Almost Verification of Programs with Concealed Components

SPLASH Doctoral Symposium 2022

Sumit Lahiri

Dr. Subhajit Roy, Thesis Advisor (PRAISE Group)
Dept. Of Computer Science & Engineering, IIT Kanpur

Program with *concealed* components

```
def main():
    ...
    a = bar();
    ...
    while (a > 42):
      ...
      a += foo();
    ...
    assert(a * a >= 900)
```

Program with *concealed* components

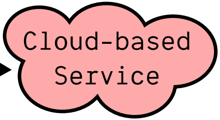
```
def main():
    ...
    a = bar();
    ...
    while (a > 42):
      ...
      a += foo();
    ...
    assert(a * a >= 900)
```

Concealed Components

Program with *concealed* components

```
def main():
    ...
    a = bar();
    ...
    while (a > 42):
        ...
        a += foo();
    ...
    assert(a * a >= 900)
```

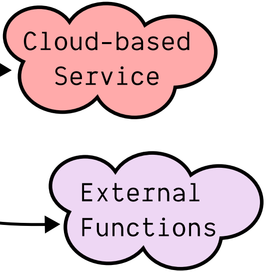
Concealed Components



Program with *concealed* components

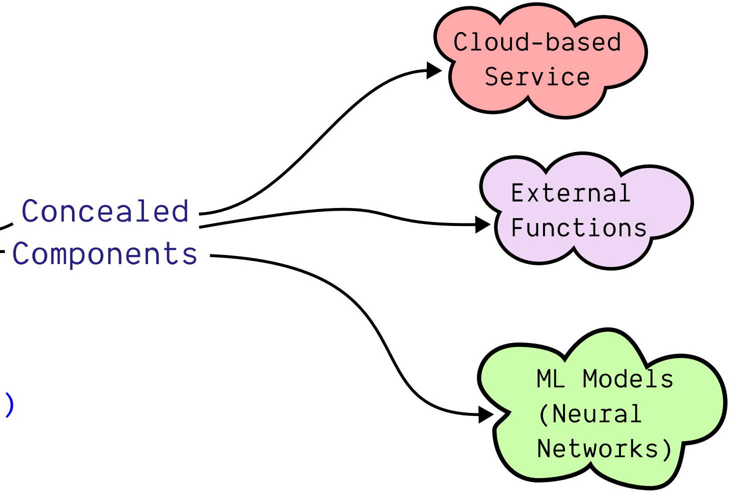
```
def main():
    ...
    a = bar();
    ...
    while (a > 42):
        ...
        a += foo();
    ...
    assert(a * a >= 900)
```

Concealed Components

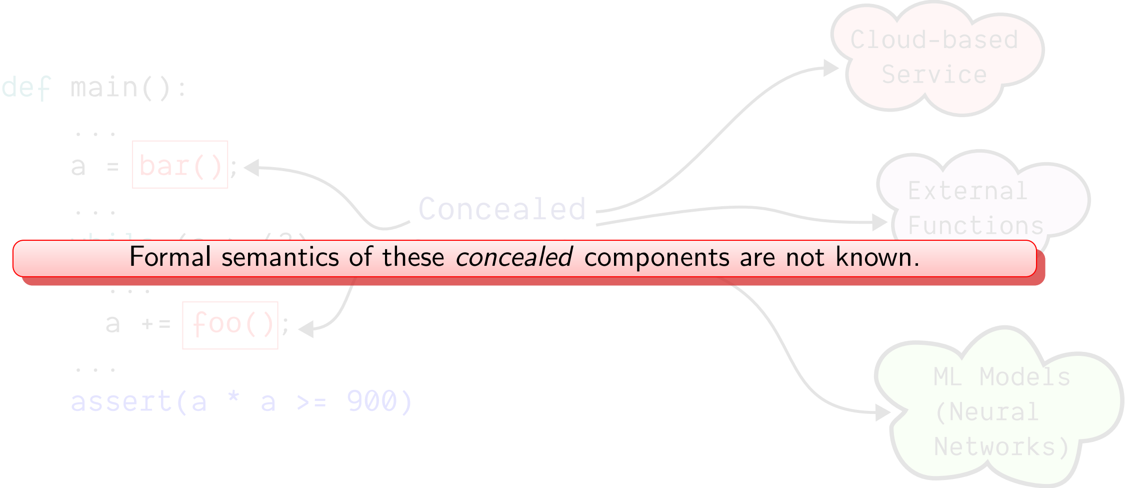


Program with *concealed* components

```
def main():
    ...
    a = bar();
    ...
    while (a > 42):
      ...
      a += foo();
    ...
    assert(a * a >= 900)
```



Program with *concealed* components



Formal semantics of these *concealed* components are not known.

Types of *Concealed* Components

Unknown Oracles

No information available about behavior of these *concealed* components.

Types of *Concealed* Components

Unknown Oracles

No information available about behavior of these *concealed* components.

Executable Oracles

Concealed components whose **input-output behavior** can be analyzed.

Types of *Concealed* Components

Unknown Oracles

No information available about behavior of these *concealed* components.

Executable Oracles

Concealed components whose **input-output behavior** can be analyzed.

Stochastic Oracles

Concealed components whose execution semantics are not known but **output** from these components can be modeled as a **probability** distribution.

Types of *Concealed* Components

Unknown Oracles

No information available about behavior of these *concealed* components.

Executable Oracles

Almost correct invariants, ISSTA 2022

Concealed components whose **input-output behavior** can be analyzed.

Stochastic Oracles

Concealed components whose execution semantics are not known but **output** from these components can be modeled as a **probability** distribution.

Types of *Concealed* Components

Unknown Oracles

No information available about behavior of these *concealed* components.

Executable Oracles

Almost correct invariants, ISSTA 2022

Concealed components whose **input-output behavior** can be analyzed.

Stochastic Oracles

Symbolic Execution for Randomized Programs

Concealed components whose execution semantics are not known but **output** from these components can be modeled as a **probability** distribution.

What is *Almost* Verification?

Almost Verification

Given a program $\mathcal{P}$ containing **concealed components**, does property ψ hold for the program $\mathcal{P}$ for all inputs with which $\mathcal{P}$ can be executed with?

What is *Almost* Verification?

Almost Verification

Given a program $\mathcal{P}$ containing **concealed components**, does property ψ hold for the program $\mathcal{P}$ for all inputs with which $\mathcal{P}$ can be executed with?

Does property ψ hold for all states of the program *modulo* the **concealed components**?

Almost correct invariants: synthesizing inductive invariants by fuzzing proofs.

Sumit Lahiri, Subhajit Roy. ISSTA 2022

Almost correct invariants

```
Precondition (φ_pre): (b0 >= 0)
int multiply (s⃗: [a0, b0, supported]){
  // loop-head
  int a = a0, b = b0, r=0; shift=0;
  while (b != 0) { // loop guard
    if (supported) {  // loop-body
      shift = __builtin_ctz(b) ;
    } else {
      shift = 0; }
    if (shift) {
      r += a << shift;
      b -= 1 << shift;
    } else {
      r += a; b -= 1;
  }}
  return r; // loop tail
}
Postcondition (φ_post): (r == a0 * b0)
```

Almost correct invariants

```
Precondition (φ_pre): (b0 >= 0)
int multiply (s⃗: [a0, b0, supported]){
  // loop-head
  int a = a0, b = b0, r=0; shift=0;
  while (b != 0) { // loop guard
    if (supported) {  // loop-body
      shift = __builtin_ctz(b) ;
    } else {
      shift = 0; }
    if (shift) {
      r += a << shift;
      b -= 1 << shift;
    } else {
      r += a; b -= 1;
  }}
  return r; // loop tail
}
Postcondition (φ_post): (r == a0 * b0)
```

Existing Tools

Logical encoding is not possible so existing tools fail!

Almost correct invariants

```
Precondition (φ_pre): (b0 >= 0)
int multiply (s⃗: [a0, b0, supported]){
  // loop-head
  int a = a0, b = b0, r=0; shift=0;
  while (b != 0) { // loop guard
    if (supported) {  // loop-body
      shift = __builtin_ctz(b) ;
    } else {
      shift = 0; }
    if (shift) {
      r += a << shift;
      b -= 1 << shift;
    } else {
      r += a; b -= 1;
  }}
  return r; // loop tail
}
Postcondition (φ_post): (r == a0 * b0)
```

Existing Tools

Logical encoding is not possible so existing tools fail!

Our tool, Achar

```
Inv: (b0 >= 0) && (a0 == a) && (r == (b0-b) * a)
```

Problem Statement

Given a specification $\{\varphi_{pre}\}\ \mathcal{P}\ \{\varphi_{post}\}$,

Lahiri et. al. ISSTA 2022

Problem Statement

Given a specification $\{\varphi_{pre}\}\ \mathcal{P}\ \{\varphi_{post}\}$, where the program $\mathcal{P}$ may contain *concealed* operations,

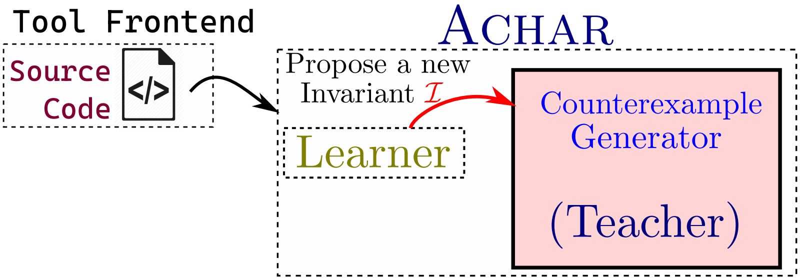
Lahiri et. al. ISSTA 2022

Problem Statement

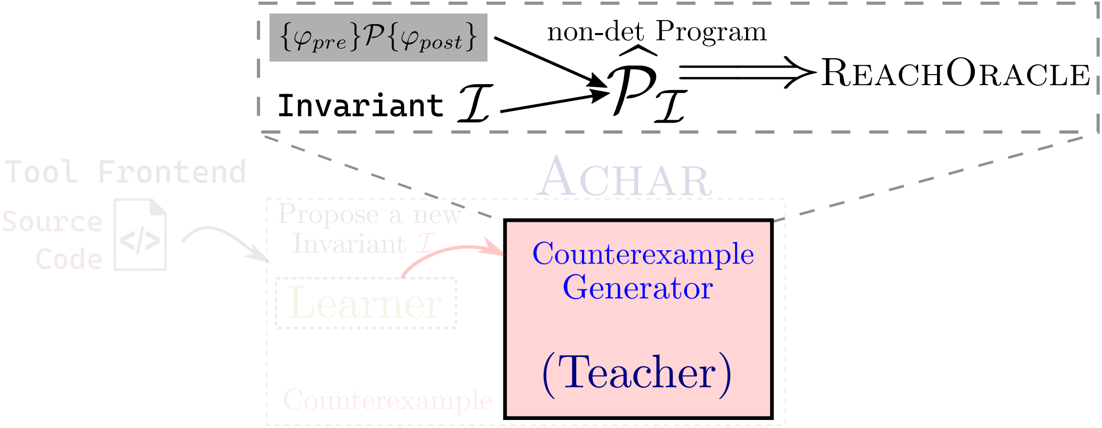
Given a specification $\{\varphi_{pre}\}\ \mathcal{P}\ \{\varphi_{post}\}$, where the program $\mathcal{P}$ may contain *concealed* operations, we attempt to synthesize an *inductive loop invariant*($\mathcal{I}$) that establishes the correctness proof.

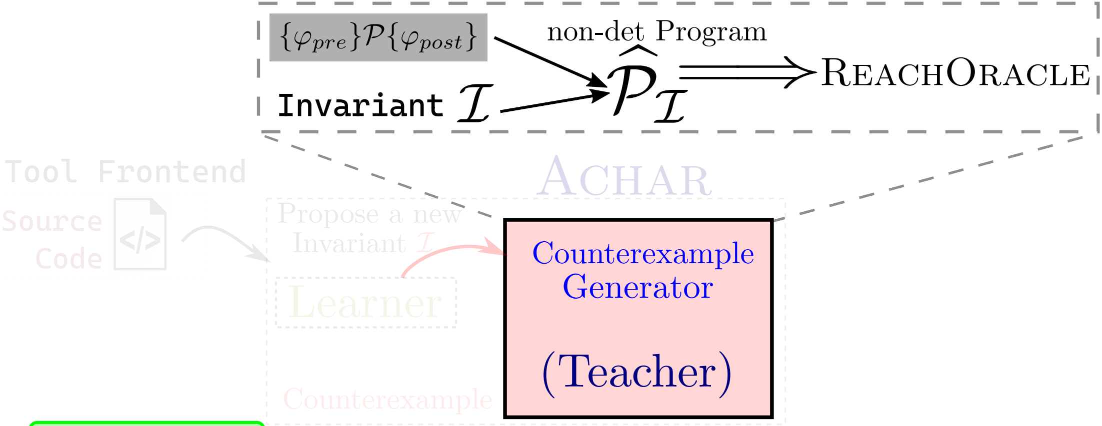
Lahiri et. al. ISSTA 2022

Learner proposes a candidate Invariant($\mathcal{I}$)



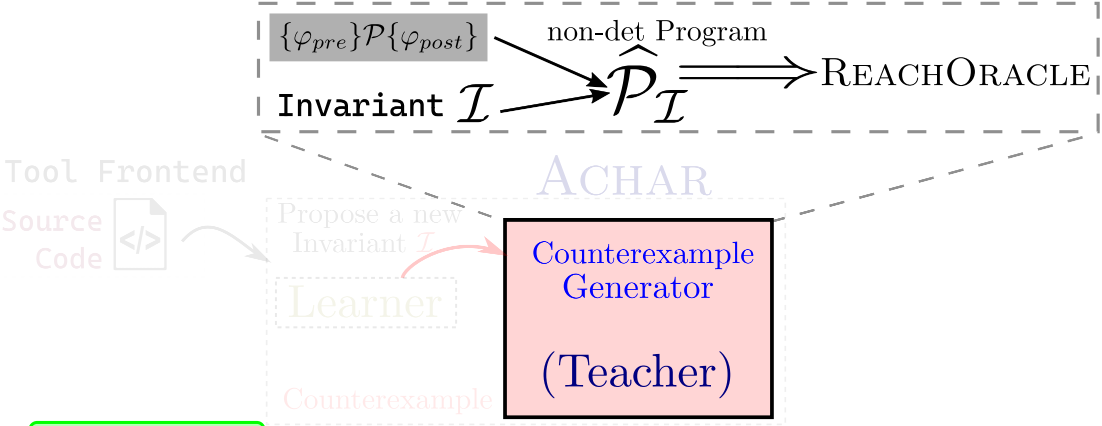
Teacher generates a non-det program for *proof fuzzing*





Proof Fuzzing

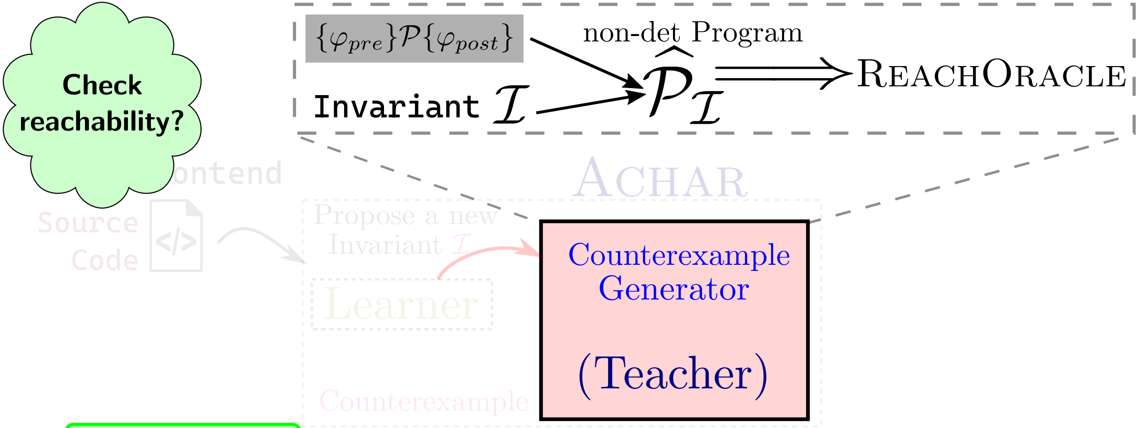
If **any** of the "*goal*" points in $\widehat{\mathcal{P}_{\mathcal{I}}}$ is reachable,



Proof Fuzzing

If **any** of the "*goal*" points in $\widehat{\mathcal{P}_{\mathcal{I}}}$ is reachable, candidate invariant, $\mathcal{I}$ is deemed *in-correct* for $\{\varphi_{pre}\}\ \mathcal{P}\ \{\varphi_{post}\}$.

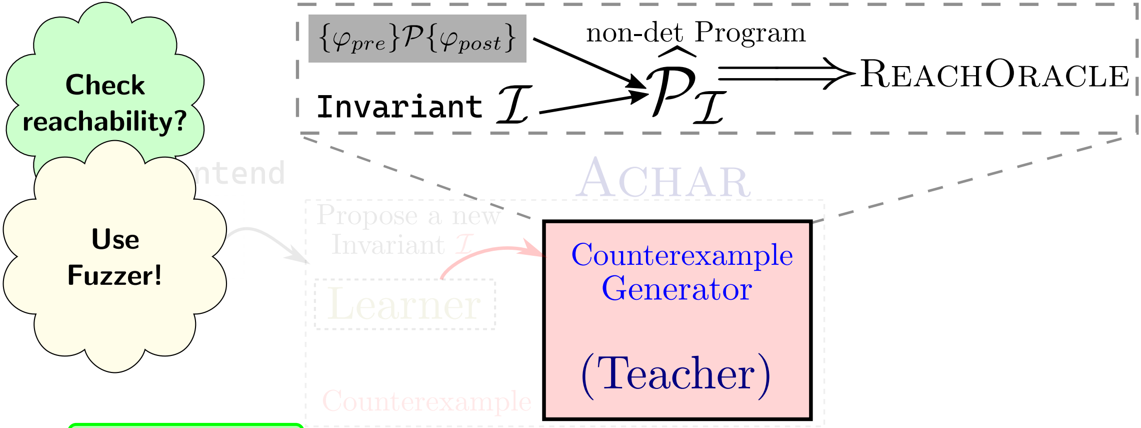
Proof fuzzing



Proof Fuzzing

If **any** of the "*goal*" points in $\widehat{\mathcal{P}_{\mathcal{I}}}$ is reachable, candidate invariant, $\mathcal{I}$ is deemed *in-correct* for $\{\varphi_{pre}\}\ \mathcal{P}\ \{\varphi_{post}\}$.

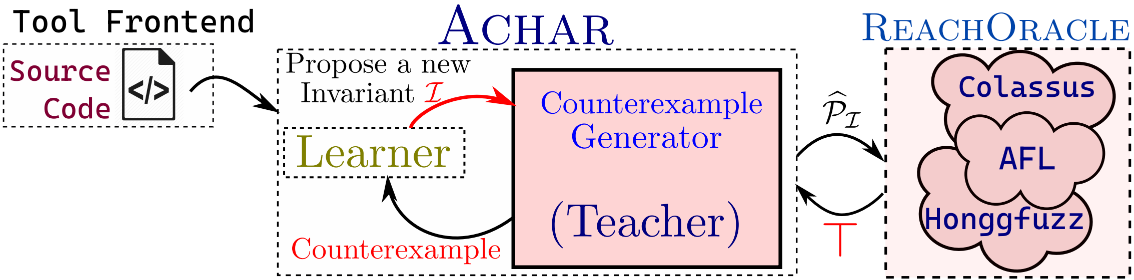
Proof fuzzing



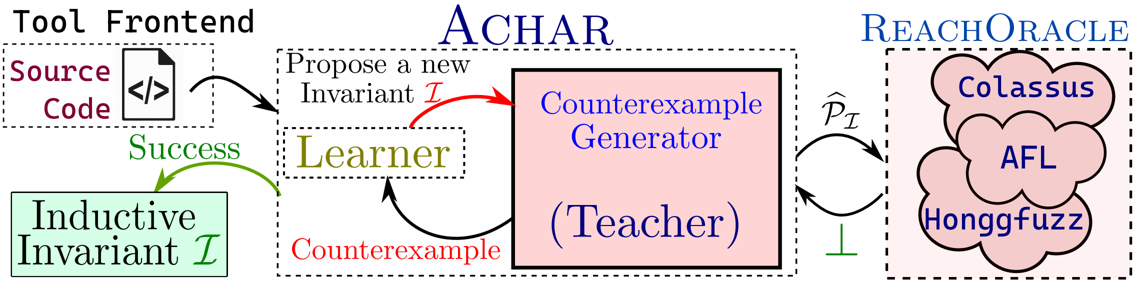
Proof Fuzzing

If **any** of the "*goal*" points in $\widehat{\mathcal{P}_{\mathcal{I}}}$ is reachable, candidate invariant, $\mathcal{I}$ is deemed *in-correct* for $\{\varphi_{pre}\}\ \mathcal{P}\ \{\varphi_{post}\}$.

ReachOracle() either returns with $\top$ (Cex.)



ReachOracle() either returns with $\perp$ (success)



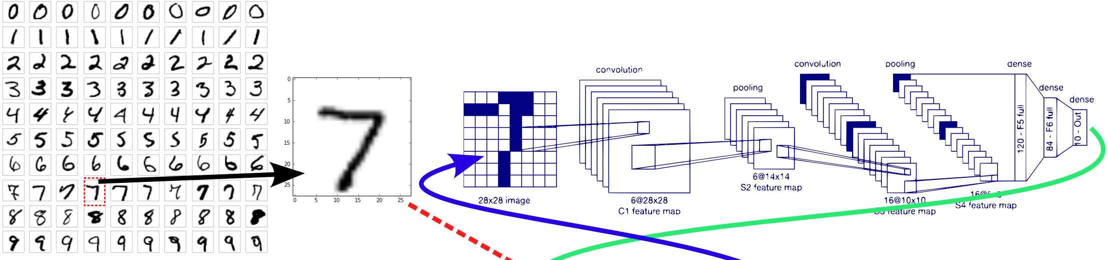
Open-program Examples

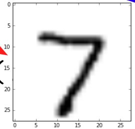
#	Description	Type of Opaque Operations	Time (sec.)	Cex.
1	Sum of Squares	External library Call & Inline Assembly	57.17	7
2	Program using `isprime()`	External library Call	238.62	8
3	Greatest Common Divisor	External library Call & Unsupported operations by theorem prover (LIA Theory)	70.67	7
4	Fibonacci Series	External library Call	253.84	9
5	Exponentiation Modulo-N	Unsupported operations by theorem prover (LIA Theory)	52.37	7
6	Fast Exponentiation	Unsupported operations by theorem prover (LIA Theory)	63.00	7
7	Integer SQRT	Unsupported operations by theorem prover (LIA Theory) in both pre-body and loop-body of the program.	209.68	7
8	Multiply Example	Compiler Primitive	81.39	7
9	Integer Cube Root	External library Call & Unsupported operations by theorem prover (LIA Theory)	102.20	8
10	Lock based program	External library Call & Inline Assembly	66.81	15
11	Algebric Expression (Cube)	External library Call & Unsupported operations by theorem prover (LIA Theory)	523.94	8
12	Integer Division	External library Call & Unsupported operations by theorem prover (LIA Theory)	67.63	7
13	Summing Handwritten Digits	Invoking a Convolutional Neural Network for predicting handwritten digits (`CNN`)	180.97	20
14	Fast Factorial	Compiler Primitive	499.05	10
15	Sum of Cubes	External library Call & Inline Assembly	140.53	8

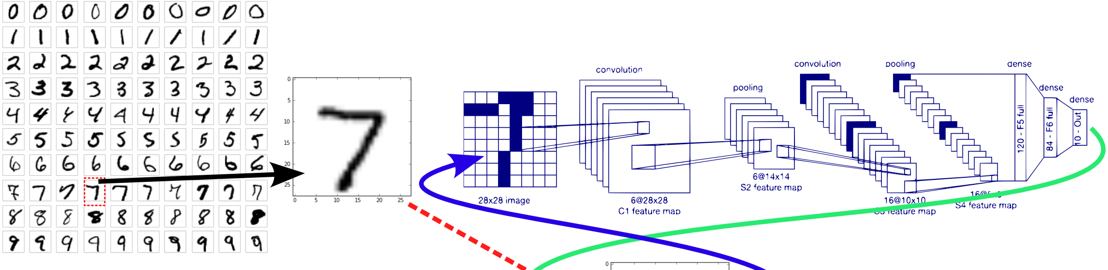
Open-program Examples

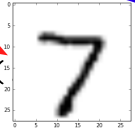
#	Description	Type of Opaque Operations	Time (sec.)	Cex.
1	Sum of Squares	External library Call & Inline Assembly	57.17	7
2	Program using `isprime()`	External library Call	238.62	8
3	Greatest Common Divisor	External library Call & Unsupported operations by theorem prover (LIA Theory)	70.67	7
4	Fibonacci Series	External library Call	253.84	9
5	Exponentiation Modulo-N	Unsupported operations by theorem prover (LIA Theory)	52.37	7
6	Fast Exponentiation	Unsupported operations by theorem prover (LIA Theory)	63.00	7
7	Integer SQRT	Unsupported operations by theorem prover (LIA Theory) in both pre-body and loop-body of the program.	209.68	7
8	Multiply Example	Compiler Primitive	81.39	7
9	Integer Cube Root	External library Call & Unsupported operations by theorem prover (LIA Theory)	102.20	8
10	Lock based program	External library Call & Inline Assembly	66.81	15
11	Algebric Expression (Cube)	External library Call & Unsupported operations by theorem prover (LIA Theory)	523.94	8
12	Integer Division	External library Call &	67.63	7
13	Summing Handwritten Digits	Invoking a Convolutional Neural Network for predicting handwritten digits (`CNN`)	180.97	20
14	Fast Factorial	Compiler Primitive	499.05	10
15	Sum of Cubes	External library Call & Inline Assembly	140.53	8

Verifying Programs invoking CNN



sum = sum + `predict(`  `)`

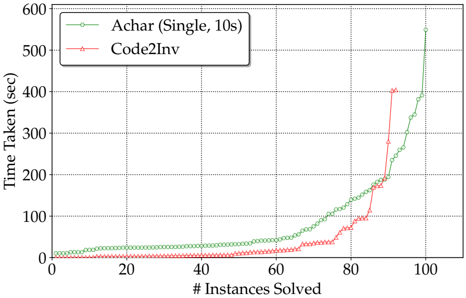


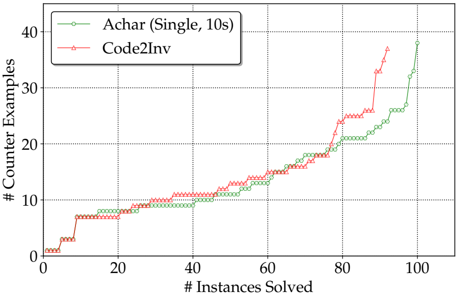
sum = sum + `predict(`  `)`

Invariant Generated

```
((index <= n) && ((sum >= ((5 * ((index/10) - 1) * (index /
10)) + ((index/10) * (index - 10 * (index/10))) - 9)) && (sum
<= ( (5 * ((index/10) - 1) * (index/10)) + ((index/10) * (index
- 10 * (index/10))) + 9))))
```

Synthesis Effectiveness & Runtime cost

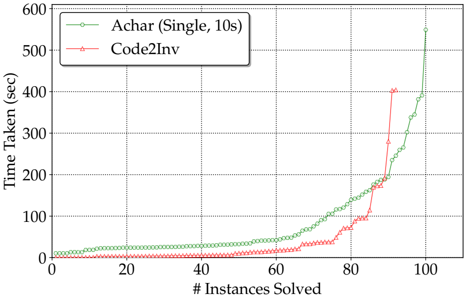


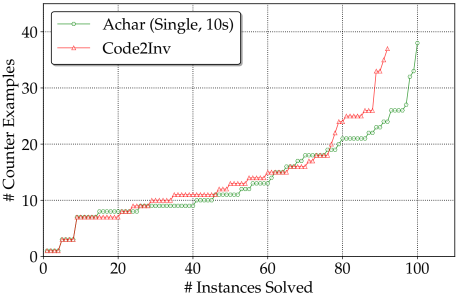


[1] Code2Inv: A Deep Learning Framework for Program Verification, Si et. al, CAV 2020

[2] Deferred concretization in symbolic execution via fuzzing, Pandey et. al, ISSTA 2019

Synthesis Effectiveness & Runtime cost



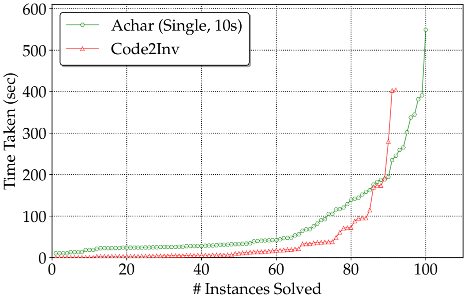


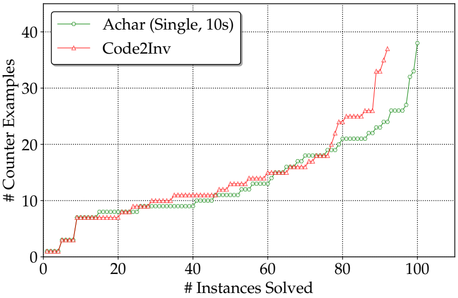
Total	Code2inv [1]
	Correct
133	92

[1] Code2Inv: A Deep Learning Framework for Program Verification, Si et. al, CAV 2020

[2] Deferred concretization in symbolic execution via fuzzing, Pandey et. al, ISSTA 2019

Synthesis Effectiveness & Runtime cost



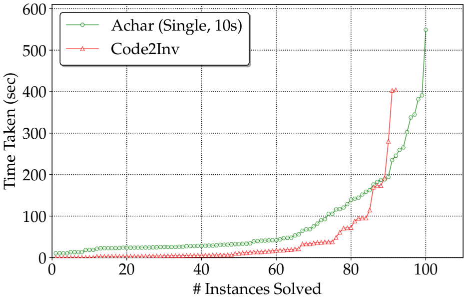


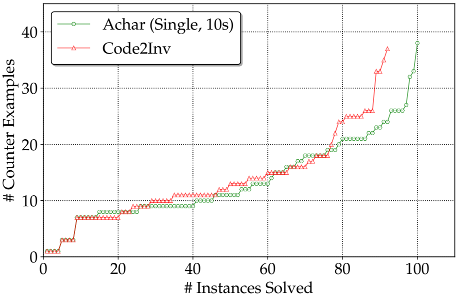
Total	**Code2inv** [1]	**Achar** (Fuzzer)	
	Correct	Correct	Wrong
133	92	**100**	5

[1] Code2Inv: A Deep Learning Framework for Program Verification, Si et. al, CAV 2020

[2] Deferred concretization in symbolic execution via fuzzing, Pandey et. al, ISSTA 2019

Synthesis Effectiveness & Runtime cost





Total	Code2inv [1]	Achar (Fuzzer)		Achar (Colassus) [2]
	Correct	Correct	Wrong	Correct
133	92	**100**	5	**102**

[1] Code2Inv: A Deep Learning Framework for Program Verification, Si et. al, CAV 2020

[2] Deferred concretization in symbolic execution via fuzzing, Pandey et. al, ISSTA 2019

Achar (ISSTA 2022)

Achar is available on Docker Hub[3] and Zenodo[4].

[3] https://hub.docker.com/r/acharver1/achar

[4] https://zenodo.org/record/6534229

Almost Verification with Stochastic Oracles

Stochastic Oracles

Concealed components whose execution semantics are not known but **output** from these components can be modeled as a **probability** distribution.

Randomized Programs

```
main (p) {
  x = 1;
  n = 0;
  while(x == 1) {
    x = bernoulli_dist(p)
    n += 1;
  }
  passert(n >= 10, 0.4);
}
```

```
main (a, b) {
  n = 1000;
  sum = 0;
  while(n > 0) {
    t1 = uniform_dist(a, b)
    t2 = uniform_dist(a, t1 + b)
    if (t1 >= t2)
      sum++;
    n -= 1;
  }
  passert(sum >= 500, 0.5);
}
```

- Sampling statements assign values to program variables from a probability distribution.

Randomized Programs

```
main (p) {
  x = 1;
  n = 0;
  while(x == 1) {
    x = bernoulli_dist(p)
    n += 1;
  }
  passert(n >= 10, 0.4);
}
```

```
main (a, b) {
  n = 1000;
  sum = 0;
  while(n > 0) {
    t1 = uniform_dist(a, b)
    t2 = uniform_dist(a, t1 + b)
    if (t1 >= t2)
      sum++;
    n -= 1;
  }
  passert(sum >= 500, 0.5);
}
```

- Sampling statements assign values to program variables from a probability distribution.
- The specifications for such programs are usually expressed as a *probabilistic* assert.

Probabilistic Inference

Objective

Given an assertion ψ, with what *probability* (p) does ψ hold according to the distribution described by the program, $\mathcal{P}$?

Problem Statement

We aim to answer the following question: What is the maximum (or minimum) probability that a program $\mathcal{P}$, terminates in a state where a predicate ψ holds?

We aim to answer the following question: What is the maximum (or minimum) probability that a program $\mathcal{P}$, terminates in a state where a predicate ψ holds?

$$\max_{\vec{x}} \; \{\Pr_{\mu}[\psi] \mid \mu = \mathcal{P}(\vec{x})\}$$

Problem Statement

We aim to answer the following question: What is the maximum (or minimum) probability that a program $\mathcal{P}$, terminates in a state where a predicate ψ holds?

$$\max_{\vec{x}} \ \{\Pr_{\mu}[\psi] \mid \mu = \mathcal{P}(\vec{x})\}$$

μ is the distribution on outputs obtained by running program, $\mathcal{P}$ on inputs $\vec{x}$.

Problem Statement

We aim to answer the following question: What is the maximum (or minimum) probability that a program $\mathcal{P}$, terminates in a state where a predicate ψ holds?

$$\max_{\vec{x}} \; \{\Pr_{\mu}[\psi] \mid \mu = \mathcal{P}(\vec{x})\}$$

μ is the distribution on outputs obtained by running program, $\mathcal{P}$ on inputs $\vec{x}$.

$$\max_{\vec{x}} \; \{\Pr_{\mu}[\psi] \mid \mu = \mathcal{P}(\vec{x})\} \bowtie f(\vec{x})$$

Problem Statement

We aim to answer the following question: What is the maximum (or minimum) probability that a program $\mathcal{P}$, terminates in a state where a predicate ψ holds?

$$\max_{\vec{x}} \ \{\Pr_{\mu}[\psi] \mid \mu = \mathcal{P}(\vec{x})\}$$

μ is the distribution on outputs obtained by running program, $\mathcal{P}$ on inputs $\vec{x}$.

$$\max_{\vec{x}} \ \{\Pr_{\mu}[\psi] \mid \mu = \mathcal{P}(\vec{x})\} \bowtie f(\vec{x})$$

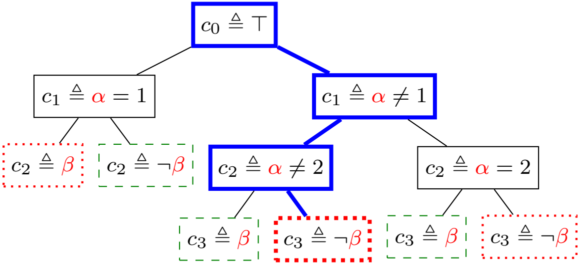
where $\bowtie \ \in \{\leq, \geq, =, \ldots\}$ is a comparison operator and f is some given function of the program inputs $\vec{x}$.

PLINKO, build on the KLEE execution engine for randomized programs with **unknown** input parameters.

Monty Hall Example (under regular SE)

```
function MONTYHALL(choice, door_switch)
    car_door ~ UniformInt(1, 3)                ▷ Sampling
    if choice == car_door then
        return ¬door_switch
    if choice ≠ 1 ∧ car_door ≠ 1 then
        host_door ← 1
    else if choice ≠ 2 ∧ car_door ≠ 2 then
        host_door ← 2
    else
        host_door ← 3
    if door_switch then
        if host_door == 1 then
            if choice == 2 then
                choice = 3
        else if host_door == 2 then
            if choice == 1 then
                choice = 3
        else
            if choice == 1 then
                choice = 2
    if choice == car_door then return 1 else return 0
```

Symbolic Variables : `door_switch`(β), `choice`(α)



Path Condition (ϕ): $(\alpha \neq 1) \wedge (\alpha \neq 2) \wedge (\neg\beta)$

Probabilistic Symbolic Variables.

These model a random sample from a known distribution over a set of values.

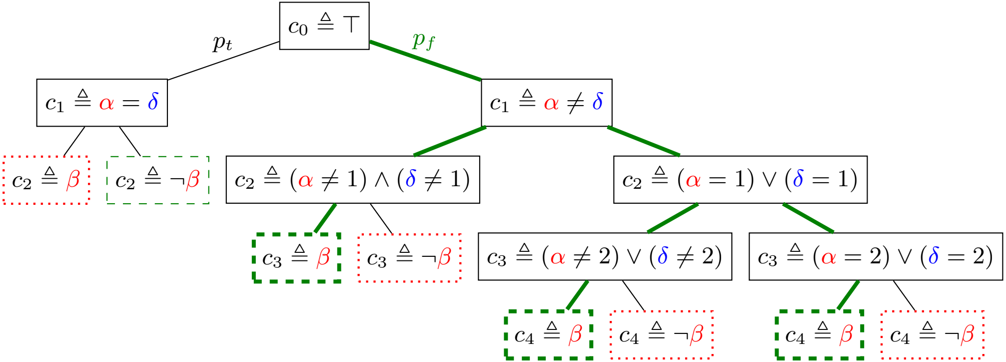
Probabilistic Symbolic Variables.

These model a random sample from a known distribution over a set of values.

- *Distribution map* (P). A map from probabilistic symbolic variables to distribution expressions.
- *Path probability* (p). For each path, we adjoin a path probability expression, p, which may be parameterized by universal symbolic variables.

Universal Symbolic Variables : `door_switch`(β), `choice`(α)

Probabilistic Symbolic Variable : `car_door`(δ) $\in \{1, 2, 3\}$



Probabilistic Query : $\forall \alpha, \beta \ . \ Enc_{\psi} = \frac{2}{3}$ under the condition $\psi \triangleq \beta \wedge$ win

Results

Table 1: Performance metrics for each of the case studies.

	Timing (sec.)						
Case Study	**KLEE**	**Z3**	**Total**	**Lines**	**Paths**	**Samples**	**Concretizations**
Freivalds'	3	26	**29**	97	2	2	$n = 2$
Freivalds' (Multiple)	6	259	**265**	96	8	21	$(n, k) = (3, 7)$
Reservoir Sampling	14	98	**112**	52	127	6	$(n, k) = (13, 7)$
Reservoir Sampling	460	1	**461**	52	4096	12	$(n, k) = (13, 1)$
Monotone Testing	6	384	**390**	69	36	1	$n = 27$
Quicksort	14	114	**128**	65	120	10	$n = 5$
Bloom Filter	18	395	**413**	386	83	8	$(m, \varepsilon) = (3, 0.39)$
Count-min Sketch	4	145	**149**	245	3	8	$(n, \varepsilon, \gamma) = (4, 0.5, 0.25)$

- Further optimizing probabilistic symbolic execution.

Conclusion

- Further optimizing probabilistic symbolic execution.
- Analyzing more complex randomized programs.

Plinko (OOPSLA2 2022)

OOPSLA Talk Details

Symbolic Execution for Randomized Programs.

Zachary Susag, Sumit Lahiri, Justin Hsu, Subhajit Roy. OOPSLA2 2022

Please attend!

Fri 9 Dec 2022
16:00 - 16:30
Seminar Room G007

Conclusion & Future Directions

- *Almost* verification of programs with *unknown* oracles.

Conclusion & Future Directions

- *Almost* verification of programs with *unknown* oracles.
- Scaling our technique to *larger* randomized programs.

Conclusion & Future Directions

- *Almost* verification of programs with *unknown* oracles.
- Scaling our technique to *larger* randomized programs.
- Improving fuzzing techniques primed for *Almost* verification applications.

Thank You!

Ph.D. supported by **TCS Research**

Travel to NZ for SPLASH Conference generously supported by **TCS Research** & **ACM SIGPLAN**