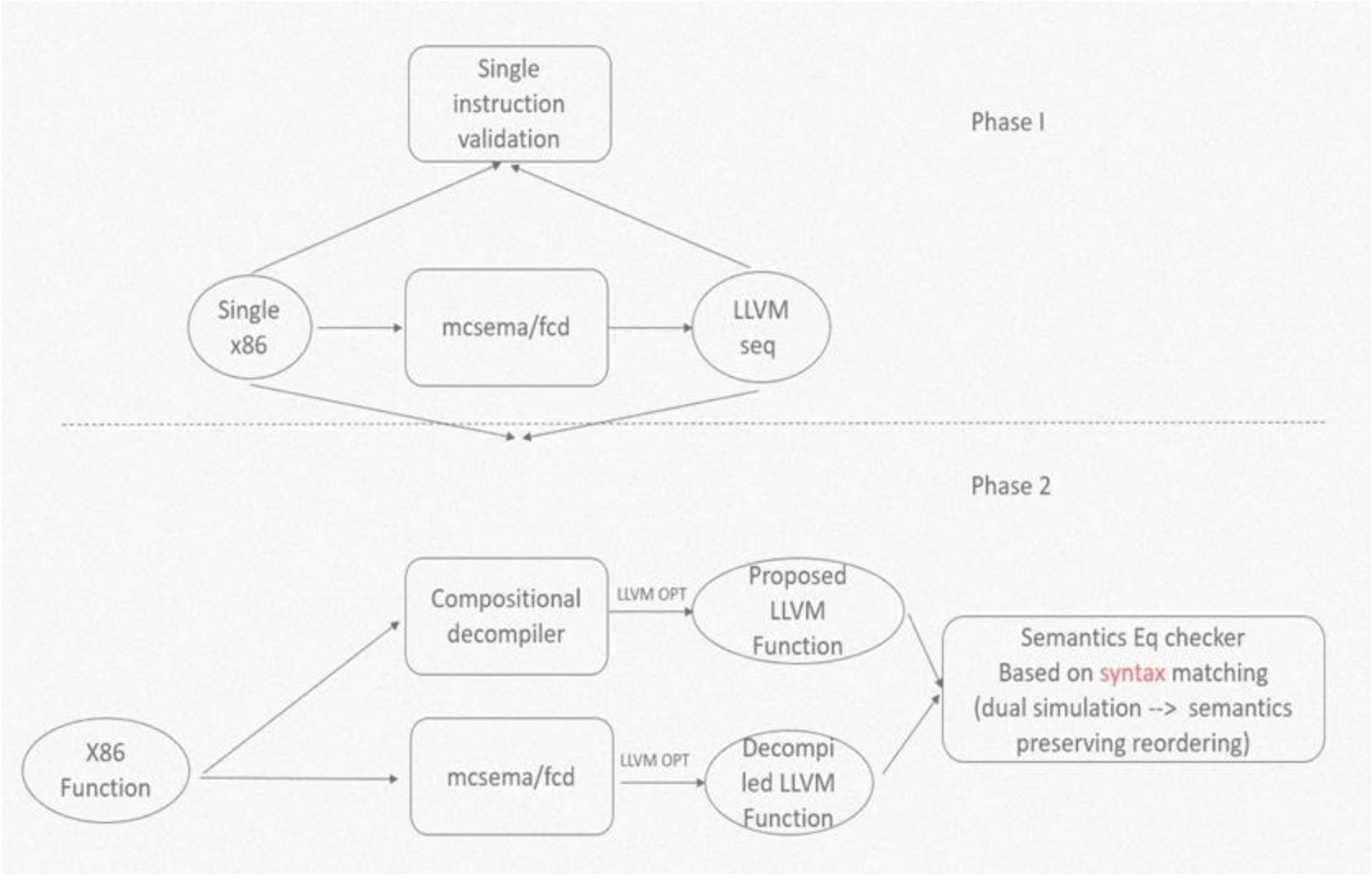
Scalable Validation of Binary Lifters



Ph.D. Final Exam Talk
by
Sandeep Dasgupta
advised by
Prof. Vikram Adve

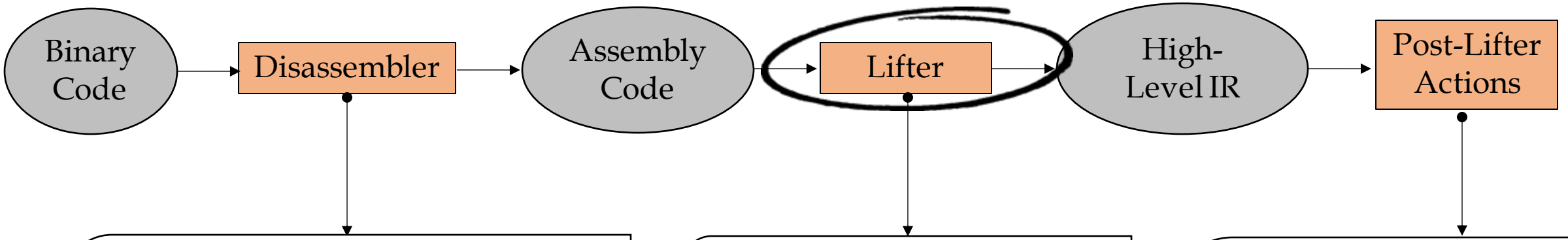
Binary Analysis is Important

The ability to directly reason about binary is important

scenarios where binary analysis is useful

- ❑ Missing source code (e.g. legacy or malware)
- ❑ Avoids trusting compilers
- ❑ Avoids separate abstractions for library code

A General Approach for Binary Analysis



Recover high-level primitives
- Instructions
- Control-flow graphs
- Functions boundaries

Enable re-targetability to multiple ISAs.

- Recovery of types, variables, and function prototypes
- Re-target or re-opt for different ISAs
- Binary Patching

Lifting is Challenging

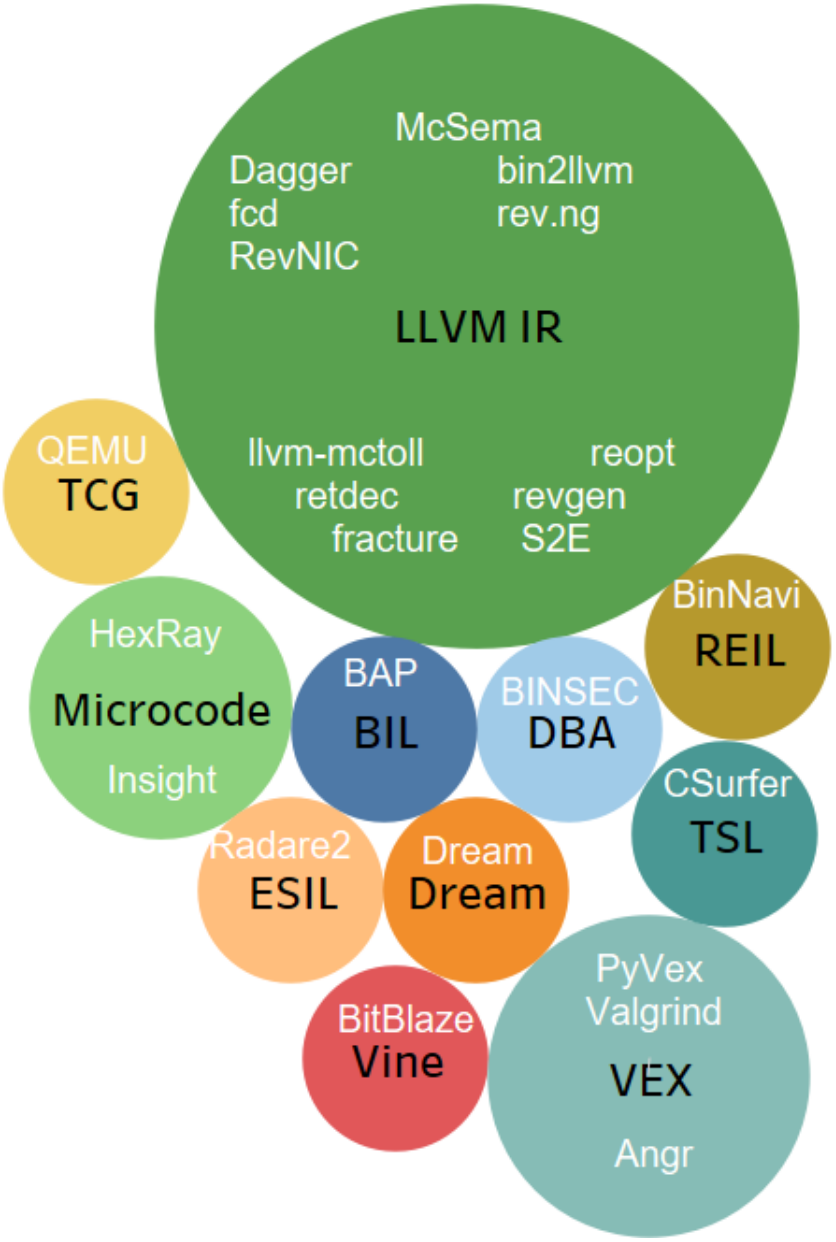
Manual encoding the effects of binary instructions is hard

- ❑ Vast number of instructions
- ❑ Standard manuals are often ambiguous, buggy, include divergence in the behaviours of variants

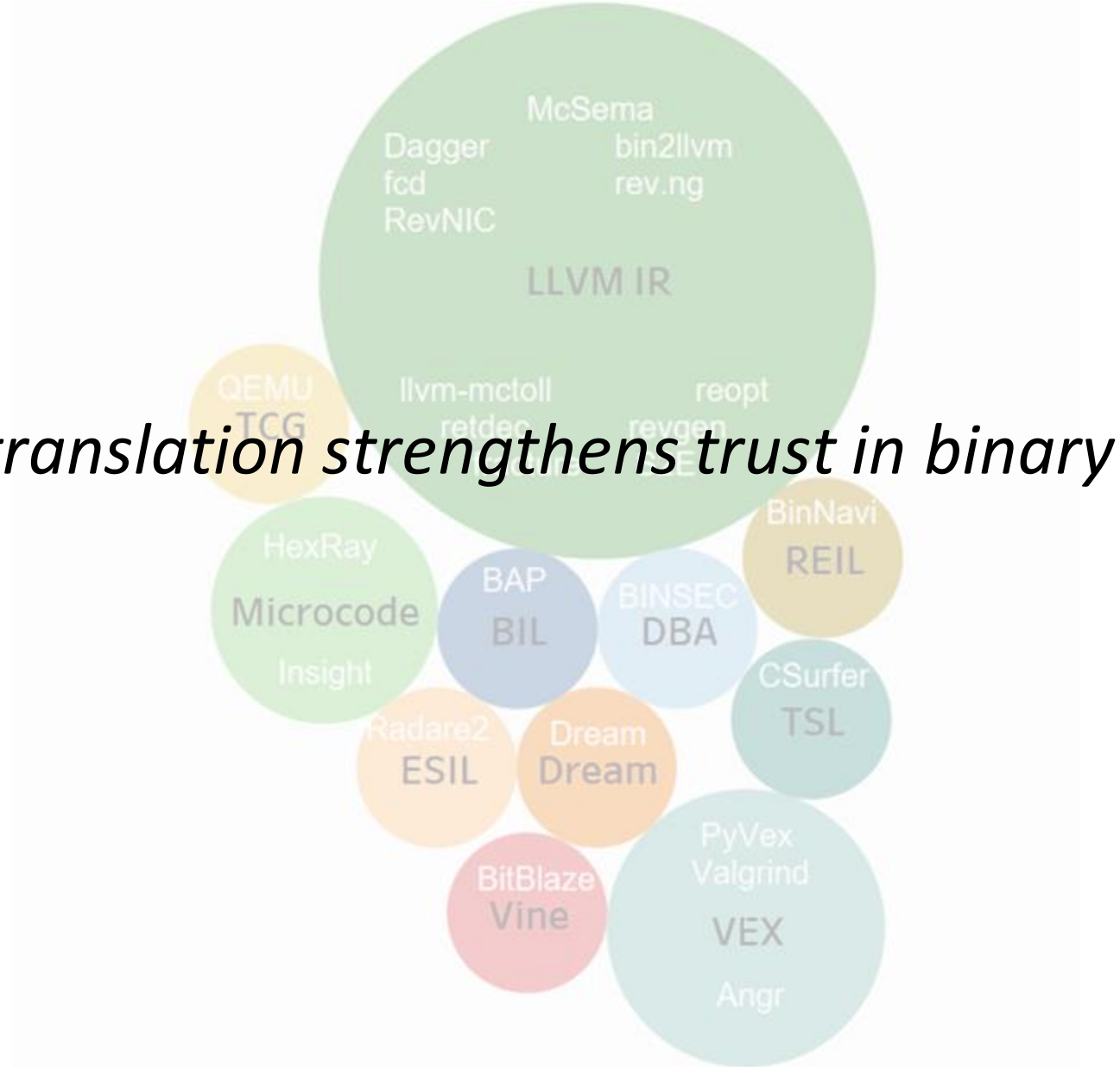
Semantics of Register Variant (`movsd %xmm1 , %xmm`)	Semantics of Memory Variant (`movsd (%rax) , %xmm0`)
S1. XMM0[63:0] ← XMM1[63:0]	S1. XMM0[63:0] ← MEM_ADDR[63:0]
S2. XMM0[127:64] (Unmodified)	S2. XMM0[127:64] ← 0

- ❑ Lack of formal operational ISA specifications (in general)

Lifting is Pivotal in Binary Analysis



Validation of Lifting is Critical



Faithful binary translation strengthens trust in binary analysis results

Thesis Statement

To develop formal and informal techniques to achieve high confidence in the correctness of binary lifting, from a complex machine ISA (e.g., x86-64) to a rich IR (e.g., LLVM IR), by leveraging the semantics of languages involved (e.g., x86-64 and LLVM IR)

Summary of Prior Work

Require random testing	Restricted to instruction- or basic-block-level validation	Require instrumentation
• Martignoni et al. ISSTA'10 • Chen et al. CLSS'15	▪ Martignoni et al. ISSTA'10, ASPLOS'12 ▪ Chen et al. CLSS'15 ▪ Meandiff - Kim et al. ASE'17	▪ Reopt-vcg, John et al. SpISA'19

Scope of the work

Validating the translation from **x86-64 programs** to **LLVM IR** using **McSema** - a mature, active maintained, and open-source lifter

Our Approach: Intuition

Observation

Most binary lifters are designed to perform simple instruction-by-instruction lifting followed by standard IR optimizations to achieve simpler IR code

Intuition

Formal translation validation of single machine instructions can be used as a building block for scalable full-program validation

Our Two-Phase Approach

Phase I **S**ingle-**I**nstruction **T**ranslation-**V**alidation (SITV)

- ❖ Translation-validation of lifted instructions in isolation
- ❖ Leverages our prior work on formalizing x86-64 semantics

Phase II **P**rogram-**l**evel **V**alidation (PLV)

- ❖ A scalable approach for full-program validation build on SITV
- ❖ Cheaper than symbolic-execution based equivalence checking

Contributions

- **Defining the formal Semantics of x86-64** (PLDI'19)
  - **Most Complete** user-level instruction semantics
  - **Faithful** up to through testing
  - **Revealed Bugs** in Intel Manual and related semantics
  - **Useful** for various formal analyses

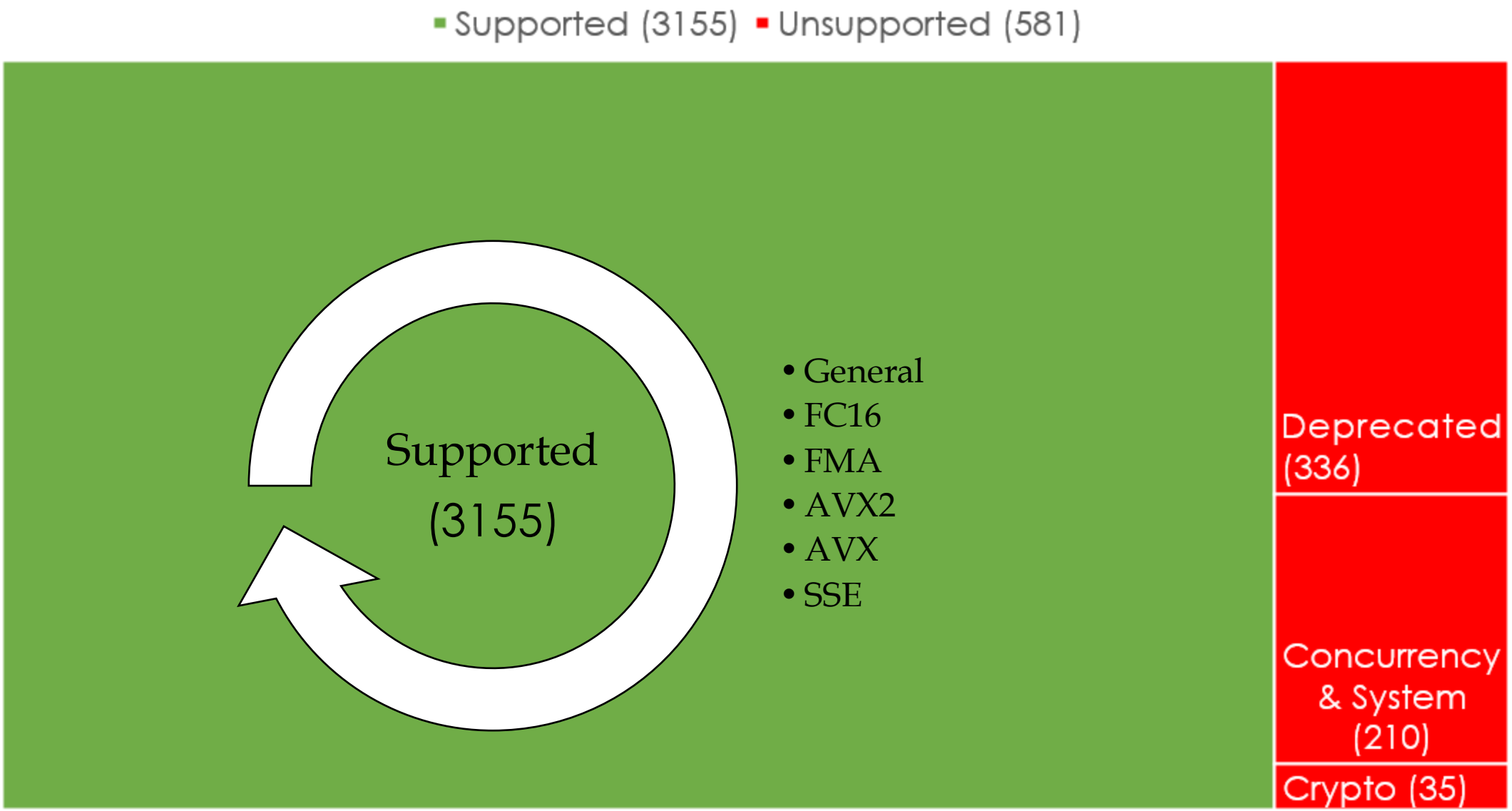
- **Developing scalable technique for validating lifters** (PLDI'20)
  - **First SIV framework** for an extensive x86-64 ISA
  - **Revealed Bugs** in a mature lifter like McSema
  - **Novel Technique** for SITV-assisted full-program validation

Defining Formal Semantics of x86-64 ISA

Challenges: *from* ISA Spec *to* Semantics

- ❑ 3000+ pages of informal description
- ❑ 996 unique mnemonics with 3736 variants
- ❑ Inconsistent behavior of variants

Scope of Work (3155 / 3736)



Approach Overview

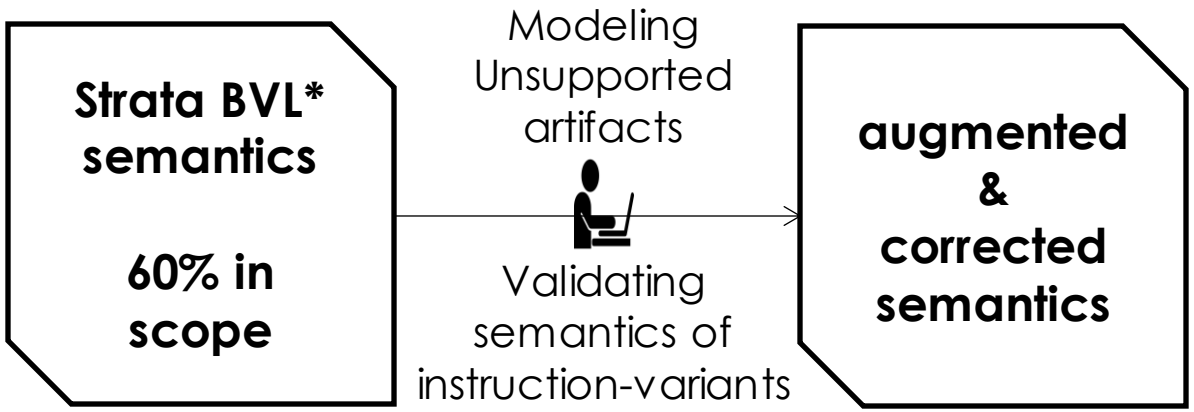
Approach Overview

Strata BVL* semantics

60% in scope

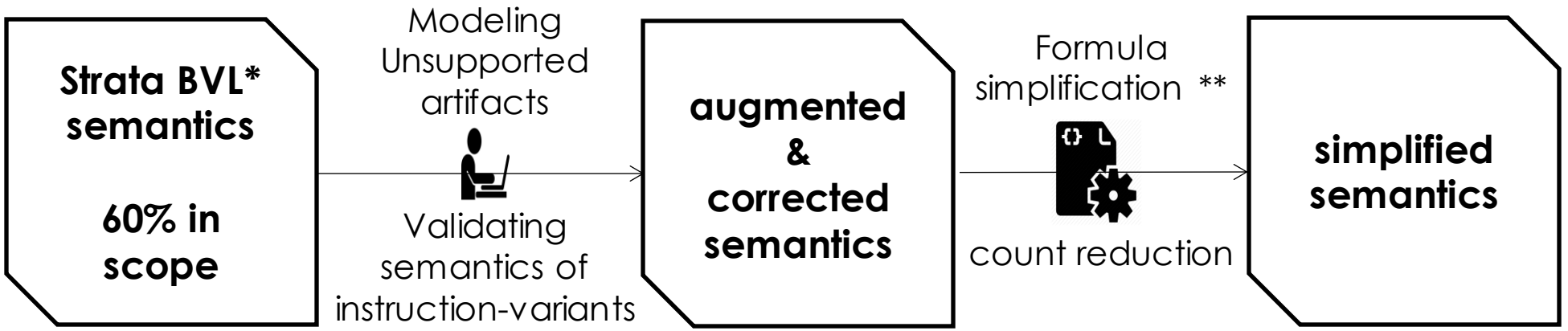
* *BVL: Bit-vector logic*

Approach Overview



* *BVL: Bit-vector logic*

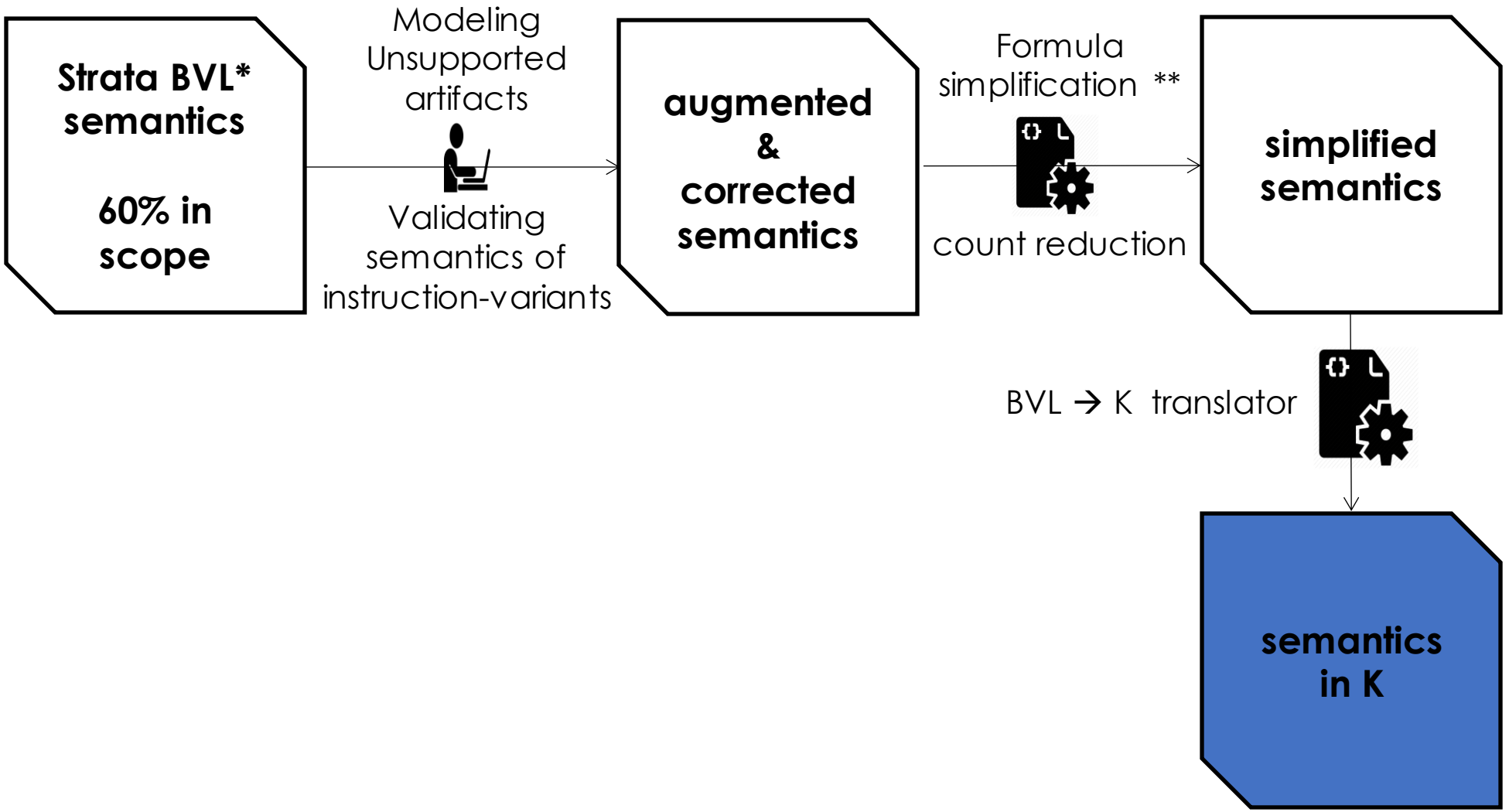
Approach Overview



* *BVL: Bit-vector logic*

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

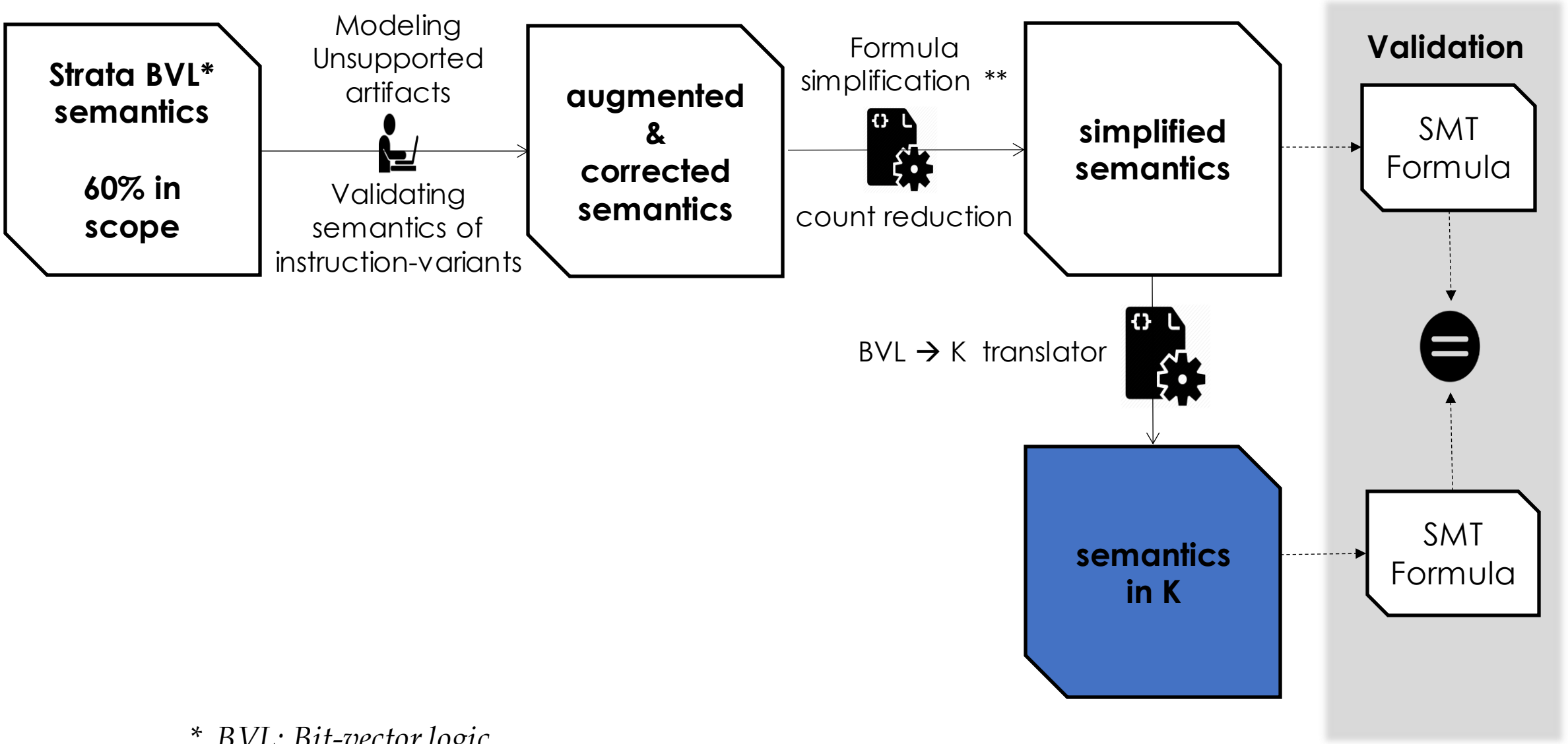
Approach Overview



* *BVL: Bit-vector logic*

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

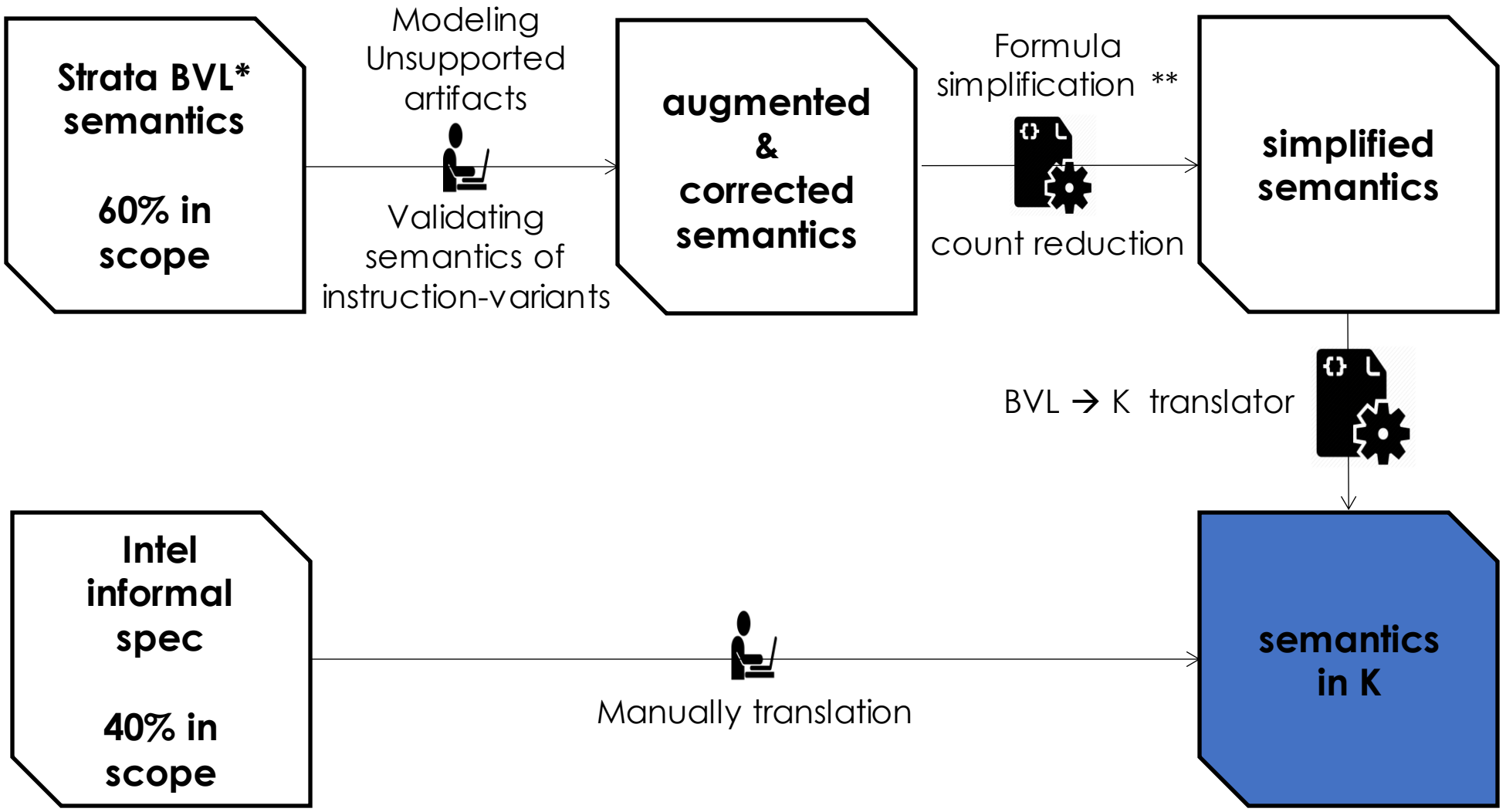
Approach Overview



* *BVL: Bit-vector logic*

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

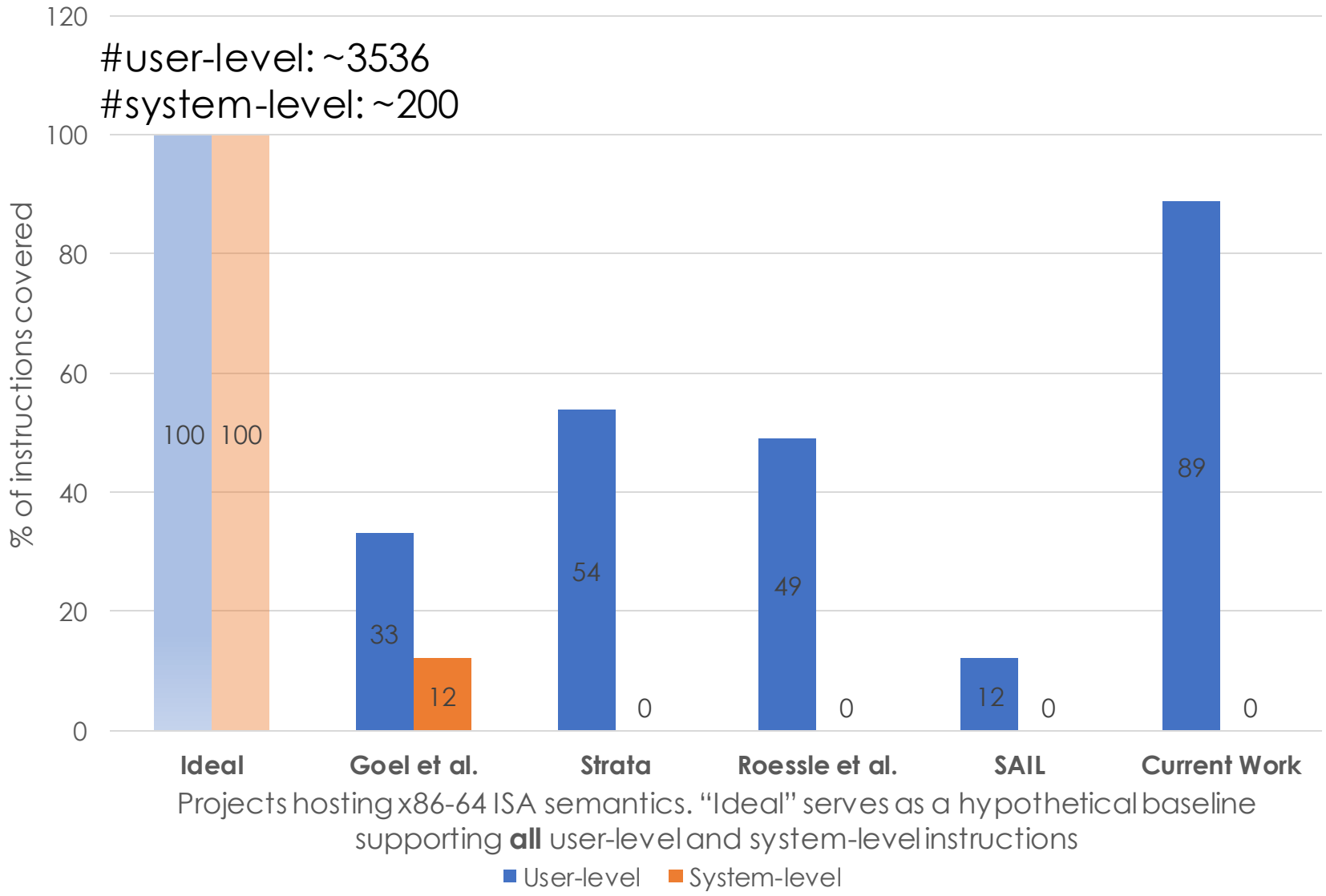
Approach Overview



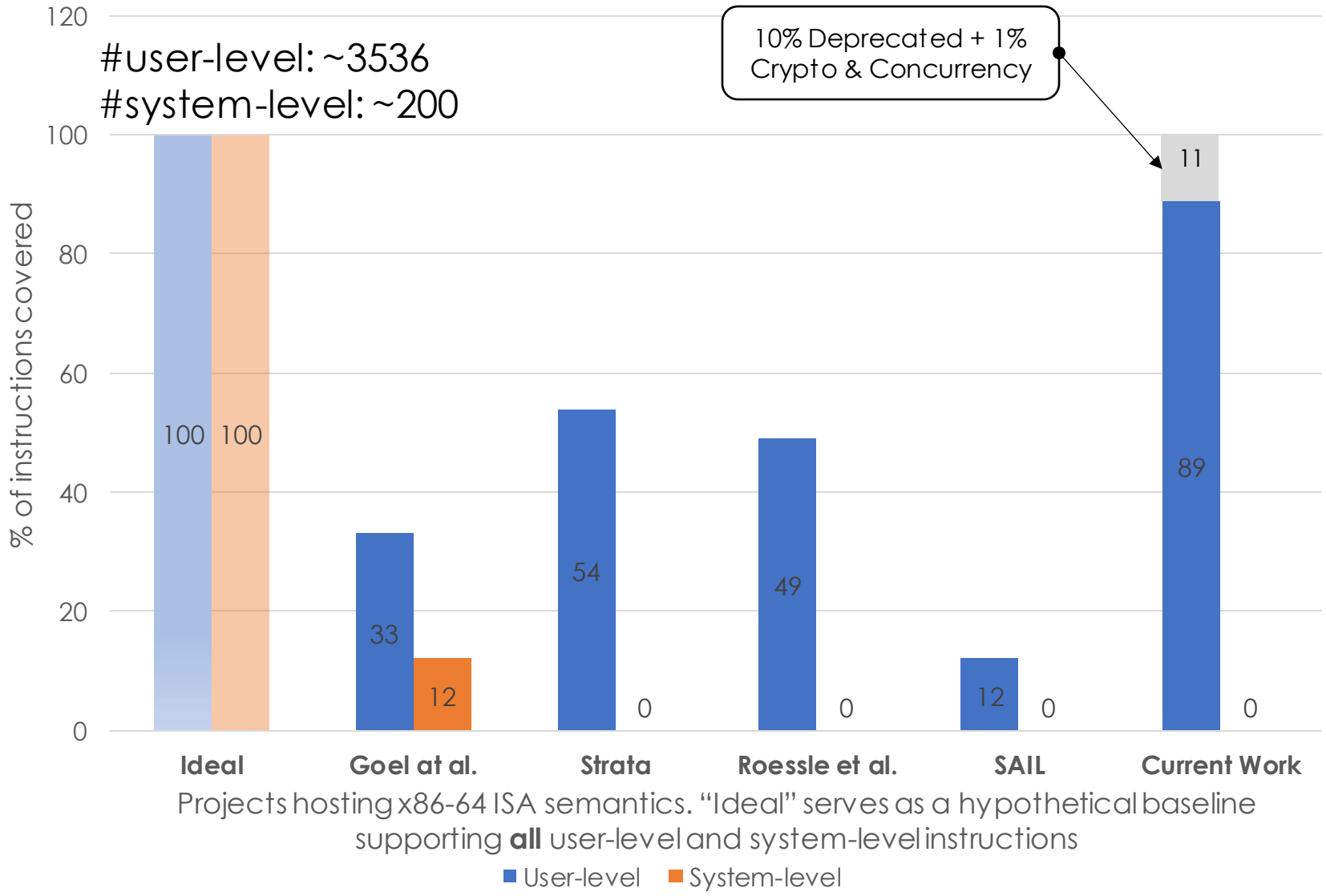
* *BVL: Bit-vector logic*

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

Support Comparison



Support Comparison



Validation of Semantics

Comparing with hardware

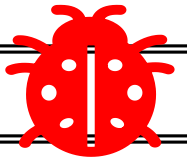
- Instruction Level Testing (7000+ inputs states)
- Program Level Testing (GCC-c torture tests)

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]

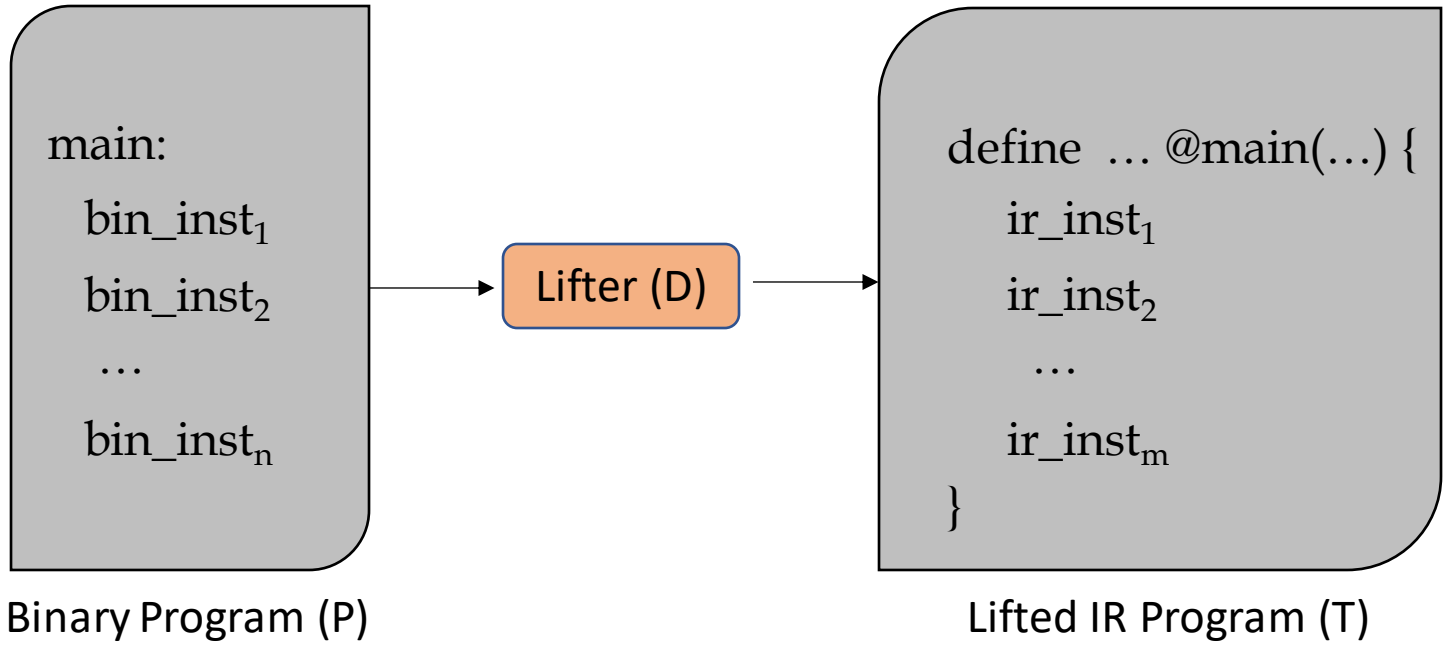
A Few Potential Applications

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

Lifter Validation: Our Approach

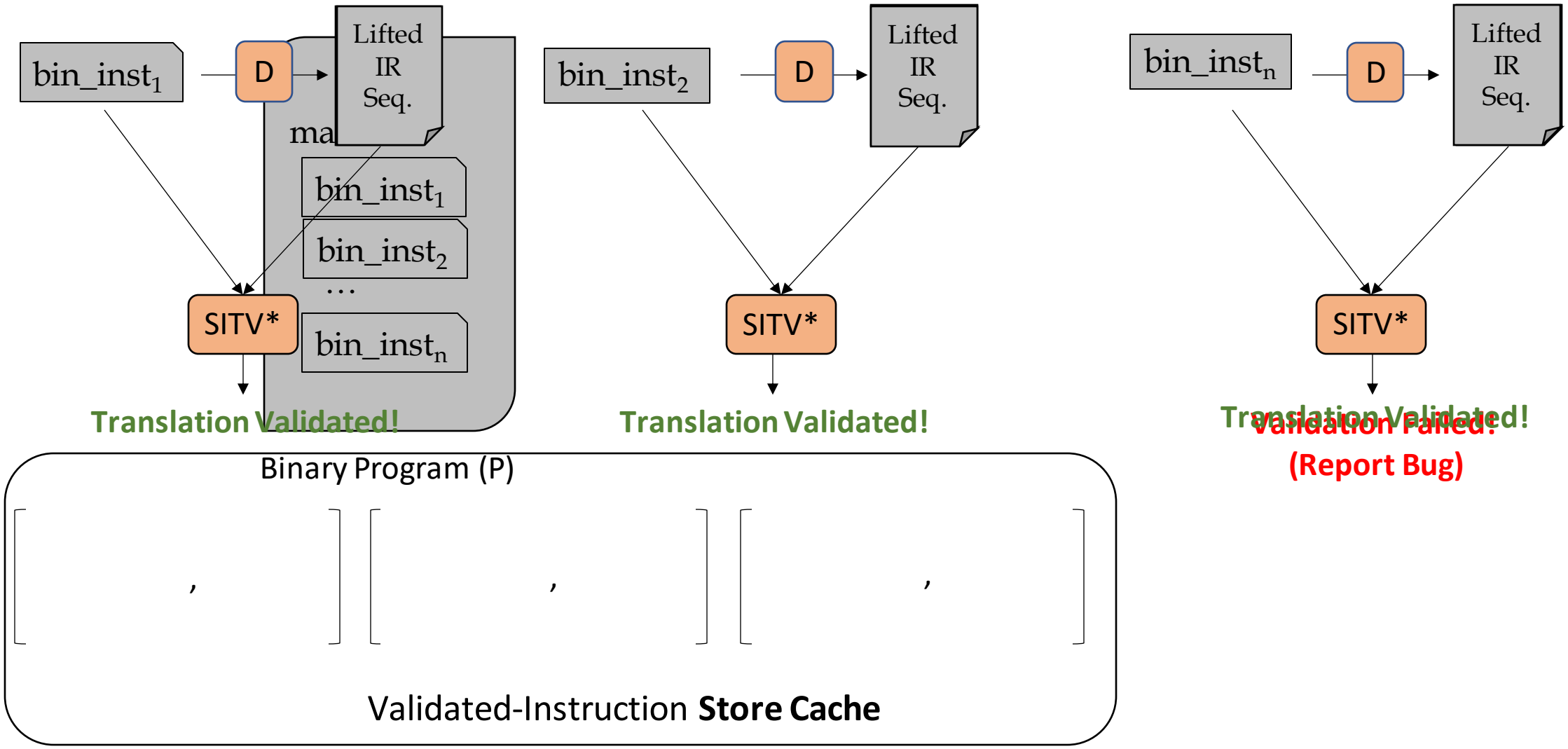
- **Phase I** **S**ingle-**I**nstruction **T**ranslation-**V**alidation (SITV)
- **Phase II** **P**rogram-**l**evel **V**alidation (PLV)

Overall Goal

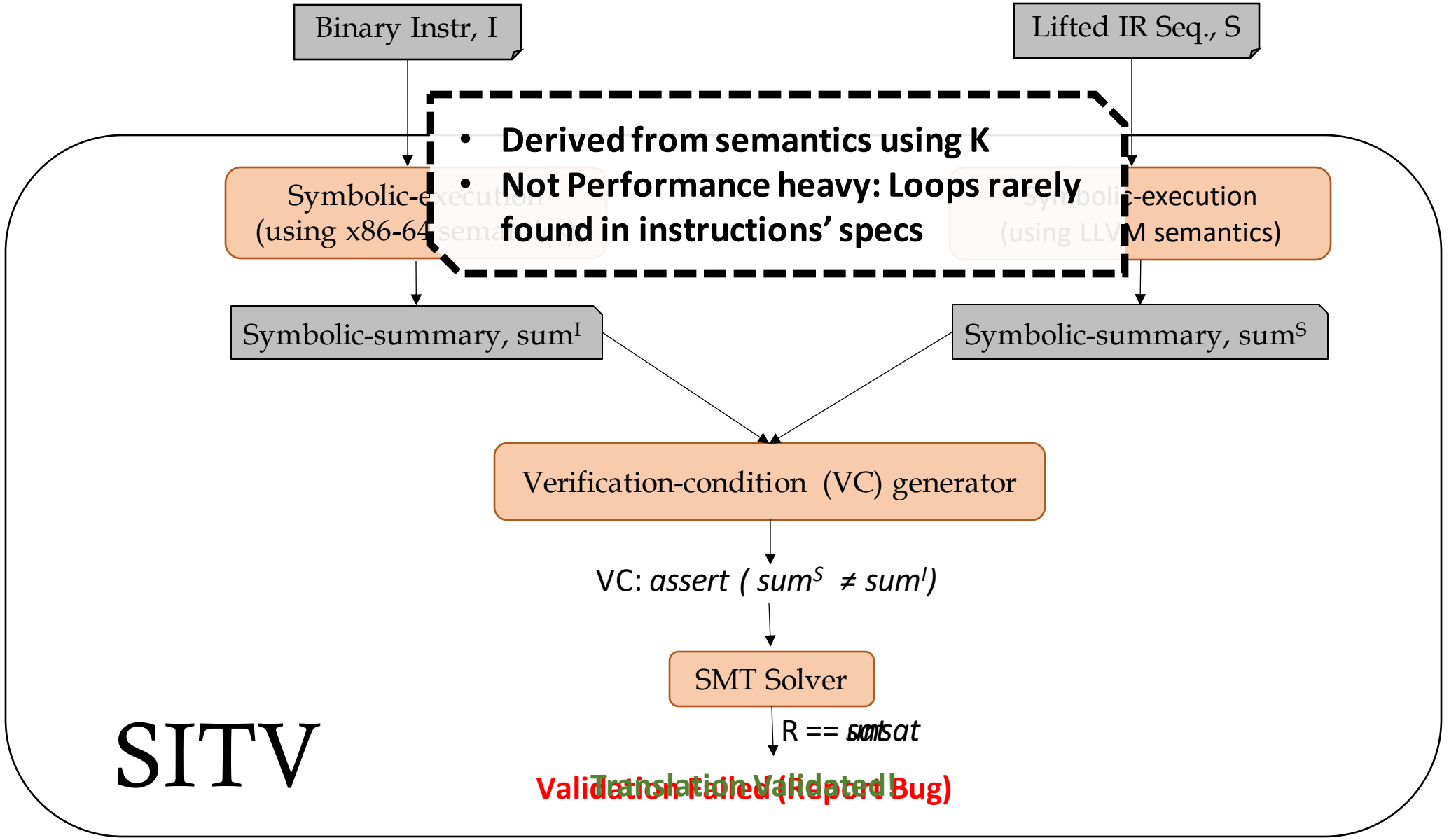


Our goal is to validate the translation from P to T

Single-Instruction Translation Validation



*SITV: Single Instruction Translation Validation Framework

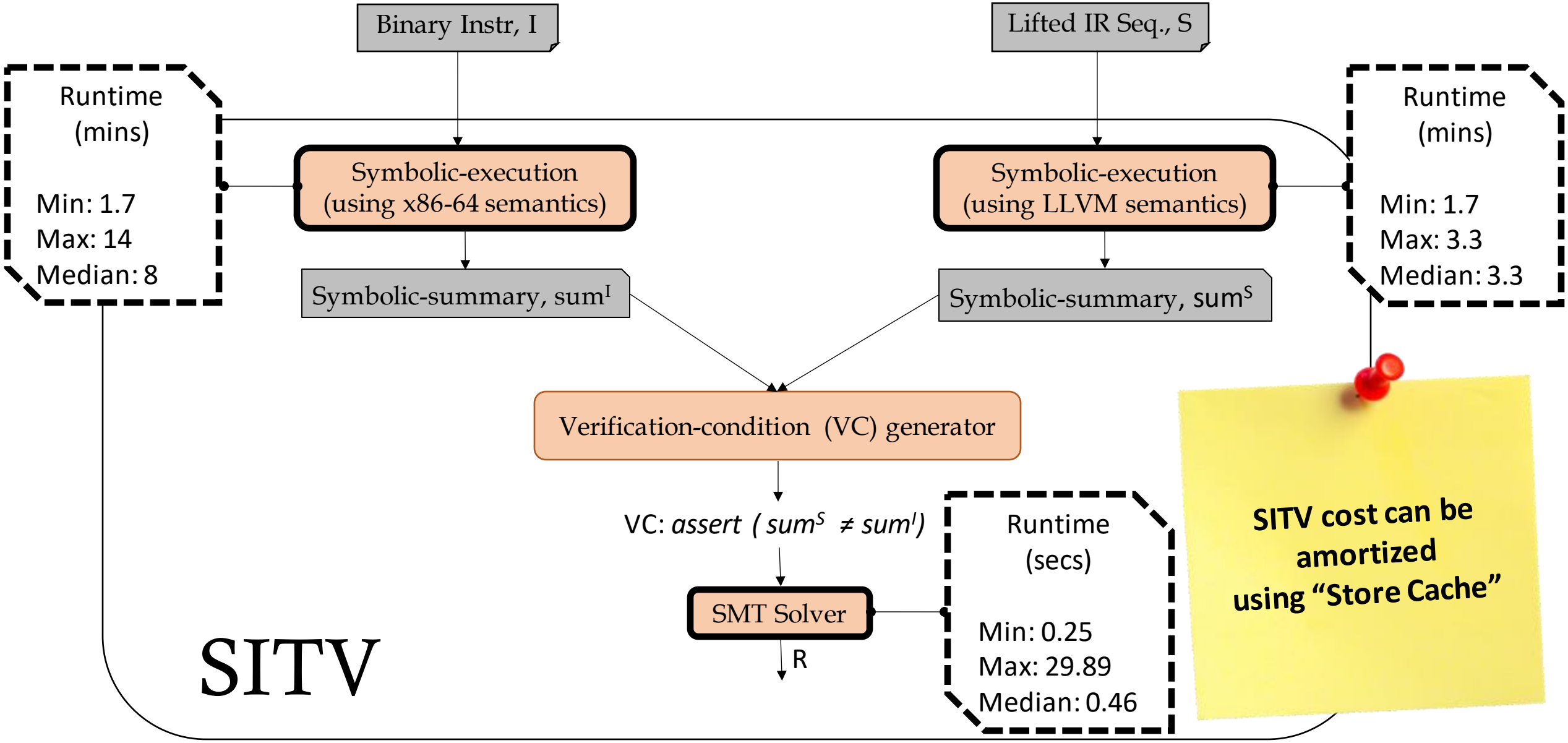

Binary Instr, I
Lifted IR Seq., S
• Derived from semantics using K
• Not Performance heavy: Loops rarely found in instructions' specs
Symbolic-execution (using x86-64 semantics)
Symbolic-execution (using LLVM semantics)
Symbolic-summary, sum^I
Symbolic-summary, sum^S
Verification-condition (VC) generator
VC: assert (sum^S ≠ sum^I)
SMT Solver
R == unsat
Translation Validated!
Validation Failed (Report Bug)
SITV

SITV: Evaluation Setup

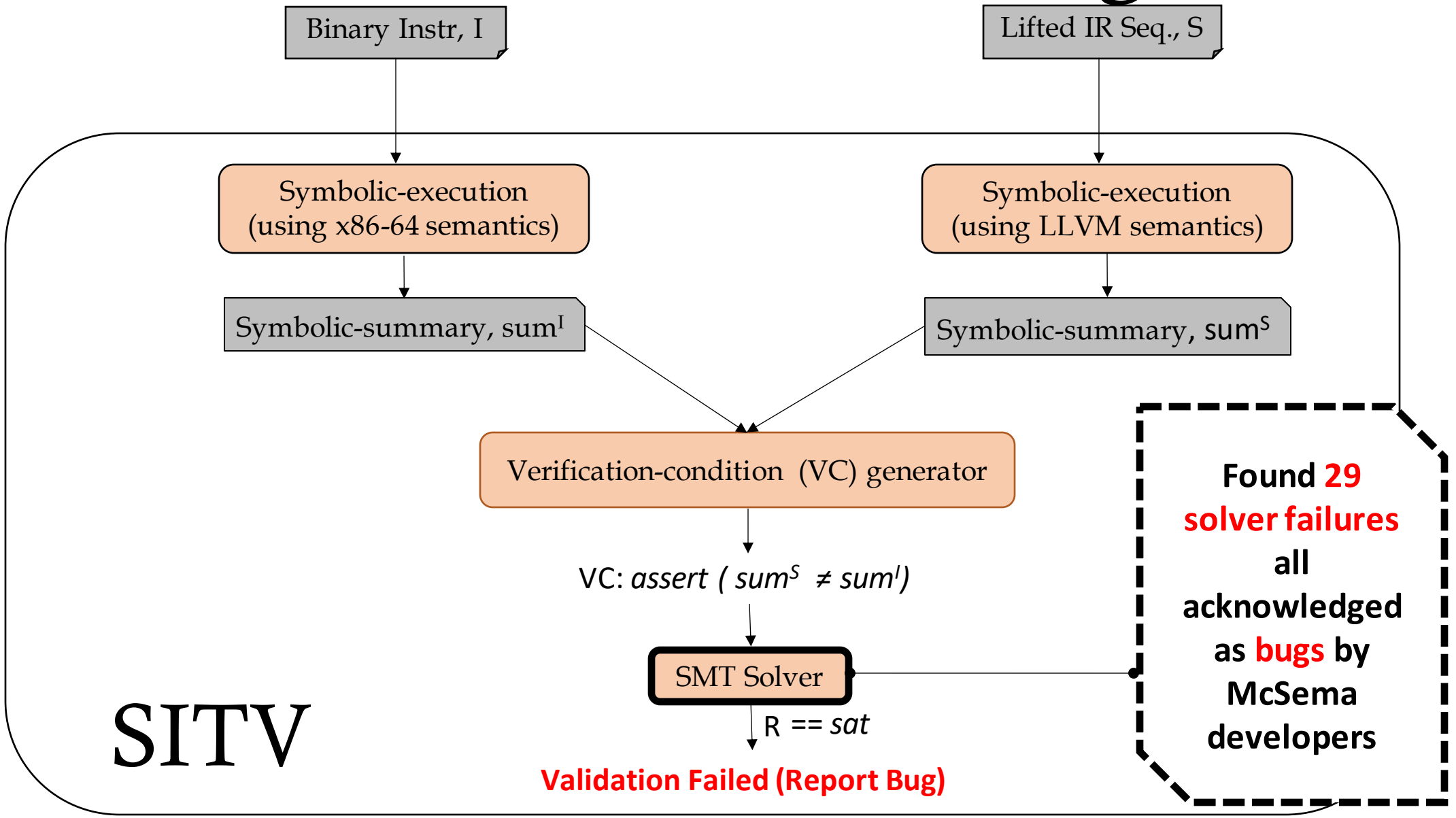
Applied translation validation on **1349 out of 3736** instruction variants

- ❑ McSema supports 1922 variants; all supported by our ISA model
- ❑ Exclude 573 because of limitations of LLVM semantics
  e.g., unsupported vector or FP types, intrinsic functions
- ❑ Solver runtime: min - 0.25 s, max – 29.89 s, median –

SITV: Performance



SITV: Revealed Bugs



SITV: A Few Reported Bugs

☑ Intel Manual Vol. 2: May 2019

xaddq %rax, %rbx

(1) temp ← %rax + %rbx
(2) %rax ← %rbx
(3) %rbx ← temp

🐞 McSema Implementation

xaddq %rax, %rbx
(with same operands)

(A) old_rbx ← %rbx
(B) temp ← %rax + %rbx
(C) %rbx ← temp
(D) %rax ← old_rbx

SITV: A Few Reported Bugs

Intel Manual Vol. 2: May 2019

pmuludqu (128-bit operands)

(1) DEST[63:0] ← DEST[31:0] * SRC[31:0]

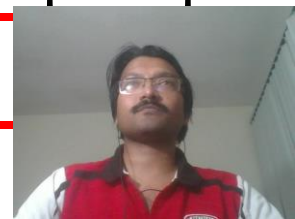
(2) DEST[127:64] ← DEST[63:32] * SRC[63:32]

McSema Implementation

pmuludqu (128-bit operands)

(1) DEST[63:0] ← DEST[31:0] * SRC[31:0]

(2) DEST[127:64] ← (unchanged)



SITV: A Few Reported Bugs

Intel Manual Vol. 2: May 2019

cmpxchgl %ecx, %ebx

```
TEMP ← ebx
IF eax = TEMP THEN
  ZF ← 1;
  ebx ← ecx;
ELSE
  ZF ← 0;
  eax ← TEMP;
  ebx ← TEMP;
FI;
```

McSema Implementation

cmpxchgl %ecx, %ebx

```
TEMP ← rbx
IF (32'0 ◦ eax) = TEMP THEN
  ZF ← 1;
  ebx ← ecx;
ELSE
  ZF ← 0;
  eax ← TEMP;
  ebx ← TEMP;
FI;
```

Lifter Validation: Our Approach

- **Phase I** **S**ingle-**I**nstruction **T**ranslation-**V**alidation (SITV)
- **Phase II** **P**rogram-**l**evel **V**alidation (PLV)

SITV⇏ PLV

```
.data
0x60f238: <GLOBL>
…
.text
someFunction:
  addq %rax, %rbx
  movq 0x60f238, %rax
```

Binary Program (P)

Pre-computed Simulated Address

```
define … @someFunction (%struct.State* %S, …) {
    %RAX = getelementptr ... %S, …;  Compute simulated RAX address
    %RBX = getelementptr ...  %S, …;  Compute simulated RBX address
    %RCX = getelementptr ...  %S, …; Compute simulated RCX address

    ; addq %rax, %rbx
    %VAL_RBX = load i64, i64* %RBX
    %VAL_RAX = load i64, i64* %RAX
    %X = add i64 %VAL_RAX, i64 %VAL_RBX
    store i64 %X, i64* %RBX

    ; mov 0x60f238, %rax
    %VAL_MEM = load i64, i64* %GLOBL
    store i64 %VAL_MEM, i64* %RAX
}
```

Lifted IR Program, T

SITV⇏ PLV

```
.data
0x60f238: <GLOBL>
…
.text
someFunction:
  addq %rax, %rbx
  movq 0x60f238, %rax
```

Binary Program (P)

```
define … @someFunction (%struct.State* %S, …) {

      %RAX = getelementptr ... %S, …;  Compute simulated RAX address
      %RBX = getelementptr ...  %S, …;  Compute simulated RBX address
      %RCX = getelementptr ...  %S, …; Compute simulated RCX address

      ; addq %rax, %rbx
      %VAL_RBX = load i64, i64* %RCX
      %VAL_RAX = load i64, i64* %RAX
      %X = add i64 %VAL_RAX, i64 %VAL_RBX
      store i64 %X, i64* %RBX

      ; mov 0x60f238, %rax
      %VAL_MEM = load i64, i64* %GLOBL
      store i64 %VAL_MEM, i64* %RAX
}
```

Lifted IR Program, T

SITV⇏ PLV

```
.data
0x60f238: <GLOBL>
…
.text
someFunction:
  addq %rax, %rbx
  movq 0x60f238, %rax
```

Binary Program (P)

```
define … @someFunction (%struct.State* %S, …) {

      %RAX = getelementptr ... %S, …;  Compute simulated RAX address
      %RBX = getelementptr ...  %S, …;  Compute simulated RBX address
      %RCX = getelementptr ...  %S, …; Compute simulated RCX address

      ; addq %rax, %rbx
      %VAL_RBX = load i64, i64* %RBX
      %VAL_RAX = load i64, i64* %RAX
      %X = add i64 %VAL_RAX, i64 %VAL_RBX
      store i64 %X, i64* %RBX

      ; mov 0x60f238, %rax
      store i64 6353464, i64* %RAX
}
```

Lifted IR Program, T

PLV: Our Approach

Compositional Lifting

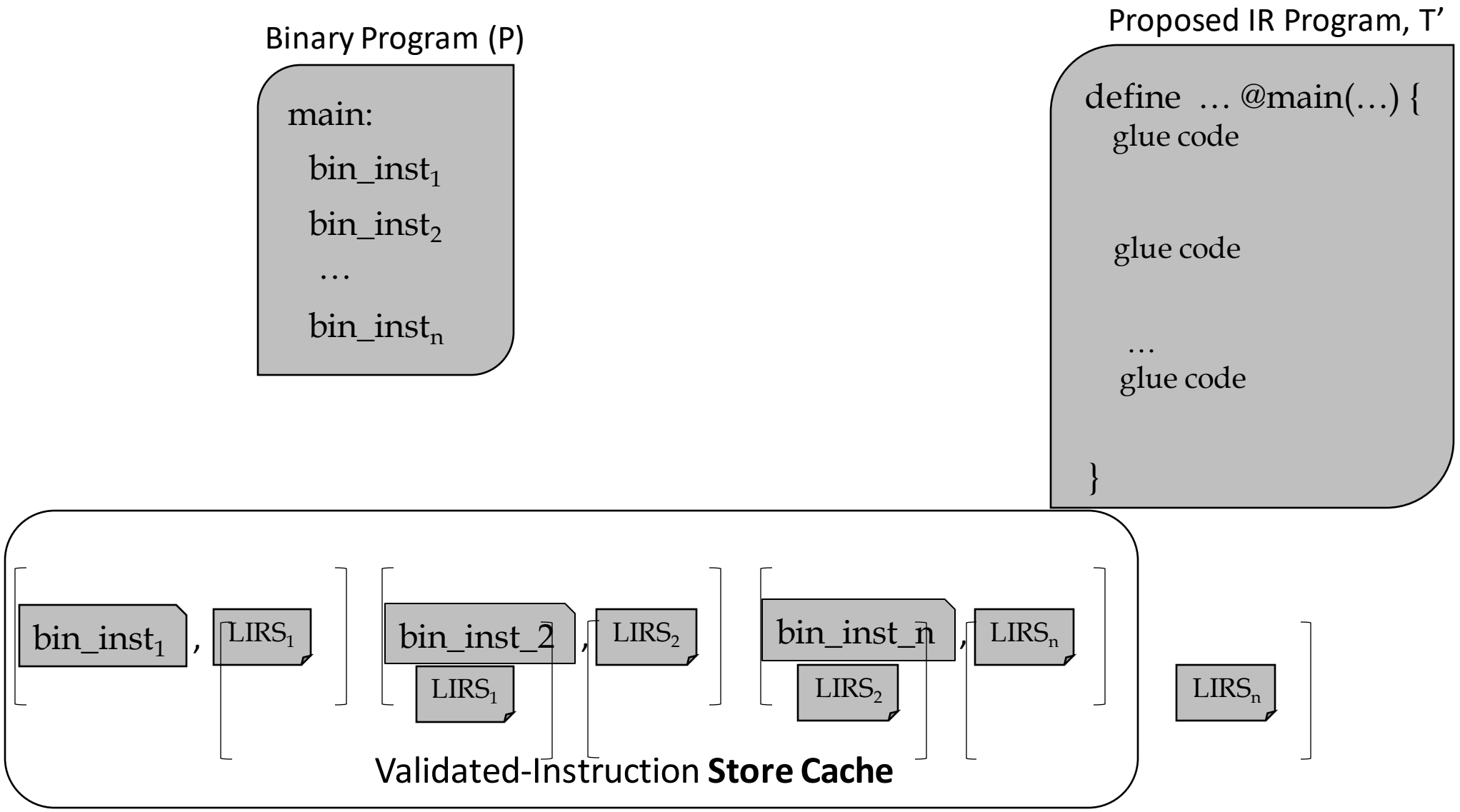
To propose an alternate reference program, T', generated by carefully stitching the validated lifted IR sequences (using SITV)

Transformation & Matching

- **Transformation:** Uses semantic preserving transformations to reduce T' and original lifted program (T) to a common form

- **Matching:** Checks the data-dependence graphs of transformed versions for graph-isomorphism

PLV: Compositional Lifting

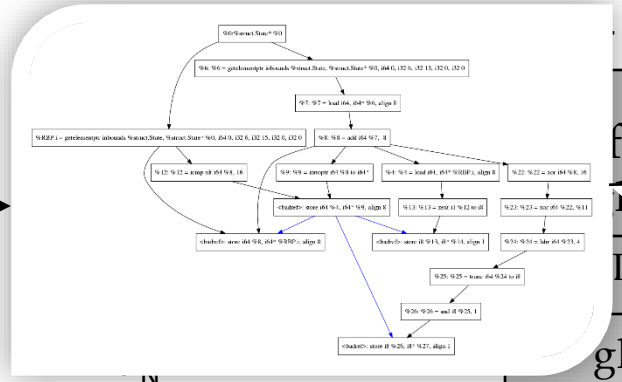


PLV: Normalization & Matching

```
define … @main(…) {
    ir_inst1
    ir_inst2
      …
    ir_instm
}
```

For each function F of T

Normalizer (LLVM passes)



Data-Dependence Graph, G_N

each function F' of T'

```
fine … @main(…) {
```

LIRS$_1$

glue code

LIRS

…

glue

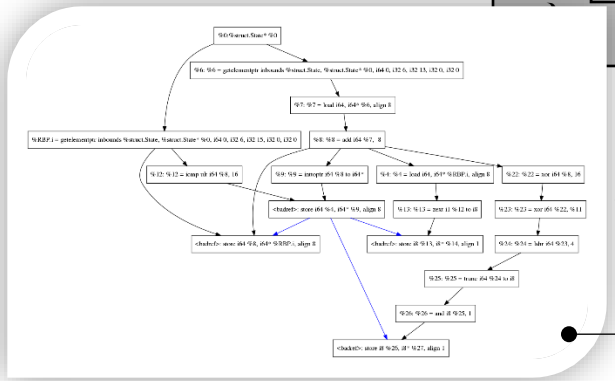
LIRS$_n$

Matcher
Are G & G'$_N$ isomorphic?

T & T' semantically equivalent

Potential Bug in Lifter

Normalizer (LLVM passes)



Data-Dependence Graph, G'$_N$

Vertex:
LLVM instruction
Edge:
SSA def-use or memory dependence relation

PLV: Extra Diagram

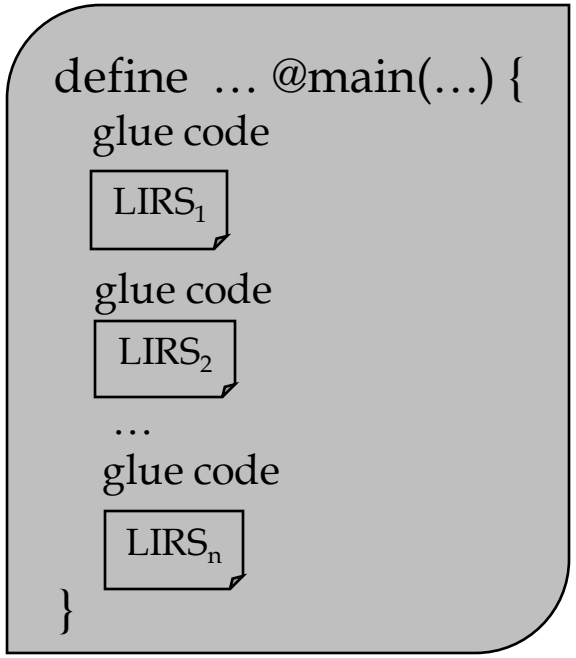
someFunc:

```
400494:  mov    %edi,-0x8(%rbp)
400497:  cmpl   $0x1,-0x8(%rbp)
40049b:  jge    4004ad
4004a1:  movl   $0x1,-0x4(%rbp)
4004a8:  jmpq   4004b4
4004ad:  movl   $0x0,-0x4(%rbp)
```

…

…

For each function F of T



Compositional Lifter: Algorithm

Inputs : **P**: x86-64 binary program.
Store: Validated pairs (<I, S>) of instruction I and lifted IR sequence S (possibly empty)
Output: Lifted IR Program T′

```
T′ ← φ
foreach I in P do
    if I not in Store then
        S ← McSema (I)
        Perform Translation Validation of I and S (Phase I)
        if Validation successful then
            Add < I , S > to Store Cache
        else
            Report Bug
        end
    else
        Extract S from Store Cache corresponding to I
    end
    T′ ← Compose(T′ , S)
end

return T′
```

Very simple in design, however,
PLV can be done prior to
No formal guarantee that T′
is the correct translation of P

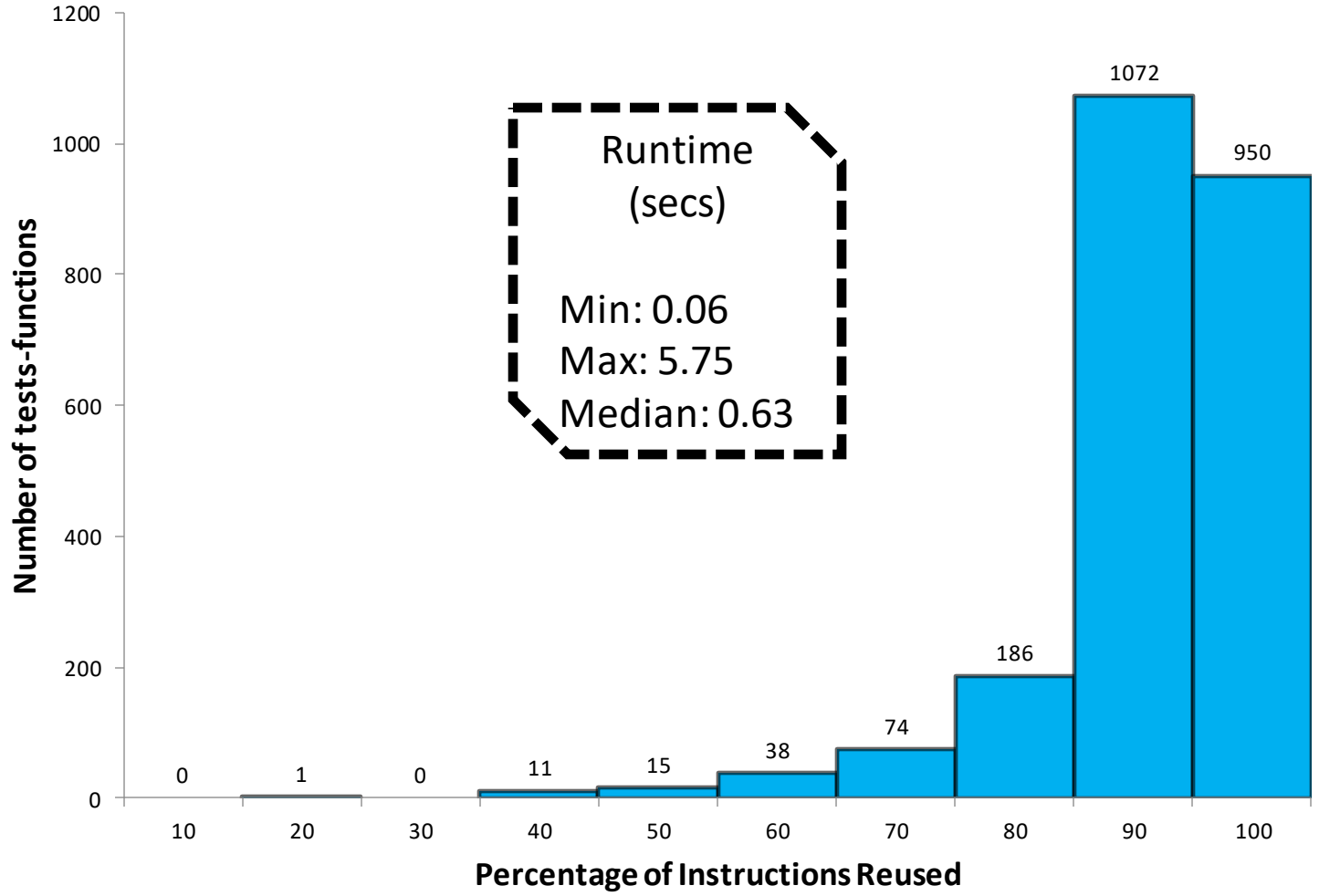
Could be done offline !

SITV

Performance depends heavily on the availability of instructions in Store Cache

Compositional Lifter: Evaluation

Evaluated on **2348 binaries** compiled from LLVM single-source benchmark functions



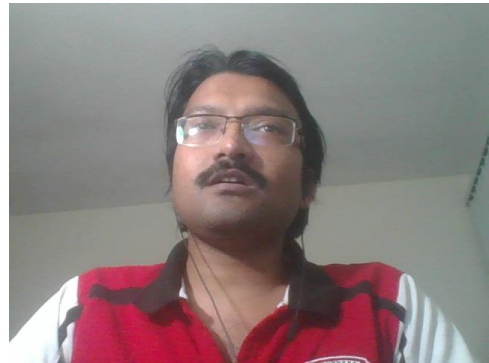
Normalizer

- Prunes-off syntactic differences between T & T' except for
  - **Names of virtual registers**, and
  - **Order of non-dependent instructions**

Optimization passes NOT formally-verified

- Uses **17 LLVM optimizations passes** (manually discovered)

mem2reg licm gvn early-cse globalopt simplifycfg basicaa aa memdep dse deadargelim libcalls-shrinkwrap tailcallelim simplifycfg basicaa aa instcombine



Matcher: Iso-Graph Algorithm

(Borrowed from Saltz at al.*)

1. **Finding ϕ, Initial Match Set, $O(n2)$**: For each node n of G, find all potential matches n' in G'

2. **Iterative Step:** Iteratively prunes out elements from ϕ of each vertex based on its parents/child relations until fixed-point is reached

Time: $O(n^2 \times |\phi|)$ and $|\phi| = O(n)$

*An Algorithm for Subgraph Pattern Matching on Large Labeled Graphs, IEEE International Congress on Big Data'14

Matcher: Iso-Graph Algorithm

(Borrowed from Saltz at al.*)

1. **Finding ϕ, Initial Match Set**: For each node n of G, find all potential matches n' in G'

2. **Iterative Step:** Iteratively prunes out elements from ϕ of each vertex based on its parents/child relations until fixed-point is reached

*An Algorithm for Subgraph Pattern Matching on Large Labeled Graphs, IEEE International Congress on Big Data'14

Constraining ϕ: Our Approach

1. **Finding ϕ, Initial Match Set**: For each node n of G, find all nodes n' in G' s.t n & n' satisfies
   - Same instruction opcode
   - Same constant operands
   - Same number of outgoing edges

| ϕ | << n

Improves the complexity of iterative step

Matcher: Evaluation

- Run Matcher on **2348** LLVM single-source benchmark functions
  - Runtime: ranges from 0.06s – 119.63s, median - 4.91s

- Proved correctness of 2189 /2348 translations; **success rate - 93%**
  - LOC of lifted IR: ranges from **86 – 32105, median - 611**
  - Remaining 159 manually inspected as false negatives; **rate - 7%**

- No real bugs found: Effectiveness evaluated using artificially injected bugs

Normalizer: Phase Ordering Problem

Observation

- ❑ Changing the order of normalizer passes improves matching results
- ❑ Not all of 17 passes are needed for every pair of functions

Intuition

To frame the problem of selecting the normalizing pass sequence as an application of pass-sequence autotuning

Autotuning Based Normalizer

Instead of using a fixed-length normalizer pass-sequence for all function pairs, we will **use an autotuner to find optimal pass-sequences one for each function pair**

Autotuning Based Normalizer

Used OpenTuner* framework for autotuning

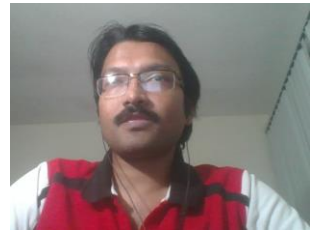
- **Search Space:** Includes passes from the 17-length pass sequence

- **Objective Function:** Maximize $\frac{t}{n}$

  $n =$ number of vertices in G

  $t =$ number of nodes in G having non-empty ϕ

* OpenTuner: An Extensible Framework for Program Autotuning, PACT'14



Autotuning Pipleline

Inputs: **F, F′** : Function pair compared for equivalence
S : Autotuner Search Space
B : Resource Budget
C : Objective-Function

Output: Set of candidate normalization passes satisfying **C** within **B**

```
candidate-passes = φ
while( B not exhausted )
        t    = Autotuner-Search(S)
        F_N  = Normalizer(F,  t)
        F'_N = Normalizer(F', t)

        if check-objective-function-is-met(C, G_N, G'_N)
                candidate-passes = candidate-passes U t
end
return candidate-passes
```

Improved Matcher Pipeline

Inputs: **F, F′**: Function pair compared for equivalence
candidate-passes: Autotuner generated candidate pass sequences

Output: **true** → F & F ′ semantically equivalent
false → F & F ′ may-be non-equivalent

```
foreach t in candidate-passes do
        F_N  = Normalizer(F,  t)
        F'_N = Normalizer(F', t)

        if IsGraphIsomorphic(G_N, G'_N)
                return true
        end
end
return false
```

Autotuning Based Normalizer: Results

- Opentuner runtime range from 10.7 s - 19.97 m, median - 6.67 m
- Reduces **false-alarm rate from 7% to 4%**
- Length of autotuned-pass-sequence: **median - 7, mean – 8** (< 17 !)

Summary

- Validation of lifters w/o instrumentation or heavyweight equivalence checking is feasible

- Capitalized on a simple insight

  Formal translation validation of single machine instructions is not only practical but also can be used as a building block for scalable full-program validation

- SITV valuable in finding real bugs in a mature lifter

- Proposed scalable full-program validation approach leveraging SITV

Questions