Compiling & Controlling Symbolic Execution

Guannan Wei

with Songlin Jia, Ruiqi Gao, Haotian Deng,
Shangyin Tan, Oliver Bračevac, and Tiark Rompf

PurPL Seminar – Dec 2, 2022

Symbolic Execution

```
x = user_input()
y = user_input()
if (x > 5) {
    if (y < 10) {
        ...
    } else {
        ...
    }
} else {
    ...
}
```

Symbolic Execution

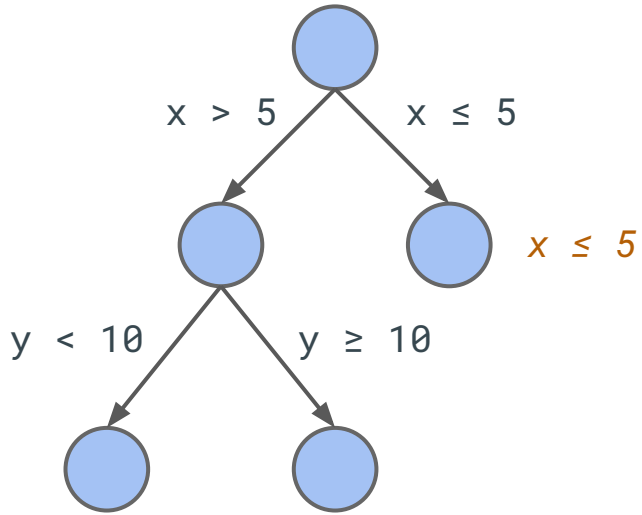
mark as symbolic

```
x = user_input()
y = user_input()
if (x > 5) {
    if (y < 10) {
        ...
    } else {
        ...
    }
} else {
    ...
}
```

Symbolic Execution

mark as symbolic

```
x = user_input()
y = user_input()
if (x > 5) {
    if (y < 10) {
        ... /* path 1 */
    } else {
        ... /* path 2 */
    }
} else {
    ... /* path 3 */
}
```



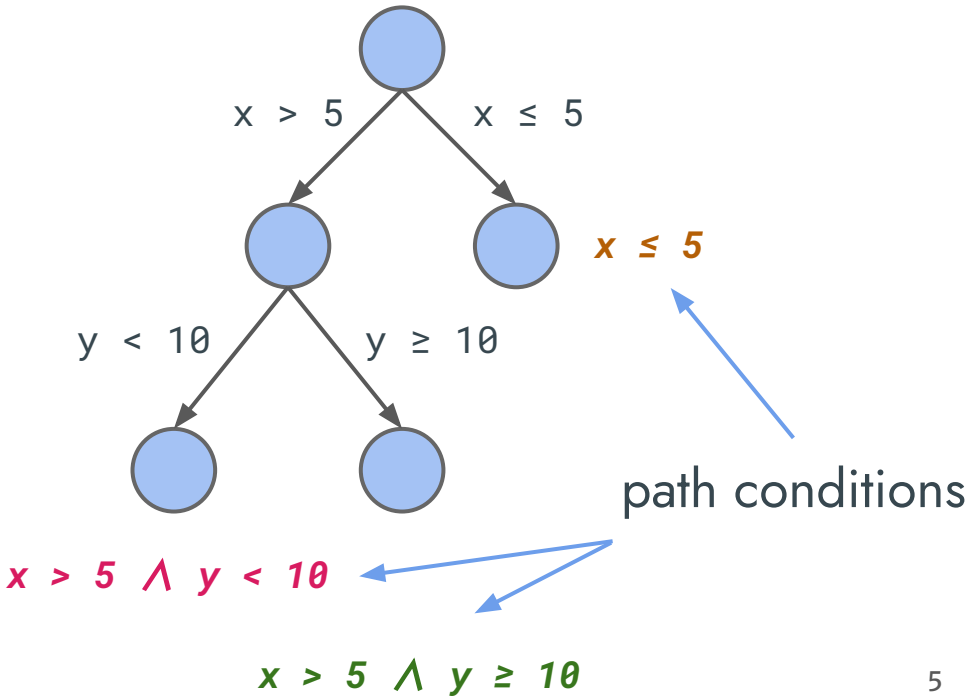
$x > 5 \land y < 10$

$x > 5 \land y \geq 10$

Symbolic Execution

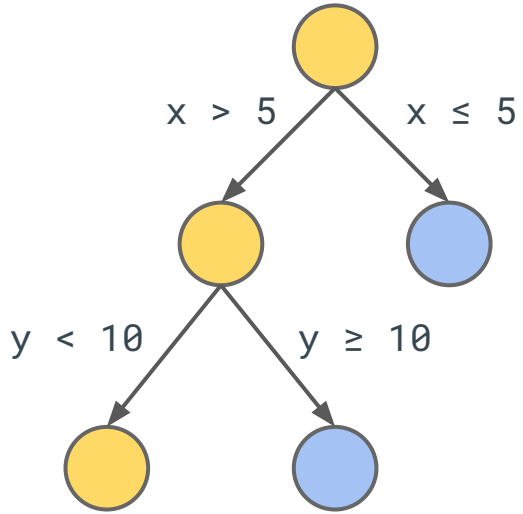
mark as symbolic

```
x = user_input()
y = user_input()
if (x > 5) {
    if (y < 10) {
        ... /* path 1 */
    } else {
        ... /* path 2 */
    }
} else {
    ... /* path 3 */
}
```



Symbolic Execution

```
x = user_input()
y = user_input()
if (x > 5) {
    if (y < 10) {
        ... /* path 1 */
    } else {
        ... /* path 2 */
    }
} else {
    ... /* path 3 */
}
```



solver(x > 5 ∧ y < 10) = { x = 6, y = 9 }

Symbolic Execution

- automatic test case generation
- bug finding and exploit generation
- program verification
- worst-case execution time analysis
- ...

Symbolic Execution Engine

a symbolic interpreter `sym-eval`

- simulates the execution nondeterministically
- records the condition of each path

```
sym-eval(prog) = {p1, p2, p3, ...}
```

Path Explosion

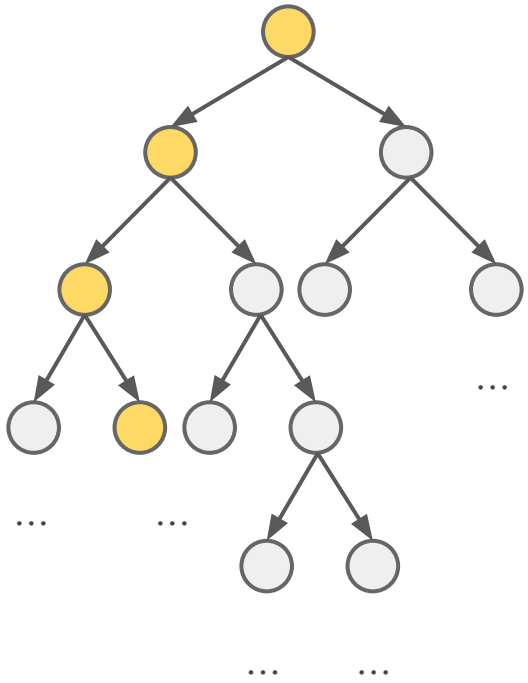
Concrete Execution

1 path

vs

Symbolic Execution

exponential number of independent paths



Path Explosion

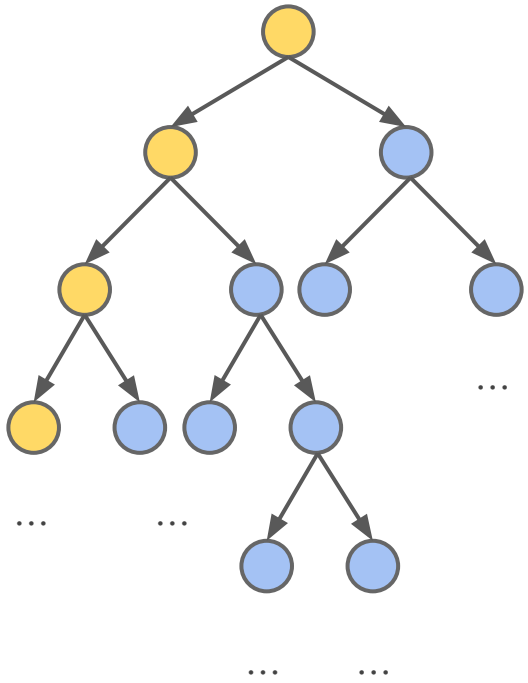
Concrete Execution

1 path

vs

Symbolic Execution

exponential number of independent paths



Path Explosion

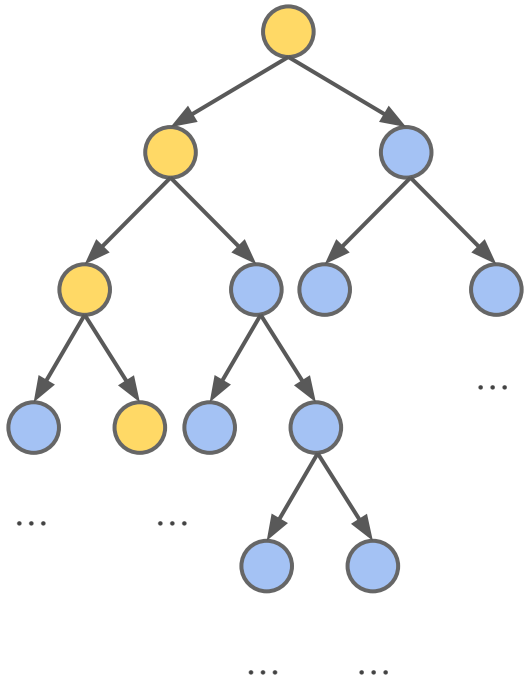
Concrete Execution

1 path

vs

Symbolic Execution

exponential number of independent paths



Path Explosion

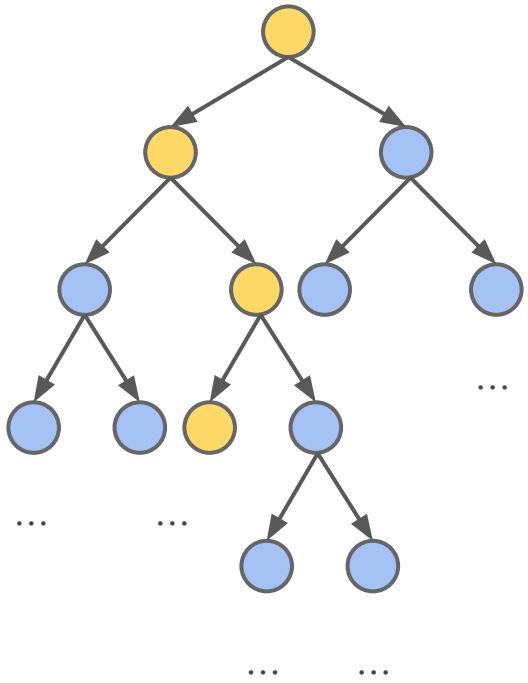
Concrete Execution

1 path

vs

Symbolic Execution

exponential number of independent paths



Path Explosion

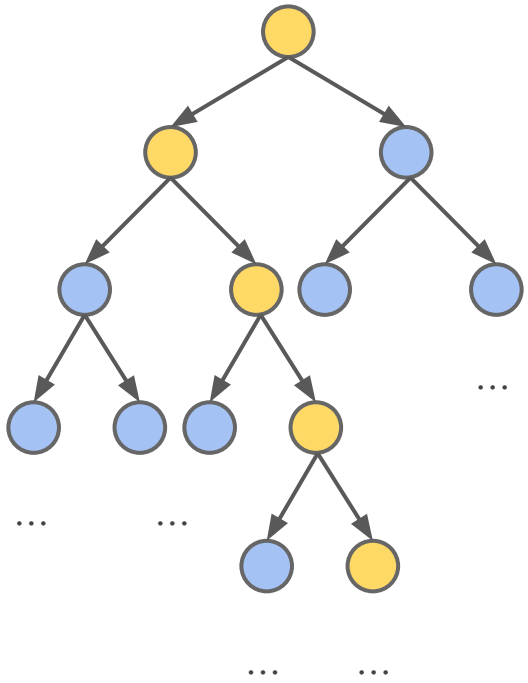
Concrete Execution

1 path

vs

Symbolic Execution

exponential number of independent paths



Path Explosion

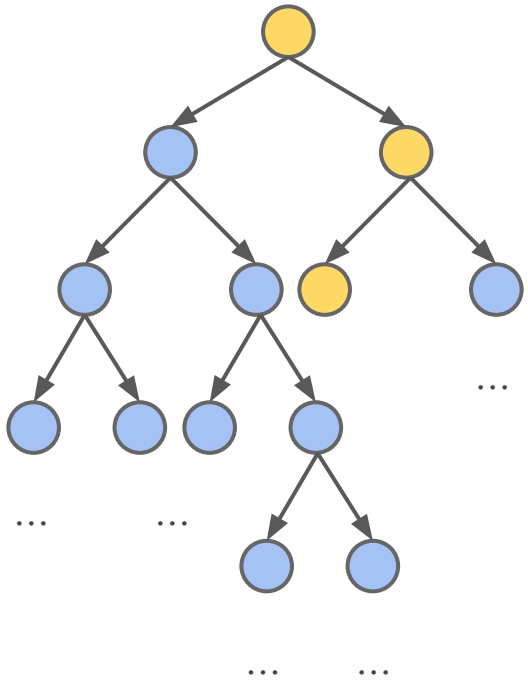
Concrete Execution

1 path

vs

Symbolic Execution

exponential number of independent paths



performance matters

Performance Matters

```
sym-eval(prog) = {p1, p2, p3, ...}
```


Data from Qsym: A practical concolic execution engine tailored for hybrid fuzzing. Yun et al., USENIX Security, 2018.

Performance Matters

```
sym-eval(prog) = {p1, p2, p3, ...}
```

symbolic interpreter performance
compared to native execution

KLEE (C++)	3,000x slower
angr (Python)	321,000x slower

Data from Qsym: A practical concolic execution engine tailored for hybrid fuzzing. Yun et al., USENIX Security, 2018.

Performance Matters

```
sym-eval(prog) = {p1, p2, p3, ...}
```

interpretation overhead

- inspecting program AST/IR
- dispatching the semantics
- recursion at meta-level

Data from Qsym: A practical concolic execution engine tailored for hybrid fuzzing. Yun et al., USENIX Security, 2018.

“To be relevant as a verification tool, symbolic execution must be compiled.”

— PEPM ‘22 Anonymous Reviewer

Symbolic-Execution Compilers

Recall that a symbolic interpreter produces path conditions:

```
sym-eval(prog) = {p1, p2, p3, ...}
```

An SE compiler converts an input program to another program:

```
sym-comp(prog) = target
```

Symbolic-Execution Compilers

Recall that a symbolic interpreter produces path conditions:

```
sym-eval(prog) = {p1, p2, p3, ...}
```

An SE compiler converts an input program to another program:

```
sym-comp(prog) = target
```

Running `target` performs the symbolic execution *w/o interpretation overhead*:

```
⟦target⟧ = {p1, p2, p3, ...}
```

From Interpreters to Compilers

Futamura Projection: deriving the
symbolic-execution compiler `sym-comp`
from the symbolic interpreter `sym-eval`
via partial evaluation or staging

LLSC: A Parallel Symbolic Execution Compiler for LLVM IR (FSE 2021 Demo)
Guannan Wei, Shangyin Tan, Oliver Bračevac, Tiark Rompf

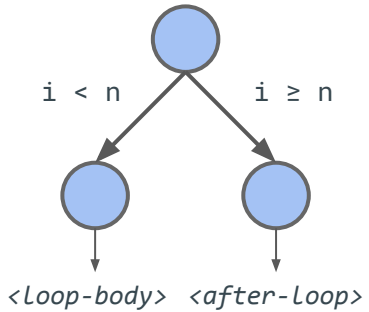
Compiling Symbolic Execution with Staging and Algebraic Effects (OOPSLA 2020)
Guannan Wei, Oliver Bračevac, Shangyin Tan, Tiark Rompf

Path Explosion, Worse

```
n = user_input()
while (i < n) {
    <loop-body>
}
<after-loop>
```

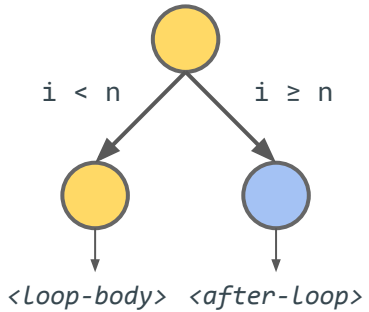
Path Explosion, Worse

```
n = user_input()
while (i < n) {
    <loop-body>
}
<after-loop>
```



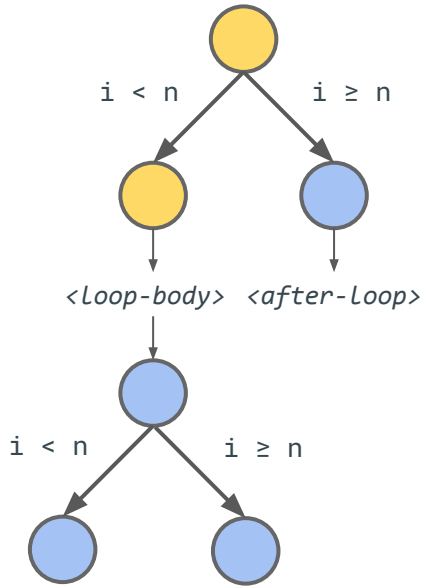
Path Explosion, Worse

```
n = user_input()
while (i < n) {
    <loop-body>
}
<after-loop>
```



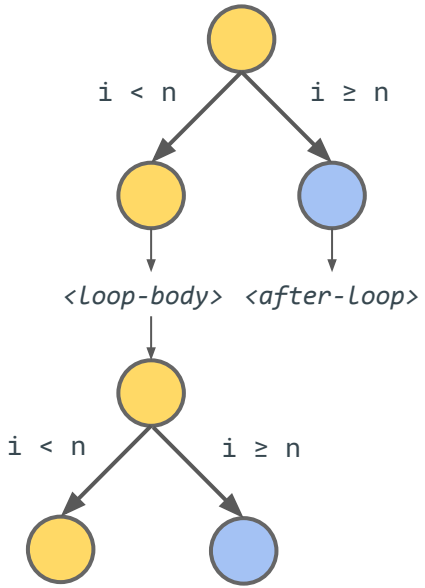
Path Explosion, Worse

```
n = user_input()
while (i < n) {
    <loop-body>
}
<after-loop>
```



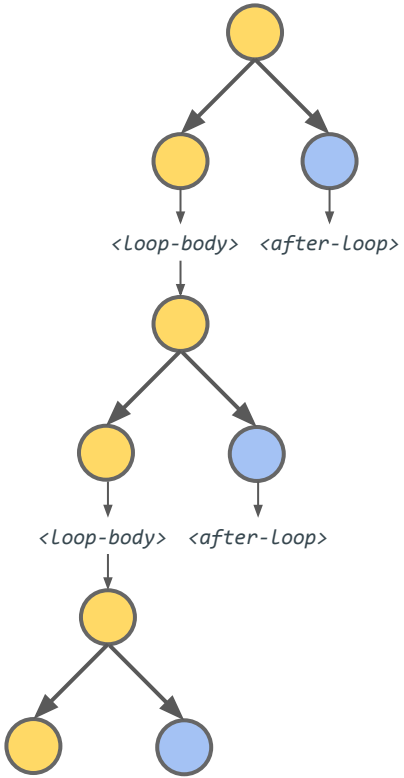
Path Explosion, Worse

```
n = user_input()
while (i < n) {
    <loop-body>
}
<after-loop>
```



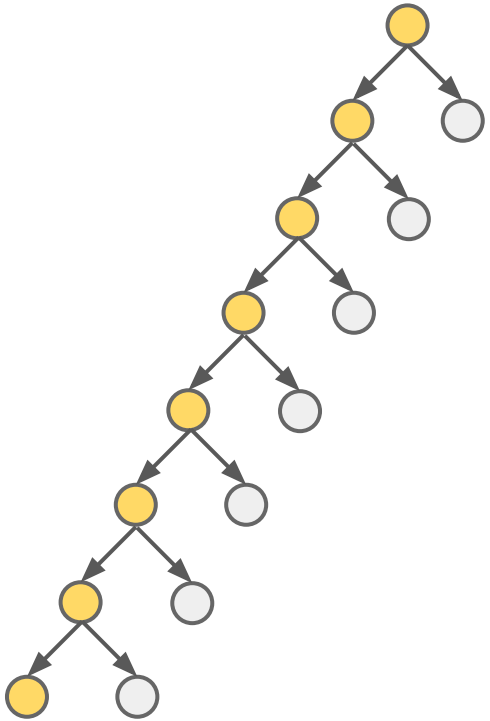
Path Explosion, Worse

```
n = user_input()
while (i < n) {
    <loop-body>
}
<after-loop>
```



Path Explosion, Worse

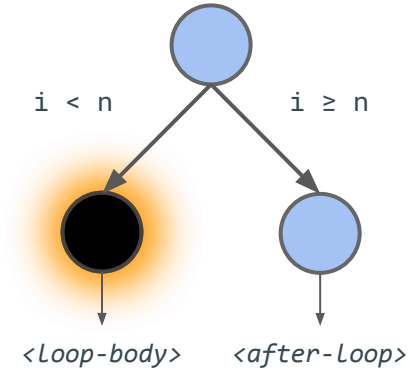
```
n = user_input()
while (i < n) {
    <loop-body>
}
<after-loop>
```



. . .

Path Explosion, Worse

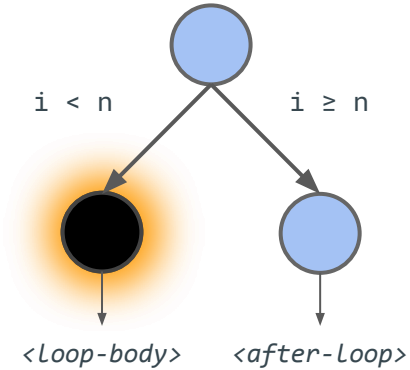
```
n = user_input()
while (i < n) {
    <loop-body>
}
<after-loop>
```



once running into the black hole,
we cannot effectively explore other parts of the program

Escaping the Black Hole

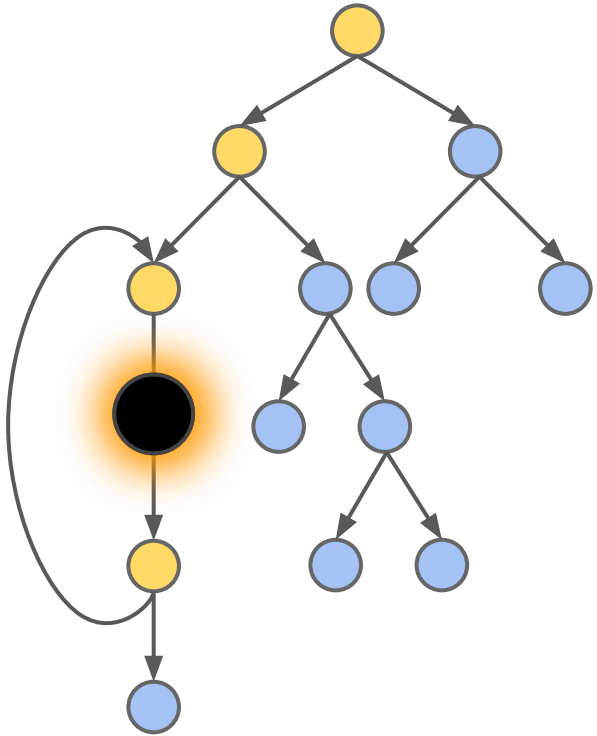
```
n = user_input()
while (i < n) {
    <loop-body>
}
<after-loop>
```



modern symbolic execution engines deploys various path selection heuristics

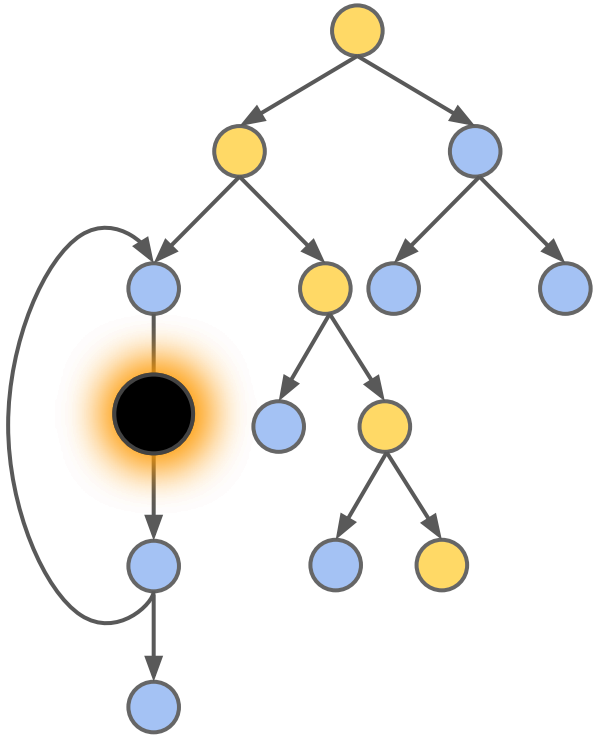
Escaping the Black Hole

- random state/path selection
- coverage-guided heuristics
- …



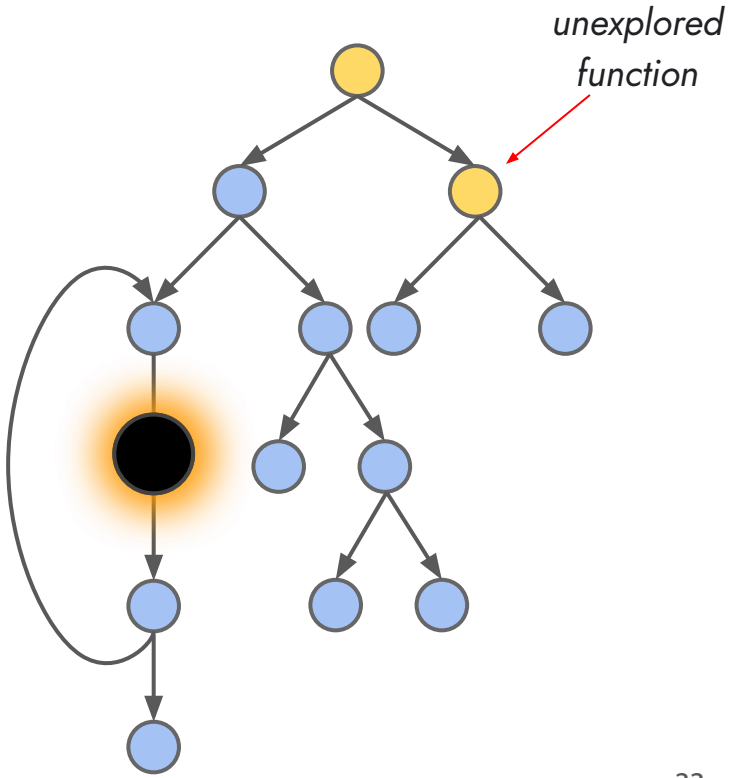
Escaping the Black Hole

- *random state/path selection*
- coverage-guided heuristics
- …



Escaping the Black Hole

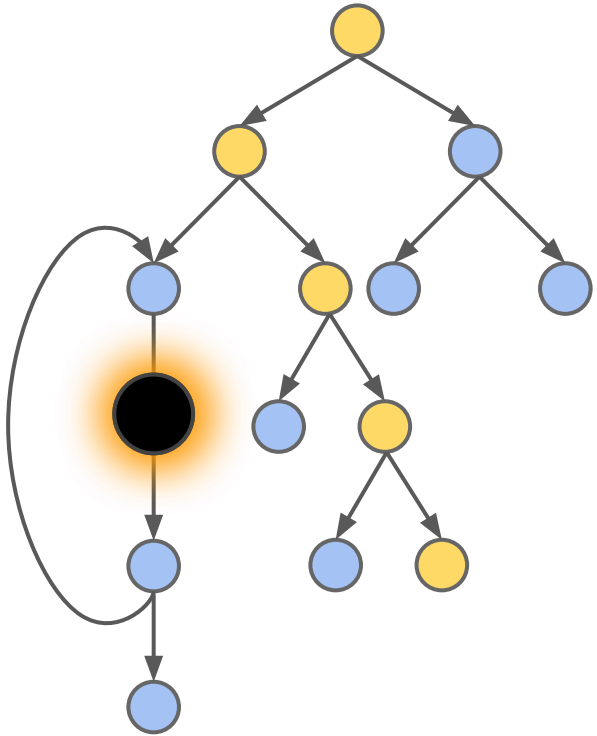
- random state/path selection
- *coverage-guided heuristics*
- …



Escaping the Black Hole

- random state/path selection
- coverage-guided heuristics
- …

Deploying path selection strategies needs the ability to *pause* and *resume* the execution of paths.



To efficiently execute and effectively explore the program,

compiled symbolic execution must be controlled.

To efficiently execute and effectively explore the program,

compiled symbolic execution must be controlled.

Solution:

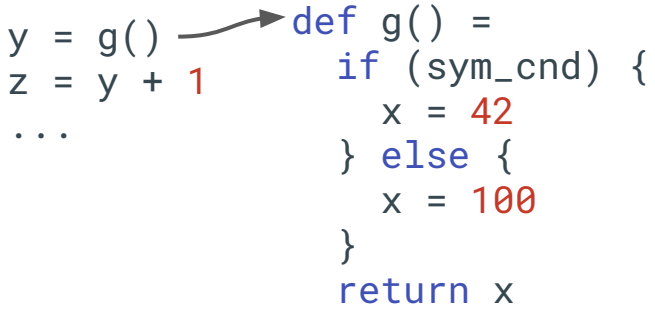
compile symbolic execution with continuations

Making Control Explicitly

represent the rest of execution as a function k *in the generated code*

Making Control Explicitly

represent the rest of execution as a function k *in the generated code*

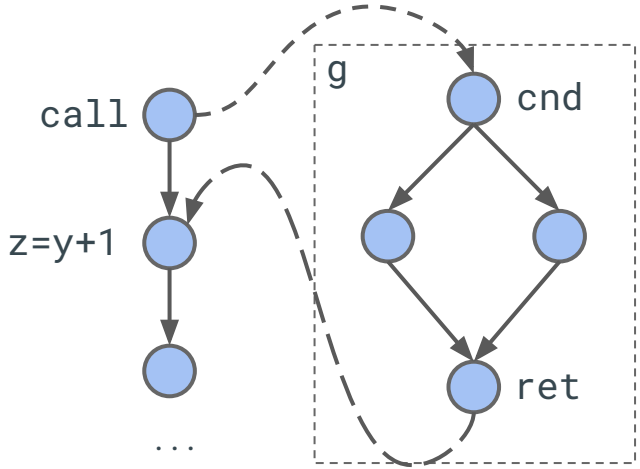


Making Control Explicitly

represent the rest of execution as a function *k* *in the generated code*

```
y = g()
z = y + 1
...
```

```
def g() =
  if (sym_cnd) {
    x = 42
  } else {
    x = 100
  }
  return x
```



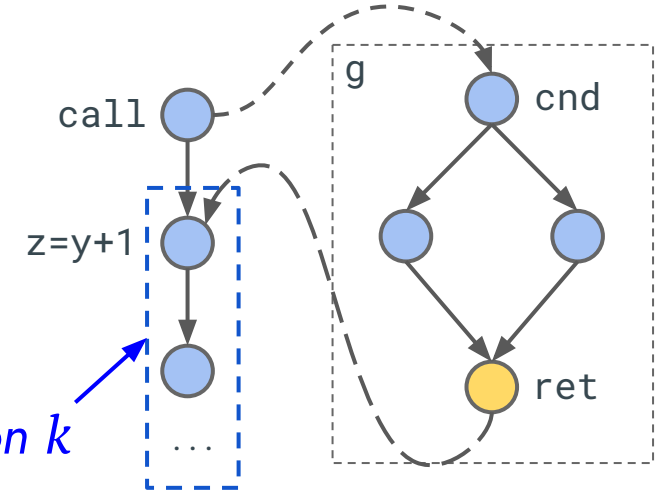
static control-flow graph

Making Control Explicitly

represent the rest of execution as a function *k* *in the generated code*

```
y = g()
z = y + 1
...
```

```
def g() =
  if (sym_cnd) {
    x = 42
  } else {
    x = 100
  }
  return x
```



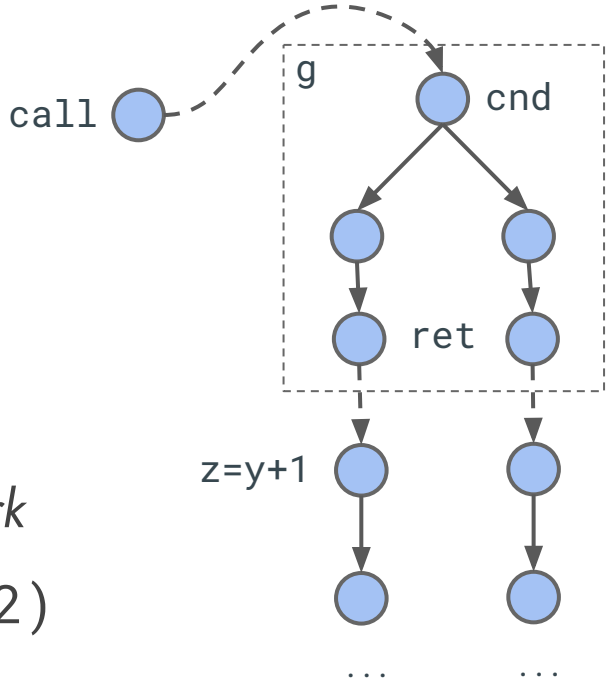
static control-flow graph

Making Control Explicitly

represent the rest of execution as a function k *in the generated code*

```
y = g()
z = y + 1
...
```

```
def g() =
  if (sym_cnd) {
    x = 42
  } else {
    x = 100
  }
  return x
```

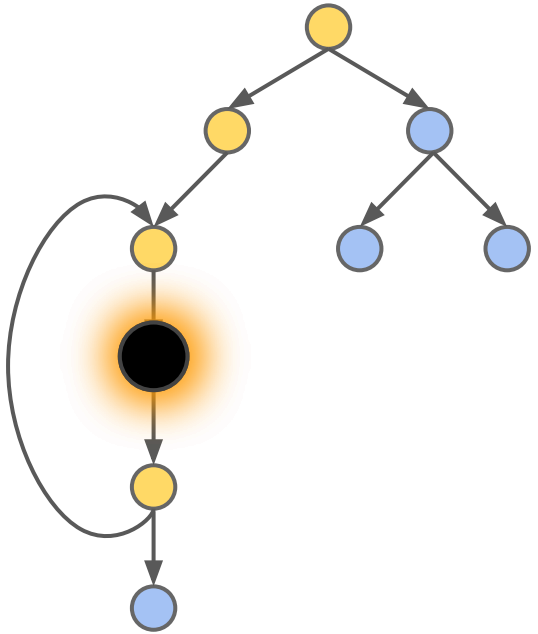


invoke and fork

$k(s1);\ k(s2)$

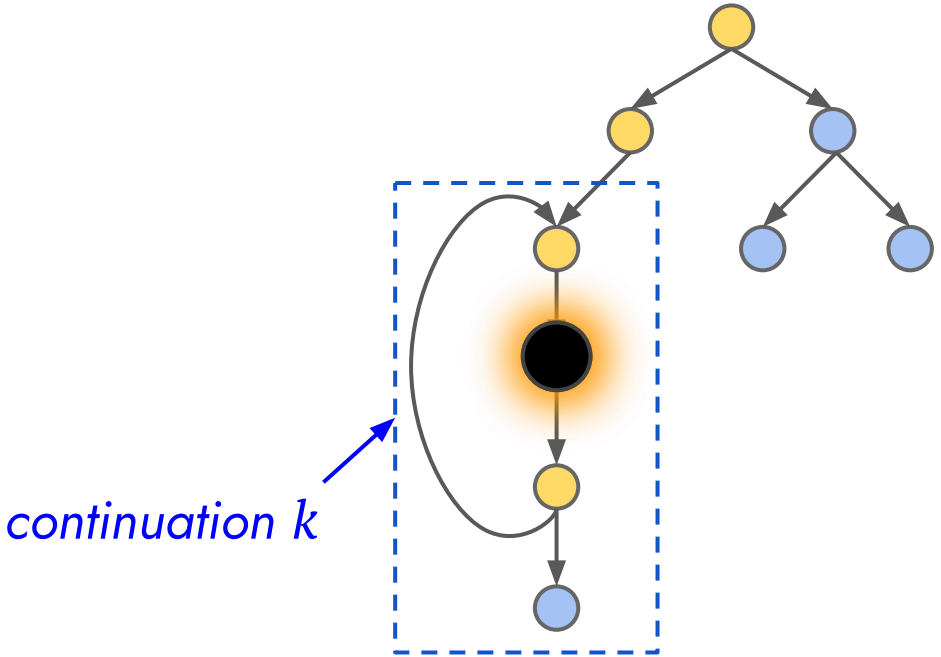
Making Control Explicitly

represent the rest of execution as a function *k* *in the generated code*



Making Control Explicitly

represent the rest of execution as a function *k* *in the generated code*

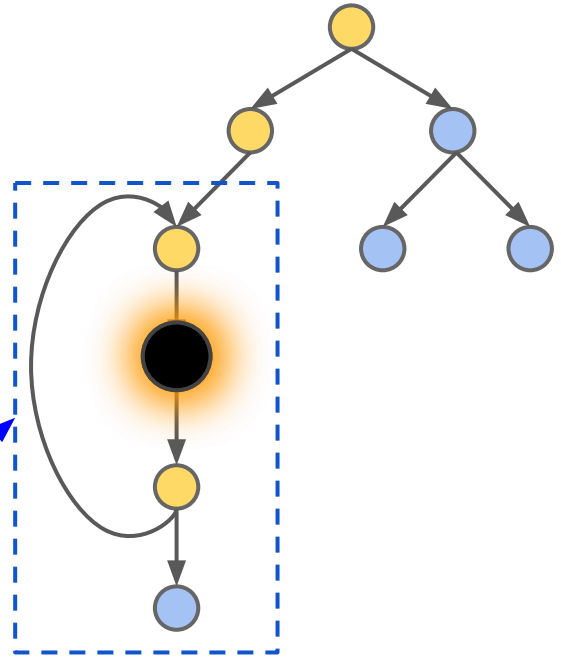


Making Control Explicitly

represent the rest of execution as a function *k* *in the generated code*

save and pause

```
scheduler.put(() => k(s))
```

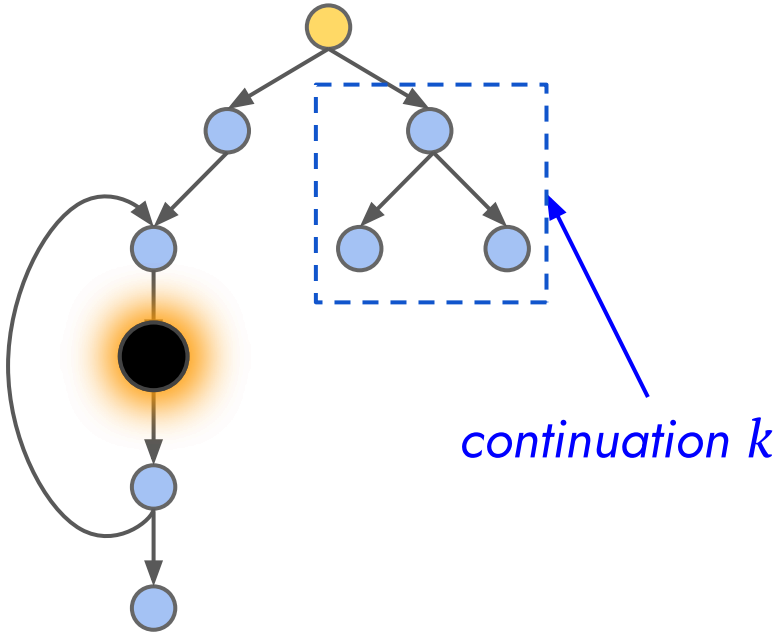


Making Control Explicitly

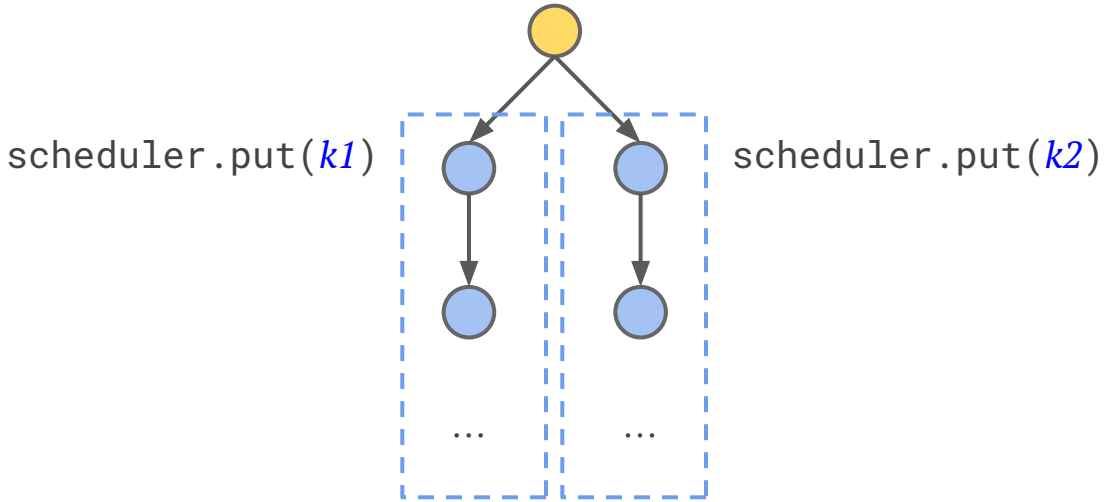
represent the rest of execution as a function k *in the generated code*

dispatch and resume

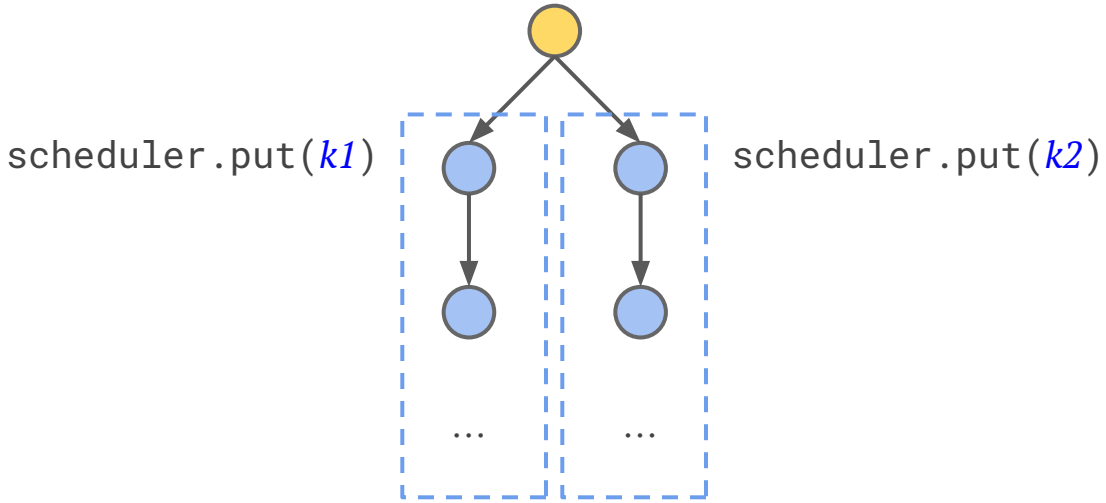
```
k = scheduler.get(); k()
```



Parallelism for Free



Parallelism for Free



thread pool

```
thread() {
  k = scheduler.get(); k()
}
```

Making Control Explicitly

represent the rest of execution as a function k *in the generated code*

- *invoke and fork*
  `k(s1); k(s2)`
- *save and pause*
  `scheduler.put(() => k(s))`
- *dispatch and resume*
  `k = scheduler.get(); k()`
- *dispatch in parallel*

Making Control Explicitly

represent the rest of execution as a function k *in the generated code*

How?

specializing a symbolic interpreter
that itself is written in continuation-passing style

```
def staged-sym-eval(p: Prog, k: Rep[State] => Rep[Unit]): Rep[Unit]
```

GenSym

compiling the symbolic execution of LLVM IR

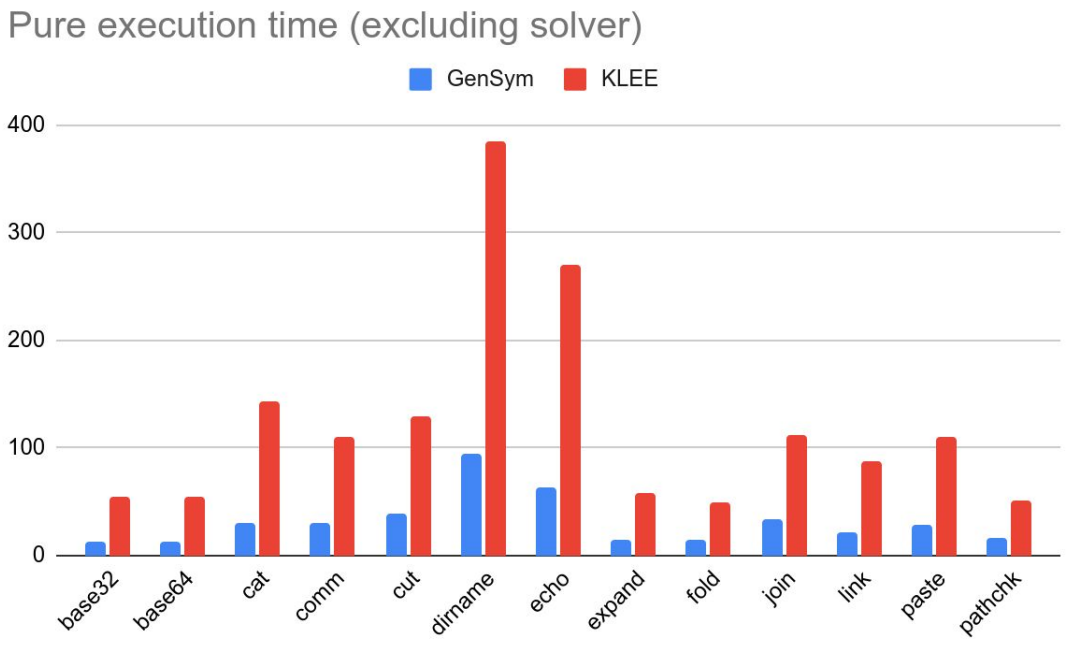
into continuation-passing style in C++

Compare GenSym with KLEE

- KLEE: state-of-the-art symbolic interpreter for LLVM IR

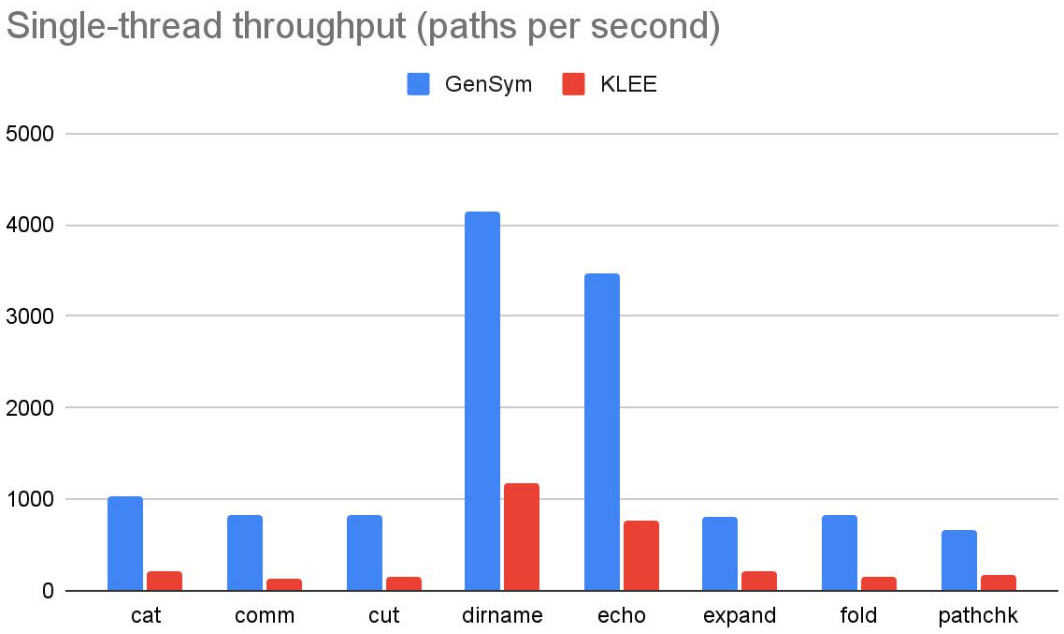
  developed since 2008
- Evaluated on a set of GNU Coreutils programs

Single-thread Pure Execution

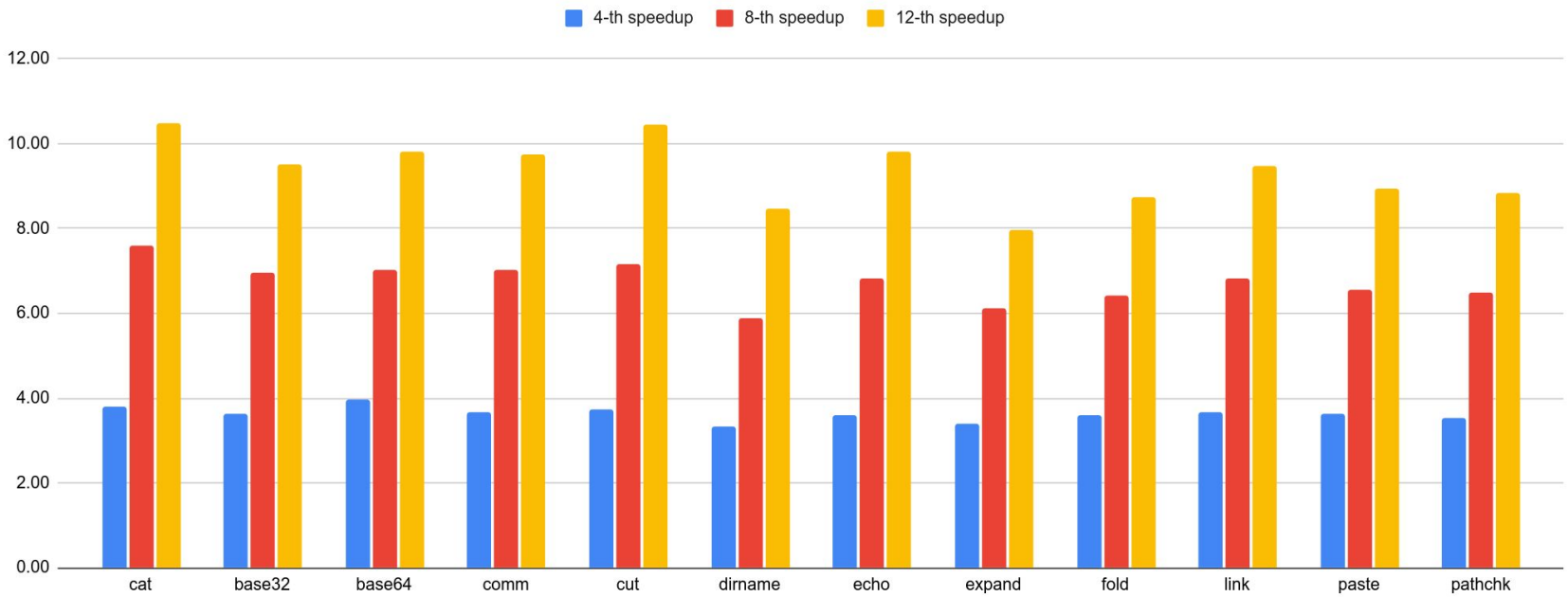


average ~4x speedups

Single-thread 1-Hour Throughput



Parallel Execution Efficiency



compiling *symbolic execution to* *continuation-passing style* *to build high-performance and parallel symbolic execution engine*

- ✦ semantics-based compilation
- ✦ zero interpretation overhead
- ✦ effectively express concurrency/parallelism
- ✦ support arbitrary path-selection heuristic

- ✦ GenSym outperforms state-of-the-art tools

`https://continuation.passing.style/GenSym`