A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture (ISA)

Sandeep Dasgupta ● Daejun Park ● Theodoros Kasampalis ● Vikram S. Adve ● Grigore Rosu
University of Illinois at Urbana Champaign
26 June 2019 @ PLDI’19
Formal ISA Semantics is Useful

Enables *direct* formal analysis of binary code

As opposed to analysing source code
Formal ISA Semantics is Useful

Validating ISA reference manuals or processor hardware

Using validation testing against an implementation
Verification or Translation Validation

- Compiler verification as in CompCert, CakeML
- e.g. Translation validation of compiler passes as in Necula et al. [PLDI’00]
## Example Formal ISA Semantics

<table>
<thead>
<tr>
<th>RISC ISAs</th>
<th>ISA</th>
<th>Formal Semantics</th>
<th>CISC ISAs</th>
<th>ISA</th>
<th>Formal Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>• seL4 '09</td>
<td></td>
<td>x86-32 (or IA32)</td>
<td>• ACL2 x86 '14</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SAIL '19</td>
<td></td>
<td></td>
<td>• Compcert '09</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Compcert '09</td>
<td></td>
<td></td>
<td>• TSL '13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• CakeML '15</td>
<td></td>
<td></td>
<td>• SAIL '17</td>
<td></td>
</tr>
<tr>
<td>RISC-V</td>
<td>• SAIL '19</td>
<td></td>
<td>x86-64</td>
<td>• ACL2 x86-64 '14</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Compcert '09</td>
<td></td>
<td></td>
<td>• Strata '16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• CakeML '18</td>
<td></td>
<td></td>
<td>• CakeML '14</td>
<td></td>
</tr>
<tr>
<td>MIPS</td>
<td>• SAIL '19</td>
<td></td>
<td></td>
<td>• Roessle et al. '19</td>
<td></td>
</tr>
<tr>
<td>POWER PC</td>
<td>• TSL '13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Compcert '09</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPARC</td>
<td>• TSL '13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Example Formal ISA Semantics

<table>
<thead>
<tr>
<th>RISC ISAs</th>
<th>Formally Semantics</th>
<th>CISC ISAs</th>
<th>Formal Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td></td>
<td>x86-32 (or IA32)</td>
<td>· ACL2 x86 '14</td>
</tr>
<tr>
<td></td>
<td>· seL4 '09</td>
<td></td>
<td>· CompCert '09</td>
</tr>
<tr>
<td></td>
<td>· SAIL '19</td>
<td></td>
<td>· TSL '13</td>
</tr>
<tr>
<td></td>
<td>· CompCert '09</td>
<td></td>
<td>· SAIL '17</td>
</tr>
<tr>
<td></td>
<td>· CakeML '15</td>
<td></td>
<td>· CakeML '14</td>
</tr>
<tr>
<td>RISC-V</td>
<td>· SAIL '19</td>
<td>x86-64</td>
<td>· ACL2 x86-64 '14</td>
</tr>
<tr>
<td></td>
<td>· CompCert '09</td>
<td></td>
<td>· Strata '16</td>
</tr>
<tr>
<td></td>
<td>· CakeML '18</td>
<td></td>
<td>· CakeML '14</td>
</tr>
<tr>
<td>MIPS</td>
<td>· SAIL '19</td>
<td></td>
<td>· Roessle et al. '19</td>
</tr>
<tr>
<td>POWER PC</td>
<td>· TSL '13</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>· CompCert '09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPARC</td>
<td>· TSL '13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Challenges: from ISA Spec to Semantics

- 3000+ pages of informal & description
- 996 unique mnemonics with 3736 variants
- Inconsistent behavior of variants
x86-64 Semantics: Previous Work

- **Direct semantics: Low instruction coverage**
  - Hunt & Goel [FMCAD’14] (~ 33%)
  - Strata semantics [PLDI’16] (~ 54%)
  - Roessle et al. [CPP’19] (~ 49%)
  - CakeML [POPL’14] (a small fraction)

- **Indirect semantics** (x86 → IR)
  - Bap, Remill, Angr etc.
Our Contribution

We defined the **most complete** and **thoroughly tested** formal semantics of **user-level** x86-64 ISA

github.com/kframework/X86-64-semantics
Our Contribution

We defined the **most complete** and **thoroughly tested** formal semantics of **user-level** x86-64 ISA

github.com/kframework/X86-64-semantics

- Most complete user-level support (3155 instruction variants)
Our Contribution

We defined the **most complete** and **thoroughly tested** formal semantics of **user-level** x86-64 ISA

- Most complete user-level support (3155 instruction variants)
- Thoroughly tested against hardware using 7000+ input states and GCC-c torture tests

[github.com/kframework/X86-64-semantics](github.com/kframework/X86-64-semantics)
Our Contribution

We defined the **most complete** and **thoroughly tested** formal semantics of **user-level** x86-64 ISA

- Most complete user-level support (3155 instruction variants)
- Thoroughly tested against hardware using 7000+ input states and GCC-c torture tests
- Found bugs in Intel manual and related projects

[github.com/kframework/X86-64-semantics](http://github.com/kframework/X86-64-semantics)
Our Contribution

We defined the **most complete** and **thoroughly tested** formal semantics of **user-level** x86-64 ISA

- Most complete user-level support (3155 instruction variants)
- Thoroughly tested against hardware using 7000+ input states and GCC-c torture tests
- Found bugs in Intel manual and related projects
- Demonstrated applicability to formal reasoning

[github.com/kframework/X86-64-semantics](https://github.com/kframework/X86-64-semantics)
Scope of Work (3155 / 3736)

- Supported (3155)
- Unsupported (581)

- General
- FC16
- FMA
- AVX2
- AVX
- SSE

Deprecated (336)
Concurrency & System (210)
Crypto (35)
Language semantics engineering framework (kframework.org)
Approach Overview
Approach Overview

Strata BVL* semantics

60% in scope

* BVL: Bit-vector logic
Approach Overview

Strata BVL* semantics

60% in scope

Modeling Unsupported artifacts

Validating semantics of instruction-variants

augmented & corrected semantics

* BVL: Bit-vector logic
**Approach Overview**

- **Strata BVL* semantics**
  - 60% in scope

  Modeling Unsupported artifacts
  
  Validating semantics of instruction-variants

  **augmented & corrected semantics**

  Formula simplification **
  
  count reduction

  **simplified semantics**

---

* BVL: Bit-vector logic

** 30+ simplification rules. BVL formula of shrxl with 8971 terms simplified to 7 terms
Approach Overview

- **Strata BVL* semantics**: 60% in scope

- **BVL → K translator**

- **Augmented & corrected semantics**

- **Formula simplification**
  - **Count reduction**

- **Simplified semantics**

* BVL: Bit-vector logic

** 30+ simplification rules. BVL formula of shrxl with 8971 terms simplified to 7 terms
Approach Overview

Strata BVL* semantics

60% in scope

Modeling Unsupported artifacts

Validating semantics of instruction-variants

augmented & corrected semantics

Formula simplification **

count reduction

simplified semantics

Validation

SMT Formula

Strata BVL* semantics

60% in scope

Augmented & corrected semantics

Formula simplification **

Count reduction

Simplified semantics

Validation

SMT Formula

* BVL: Bit-vector logic

** 30+ simplification rules. BVL formula of shrxl with 8971 terms simplified to 7 terms
Approach Overview

- Strata BVL* semantics
  - 60% in scope
  - Modeling Unsupported artifacts
    - Validating semantics of instruction-variants

- BVL → K translator
  - Formula simplification **
  - Count reduction

- Simplified semantics

- Intel informal spec
  - 40% in scope
  - Manually translation

- Augmented & corrected semantics

- Semantics in K

---

* BVL: Bit-vector logic

** 30+ simplification rules. BVL formula of shrxl with 8971 terms simplified to 7 terms
**Approach Overview**

**Strata BVL**
- Semantics: 60% in scope
- Modeling unsupported artifacts
- Validating semantics of instruction-variants

**Intel informal spec**
- 40% in scope
- Manually translation

**Augmented & Corrected Semantics**
- Formula simplification
- Count reduction

**Simplified Semantics**

**BVL → K translator**

- 3.5 man-months

**Modeling Unsupported Artifacts**

**Validating Semantics of Instruction-Variants**

**Intel Informal Spec**

**40% in Scope**

**Manually Translation**

- 2.5 man-months (~5200 rules)

---

* BVL: Bit-vector logic

** 30+ simplification rules. BVL formula of shrxl with 8971 terms simplified to 7 terms
Support Comparison

Projects hosting x86-64 ISA semantics. "Ideal" serves as a hypothetical baseline supporting all user-level and system-level instructions.

- **Ideal**: 100\% user-level, 100\% system-level
- **Goel et al.**: 33\% user-level, 12\% system-level
- **Strata**: 54\% user-level, 0\% system-level
- **Roessle et al.**: 49\% user-level, 0\% system-level
- **SAIL**: 12\% user-level, 0\% system-level
- **Current Work**: 89\% user-level, 0\% system-level

#user-level: ~3536
#system-level: ~200
Support Comparison

Projects hosting x86-64 ISA semantics. "Ideal" serves as a hypothetical baseline supporting all user-level and system-level instructions.

- **Ideal**
  - User-level: 100%
  - System-level: 100%

- **Goel et al.**
  - User-level: 33%
  - System-level: 12%

- **Strata**
  - User-level: 54%
  - System-level: 0%

- **Roessle et al.**
  - User-level: 49%
  - System-level: 0%

- **SAIL**
  - User-level: 12%
  - System-level: 0%

- **Current Work**
  - User-level: 89%
  - System-level: 11%

#user-level: ~3536
#system-level: ~200

10% Deprecated + 1% Crypto & Concurrency
Validation of Semantics

Comparing with hardware

- Instruction Level Testing (7000+ inputs states)

Comparing with Stoke

- Program Level Testing (GCC-c torture tests)

Comparing SMT formula

12+ Bugs reported
- Intel Manual
- Strata formulas

40+ Bugs reported
In Stoke
A Few Reported Bugs

Intel Manual Vol. 2: March 2018

VPSRAVD (VEX.128 version)
COUNT_0 ← SRC2[31 : 0]
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[100 : 96]
DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
(* Repeat shift operation for 2nd through 4th dwords *)
DEST[127:96] ← SignExtend(SRC1[127:96] >> COUNT_3);
DEST[MAXVL-1:128] ← 0;

Intel Manual Vol. 2: May 2019

VPSRAVD (VEX.128 version)
COUNT_0 ← SRC2[31 : 0]
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[127 : 96]
DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
(* Repeat shift operation for 2nd through 4th dwords *)
DEST[127:96] ← SignExtend(SRC1[127:96] >> COUNT_3);
DEST[MAXVL-1:128] ← 0;
A Few Reported Bugs

Stoke Implementation May 2018

VCVTSD (VEX.128 encoded version)
If 64-Bit Mode And OperandSize = 64
THEN
  DEST[63:0] \leftarrow \text{Convert Integer To Double Precision Floating Point}(SRC2[63:0]);
ELSE
  DEST[63:0] \leftarrow \text{Convert Integer To Double Precision Floating Point}(SRC2[31:0]);
Fi;

DEST[127:64] \leftarrow \text{(Unmodified)}

Intel Manual Vol. 2: May 2019

VCVTSD (VEX.128 encoded version)
If 64-Bit Mode And OperandSize = 64
THEN
  DEST[63:0] \leftarrow \text{Convert Integer To Double Precision Floating Point}(SRC2[63:0]);
ELSE
  DEST[63:0] \leftarrow \text{Convert Integer To Double Precision Floating Point}(SRC2[31:0]);
Fi;

DEST[127:64] \leftarrow SRC1[127:64]
A Few Reported Bugs

Stoke Implementation May 2018

PSLLD (with 64-bit operand)
IF (COUNT > 31)
THEN
    DEST[64:0] ← 0000000000000000H;
ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] << COUNT[31:0]);
FI;

Intel Manual Vol. 2: May 2019

PSLLD (with 64-bit operand)
IF (COUNT > 31)
THEN
    DEST[64:0] ← 0000000000000000H;
ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] << COUNT);
    DEST[63:32] ← ZeroExtend(DEST[63:32] << COUNT);
FI;
A Few Potential Applications

- Program verification
- Translation validation of compiler optimization
- Security vulnerability tracking
```c
uintptr_t safe_addptr (int *of, uint64_t a, uint64_t b) {
    uintptr_t r = a + b;
    if ( r < a ) // Condition not sufficient to prevent
        // overflow in case of 32-bit compilation
        *of = 1;
    return r;
}
```

Code snippet borrowed from HiStar kernel\textsuperscript{OSDI'06}, in which KLEE\textsuperscript{OSDI'08} found a security vulnerability
Use symbolic-execution to find an input \((a,b)\) such that

\[
\text{uintptr_t } r = a + b \\
r < a
\]

// Overflow detected
*of = 1

// Overflow not detected

return r
Use symbolic-execution to find an input \((a, b)\) such that

\[
\text{No overflow detected}
\]

\[
i.e. \ (a + b \mod 2^{32}) \geq a
\]

```c
uintptr_t r = a + b
r < a

// Overflow detected
*of = 1

// Overflow not detected

return r
```
Use symbolic-execution to find an input \((a,b)\) such that

1. **No overflow detected**
   
   \[ a + b \geq 2^{32} \]

2. **Overflow occurs**
   
   \[ a + b \geq 2^{32} \]
What's Next?

- To validate the translation in compiler backends and decompilers.
- To generate high-coverage test-inputs.
- Encourage and support external users of the semantics.
Summary

- Most complete user-level x86-64 semantics
- Thoroughly tested
- Applicable to different formal reasoning setup
- Publicly available

github.com/kframework/X86-64-semantics