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 <u>direct</u> 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

Formal ISA Semantics is Useful

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>
<tr><th colspan="2">RISC ISAs</th><th colspan="2">CISC ISAs</th></tr>
<tr><th>ISA</th><th>Formal Semantics</th><th>ISA</th><th>Formal Semantics</th></tr>
<tr><td>ARM</td><td>• seL4 '09
• SAIL '19
• Compcert '09
• CakeML '15</td><td rowspan="2">x86-32 (or IA32)</td><td rowspan="2">• ACL2 x86 '14
• Compcert '09
• TSL '13
• SAIL '17
• CakeML '14</td></tr>
<tr><td>RISC-V</td><td>• SAIL '19
• Compcert '09
• CakeML '18</td></tr>
<tr><td>MIPS</td><td>• SAIL '19</td><td rowspan="3">x86-64</td><td rowspan="3">• ACL2 x86-64 '14
• Strata '16
• CakeML '14
• Roessle et al. '19</td></tr>
<tr><td>POWER PC</td><td>• TSL '13
• Compcert '09</td></tr>
<tr><td>SPARC</td><td>• TSL '13</td></tr>
</table>

Example Formal ISA Semantics

<table>
<tr><th colspan="2">RISC ISAs</th><th colspan="2">CISC ISAs</th></tr>
<tr><td>ISA</td><td>Formal Semantics</td><td>ISA</td><td>Formal Semantics</td></tr>
<tr><td>ARM</td><td>• seL4 ’09
• SAIL ’19
• Compcert ’09
• CakeML ’15</td><td rowspan="2">x86-32 (or IA32)</td><td rowspan="2">• ACL2 x86 ’14
• Compcert ’09
• TSL ’13
• SAIL ’17
• CakeML ’14</td></tr>
<tr><td>RISC-V</td><td>• SAIL ’19
• Compcert ’09
• CakeML ’18</td></tr>
<tr><td>MIPS</td><td>• SAIL ’19</td><td rowspan="3">x86-64</td><td rowspan="3">• ACL2 x86-64 ’14
• Strata ’16
• CakeML ’14
• Roessle et al. ’19</td></tr>
<tr><td>POWER PC</td><td>• TSL ’13
• Compcert ’09</td></tr>
<tr><td>SPARC</td><td>• TSL ’13</td></tr>
</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

github.com/kframework/X86-64-semantics

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

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)
- ❑ Thoroughly tested against hardware using 7000+ input states and GCC-c torture tests
- ❑ Found bugs in Intel manual and related projects

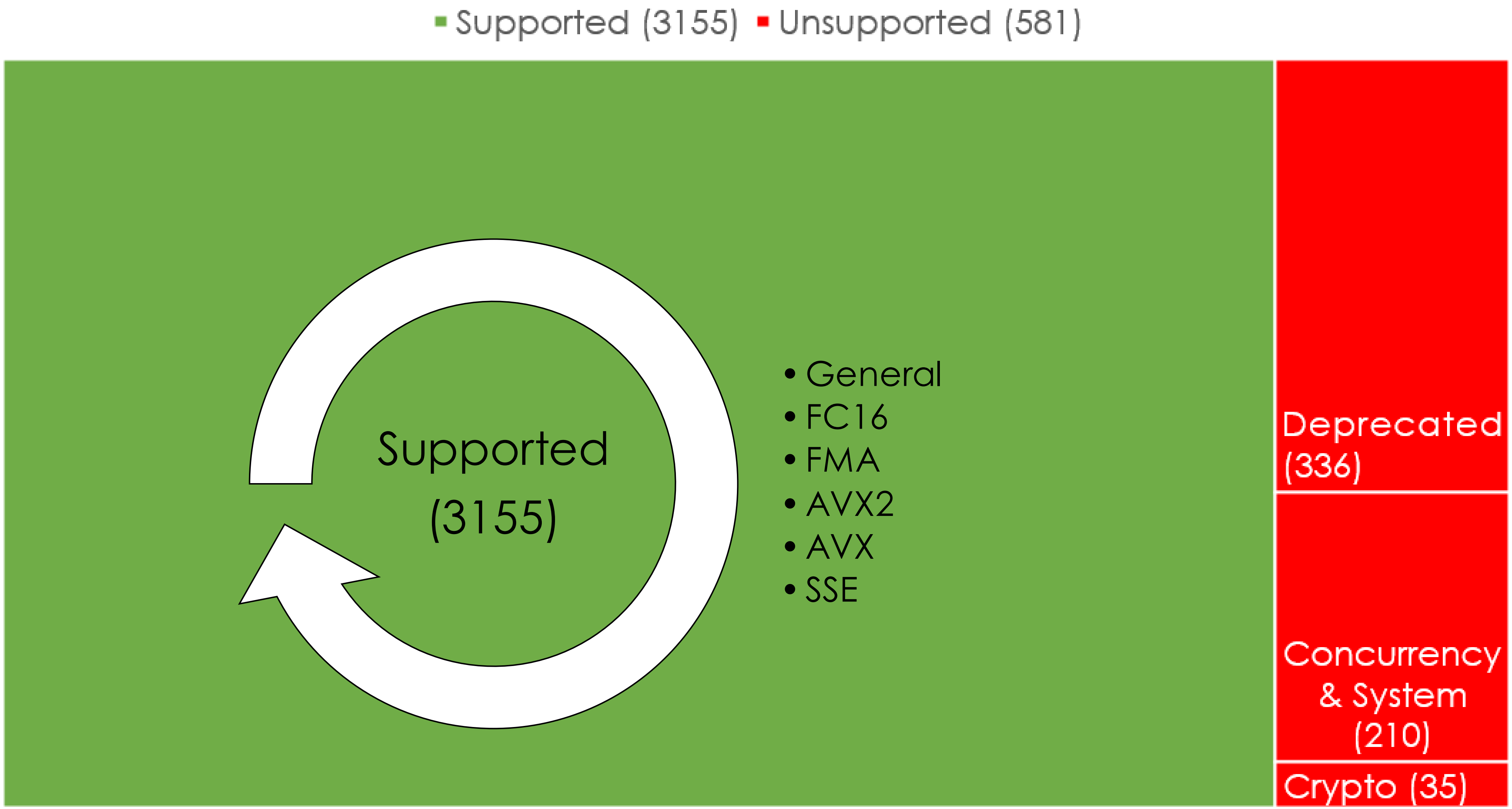
Our Contribution

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

github.com/kframework/X86-64-semantics

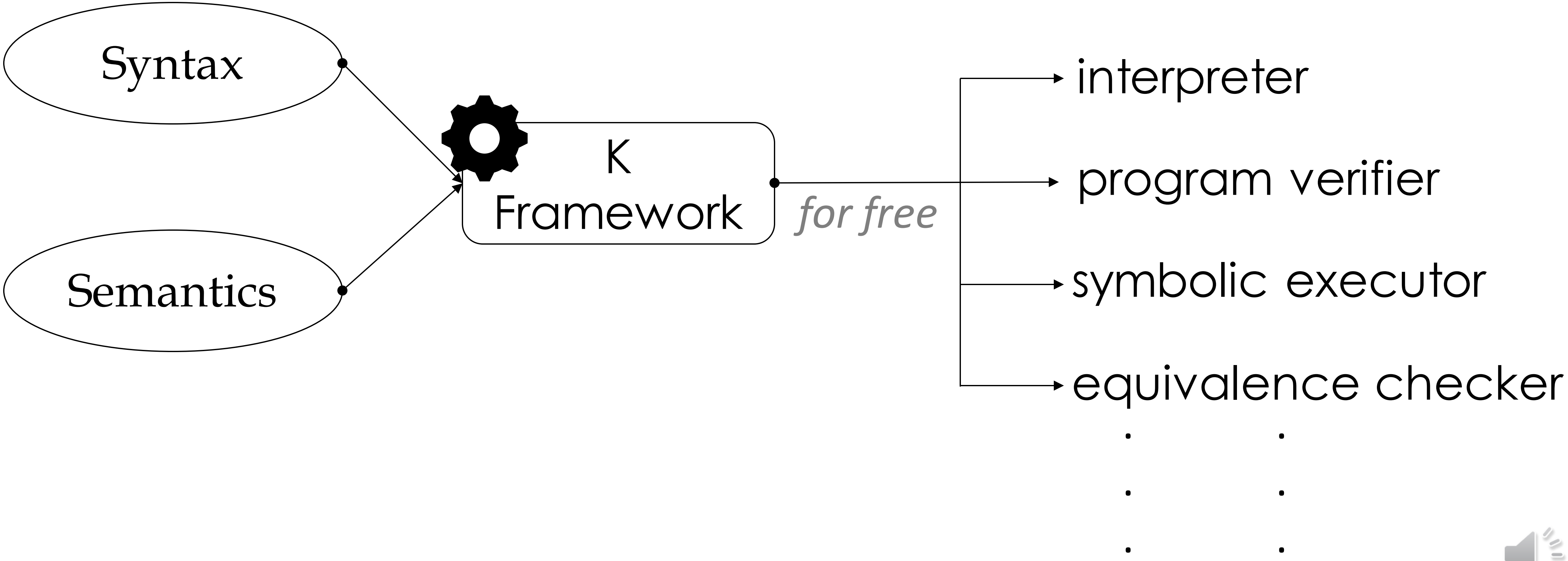
- ❑ Most complete user-level support (3155 instruction variants)
- ❑ Thoroughly tested against hardware using 7000+ input states and GCC-c torture tests
- ❑ Found bugs in Intel manual and related projects
- ❑ Demonstrated applicability to formal reasoning

Scope of Work (3155 / 3736)



K Framework [Rosu et al. 2010]

Language semantics engineering framework (kframework.org)



Approach Overview

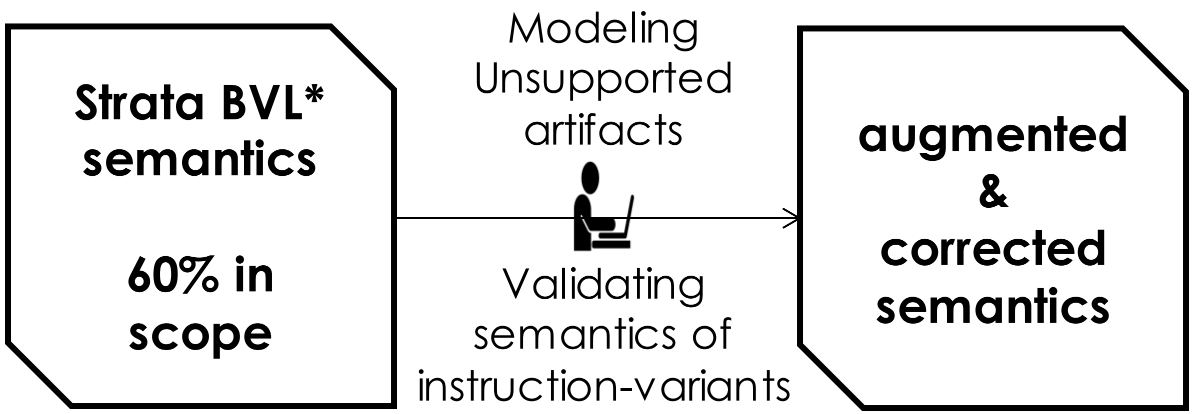
Approach Overview

Strata BVL* semantics

60% in scope

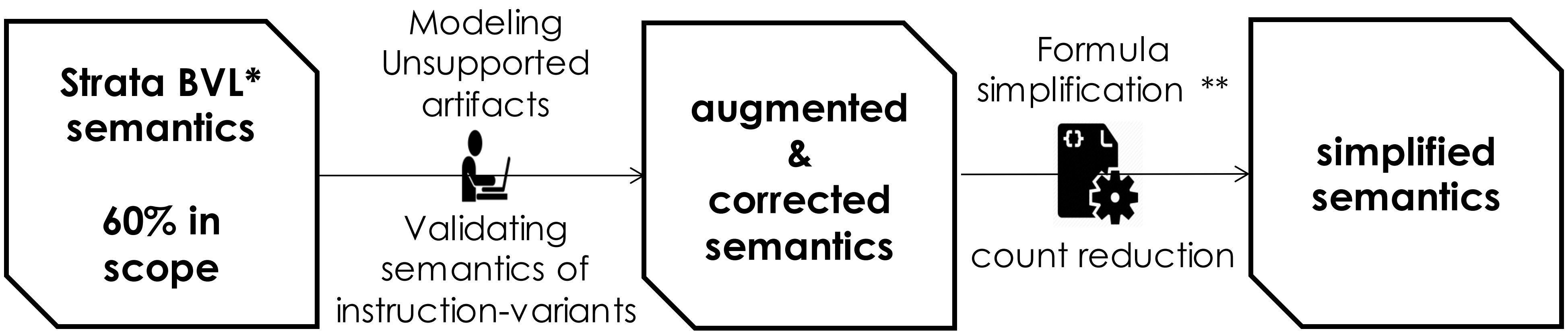
** BVL: Bit-vector logic*

Approach Overview



* *BVL: Bit-vector logic*

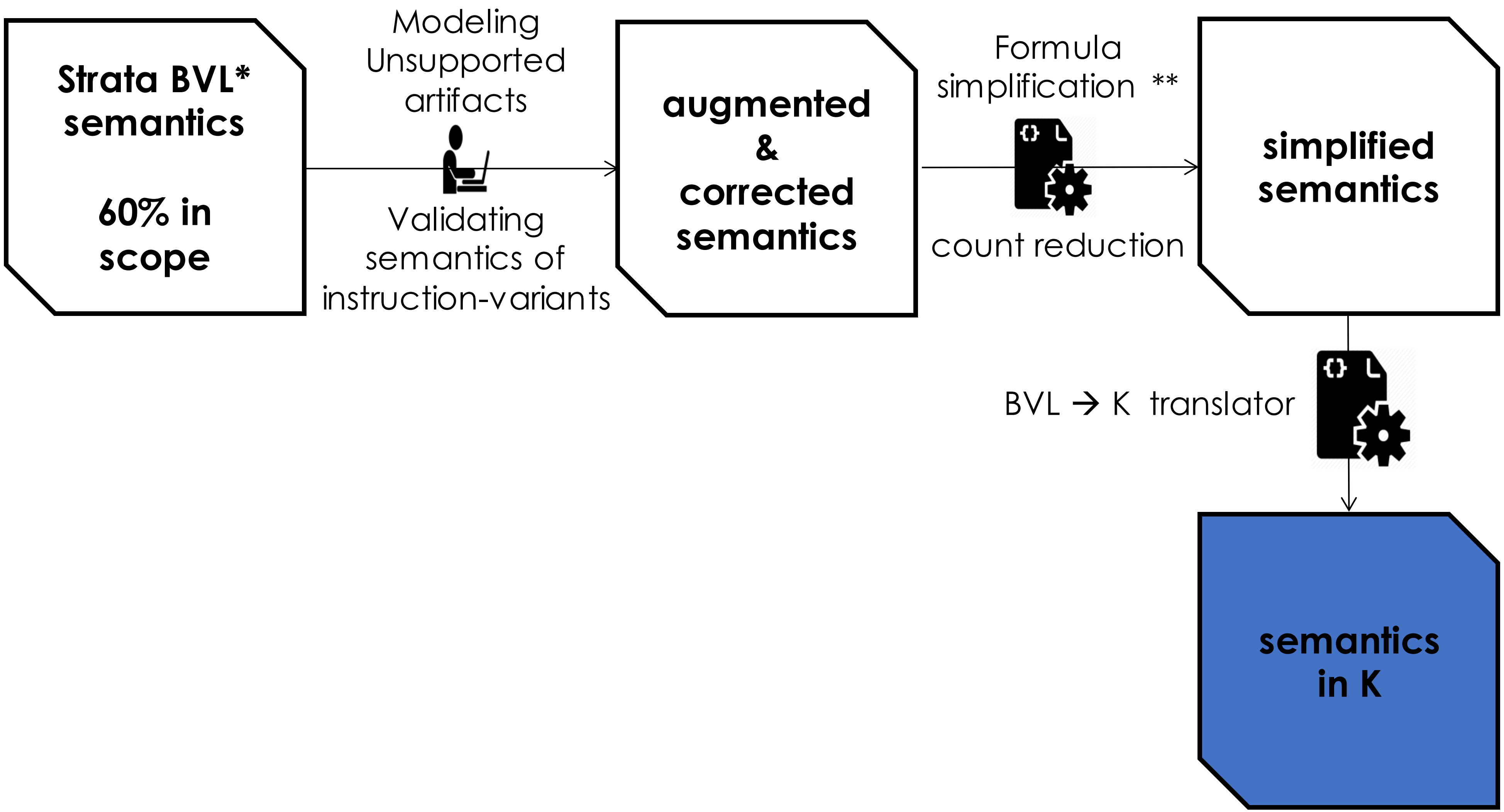
Approach Overview



* *BVL: Bit-vector logic*

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

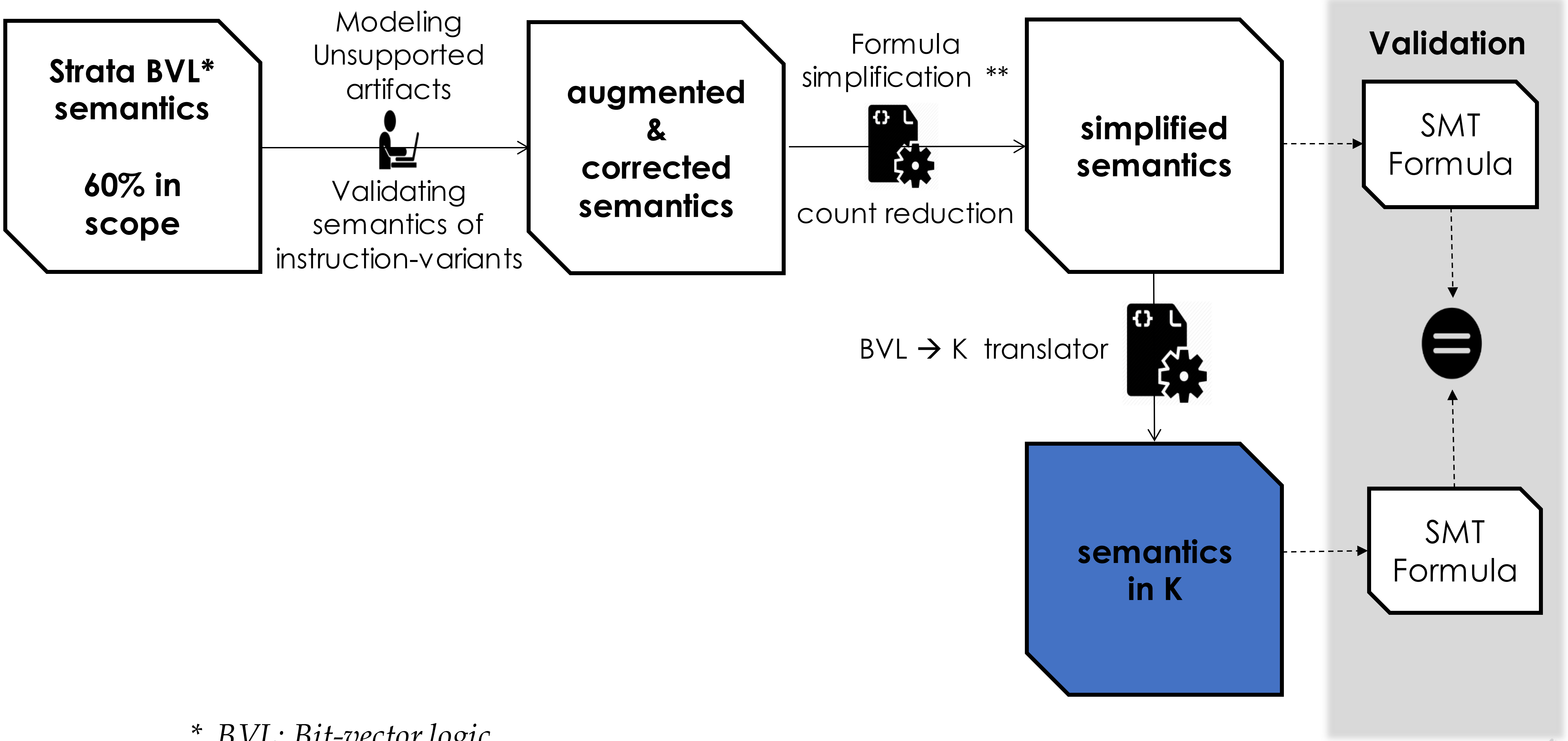
Approach Overview



* *BVL: Bit-vector logic*

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

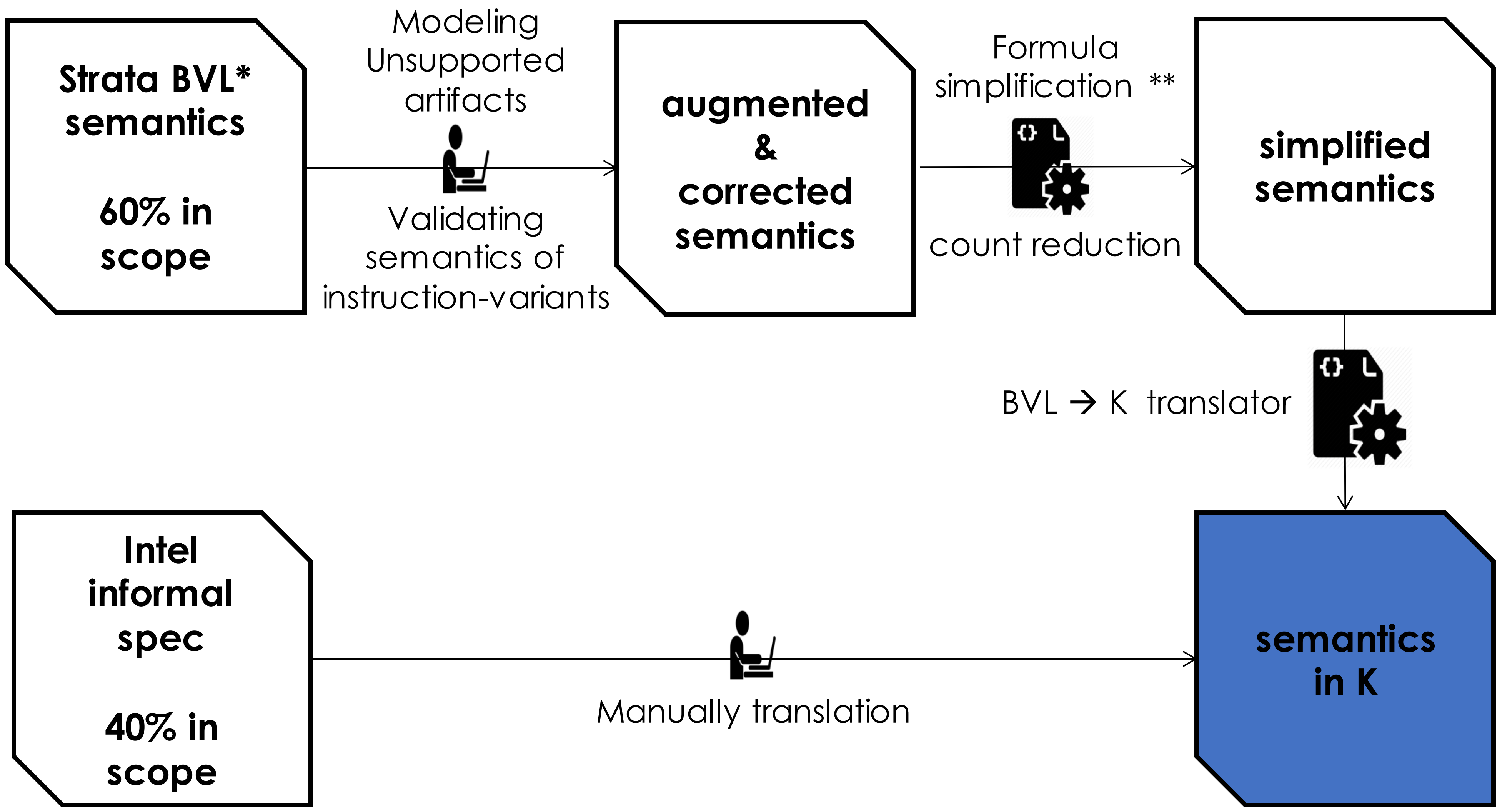
Approach Overview



* *BVL: Bit-vector logic*

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

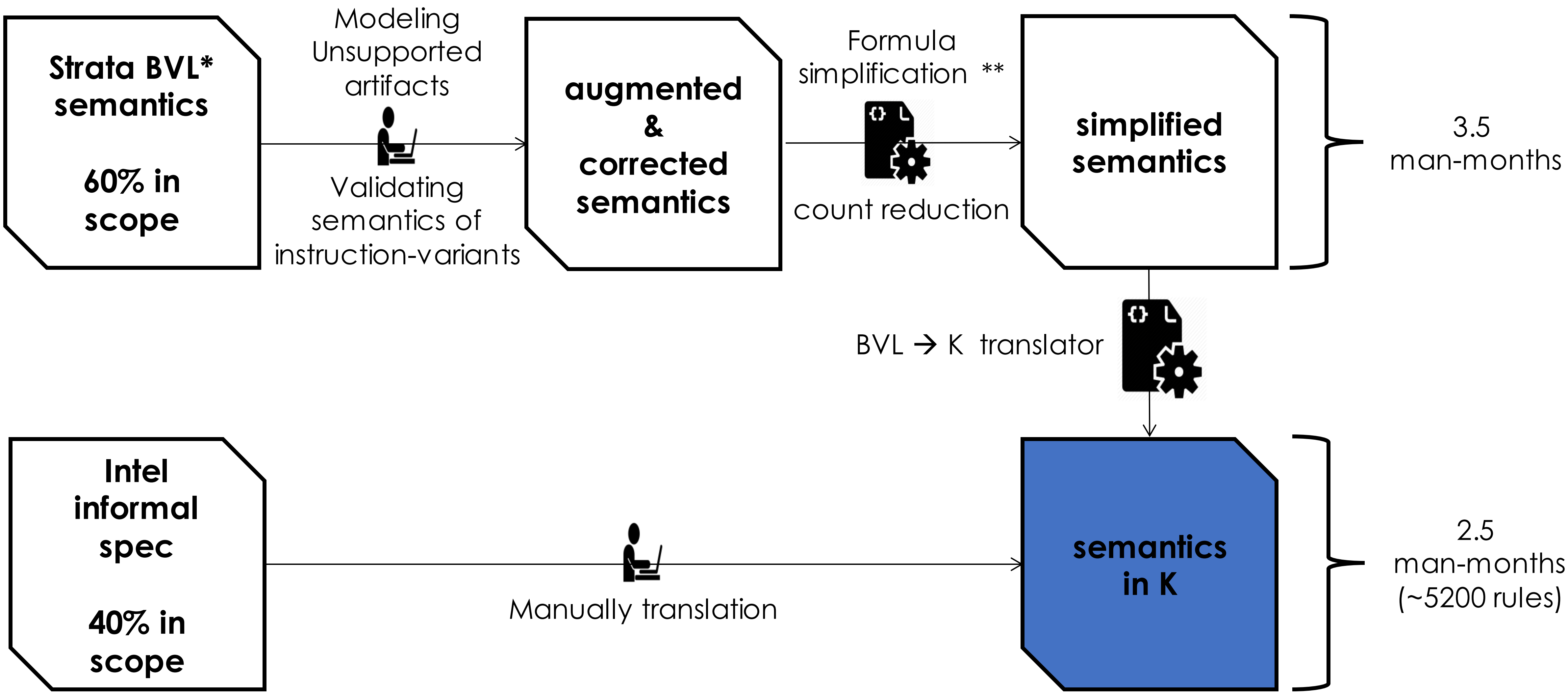
Approach Overview



* *BVL: Bit-vector logic*

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

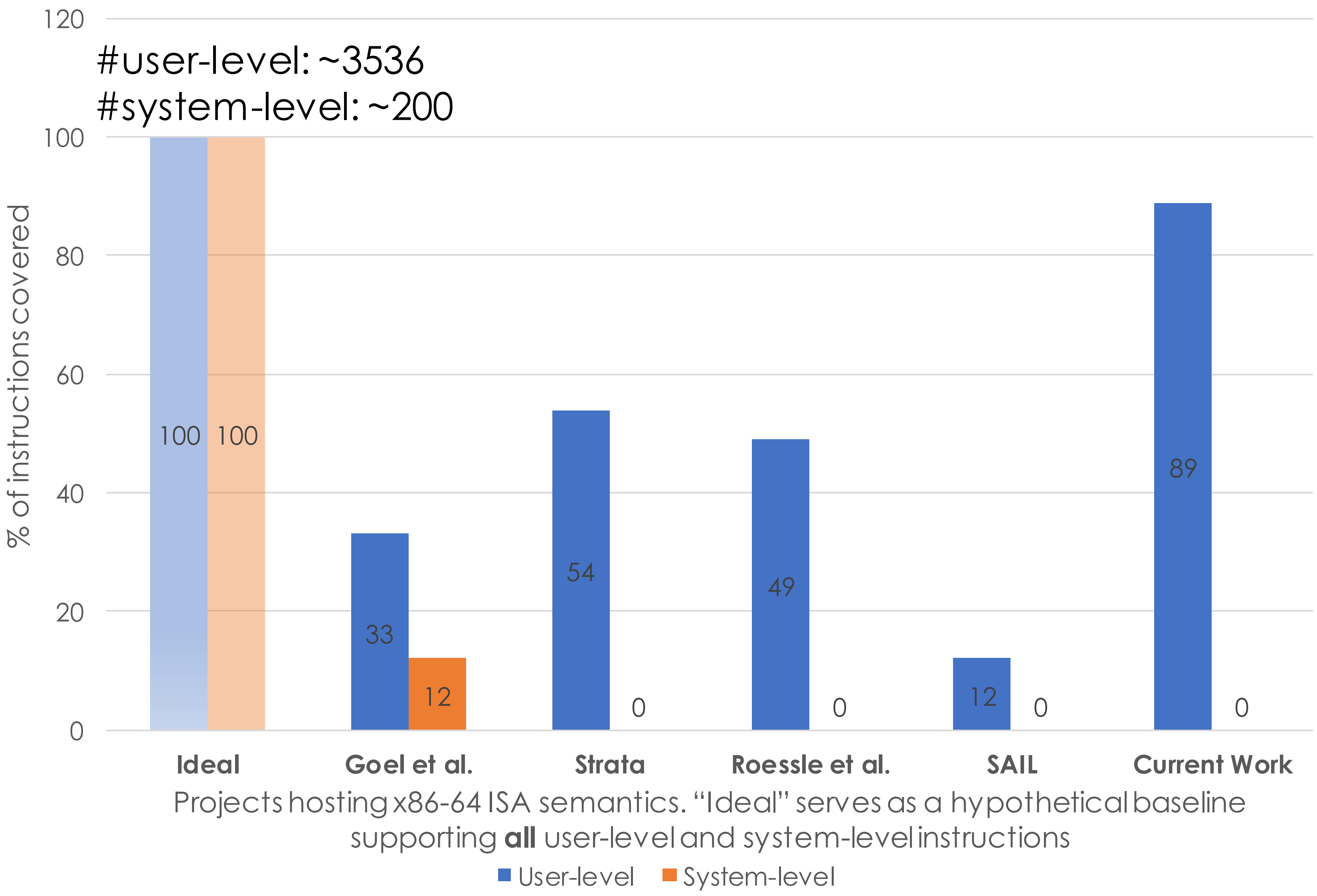
Approach Overview



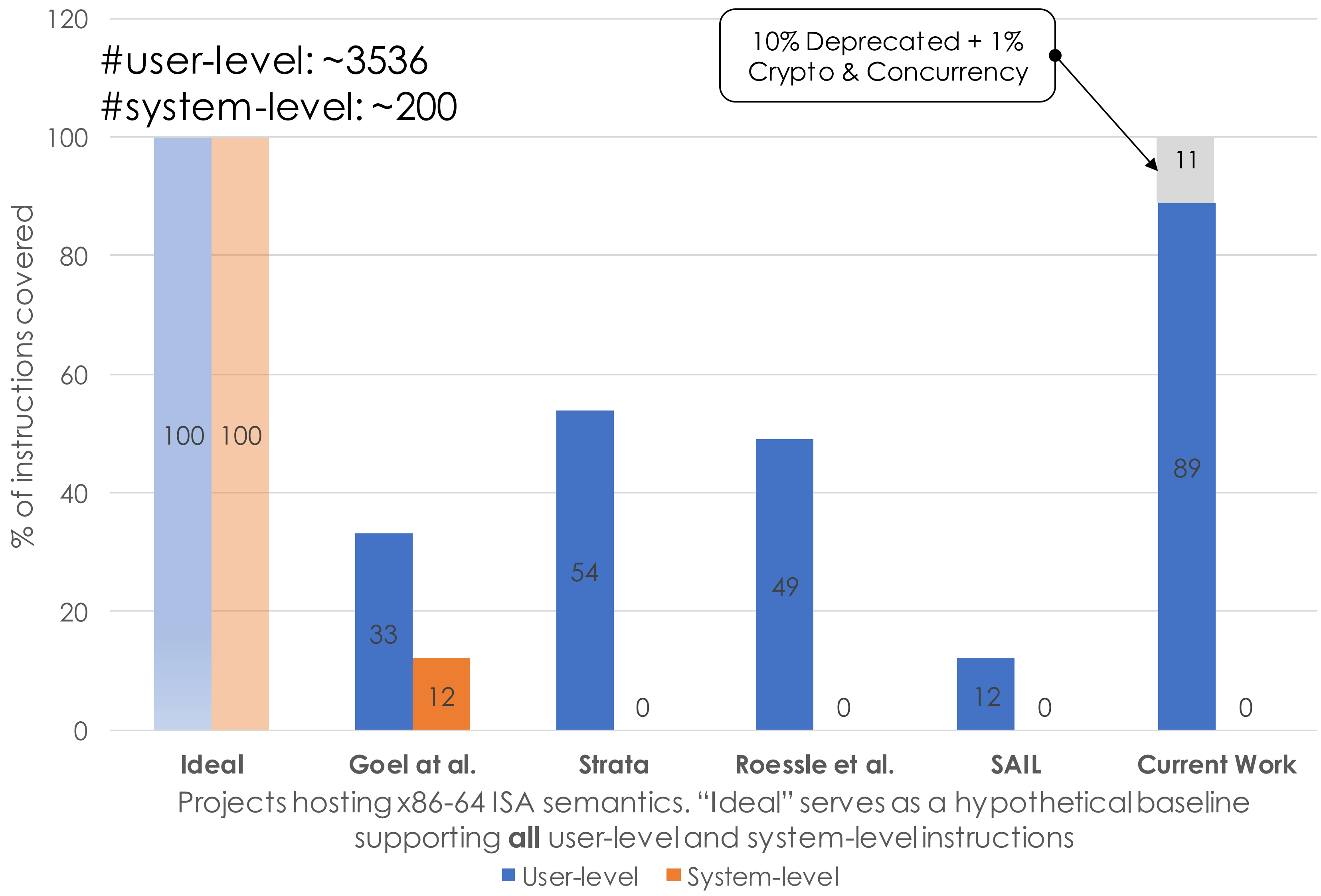
* *BVL: Bit-vector logic*

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

Support Comparison



Support Comparison



Validation of Semantics

Comparing with hardware

- Instruction Level Testing (7000+ inputs states)
- Program Level Testing (GCC-c torture tests)

Comparing with Stoke

- 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

```
VCVTSI2SD (VEX.128 encoded version)
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] ←(Unmodified)
```

Intel Manual Vol. 2: May 2019

```
VCVTSI2SD (VEX.128 encoded version)
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] ←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]);
        DEST[63:32] ← ZeroExtend(DEST[63:32] << COUNT [63:32]);
    FI;
```

Intel Manual Vol. 2: May 2019

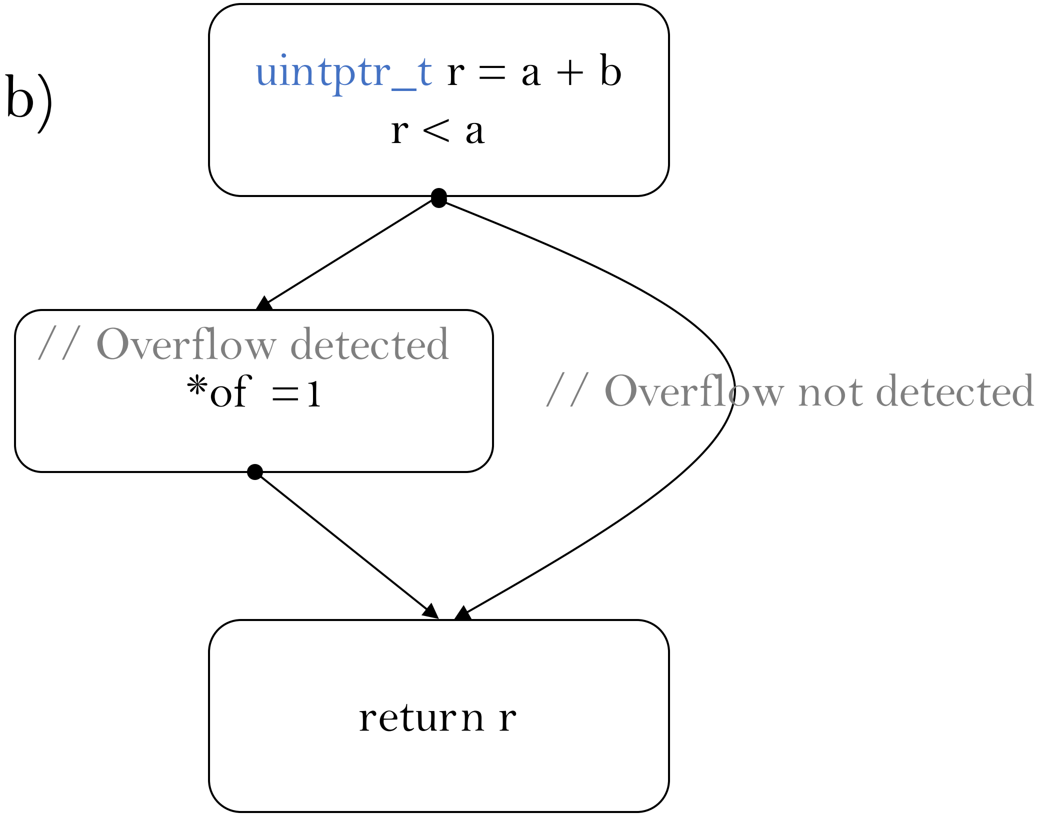
```
PSLLD (with 64-bit operand)
    IF (COUNT > 31)
    THEN
        DEST[64:0] ← 0000000000000000H;
    ELSE
        DEST[31:0] ← ZeroExtend(DEST[31:0] << COUNT);
        DEST[63:32] ← ZeroExtend(DEST[63:32] << COUNT);
    FI;
```

A Few Potential Applications

- Program verification
- Translation validation of compiler optimization
- Security vulnerability tracking

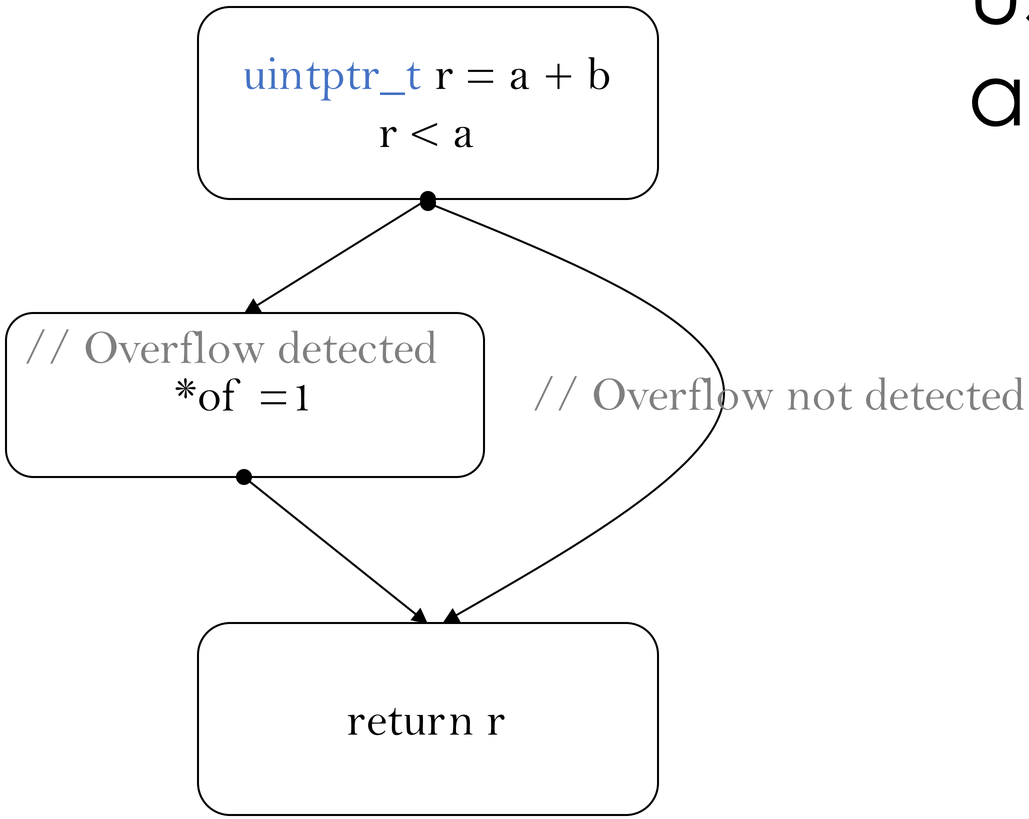
Security Vulnerability Tacking

```
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 ;
}
```



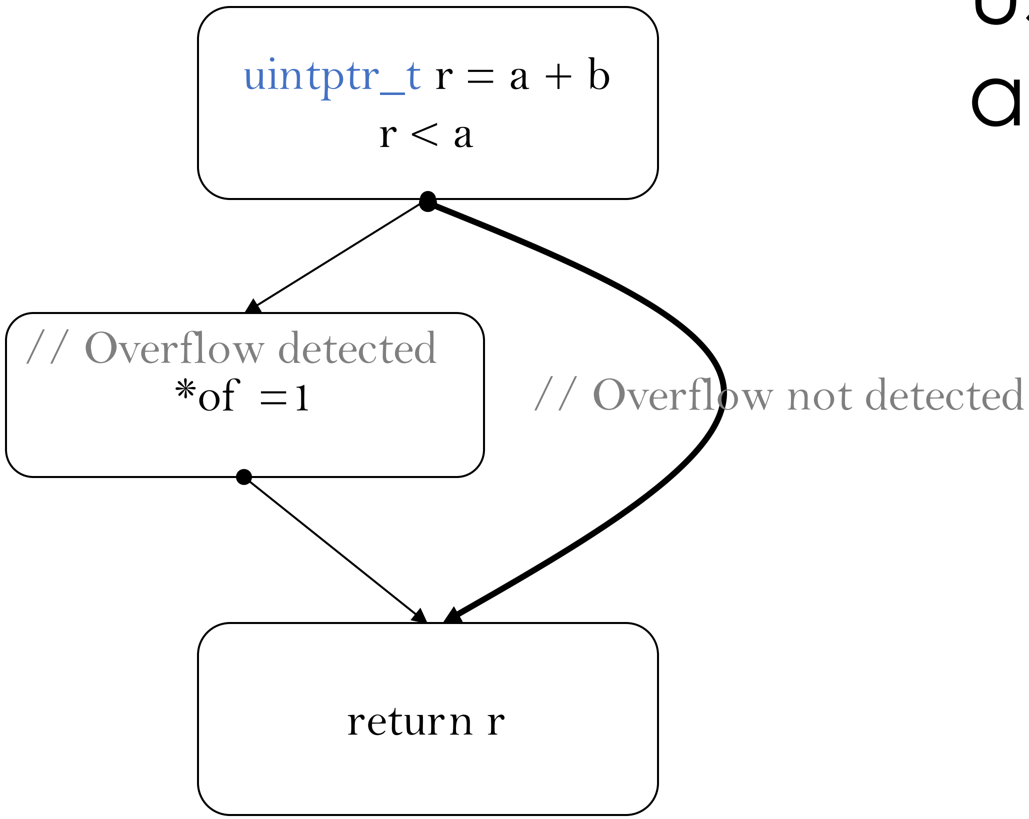
Code snippet borrowed from HiStar kernel[OSDI'06], in which KLEE[OSDI'08] found a security vulnerability

Security Vulnerability Tacking



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

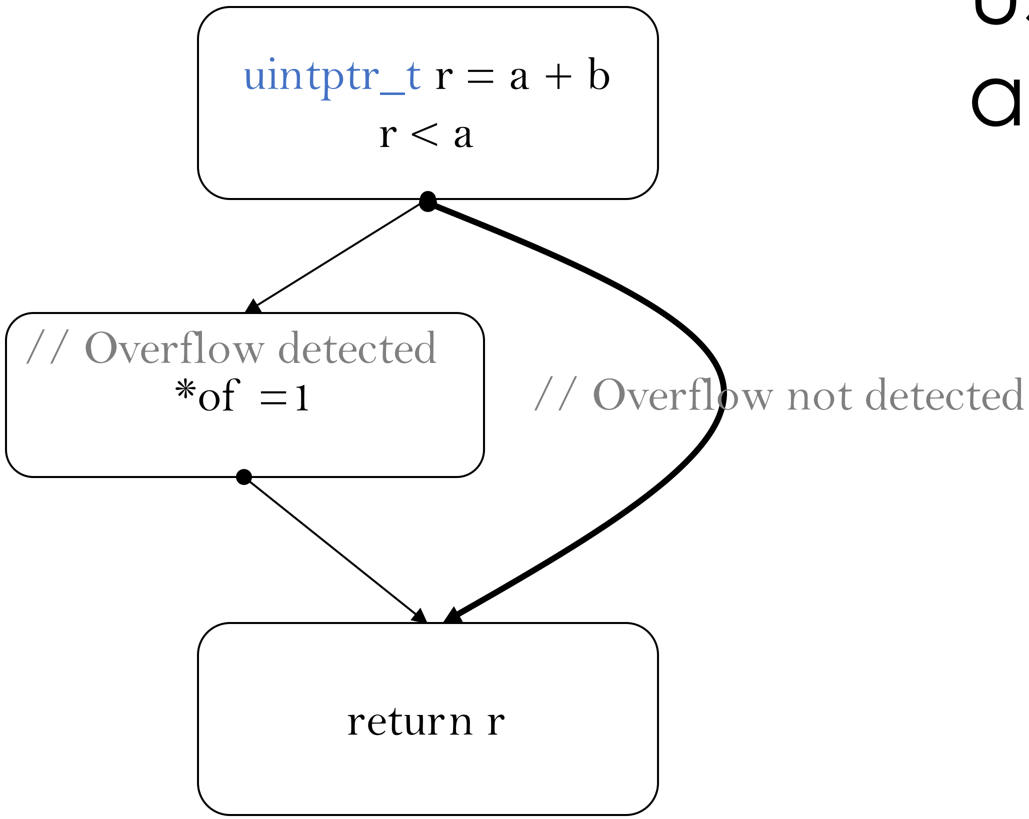
Security Vulnerability Tacking



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

1 No overflow detected
i.e. (a + b mod 2 ^ 32) ≥ a

Security Vulnerability Tacking



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

① No overflow detected
i.e. (a + b mod 2 ^ 32) ≥ a

And

② Overflow occurs
i.e. a + b ≥ 2^32

What's Next?

- To validate the translation in
  - compiler backends
  - 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