

Compiling & Controlling Symbolic Execution

Guannan Wei

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



Northeastern University - Dec 1 2023



Symbolic Execution

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

Symbolic Execution

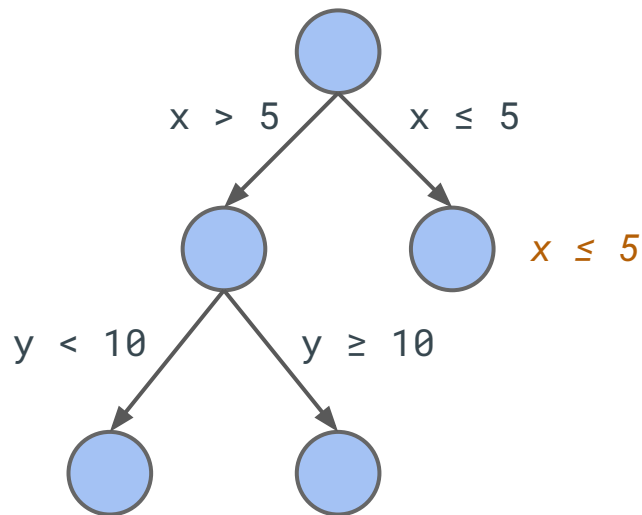
*mark as
symbolic*

```
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 */
}
```



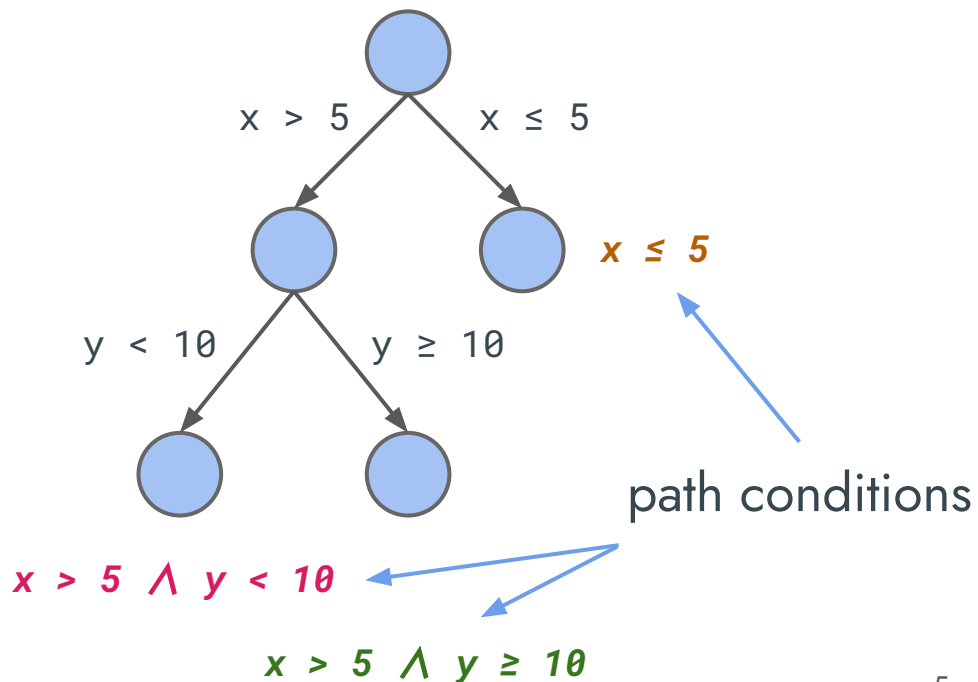
$x > 5 \wedge y < 10$

$x > 5 \wedge y \geq 10$

Symbolic Execution

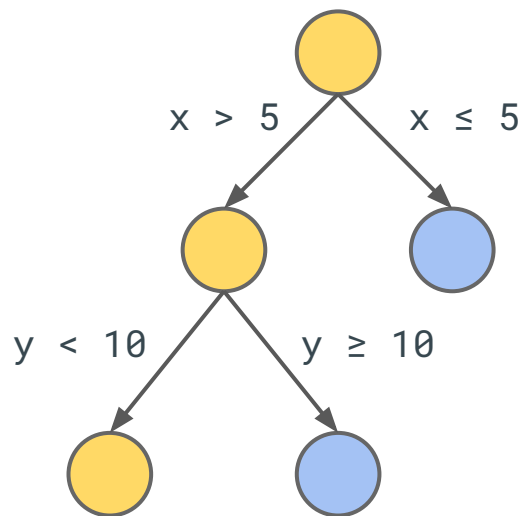
mark as
symbolic

```
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 */
}
```



$\text{solver}(x > 5 \wedge y < 10) = \{ x = 6, y = 9 \}$

Symbolic Execution – Applications

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

Symbolic Execution – Applications

KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

Cristian Cadar, Daniel Dunbar, Dawson Engler *
Stanford University

Abstract

We present a new symbolic execution tool, KLEE, capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs. We used KLEE to thoroughly check all 89 stand-alone programs in the GNU COREUTILS utility suite, which form the core user-level environment installed on millions of Unix systems, and arguably are the single most heavily tested set of open-source programs in existence. KLEE-generated tests achieve high line coverage — on average over 90% per tool (median: over 94%) — and significantly beat

symbolic values and replace corresponding concrete program operations with ones that manipulate symbolic values. When program execution branches based on a symbolic value, the system (conceptually) follows both branches, on each path maintaining a set of constraints called the *path condition* which must hold on execution of that path. When a path terminates or hits a bug, a test case can be generated by solving the current path condition for concrete values. Assuming deterministic code, feeding this concrete input to a raw, unmodified version of the checked code will make it follow the same path and hit the same bug.

Symbolic Execution – Applications

KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

Cristian Cadar, Daniel Dunbar, Dawson Engler *
Stanford University

Abstract

We present a new symbolic execution tool, KLEE, capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs. We used KLEE to thoroughly check all 89 stand-alone programs in the GNU COREUTILS utility suite, which form the core user-level environment installed on millions of Unix systems, and arguably are the single most heavily tested set of open-source programs in existence. KLEE-generated tests achieve high line coverage — on average over 90% per tool (median: over 94%) — and significantly beat

symbolic values
operations
When prog
value, the s
on each pa
path condi
path. Whe
can be gen
for concret
ing this co
the checke
hit the sam

and many others ...

Higher-Order Symbolic Execution via Contracts

Sam Tobin-Hochstadt David Van Horn

Northeastern University
{samth,dvanhorn}@ccs.neu.edu

Abstract

We present a new approach to automated reasoning about higher-order programs by extending symbolic execution to use behavioral contracts as symbolic values, enabling *symbolic approximation of higher-order behavior*.

Our approach is based on the idea of an *abstract* reduction semantics that gives an operational semantics to programs with both concrete and symbolic components. Symbolic components are approximated by their contract and our semantics gives an operational interpretation of contracts-as-values. The result is a executable semantics that soundly predicts program behavior, including contract failures, for all

verification and analysis challenging as well as requiring substantial effort to write sufficient specifications.

The problem of program analysis and verification in the presence of missing *data* has been widely studied, producing many effective tools that apply *symbolic execution* to non-deterministically consider many or all possible inputs. These tools typically determine constraints on the missing data, and reason using these constraints. Since the central lesson of higher-order programming is that computation *is* data, we propose symbolic execution of higher-order programs for reasoning about systems with omitted components, taking specifications to be our constraints.

Symbolic Execution Engine

a concrete interpreter $\text{eval}: \text{Prog} \rightarrow (\text{Value}, \text{State})$

- simulates the execution deterministically



Symbolic Execution Engine

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

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



Path Explosion

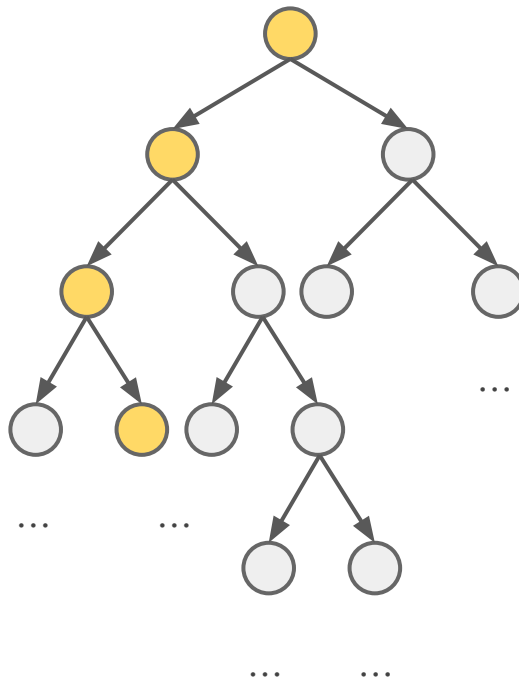
Concrete Execution

1 path

vs

Symbolic Execution

exponential number of
independent paths



Path Explosion

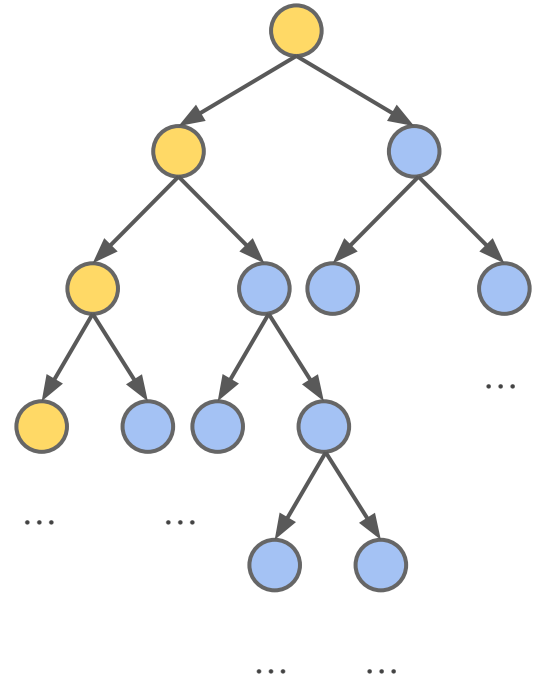
Concrete Execution

1 path

vs

Symbolic Execution

exponential number of
independent paths



Path Explosion

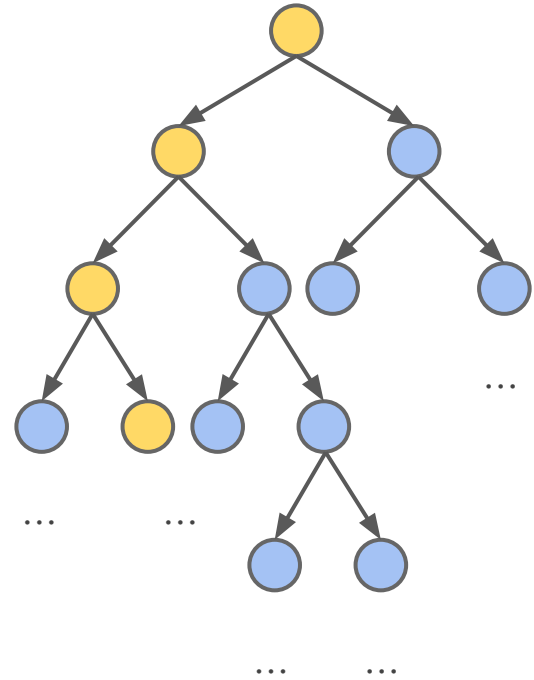
Concrete Execution

1 path

vs

Symbolic Execution

exponential number of
independent paths



Path Explosion

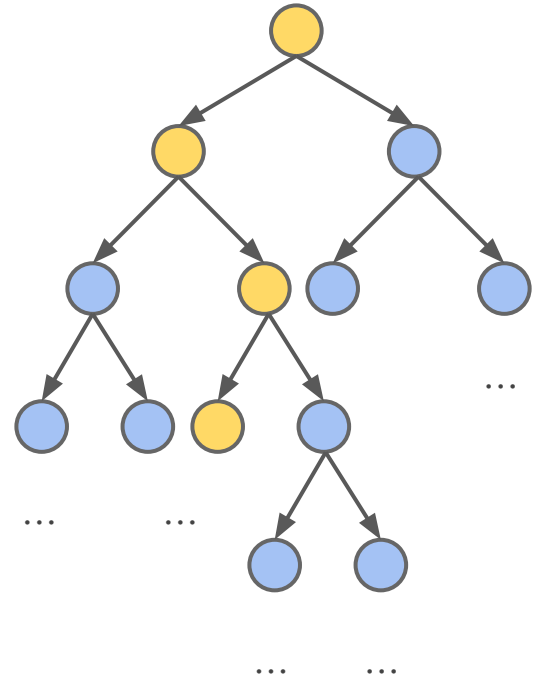
Concrete Execution

1 path

vs

Symbolic Execution

exponential number of
independent paths



Path Explosion

Concrete Execution

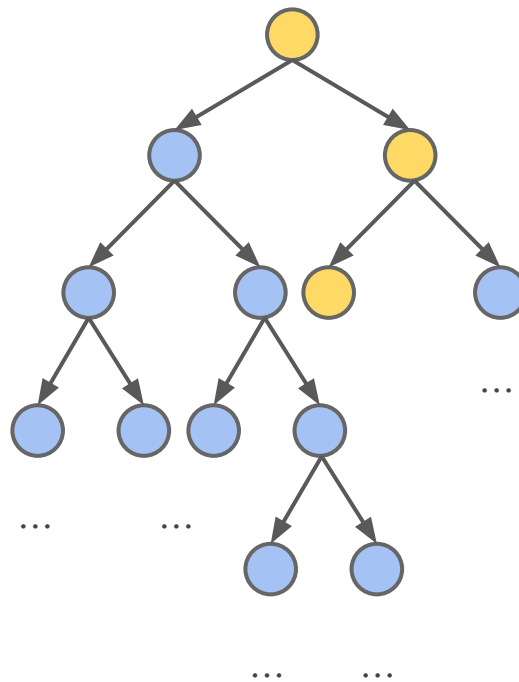
1 path

vs

Symbolic Execution

exponential number of
independent paths

performance matters



Performance Matters

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

symbolic interpreter performance
compared to native execution

<i>KLEE</i> (C++)	3,000x slower
<i>angr</i> (Python)	321,000x slower

Performance Matters

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

interpretation overhead

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

Symbolic-Execution Compilers

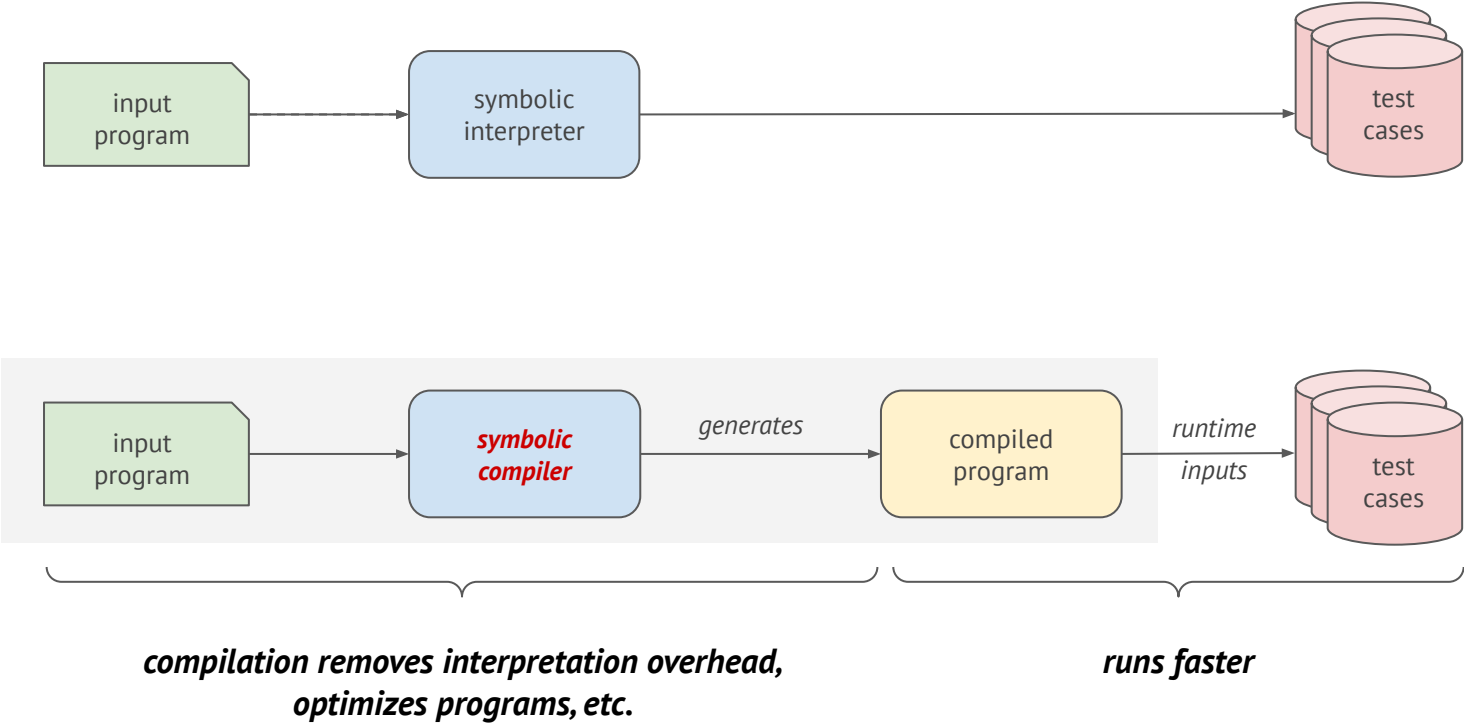
To remove these overheads,

compilation is inevitable.

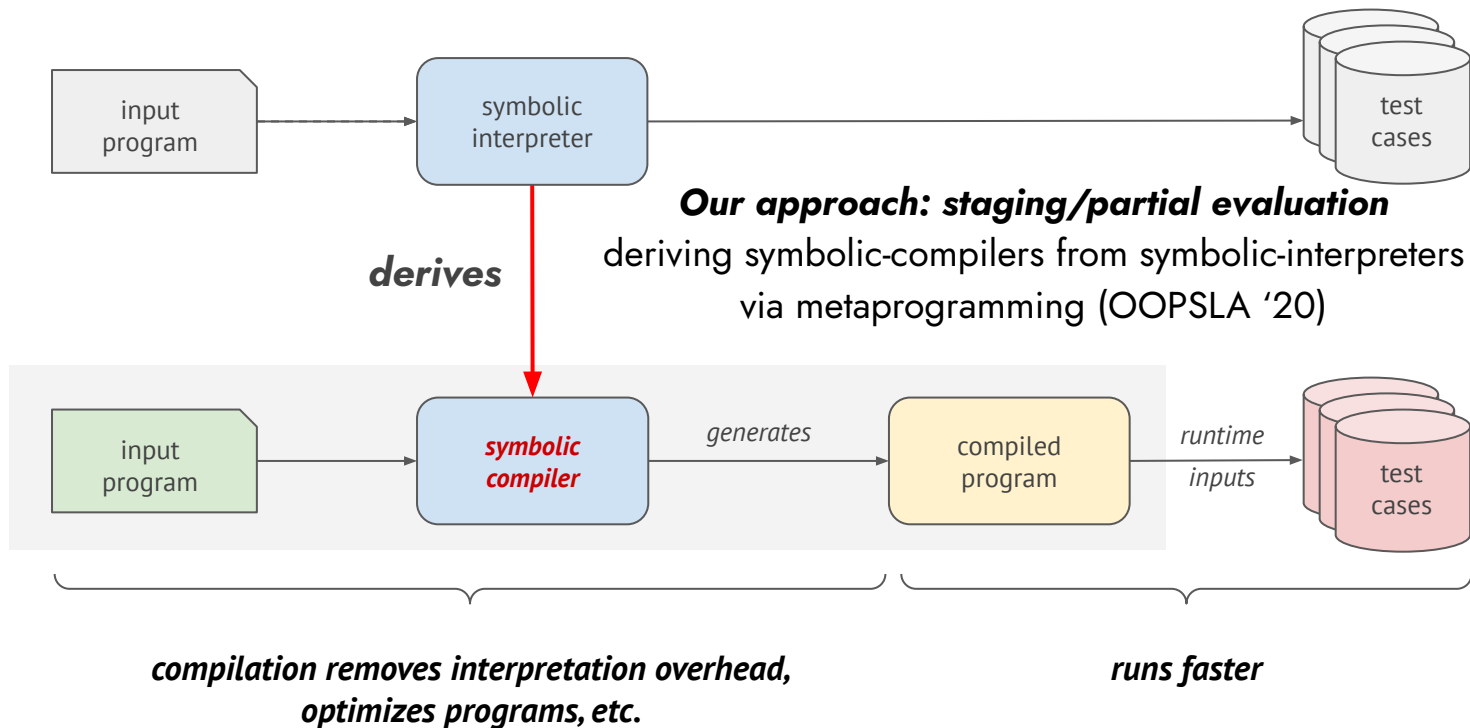
Symbolic-Execution Compilers



Symbolic-Execution Compilers



Symbolic-Execution Compilers



Symbolic-Execution Compilers

Abstract Compilation: A New Implementation Paradigm for Static Analysis

Dominique Boucher and Marc Feeley

Département d'informatique et de recherche opérationnelle (IRO)
Université de Montréal
C.P. 6128, succ. centre-ville, Montréal, Québec, Canada H3C 3J7
E-mail: {boucherd,feeley}@iro.umontreal.ca

Abstract. For large programs, static analysis can be one of the most time-consuming phases of the whole compilation process. We propose a new paradigm for the implementation of static analyses that is inspired by partial evaluation techniques. Our paradigm does not reduce the complexity of these analyses, but it allows an *efficient implementation*. We illustrate this paradigm by its application to the problem of control flow analysis of functional programs. We show that the analysis can be sped up by a factor of 2 over the usual abstract interpretation method.

input
program

input
program

test
cases

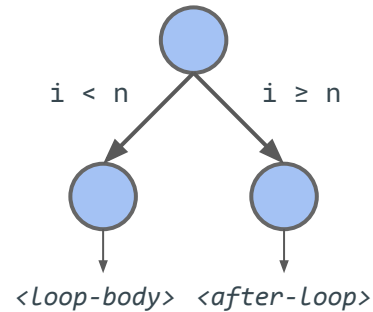
test
cases

Path Explosion, Worse

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

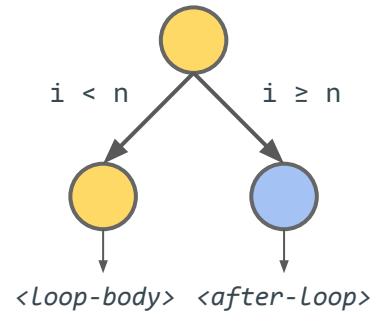
Path Explosion, Worse

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



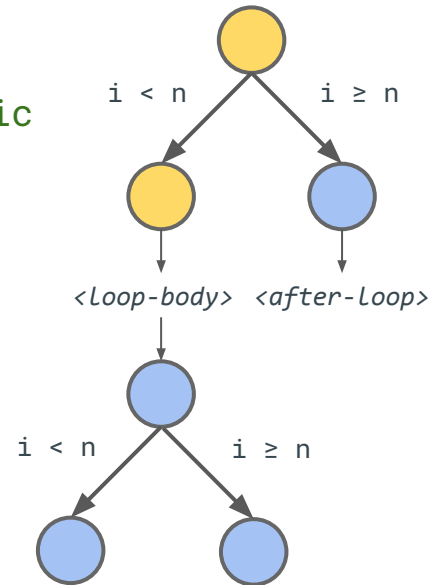
Path Explosion, Worse

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



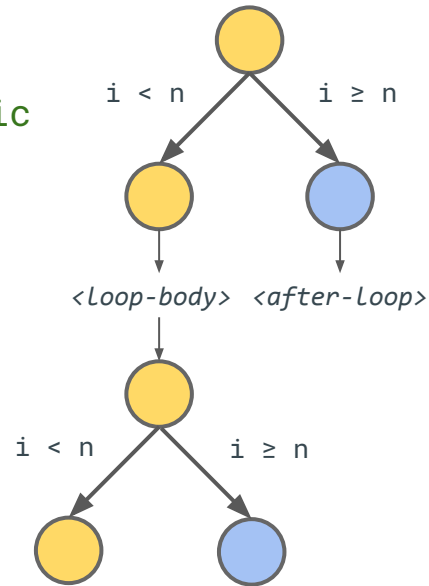
Path Explosion, Worse

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



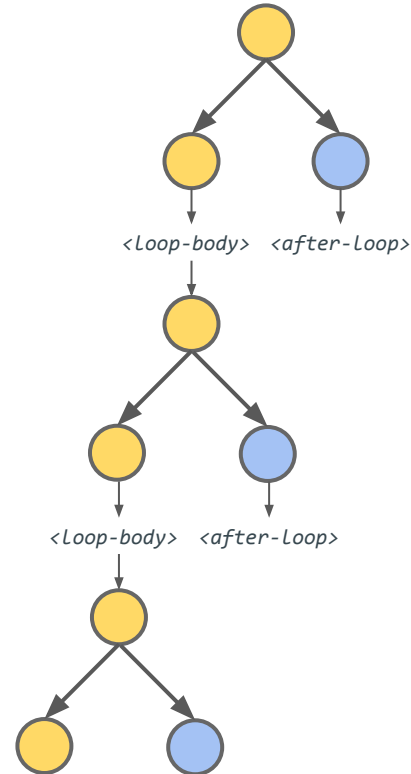
Path Explosion, Worse

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



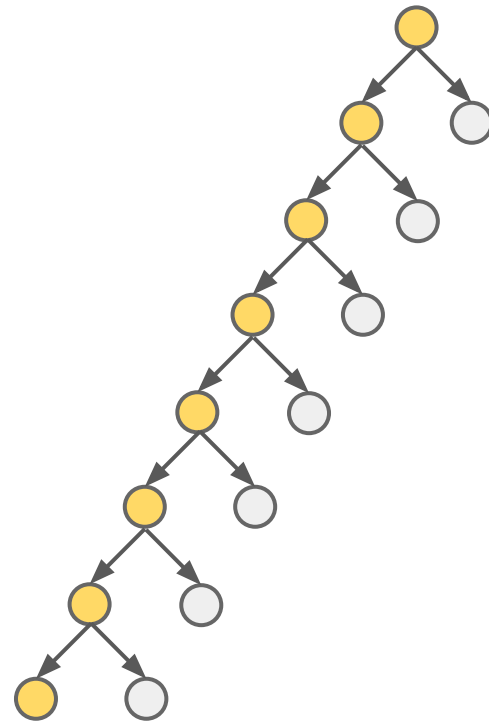
Path Explosion, Worse

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



Path Explosion, Worse

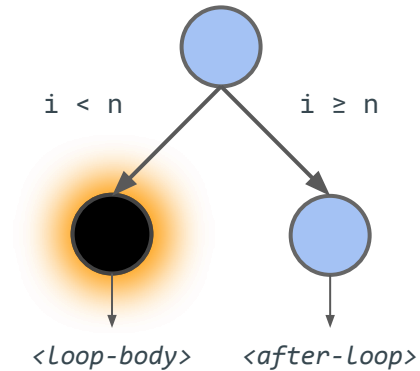
```
n = user_input() // i.e. symbolic  
while (i < n) {  
    <Loop-body>  
}  
<after-Loop>
```



...

Path Explosion, Worse

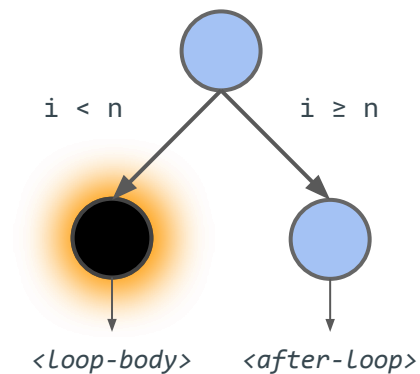
```
n = user_input() // i.e. symbolic
while (i < n) {
    <Loop-body>
}
<after-Loop>
```



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

Escaping the Black Hole

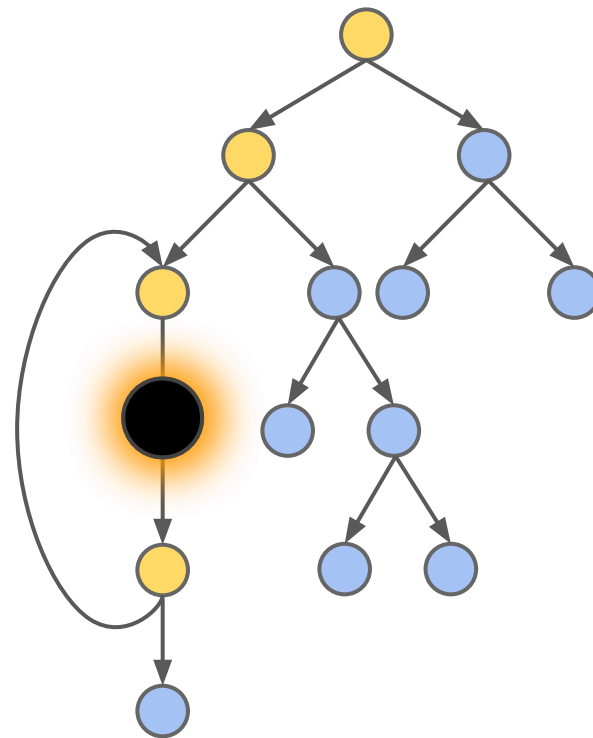
```
n = user_input() // i.e. symbolic
while (i < n) {
    <Loop-body>
}
<after-Loop>
```



Traditional wisdom: deploys clever path selection heuristics

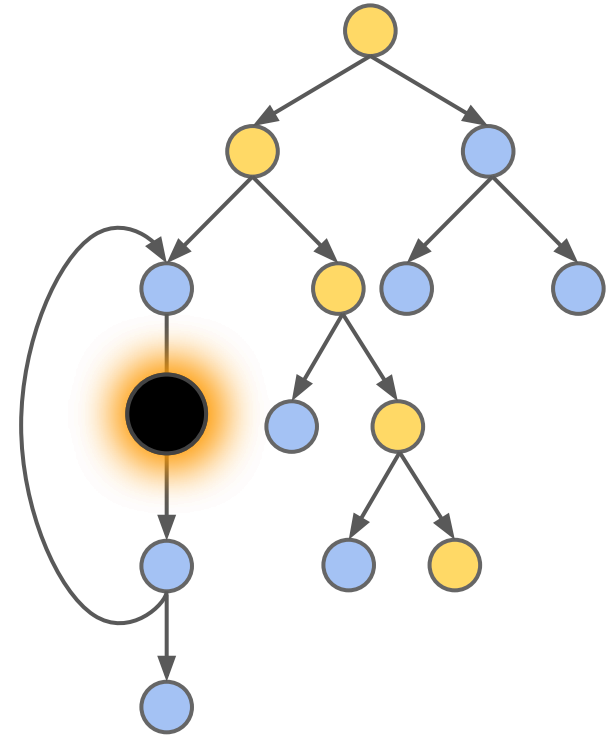
Escaping the Black Hole

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



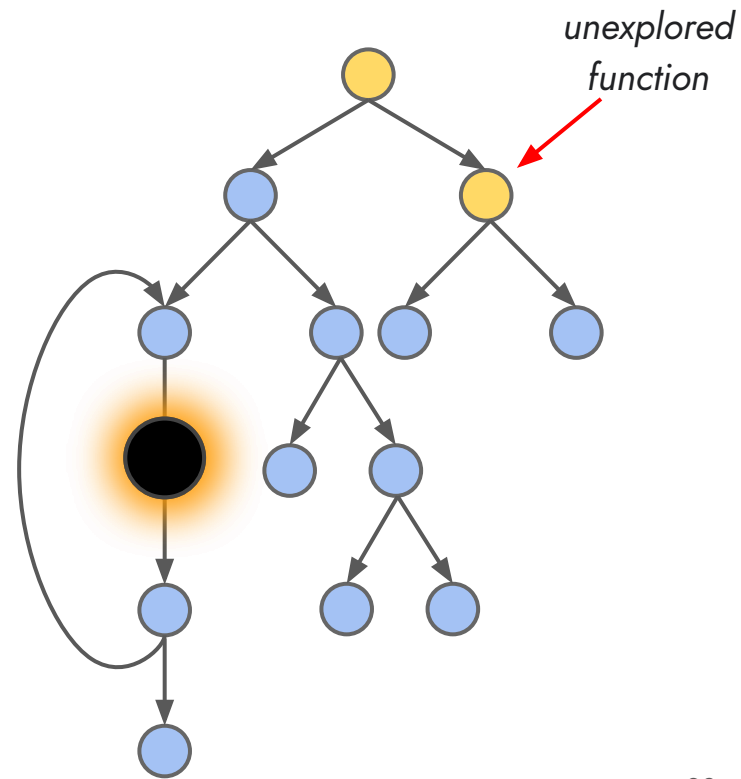
Escaping the Black Hole

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



Escaping the Black Hole

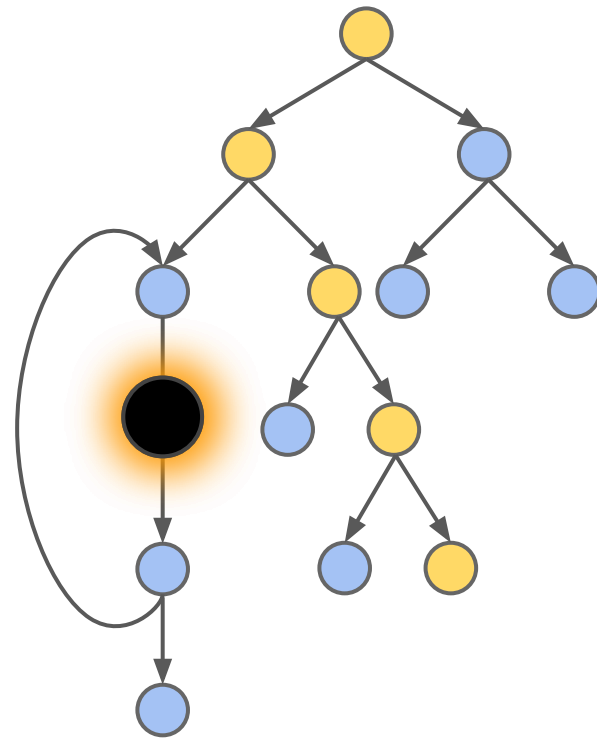
- random state/path selection
- *coverage-guided heuristics*
- ...



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

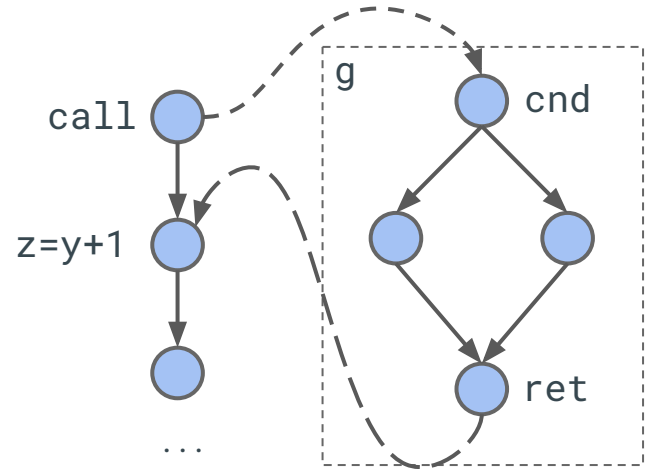
```
y = g()
z = y + 1
...
def g() =
  if (sym_cnd) {
    x = 42
  } else {
    x = 100
  }
  return x
```



Making Control Explicitly

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

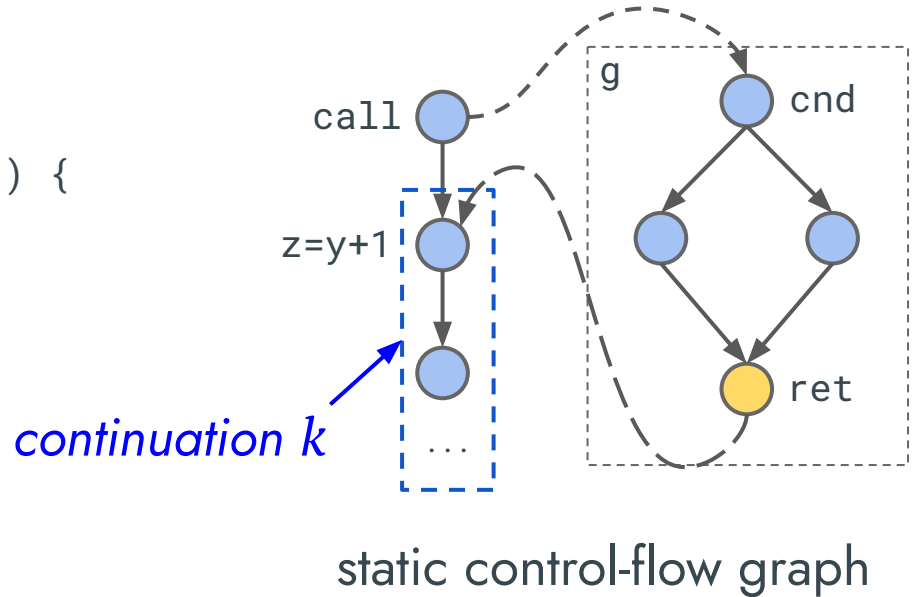
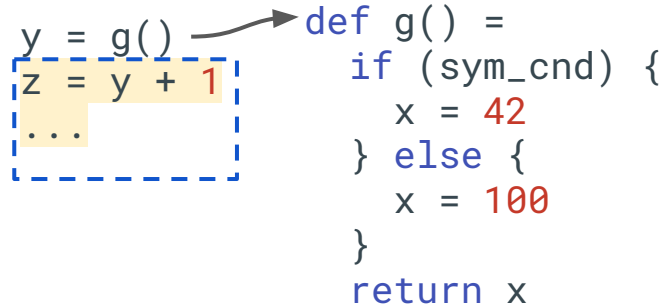
```
y = g()
z = y + 1
...
def g() =
  if (sym_cnd) {
    x = 42
  } else {
    x = 100
  }
  return x
```



static control-flow graph

Making Control Explicitly

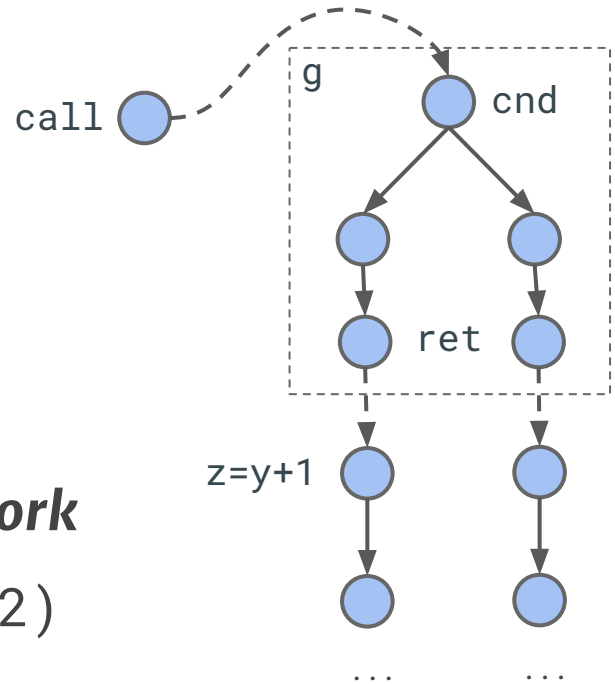
represent the rest of execution as a function *k* in the generated code



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
```

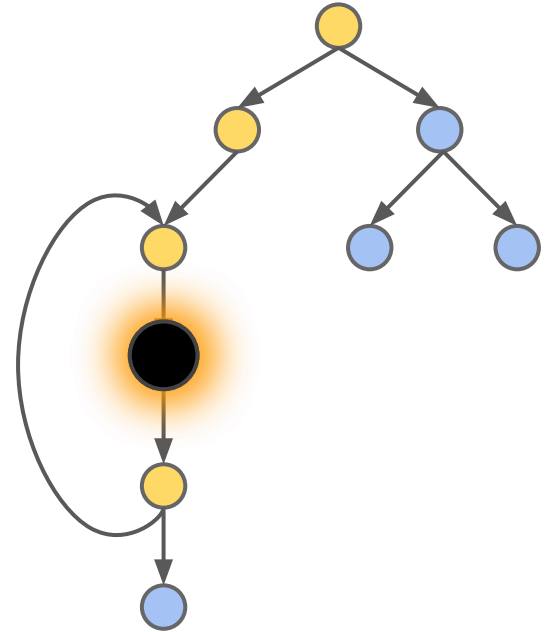


invoke and fork

k(s1); *k*(s2)

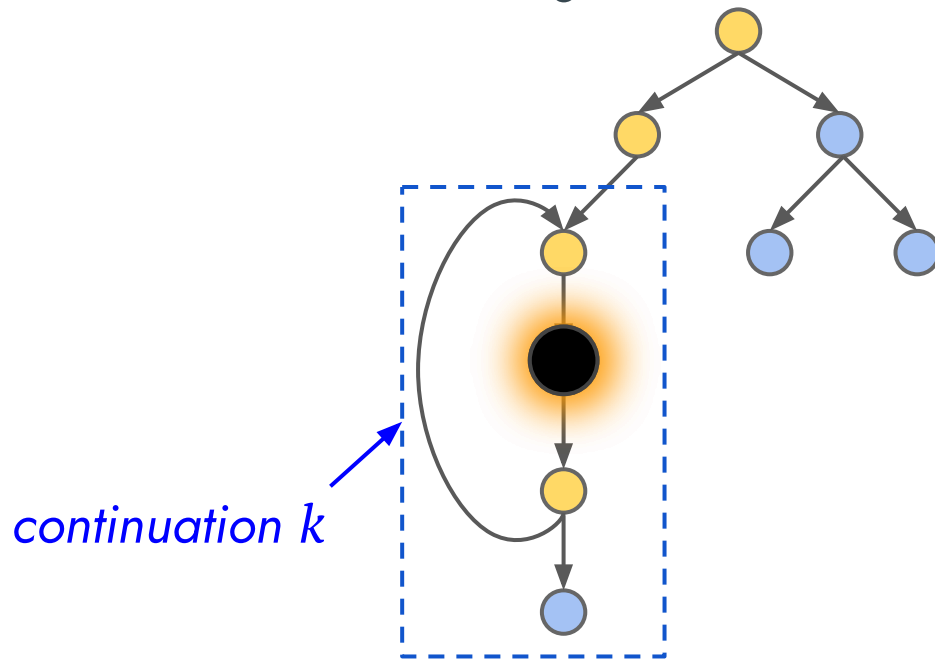
Making Control Explicitly

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



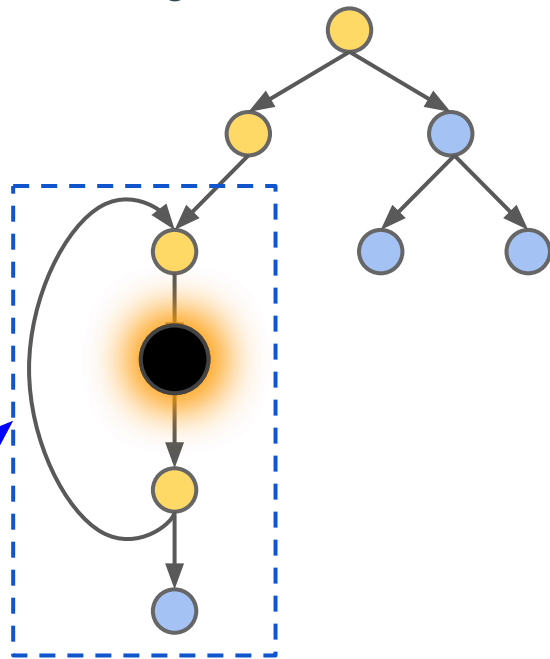
Making Control Explicitly

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

save and pause

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

continuation k

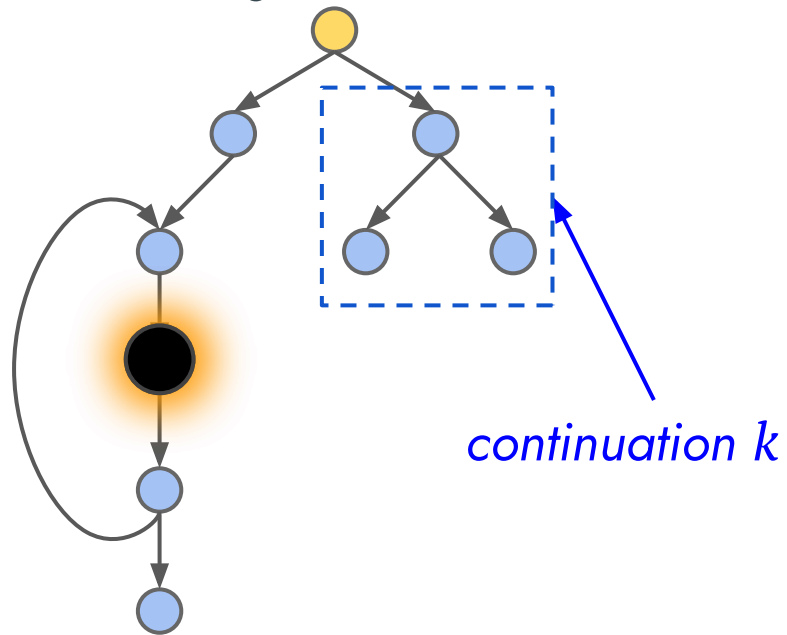


Making Control Explicitly

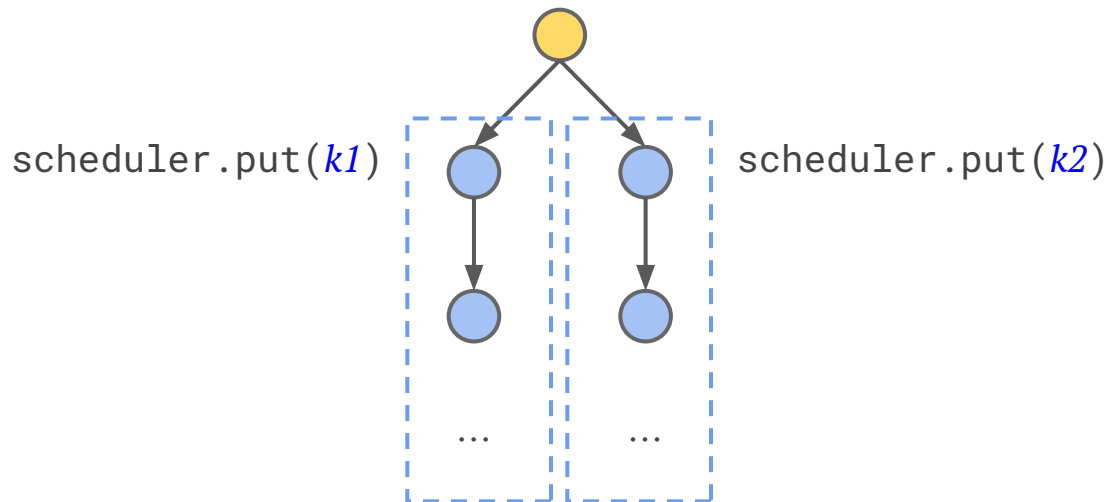
represent the rest of execution as a function k in the generated code

dispatch and resume

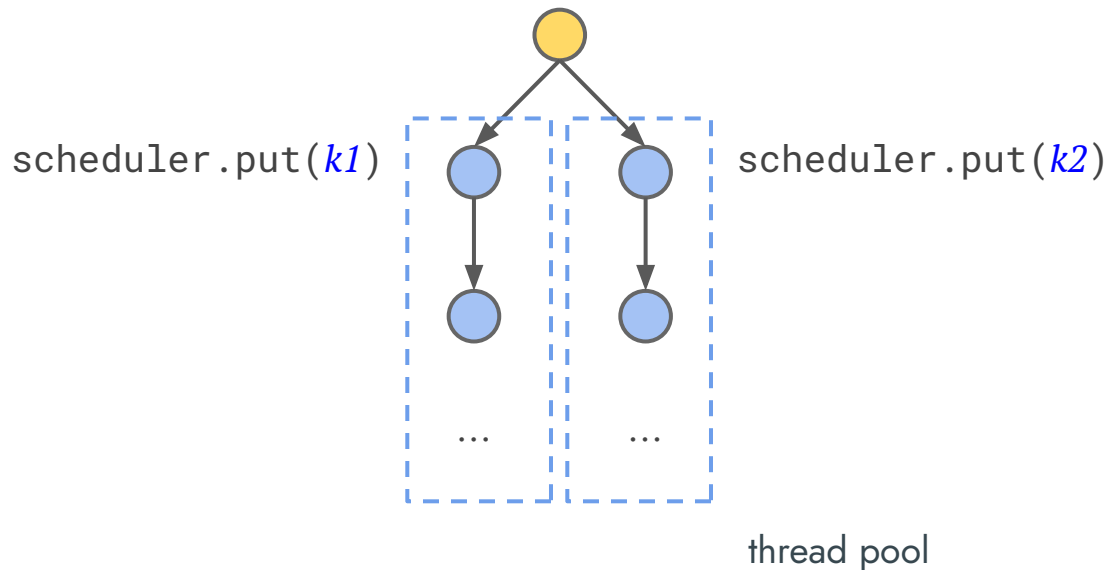
```
 $k$  = scheduler.get();  $k$ ()
```



Parallelism for Free



Parallelism for Free



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

Controlling Symbolic Execution

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

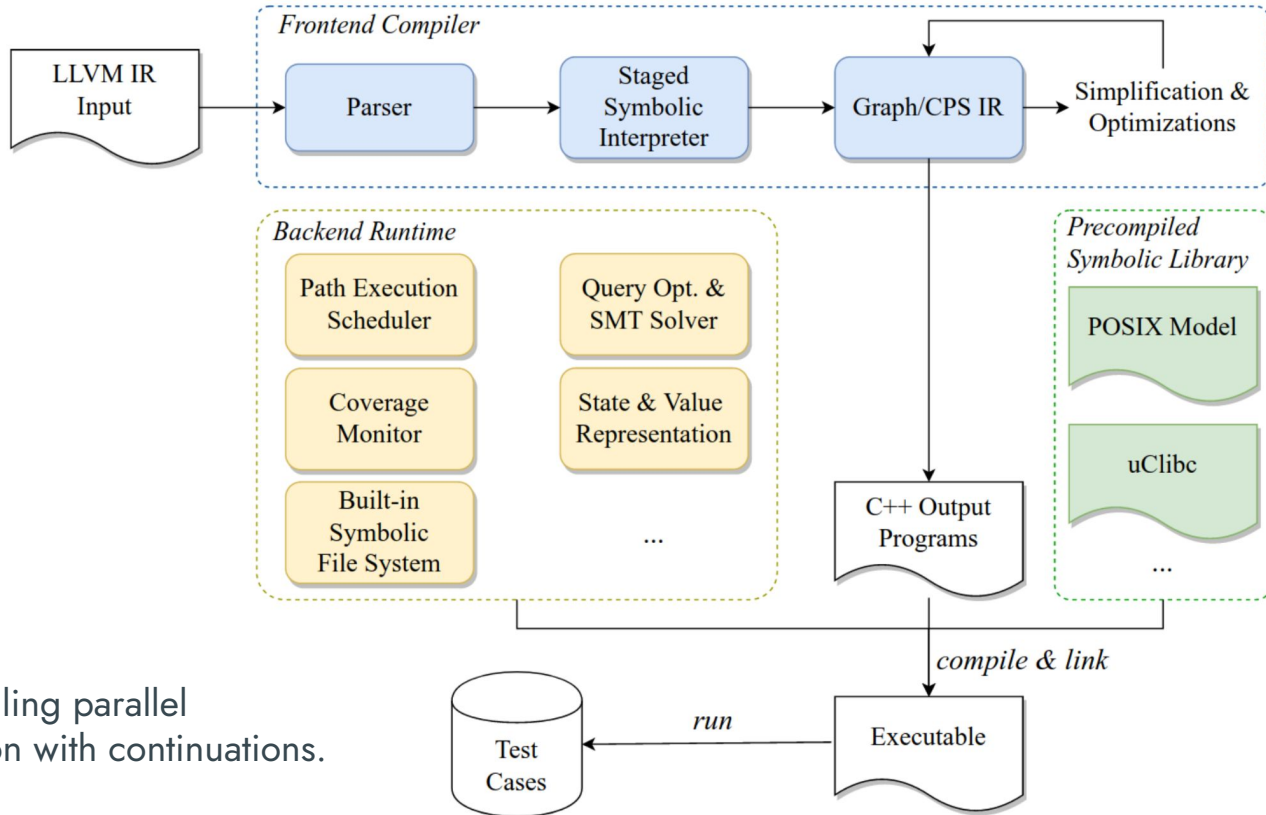
A Continuation View of Symbolic Execution

- Nondeterministic symbolic execution
 - Fork, pause, switch, resume, etc.

A Continuation View of Symbolic Execution

- Nondeterministic symbolic execution
 - Fork, pause, switch, resume, etc.
- Concolic execution
 - Deterministic symbolic execution, control guided by concrete inputs
 - Ongoing work: concolic execution for WebAssembly
- State-merging symbolic execution
 - Fork, but with join points
 - Idea: Synchronization of two parallel/concurrency continuations
- Other strategies or heuristics?

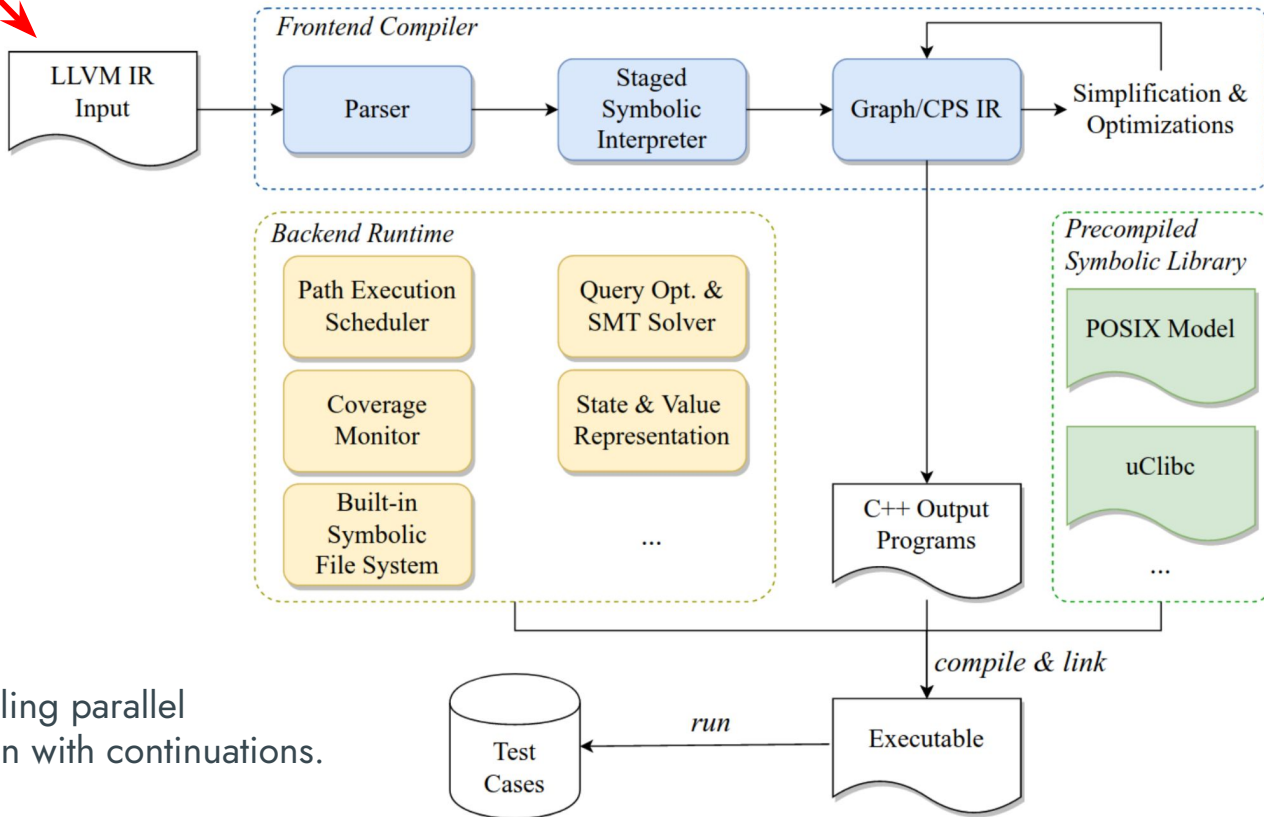
GenSym



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

GenSym

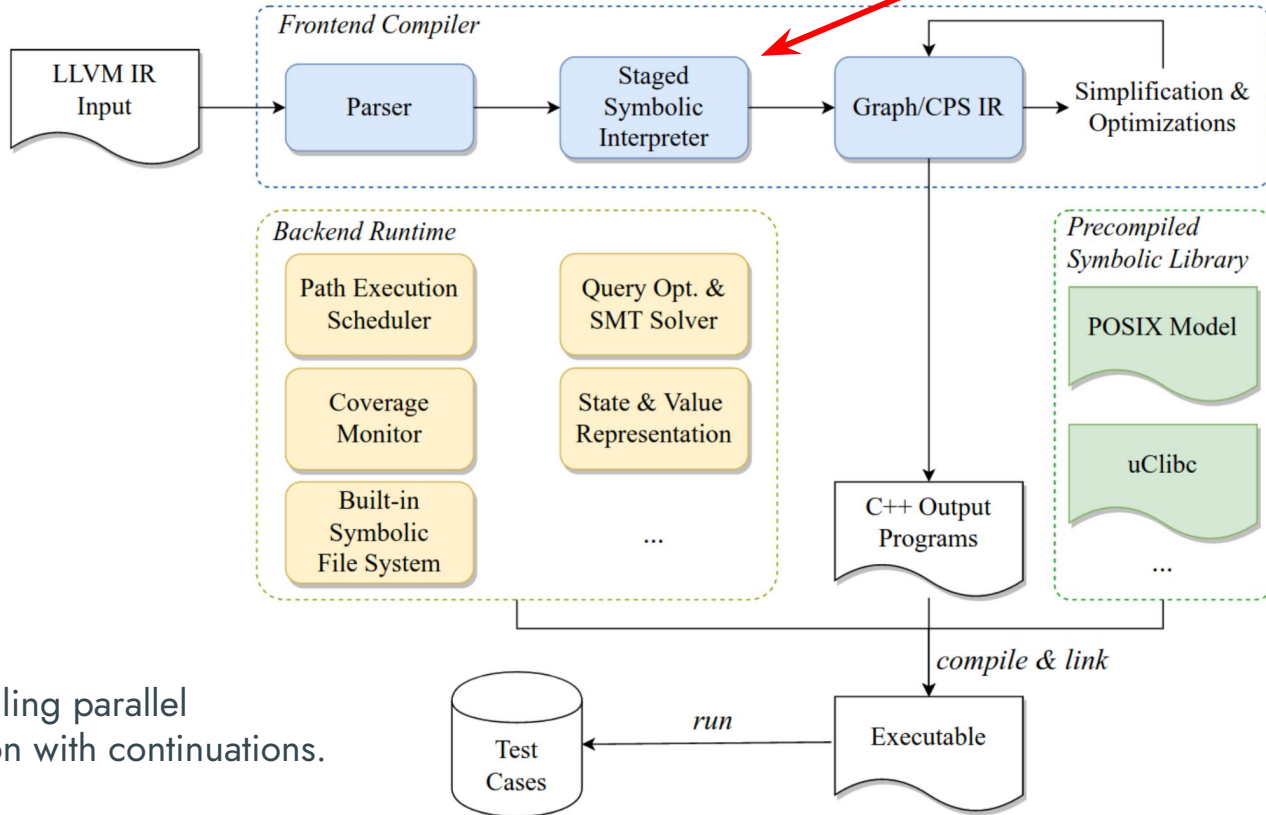
Takes general LLVM IR inputs



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

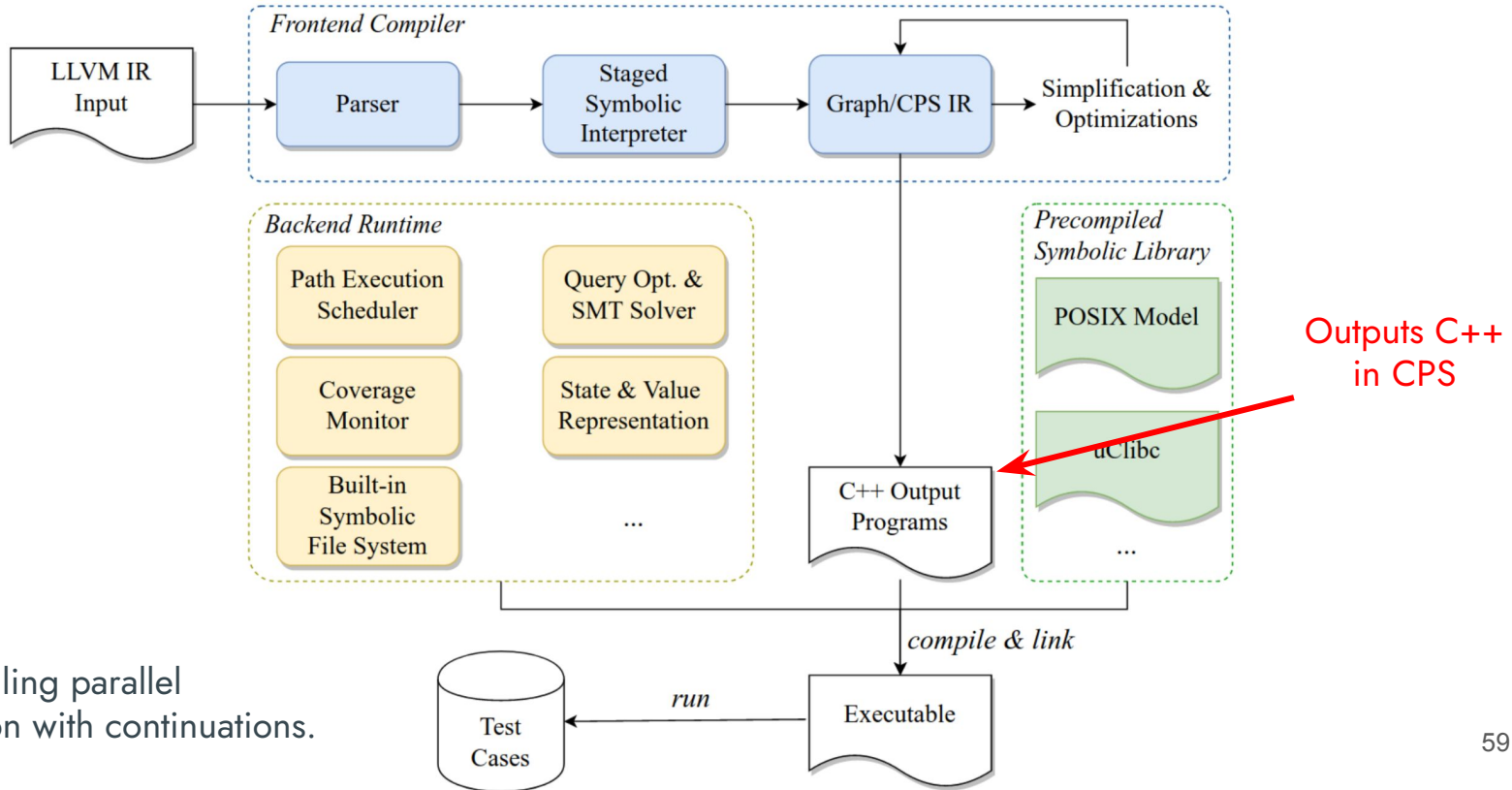
GenSym

Written in
Scala/LMS



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

GenSym

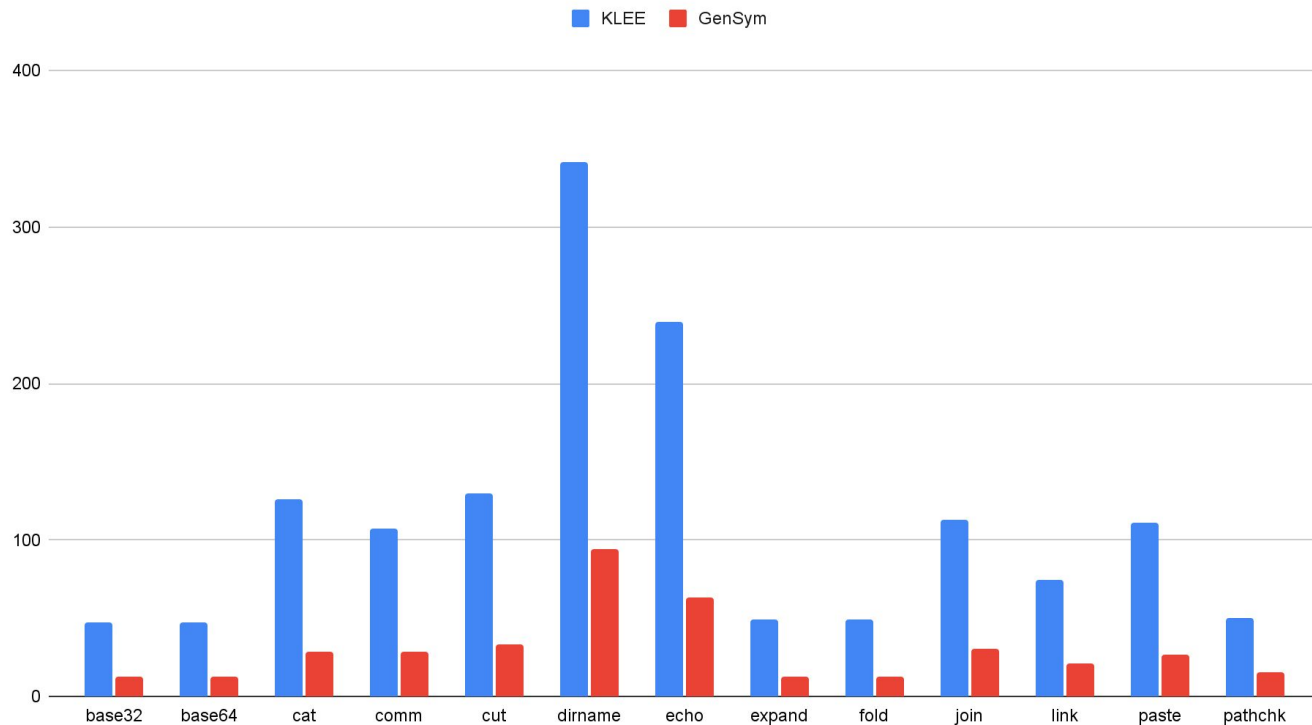


[ICSE '23] Compiling parallel symbolic execution with continuations.

GenSym: Performance Evaluation

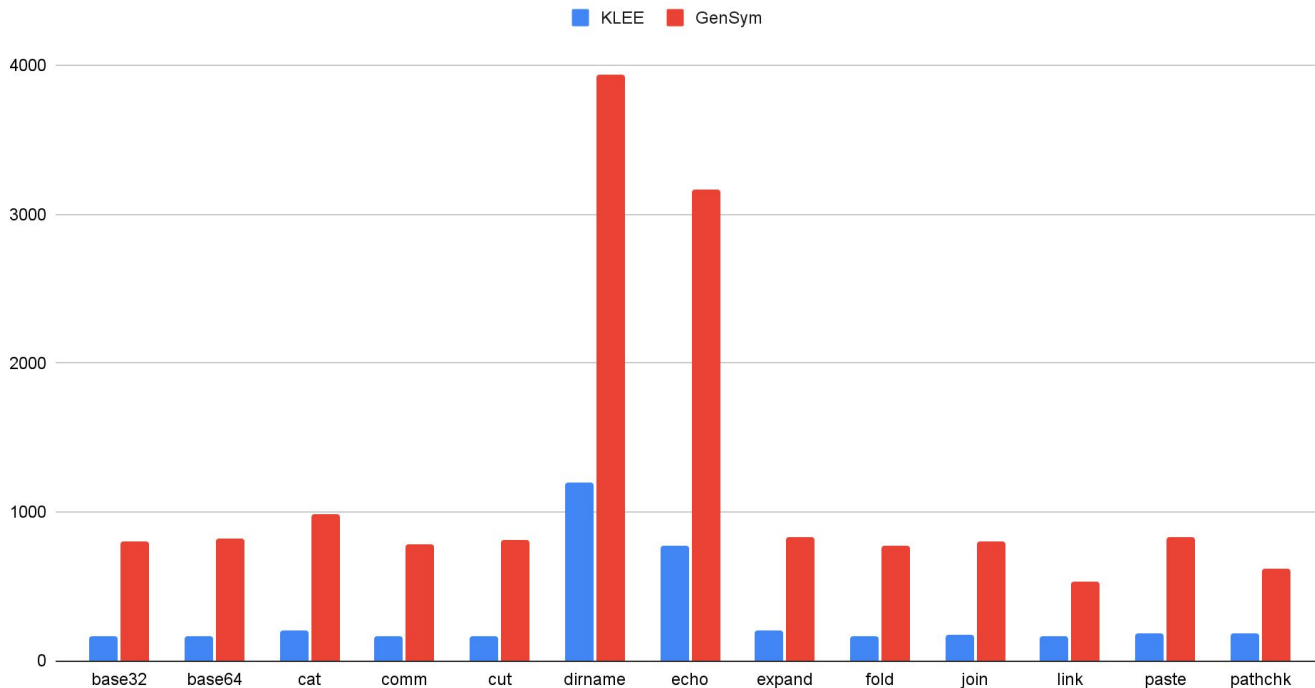
- KLEE: state-of-the-art symbolic interpreter for LLVM IR
 - Actively developed over 15+ years
 - Written in C++
- Evaluated on a set of GNU Coreutils programs
 - Using POSIX file system and uClibc library
 - Average program size: 28k LOC of LLVM IR instructions

Single-thread Pure Execution



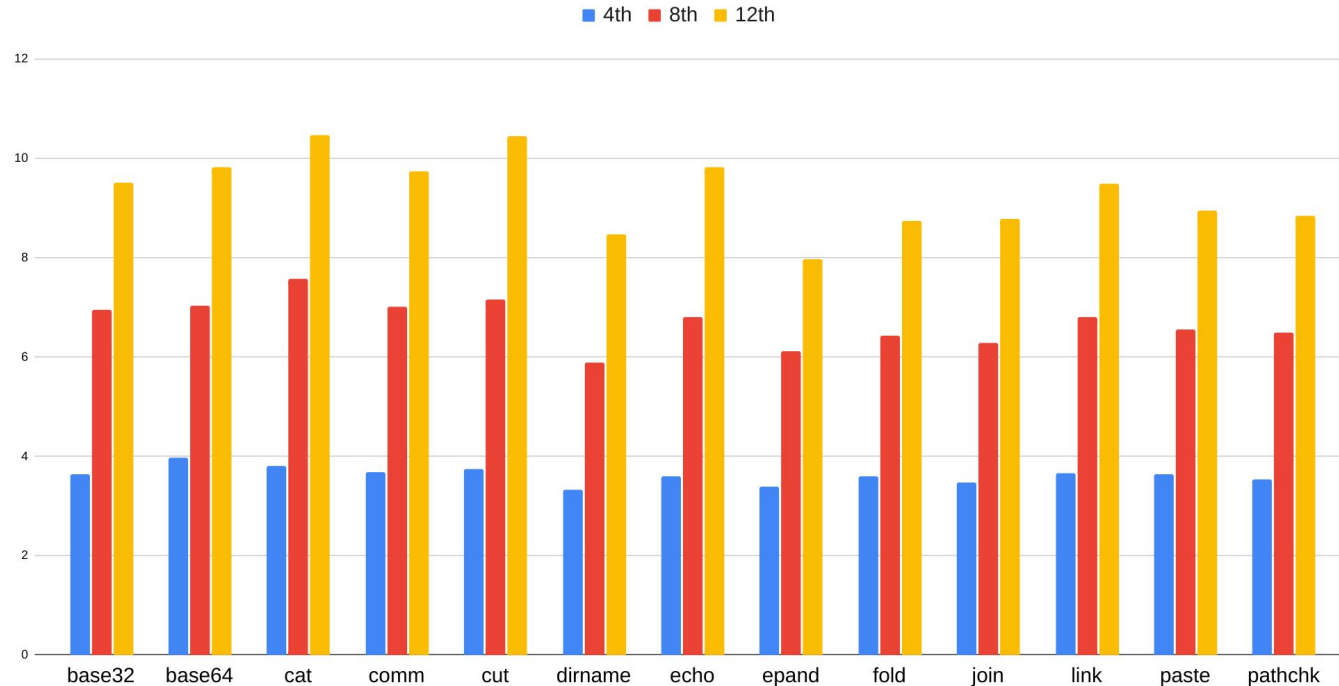
~4x speedups

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
 - Branching 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
 - Branching as concurrency/parallelism
 - Path-selection heuristics

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.