Compiling & Controlling Symbolic Execution

Guannan Wei

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

Midwest PL Summit – Oct 6 2023 – University of Michigan

PURPL

PURDUE UNIVERSITY

Symbolic Execution

```
x = user_input()
y = user_input()

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

Symbolic Execution

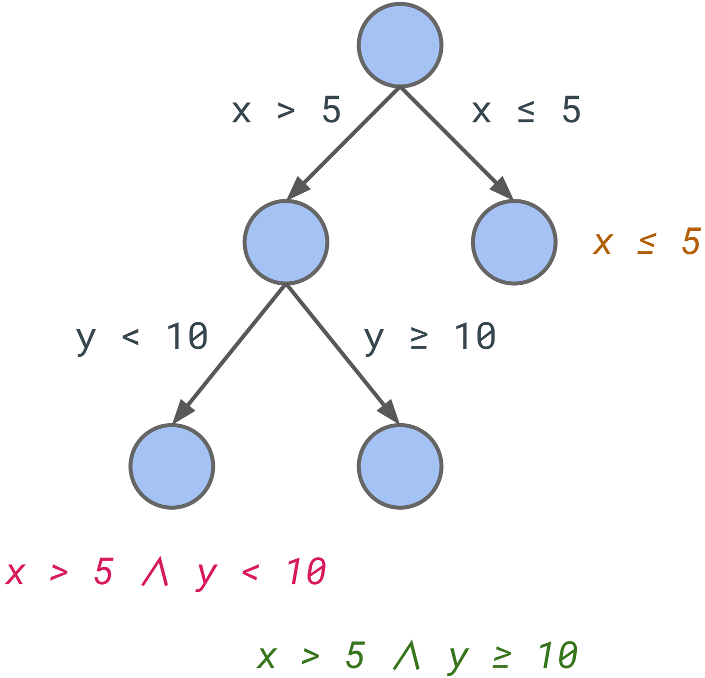
mark as symbolic

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

Symbolic Execution

mark as symbolic

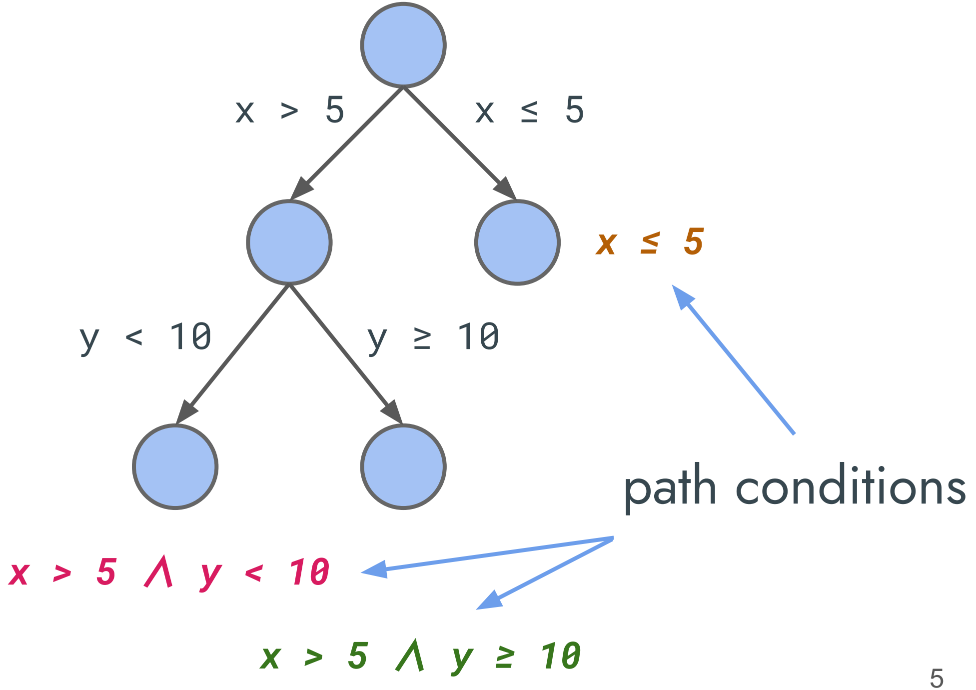
```
x = user_input()
y = user_input()
if (x > 5) {
    if (y < 10) {
        ... /* path 1 */
    } else {
        ... /* path 2 */
    }
} else {
    ... /* path 3 */
}
```



Symbolic Execution

mark as symbolic

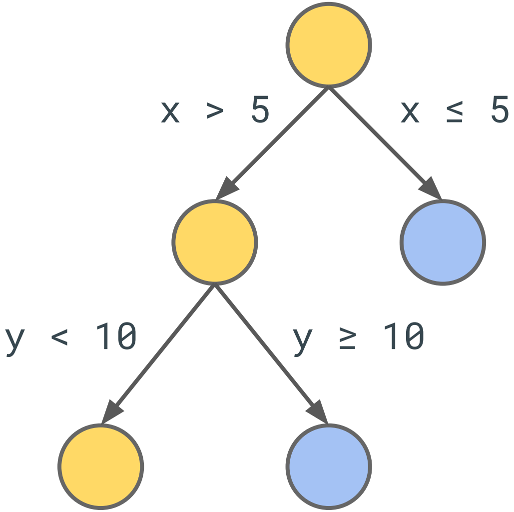
```
x = user_input()
y = user_input()
if (x > 5) {
    if (y < 10) {
        ... /* path 1 */
    } else {
        ... /* path 2 */
    }
} else {
    ... /* path 3 */
}
```



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 – Applications

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

Symbolic Execution Engine

a concrete interpreter `eval: Prog → (Value, State)`

- simulates the execution deterministically

Symbolic Execution Engine

a *symbolic* interpreter $\texttt{eval}_{\texttt{sym}}$`: Prog → Set[(Value, State, PC)]`

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

Path Explosion

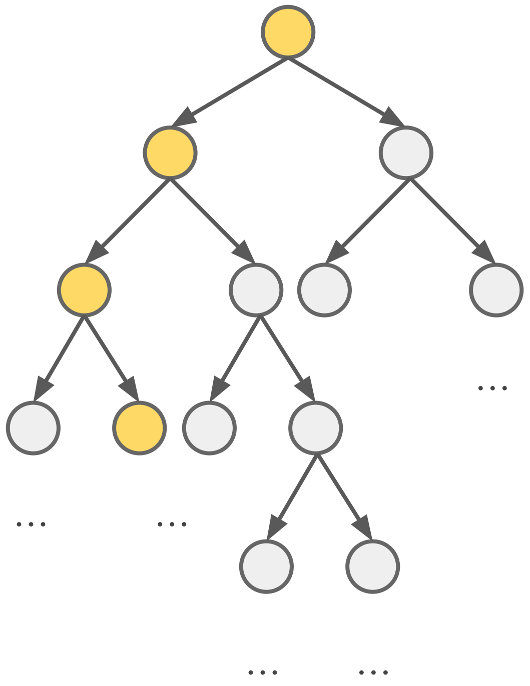
Concrete Execution

1 path

vs

Symbolic Execution

exponential number of independent paths



Path Explosion

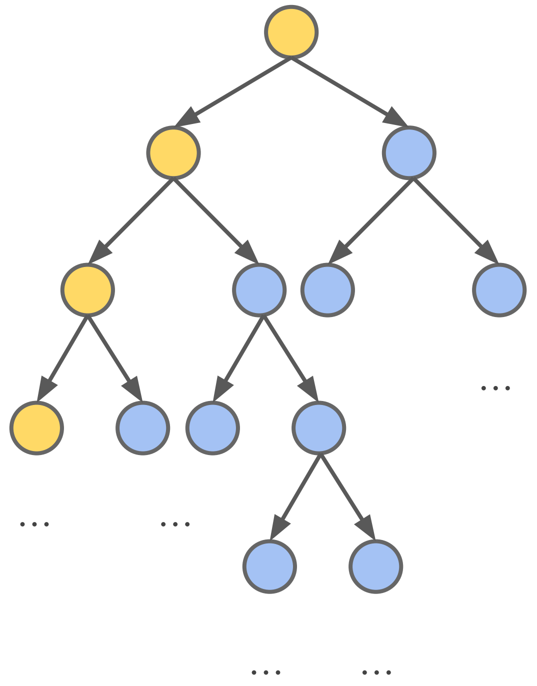
Concrete Execution

1 path

vs

Symbolic Execution

exponential number of independent paths



Path Explosion

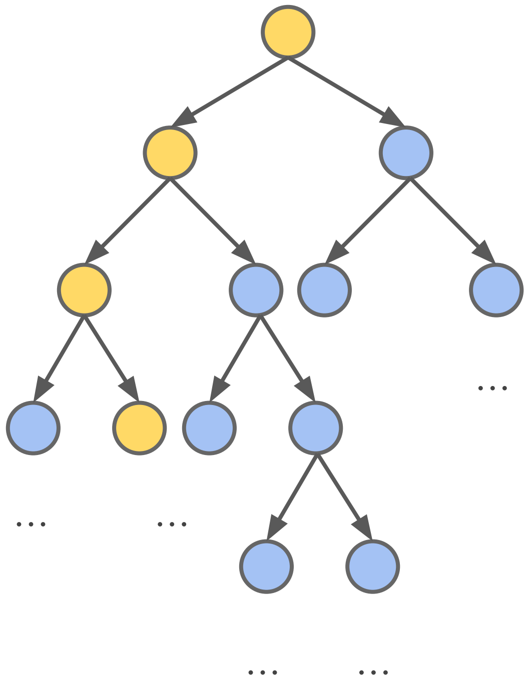
Concrete Execution

1 path

vs

Symbolic Execution

exponential number of independent paths



Path Explosion

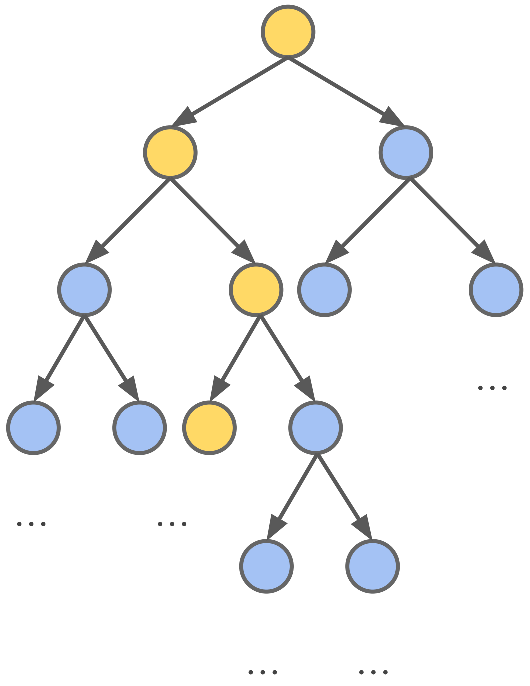
Concrete Execution

1 path

vs

Symbolic Execution

exponential number of independent paths



Path Explosion

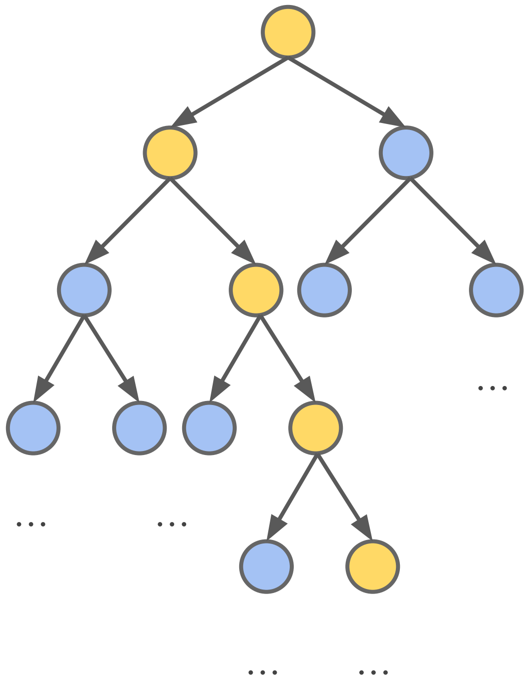
Concrete Execution

1 path

vs

Symbolic Execution

exponential number of independent paths



Path Explosion

Concrete Execution

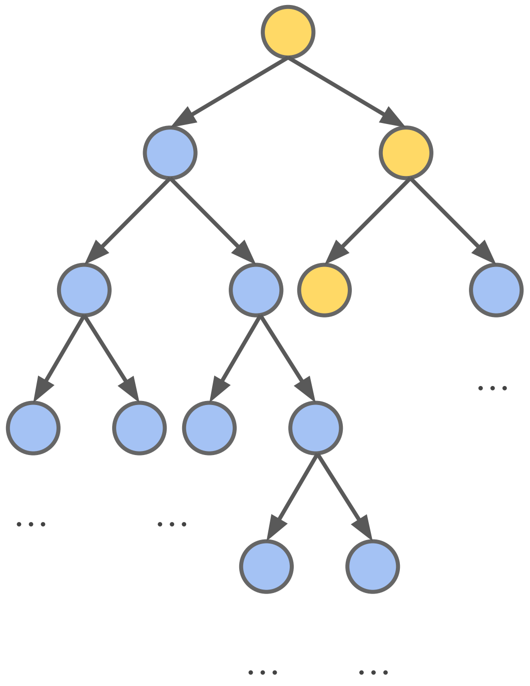
1 path

vs

Symbolic Execution

exponential number of independent paths

performance matters



Performance Matters

$\texttt{eval}_{\texttt{sym}}\texttt{: Prog} \rightarrow \texttt{Set[(Value, State, PC)]}$

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

$\texttt{eval}_{\texttt{sym}} : \texttt{Prog} \rightarrow \texttt{Set[(Value, State, PC)]}$

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.

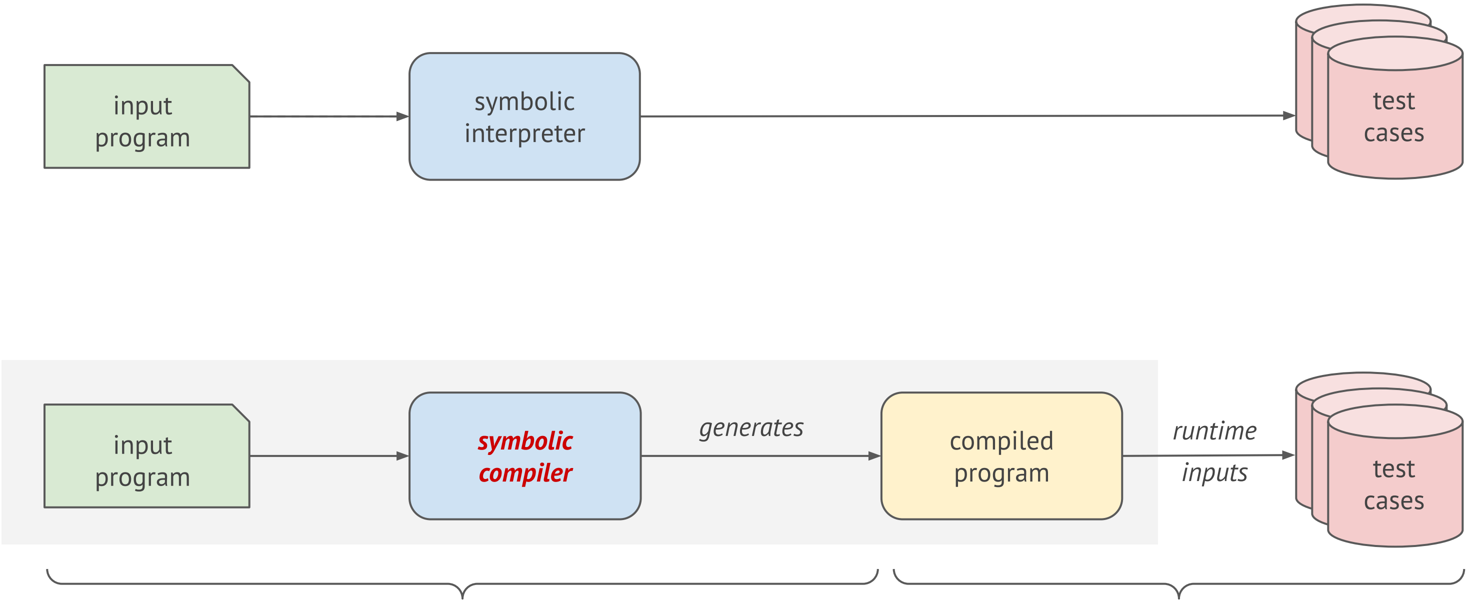
Symbolic-Execution Compilers

To remove these overheads,

compilation is inevitable.

Symbolic-Execution Compilers

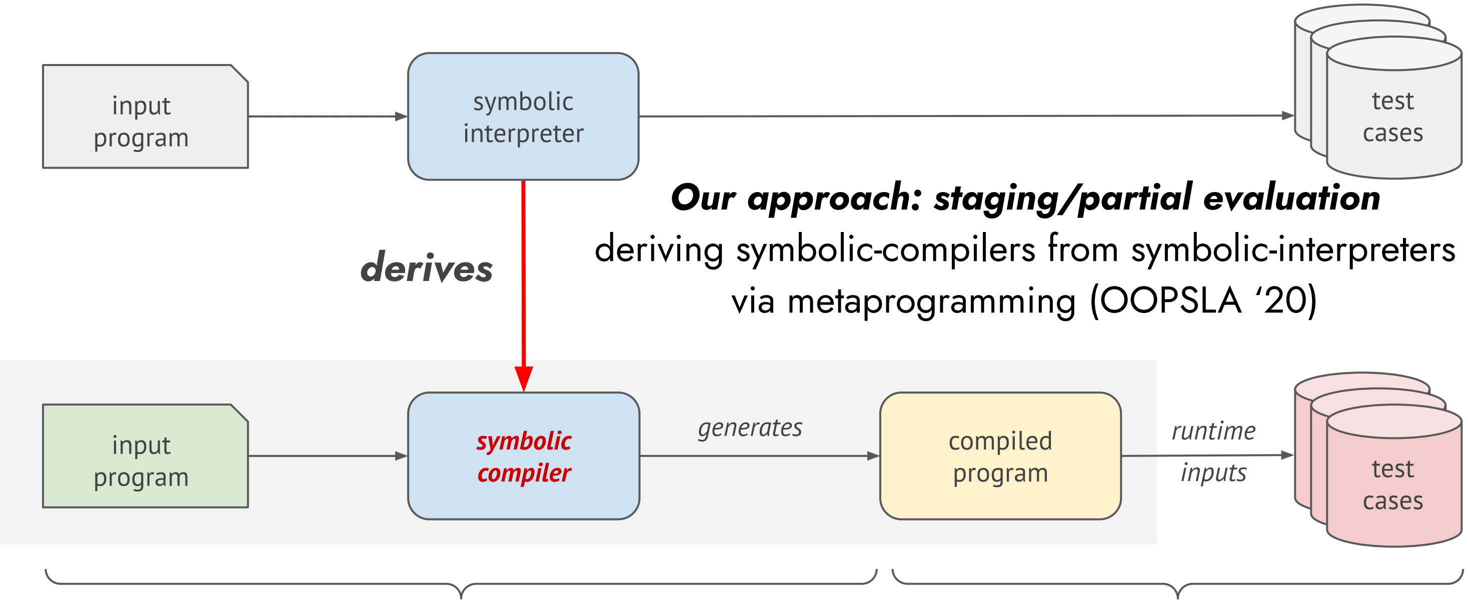
Symbolic-Execution Compilers



compilation removes interpretation overhead, optimizes programs, etc.

runs faster

Symbolic-Execution Compilers



compilation removes interpretation overhead, optimizes programs, etc.

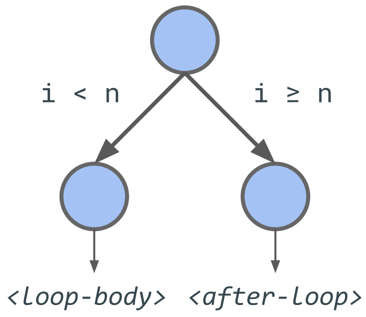
runs faster

Path Explosion, Worse

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

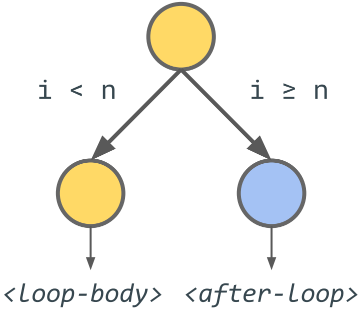
Path Explosion, Worse

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



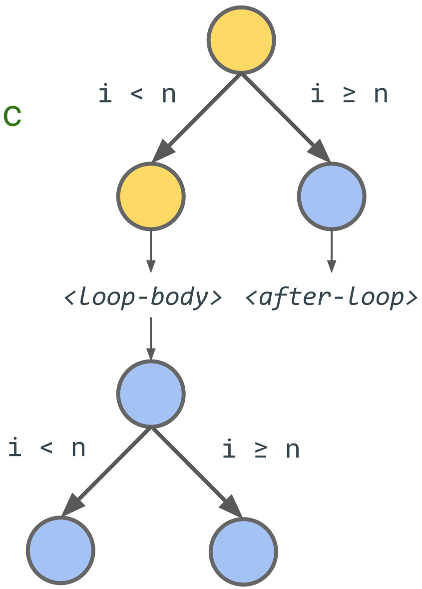
Path Explosion, Worse

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



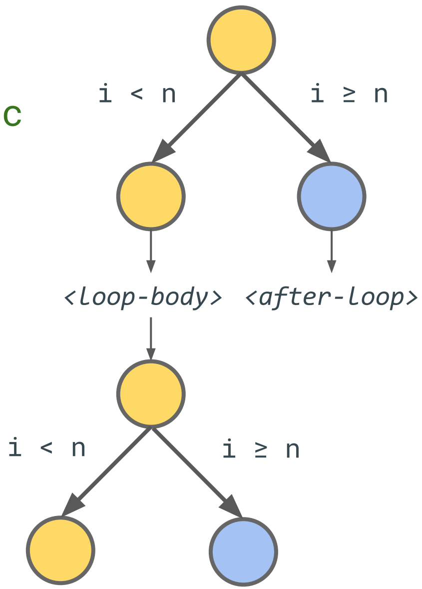
Path Explosion, Worse

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



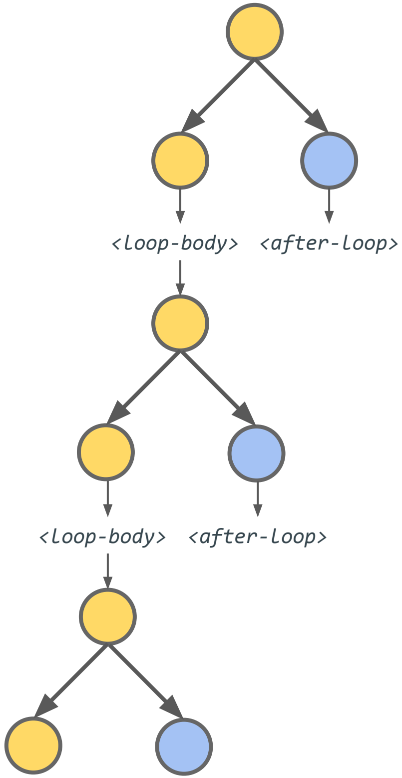
Path Explosion, Worse

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



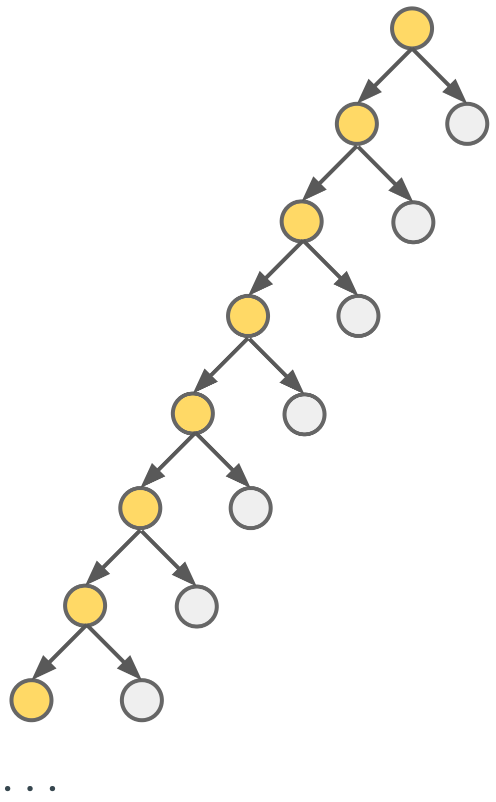
Path Explosion, Worse

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



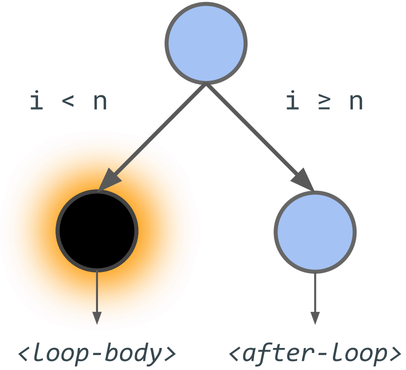
Path Explosion, Worse

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



Path Explosion, Worse

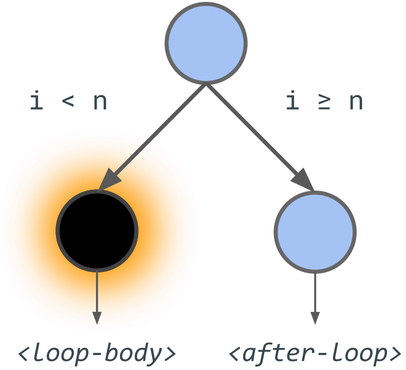
```
n = user_input() // i.e. symbolic
while (i < n) {
    <loop-body>
}
<after-loop>
```



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

Escaping the Black Hole

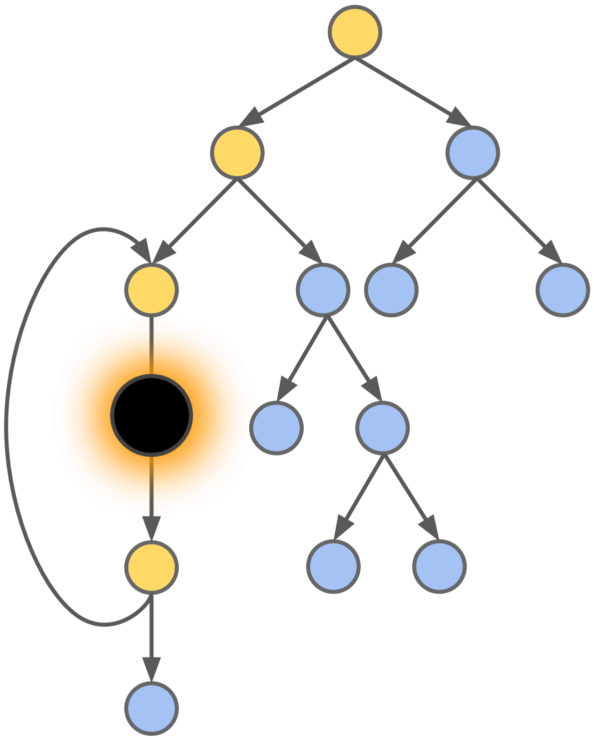
```
n = user_input() // i.e. symbolic
while (i < n) {
    <loop-body>
}
<after-loop>
```



Traditional wisdom: deploys clever path selection heuristics

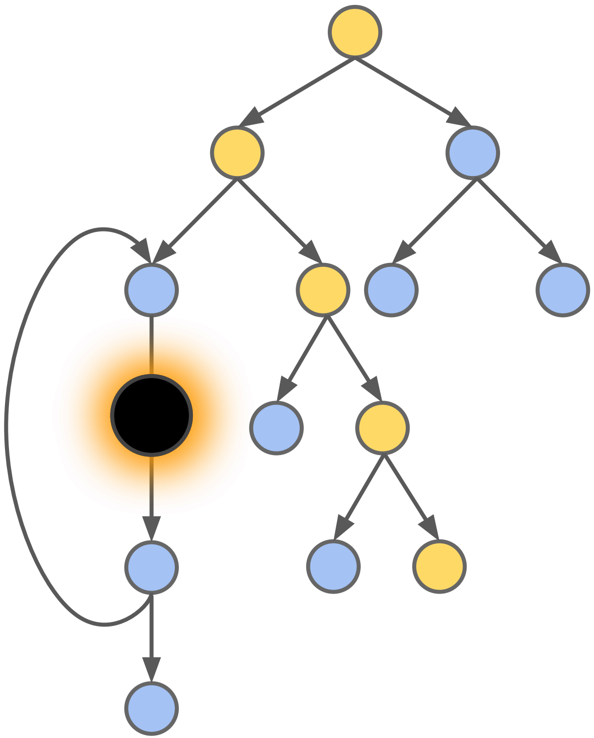
Escaping the Black Hole

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



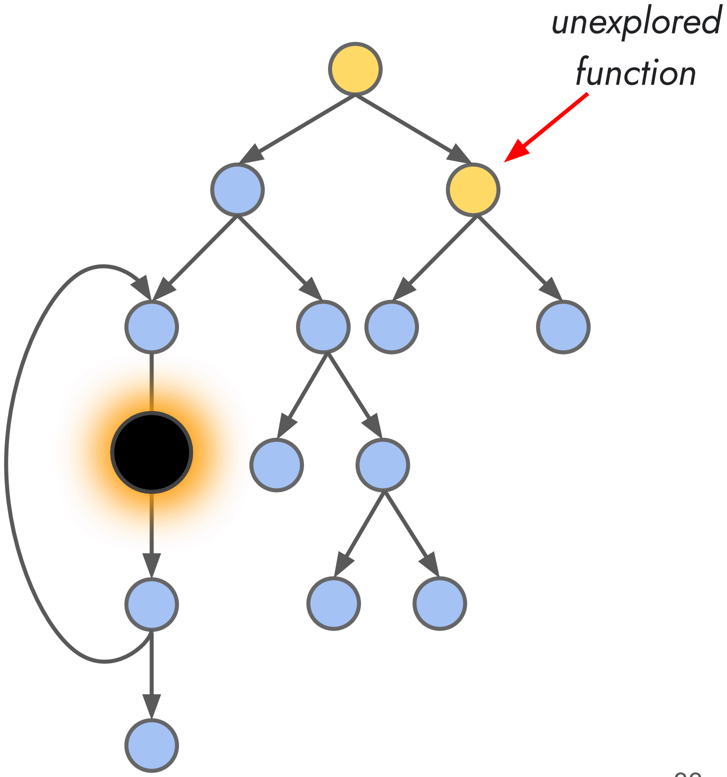
Escaping the Black Hole

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



Escaping the Black Hole

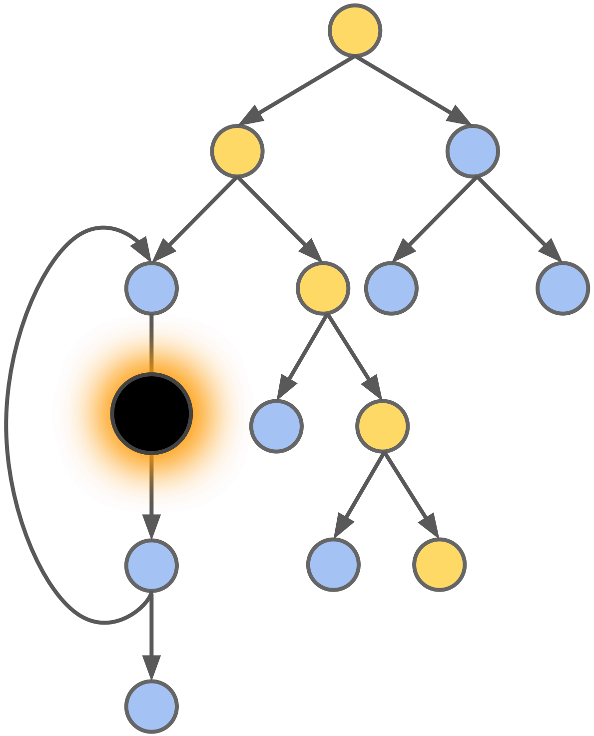
- random state/path selection
- *coverage-guided heuristics*
- ...



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.

How can we do that without an external interpreter/engine to control the execution?

To efficiently execute and effectively explore the program, compiled symbolic execution must be controlled.

How can we do that without an external interpreter/engine to control the execution?

***Solution**: Compile with continuations, enabling the program to "control" itself.*

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
...
```

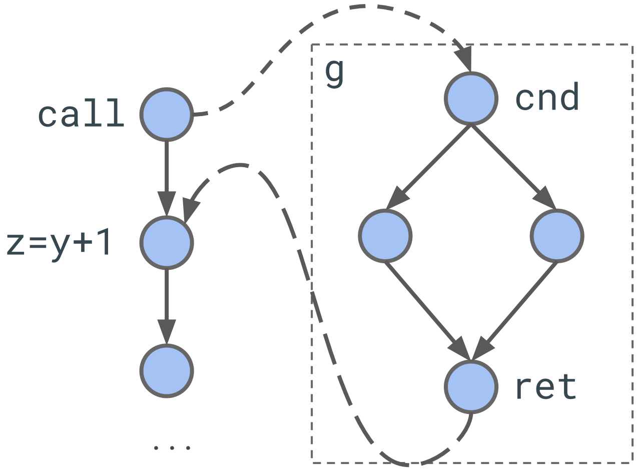
```
def g() =
  if (sym_cnd) {
    x = 42
  } else {
    x = 100
  }
  return x
```

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
```



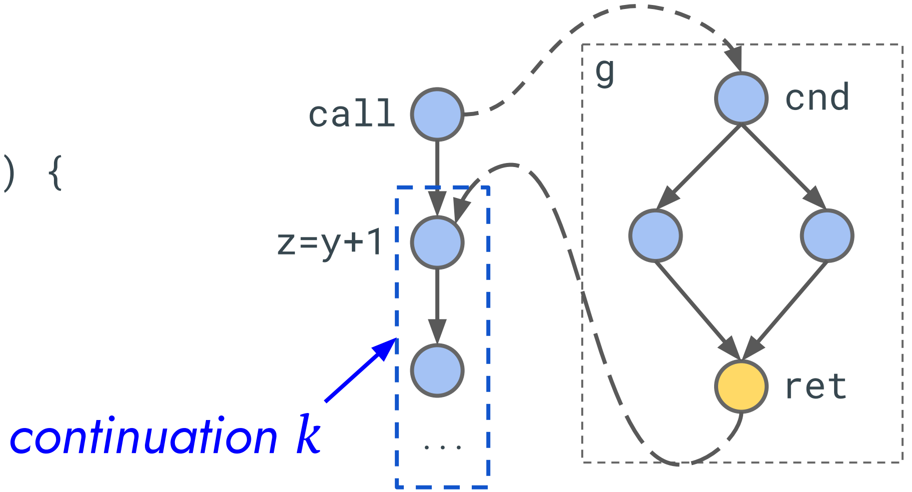
static control-flow graph

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
```



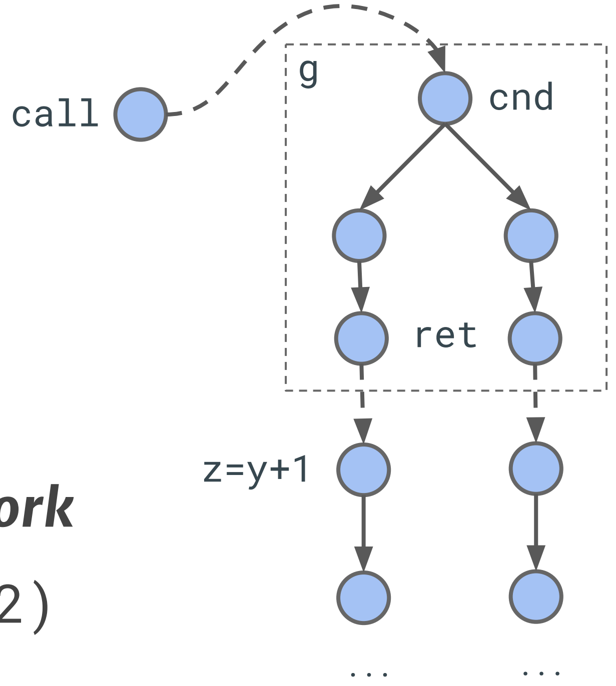
static control-flow graph

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
```

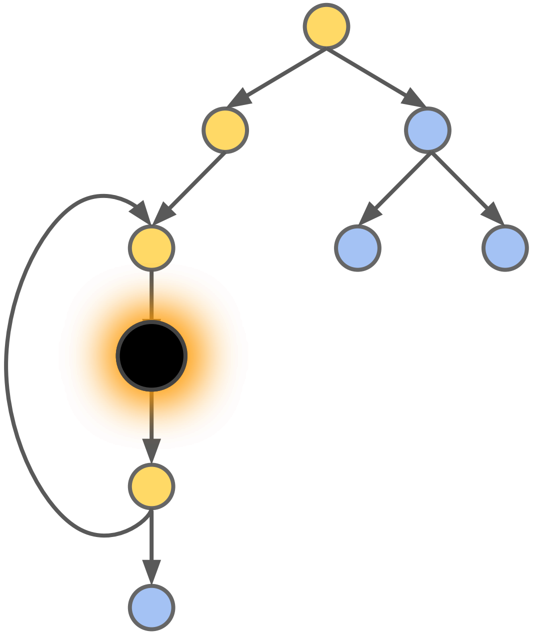


invoke and fork

k(s1); k(s2)

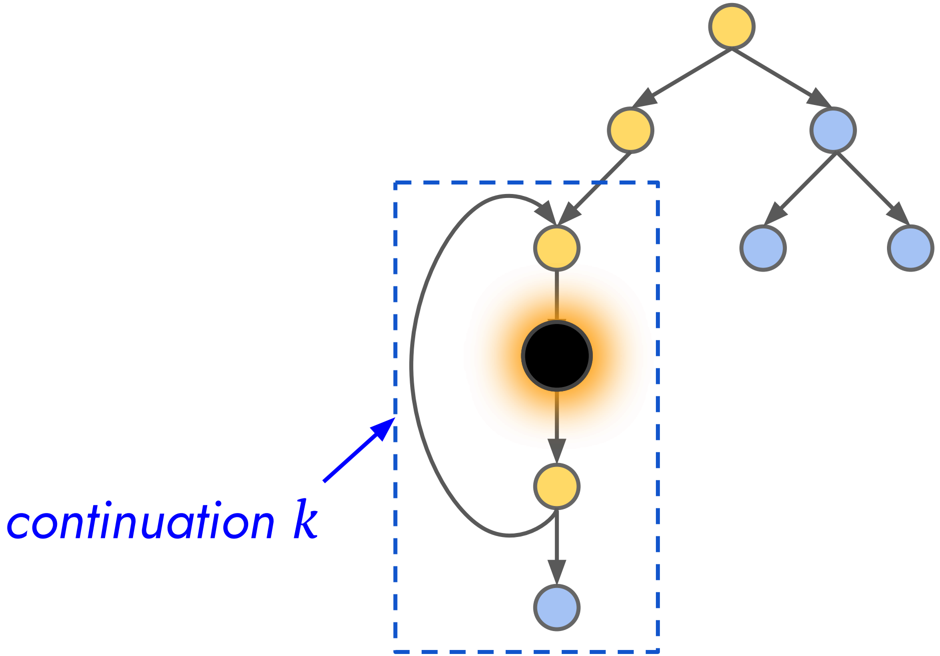
Making Control Explicitly

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*

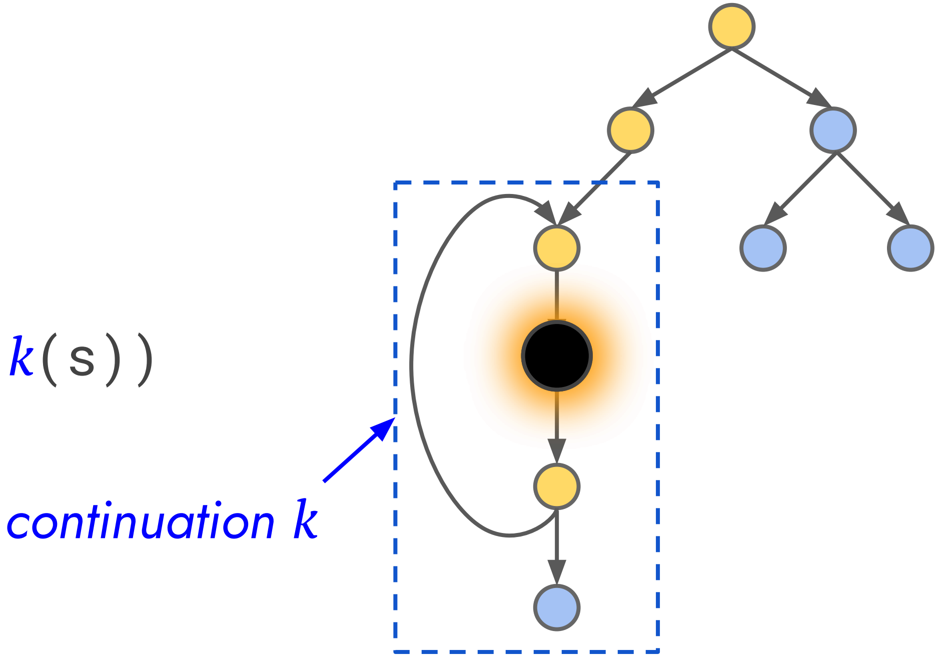


Making Control Explicitly

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

save and pause

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

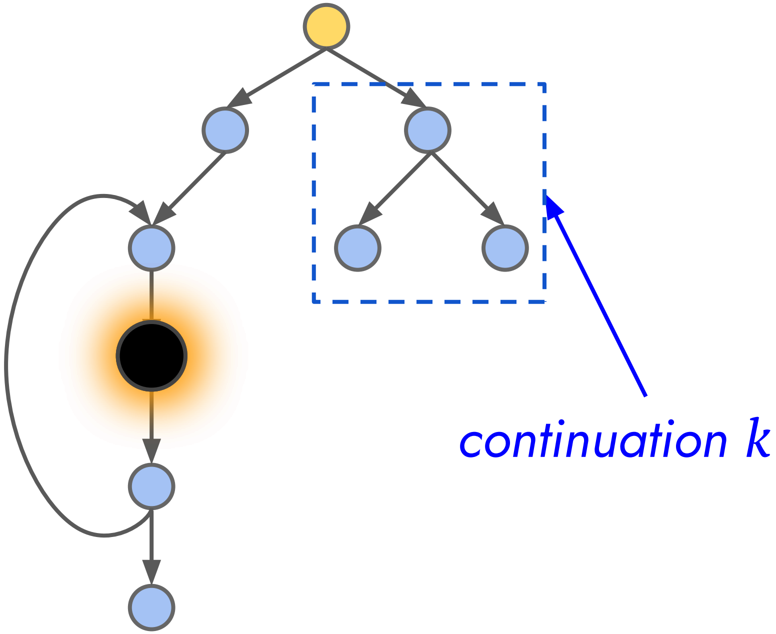


Making Control Explicitly

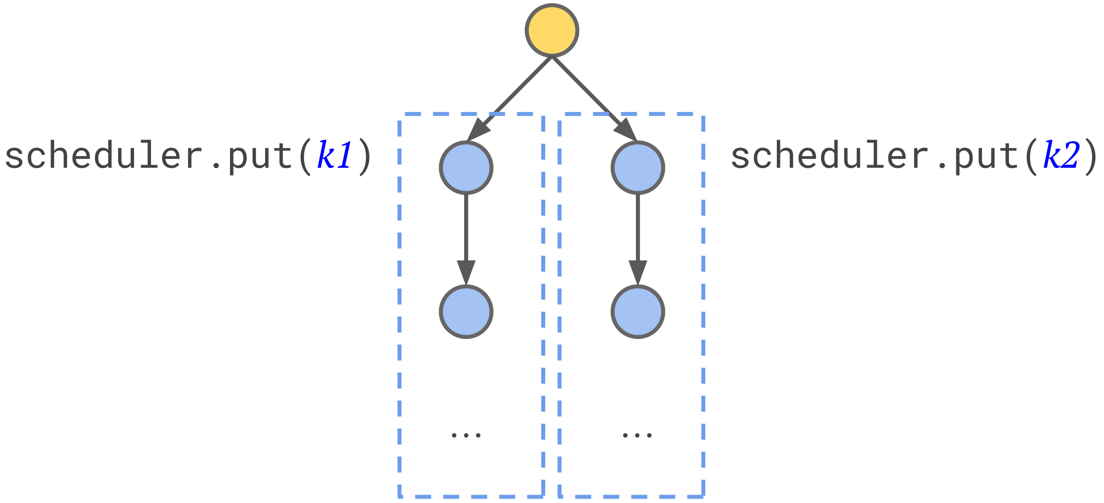
represent the rest of execution as a function k *in the generated code*

dispatch and resume

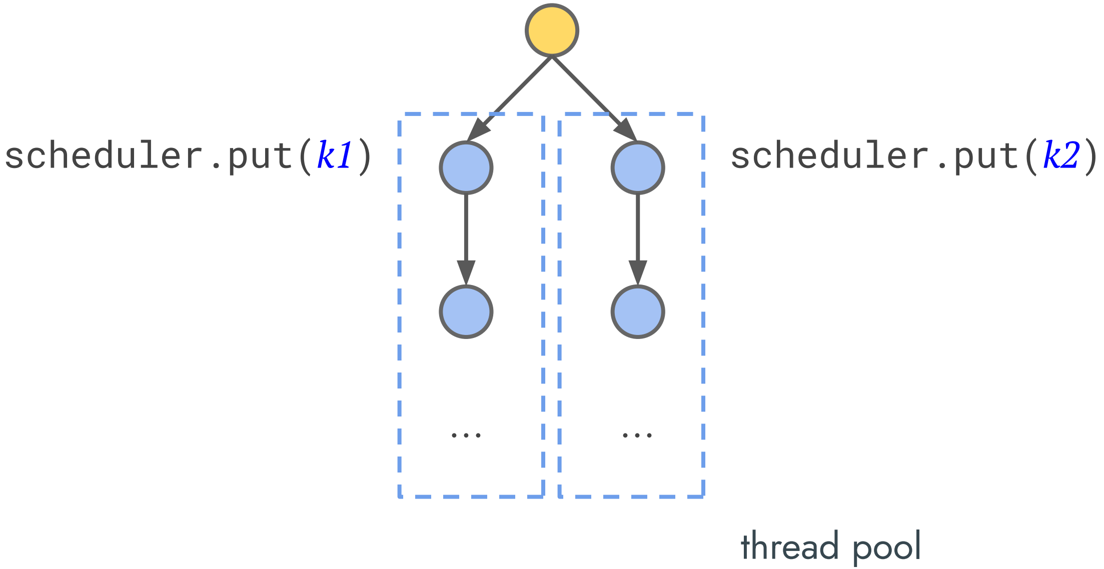
k = `scheduler.get();` k`()`



Parallelism for Free



Parallelism for Free




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

Making Control Explicitly

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

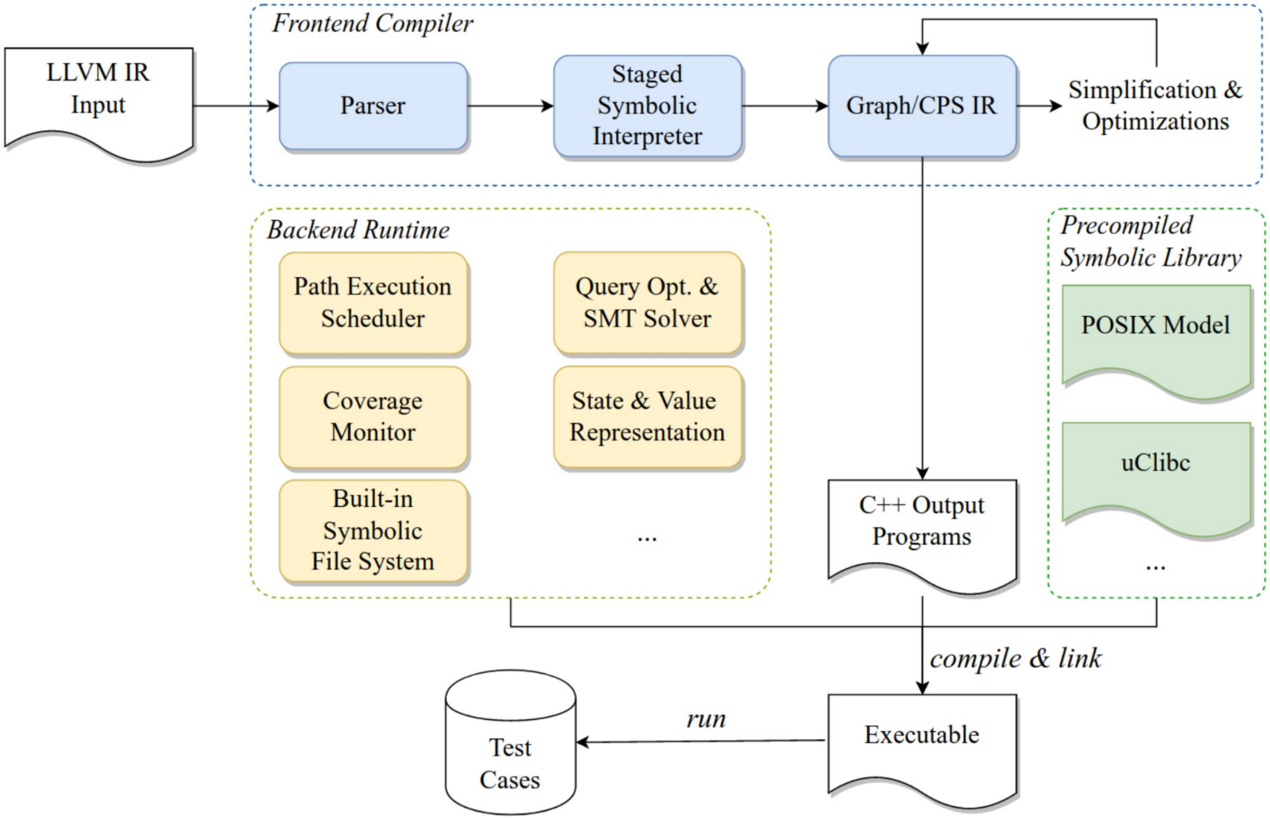
- *invoke and fork*
  `k(s1); k(s2)`
- *save and pause*
  `scheduler.put(() => k(s))`
- *dispatch and resume*
  `k = scheduler.get(); k()`
- *dispatch in parallel*

Compiling Symbolic Execution with Continuations

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

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

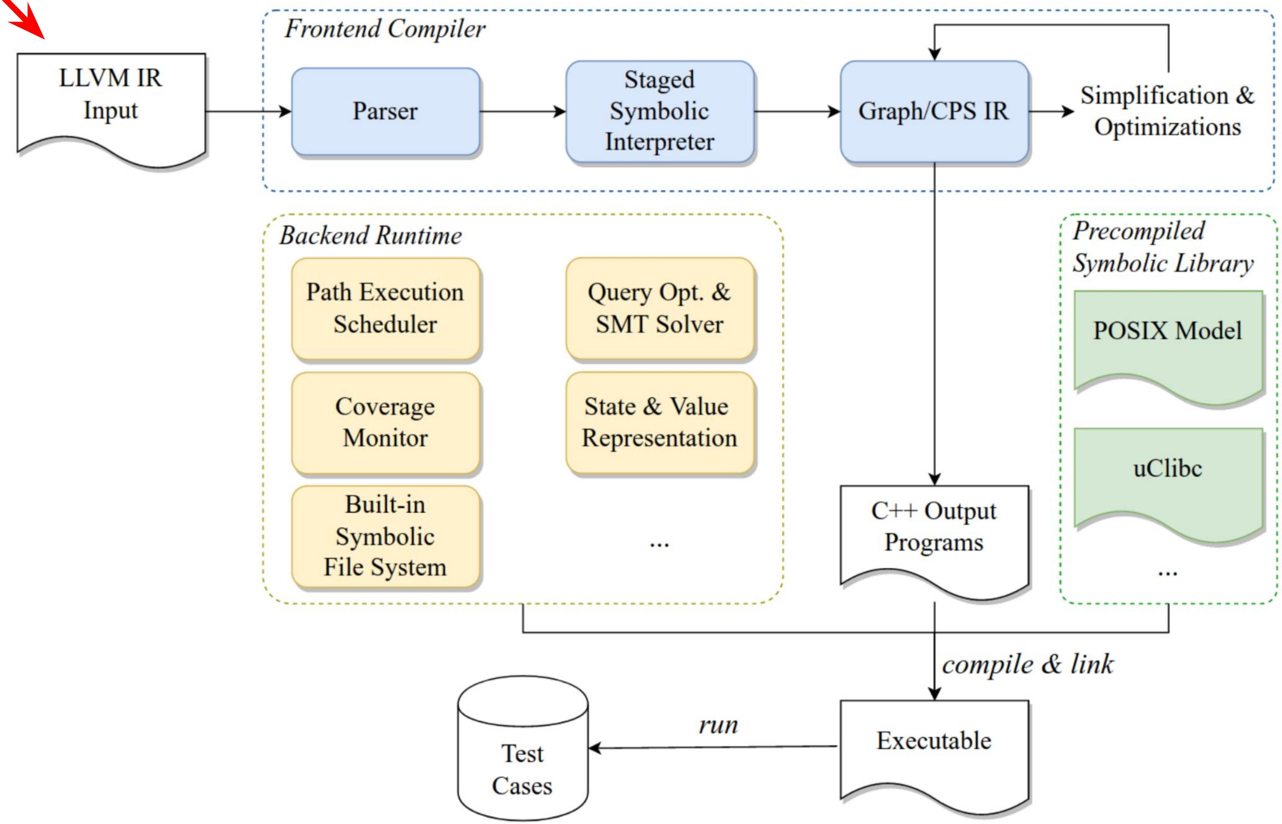
GenSym



Frontend Compiler
LLVM IR Input
Parser
Staged Symbolic Interpreter
Graph/CPS IR
Simplification & Optimizations
Backend Runtime
Path Execution Scheduler
Query Opt. & SMT Solver
Coverage Monitor
State & Value Representation
Built-in Symbolic File System
...
C++ Output Programs
Precompiled Symbolic Library
POSIX Model
uClibc
...
compile & link
run
Test Cases
Executable

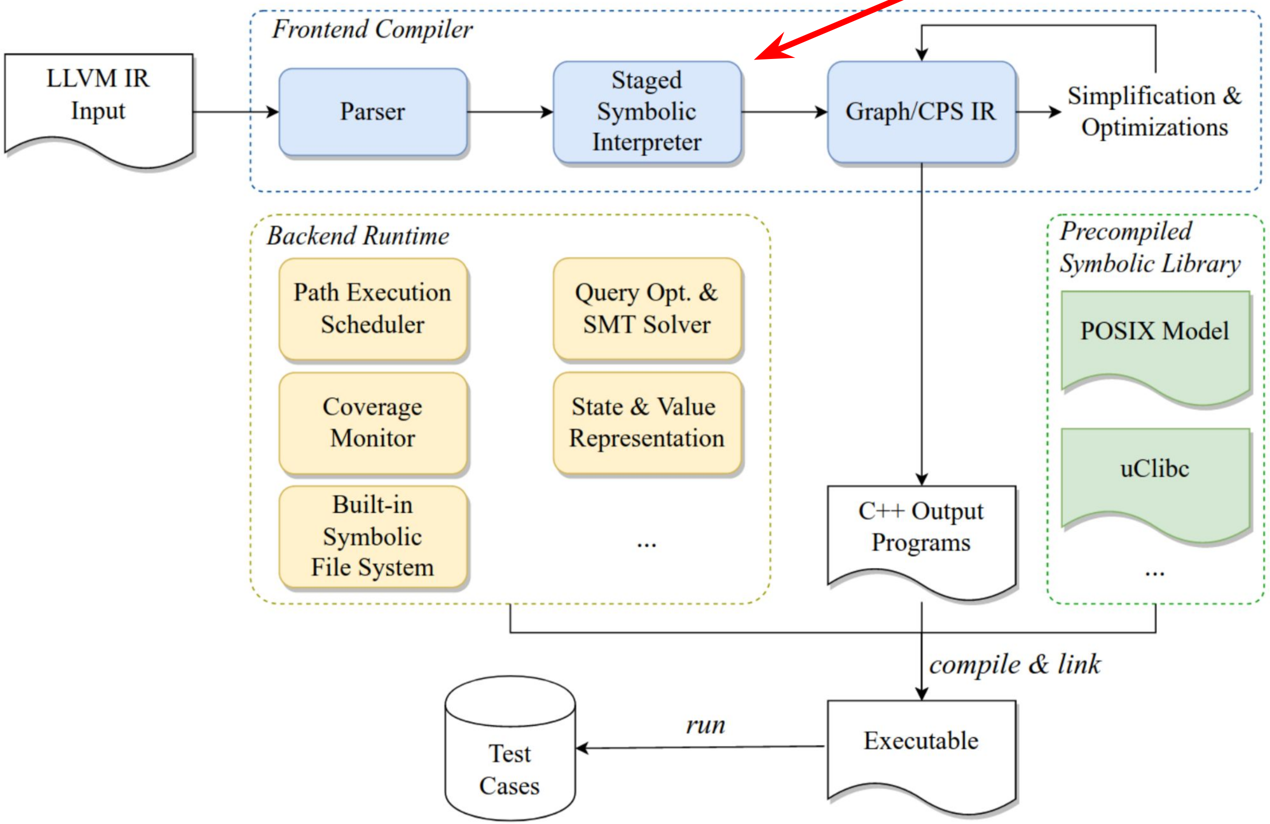
Takes general LLVM IR inputs

GenSym

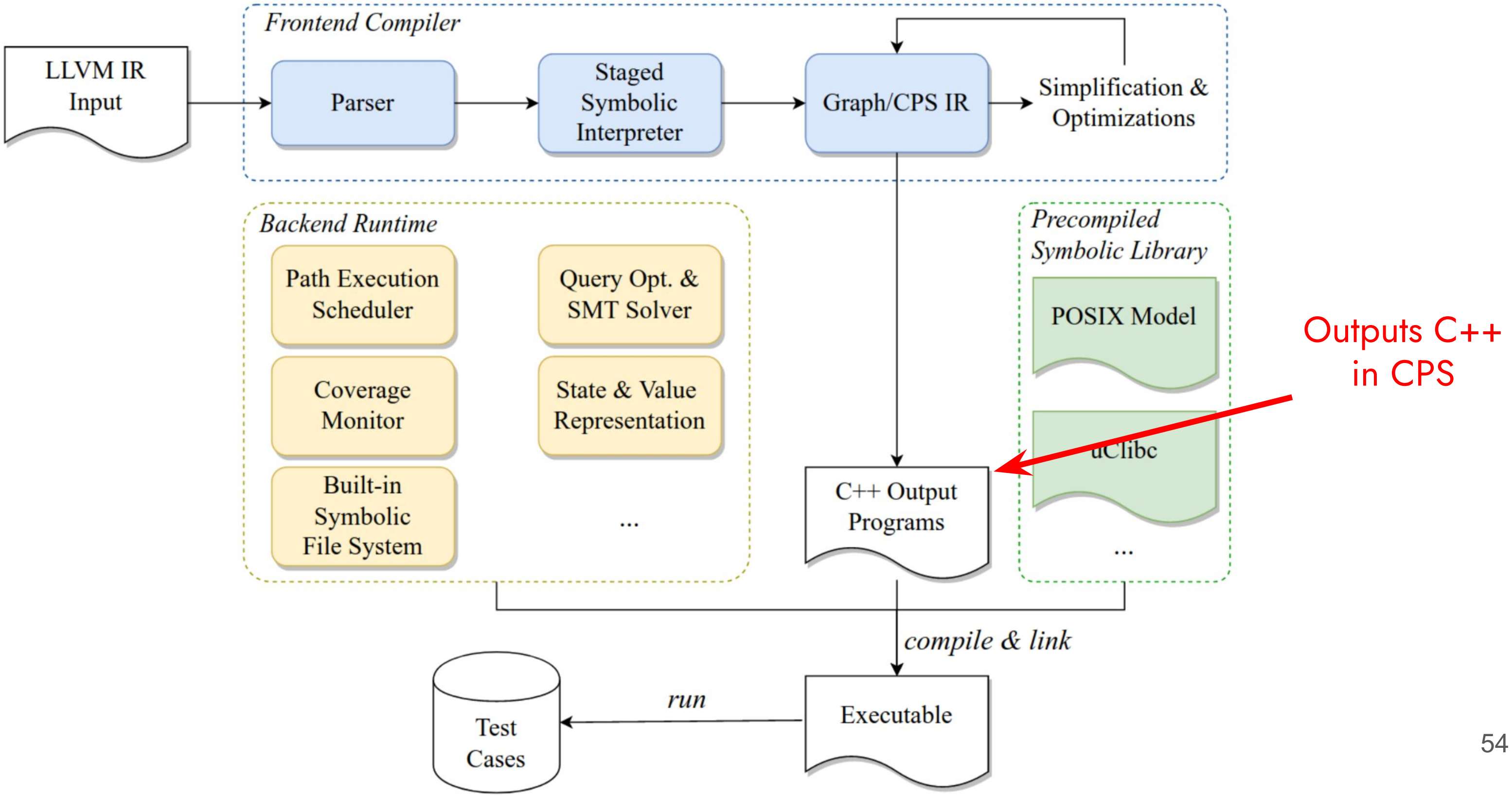


GenSym

Written in Scala/LMS



GenSym



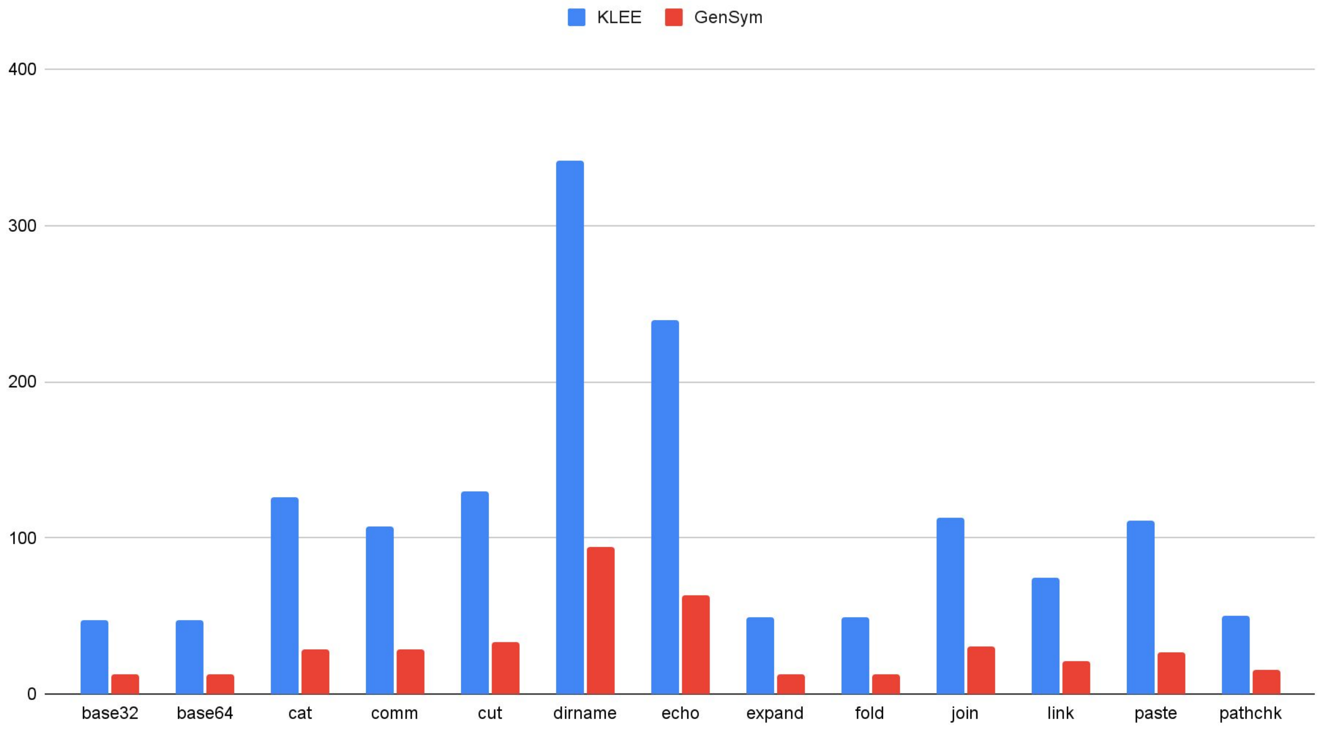
Frontend Compiler
LLVM IR Input
Parser
Staged Symbolic Interpreter
Graph/CPS IR
Simplification & Optimizations
Backend Runtime
Path Execution Scheduler
Query Opt. & SMT Solver
Coverage Monitor
State & Value Representation
Built-in Symbolic File System
...
Precompiled Symbolic Library
POSIX Model
uClibc
...
C++ Output Programs
compile & link
run
Test Cases
Executable
Outputs C++ in CPS

GenSym: Performance Evaluation

- KLEE: state-of-the-art symbolic interpreter for LLVM IR
  - Has been actively developed over 15+ years
  - Written in C++

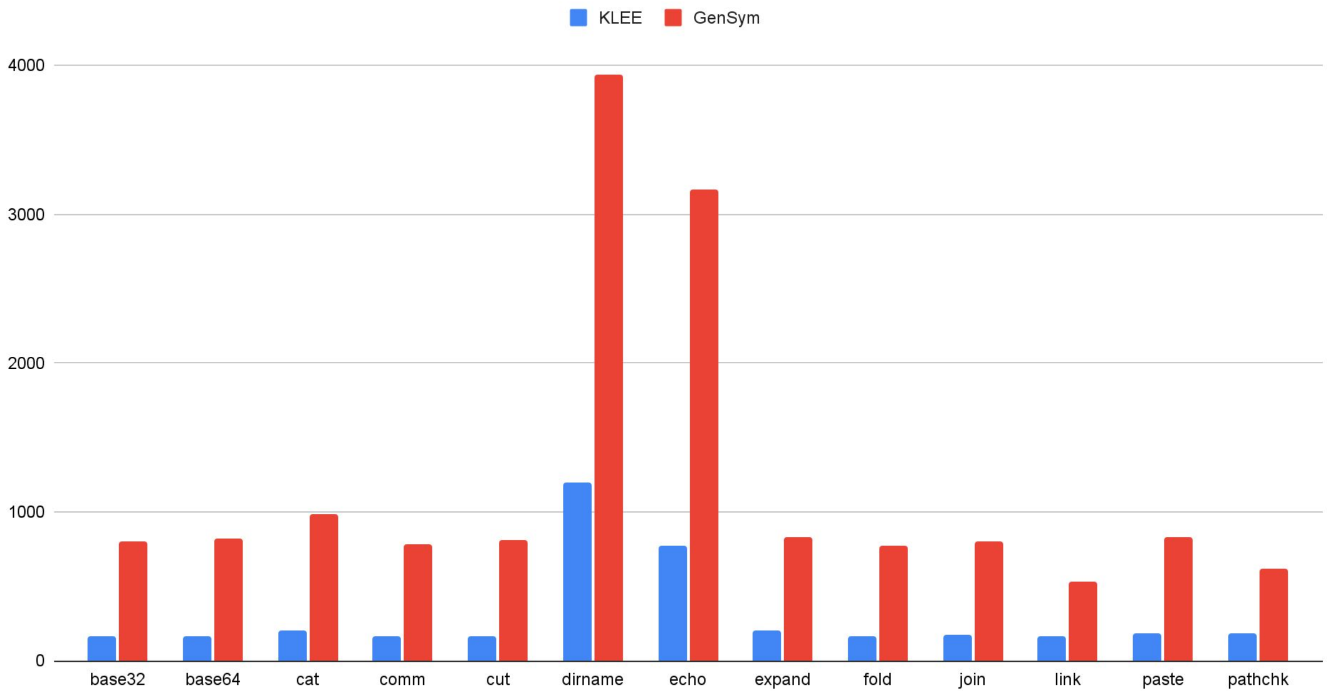
- Evaluated on a set of GNU Coreutils programs
  - Using POSIX file system and uClibc library
  - Average program size 28k LOC of LLVM IR instructions

Single-thread Pure Execution



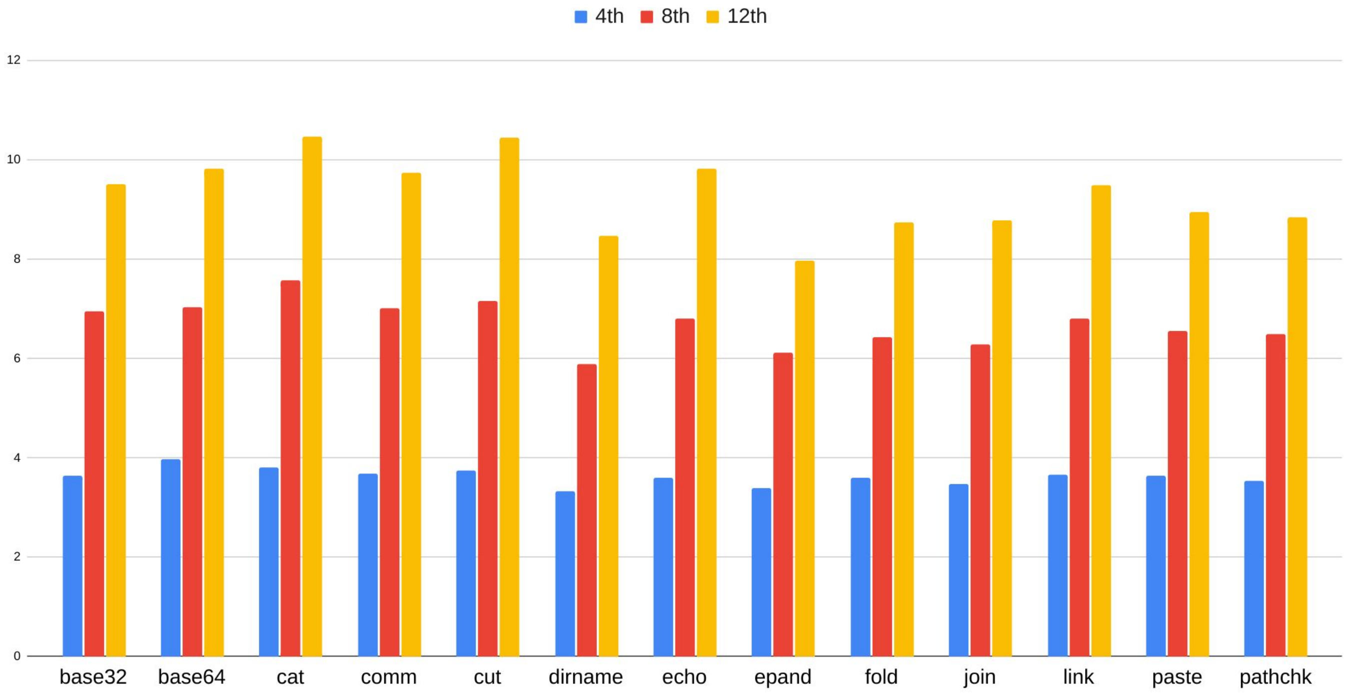
~4x speedups

Single-thread Throughput



Number of explored paths per second in 1 hour: **4.3x more paths on avg**.

Parallel Execution Efficiency



Speedups using more cores/threads

4 threads - 3.6x
8 threads - 6.7x
12 threads - 9.3x

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

- ★ Efficient
  - ○ Semantics-based compilation
  - ○ Outperforms state-of-the-art tools
- ★ Effective
  - ○ Forking as concurrency/parallelism
  - ○ Path-selection heuristics

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

[ICSE '23] Compiling parallel symbolic execution with continuations.

[OOPSLA '20] Compiling symbolic execution with staging and algebraic effects.

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

- ★ Efficient
  - Semantics-based compilation
  - Outperforms state-of-the-art tools
- ★ Effective
  - Forking as concurrency/parallelism
  - Path-selection heuristics

I'm on the academic job market; happy to chat more about my research!

Questions?

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

[ICSE '23] Compiling parallel symbolic execution with continuations.
[OOPSLA '20] Compiling symbolic execution with staging and algebraic effects.