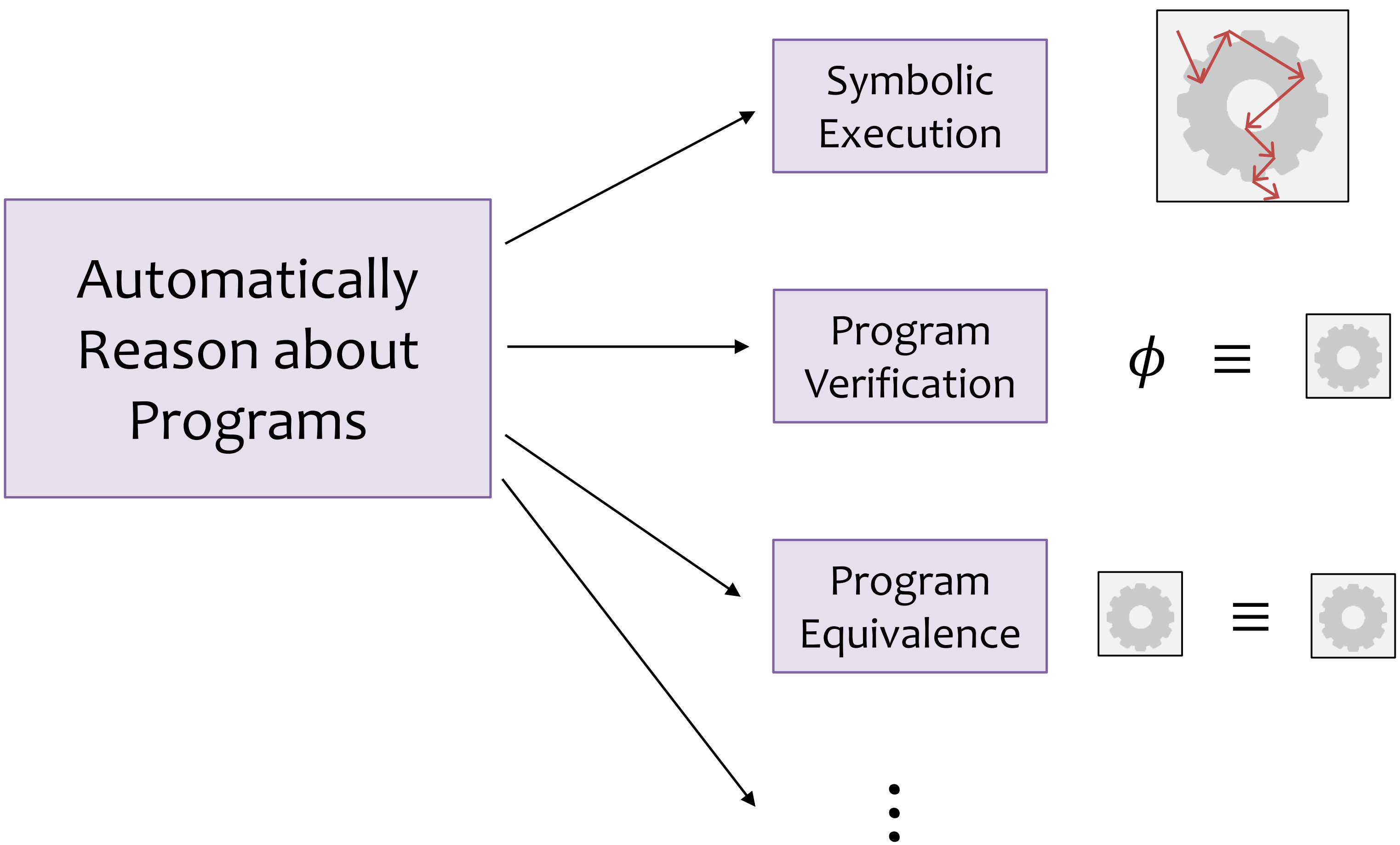
Stratified Synthesis: Automatically Learning the x86-64 Instruction Set

Stefan Heule, Eric Schkufza, Rahul Sharma, Alex Aiken

PLDI, Santa Barbara, June 16, 2016

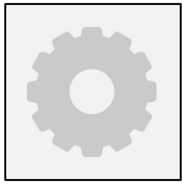
Motivation


Automatically Reason about Programs
Symbolic Execution
Program Verification
$\phi \equiv$
Program Equivalence
$\equiv$

Automatically reasoning about programs requires

Formal Semantics

x86-64



```
  testq %rdi, %rdi
  je .L1
  xorq %rax, %rax
.L0:
  movq %rdi, %rdx
  andq $0x1, %rdx
  addq %rdx, %rax
  shrq $0x1, %rdi
  jne .L0
  cltq
  retq
.L1:
  xorq %rax, %rax
  retq
```

x86-64 Semantics

64-bit bit-vector addition

```
addq $0x1, %rax
```

$$\text{rax} \leftarrow \text{rax} \;+_{64}\; 1_{64}$$

64-bit constant

previous value of rax

x86-64 Semantics

`addq $0x1, %rax`	$\mathrm{rax} \leftarrow \mathrm{rax} +_{64} 1_{64}$
`addb $0x1, %al`	$\mathrm{al} \leftarrow \mathrm{al} +_{8} 1_{8}$

x86-64 Semantics

`addq $0x1, %rax` $\quad rax \leftarrow rax +_{64} 1_{64}$

`addb $0x1, %al` $\quad al \leftarrow al +_{8} 1_{8}$

x86-64 Semantics

```
addq $0x1, %rax
```

$\mathrm{rax} \leftarrow \mathrm{rax} +_{64} 1_{64}$

```
addb $0x1, %al
```

$\mathrm{al} \leftarrow \mathrm{al} +_{8} 1_{8}$

$\mathrm{rax} \leftarrow \mathrm{rax}[63:8] \circ (\mathrm{rax}[7:0] +_{8} 1_{8})$

x86-64 Semantics

```
addq $0x1, %rax
```
$$\text{rax} \leftarrow \text{rax} +_{64} 1_{64}$$

```
addb $0x1, %al
```
$$\text{rax} \leftarrow \text{rax}[63:8] \circ (\text{rax}[7:0] +_{8} 1_{8})$$

```
addw $0x1, %ax
```
$$\text{rax} \leftarrow \text{rax}[63:16] \circ (\text{rax}[15:0] +_{16} 1_{16})$$

```
addl $0x1, %eax
```
$$\text{rax} \leftarrow \text{rax}[63:32] \circ (\text{rax}[31:0] +_{32} 1_{32})$$

x86-64 Semantics

`addq $0x1, %rax`	$\text{rax} \leftarrow \text{rax} \ +_{64} \ 1_{64}$
`addb $0x1, %al`	$\text{rax} \leftarrow \text{rax}[63:8] \circ (\text{rax}[7:0] +_{8} \ 1_{8})$
`addw $0x1, %ax`	$\text{rax} \leftarrow \text{rax}[63:16] \circ (\text{rax}[15:0] +_{16} \ 1_{16})$
`addl $0x1, %eax`	$\text{rax} \leftarrow 0_{32} \circ (\text{rax}[31:0] \ +_{32} \ 1_{32})$

x86-64 Semantics

`addq $0x1, %rax`	$rax \leftarrow rax \ +_{64} \ 1_{64}$
`addb $0x1, %al`	$rax \leftarrow rax[63:8] \circ (rax[7:0] +_{8} \ 1_{8})$
`addw $0x1, %ax`	$rax \leftarrow rax[63:16] \circ (rax[15:0] +_{16} \ 1_{16})$
`addl $0x1, %eax`	$rax \leftarrow 0_{32} \circ (rax[31:0] \ +_{32} \ 1_{32})$

$$zf \leftarrow 0_{32} = (eax \ +_{32} \ 1_{32})$$

$$cf \leftarrow \big((0_{1} \circ eax) \ +_{33} \ 1_{33}\big)[32,32]$$

$$sf \leftarrow (eax \ +_{32} \ 1_{32})[31,31]$$

$$of \leftarrow \neg eax[31,31] \wedge (eax \ +_{32} \ 1_{32})[31,31]$$

$$\begin{aligned} pf \leftarrow\ & (eax \ +_{32} \ 1_{32})[0,0] \oplus (eax \ +_{32} \ 1_{32})[1,1] \oplus \\ & (eax \ +_{32} \ 1_{32})[2,2] \oplus (eax \ +_{32} \ 1_{32})[3,3] \oplus \\ & (eax \ +_{32} \ 1_{32})[4,4] \oplus (eax \ +_{32} \ 1_{32})[5,5] \oplus \\ & (eax \ +_{32} \ 1_{32})[6,6] \oplus (eax \ +_{32} \ 1_{32})[7,7] \end{aligned}$$

Related Work

Manual partial specifications

- CompCert [CACM'09], BAP [CAV'11], BitBlaze [ICISS'08], Codesurfer/x86 [ETAPS'05], McVeto [CAV'10], STOKE [ASPLOS'13], Jakstab [CAV'08], many others

Taly/Godefroid [PLDI'12]

- Automatically synthesize specification from templates
- Only 534 instructions

Automatically Learn a Specification for the x86-64 ISA

Bit-vector formulas of input-output behavior

Strategy: Split Instruction Set

All instructions

Base set

Remaining Instructions

Specify manually

Learn specification automatically

Strategy: Using Program Synthesis

1. How do we synthesize programs?

2. Formal guarantee? $i \equiv \phi$

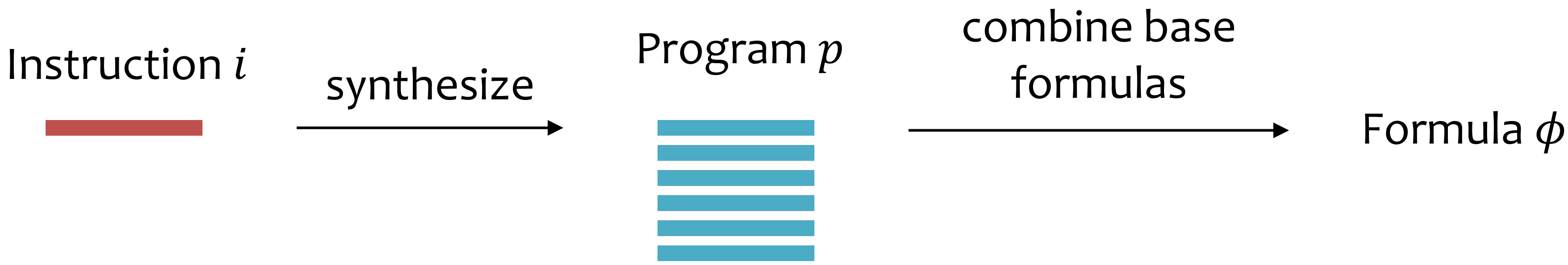
Strategy: Using Program Synthesis

1 How do we synthesize programs?

Randomized search
Guided by cost function
Based on test-cases

Using STOKE [ASPLOS'13]

Strategy: Using Program Synthesis



$$p \equiv \phi$$

2 Formal guarantee?

$$i \equiv \phi$$

Strategy: Using Program Synthesis

Instruction i $\xrightarrow{\text{synthesize}}$ Program p $\xrightarrow{\text{combine base formulas}}$ Formula ϕ

$$i \cancel{\equiv} p \equiv \phi$$

2 Formal guarantee?

$$i \equiv \phi$$

Strategy: Using Program Synthesis

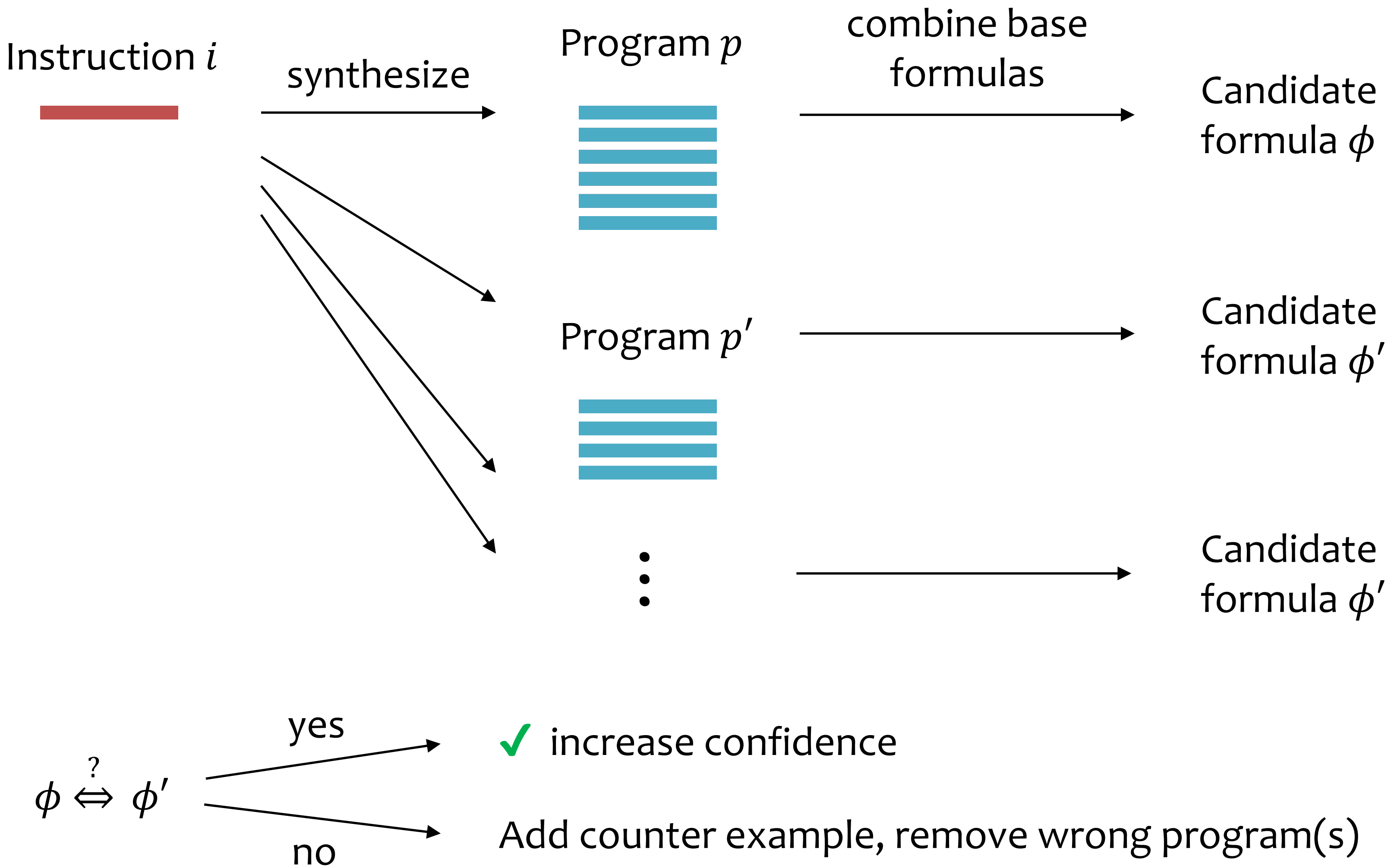
Instruction i $\xrightarrow{\text{synthesize}}$ Program p $\xrightarrow{\text{combine base formulas}}$ Candidate formula ϕ

$$i \not\equiv p \equiv \phi$$

2 Formal guarantee?

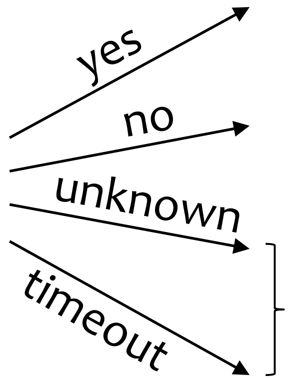
$$i \equiv \phi$$

Strategy: Using Program Synthesis



Solver Imprecision

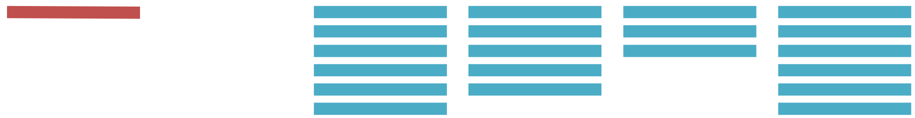
$\phi \overset{?}{\Leftrightarrow} \phi'$



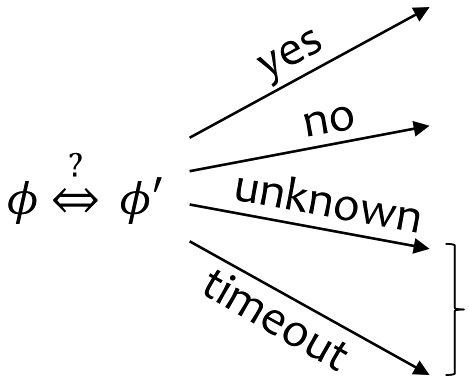
Increase confidence

Remove incorrect program(s)

No information about equivalence



Solver Imprecision



Increase confidence

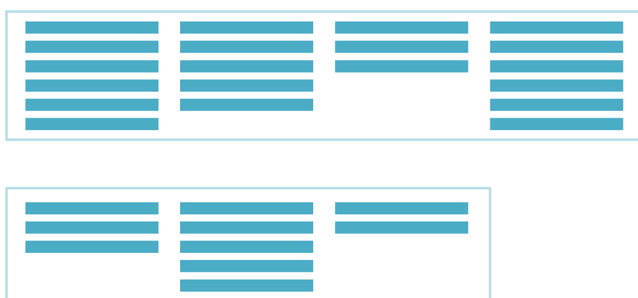
Remove incorrect program(s)

No information about equivalence

Solver Imprecision

$\phi \overset{?}{\Leftrightarrow} \phi'$

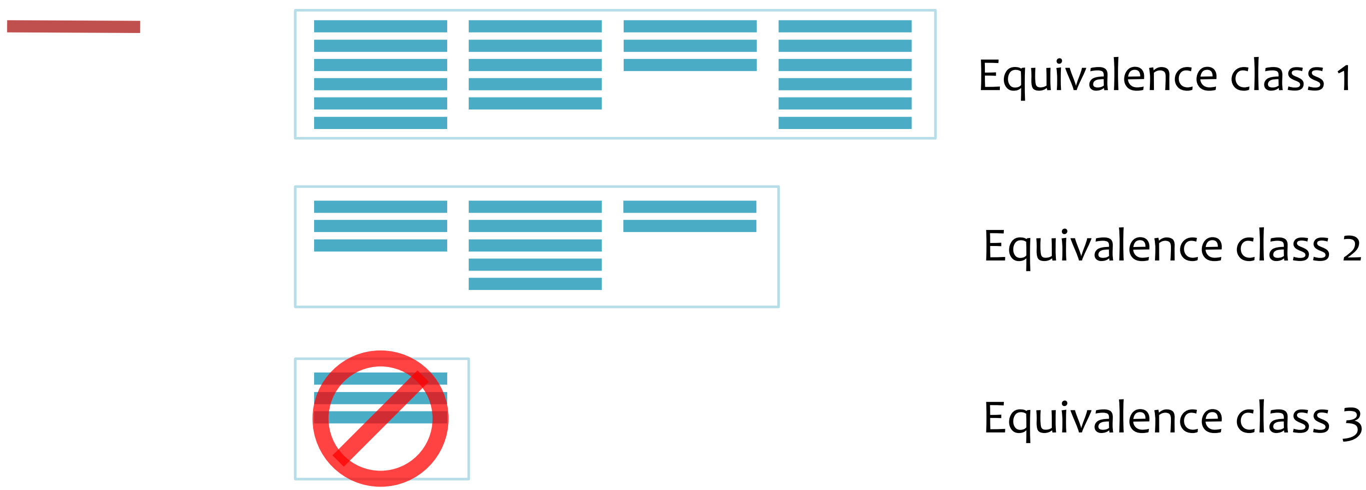
- $\xrightarrow{\text{yes}}$ Increase confidence
- $\xrightarrow{\text{no}}$ Remove incorrect program(s)
- $\xrightarrow{\text{unknown}}$, $\xrightarrow{\text{timeout}}$ No information about equivalence



Equivalence class 1

Equivalence class 2

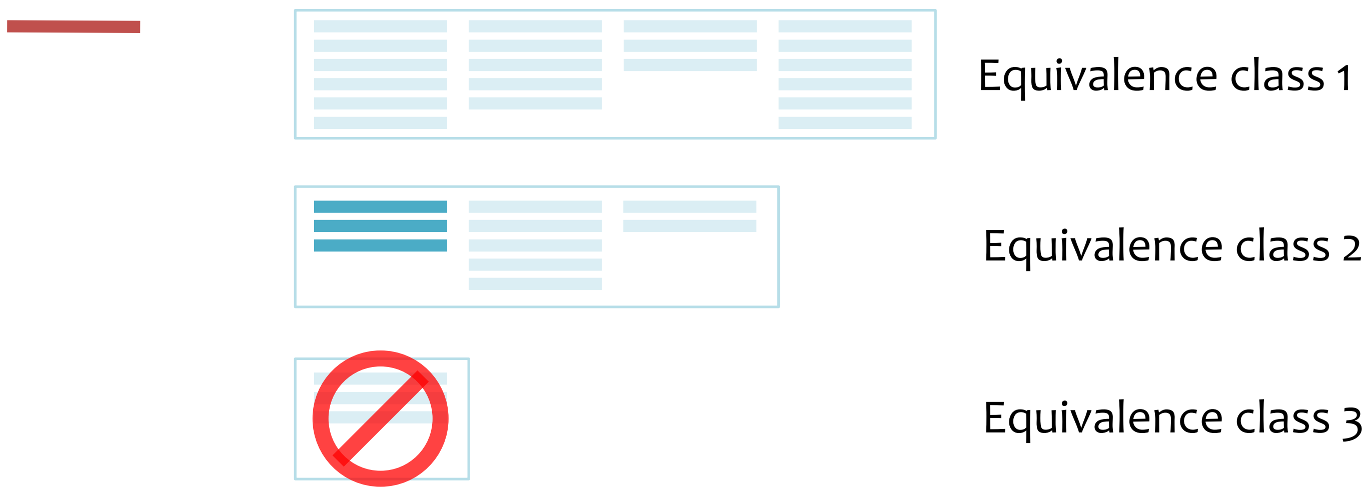
Picking a Program



Prefer programs whose formulas are

- **Precise** (fewest uninterpreted functions)
- **Fast** (fewest non-linear arithmetic operations)
- **Simple** (fewest nodes)

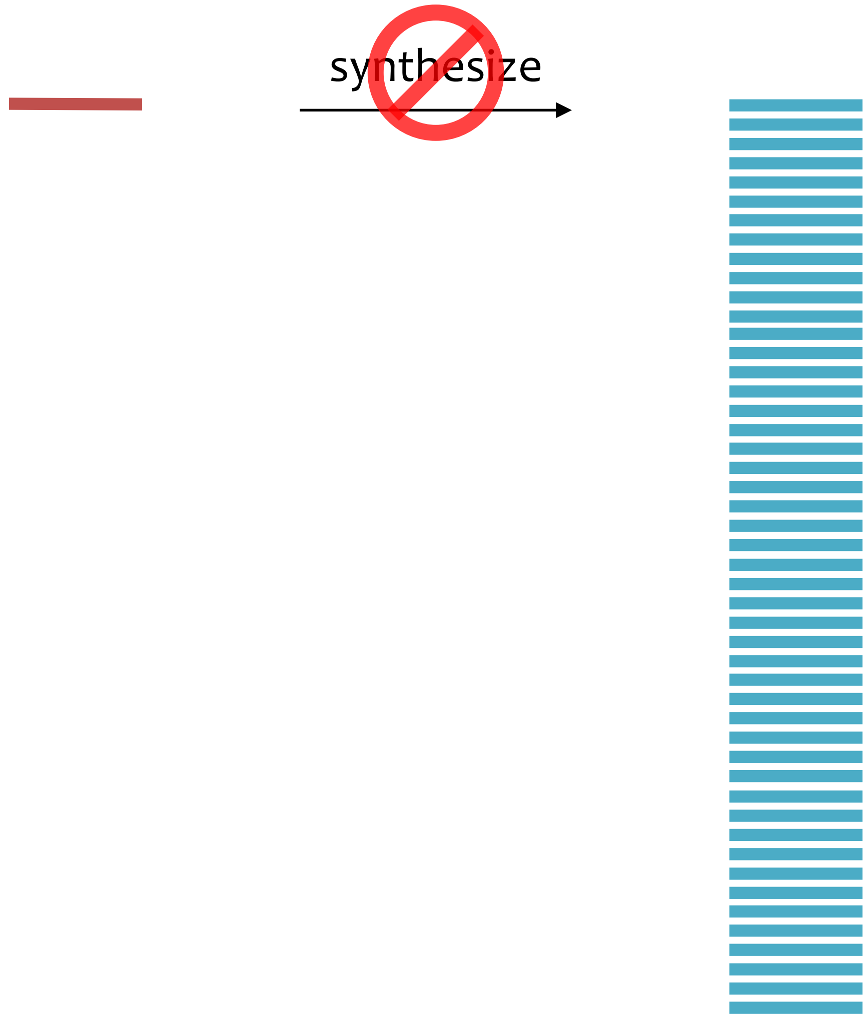
Picking a Program



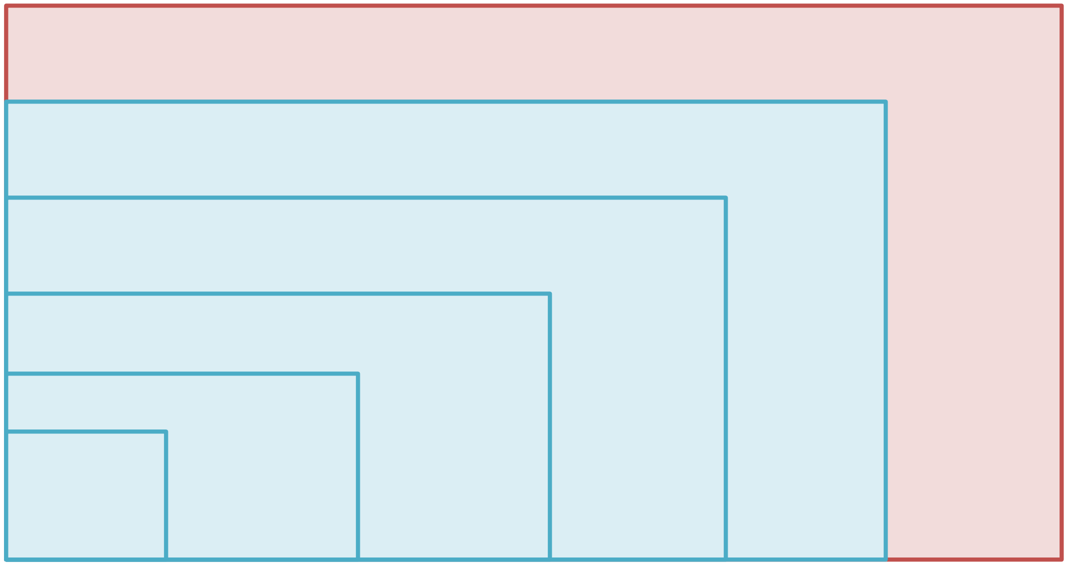
Prefer programs whose formulas are

– **Precise** (fewest uninterpreted functions)
– **Fast** (fewest non-linear arithmetic operations)
– **Simple** (fewest nodes)

Problem: Synthesis Limitations



Solution: Stratified Search



Generalizing Formulas

Learn

`addw %ax, %dx` → ☰ → $dx \leftarrow dx +_{16} ax$

Rename

`addw %cx, %bx` $bx \leftarrow bx +_{16} cx$ ✓

`addw (%rsp), %dx` $dx \leftarrow dx +_{16} M[rsp]$ ✓

`addw $0x5, %dx` $dx \leftarrow dx +_{16} 5_{16}$ ✓

Generalization Summary

1. Learn formula for register-only instructions

2. Generalize formulas
   - To other types of operands

3. Check on test inputs

What if Generalization Impossible?

```
shufps $0xb3, %xmm0, %xmm1
```

Problem: No corresponding register-only variant

Solution: Brute force a formula for every constant

Experiment

Base set (51 instructions)

– Integer, bitwise and float operations
– Data movement (including conditional move)
– Conversion operations

Pseudo instructions (11 templates)

– Split and combine registers
– Changing status flags

Goal

Total instructions		3,684
Out-of-scope		
– System instructions	`invpcid, jle`	302
– Crypto instructions	`aeskeygenassist`	35
– Deprecated instructions	`fadd`	332
– String instructions	`scasq`	97
Goal instructions		2,918

Results

Base set	51
Pseudo instructions	11
Register-only instructions learned	692
Generalized	984
8-bit constant instructions learned	119.42
Total formulas learned	**1,795.42**

Evaluation: Are the Formulas Correct?

Compare with handwritten formulas (from STOKE)

Available for comparison	1,431.91
Automatically proven equivalent	1,377.91
Equivalent with additional lemma	4

Evaluation: Are the Formulas Correct?

Compare with handwritten formulas (from STOKE)

Availab	1.91
Automa	7.91
Equival	4

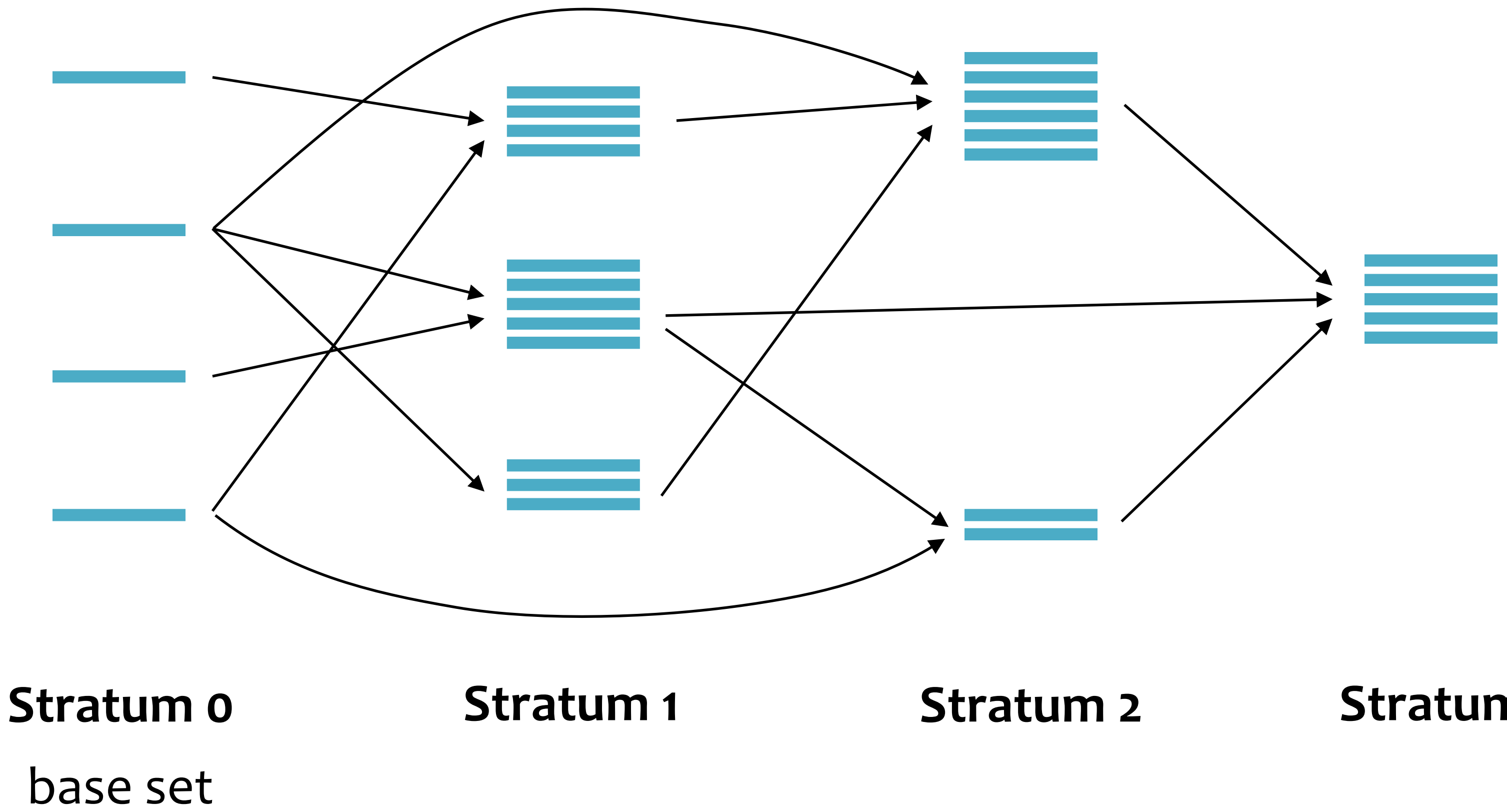
$$\mathrm{fadd}(a, b) = \mathrm{fadd}(b, a)$$

Evaluation: Are the Formulas Correct?

Compare with handwritten formulas (from STOKE)

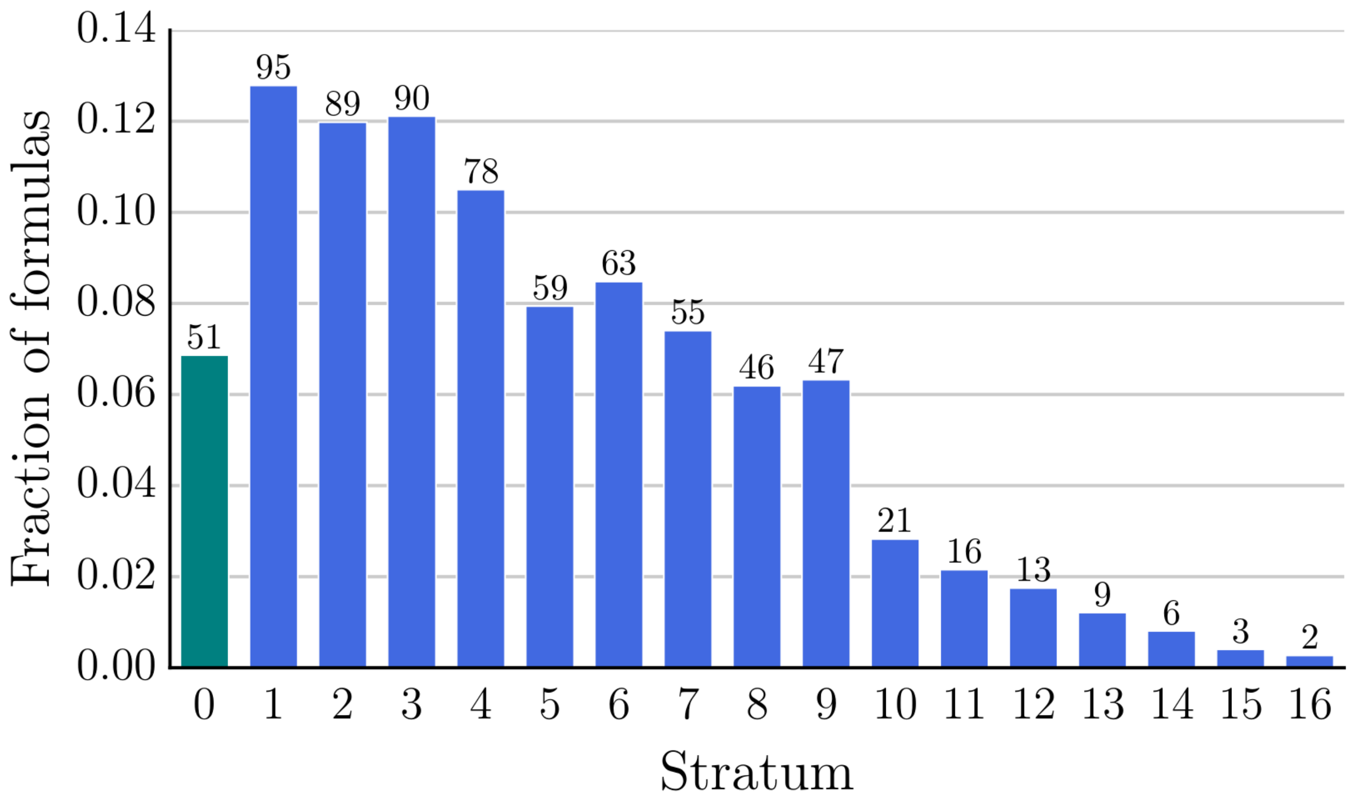
Available for comparison	1,431.91
Automatically proven equivalent	1,377.91
Equivalent with additional lemma	4
Semantically different	50
Handwritten formula correct	0
Learned formula correct	50

Evaluation: Is Stratification Necessary?



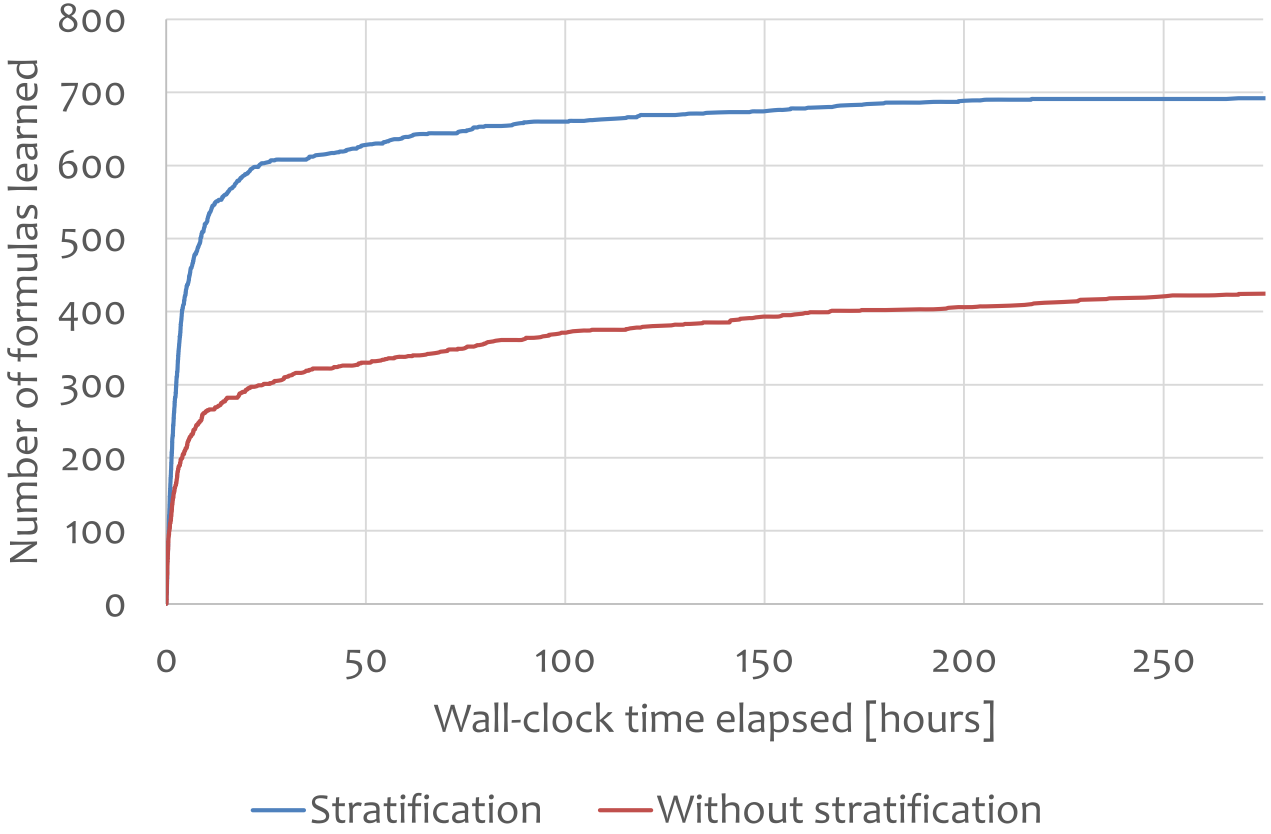
$$\text{stratum}(i) = \begin{cases} 0 & \text{if } i \in \text{baseset} \\ 1 + \max\limits_{i' \in M(i)} \text{stratum}(\text{i}') & \text{otherwise} \end{cases}$$

Evaluation: Is Stratification Necessary?

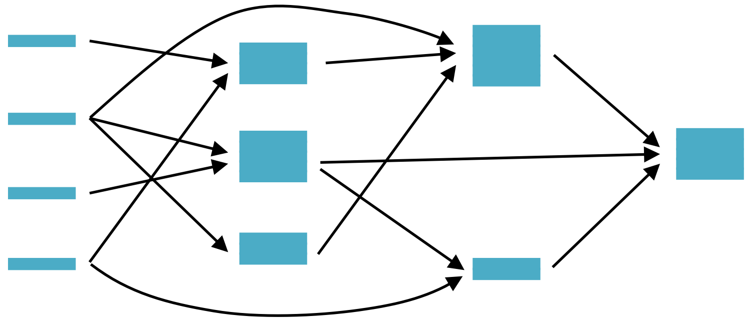


$$\text{stratum}(i) = \begin{cases} 0 & \text{if } i \in \text{baseset} \\ 1 + \max\limits_{i' \in M(i)} \text{stratum}(\mathrm{i}') & \text{otherwise} \end{cases}$$

Evaluation: Is Stratification Necessary?

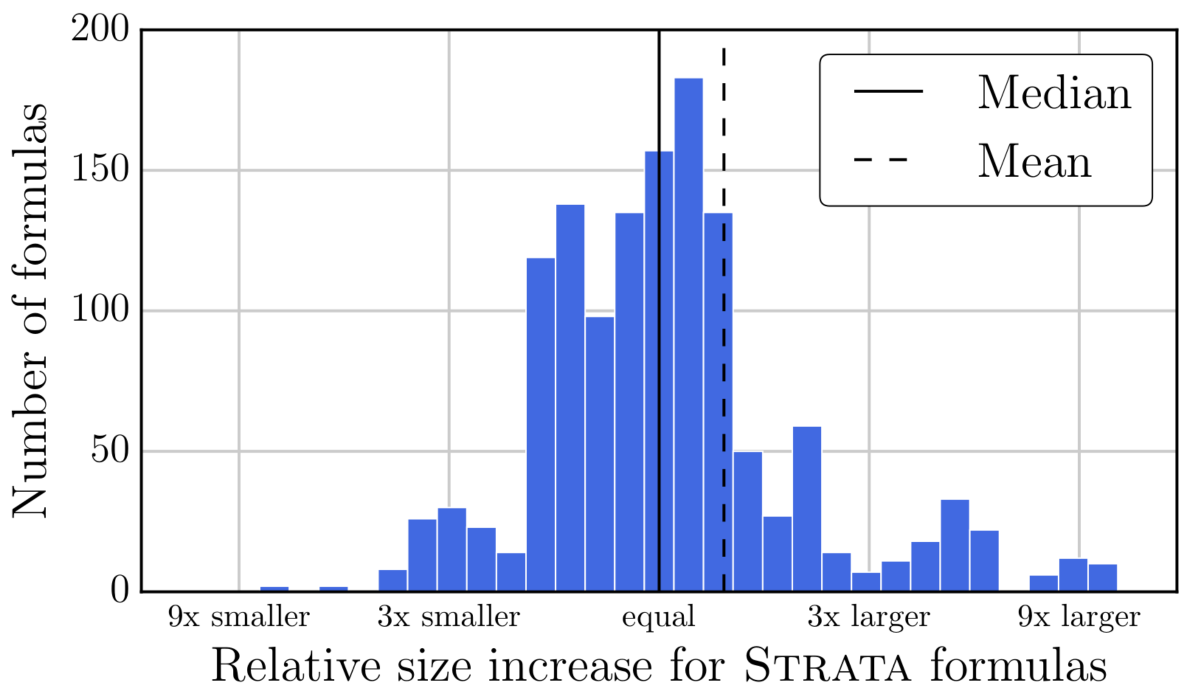


Evaluation: Size of Learned Formulas



Fully inlined: 3526 instructions

$$\frac{\text{number of nodes in learned formula}}{\text{number of nodes in handwritten formula}}$$



Conclusions

1. Automatically learned 1,795 formulas
2. Stratification key to scale program synthesis
3. Compare to hand-written specification
   - More correct, equally precise, same size

Source code, formulas, experimental results

https://github.com/StanfordPL/strata/

Backup Slides

Limitations

1. Missing base instructions

   Some integer and floating point operations are missing

2. Program synthesis limits

   Shortest known program is long and outside of reach

   e.g., byte-vectorized operation

3. Cost function limitation

   For one bit of output, the cost function does not give enough signal

4. Crazy instructions

SMT solver usage (Z3)

Total decisions	7,075	
Equivalent	6,669	(94.26%)
New equivalence class	356	(5.03%)
Counter-examples	50	(0.71%)

Timeouts (45 seconds):	3

Experiment Details

Intel Xeon E5-2697 (28 cores) at 2.6 GHz

– 268.86 hours (register-only)

– 159.12 hours (8-bit constants)

Total of 11,983.37 core hours

Test Cases

Random inputs (random machine state)

"Interesting" bit-patterns

$0, 1, -1, 2^n, \text{NaN}, \text{Infinity}$

Test cases learned from counter-examples

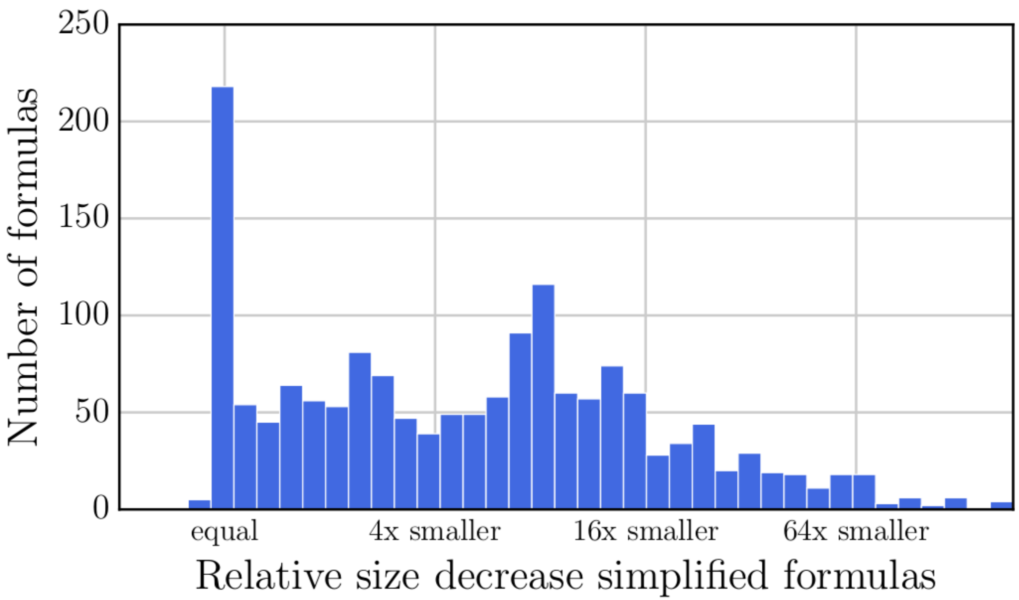
Evaluation: Simplifcation

Formulas are simplified

– Constant propagation $2_{64} *_{64} 4_{64} \equiv 8_{64}$

– Move bit-selection over concatenation

$$(0_{64} \circ \mathrm{rax})[63,0] \equiv \mathrm{rax}$$



Evaluation: Formula Precision

Formula precision (number of uninterpreted functions)

– Learned formulas equally precise in all but 4 cases

Formula quality (number of non-linear operations)

– Learned formulas contain same number of non-linear operations, except for 11 cases