTRUCTION INTERPRETER
EXECUTING PROGRAMS ON X86 MODEL

BINARY PROGRAM VERIFICATION

CLOCK FUNCTION APPROACH
SYMBOLIC EXECUTION

CONCLUSION AND FUTURE WORK
Our Goals

1. Develop an accurate, non-idealized model of the x86 Instruction Set Architecture (ISA)
Our Goals

1. Develop an accurate, non-idealized model of the x86 Instruction Set Architecture (ISA)

2. Develop automated procedures for reasoning about x86 machine code
OUR GOALS

1. Develop an accurate, non-idealized model of the x86 Instruction Set Architecture (ISA)

2. Develop automated procedures for reasoning about x86 machine code

Infrastructure for verification of linux utilities like cat and od
Why Do We Care?

- Analysis of high-level programs is not good enough.
**Why Do We Care?**

- Analysis of high-level programs is not good enough.
- High-level programs are not always available.
Why Do We Care?

- Analysis of high-level programs is not good enough.
- High-level programs are not always available.
- **Formal verification** of machine code!
**Why Do We Care?**

- Analysis of high-level programs is not good enough.
- High-level programs are not always available.
- **Formal verification** of machine code!
Why Do We Care?

- Analysis of high-level programs is not good enough.
- High-level programs are not always available.
- **Formal verification** of machine code!
  - Formal Model of the x86 ISA
Why Do We Care?

- Analysis of high-level programs is not good enough.
- High-level programs are not always available.
- Formal verification of machine code!
  - Formal Model of the x86 ISA
  - Reason about machine code on this model
OUTLINE

INTRODUCTION

RELATED WORK

X86 ISA MODEL

X86 INSTRUCTION INTERPRETER

EXECUTING PROGRAMS ON X86 MODEL

BINARY PROGRAM VERIFICATION

CLOCKS FUNCTION APPROACH

SYMBOLIC EXECUTION

CONCLUSION AND FUTURE WORK
Machine Code Verification on Formal Processor Models

- [Yu '96]: Berkeley String Library
- [Bevier '87]: OS Kernel Verification
- [Feng, et al. '09]: Hoare-style logics
- [Myreen '08]: Decompilation into Logic
- [Matthews, et al. '06]: VCG via Theorem Proving

Degree of Automation in Reasoning

Manual vs. Automated
Our Goals, revisited

1. Develop an **accurate**, non-idealized, formal, and executable model of the x86 ISA
1. Develop an accurate, **non-idealized**, formal, and executable model of the x86 ISA
### Our Goals, Revisited

1. Develop an accurate, non-idealized, *formal*, and executable model of the x86 ISA
Our Goals, revisited

1. Develop an accurate, non-idealized, formal, and executable model of the x86 ISA
Our Goals, revisited

1. Develop an accurate, non-idealized, formal, and executable model of the x86 ISA
Our Goals, revisited

1. Develop an accurate, non-idealized, formal, and **executable** model of the x86 ISA

   ▶ Specifications: Intel’s Software Developer’s Manuals
1. Develop an accurate, non-idealized, formal, and executable model of the x86 ISA

- Specifications: Intel’s Software Developer’s Manuals
- ~4000 pages of prose
Our Goals, revisited

1. Develop an accurate, non-idealized, formal, and executable model of the x86 ISA
   - Specifications: Intel’s Software Developer’s Manuals
   - ~4000 pages of prose
   - Model should emulate the real machine
Our Goals, Revisited

1. Develop an accurate, non-idealized, formal, and executable model of the x86 ISA

   ▶ Specifications: Intel’s Software Developer’s Manuals
   ▶ ~4000 pages of prose
   ▶ Model should emulate the real machine
   ▶ Co-simulations
1. Develop an accurate, non-idealized, formal, and executable model of the x86 ISA

- Specifications: Intel’s Software Developer’s Manuals
- ~4000 pages of prose
- Model should emulate the real machine
- Co-simulations
- Need executability to do co-simulations
Our Goals, revisited

1. Develop an accurate, non-idealized, formal, and executable model of the x86 ISA
Our Goals, revisited

1. Develop an accurate, non-idealized, formal, and executable model of the x86 ISA

2. Develop automated procedures for reasoning about x86 machine code
Our Goals, revisited

1. Develop an accurate, non-idealized, formal, and executable model of the x86 ISA

2. Develop automated procedures for reasoning about x86 machine code
   - Functional correctness of machine code
Our Goals, Revisited

1. Develop an accurate, non-idealized, formal, and executable model of the x86 ISA

2. Develop automated procedures for reasoning about x86 machine code
   - Functional correctness of machine code
   - Minimize lemma construction
OUTLINE

INTRODUCTION

RELATED WORK

X86 ISA MODEL

X86 INSTRUCTION INTERPRETER

EXECUTING PROGRAMS ON X86 MODEL

BINARY PROGRAM VERIFICATION

CLOCK FUNCTION APPROACH

SYMBOLIC EXECUTION

CONCLUSION AND FUTURE WORK
OUTLINE

INTRODUCTION

RELATED WORK

X86 ISA MODEL

X86 INSTRUCTION INTERPRETER

EXECUTING PROGRAMS ON X86 MODEL

BINARY PROGRAM VERIFICATION

CLOCK FUNCTION APPROACH

SYMBOLIC EXECUTION

CONCLUSION AND FUTURE WORK
FORMALIZING X86 ISA IN ACL2

ACL2?
FORMALIZING X86 ISA IN ACL2

ACL2?

- A Computational Logic for Applicative Common Lisp
FORMALIZING X86 ISA IN ACL2

ACL2?

▶ A Computational Logic for Applicative Common Lisp
▶ Programming language
FORMALIZING X86 ISA IN ACL2

ACL2?

- *A Computational Logic for Applicative Common Lisp*
- Programming language
- Mathematical logic
FORMALIZING X86 ISA IN ACL2

ACL2?

- *A Computational Logic for Applicative Common Lisp*
- Programming language
- Mathematical logic
- Mechanical theorem prover
Our x86 ISA model has been formalized using an interpreter approach to operational semantics.
Our x86 ISA model has been formalized using an *interpreter approach to operational semantics.*

Semantics of a program is given by the effect it has on the state of the machine.
Our x86 ISA model has been formalized using an *interpreter approach to 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.
## X86 State

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers</td>
<td>general-purpose, segment, debug, control, model-specific registers</td>
</tr>
<tr>
<td>rip</td>
<td>instruction pointer</td>
</tr>
<tr>
<td>flg</td>
<td>64-bit flags register</td>
</tr>
<tr>
<td>mem</td>
<td>physical memory (4096 TB)</td>
</tr>
</tbody>
</table>
Run Function

Recursively defined interpreter that specifies the x86 model
Run Function

Recursively defined interpreter that specifies the x86 model

run (n, x86):

if n == 0:
  return (x86)
else
  if halt instruction encountered:
    return (x86)
  else
    run (n - 1, step (x86))
Step Function

```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

- **INPUT:** x86 state
  Decoded components of the instruction

- **OUTPUT:** Next x86 state
**Instruction Semantic Functions**

- **INPUT:** x86 state
  Decoded components of the instruction

- **OUTPUT:** Next x86 state

- A semantic function describes the effects of executing an instruction.
Instruction Semantic Functions

- **Input:** x86 state
  - Decoded components of the instruction
- **Output:** Next x86 state

- A semantic function describes the effects of executing an instruction.
- Every instruction in the model has its own semantic function.
X86 MODEL

We use Intel’s Software Developer’s Manuals as our specification.

- 64-bit mode
X86 MODEL

We use Intel’s Software Developer’s Manuals as our specification.

- 64-bit mode
- Model entire $2^{52}$ bytes (4096 TB) of memory
X86 Model

We use Intel’s Software Developer’s Manuals as our specification.

- 64-bit mode
- Model entire $2^{52}$ bytes (4096 TB) of memory
- All addressing modes
X86 MODEL

We use Intel’s Software Developer’s Manuals as our specification.

- 64-bit mode
- Model entire $2^{52}$ bytes (4096 TB) of memory
- All addressing modes
- 118 user-mode instructions (219 opcodes)
X86 Model

We use Intel’s Software Developer’s Manuals as our specification.

- 64-bit mode
- Model entire $2^{52}$ bytes (4096 TB) of memory
- All addressing modes
- 118 user-mode instructions (219 opcodes)
- Execution speed: ~3.3 million instructions/second
X86 MODEL

We use Intel’s Software Developer’s Manuals as our specification.

- 64-bit mode
- Model entire $2^{52}$ bytes (4096 TB) of memory
- All addressing modes
- 118 user-mode instructions (219 opcodes)
- Execution speed: ~3.3 million instructions/second
- +40,000 lines of code
OUTLINE

INTRODUCTION

RELATED WORK

X86 ISA MODEL

X86 INSTRUCTION INTERPRETER

EXECUTING PROGRAMS ON X86 MODEL

BINARY PROGRAM VERIFICATION

CLOCK FUNCTION APPROACH

SYMBOLIC EXECUTION

CONCLUSION AND FUTURE WORK
EXECUTING BINARY PROGRAMS ON X86 MODEL

Co-simulation

State-by-State Diff

ACL2 printing functions

GDB scripts, Formatting functions

X86 Model in ACL2

X86 Run Function
X86 Step Function
X86 Instruction Semantic Functions

X86 State

Registers
Instruction Pointer
Flags
Memory

Real Machine

Machine State

Registers
Instruction Pointer
Flags
Memory

Subset Operation

Are program opcodes a subset of implemented opcodes?

No --- implement required opcodes

Yes

Transform Operation

Objdump, Shell Scripts, Python

(defconst *program-binary* ...)

Implemented Opcodes

Program Opcodes

GCC/LLVM Compiler

010110
110011
101000
0001

01101

20/33
OUTLINE

INTRODUCTION

RELATED WORK

X86 ISA MODEL
  X86 INSTRUCTION INTERPRETER
  EXECUTING PROGRAMS ON X86 MODEL

BINARY PROGRAM VERIFICATION
  CLOCK FUNCTION APPROACH
  SYMBOLIC EXECUTION

CONCLUSION AND FUTURE WORK
OUTLINE

INTRODUCTION

RELATED WORK

X86 ISA MODEL

X86 INSTRUCTION INTERPRETER

EXECUTING PROGRAMS ON X86 MODEL

BINARY PROGRAM VERIFICATION

CLOCKS FUNCTION APPROACH

SYMBOLIC EXECUTION

CONCLUSION AND FUTURE WORK
CODE PROOFS: CLOCK FUNCTION APPROACH

▶ Write the program’s specification
Code Proofs: Clock Function Approach

- Write the program’s specification
- Write the algorithm used in the program
Code Proofs: Clock Function Approach

- Write the program’s specification
- Write the algorithm used in the program
- Prove that the algorithm satisfies the specification
CODE PROOFS: CLOCK FUNCTION APPROACH

- Write the program’s specification
- Write the algorithm used in the program
- Prove that the algorithm satisfies the specification
- Define clock functions
CODE PROOFS: CLOCK FUNCTION APPROACH

- Write the program’s **specification**
- Write the **algorithm** used in the program
- Prove that the **algorithm satisfies the specification**
- Define **clock functions**
- Prove that the **program implements the algorithm**
CODE PROOFS: CLOCK FUNCTION APPROACH

▶ Write the program’s specification
▶ Write the algorithm used in the program
▶ Prove that the algorithm satisfies the specification
▶ Define clock functions
▶ Prove that the program implements the algorithm
▶ Prove that the program satisfies the specification
OUTLINE

INTRODUCTION

RELATED WORK

X86 ISA MODEL
  X86 INSTRUCTION INTERPRETER
  EXECUTING PROGRAMS ON X86 MODEL

BINARY PROGRAM VERIFICATION
  CLOCK FUNCTION APPROACH
  SYMBOLIC EXECUTION

CONCLUSION AND FUTURE WORK
SYMBOLOIC EXECUTION IN ACL2

- Symbolic Execution: Executing functions on symbolic data
Symbolic Execution in ACL2

- **Symbolic Execution**: Executing functions on symbolic data; can be used as a *proof procedure*
Symbolic Execution in ACL2

- **Symbolic Execution**: Executing functions on symbolic data; can be used as a *proof procedure*

- **GL**: verified framework for proving ACL2 theorems involving *finite objects*
Symbolic Execution in ACL2

- **Symbolic Execution**: Executing functions on symbolic data; can be used as a *proof procedure*

- **GL**: verified framework for proving ACL2 theorems involving finite objects
Symbolic Execution in ACL2

- **Symbolic Execution**: Executing functions on symbolic data; can be used as a *proof procedure*

- **GL**: verified framework for proving ACL2 theorems involving finite objects

- **Symbolic objects**: finite objects represented by boolean expressions
Symbolic Execution in ACL2

- **Symbolic Execution**: Executing functions on symbolic data; can be used as a *proof procedure*

- **GL**: verified framework for proving ACL2 theorems involving finite objects

- Symbolic objects: finite objects represented by boolean expressions
Symbolic Execution in ACL2

- **Symbolic Execution**: Executing functions on symbolic data; can be used as a *proof procedure*

- **GL**: verified framework for proving ACL2 theorems involving finite objects

- **Symbolic objects**: finite objects represented by boolean expressions

- Computations involving these symbolic objects done using **verified BDD operations**
DEMO

Automatic correctness proof for an x86 popcount binary program, for counting the number of non-zero bits in the bit-level representation of an unsigned integer input.
Code Proofs: Symbolic Execution Approach

- Write the program’s specification
Code Proofs: Symbolic Execution Approach

- Write the program’s specification
- Prove that the program satisfies the specification (fully automatic)
CODE PROOFS: SYMBOLIC EXECUTION APPROACH

- No lemma construction needed; proof done fully automatically
**Code Proofs: Symbolic Execution Approach**

- No lemma construction needed; proof done fully automatically
- Reason *directly* about semantics of programs (+40,000 LoC)
CODE PROOFS: SYMBOLIC EXECUTION APPROACH

- No lemma construction needed; proof done fully automatically
- Reason *directly* about semantics of programs (+40,000 LoC)
- Proofs of correctness of larger programs to be obtained compositionally using traditional theorem proving techniques
OUTLINE

INTRODUCTION

RELATED WORK

X86 ISA MODEL
  X86 INSTRUCTION INTERPRETER
  EXECUTING PROGRAMS ON X86 MODEL

BINARY PROGRAM VERIFICATION
  CLOCK FUNCTION APPROACH
  SYMBOLIC EXECUTION

CONCLUSION AND FUTURE WORK
CONCLUSION

- Executable, formal model of a significant subset of x86 ISA
CONCLUSION

- Executable, formal model of a significant subset of x86 ISA
- No simplification of the semantics of x86 instructions
CONCLUSION

- Executable, formal model of a significant subset of x86 ISA
- No simplification of the semantics of x86 instructions
- X86 ISA model capable of running and reasoning about real x86 binary programs
PAPERS

- [ACL2 Workshop’13]: S. Goel, W. Hunt, and M. Kaufmann
  Abstract Stobjs and Their Application to ISA Modeling

- [VSTTE’13]: S. Goel and W. Hunt
  Automated Code Proofs on a Formal Model of the X86
PAPERS

- [ACL2 Workshop’13]: S. Goel, W. Hunt, and M. Kaufmann
  *Abstract Stobjs and Their Application to ISA Modeling*

- [VSTTE’13]: S. Goel and W. Hunt
  *Automated Code Proofs on a Formal Model of the X86*
**Future Work**

- Add **system calls** to enable reasoning about I/O (open, read, write, etc.)
**Future Work**

- Add **system calls** to enable reasoning about I/O (open, read, write, etc.)

- Further **automate** the **co-simulation** framework
  - Automated test case generation
  - Enhance GDB mode framework
FUTURE WORK

▶ Add **system calls** to enable reasoning about I/O (open, read, write, etc.)

▶ Further **automate** the **co-simulation** framework
  ▶ Automated test case generation
  ▶ Enhance GDB mode framework

▶ Build automated binary **program annotation and instrumentation** tools
Future Work

- Add **system calls** to enable reasoning about I/O (open, read, write, etc.)
- Further **automate** the **co-simulation** framework
  - Automated test case generation
  - Enhance GDB mode framework
- Build automated binary **program annotation and instrumentation** tools
- Infrastructure for verification of linux utilities like `cat` and `od`
A Formal Model of the X86 ISA for
Binary Program Verification

Shilpi Goel

The University of Texas at Austin

April 2, 2013