Program Testing via Symbolic Execution

Daniel Dunbar

Introduction

- Motivation
    - Manual testing is difficult

Introduction

- Motivation
    - Manual testing is difficult
        - Requires knowledge of code

Introduction

- Motivation
  - Manual testing is difficult
    - Requires knowledge of code
    - Requires constant maintenance

Introduction

- Motivation
  - Manual testing is difficult
    - Requires knowledge of code
    - Requires constant maintenance
  - Random testing is ineffective
    - Hard to deeply probe programs

Introduction

- Motivation
  - Manual testing is difficult
    - Requires knowledge of code
    - Requires constant maintenance
  - Random testing is ineffective
    - Hard to deeply probe programs
- Symbolic execution?
  - Automatic
  - Can check for error conditions
  - Can reach deep code paths

Overview

- KLEE: A Symbolic Virtual Machine
  - Basic Architecture
  - Constraint Solver Optimizations

Overview

- KLEE: A Symbolic Virtual Machine
  - Basic Architecture
  - Constraint Solver Optimizations
- A Modeling Environment for UNIX

Overview

- KLEE: A Symbolic Virtual Machine
  - Basic Architecture
  - Constraint Solver Optimizations
- A Modeling Environment for UNIX
- Case Study: Coreutils
  - Over 80% coverage on majority of tools
  - Overall coverage better than developer tests

What is Symbolic Execution?

What is Symbolic Execution?

- Symbolic

  Replace *concrete* program values with *symbolic* variables

What is Symbolic Execution?

- Symbolic
  Replace *concrete* program values with *symbolic* variables
- Execution
  Program instructions become operations on *expressions*

What is Symbolic Execution?

- Symbolic

  Replace *concrete* program values with *symbolic* variables

- Execution

  Program instructions become operations on *expressions*

- Also:

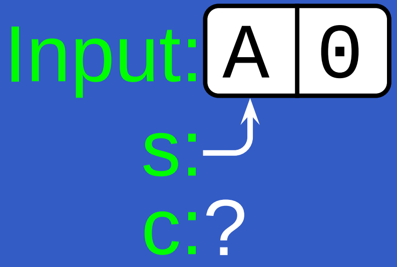
  Record branch choices in a *path condition*

What is Symbolic Execution?

```
void escape(char *s,
            char *out) {
  while (*s != 0) {
    char c = *s++;
    if (c == ’\\’)
      c = *s++;
    *out++ = c;
  }
}
```

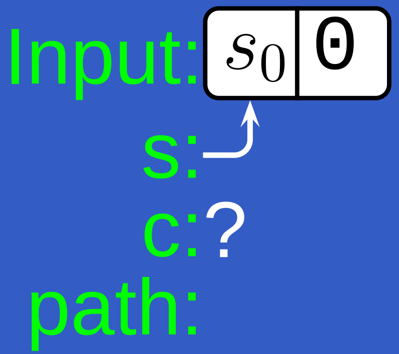
What is Symbolic Execution?

```
→ void escape(char *s,
              char *out) {
    while (*s != 0) {
      char c = *s++;
      if (c == ’\\’)
        c = *s++;
      *out++ = c;
    }
  }
```



What is Symbolic Execution?

```
→ void escape(char *s,
                    char *out) {
    while (*s != 0) {
      char c = *s++;
      if (c == ’\\’)
        c = *s++;
      *out++ = c;
    }
  }
```



Input: s_0 | 0

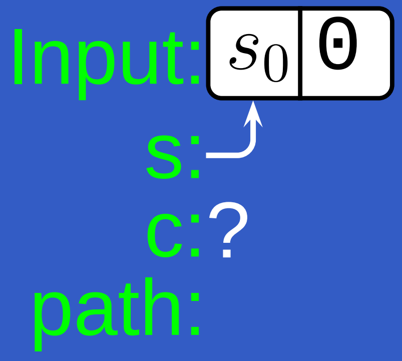
s:

c:?

path:

What is Symbolic Execution?

```
  void escape(char *s,
              char *out) {
→   while (*s != 0) {
      char c = *s++;
      if (c == '\\')
        c = *s++;
      *out++ = c;
    }
  }
```



Input: s_0 | 0

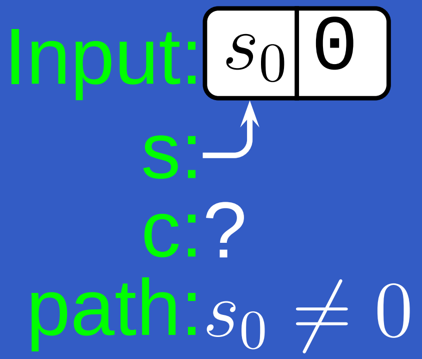
s:

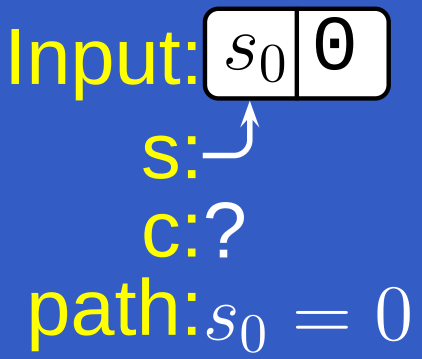
c:?

path:

What is Symbolic Execution?

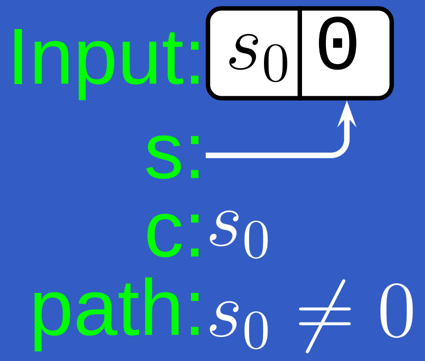
```
  void escape(char *s,
              char *out) {
    while (*s != 0) {
→     char c = *s++;
      if (c == ’\\’)
        c = *s++;
      *out++ = c;
    }
→ }
```





What is Symbolic Execution?

```
void escape(char *s,
            char *out) {
  while (*s != 0) {
    char c = *s++;
    if (c == '\\')
      c = *s++;
    *out++ = c;
  }
}
```



Input: $\boxed{s_0 \mid 0}$

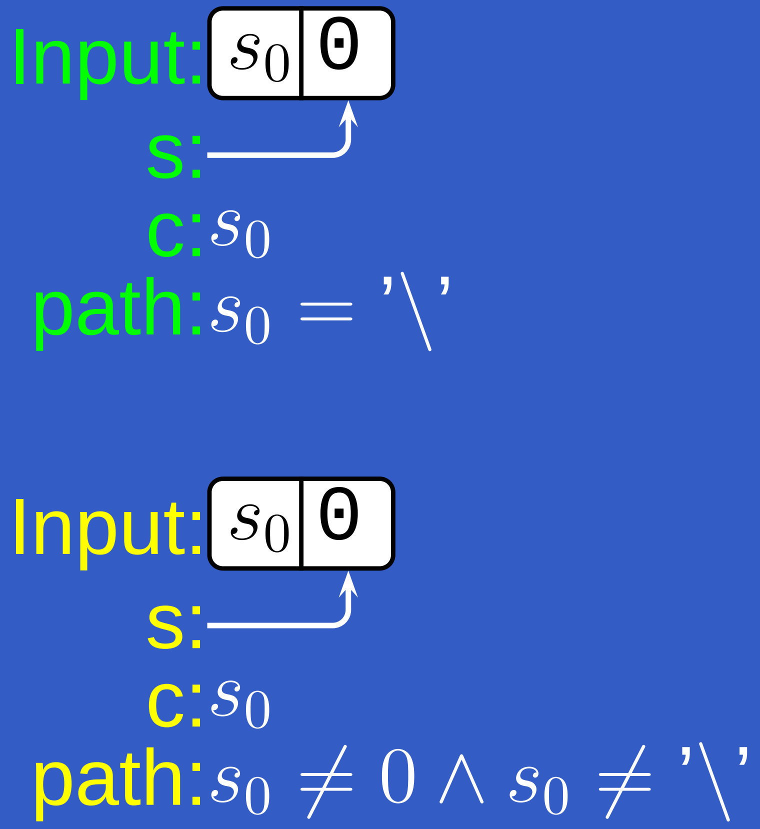
s:

c: s_0

path: $s_0 \neq 0$

What is Symbolic Execution?

```
   void escape(char *s,
               char *out) {
     while (*s != 0) {
       char c = *s++;
       if (c == '\\')
→        c = *s++;
→      *out++ = c;
     }
   }
```

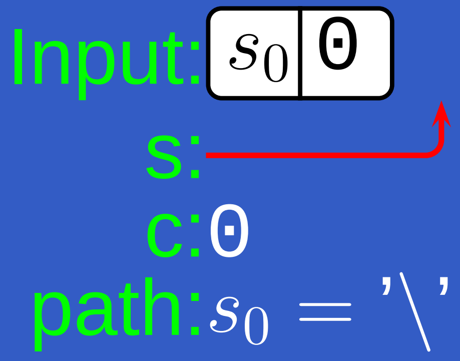


Input: $\boxed{s_0 \mid 0}$
s:
c: s_0
path: $s_0 = '\backslash'$

Input: $\boxed{s_0 \mid 0}$
s:
c: s_0
path: $s_0 \neq 0 \wedge s_0 \neq '\backslash'$

What is Symbolic Execution?

```
  void escape(char *s,
              char *out) {
    while (*s != 0) {
      char c = *s++;
      if (c == ’\\’)
        c = *s++;
→     *out++ = c;
    }
  }
```



Input: $\boxed{s_0 \mid 0}$

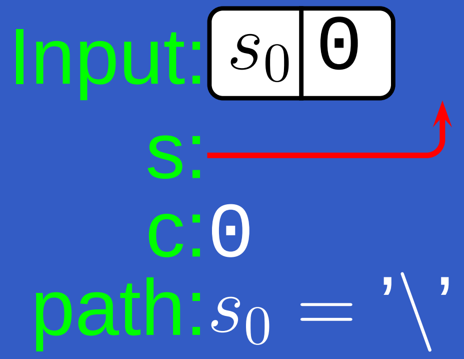
s:

c:0

path: $s_0 = '\backslash'$

What is Symbolic Execution?

```
void escape(char *s,
            char *out) {
  while (*s != 0) {
    char c = *s++;
    if (c == '\\')
      c = *s++;
    *out++ = c;
  }
}
```



Input: s_0 0

s:

c:0

path:$s_0 = '\backslash'$

The KLEE Architecture

- Based on experience with EXE system
- Goals
  - Simplicity
  - Scalability
  - Speed

The KLEE Architecture: Simplicity

- Interpreter for LLVM Assembly Language
    - Precise semantics
    - Multiple language support
    - Mature binary tools

The KLEE Architecture: Simplicity

- Interpreter for LLVM Assembly Language
    - Precise semantics
    - Multiple language support
    - Mature binary tools
- Easy to understand
    - Simple interpreter loop
    - Explicit process representation
    - Modular search & constraint solving

The KLEE Architecture: Scalability

- Fine-grained heap memory
  - Object level copy-on-write

The KLEE Architecture: Scalability

- Fine-grained heap memory
  - Object level copy-on-write
- Careful data structure selection
  - Persistent heap, expressions, path data
  - Compact expression representation

The KLEE Architecture: Scalability

- Fine-grained heap memory
    - Object level copy-on-write
- Careful data structure selection
    - Persistent heap, expressions, path data
    - Compact expression representation
- Handle over 100k processes for small programs

The KLEE Architecture: Speed

- Constraint solving dominates run-time

The KLEE Architecture: Speed

- Constraint solving dominates run-time
- Keep constraints simple
  - Canonicalize simple forms
  - Cache concrete values and indices
  - Dynamically rewrite path constraints

The KLEE Architecture: Speed

- Constraint solving dominates run-time
- Keep constraints simple
  - Canonicalize simple forms
  - Cache concrete values and indices
  - Dynamically rewrite path constraints
- Optimize constraint solving

Solver Optimization

- Take advantage of the nature of symbolic execution

Solver Optimization

- Take advantage of the nature of symbolic execution
- Program decomposition
  - Queries deal with distinct input variables

Solver Optimization

- Take advantage of the nature of symbolic execution
- Program decomposition
  - Queries deal with distinct input variables
- Considerable redundancy
  - Processes *accrue* constraints
  - Static set of branches

Independent Constraint Opt.

- Path constraints may refer to many variables
  - Path: $s_0 = \text{'A'} \wedge s_1 = 0 \wedge y_0 < 100$

Independent Constraint Opt.

- Path constraints may refer to many variables
  - Path: $s_0 = \text{'A'} \wedge s_1 = 0 \wedge y_0 < 100$
- Individual expressions only refer to a few
  - Branch condition: $y = 10$

Independent Constraint Opt.

- Path constraints may refer to many variables
  - Path: $s_0 = \text{'A'} \wedge s_1 = 0 \wedge y_0 < 100$
- Individual expressions only refer to a few
  - Branch condition: $y = 10$
- Independent constraint optimization:
  - Group constraints into disjoint subsets based on referenced variables
  - Only pass dependent constraints to solver
  - Query: $y < 100 \Rightarrow y = 10$

Counterexample Cache

- Take advantage of “free” counterexamples

Counterexample Cache

- Take advantage of “free” counterexamples
- Cache $C \mapsto \{A, \perp\}$, where C is a set of constraints and A a satisfying assignment

Counterexample Cache

- Take advantage of “free” counterexamples
- Cache $C \mapsto \{A, \perp\}$, where C is a set of constraints and A a satisfying assignment
- When performing a query with constraints C:

Counterexample Cache

- Take advantage of “free” counterexamples
- Cache $C \mapsto \{A, \bot\}$, where C is a set of constraints and A a satisfying assignment
- When performing a query with constraints C:

1. If (C, x) in cache, return x

Counterexample Cache

- Take advantage of “free” counterexamples
- Cache $C \mapsto \{A, \perp\}$, where C is a set of constraints and A a satisfying assignment
- When performing a query with constraints C:

1. If (C, x) in cache, return x
2. If $(C', \perp), C' \subset C$ in cache, return $\perp$

Counterexample Cache

- Take advantage of "free" counterexamples
- Cache $C \mapsto \{A, \perp\}$, where C is a set of constraints and A a satisfying assignment
- When performing a query with constraints C:

1. If (C, x) in cache, return x
2. If $(C', \perp), C' \subset C$ in cache, return $\perp$
3. If $(C', A), C' \supset C$ in cache, return A

Counterexample Cache

- Take advantage of “free” counterexamples
- Cache $C \mapsto \{A, \perp\}$, where C is a set of constraints and A a satisfying assignment
- When performing a query with constraints C:

1. If (C, x) in cache, return x
2. If $(C', \perp), C' \subset C$ in cache, return $\perp$
3. If $(C', A), C' \supset C$ in cache, return A
4. Otherwise, for each $(C', A), C' \subset C$ in cache, *speculatively* evaluate C using A

Counterexample Cache Example

```
void foo(int x) {
  if (x < 100)
    ...
  if (x != 50)
    ...
}
```

Counterexample Cache Example

```
void foo(int x) {
  if (x < 100)
    ...
  if (x != 50)
    ...
}
```

- Cache: $\{\}$

Counterexample Cache Example

```
  void foo(int x) {
→   if (x < 100)
      ...
    if (x != 50)
      ...
  }
```

- Cache: $\{\}$
- Query: $x < 100$, Cache: $\{(x < 100, \{x : 0\})\}$

Counterexample Cache Example

```
void foo(int x) {
  if (x < 100)
    ...
  if (x != 50)
    ...
}
```

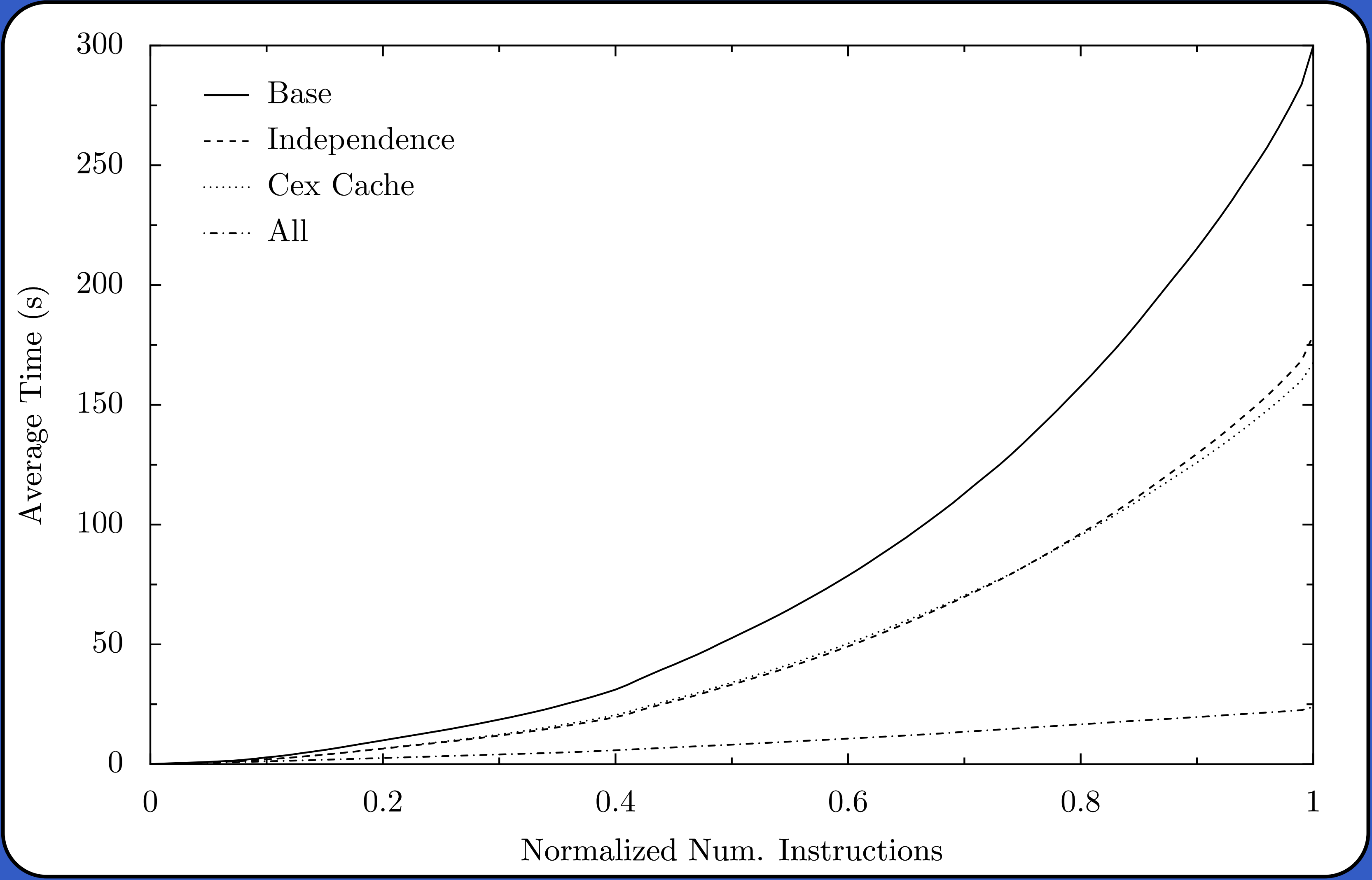
- Cache: $\{\}$
- Query: $x < 100$, Cache: $\{(x < 100, \{x : 0\})\}$
- Query: $x < 100 \wedge x \neq 50$

Counterexample Cache Example

```
  void foo(int x) {
    if (x < 100)
      ...
→   if (x != 50)
      ...
  }
```

- Cache: $\{\}$
- Query: $x < 100$, Cache: $\{(x < 100, \{x : 0\})\}$
- Query: $x < 100 \wedge x \neq 50$
  - Evaluate with $\{x : 0\}$
  - $0 < 100 \wedge 0 \neq 50$
  - Found assignment

Optimization Results



Environment Modeling

- Testing real applications requires providing a realistic environment
  - system libraries
  - operating system

Environment Modeling

- Testing real applications requires providing a realistic environment
  - system libraries
  - operating system
- Our approach:
  - Compile and execute using μlibc
  - Implement UNIX systems calls in separate modeling library

Environment Modeling (2)

- Idea: Overlay *symbolic* environment onto operating system
  - Program sees “virtual” symbolic resources and actual OS resources

Environment Modeling (2)

- Idea: Overlay *symbolic* environment onto operating system
  - Program sees “virtual” symbolic resources and actual OS resources
- Implemented by routing system calls through model routines
  - Symbolic resources are handled in model
  - Actual resources are forwarded to OS

Environment Modeling (3)

- Currently model supports symbolic files and program arguments

Environment Modeling (3)

- Currently model supports symbolic files and program arguments
- Supports `open()`, `close()`, `read()`, `write()`, `lseek()`, `stat()`, `fstat()` as well as simple stubs for several more

Environment Modeling (3)

- Currently model supports symbolic files and program arguments
- Supports `open()`, `close()`, `read()`, `write()`, `lseek()`, `stat()`, `fstat()` as well as simple stubs for several more
- Implementation is about 800 lines of C code

Testing Process

- Test generation:
    - The application is linked with μclibc and the model and run with KLEE

Testing Process

- Test generation:
  - The application is linked with μclibc and the model and run with KLEE
  - Additional arguments specify what parts of environment to make symbolic

Testing Process

- Test generation:
    - The application is linked with μclibc and the model and run with KLEE
    - Additional arguments specify what parts of environment to make symbolic
    - KLEE generates test cases for any errors and for coverage

Testing Process (2)

- Test verification:
  - Programs are linked with `gcc` using μlibc and stripped down model

Testing Process (2)

- Test verification:
  - Programs are linked with `gcc` using μlibc and stripped down model
  - Tests are replayed natively

Testing Process (2)

- Test verification:
  - Programs are linked with `gcc` using μlibc and stripped down model
  - Tests are replayed natively
  - Protects against modeling and system errors

Coreutils Background

- GNU implementation of standard UNIX utilities

Coreutils Background

- GNU implementation of standard UNIX utilities
- Core of most Linux installations

Coreutils Background

- GNU implementation of standard UNIX utilities
- Core of most Linux installations
- Encompasses 90 applications of varying size and focus
    - File system: `ls, dd, chmod`
    - Numeric: `factor, seq`
    - System: `printenv, hostname`
    - Text Processing: `sort, head, od`

Coreutils Case Study

- All 90 applications were tested on latest public release

Coreutils Case Study

- All 90 applications were tested on latest public release
- Focused on tool front-ends

Coreutils Case Study

- All 90 applications were tested on latest public release
- Focused on tool front-ends
- Covers 20k combined executable lines of code

Coreutils Case Study

- All 90 applications were tested on latest public release
- Focused on tool front-ends
- Covers 20k combined executable lines of code
- Mostly automatic
  - One source code modification required
  - Most programs tested using a generic configuration

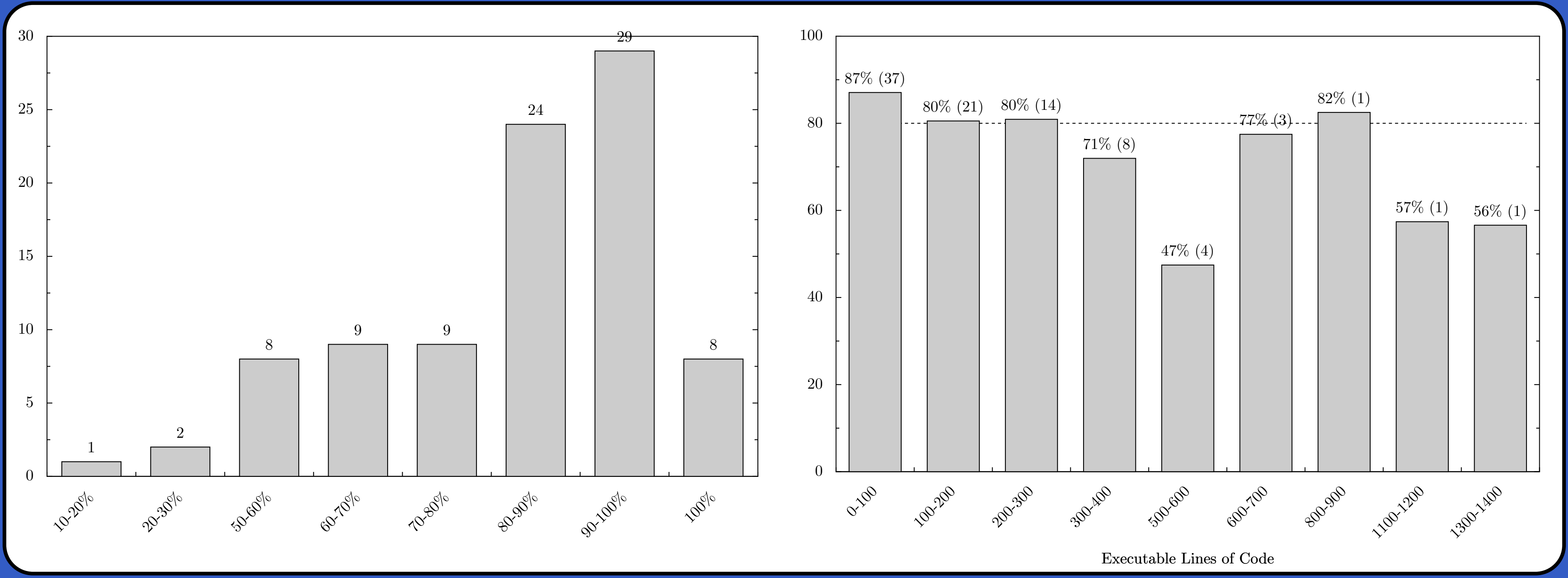
Results: Bugs

- Six bugs found
  - One fixed in developer repository
  - Three resulted in segmentation fault
  - Two allowed arbitrary heap smashing

```
ptx -F\\ abcdefghijklmnopqrstuvwxyz
paste -d\\ abcdefghijklmnopqrstuvwxyz
seq -f %0 1
mkdir -Z a b
mkfifo -Z a b
mknod -Z a b p
```

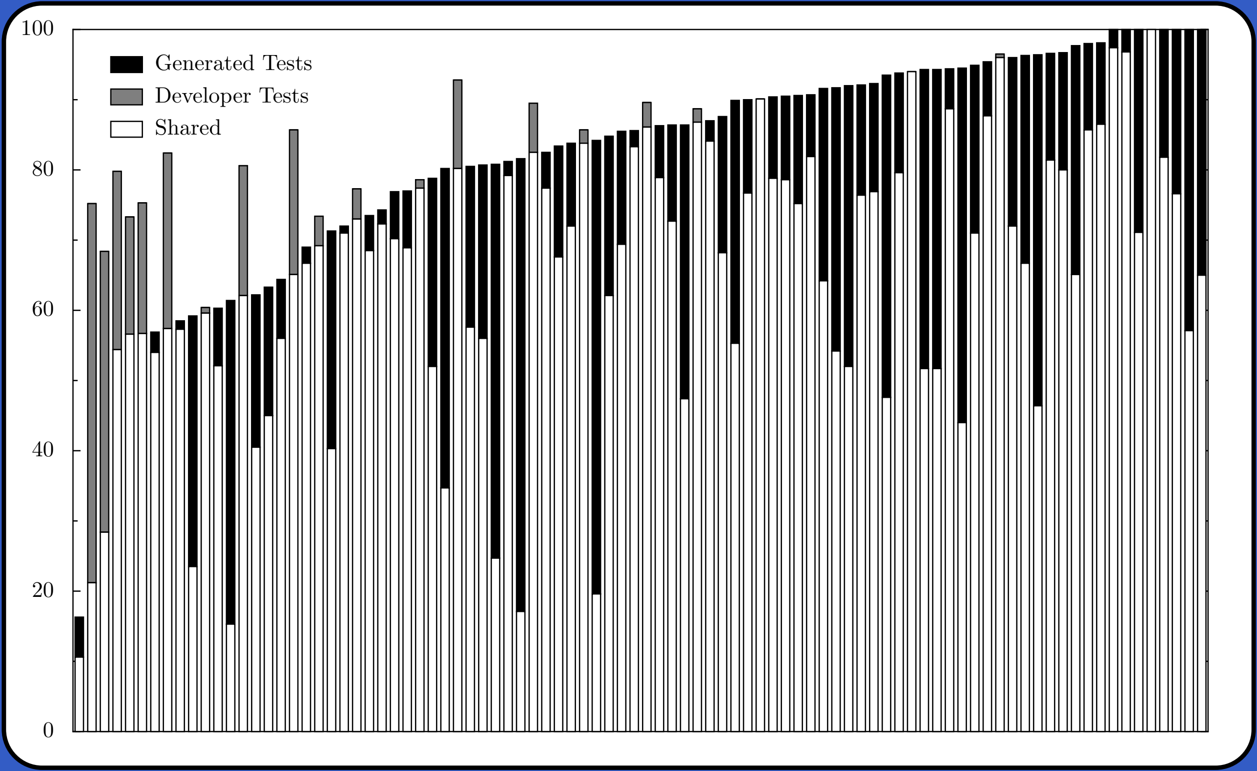
Results: Coverage

Overall coverage: 70.6%
Over 80% on $61/90$



Results: Vs. Manual

Developers tests overall coverage: 67.4%.
Generated tests outperform on majority of tools.



Conclusion

- Symbolic execution can provide an effective testing tool

Conclusion

- Symbolic execution can provide an effective testing tool
- Can scale to real applications
  - Careful attention to scalability
  - Take advantage of specialized domain to optimize constraint solving

Conclusion

- Symbolic execution can provide an effective testing tool
- Can scale to real applications
  - Careful attention to scalability
  - Take advantage of specialized domain to optimize constraint solving
- Requires only modest amount of effort to construct an adequate model

Conclusion

- Symbolic execution can provide an effective testing tool
- Can scale to real applications
  - Careful attention to scalability
  - Take advantage of specialized domain to optimize constraint solving
- Requires only modest amount of effort to construct an adequate model
- Can beat manual testing out-of-the-box

Questions?