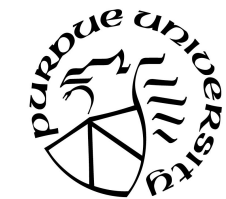




# *Compiling Parallel Symbolic Execution with Continuations*

**Guannan Wei<sup>P</sup>**, Songlin Jia<sup>P</sup>, Ruiqi Gao<sup>P</sup>, Haotian Deng<sup>P</sup>,  
Shangyin Tan<sup>B</sup>, Oliver Bračevac<sup>P</sup>, and Tiark Rompf<sup>P</sup>

<sup>P</sup>Purdue University, <sup>B</sup>UC Berkeley



ICSE 2023 – Remote Presentation



# Symbolic Execution

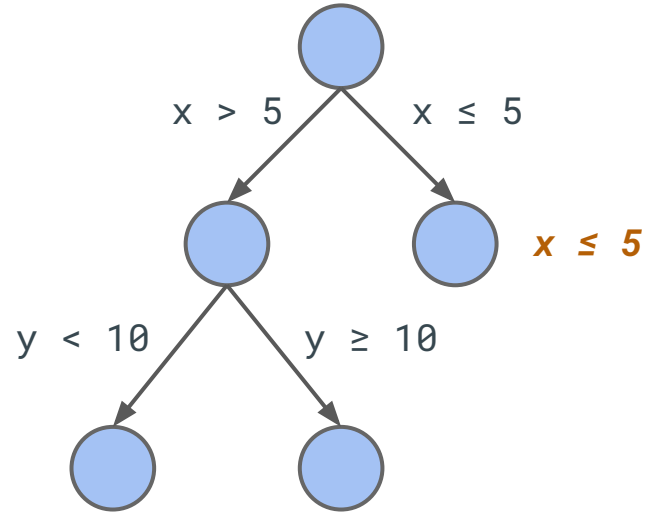
*mark as  
symbolic*

```
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 */
}
```



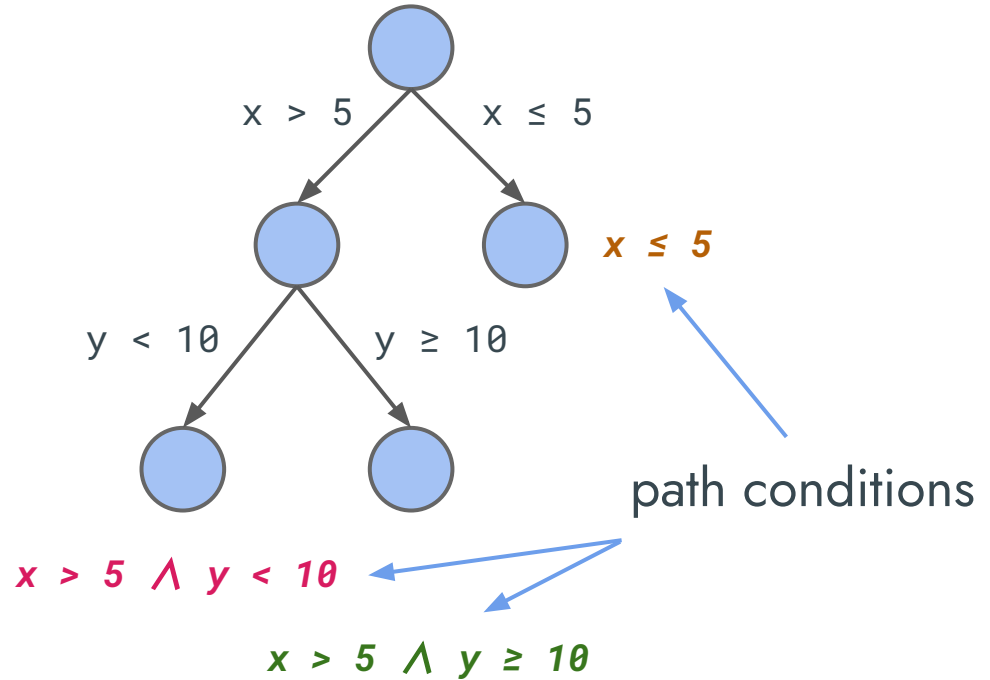
$x > 5 \wedge y < 10$

$x > 5 \wedge y \geq 10$

# Symbolic Execution

mark as  
symbolic

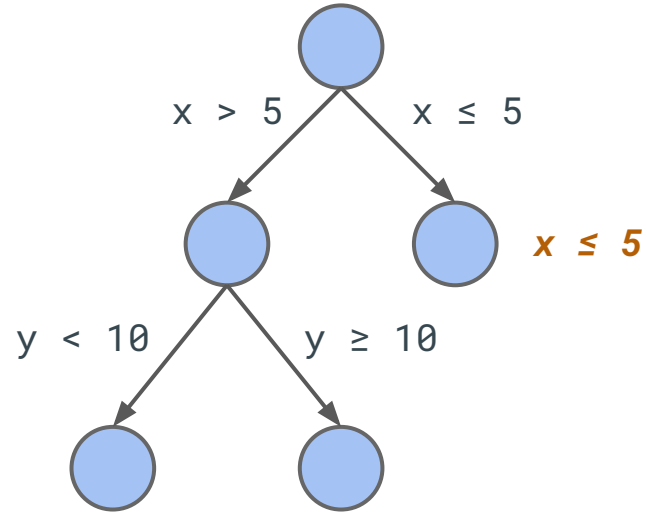
```
x = user_input()
y = user_input()
if (x > 5) {
  if (y < 10) {
    ... /* path 1 */
  } else {
    ... /* path 2 */
  }
} else {
  ... /* path 3 */
}
```



# Symbolic Execution

mark as  
symbolic

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

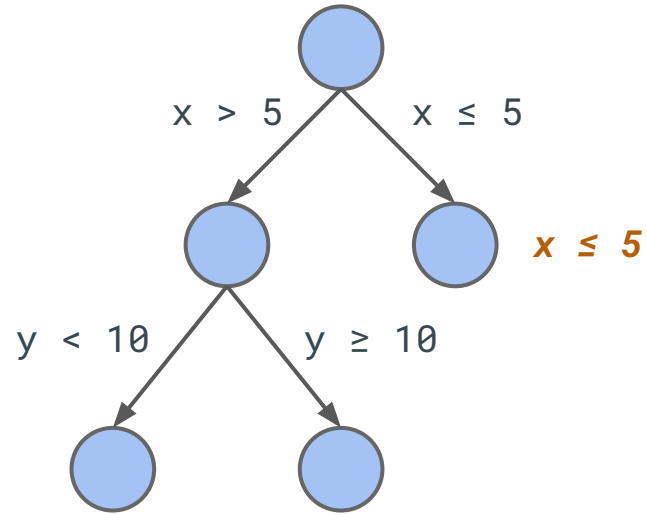


$$\text{solver}(x > 5 \wedge y < 10) = \{ x = 6, y = 9 \}$$

# Symbolic Execution

mark as  
symbolic

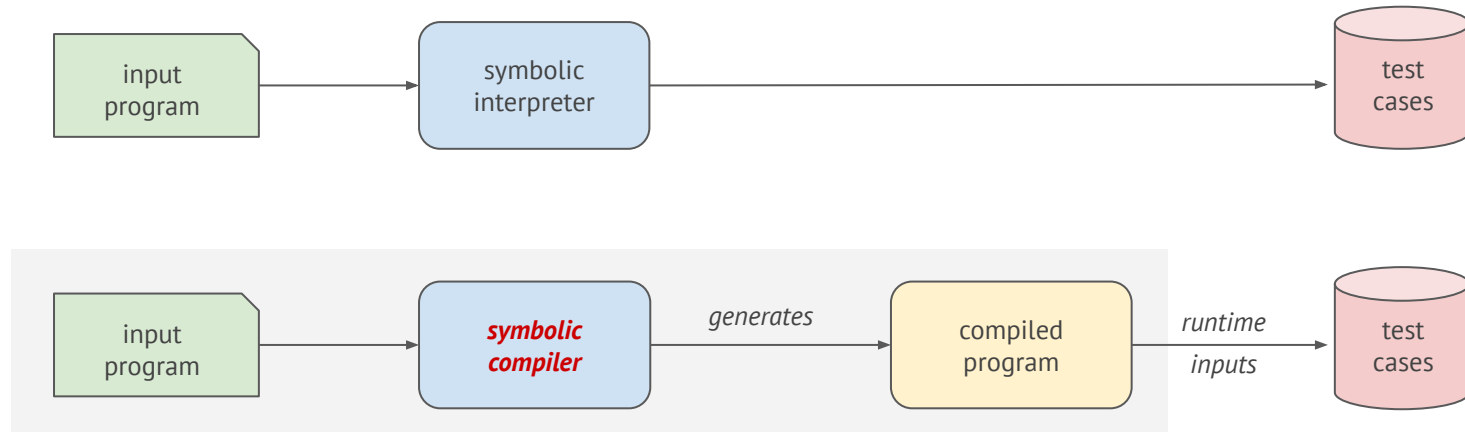
```
x = user_input()
y = user_input()
if (x > 5) {
  if (y < 10) {
    ... /* path 1 */
  } else {
    ... /* path 2 */
  }
} else {
  ... /* path 3 */
}
```



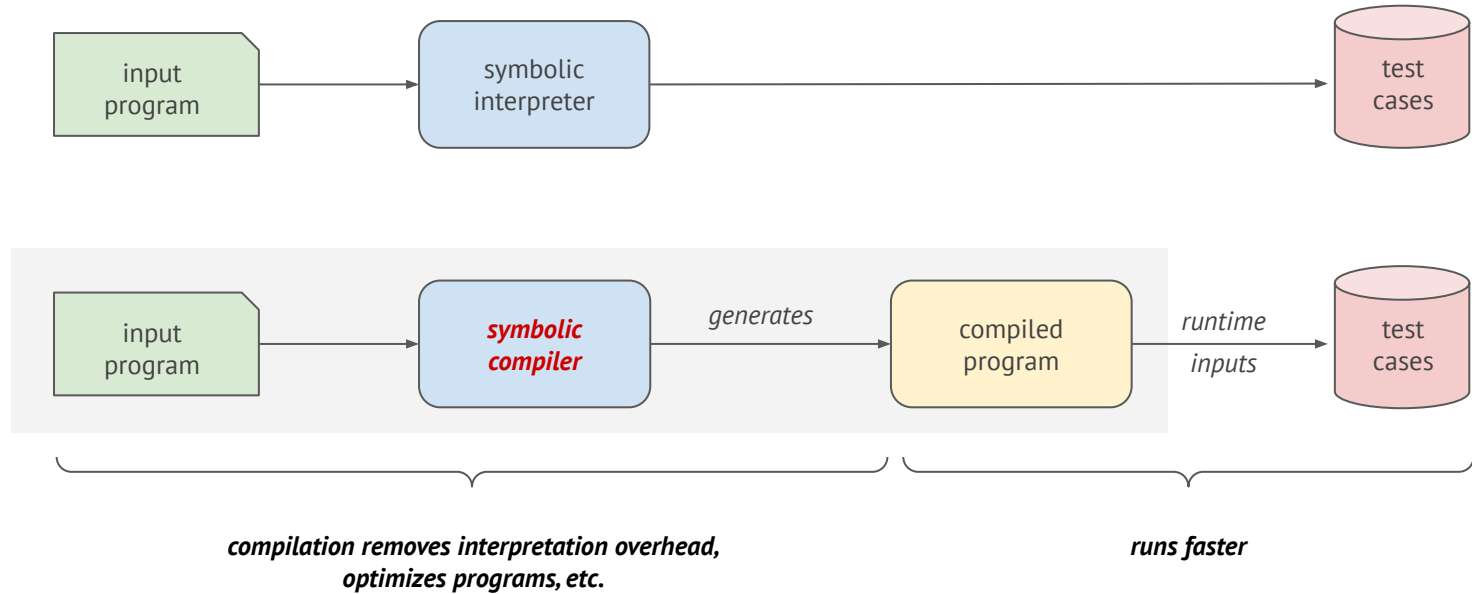
$$\text{solver}(x > 5 \wedge y < 10) = \{ x = 6, y = 9 \}$$

Useful in program testing, verification, bug finding, etc.

# *Compiling* Symbolic Execution

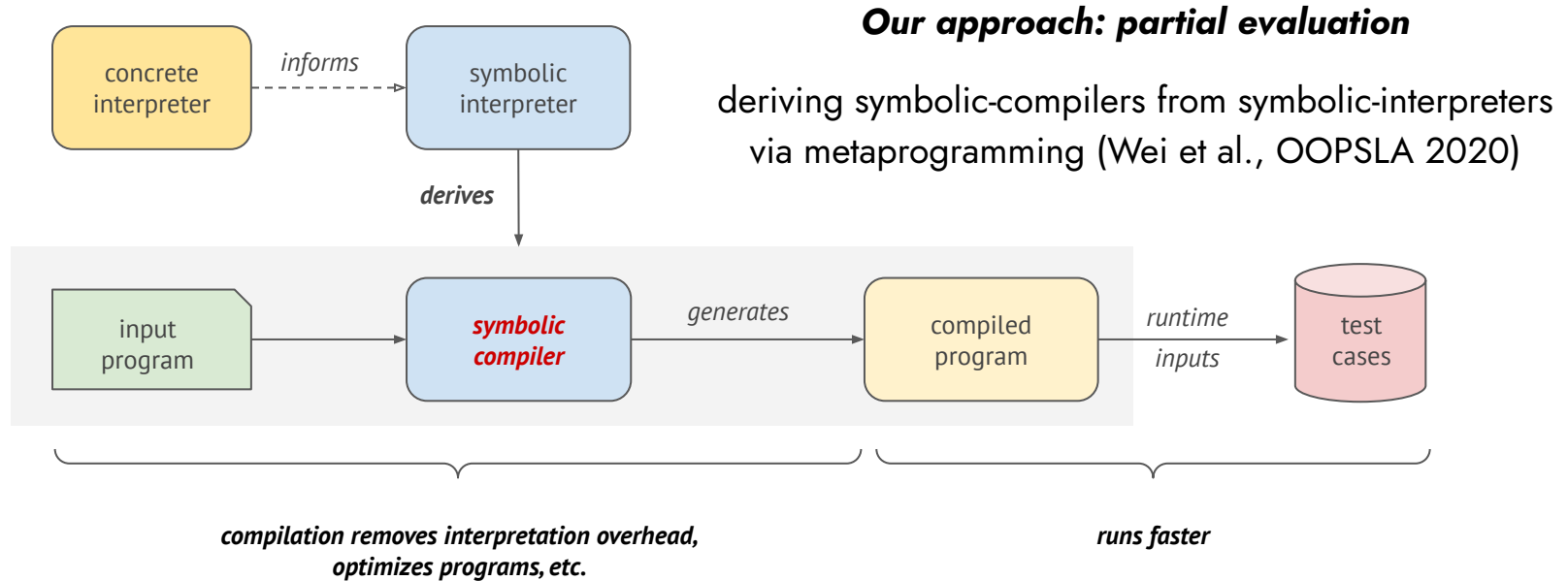


# *Compiling* Symbolic Execution





# *Compiling* Symbolic Execution



# Compiling Symbolic Execution

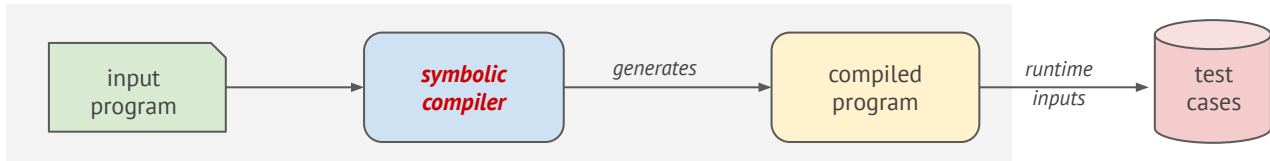
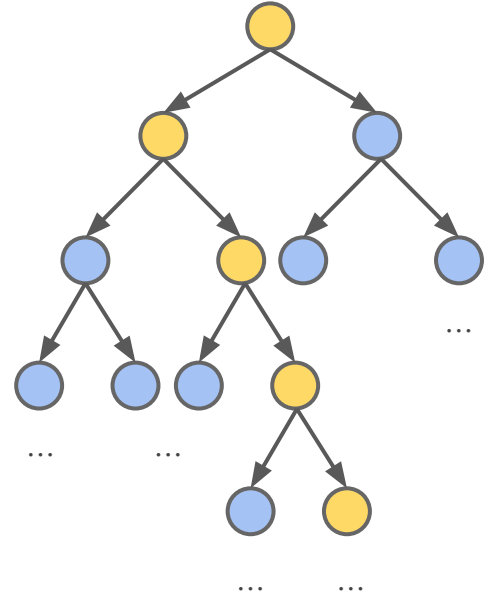
Concrete Execution

1 path

vs

Symbolic Execution

exponential number of independent paths



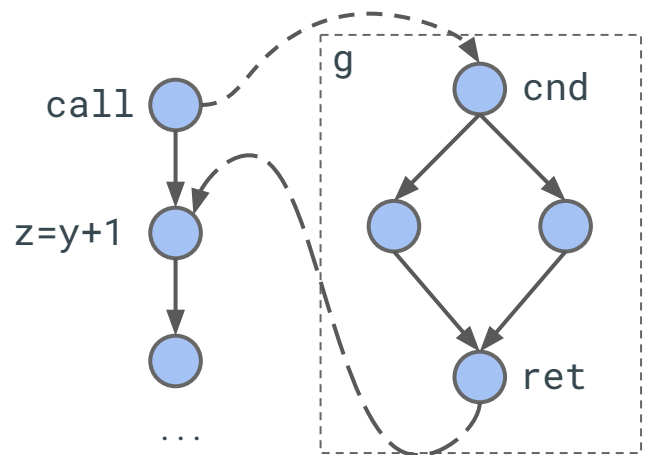




# Compiling Symbolic Execution with *Continuations*

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

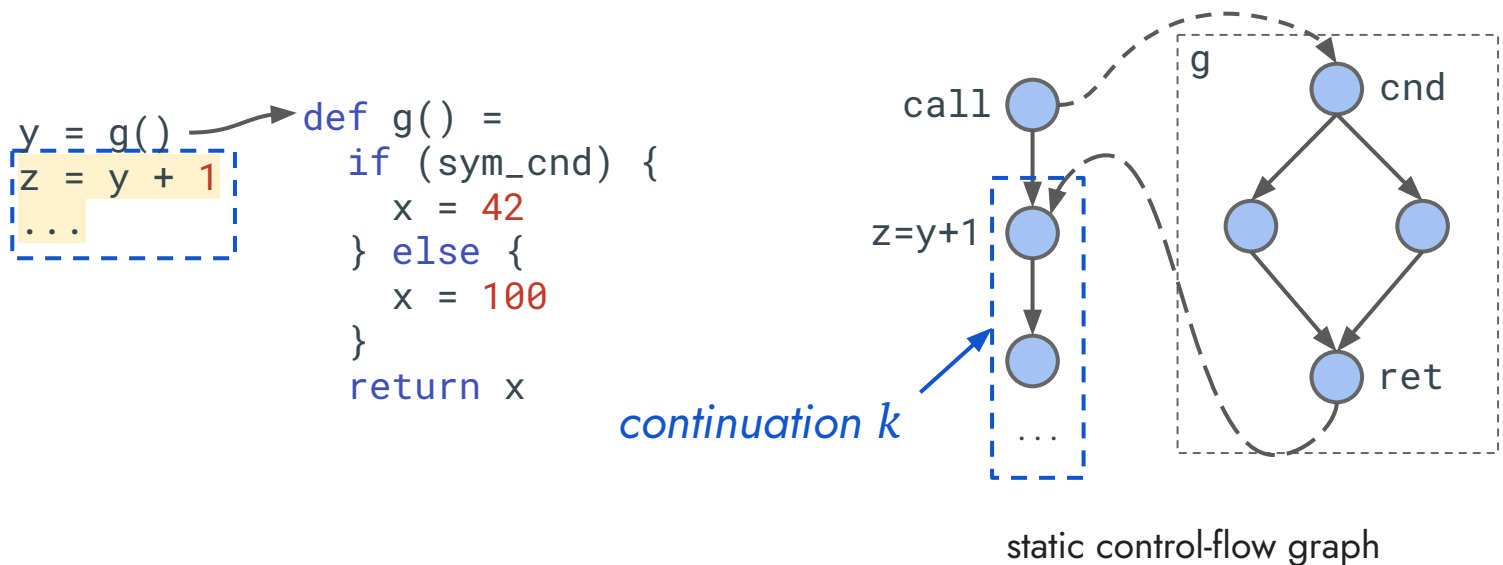
```
y = g()
z = y + 1
...
def g() =
  if (sym_cnd) {
    x = 42
  } else {
    x = 100
  }
  return x
```



static control-flow graph

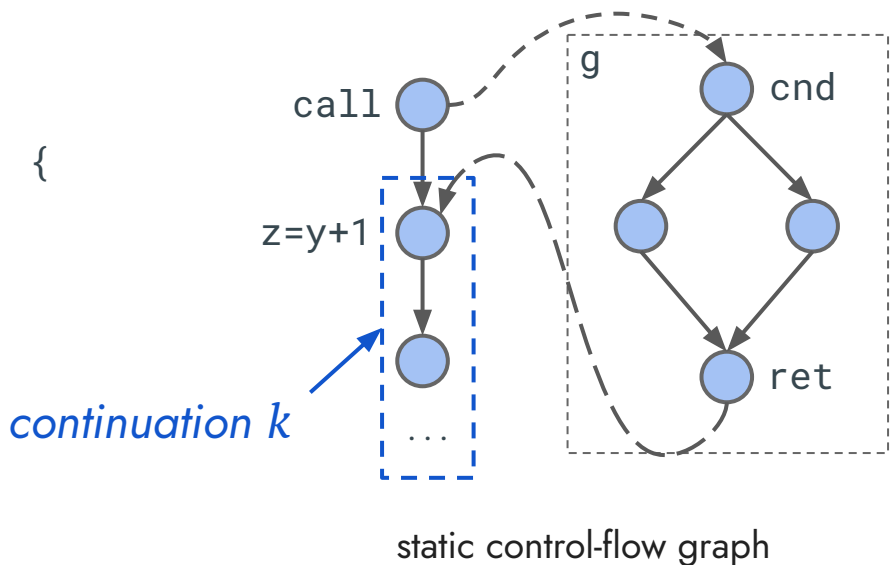
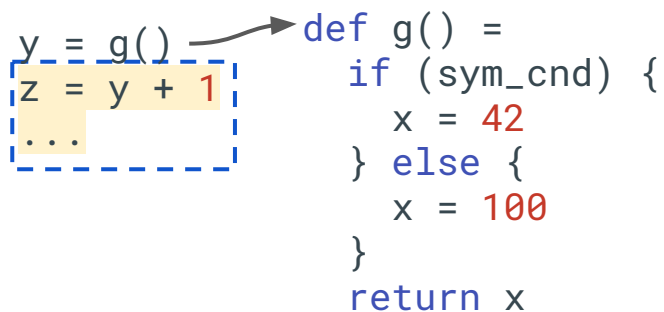
# Compiling Symbolic Execution with *Continuations*

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



# Compiling Symbolic Execution with *Continuations*

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



**Invoke and fork**

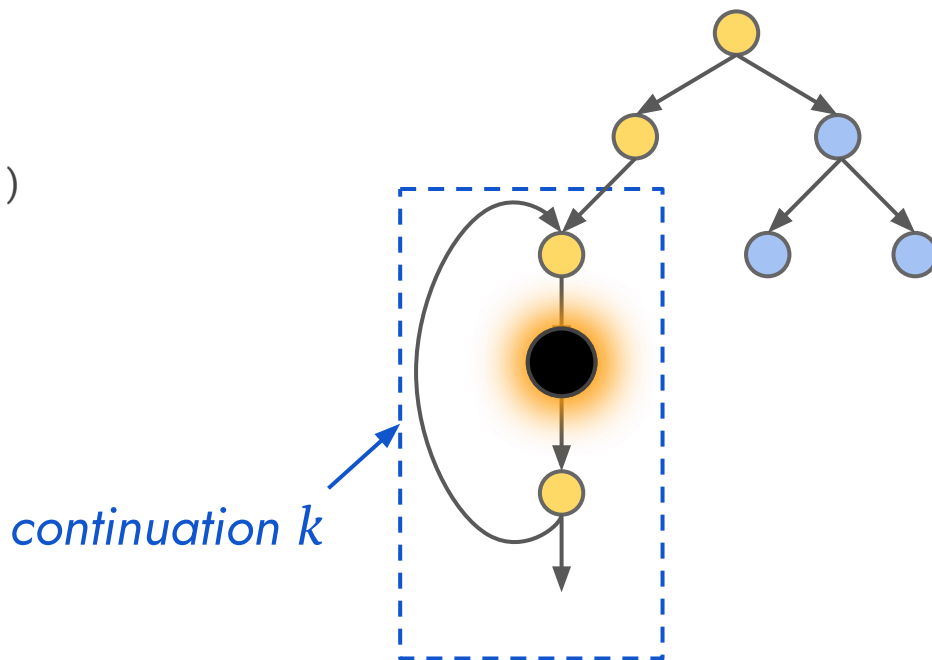
$k(s1); k(s2)$

# Compiling Symbolic Execution with *Continuations*

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

## **Save and pause**

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





# Compiling Symbolic Execution with *Continuations*

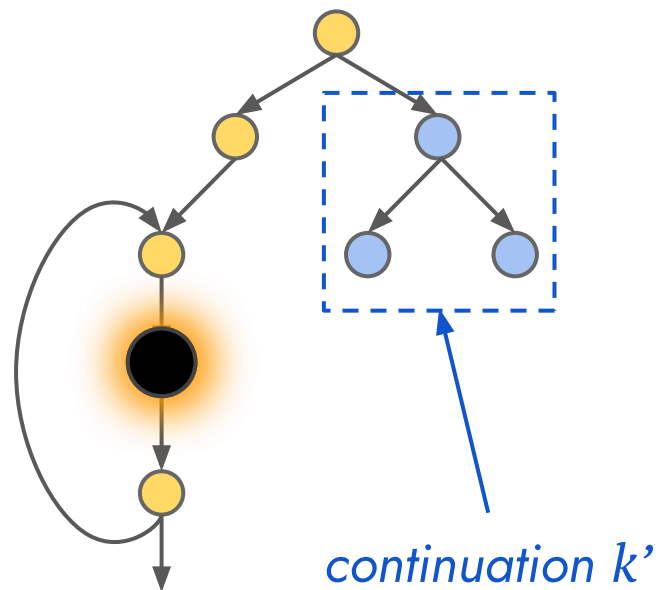
Represent the rest of the execution as a function *k* in the generated code

## **Save and pause**

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

## **Dispatch and resume**

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



# Compiling Symbolic Execution with *Continuations*

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

## **Save and pause**

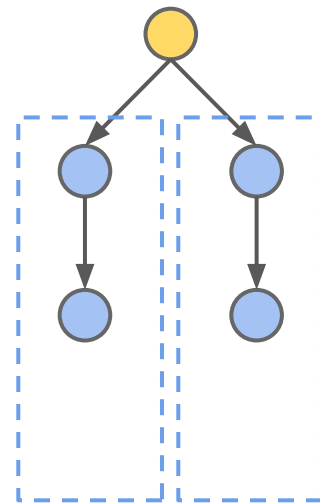
```
scheduler.put(() => k(s))
```

## **Dispatch and resume**

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

## **Parallelism for Free**

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



```
scheduler.put(k1) scheduler.put(k2)
```

# GenSym Implementation & Evaluation

- **GenSym**
  - Compiles LLVM IR to C++
  - Written in Scala/LMS as a staged symbolic interpreter
  - Implements path forking, switching, and parallelism using continuations

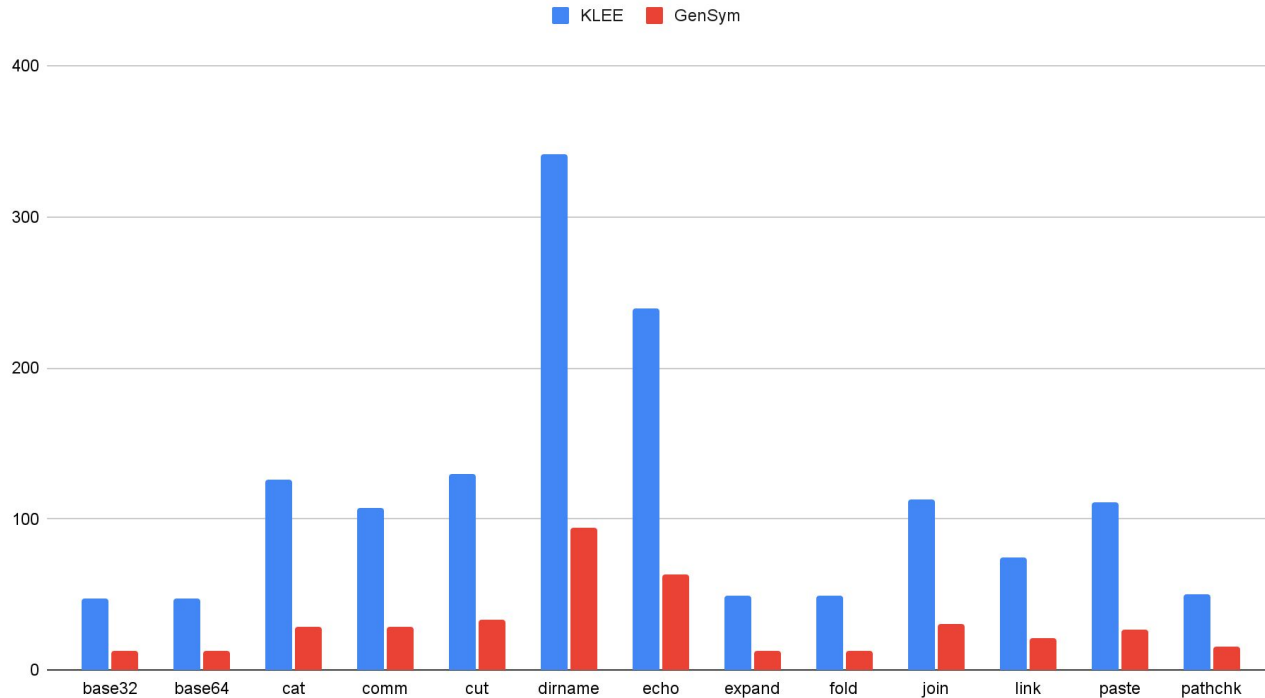
# GenSym Implementation & Evaluation

- **GenSym**
  - Compiles LLVM IR to C++
  - Written in Scala/LMS as a staged symbolic interpreter
  - Implements path forking, switching, and parallelism using continuations
- **Benchmarks**
  - A subset of GNU Coreutils (using POSIX file system and uClibc library)
  - Average program size 28334 LOC of LLVM IR

# GenSym Implementation & Evaluation

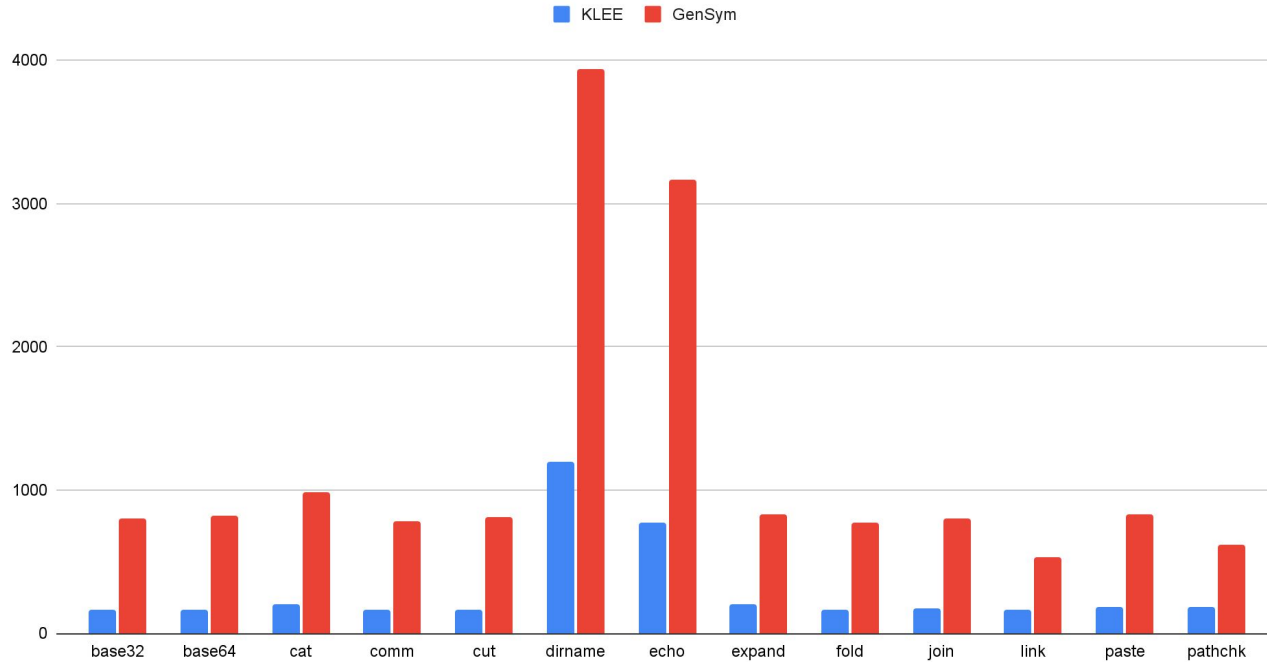
- **GenSym**
  - Compiles LLVM IR to C++
  - Written in Scala/LMS as a staged symbolic interpreter
  - Implements path forking, switching, and parallelism using continuations
- **Benchmarks**
  - A subset of GNU Coreutils (using POSIX file system and uClibc library)
  - Average program size 28334 LOC of LLVM IR
- **Performance evaluation**
  - Compared with KLEE, a state-of-the-art symbolic interpreter

# GenSym - Evaluation



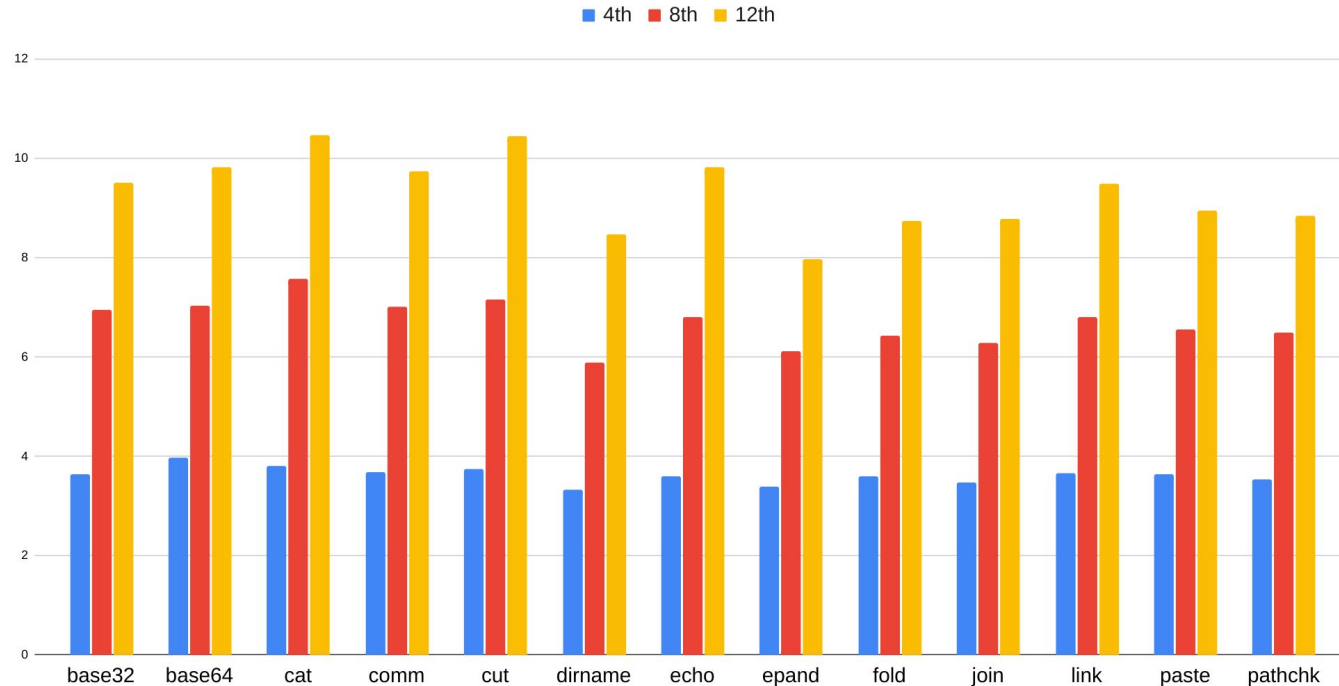
Single-thread pure execution time (sec)  
excluding solver: **4x faster on avg**

# GenSym - Evaluation



Single-thread path throughput  
(paths per second) of 1-hour running: **4.3x on avg.**

# GenSym - Evaluation



Speedups using more threads

4th - 3.6x

8th - 6.7x


12th - 9.3x



# More in the Paper

- Design and Implementation
- An example of compiled code
- Parallelism
- Compile-time optimizations
- Generative environment modeling
- Evaluation details (optimizations, compilation overhead, etc.)

2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)



## Compiling Parallel Symbolic Execution with Continuations

Guannan Wei\*, Songlin Jia\*, Ruiqi Gao\*, Haotian Deng\*, Shangyin Tan<sup>†</sup>, Oliver Bračevac\*, Tiark Rompf\*  
\*Department of Computer Science, Purdue University West Lafayette, IN, USA  
{guannanwei,jia137,gao606,deng254,bracevac,tiark}@purdue.edu  
<sup>†</sup>Department of EECS, UC Berkeley Berkeley, CA, USA  
shangyin@berkeley.edu

*Abstract*—Symbolic execution is a powerful program analysis and testing technique. Symbolic execution engines are usually implemented as interpreters, and the induced interpretation overhead can dramatically inhibit performance. Alternatively, implementation choices based on instrumentation provide a limited ability to transform programs. However, the use of compilation and code generation techniques beyond simple instrumentation remains underexplored for engine construction, leaving potential performance gains untapped.

In this paper, we show how to tap some of these gains using sophisticated compilation techniques: We present GENSYM, an optimizing symbolic-execution compiler that generates symbolic code which explores paths and generates tests in parallel. The key insight of GENSYM is to compile symbolic execution tasks into cooperative concurrency via continuation-passing style, which further enables efficient parallelism. The design and implementation of GENSYM is based on partial evaluation and generative programming techniques, which make it high-level and performant at the same time. We compare the performance of GENSYM against the prior symbolic-execution compiler LLSC and the state-of-the-art symbolic interpreter KLEE. The results show an average 4.6× speedup for sequential execution and 9.4× speedup for parallel execution on 20 benchmark programs.

*Index Terms*—symbolic execution, compiler, code generation, metaprogramming, continuation

as compilers. This practice misses opportunities to further improve the performance of SE and advance the development of engines.

In this paper, we study constructing scalable SE engines using code generation and compiler techniques. We present the design, implementation, and evaluation of GENSYM, a symbolic-execution compiler for the LLVM intermediate representation (IR). Given an input LLVM IR program, GENSYM generates C++ code that schedules parallel path exploration and orchestrates SMT solver invocations. Running this C++ program generates test cases for explored paths and failed assertions. The novel contribution of GENSYM is its aggressive use of code generation techniques for a *systematic, principled, and high-level* construction process yielding *optimized and performant* symbolic execution engines.

**From Interpreters to Compilers.** Interpreters are an easy and high-level approach for building symbolic execution engines. However, a naively built interpreter without carefully engineered optimizations can be orders of magnitude slower than a compiled program. One of the major contributors to the slowdown is the interpretation overhead [14] from inspecting

# Conclusion

*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
- ◆ Allow using path-selection heuristics



◆ GenSym: artifact available and reusable  
<https://continuation.passing.style/GenSym>

