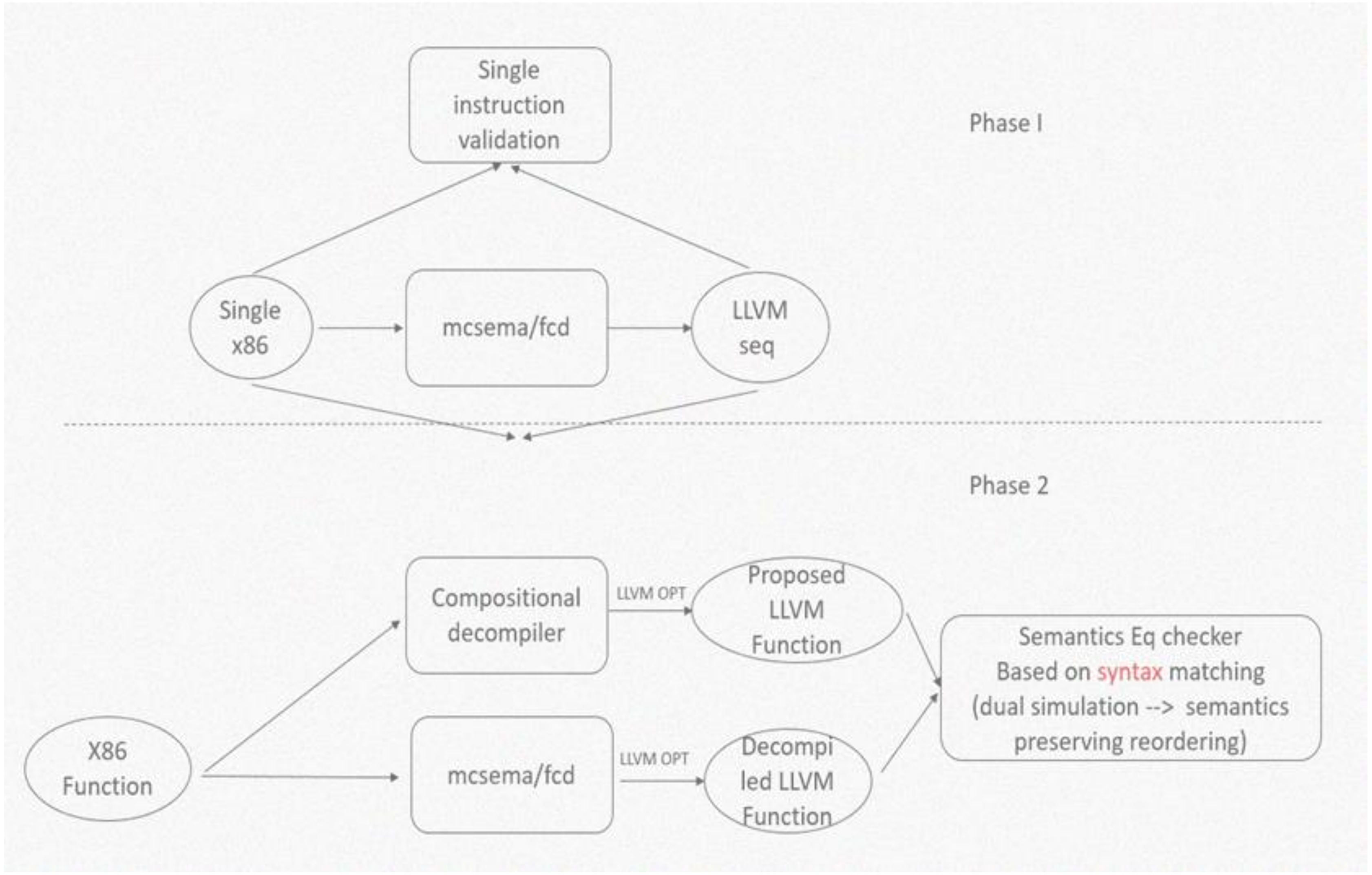
Scalable Validation of Binary Lifters



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

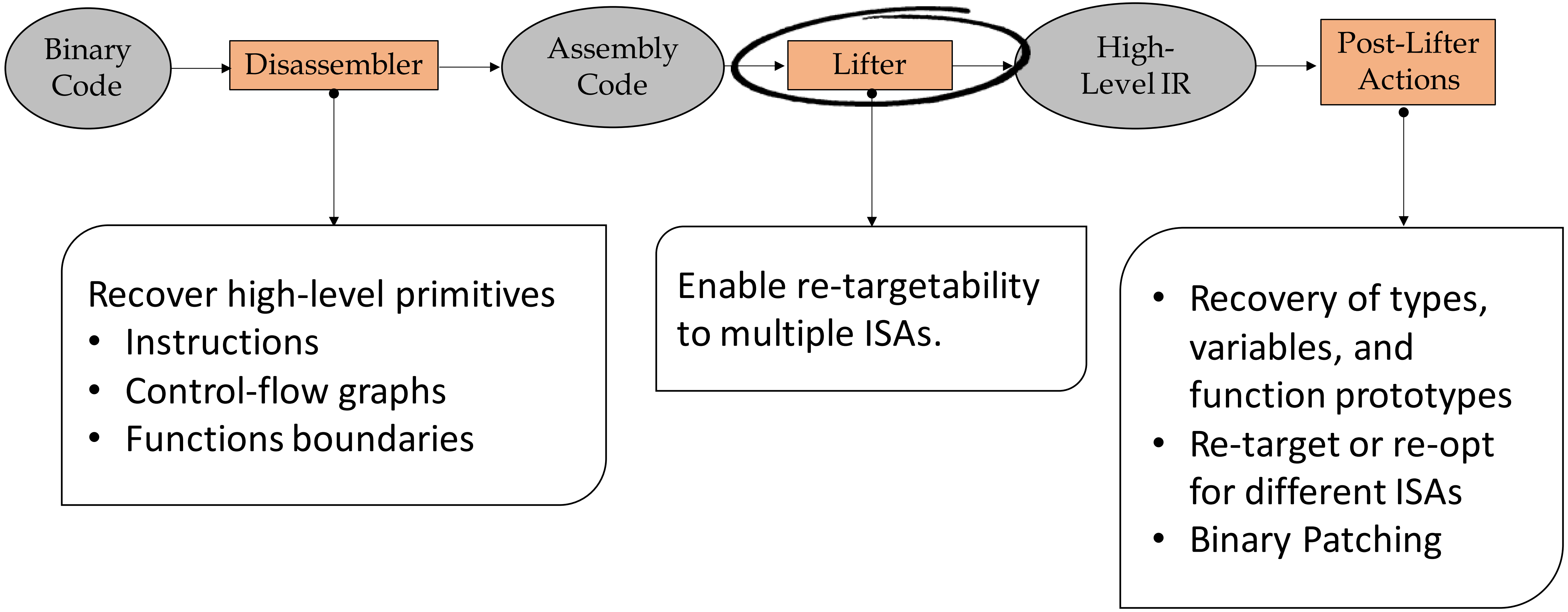
Binary Analysis is Important

The ability to directly reason about binary is important

<u>scenarios where binary analysis is useful</u>

- ❑ 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.

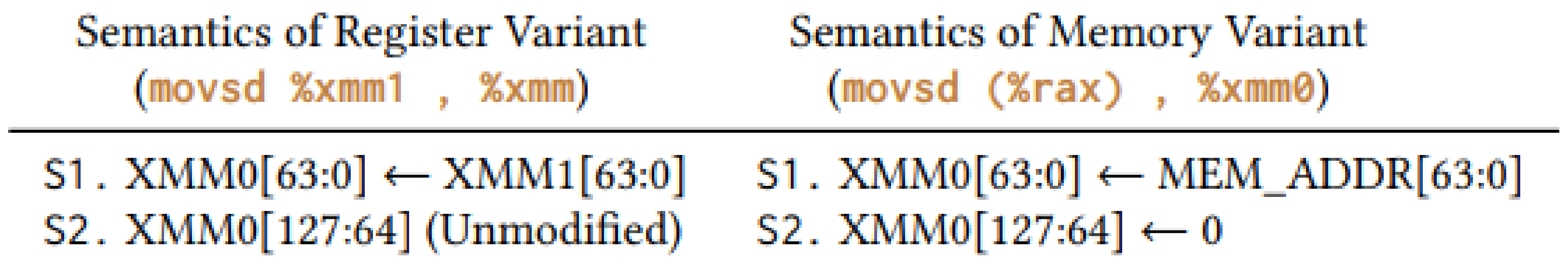
- Recovery of types, variables, and function prototypes
- Re-target or re-opt for different ISAs
- Binary Patching

Lifting is Challenging

Manual encoding the effects of binary instructions is hard

- ❑ Vast number of instructions

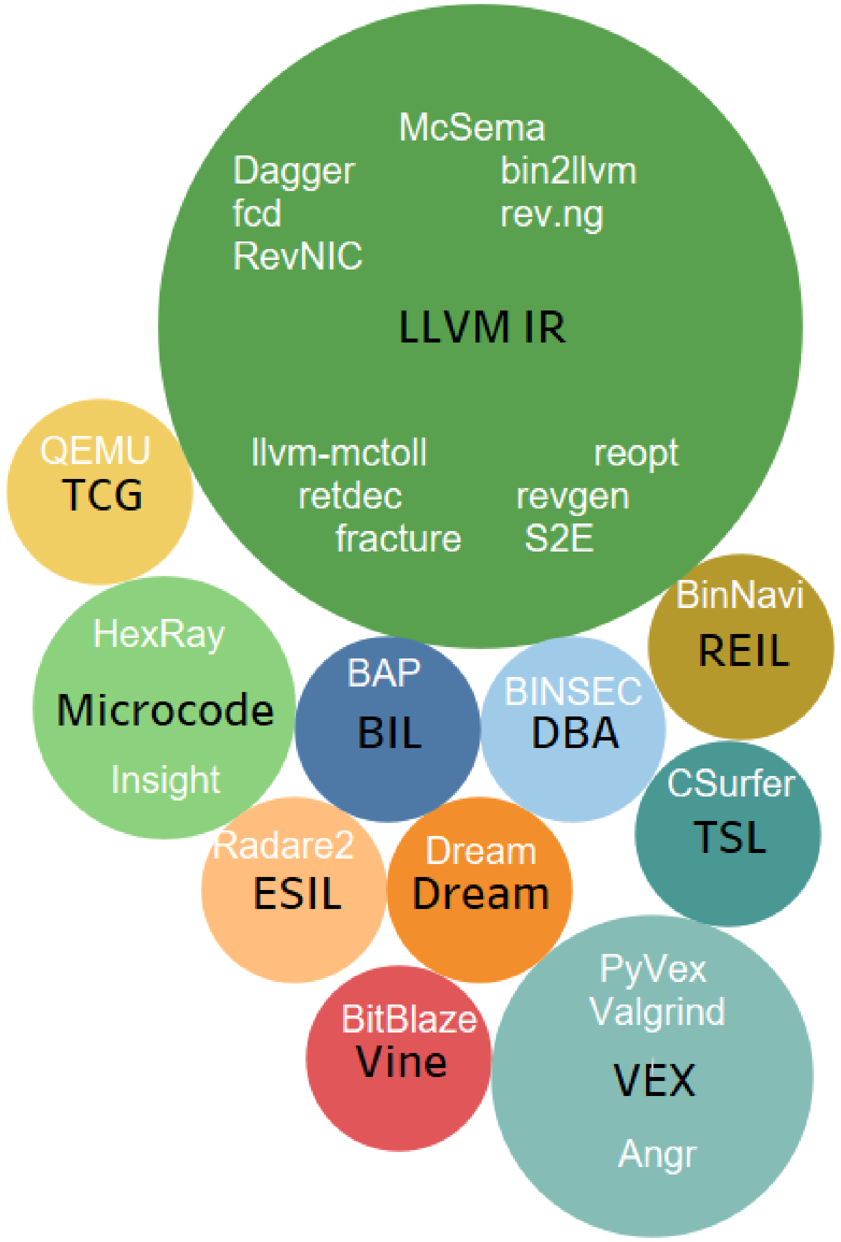
- ❑ Standard manuals are often ambiguous, buggy, include divergence in the behaviours of variants



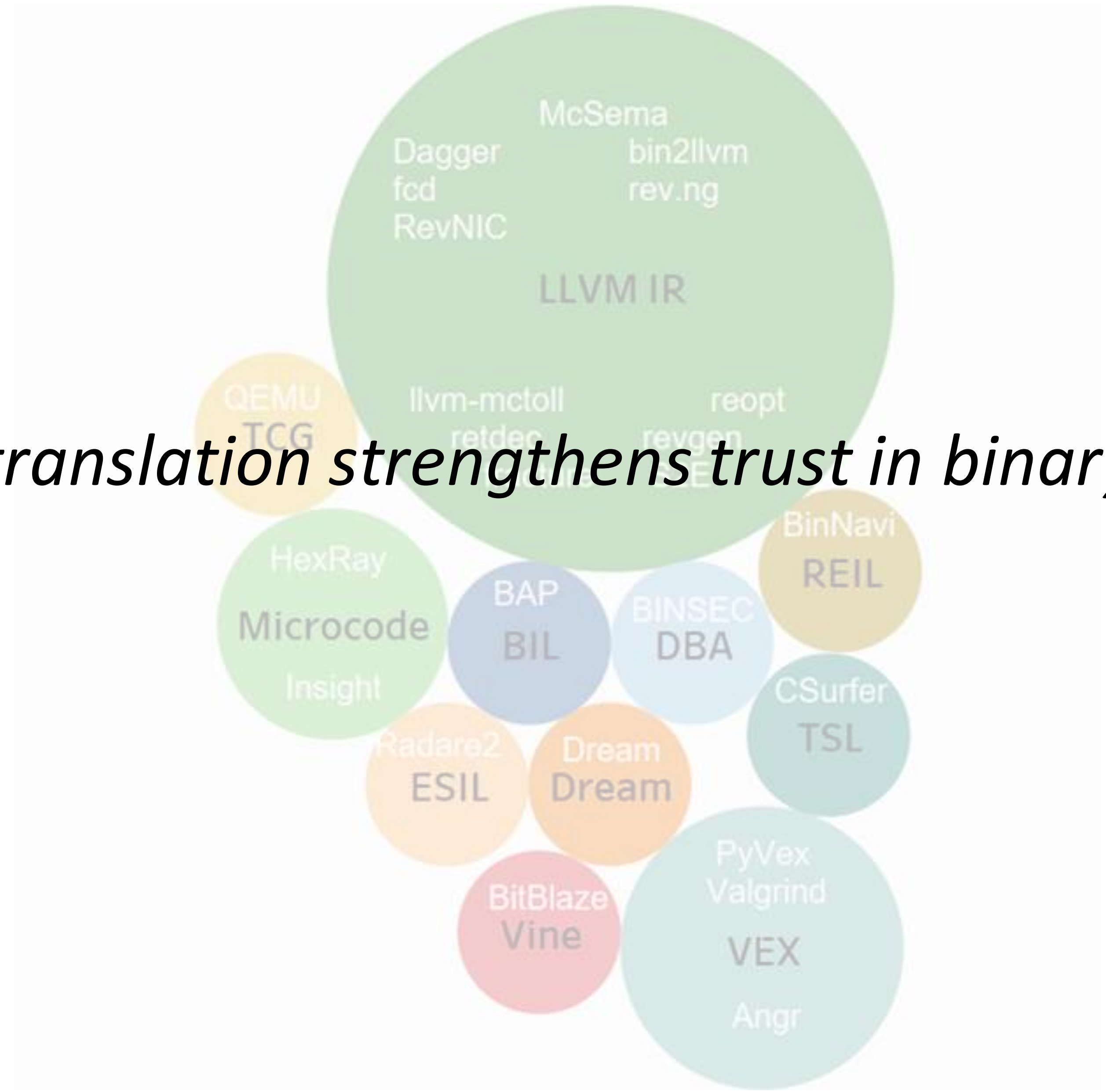
Semantics of Register Variant (`movsd %xmm1 , %xmm`)	Semantics of Memory Variant (`movsd (%rax) , %xmm0`)
S1. XMM0[63:0] $\leftarrow$ XMM1[63:0]	S1. XMM0[63:0] $\leftarrow$ MEM_ADDR[63:0]
S2. XMM0[127:64] (Unmodified)	S2. XMM0[127:64] $\leftarrow$ 0

- ❑ 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

- Martignoni et al. ISSTA'10
- Chen et al. CLSS'15

Restricted to instruction- or basic-block-level validation

- Martignoni et al. ISSTA'10, ASPLOS'12
- Chen et al. CLSS'15
- Meandiff - Kim et al. ASE'17

Require instrumentation

- 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

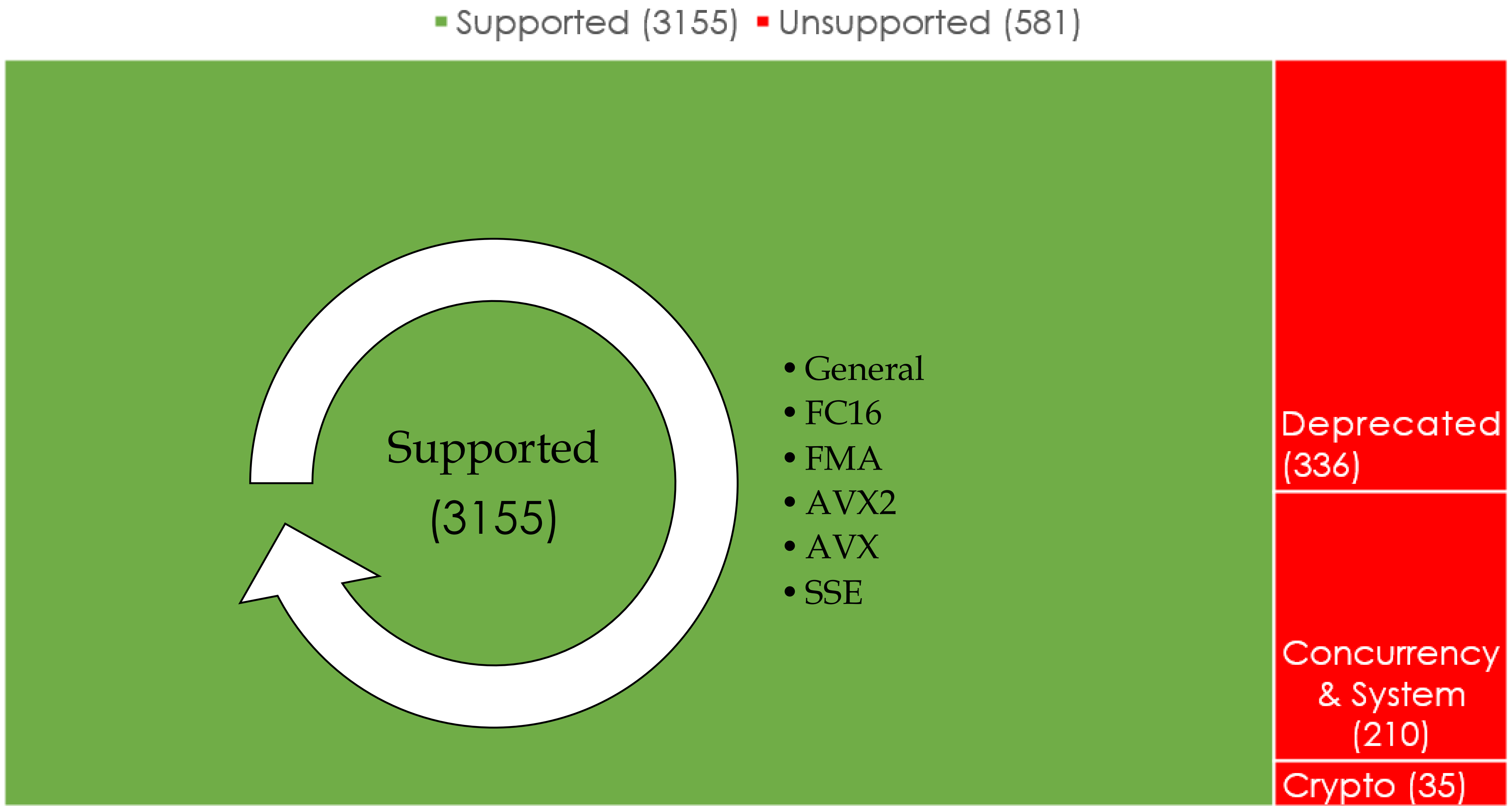
- ❑ **Developing scalable technique for validating lifters** (PLDI’20)
  - **First SIV framework** for an extensive x86-64 ISA
  - **Revealed Bugs** in a mature lifter like McSema
  - **Novel Technique** for SITV-assisted full-program validation

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

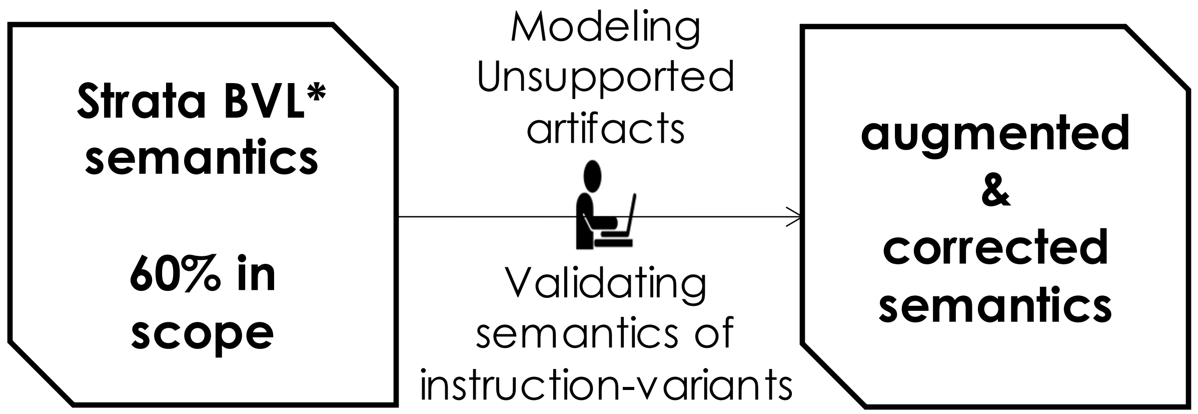
Approach Overview

Strata BVL*
semantics

60% in scope

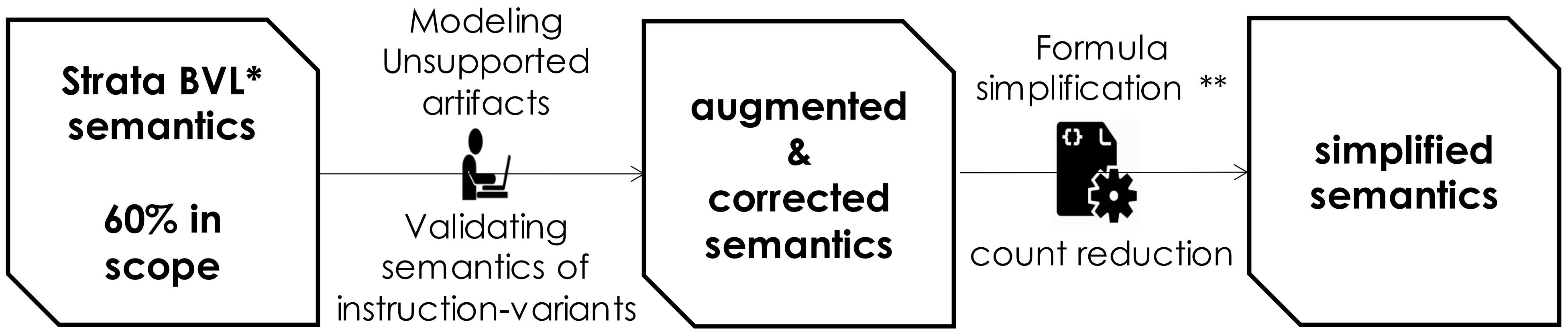
** BVL: Bit-vector logic*

Approach Overview


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

* *BVL: Bit-vector logic*

Approach Overview

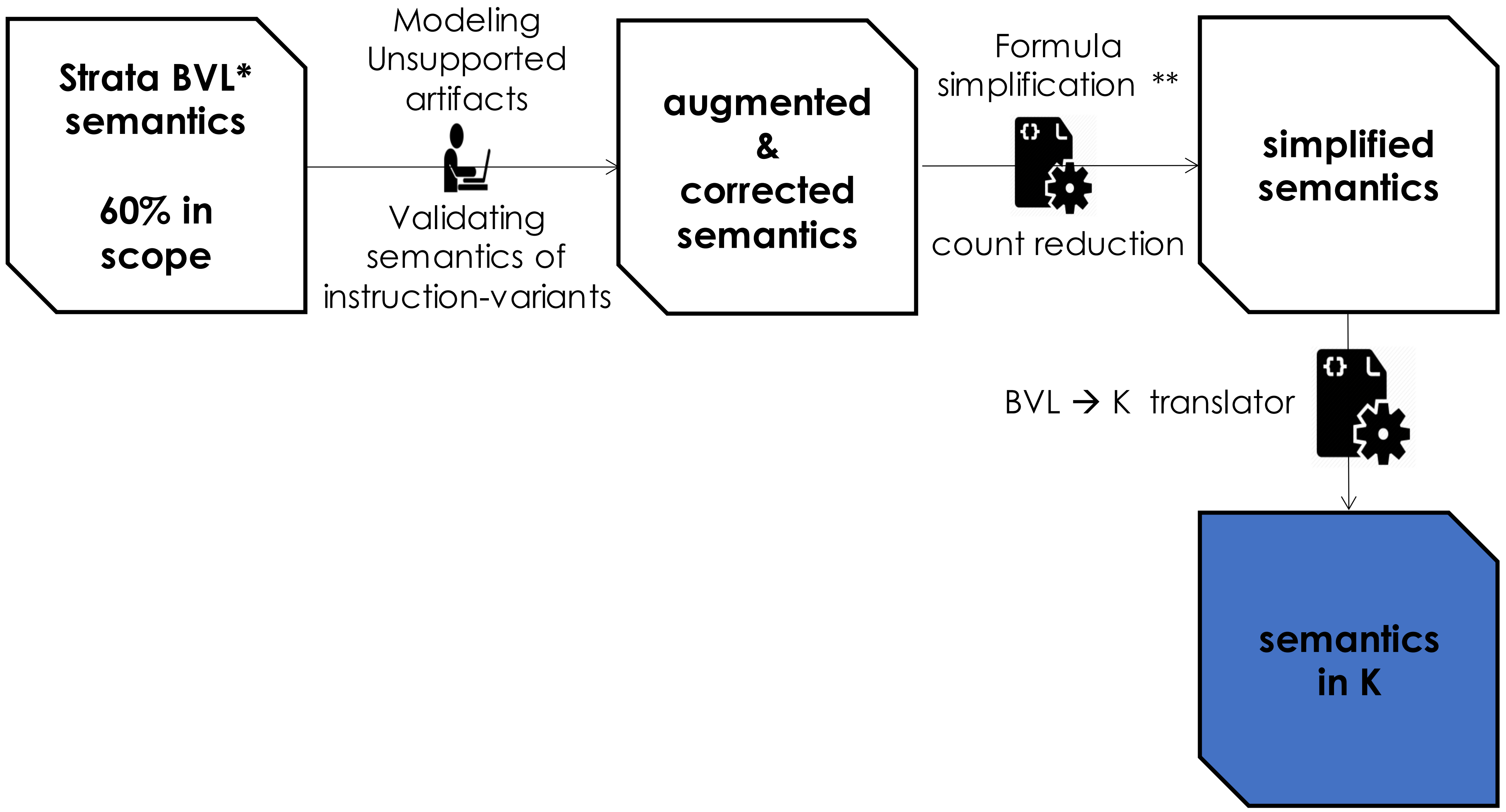


Strata BVL* semantics
60% in scope
Modeling Unsupported artifacts
Validating semantics of instruction-variants
augmented & corrected semantics
Formula simplification **
count reduction
simplified semantics

* *BVL: Bit-vector logic*

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

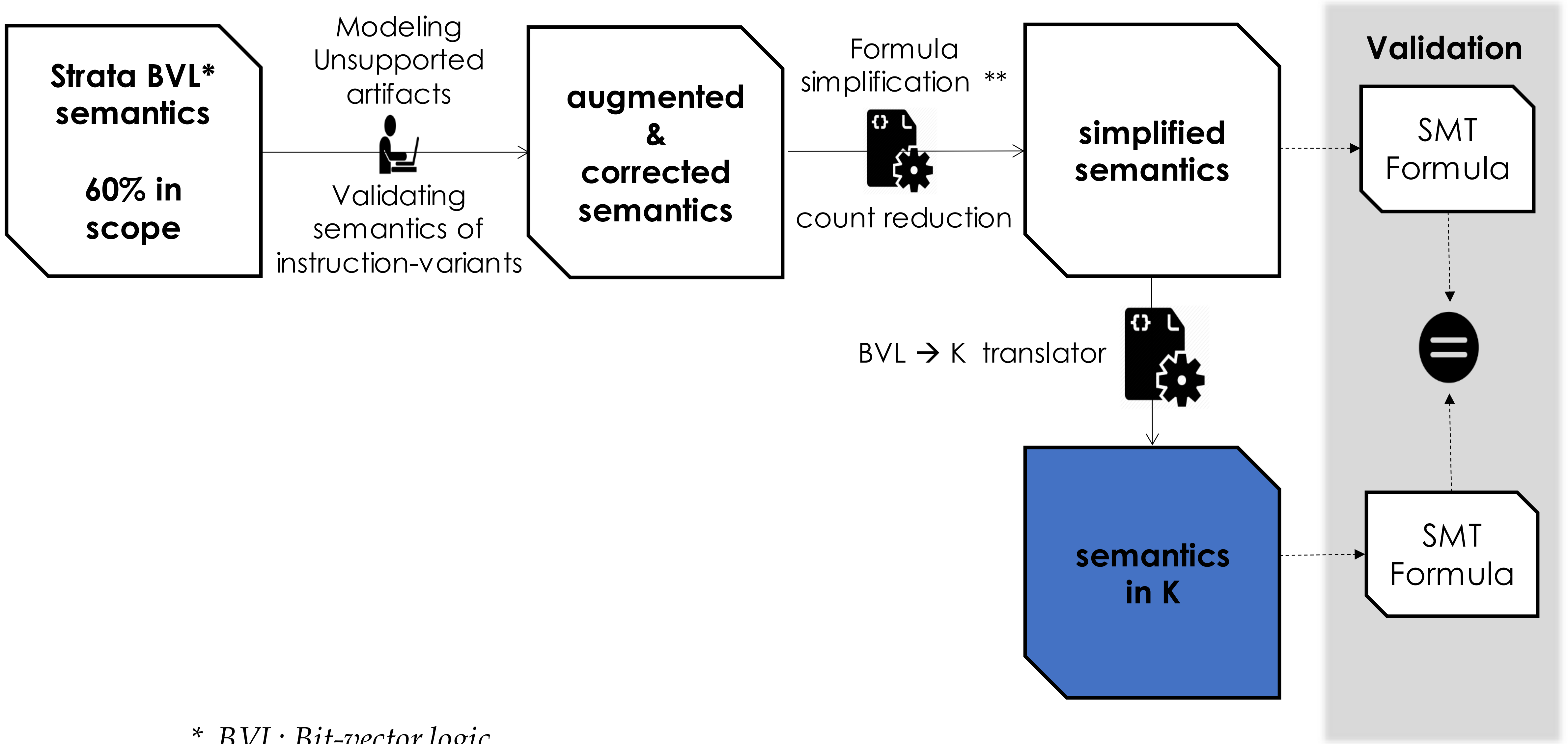
Approach Overview


Strata BVL* semantics
60% in scope
Modeling Unsupported artifacts
Validating semantics of instruction-variants
augmented & corrected semantics
Formula simplification **
count reduction
simplified semantics
BVL → K translator
semantics in K

* *BVL: Bit-vector logic*

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

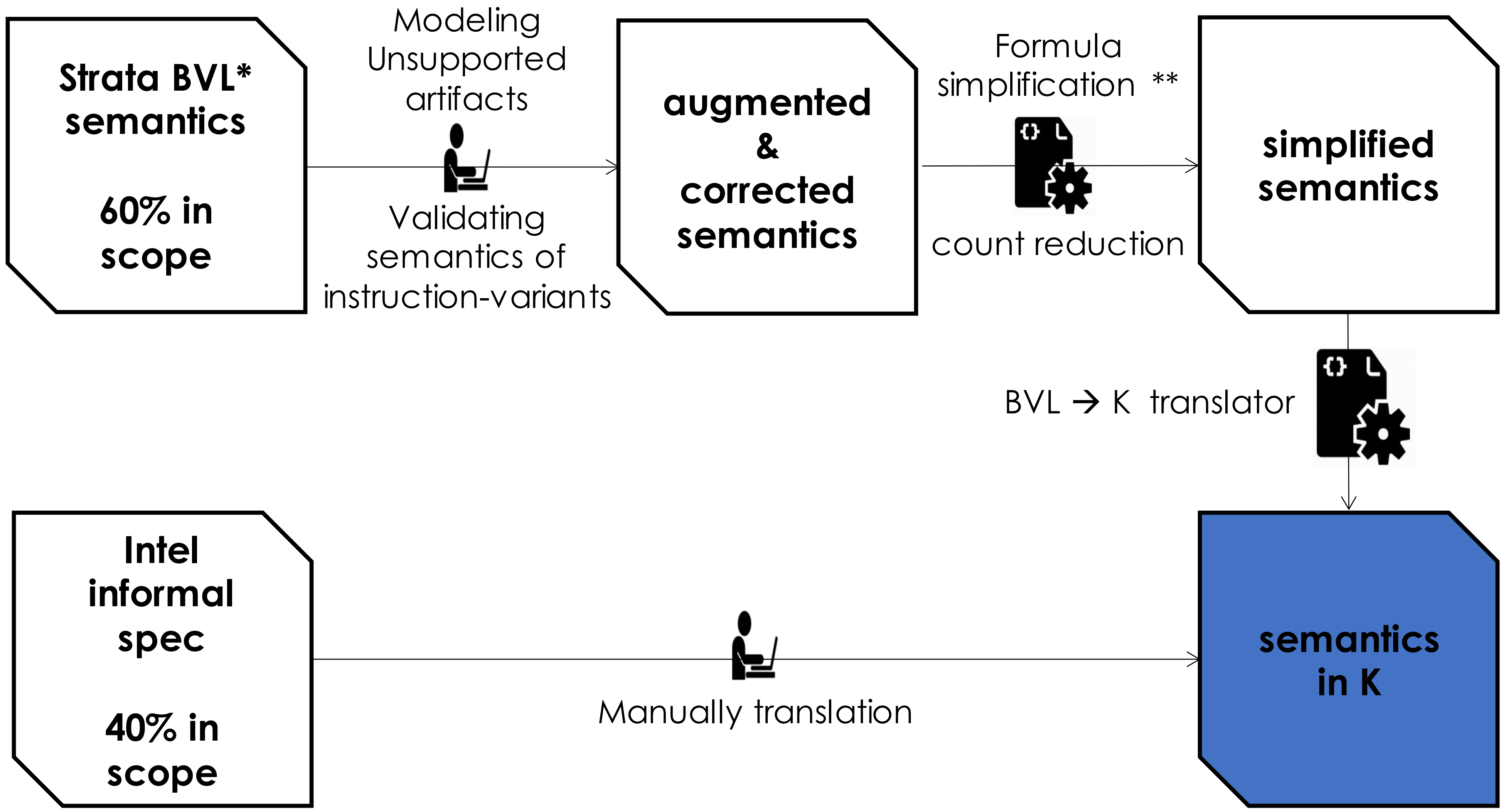
Approach Overview



* *BVL: Bit-vector logic*

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

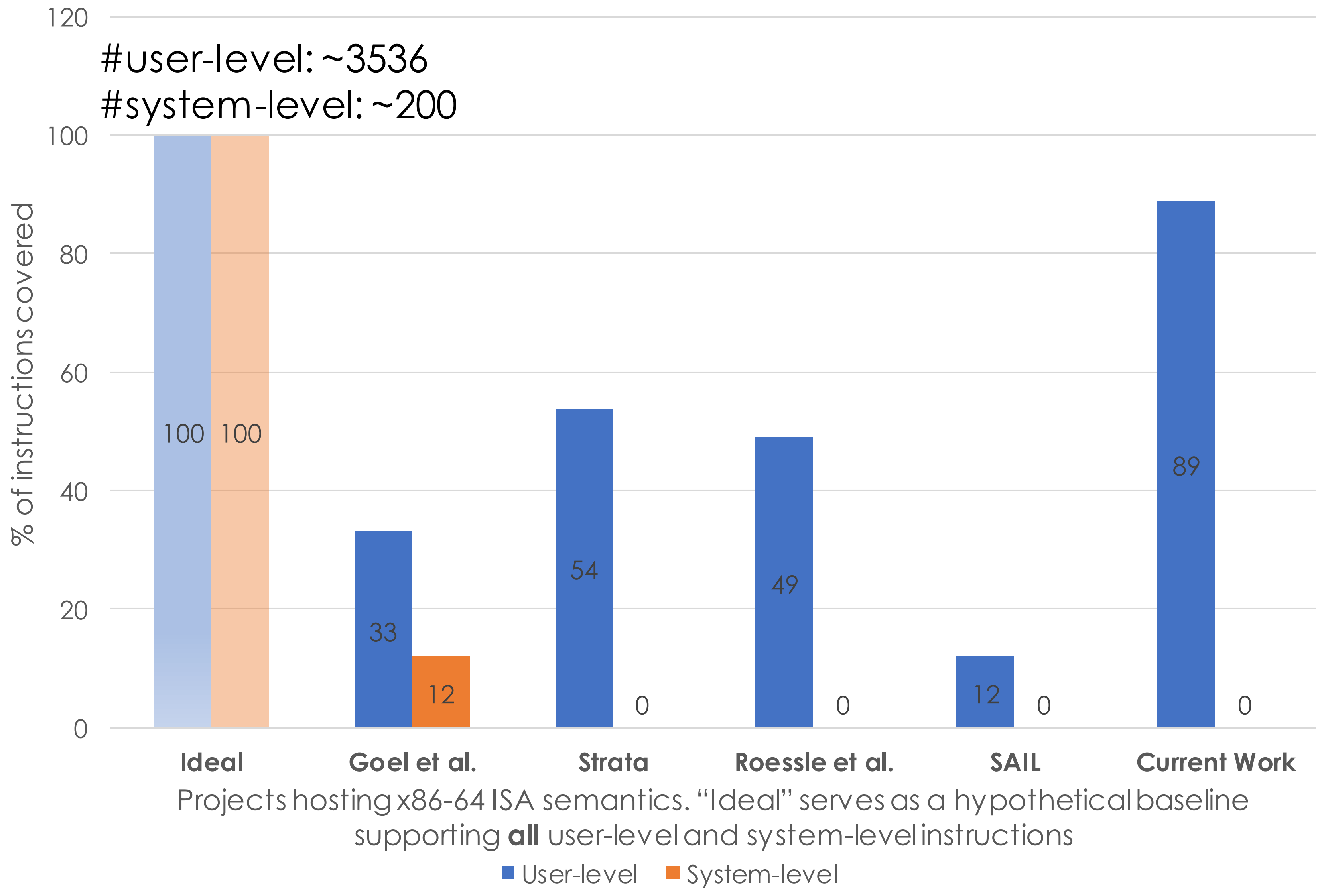
Approach Overview



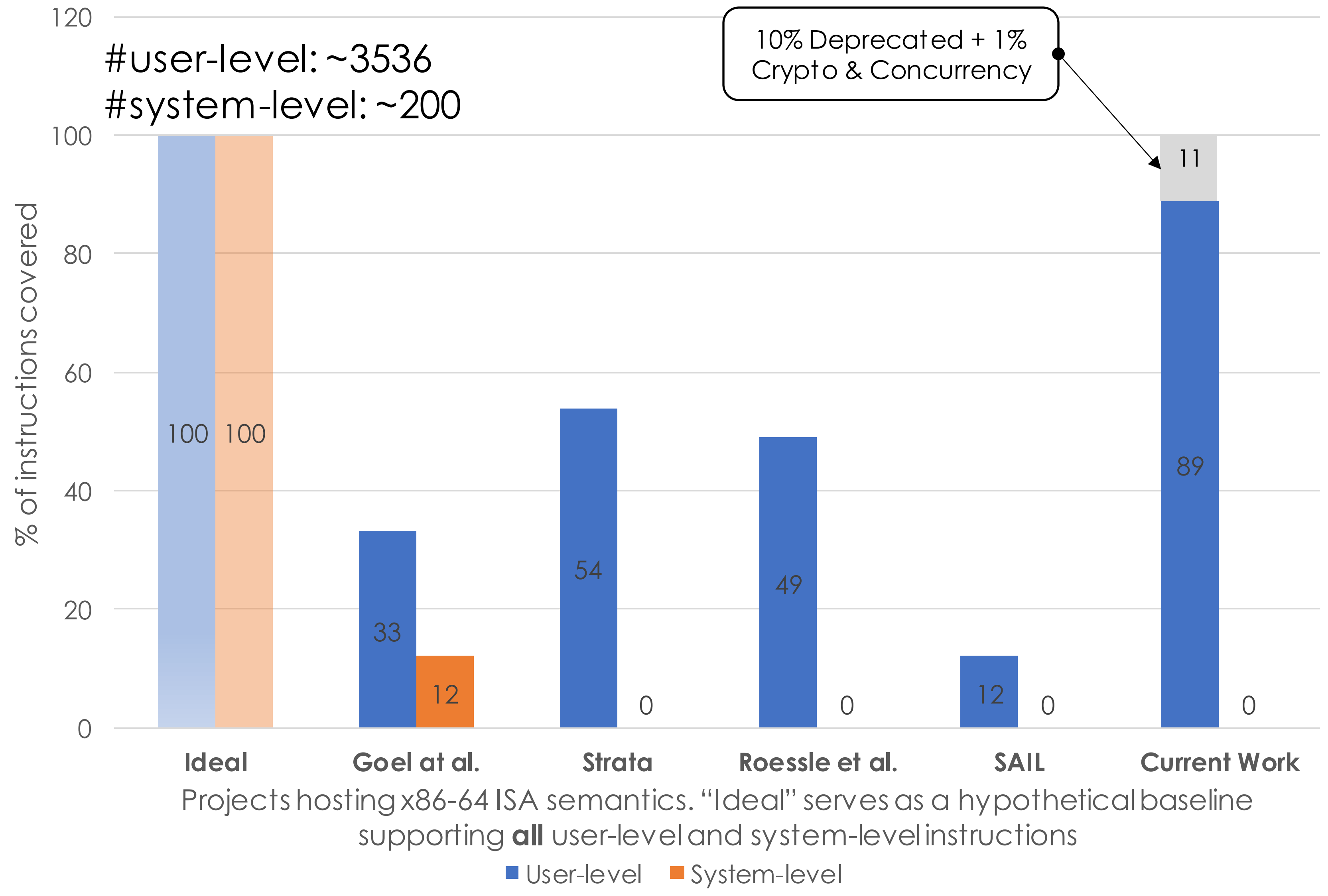
* *BVL: Bit-vector logic*

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

Support Comparison

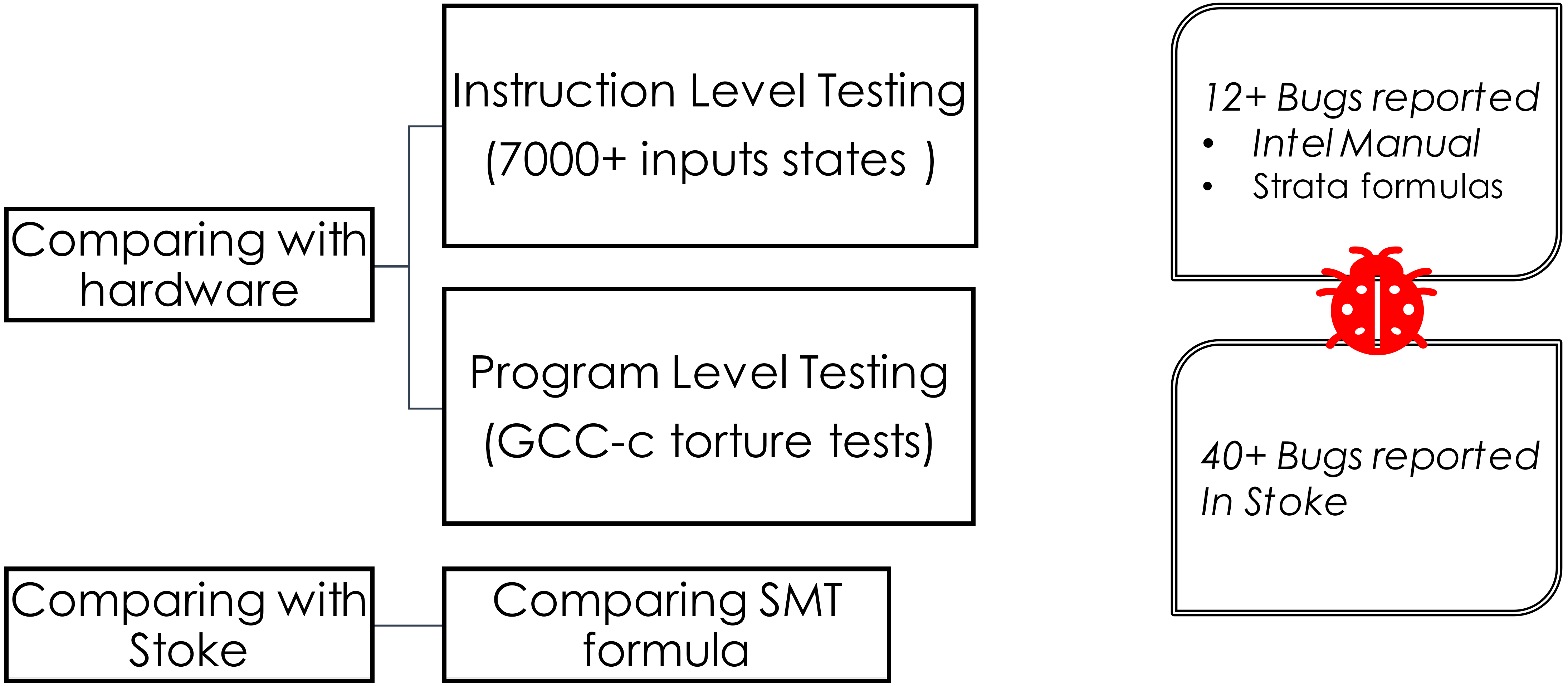


Support Comparison



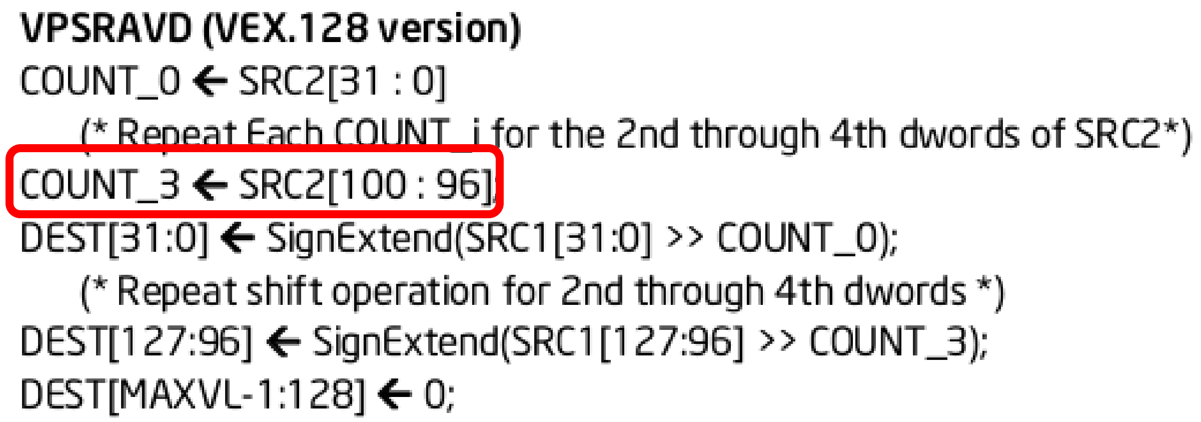
Projects hosting x86-64 ISA semantics. "Ideal" serves as a hypothetical baseline supporting **all** user-level and system-level instructions

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

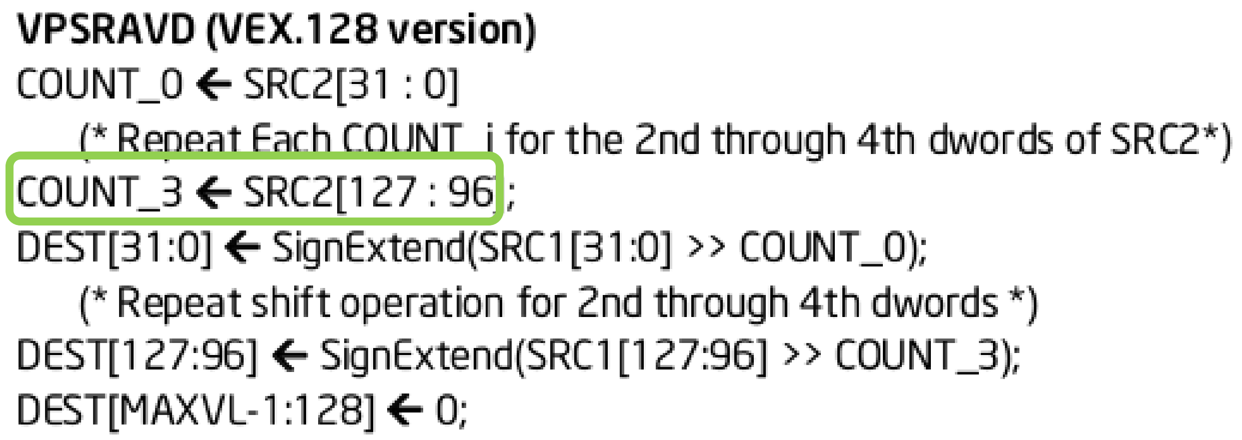
Intel Manual Vol. 2: March 2018



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

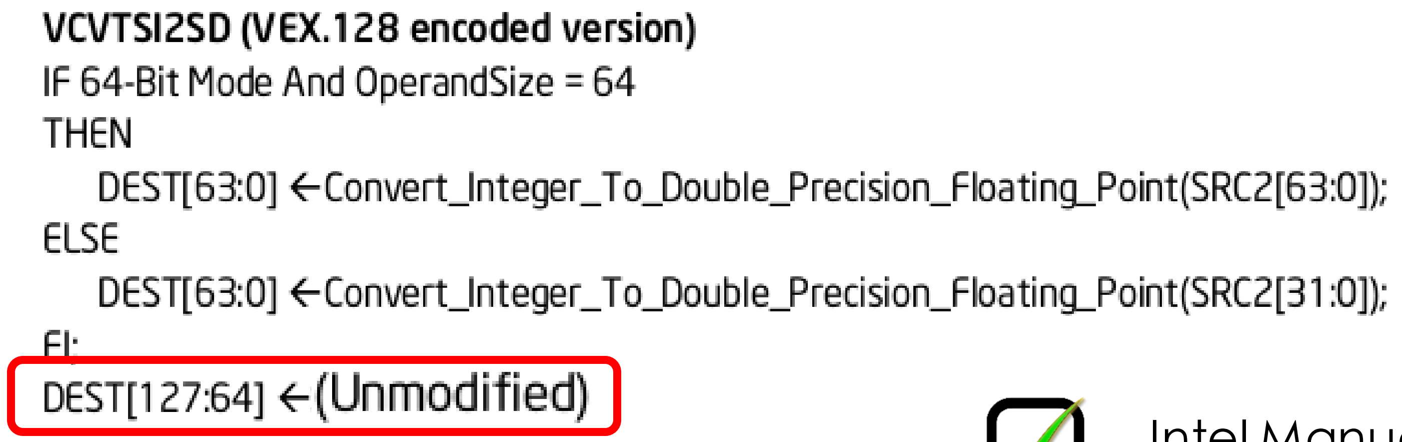


VPSRAVD (VEX.128 version)

```
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;
```

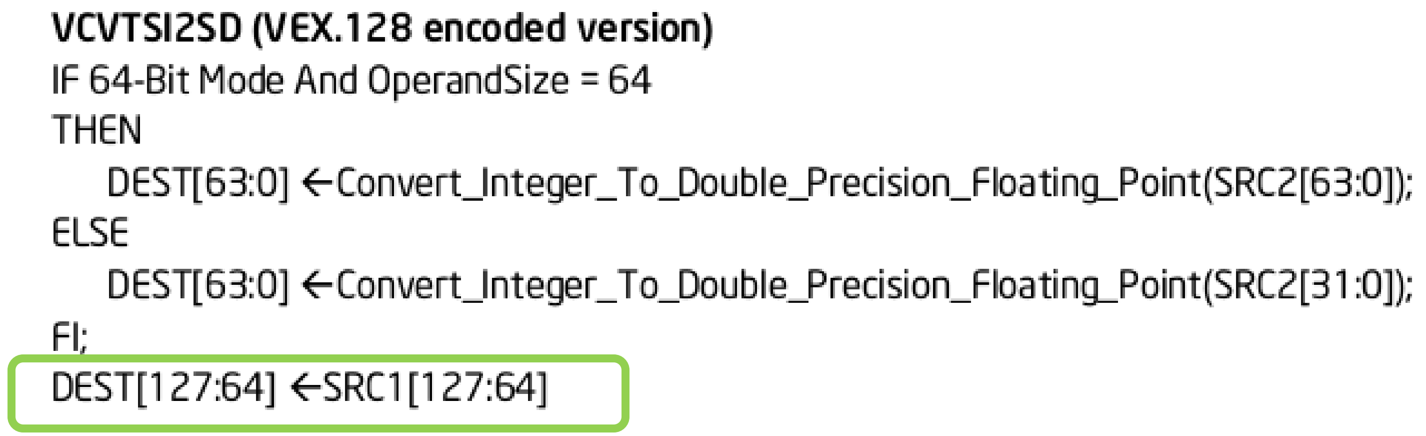
A Few Reported Bugs

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]
```

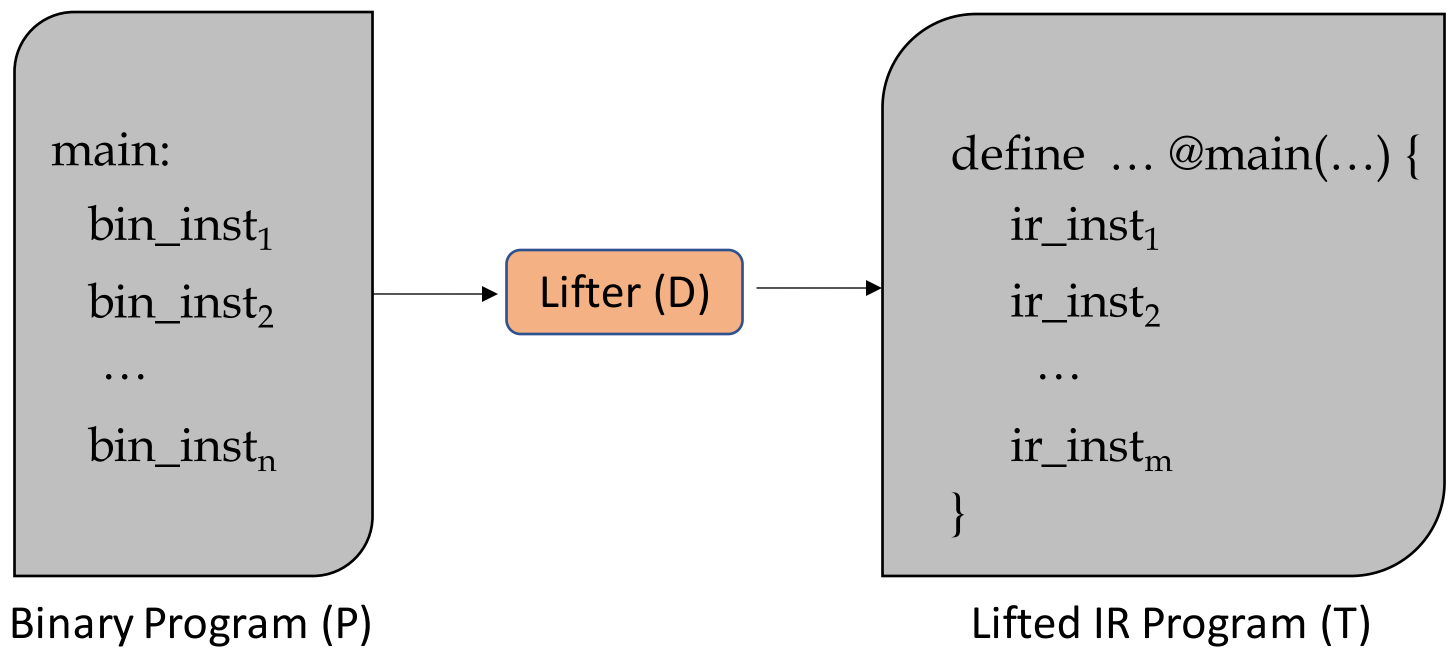
A Few Potential Applications

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

Lifter Validation: Our Approach

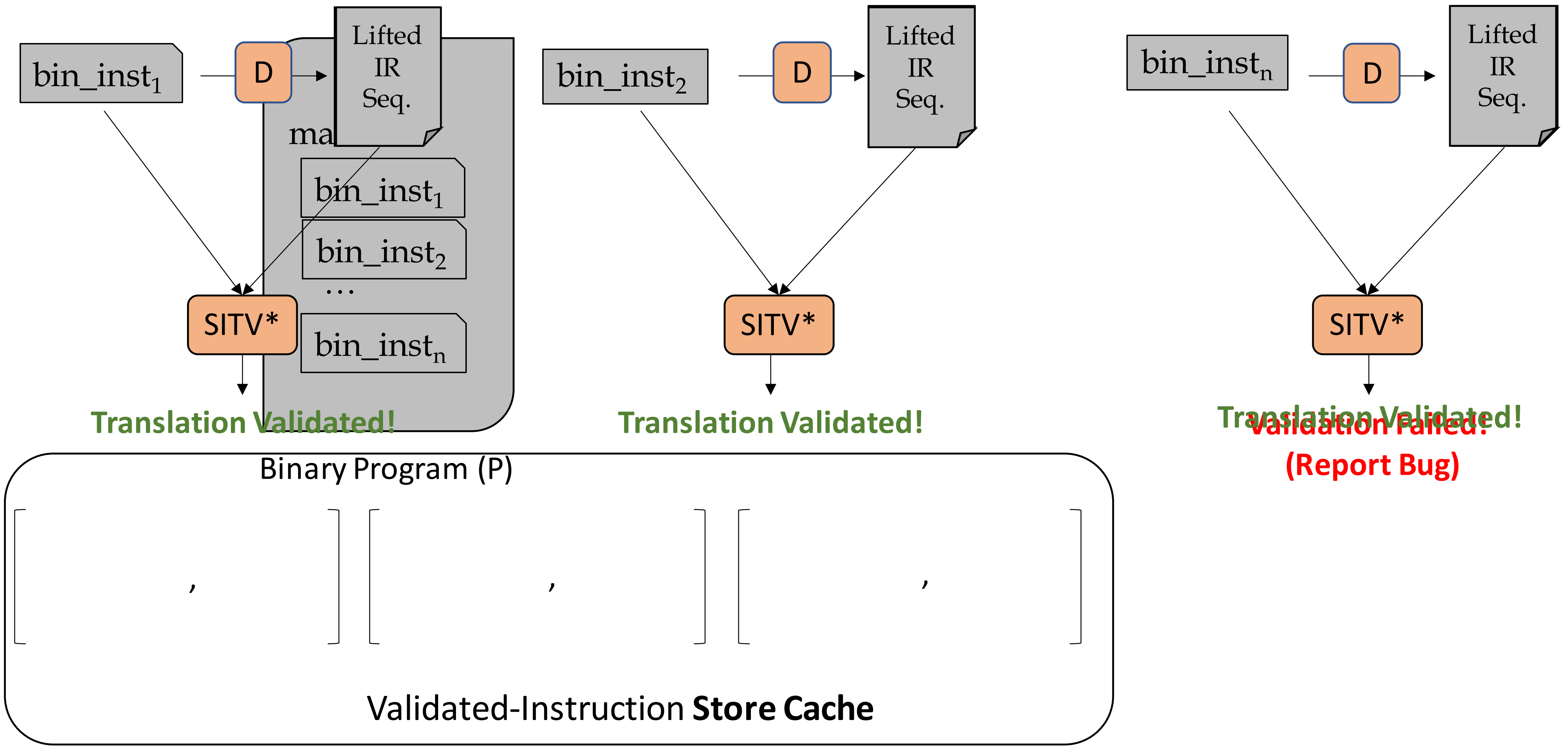
- **Phase I** **S**ingle-**I**nstruction **T**ranslation-**V**alidation (SITV)
- **Phase II** **P**rogram-**l**evel **V**alidation (PLV)

Overall Goal

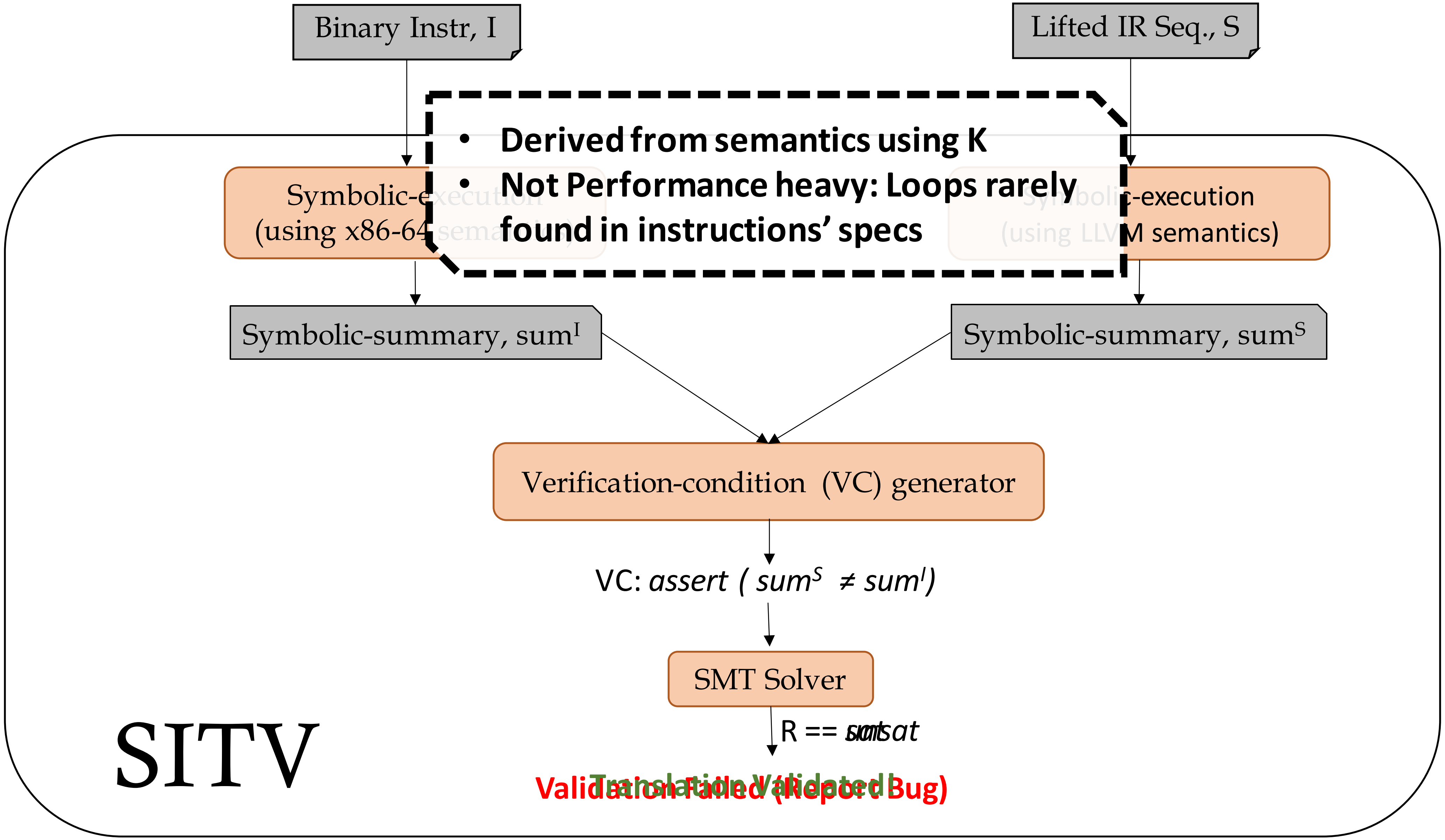


Our goal is to validate the translation from P to T

Single-Instruction Translation Validation



*SITV: Single Instruction Translation Validation Framework


Binary Instr, I
Lifted IR Seq., S
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* / *sat*
Translation Validated!
Validation Failed (Report Bug)
SITV

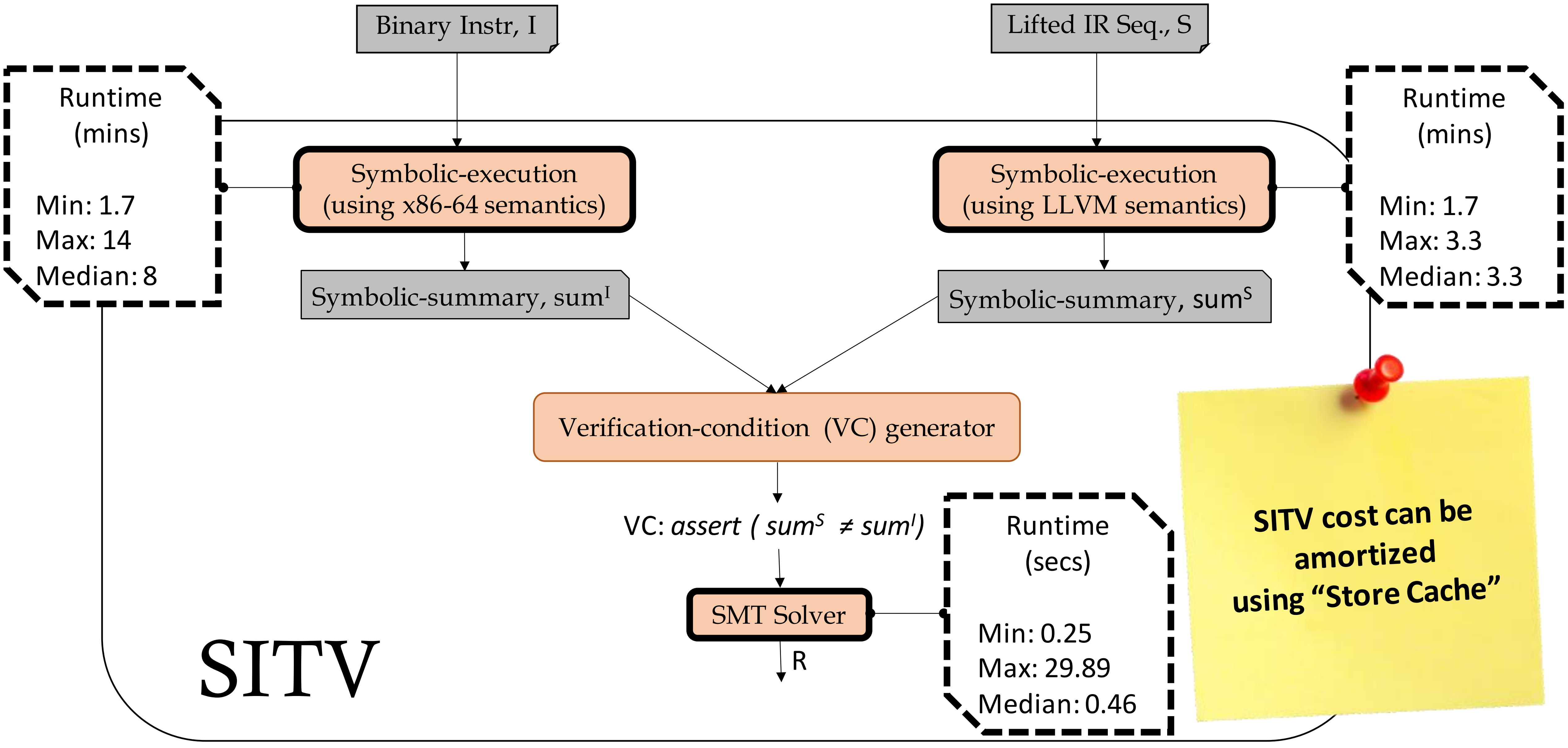
- **Derived from semantics using K**
- **Not Performance heavy: Loops rarely found in instructions' specs**

SITV: Evaluation Setup

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

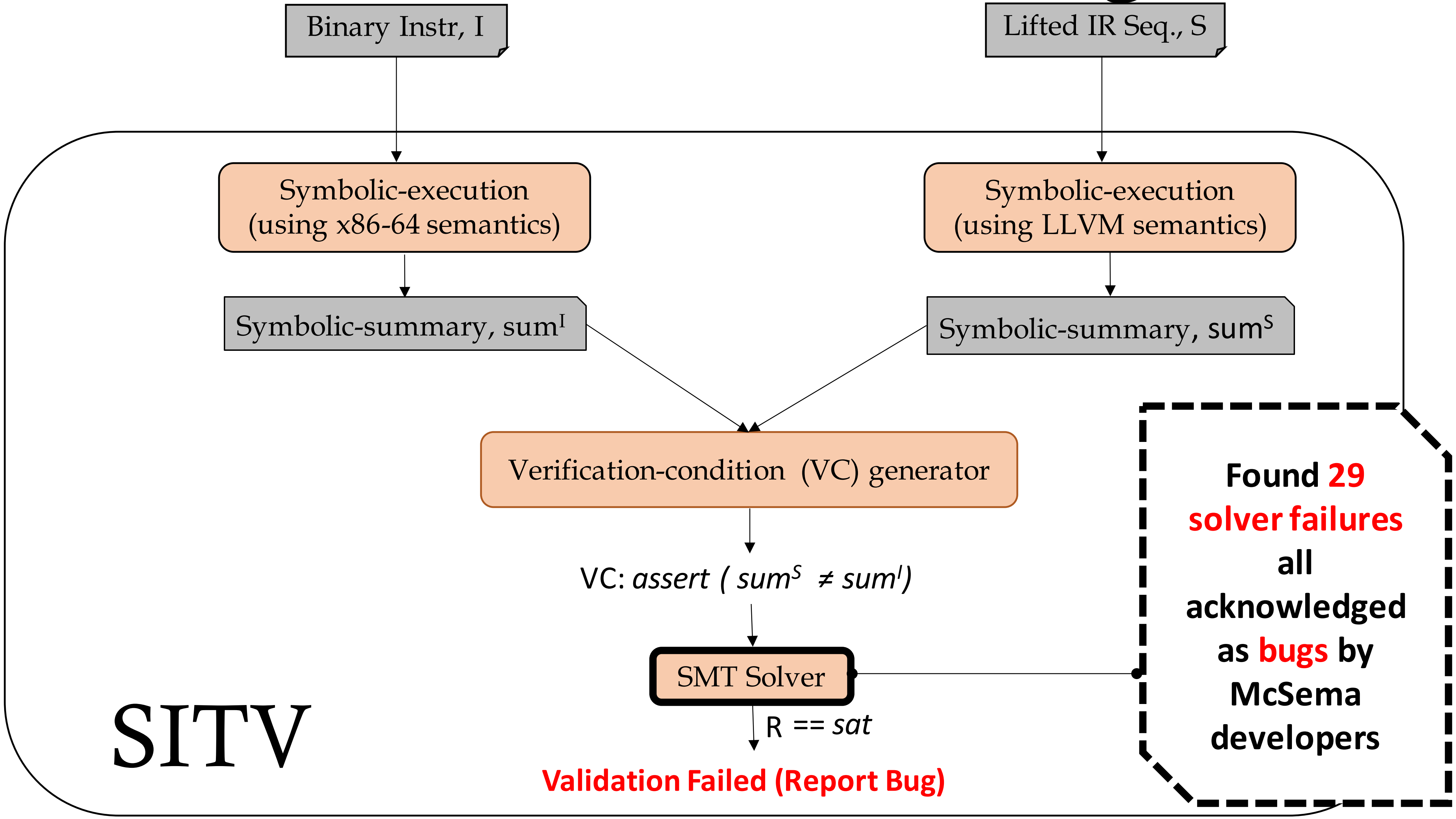
- ❑ McSema supports 1922 variants; all supported by our ISA model
- ❑ Exclude 573 because of limitations of LLVM semantics e.g., unsupported vector or FP types, intrinsic functions
- ❑ Solver runtime: min - 0.25 s, max – 29.89 s, median –

SITV: Performance



SITV cost can be amortized using "Store Cache"

SITV: Revealed Bugs



Binary Instr, I
Lifted IR Seq., S
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 ($sum^S \neq sum^I$)*
SMT Solver
R == *sat*
Validation Failed (Report Bug)
SITV

Found 29 solver failures all acknowledged as bugs by McSema developers

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]

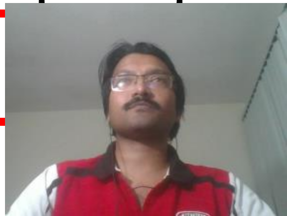
(2) DEST[127:64] ← DEST[63:32] * SRC[63:32]

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

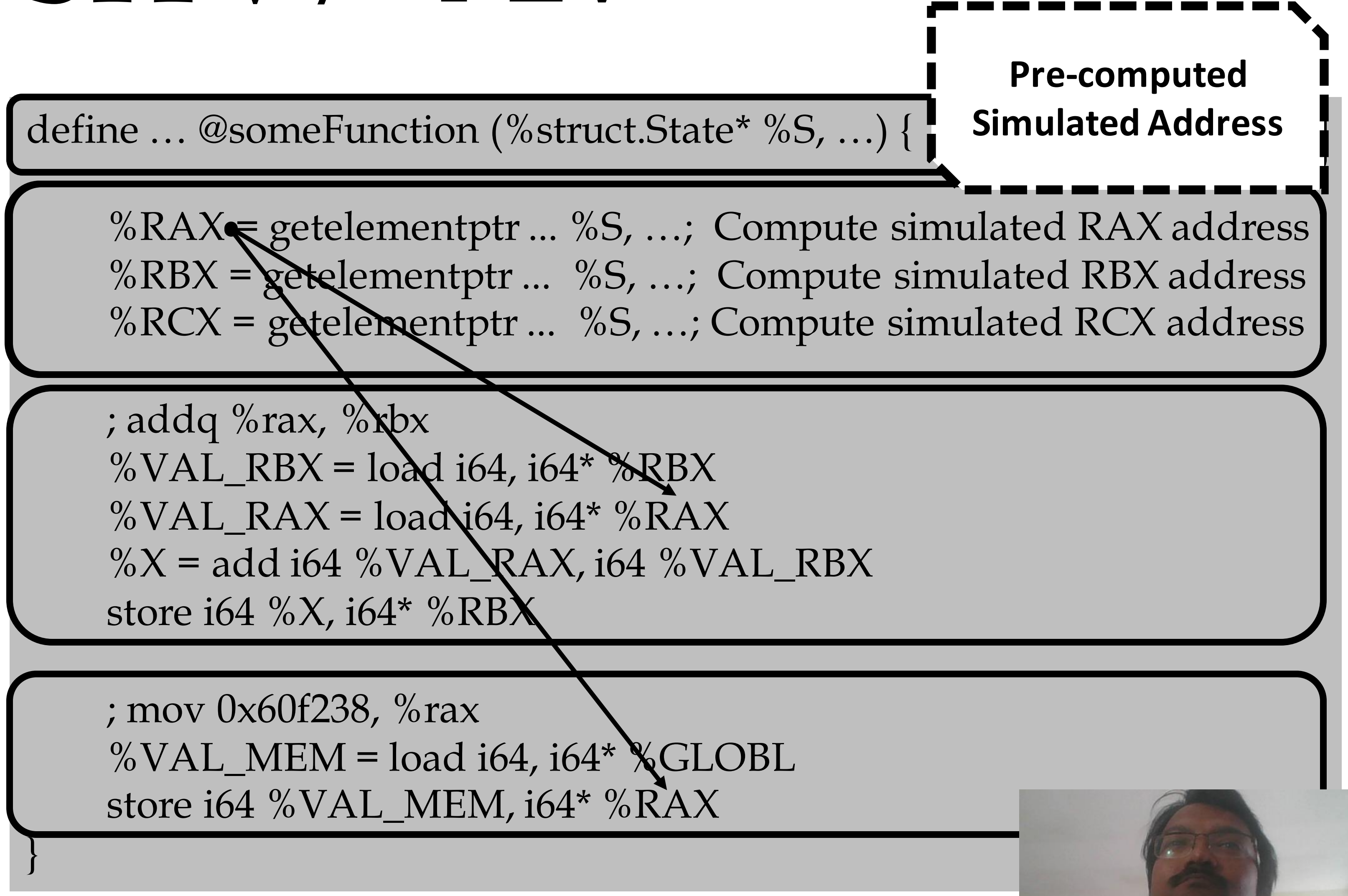
- **Phase I** **S**ingle-**I**nstruction **T**ranslation-**V**alidation (SITV)

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

SITV$\nRightarrow$ 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
    %VAL_MEM = load i64, i64* %GLOBL
    store i64 %VAL_MEM, i64* %RAX
}
```

Lifted IR Program, T

SITV $\not\Rightarrow$ 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 $\not\Rightarrow$ 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

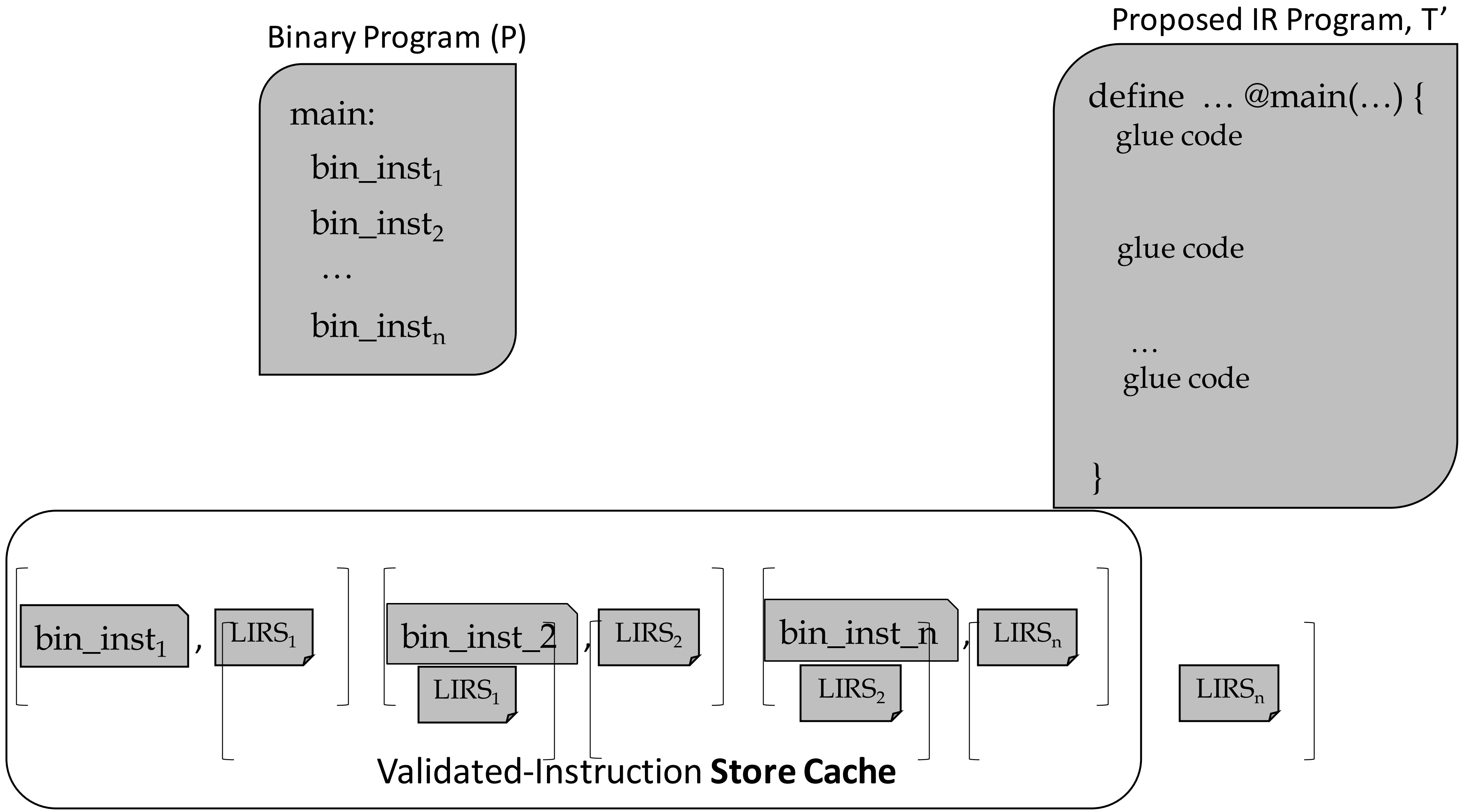
To propose an alternate reference program, T’, generated by carefully stitching the validated lifted IR sequences (using SITV)

Transformation & Matching

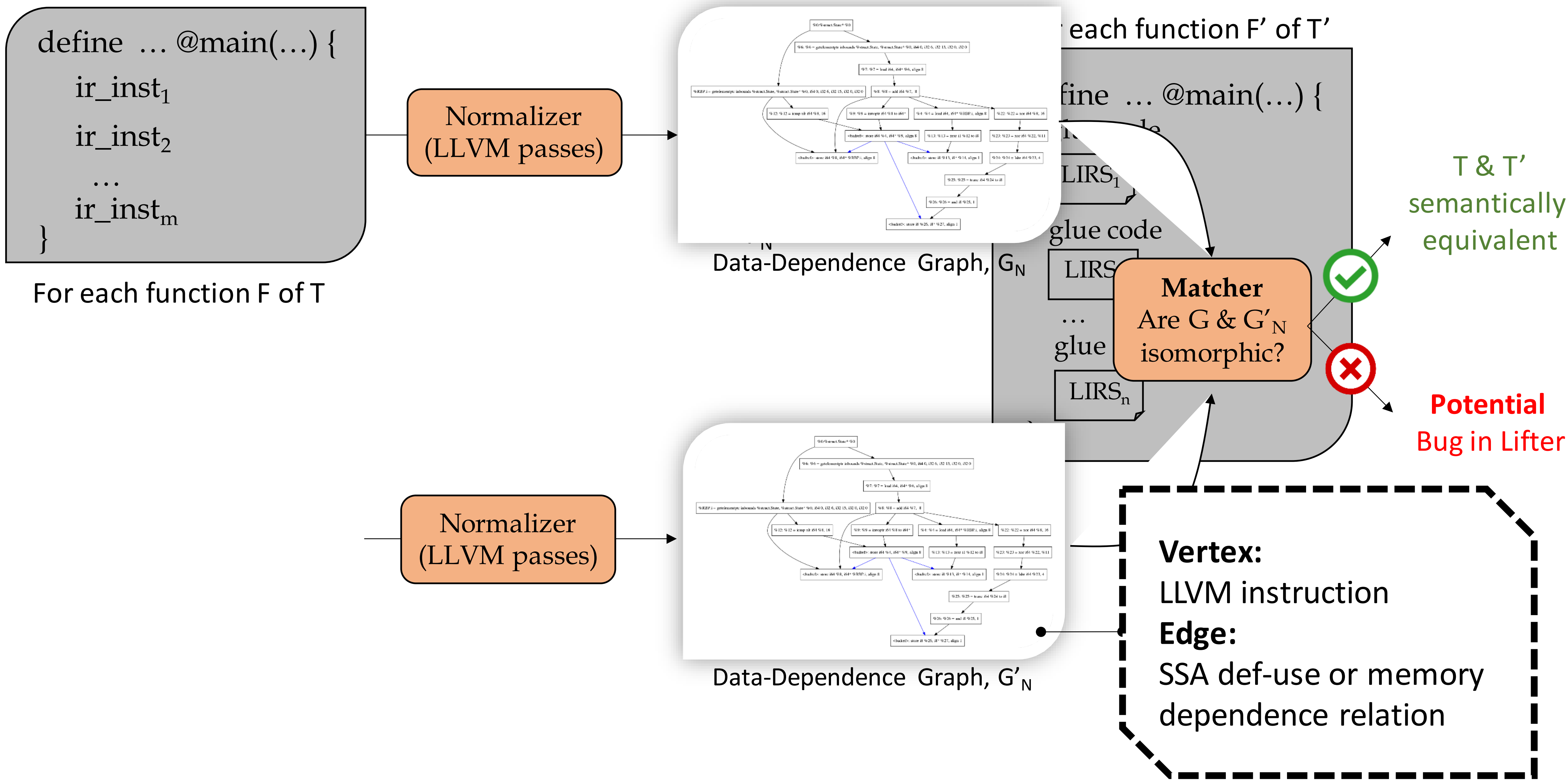
- ❑ **Transformation:** Uses semantic preserving transformations to reduce T’ and original lifted program (T) to a common form

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

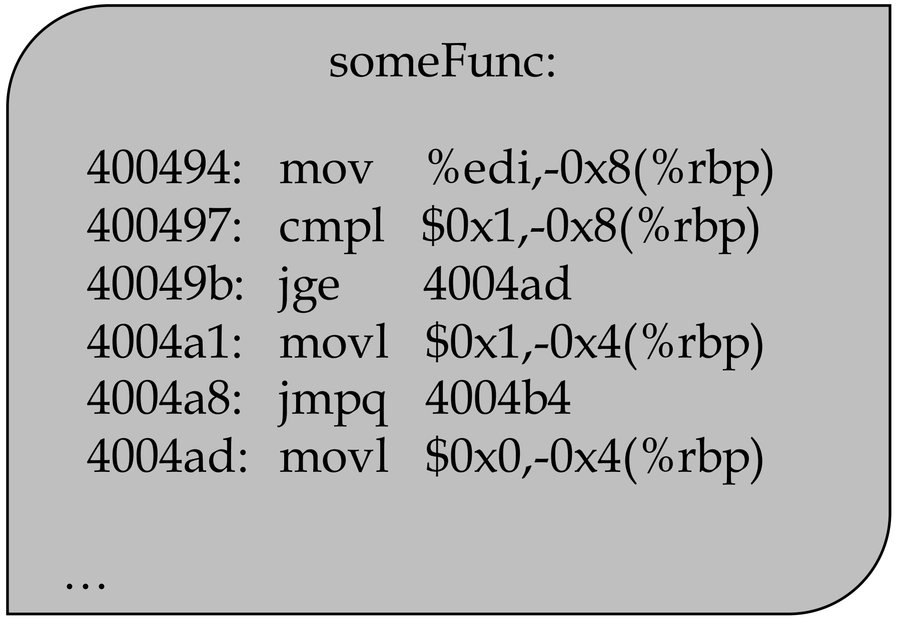
PLV: Compositional Lifting



PLV: Normalization & Matching



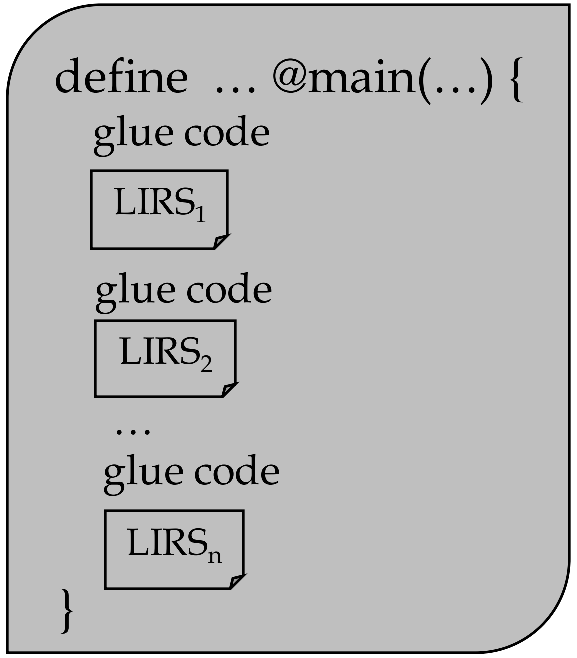
PLV: Extra Diagram



```
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



```
define  … @main(…) {
  glue code
  LIRS_1
  glue code
  LIRS_2
  …
  glue code
  LIRS_n
}
```

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′
```

Could be done offline !

SITV

SITV

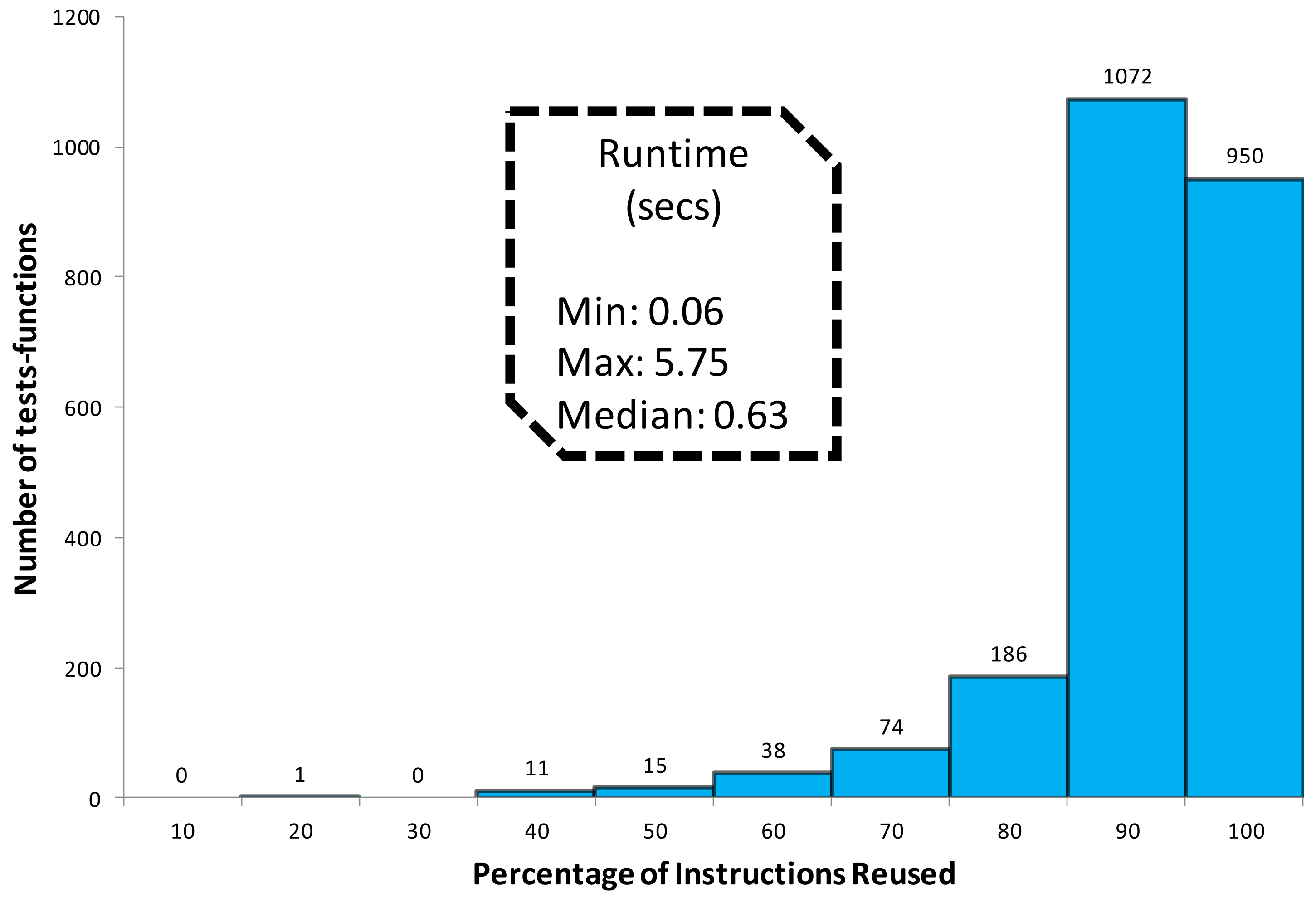
Very simple in design, however, PLV can be done prior to SITV

No formal guarantee that T′ is the correct translation of P

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



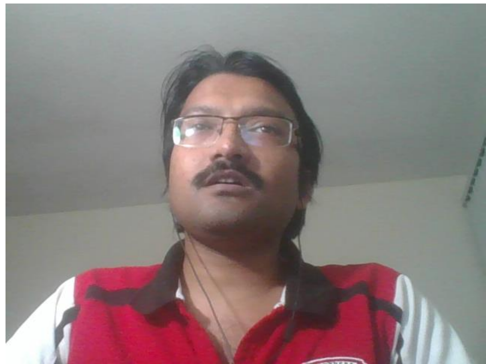
Normalizer

- ❑ 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

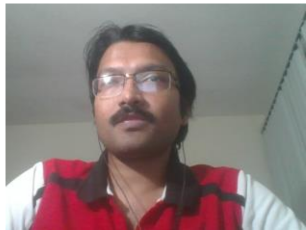
- **Search Space:** Includes passes from the 17-length pass sequence

- **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

- ~~Matcher soundness ??~~
- ~~Compd not formalized?~~
- ~~Matcher False negatives? More structer graph iso?~~
- Control flow violations?

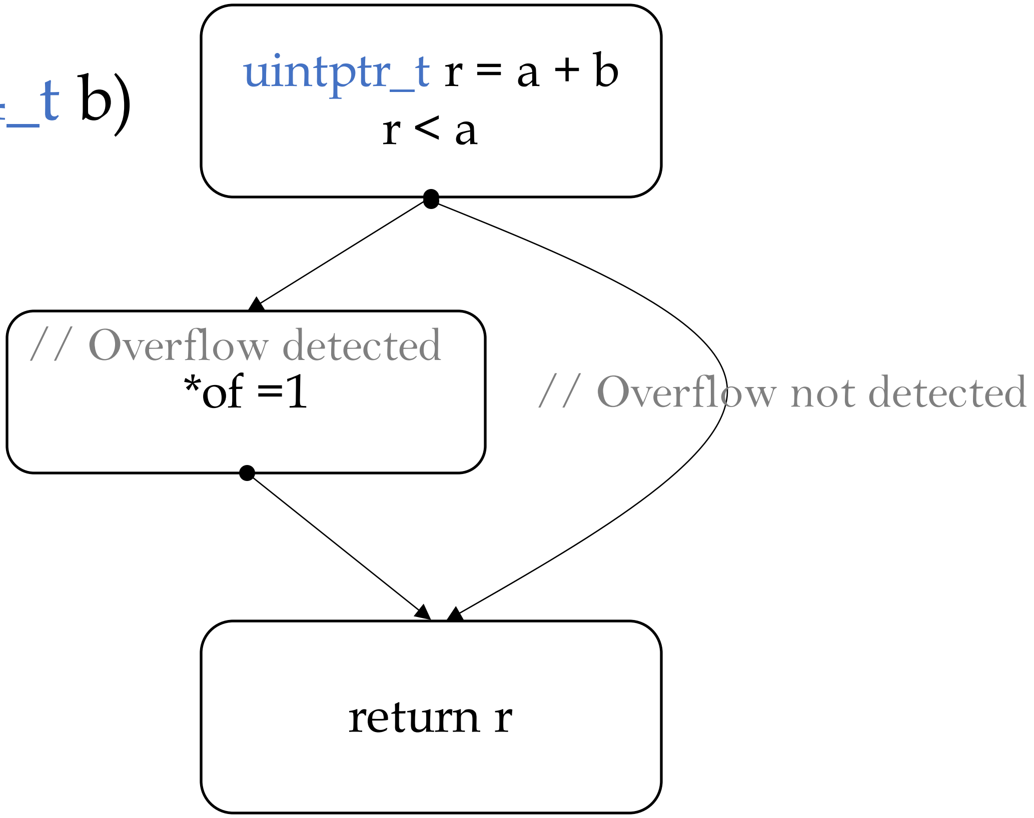
- ~~Examples of Binary Analysis?~~
- ~~Different ways of doing binary analysis?~~
- ~~Detail about cmpxchg? Why max time?~~

- ~~Halide Paper?~~
- Add more Appl, Bug in x86-64 sema, SIV Bugs

- Outline
- Shake

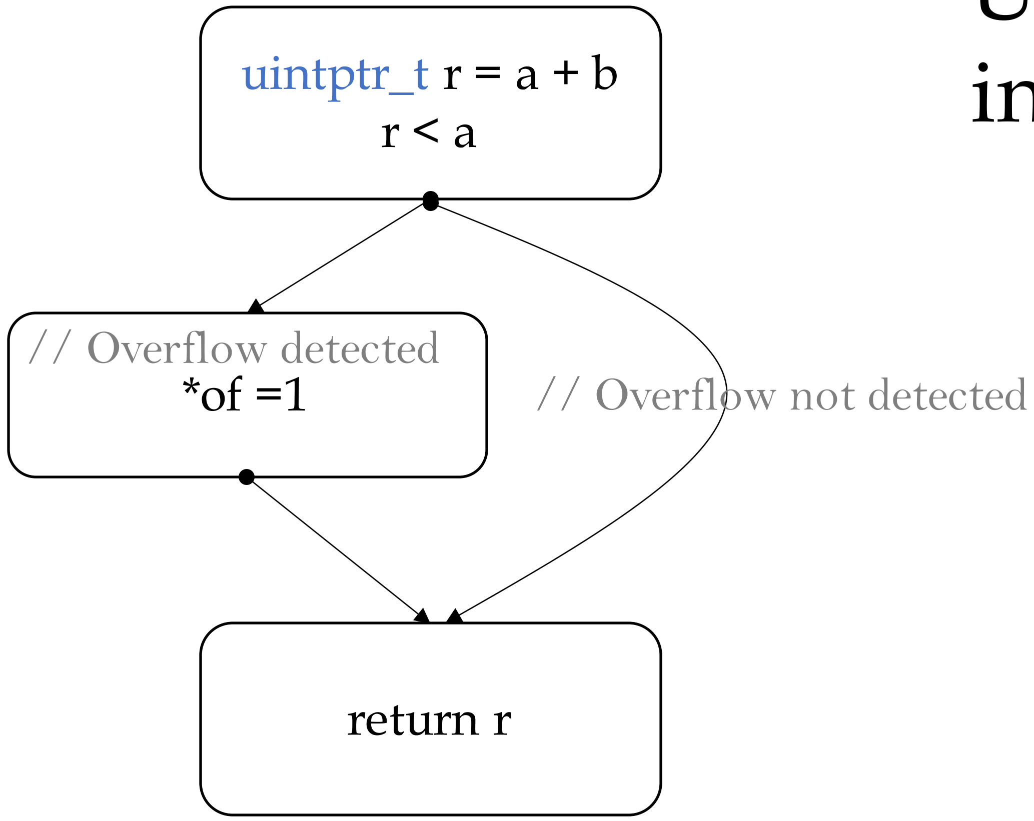
Security Vulnerability Tacking

```
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 ;
}
```



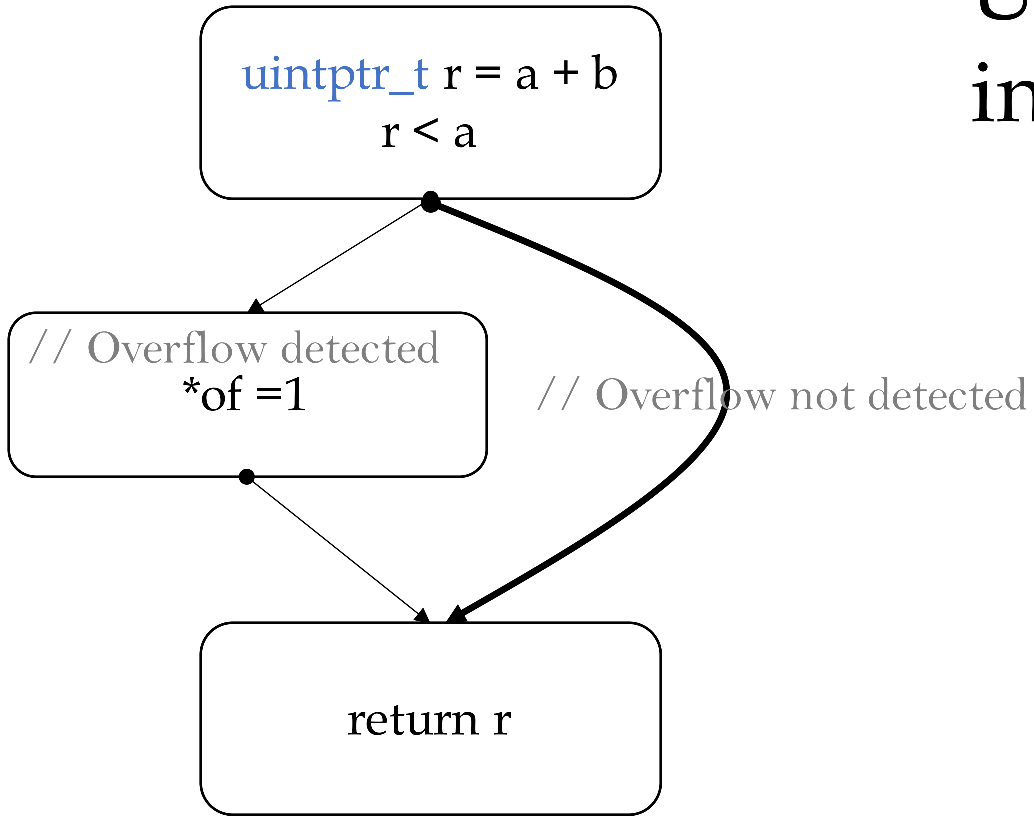
Code snippet borrowed from HiStar kernel^OSDI'06^, in which KLEE^OSDI'08^ found a security vulnerability

Security Vulnerability Tacking



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

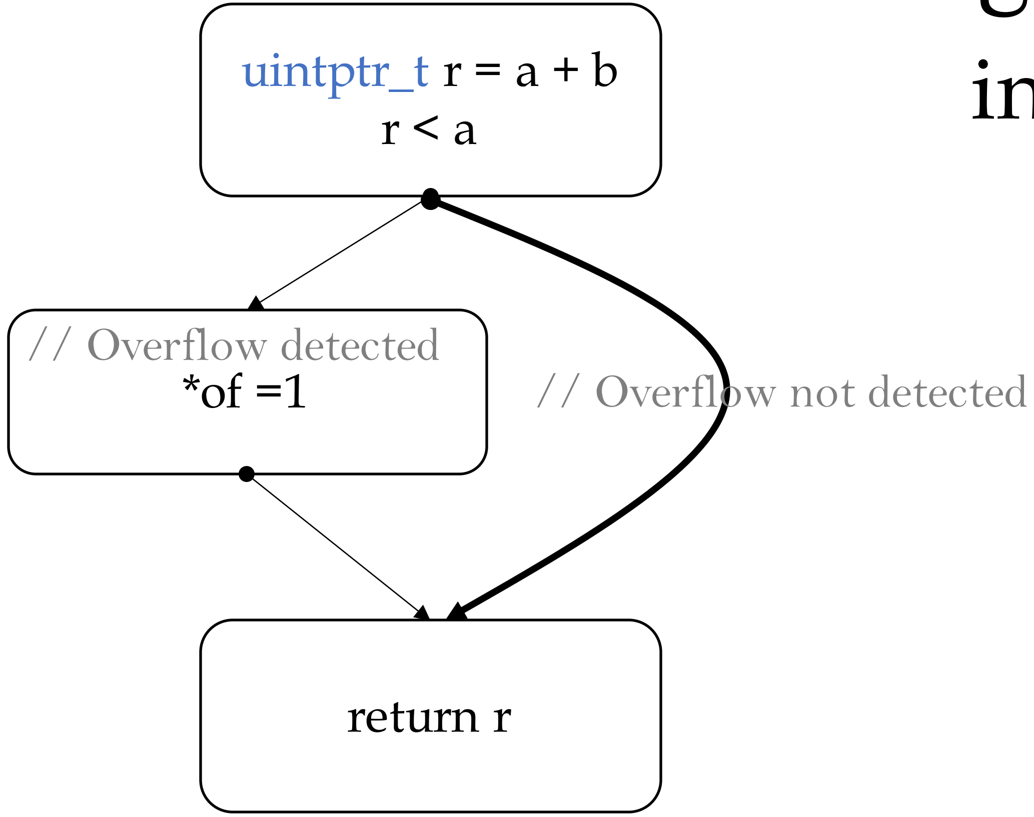
Security Vulnerability Tacking



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

Limitations

- ❑ **Incomplete LLVM Semantics**
- ❑ **Normalizer not formally-verified**

Simplification Rules

- BV[I:J] ∘ BV[J:K] → BV[I:K]
- BV[0 : bitwidth(BV)-1] → BV
- (BV1[0:63] ∘ BV2[0:63])[0:31] → BV2[0:31]
- (BV1[0:63] ∘ BV2[0:63])[64:96] → BV1[0:31]
- (BV1[0:63] ∘ BV2[0:63])[32:96] → (BV1[0:31] ∘ BV2[32:63])
- (BV[32:63])[0:8] → BV[32:39]
- (BV1 boolOp BV2)[I:J] → BV1[I:J] & BV2[I:J]
- (cond ? BV1:BV2)[I:J] → (cond ? BV1[I:J]:BV2[I:J])
- BV ∘ (cond ? BV2 : BV3) → (cond ? BV ∘ BV1: BV ∘ BV2)
- (cond ? BV1 : BV2) binOp (cond ? BV3 : BV4) → (cond ? BV1 binOp BV3 : BV2 binOp BV4)
- add_double(A, 0) → A if MSB of A is 0 /* To avoid A being -0.0 */

Sound transpilation from binary to machine-independent code, Roberto et al. SBMF 2017

- Formally modeled BIL IR in the interactive theorem prover HOL4
- Implemented a verified transpiler for ARMv8 programs to BIL IR
- Handling other machine architectures (e.g. x86, x64 MIPS) require developing new transpilers
- Verified transpilation of an instruction takes ~ 9 s
- Evaluated on few examples; Biggest ones are
  - bignum library function with 141 Arm instructions → 907 lines of BIL
  - AES functions with 535 Arm instructions → 3920 lines of BIL

Why LLVM is Pervasively as Lifted IR

- ❑ Mature symbolic Analysis tools like KLEE
- ❑ Industry standard optimization passes used for re-optimzation and re-targeting
- ❑ Clang LibTooling for efficient instrumentation
- ❑ Decompilers like llvm-mctoll* makes heavy use of LLVM compilation pileline

* Aaron Smith and S. Bharadwaj Yadavalli. 2018. LLVM Based Binary Raiser: llvm-mctoll

Matching Results: Spec2006

- Out of total 3870 functions, success rate is 60%
- Working on the 40% failure cases (potential false negatives)

* Aaron Smith and S. Bharadwaj Yadavalli. 2018. LLVM Based Binary Raiser: llvm-mctoll

SITV: Jump

conc_pc: jz 8
(conc_pc is the concrete value of PC for current instruction in isolation)

X86-64 Semantics
RIP = conc_pc
RIP summary: SYMX_ZF ==1 ? conc_pc + 8 : conc_pc + 2

LLVM Semantics
PC = conc_pc
PC summary: SYML_ZF ==1? conc_pc + 8 : conc_pc + 2
Current Block summary: SYML_ZF == 1 ? conc_pc + 8 : conc_pc + 2

Equivalence checks
Precondition:
1. SYMX_ZF = SYML_ZF

Assert:
1. RIP summary = PC summary
2. RIP summary = Current Block summary

Compositional Lifting: Jump

During Compositional Lifting, the conc_pc value need to be fixed using the actual value of PC w.r.t the program context

SITV: Data Access Instructions

movq 0x602040, %

- ❑ During SITV, w/o full program context, we can only validate the fact whether the constant 0x602040 (which could potentially be an address) is correctly moved to the destination register.

- ❑ However, we tested both the behaviors.

Repeat Instruction Validation

- We symbolically executing those instruction with symbolic input state and comparing the summaries (using solver checks) of any single ith iteration of the two loops.

- Moreover, such loops are bounded by a constant thus must terminate.

- Equivalence check is preconditioned on the fact that the register or memory value, corresponding to the loop trip count, are asserted to be equivalent.

Simulation Testing Based Approaches

Path-exploration lifting: Hi-fi tests for Lo-fi emulators, ASPLOS'12
by Martignoni et al.

- ❑ Symbolic execution of a *Hi-Fi emulator* to generate test-cases to validate a *Lo-Fi emulator*
- ❑ Hardware co-simulation testing of Lo-Fi emulator using generated test-cases
- ❑ Tested single instruction as opposed to multiple-instruction sequences

Formal Method Based Approaches

Testing Intermediate Representations for Binary Analysis, ASE’17
by Kim et al.

- ❑ Differential testing of three binary lifters – BAP, BINSEC, and PyVEX
- ❑ Translated the respective IRs to a common representation to be compared using SAT solver
- ❑ Ignored instructions whose semantics are not “explicitly” exposed in IR
- ❑ Tested single instruction as opposed to multiple-instruction sequences

Formal Method Based Approaches

Towards verified binary raising, SpISA'19 *by John et al.*

- ❑ Validates the translation of basic-blocks in isolation
- ❑ Assisted by various manually written annotations, which are prone to errors

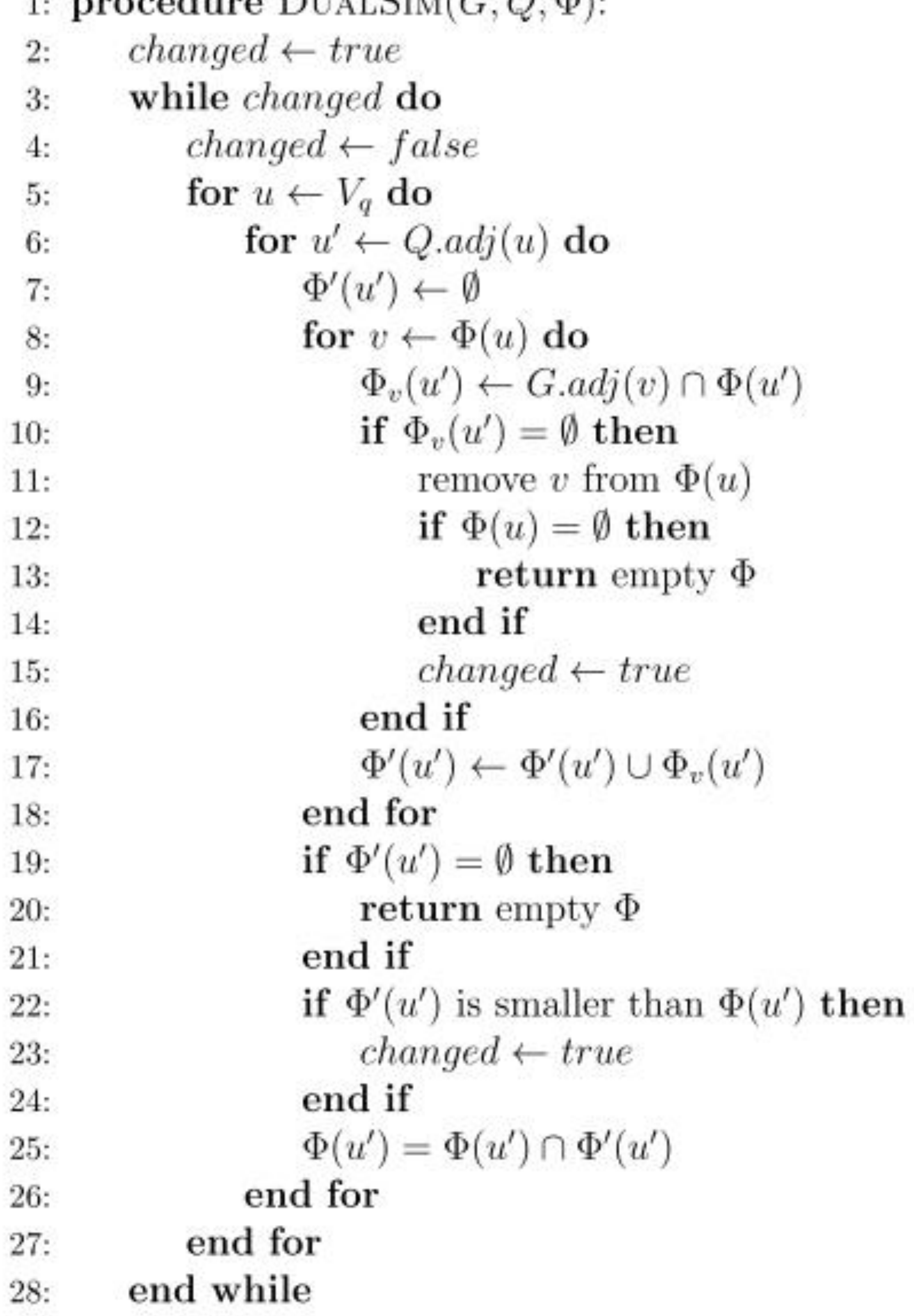
Matcher

Inputs: T: McSema-lifted IR.
T ′ : Compositional Lifter lifted IR.
Output: true → T & T ′ semantically equivalent
false → T & T ′ may-be non-equivalent

```
foreach corresponding function pair (F,F ′ ) in (T, T ′ ) do
        $F_N$ = Normalizer(F)
        $F'_N$ = Normalizer(F′)

        $G_N$ = DataDependenceGraph($F_N$)
        $G'_N$ = DataDependenceGraph($F'_N$)

        if not IsGraphIsomorphic*($G_N$, $G'_N$)
                // A potential bug in McSema while lifting
                return false
        end
end
return true
```

Subgraph Isomorphism

*A graph is **subgraph isomorphi** another if the first graph matcl subgraph of the second structu semantically.*



```
procedure DUALSIM($G, Q, \Phi$):
    $changed \leftarrow true$
    while $changed$ do
        $changed \leftarrow false$
        for $u \leftarrow V_q$ do
            for $u' \leftarrow Q.adj(u)$ do
                $\Phi'(u') \leftarrow \emptyset$
                for $v \leftarrow \Phi(u)$ do
                    $\Phi_v(u') \leftarrow G.adj(v) \cap \Phi(u')$
                    if $\Phi_v(u') = \emptyset$ then
                        remove $v$ from $\Phi(u)$
                        if $\Phi(u) = \emptyset$ then
                            return empty $\Phi$
                        end if
                        $changed \leftarrow true$
                    end if
                    $\Phi'(u') \leftarrow \Phi'(u') \cup \Phi_v(u')$
                end for
                if $\Phi'(u') = \emptyset$ then
                    return empty $\Phi$
                end if
                if $\Phi'(u')$ is smaller than $\Phi(u')$ then
                    $changed \leftarrow true$
                end if
                $\Phi(u') = \Phi(u') \cap \Phi'(u')$
            end for
        end for
    end while
```

Co-inductive Reasoning

```
int s = 0; int n = N;
while (n > 0) { s = s + n; n = n - 1; }
return s;
```

Spec1:- Main Configuration

Pre-condition:

1. $s = 0$
2. $n = N$
3. $0 \leq N < 2^{32}$
4. $0 \leq N(N+1)/2 < 2^{32}$

Post-condition:

1. $s = N(N+1)/2$
2. $n = 0$

Spec2:- Loop Invariant

Pre-condition:

1. $s = B$
2. $n = A$
3. $0 \leq A < 2^{32}$
4. $0 \leq B < 2^{32}$
5. $0 \leq B + A(A+1)/2 < 2^{32}$

Post-condition:

1. $s = B + A(A+1)/2$
2. $n = 0$

Proving Spec1 assuming Spec2 is *met*

Prove Steps

1. Start with s = 0 and n = N s.t. $0 \leq N < 2^{32}$
2. Sym-exec till loop header L1
3. Summary till that point: n = N & s = 0 ⇒ **PREC2** as PREC1 is true.
4. Using induction, s = 0 + N(N+1)/2 and n = 0 ⇒ **POSTC1**
5. **Spec1 is met**

Spec1:- Main Configuration (To Be Proved)

Pre-condition (**PREC1**):
1. s = 0
2. n = N
3. $0 \leq N < 2^{32}$
4. $0 \leq N(N+1)/2 < 2^{32}$

Post-condition (**POSTC1**):
1. s = N(N+1)/2
2. n = 0

Code

```
int s = 0; int n = N;
L1: while (n > 0) {
            s = s + n;
            n = n - 1;
            }
L:
return s;
```

Spec2:- Loop Invariant (Assumed True)

Pre-condition (**PREC2**):
1. s = B
2. n = A
3. $0 \leq A < 2^{32}$
4. $0 \leq B < 2^{32}$
5. $0 \leq B + A(A+1)/2 < 2^{32}$

Post-condition:
1. s = B + A(A+1)/2
2. n = 0

Proving Spec2 assuming Spec1 is *met*

Steps

1. Start with s = B and n = A s.t. $0 \leq A,B < 2^{32}$ & $0 \leq B + A(A+1)/2 < 2^{32}$
2. If $A \leq 0$
   1. A = 0 (since $0 \leq A,B < 2^{32}$)
   2. Sym-ex till L gives n = 0 and s = B ⇒ **POSTC2**
3. If A > 0
   1. sym-exec one-loop iteration till L1 gives n′ = A-1 and s′ = B + A ⇒ **PREC2** as A > 0 ⇒ $A – 1 \geq 0$ and $0 \leq B + A(A+1)/2 < 2^{32}$ ⇒ $0 \leq B + A < 2^{32}$
   2. Using Induction: n′′ = B+A + (A-1)A/2 = B+A(A+1)/2 and n′′ = 0 ⇒ **POSTC2**
4. **Spec2 is met**

Spec1:- Main Configuration (Assumed True)

Pre-condition:
1. s = 0
2. n = N
3. $0 \leq N < 2^{32}$
4. $0 \leq N(N+1)/2 < 2^{32}$

Post-condition (**POSTC1**):
1. s = N(N+1)/2
2. n = 0

```
int s = 0; int n = N;
L1: while (n > 0) {
        s = s + n;
        n = n - 1;
        }
L:
return s;
```

Spec2:- Loop Invariant (To be Proved)

Pre-condition (**PREC2**):
1. s = B
2. n = A
3. $0 \leq A < 2^{32}$
4. $0 \leq B < 2^{32}$
5. $0 \leq B + A(A+1)/2 < 2^{32}$

Post-condition (**POSTC2**):
1. s = B + A(A+1)/2
2. n = 0

Extension to Other Lifters

Can be extended to other lifters, provided

1. Formal semantics of ISA and target language are available.
2. Target language amenable to normalization using semantics-preserving transformations.

Engineering effort

- ❑ SITV can be applied as is as long as (1) is satisfied
- ❑ PLV has three components
  - ▪ Compositional Lifting: The “glue code” that the Compositional lifter uses for lifting is specific the lifter (under test) and hence need to be discovered for each new lifter.
  - ▪ Normalization: Needs (2) to be satisfied
  - ▪ Matcher: Can be applied as is

Future Work

- ❑ **Formally verifying Normalizer**
- ❑ **Efficient matching**
  - ▪ e.g., based on iteratively pruning the matched sub-graphs and look for more isomorphic matches after normalizing the residual graph
- ❑ **Efficient Autotuning**

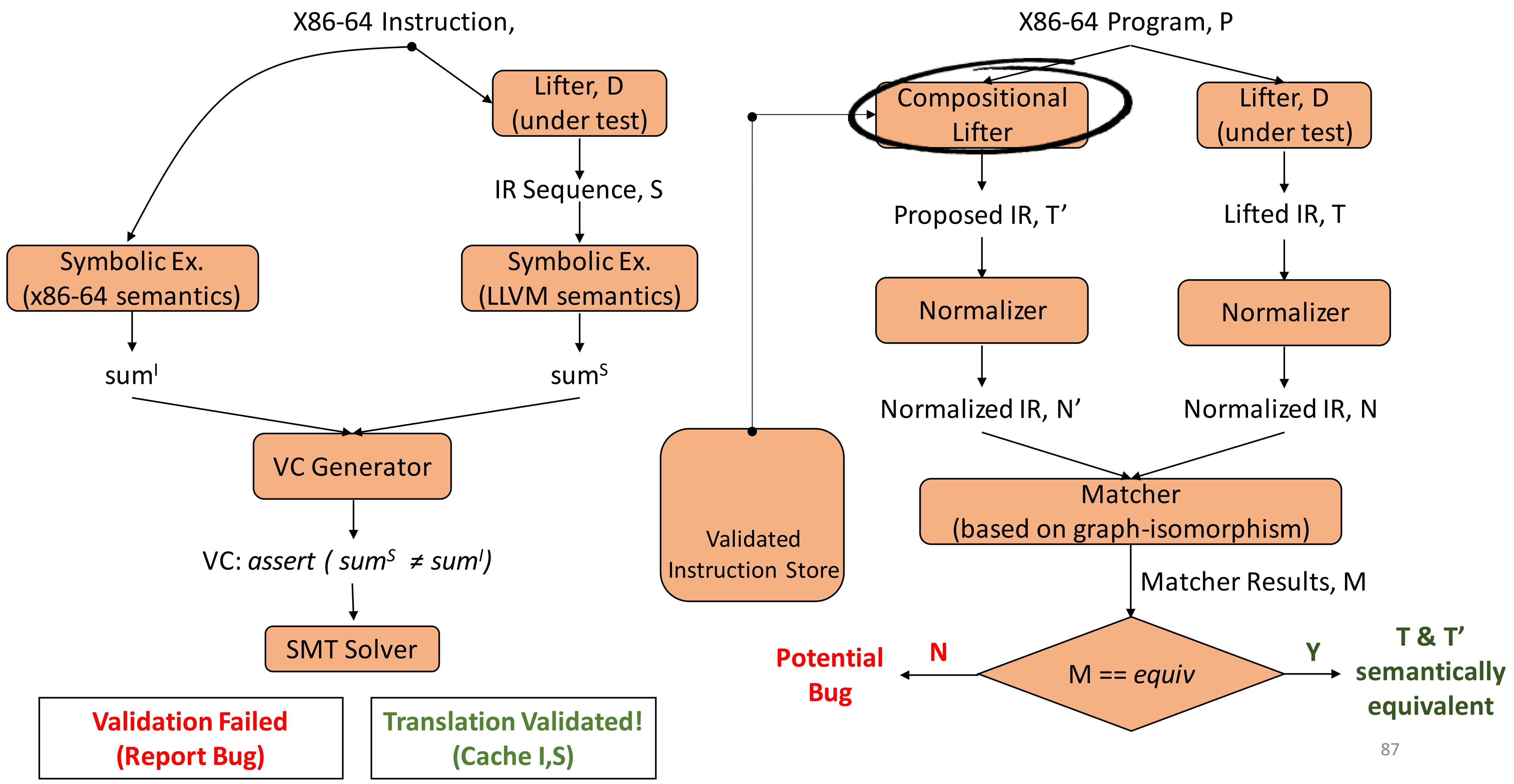
SITV$\nRightarrow$ PLV

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

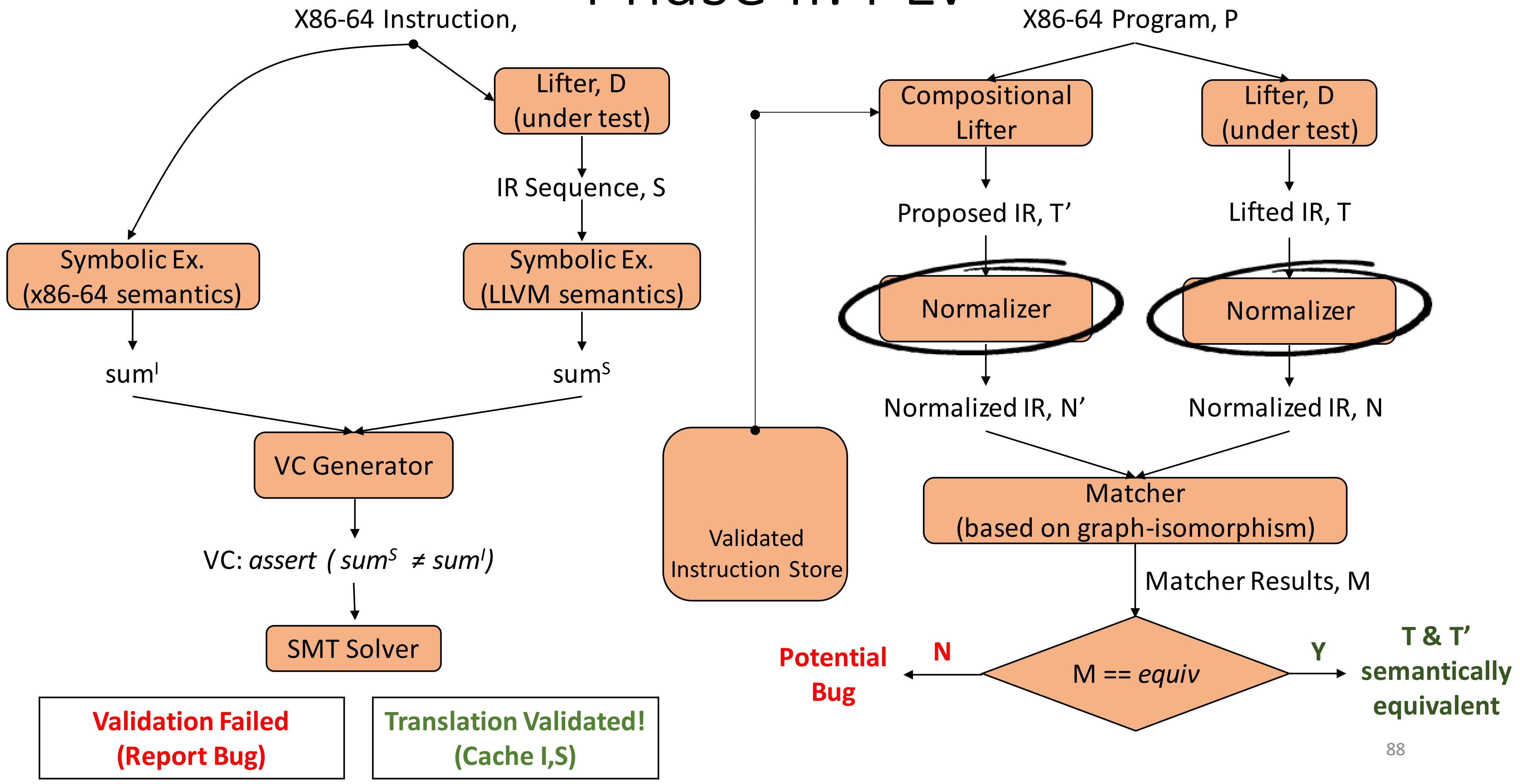
Binary Program (P)

```
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

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

    ; 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
}
```

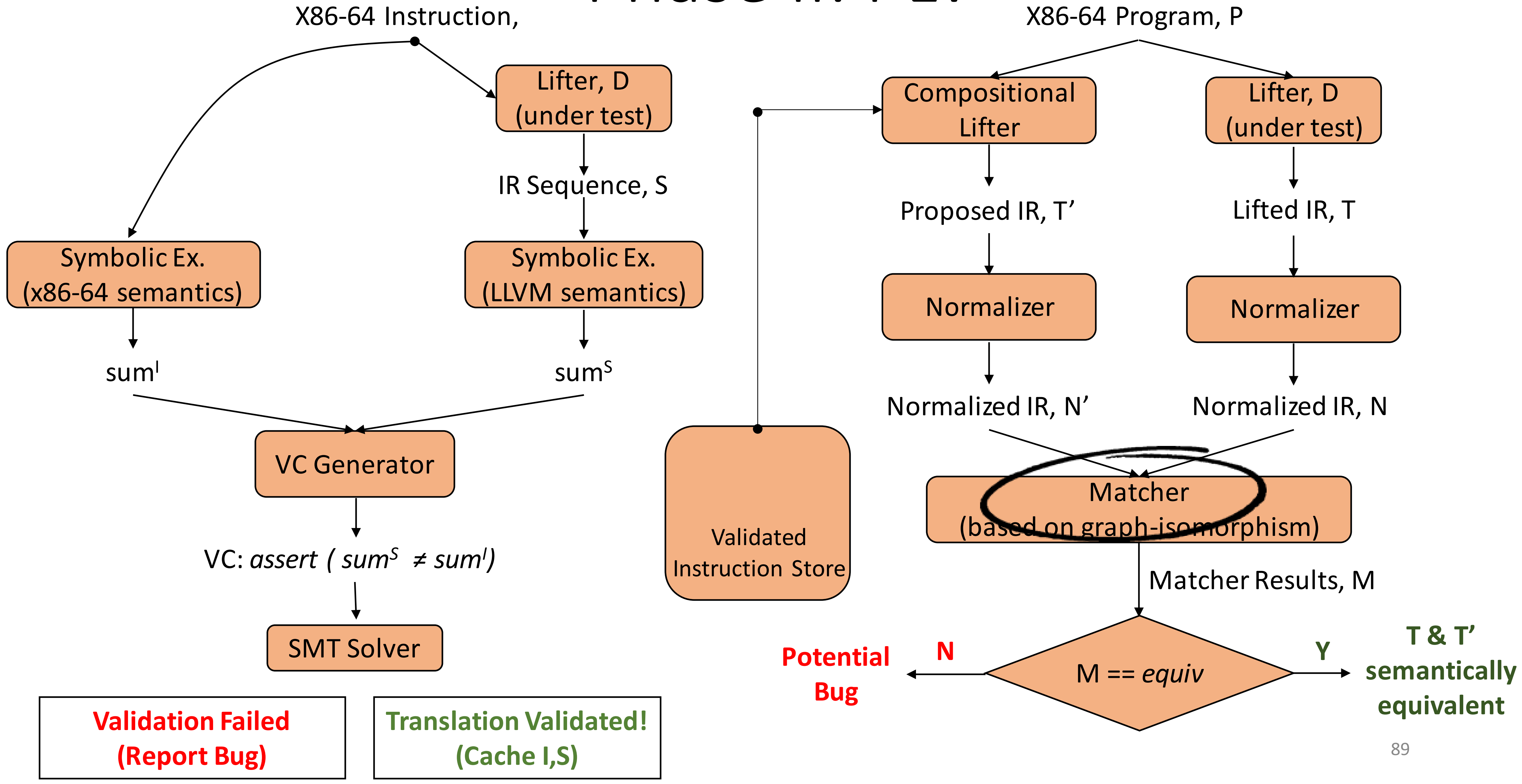
Lifted IR Program, T


X86-64 Instruction,
Lifter, D (under test)
IR Sequence, S
Symbolic Ex. (x86-64 semantics)
Symbolic Ex. (LLVM semantics)
sum^I
sum^S
VC Generator
VC: assert (sum^S ≠ sum^I)
SMT Solver
Validation Failed (Report Bug)
Translation Validated! (Cache I,S)
Validated Instruction Store
X86-64 Program, P
Compositional Lifter
Lifter, D (under test)
Proposed IR, T'
Lifted IR, T
Normalizer
Normalizer
Normalized IR, N'
Normalized IR, N
Matcher (based on graph-isomorphism)
Matcher Results, M
M == equiv
N
Y
Potential Bug
T & T' semantically equivalent

Phase II: PLV

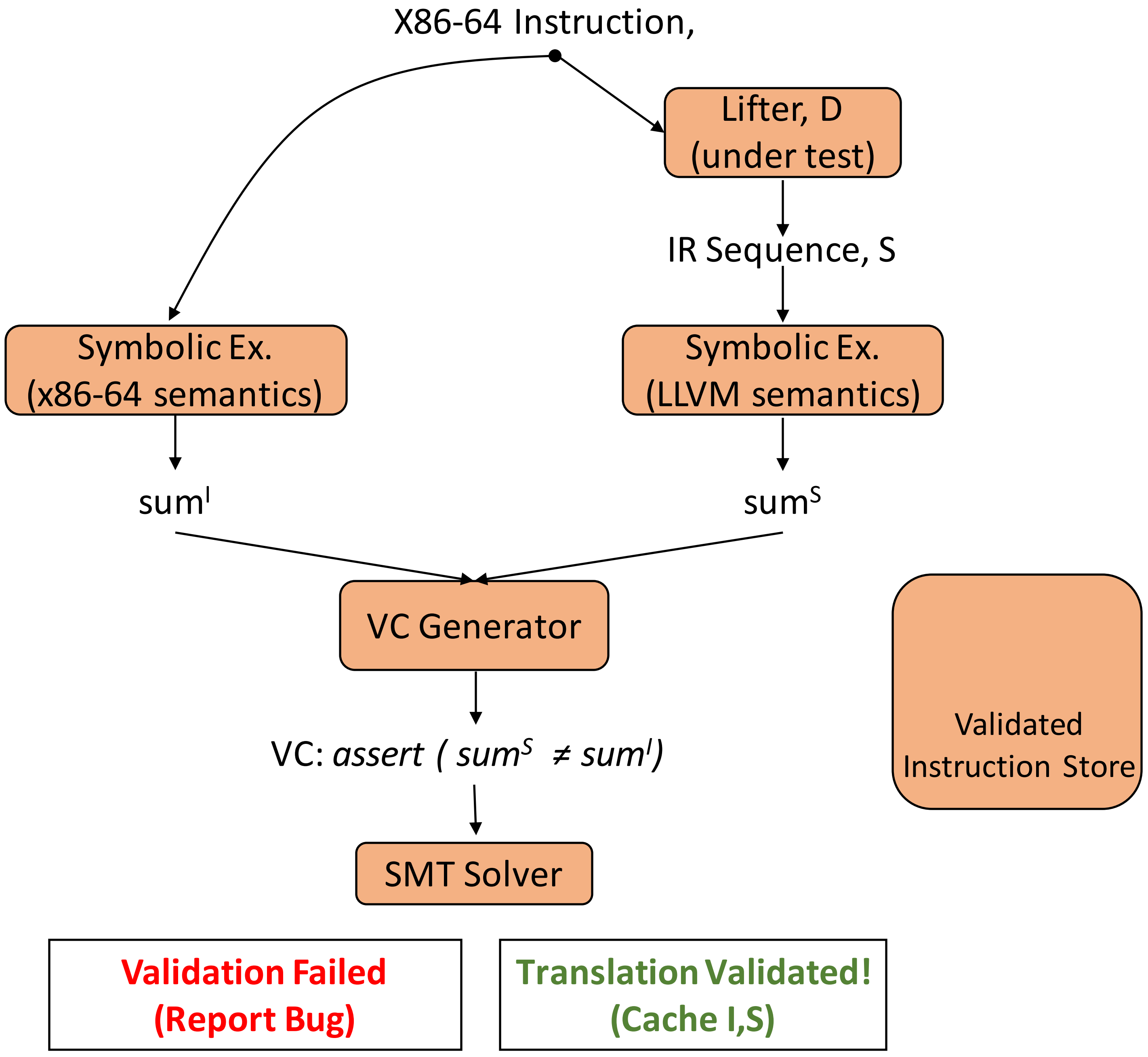

X86-64 Instruction,
Lifter, D (under test)
IR Sequence, S
Symbolic Ex. (x86-64 semantics)
Symbolic Ex. (LLVM semantics)
sum^I
sum^S
VC Generator
VC: assert (sum^S ≠ sum^I)
SMT Solver
Validation Failed (Report Bug)
Translation Validated! (Cache I,S)
Validated Instruction Store
X86-64 Program, P
Compositional Lifter
Lifter, D (under test)
Proposed IR, T'
Lifted IR, T
Normalizer
Normalizer
Normalized IR, N'
Normalized IR, N
Matcher (based on graph-isomorphism)
Matcher Results, M
M == equiv
N
Y
Potential Bug
T & T' semantically equivalent

Phase II: PLV



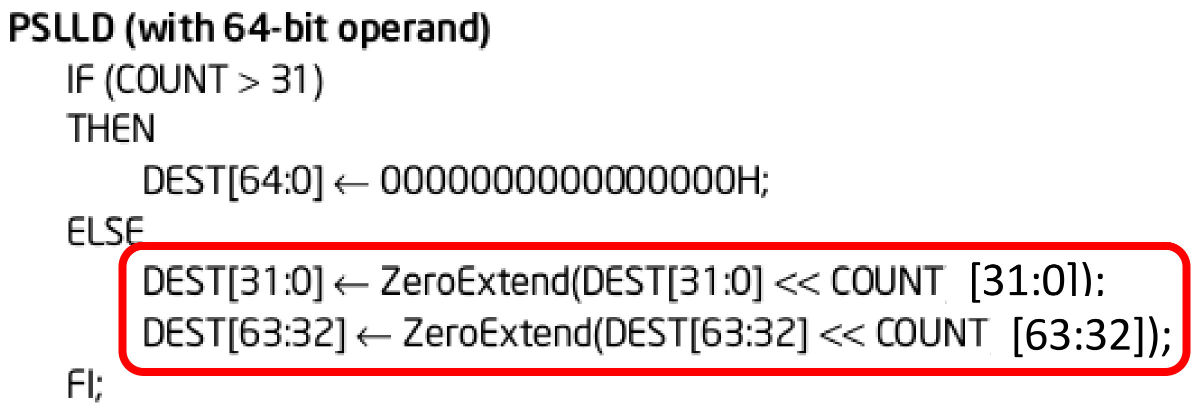
X86-64 Instruction,
Lifter, D (under test)
IR Sequence, S
Symbolic Ex. (x86-64 semantics)
Symbolic Ex. (LLVM semantics)
sum^I
sum^S
VC Generator
VC: assert (sum^S ≠ sum^I)
SMT Solver
Validation Failed (Report Bug)
Translation Validated! (Cache I,S)
Validated Instruction Store
X86-64 Program, P
Compositional Lifter
Lifter, D (under test)
Proposed IR, T'
Lifted IR, T
Normalizer
Normalizer
Normalized IR, N'
Normalized IR, N
Matcher (based on graph-isomorphism)
Matcher Results, M
M == equiv
N
Potential Bug
Y
T & T' semantically equivalent

Phase I: SIV


X86-64 Instruction,
Lifter, D (under test)
IR Sequence, S
Symbolic Ex. (x86-64 semantics)
Symbolic Ex. (LLVM semantics)
sum^I
sum^S
VC Generator
VC: *assert ($sum^S \neq sum^I$)*
SMT Solver
Validated Instruction Store
Validation Failed (Report Bug)
Translation Validated! (Cache I,S)

A Few Reported Bugs

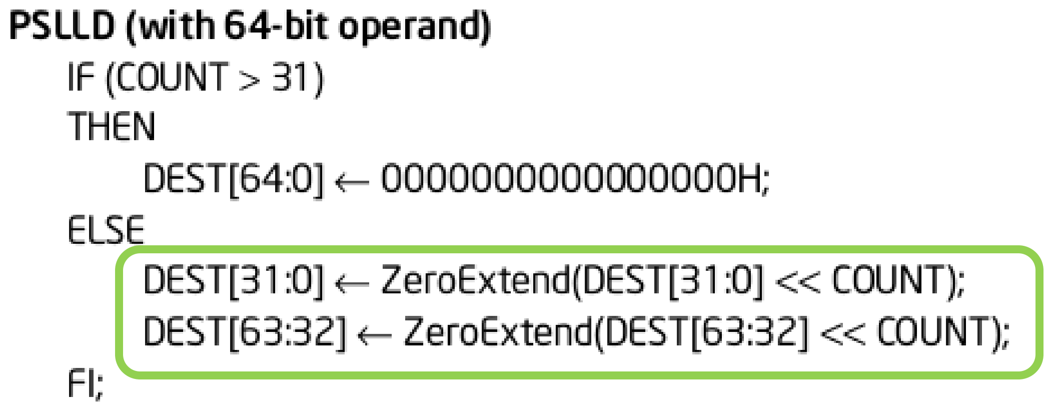
Stoke Implementation May 2018



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;
```

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