L33-34: Symbolic Execution

Software Analysis

IIT Guwahati · Oct 22 & 27

Motivation (1/5)

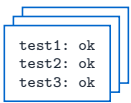
- Writing and maintaining tests is tedious and error-prone

Motivation (2/5)

- Writing and maintaining tests is tedious and error-prone
- **Idea: Automated Test Generation**

Motivation (3/5)

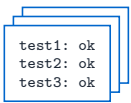
- Writing and maintaining tests is tedious and error-prone
- **Idea: Automated Test Generation**



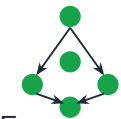
Generate a regression test suite

Motivation (4/5)

- Writing and maintaining tests is tedious and error-prone
- **Idea: Automated Test Generation**



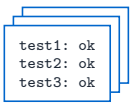
Generate a regression test suite



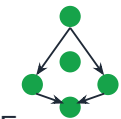
Execute all reachable statements

Motivation (5/5)

- Writing and maintaining tests is tedious and error-prone
- **Idea: Automated Test Generation**



Generate a regression test suite



Execute all reachable statements

```
assert(x != 0);
```

Catch any **assertion violations**

Random Testing (1/2)

Random testing

Generate random inputs
Execute the program on those (concrete) inputs

```
1 void test_me(int x) {
2   if (x == 94389) {
3     ERROR;
4   }
5 }
```

Random Testing (2/2)

Random testing

Generate random inputs
Execute the program on those (concrete) inputs

```
1 void test_me(int x) {
2   if (x == 94389) {
3     ERROR;
4   }
5 }
```

Problem: probability of reaching the error could be astronomically small



all 2^{32} inputs; grey: random tests

Probability of ERROR per test:
$1/2^{32} \approx 0.000000023\%$

Concrete versus Symbolic Execution (1/5)

```
1 int check(int x) {
2   int y = x + 1;
3   if (y > 10)
4     return 1;
5   return 0;
6 }
```

Cadar and Sen, CACM 2013

Concrete versus Symbolic Execution (2/5)

```
1 int check(int x) {
2   int y = x + 1;
3   if (y > 10)
4     return 1;
5   return 0;
6 }
```

Concrete: `x = 5`

line 2 `y = 6`

line 3 `6 > 10`: **false** $\Rightarrow$ returns 0

Cadar and Sen, CACM 2013

Concrete versus Symbolic Execution (3/5)

```
1 int check(int x) {
2   int y = x + 1;
3   if (y > 10)
4     return 1;
5   return 0;
6 }
```

Concrete: `x = 5`

line 2 `y = 6`

line 3 `6 > 10`: **false** $\Rightarrow$ returns 0

Symbolic: `x` $= x_0$ (any value)

line 2 $y = x_0 + 1$

Cadar and Sen, CACM 2013

Concrete versus Symbolic Execution (4/5)

```
1 int check(int x) {
2   int y = x + 1;
3   if (y > 10)
4     return 1;
5   return 0;
6 }
```

Concrete: x = 5

line 2 y = 6

line 3 6 > 10: **false** $\Rightarrow$ returns 0

Symbolic: x $= x_0$ (any value)

line 2 $y = x_0 + 1$

line 3 both sides are possible:

path A: $x_0 + 1 > 10$ $\Rightarrow$ returns 1

path B: $x_0 + 1 \leq 10$ $\Rightarrow$ returns 0

Cadar and Sen, CACM 2013

Concrete versus Symbolic Execution (5/5)

```
1 int check(int x) {
2   int y = x + 1;
3   if (y > 10)
4     return 1;
5   return 0;
6 }
```

Concrete: x = 5

line 2 y = 6

line 3 6 > 10: **false** $\Rightarrow$ returns 0

Symbolic: x = x_0 (any value)

line 2 $y = x_0 + 1$

line 3 both sides are possible:

path A: $x_0 + 1 > 10$ $\Rightarrow$ returns 1

path B: $x_0 + 1 \leq 10$ $\Rightarrow$ returns 0

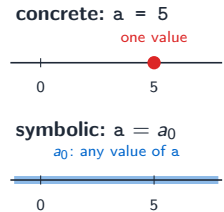
Solve each condition: A: $x_0 = 10$, B: $x_0 = 0$

One test input per path

Cadar and Sen, CACM 2013

Symbolic Execution (1/3)

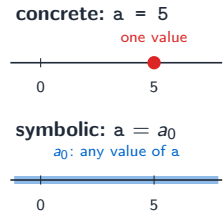
- Reason about the behavior of a program by "executing" it with **symbolic values**
  A symbolic value, e.g. a_0, stands for *any* input value



King, CACM 1976; Clarke, IEEE TSE 1976; Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution

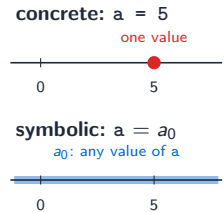
Symbolic Execution (2/3)

- Reason about the behavior of a program by "executing" it with **symbolic values**
  A symbolic value, e.g. a_0, stands for *any* input value
- Originally proposed by James King (1976, CACM) and Lori Clarke (1976, IEEE TSE)



Symbolic Execution (3/3)

- Reason about the behavior of a program by "executing" it with **symbolic values**
  A symbolic value, e.g. a_0, stands for *any* input value
- Originally proposed by James King (1976, CACM) and Lori Clarke (1976, IEEE TSE)
- Became practical around 2005 because of advances in constraint solving (**SMT solvers**)

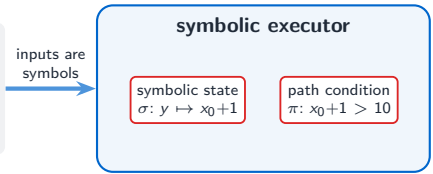


King, CACM 1976; Clarke, IEEE TSE 1976; Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution

How Symbolic Execution Works (1/3)

```
int check(int x) {
  int y = x + 1;
  if (y > 10)
    return 1;
  return 0;
}
```

check.c

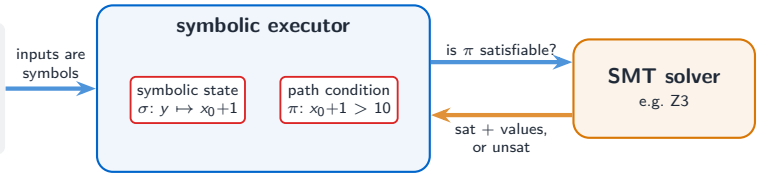


Cadar and Sen, CACM 2013

How Symbolic Execution Works (2/3)

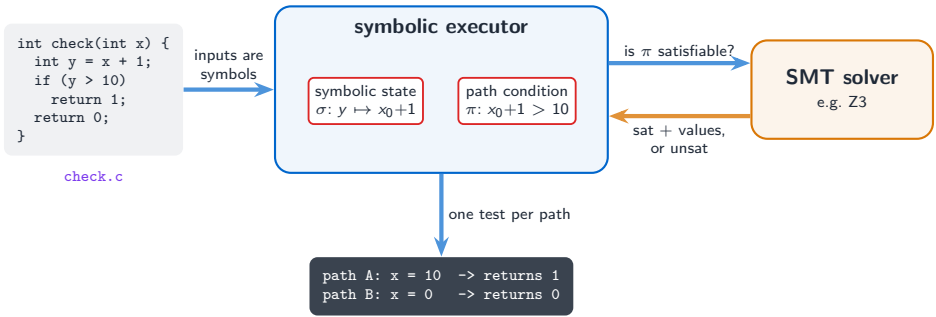
```
int check(int x) {
  int y = x + 1;
  if (y > 10)
    return 1;
  return 0;
}
```

check.c



Cadar and Sen, CACM 2013

How Symbolic Execution Works (3/3)



Cadar and Sen, CACM 2013

Satisfiability of Formulas (1/5)

Determine whether a path is **feasible**: check if its path condition is **satisfiable**

Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution; de Moura and Bjørner, TACAS 2008

Satisfiability of Formulas (2/5)

Determine whether a path is **feasible**: check if its path condition is **satisfiable**

- Done by **SMT solvers**, e.g. Z3, Yices, STP[2]
  SMT = satisfiability modulo theories[1]

[1] **SMT solver**: a SAT solver (boolean variables only) for the and/or/not structure, plus a solver for each **theory**, which fixes the meaning of $+$, $<$, ... on integers, bit-vectors, arrays or floating point.

[2] **Z3**: Microsoft Research (de Moura and Bjørner, 2008). **Yices**: SRI International (Dutertre and de Moura). **STP**: Stanford (Ganesh and Dill, 2007), the solver used by KLEE.

Satisfiability of Formulas (3/5)

Determine whether a path is **feasible**: check if its path condition is **satisfiable**

- Done by **SMT solvers**, e.g. Z3, Yices, STP[2]
  SMT = satisfiability modulo theories[1]
- For a satisfiable formula, solvers also give a **concrete solution**

[1] **SMT solver**: a SAT solver (boolean variables only) for the and/or/not structure, plus a solver for each **theory**, which fixes the meaning of $+$, $<$, ... on integers, bit-vectors, arrays or floating point.

[2] **Z3**: Microsoft Research (de Moura and Bjørner, 2008). **Yices**: SRI International (Dutertre and de Moura). **STP**: Stanford (Ganesh and Dill, 2007), the solver used by KLEE.

Satisfiability of Formulas (4/5)

Determine whether a path is **feasible**: check if its path condition is **satisfiable**

- Done by **SMT solvers**, e.g. Z3, Yices, STP[2]
  SMT = satisfiability modulo theories[1]
- For a satisfiable formula, solvers also give a **concrete solution**
- Examples:
  $a_0 + b_0 > 1$: satisfiable, one solution: $a_0 = 1, b_0 = 1$

[1] **SMT solver**: a SAT solver (boolean variables only) for the and/or/not structure, plus a solver for each **theory**, which fixes the meaning of $+$, $<$, ... on integers, bit-vectors, arrays or floating point.

[2] **Z3**: Microsoft Research (de Moura and Bjørner, 2008). **Yices**: SRI International (Dutertre and de Moura). **STP**: Stanford (Ganesh and Dill, 2007), the solver used by KLEE.

Satisfiability of Formulas (5/5)

Determine whether a path is **feasible**: check if its path condition is **satisfiable**

- Done by **SMT solvers**, e.g. Z3, Yices, STP[2]
  SMT = satisfiability modulo theories[1]
- For a satisfiable formula, solvers also give a **concrete solution**
- Examples:
  $a_0 + b_0 > 1$: satisfiable, one solution: $a_0 = 1, b_0 = 1$
  $(a_0 + b_0 < 0) \wedge (a_0 - 1 > 5) \wedge (b_0 > 0)$: unsatisfiable
  ($a_0 > 6$ and $b_0 > 0$ give $a_0 + b_0 > 6$)

[1] **SMT solver**: a SAT solver (boolean variables only) for the and/or/not structure, plus a solver for each **theory**, which fixes the meaning of $+$, $<$, ... on integers, bit-vectors, arrays or floating point.

[2] **Z3**: Microsoft Research (de Moura and Bjørner, 2008). **Yices**: SRI International (Dutertre and de Moura). **STP**: Stanford (Ganesh and Dill, 2007), the solver used by KLEE.

Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution; de Moura and Bjørner, TACAS 2008

The Rules of Symbolic Execution (1/5)

Per path: σ = symbolic state, π = path condition[1]; start: σ empty, $\pi = \mathit{true}$

Statement	What symbolic execution does
`x = input()`	$\sigma(x) := x_0$, a fresh symbol

Cadar and Sen, CACM 2013; King, CACM 1976

The Rules of Symbolic Execution (2/5)

Per path: σ = symbolic state, π = path condition[1]; start: σ empty, $\pi = \mathit{true}$

Statement	What symbolic execution does
`x = input()`	$\sigma(x) := x_0$, a fresh symbol
`x = e`	$\sigma(x) := \sigma(e)$, i.e. e with every variable replaced by its expression

Cadar and Sen, CACM 2013; King, CACM 1976

The Rules of Symbolic Execution (3/5)

Per path: σ = symbolic state, π = path condition[1]; start: σ empty, $\pi = \mathit{true}$

Statement	What symbolic execution does
`x = input()`	$\sigma(x) := x_0$, a fresh symbol
`x = e`	$\sigma(x) := \sigma(e)$, i.e. e with every variable replaced by its expression
`if (c)`	fork: true side $\pi \wedge \sigma(c)$, false side $\pi \wedge \neg\sigma(c)$ drop a side whose path condition is unsatisfiable

Cadar and Sen, CACM 2013; King, CACM 1976

The Rules of Symbolic Execution (4/5)

Per path: σ = symbolic state, π = path condition[1]; start: σ empty, $\pi = \mathit{true}$

Statement	What symbolic execution does
`x = input()`	$\sigma(x) := x_0$, a fresh symbol
`x = e`	$\sigma(x) := \sigma(e)$, i.e. e with every variable replaced by its expression
`if (c)`	fork: true side $\pi \wedge \sigma(c)$, false side $\pi \wedge \neg\sigma(c)$ drop a side whose path condition is unsatisfiable
`assert(c)`	report an error if $\pi \wedge \neg\sigma(c)$ is satisfiable

Cadar and Sen, CACM 2013; King, CACM 1976

The Rules of Symbolic Execution (5/5)

Per path: σ = symbolic state, π = path condition[1]; start: σ empty, $\pi = \mathit{true}$

Statement	What symbolic execution does
`x = input()`	$\sigma(x) := x_0$, a fresh symbol
`x = e`	$\sigma(x) := \sigma(e)$, i.e. e with every variable replaced by its expression
`if (c)`	fork: true side $\pi \wedge \sigma(c)$, false side $\pi \wedge \neg\sigma(c)$ drop a side whose path condition is unsatisfiable
`assert(c)`	report an error if $\pi \wedge \neg\sigma(c)$ is satisfiable
end of path	solve π: a concrete test input for this path

[1] **Quantifier-free**: π has no "for all" ($\forall$) and no "there exists" ($\exists$); its only variables are the input symbols, and we ask whether some values for them make it true.

Cadar and Sen, CACM 2013; King, CACM 1976

Example (1/6)

```
1 void g(int a, int b) {
2   int c = a - b;
3   if (c > 0)
4     c = c * 2;
5   assert(c != 6);
6 }
```

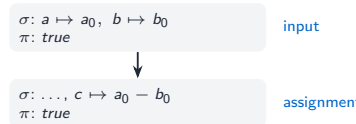
σ: $a \mapsto a_0,\ b \mapsto b_0$
π: $true$

input

Cadar and Sen, CACM 2013

Example (2/6)

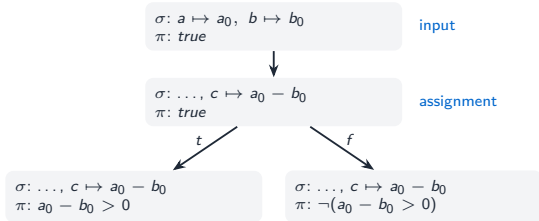
```
1 void g(int a, int b) {
2   int c = a - b;
3   if (c > 0)
4     c = c * 2;
5   assert(c != 6);
6 }
```



Cadar and Sen, CACM 2013

Example (3/6)

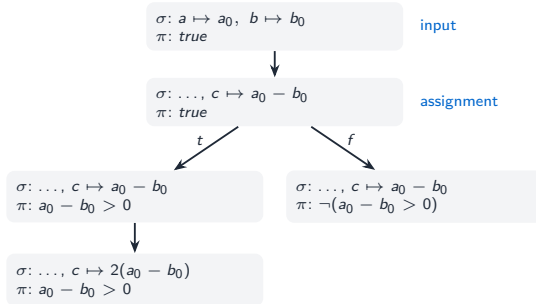
```
1 void g(int a, int b) {
2   int c = a - b;
3   if (c > 0)
4     c = c * 2;
5   assert(c != 6);
6 }
```



Cadar and Sen, CACM 2013

Example (4/6)

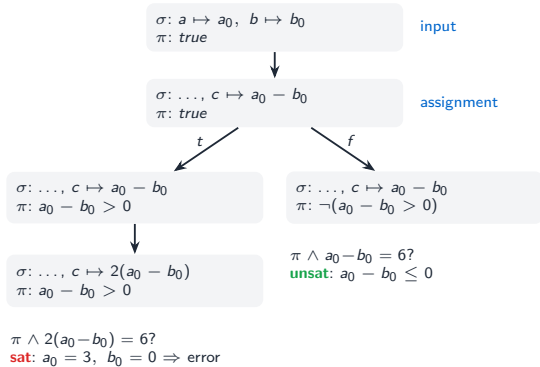
```
1 void g(int a, int b) {
2   int c = a - b;
3   if (c > 0)
4     c = c * 2;
5   assert(c != 6);
6 }
```



Cadar and Sen, CACM 2013

Example (5/6)

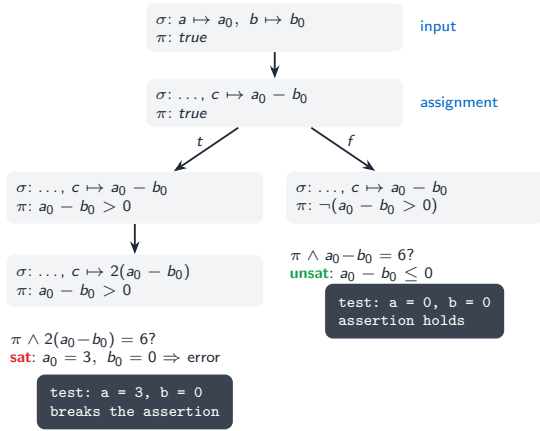
```
1 void g(int a, int b) {
2   int c = a - b;
3   if (c > 0)
4     c = c * 2;
5   assert(c != 6);
6 }
```



Cadar and Sen, CACM 2013

Example (6/6)

```
1 void g(int a, int b) {
2   int c = a - b;
3   if (c > 0)
4     c = c * 2;
5   assert(c != 6);
6 }
```



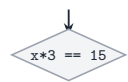
Cadar and Sen, CACM 2013

Symbolic Execution with a Theorem Prover (1/6)

At each branch, a **theorem prover** decides if a side can be taken

```
1  void test_me(int x) {
2    if (x*3 == 15) {
3      if (x % 5 == 0)
4        std::puts("OK");
5      else {
6        std::puts("something bad");
7        ERROR;
8      }
9    } else
10     std::puts("OK");
11 }
```

Query to the prover **Answer**



Symbolic Execution with a Theorem Prover (2/6)

At each branch, a **theorem prover** decides if a side can be taken

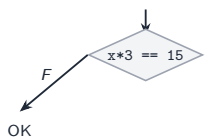
```
1  void test_me(int x) {
2    if (x*3 == 15) {
3      if (x % 5 == 0)
4        std::puts("OK");
5      else {
6        std::puts("something bad");
7        ERROR;
8      }
9    } else
10     std::puts("OK");
11 }
```

Query to the prover

$x_0 * 3 \neq 15$?

Answer

yes: false side reachable

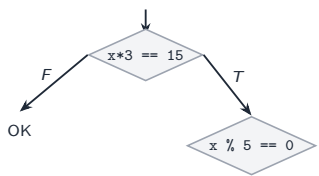


Symbolic Execution with a Theorem Prover (3/6)

At each branch, a **theorem prover** decides if a side can be taken

```
1  void test_me(int x) {
2    if (x*3 == 15) {
3      if (x % 5 == 0)
4        std::puts("OK");
5      else {
6        std::puts("something bad");
7        ERROR;
8      }
9    } else
10     std::puts("OK");
11 }
```

Query to the prover	Answer
$x_0 * 3 \neq 15$?	yes: false side reachable
$x_0 * 3 = 15$?	yes: true side reachable

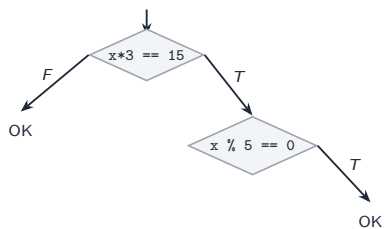


Symbolic Execution with a Theorem Prover (4/6)

At each branch, a **theorem prover** decides if a side can be taken

```
1  void test_me(int x) {
2    if (x*3 == 15) {
3      if (x % 5 == 0)
4        std::puts("OK");
5      else {
6        std::puts("something bad");
7        ERROR;
8      }
9    } else
10     std::puts("OK");
11 }
```

Query to the prover	Answer
$x_0 * 3 \neq 15$?	yes: false side reachable
$x_0 * 3 = 15$?	yes: true side reachable
$x_0 * 3 = 15 \wedge x_0 \bmod 5 = 0$?	yes: prints OK

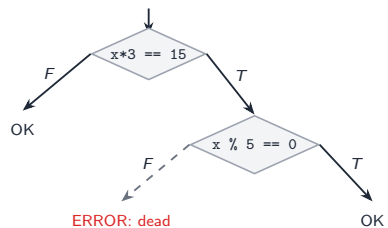


Symbolic Execution with a Theorem Prover (5/6)

At each branch, a **theorem prover** decides if a side can be taken

```
1  void test_me(int x) {
2    if (x*3 == 15) {
3      if (x % 5 == 0)
4        std::puts("OK");
5      else {
6        std::puts("something bad");
7        ERROR;
8      }
9    } else
10     std::puts("OK");
11 }
```

Query to the prover	Answer
$x_0 * 3 \neq 15$?	**yes**: false side reachable
$x_0 * 3 = 15$?	**yes**: true side reachable
$x_0 * 3 = 15 \wedge x_0 \bmod 5 = 0$?	**yes**: prints OK
$x_0 * 3 = 15 \wedge x_0 \bmod 5 \neq 0$?	**no**: dead code

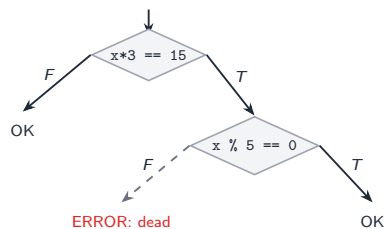


Symbolic Execution with a Theorem Prover (6/6)

At each branch, a **theorem prover** decides if a side can be taken

```
1  void test_me(int x) {
2    if (x*3 == 15) {
3      if (x % 5 == 0)
4        std::puts("OK");
5      else {
6        std::puts("something bad");
7        ERROR;
8      }
9    } else
10     std::puts("OK");
11 }
```

Query to the prover	Answer
$x_0 * 3 \neq 15$?	**yes**: false side reachable
$x_0 * 3 = 15$?	**yes**: true side reachable
$x_0 * 3 = 15 \wedge x_0 \bmod 5 = 0$?	**yes**: prints OK
$x_0 * 3 = 15 \wedge x_0 \bmod 5 \neq 0$?	**no**: dead code



Problem: does not scale to large programs

Quiz: Path Conditions (1/4)

```
1 void h(int x, int y) {
2   int z = x + y;
3   if (z > 10)
4     if (x < 0)
5       if (y < 5)
6         ERROR;
7 }
```

1. Is ERROR reachable?
2. Is the path $x_0 + y_0 > 10 \wedge x_0 < 0 \wedge \neg(y_0 < 5)$ feasible?
3. How many of the four paths are feasible?

Treat `int` as mathematical integers.

Cadar and Sen, CACM 2013

Quiz: Path Conditions (2/4)

```
1 void h(int x, int y) {
2   int z = x + y;
3   if (z > 10)
4     if (x < 0)
5       if (y < 5)
6         ERROR;
7 }
```

1. Is ERROR reachable? **NO**

2. Is the path $x_0 + y_0 > 10 \wedge x_0 < 0 \wedge \neg(y_0 < 5)$ feasible? [ ]

3. How many of the four paths are feasible? [ ]

1: $x_0 < 0,\ y_0 < 5 \Rightarrow x_0 + y_0 < 5$, but $x_0 + y_0 > 10$

Treat `int` as mathematical integers.

Quiz: Path Conditions (3/4)

```
1 void h(int x, int y) {
2   int z = x + y;
3   if (z > 10)
4     if (x < 0)
5       if (y < 5)
6         ERROR;
7 }
```

1. Is ERROR reachable? **NO**

2. Is the path $x_0 + y_0 > 10 \wedge x_0 < 0 \wedge \neg(y_0 < 5)$ feasible? **YES**

3. How many of the four paths are feasible? [ ]

1: $x_0 < 0,\ y_0 < 5 \Rightarrow x_0 + y_0 < 5$, but $x_0 + y_0 > 10$
2: e.g. $x_0 = -1,\ y_0 = 12$

Treat `int` as mathematical integers.

Cadar and Sen, CACM 2013

Quiz: Path Conditions (4/4)

```
1 void h(int x, int y) {
2   int z = x + y;
3   if (z > 10)
4     if (x < 0)
5       if (y < 5)
6         ERROR;
7 }
```

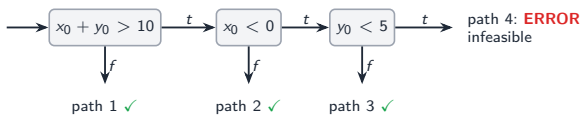
1. Is ERROR reachable? **NO**

2. Is the path $x_0 + y_0 > 10 \wedge x_0 < 0 \wedge \neg(y_0 < 5)$ feasible? **YES**

3. How many of the four paths are feasible? 3

1: $x_0 < 0,\ y_0 < 5 \Rightarrow x_0 + y_0 < 5$, but $x_0 + y_0 > 10$
2: e.g. $x_0 = -1,\ y_0 = 12$

Treat `int` as mathematical integers.



Cadar and Sen, CACM 2013

Example

```
1  void f(bool a, int b, bool c) {
2    int x = 0, y = 0, z = 0;
3    if (a) {
4      x = -2;
5    }
6    if (b > 5) {
7      if (!a && c) {
8        y = 1;
9      }
10     z = 2;
11   }
12   assert(x + y + z != 3);
13 }
```

Three inputs: a, b, c

Three conditionals

`assert(e)`: abort the program if e is false

Question: is there an input for which the assertion fails?

Example: Concrete Execution (1/6)

```
1  void f(bool a, int b, bool c) {
2    int x = 0, y = 0, z = 0;
3    if (a) {
4      x = -2;
5    }
6    if (b > 5) {
7      if (!a && c) {
8        y = 1;
9      }
10     z = 2;
11   }
12   assert(x + y + z != 3);
13 }
```

Input: `a = true, b = 1, c = true`

line 1 `a = true, b = 1, c = true`

Example: Concrete Execution (2/6)

```
1  void f(bool a, int b, bool c) {
2    int x = 0, y = 0, z = 0;
3    if (a) {
4      x = -2;
5    }
6    if (b > 5) {
7      if (!a && c) {
8        y = 1;
9      }
10     z = 2;
11   }
12   assert(x + y + z != 3);
13 }
```

Input: `a = true, b = 1, c = true`

line 1 `a = true, b = 1, c = true`

line 2 `x = y = z = 0`

Example: Concrete Execution (3/6)

```
1  void f(bool a, int b, bool c) {
2    int x = 0, y = 0, z = 0;
3    if (a) {
4      x = -2;
5    }
6    if (b > 5) {
7      if (!a && c) {
8        y = 1;
9      }
10     z = 2;
11   }
12   assert(x + y + z != 3);
13 }
```

Input: `a = true, b = 1, c = true`

line 1 `a = true, b = 1, c = true`

line 2 `x = y = z = 0`

line 3 a: **true**

Example: Concrete Execution (4/6)

```
1  void f(bool a, int b, bool c) {
2    int x = 0, y = 0, z = 0;
3    if (a) {
4      x = -2;
5    }
6    if (b > 5) {
7      if (!a && c) {
8        y = 1;
9      }
10     z = 2;
11   }
12   assert(x + y + z != 3);
13 }
```

Input: `a = true, b = 1, c = true`

line 1 `a = true, b = 1, c = true`

line 2 `x = y = z = 0`

line 3 a: **true**

line 4 `x = -2`

Example: Concrete Execution (5/6)

```
1  void f(bool a, int b, bool c) {
2    int x = 0, y = 0, z = 0;
3    if (a) {
4      x = -2;
5    }
6    if (b > 5) {
7      if (!a && c) {
8        y = 1;
9      }
10     z = 2;
11   }
12   assert(x + y + z != 3);
13 }
```

Input: `a = true, b = 1, c = true`

line 1 `a = true, b = 1, c = true`

line 2 `x = y = z = 0`

line 3 `a`: **true**

line 4 `x = -2`

line 6 `1 > 5`: **false**

Example: Concrete Execution (6/6)

```
1  void f(bool a, int b, bool c) {
2    int x = 0, y = 0, z = 0;
3    if (a) {
4      x = -2;
5    }
6    if (b > 5) {
7      if (!a && c) {
8        y = 1;
9      }
10     z = 2;
11   }
12   assert(x + y + z != 3);
13 }
```

Input: `a = true, b = 1, c = true`

line 1	`a = true, b = 1, c = true`
line 2	`x = y = z = 0`
line 3	`a`: **true**
line 4	`x = -2`
line 6	`1 > 5`: **false**
line 12	`-2 + 0 + 0 != 3` ✓

Example: Symbolic Execution (1/7)

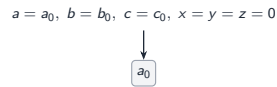
```
1  void f(bool a, int b, bool c) {
2    int x = 0, y = 0, z = 0;
3    if (a) {
4      x = -2;
5    }
6    if (b > 5) {
7      if (!a && c) {
8        y = 1;
9      }
10     z = 2;
11   }
12   assert(x + y + z != 3);
13 }
```

$a = a_0,\ b = b_0,\ c = c_0,\ x = y = z = 0$

Node: a conditional. Edge: the code run between two conditionals. t/f: true/false.

Example: Symbolic Execution (2/7)

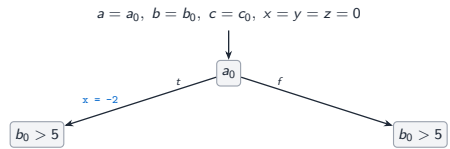
```
1  void f(bool a, int b, bool c) {
2    int x = 0, y = 0, z = 0;
3    if (a) {
4      x = -2;
5    }
6    if (b > 5) {
7      if (!a && c) {
8        y = 1;
9      }
10     z = 2;
11   }
12   assert(x + y + z != 3);
13 }
```



Node: a conditional. Edge: the code run between two conditionals. t/f: true/false.

Example: Symbolic Execution (3/7)

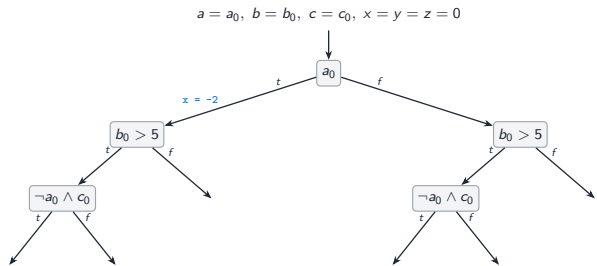
```
1  void f(bool a, int b, bool c) {
2    int x = 0, y = 0, z = 0;
3    if (a) {
4      x = -2;
5    }
6    if (b > 5) {
7      if (!a && c) {
8        y = 1;
9      }
10     z = 2;
11   }
12   assert(x + y + z != 3);
13 }
```



Node: a conditional. Edge: the code run between two conditionals. t/f: true/false.

Example: Symbolic Execution (4/7)

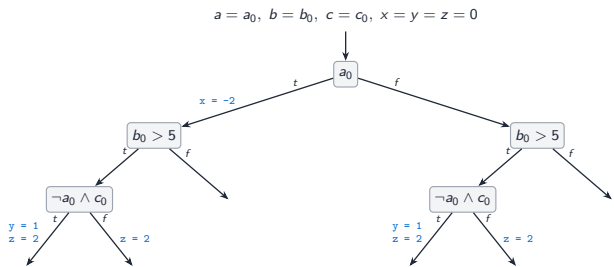
```
1  void f(bool a, int b, bool c) {
2    int x = 0, y = 0, z = 0;
3    if (a) {
4      x = -2;
5    }
6    if (b > 5) {
7      if (!a && c) {
8        y = 1;
9      }
10     z = 2;
11   }
12   assert(x + y + z != 3);
13 }
```



Node: a conditional. Edge: the code run between two conditionals. t/f: true/false.

Example: Symbolic Execution (5/7)

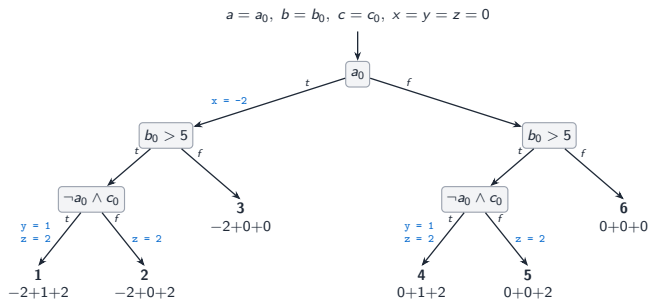
```
1  void f(bool a, int b, bool c) {
2    int x = 0, y = 0, z = 0;
3    if (a) {
4      x = -2;
5    }
6    if (b > 5) {
7      if (!a && c) {
8        y = 1;
9      }
10     z = 2;
11   }
12   assert(x + y + z != 3);
13 }
```



Node: a conditional. Edge: the code run between two conditionals. t/f: true/false.

Example: Symbolic Execution (6/7)

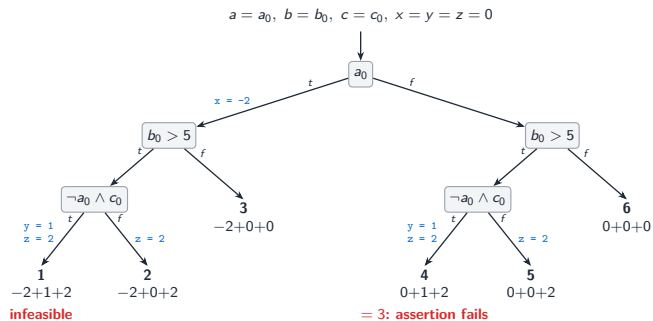
```
1  void f(bool a, int b, bool c) {
2    int x = 0, y = 0, z = 0;
3    if (a) {
4      x = -2;
5    }
6    if (b > 5) {
7      if (!a && c) {
8        y = 1;
9      }
10     z = 2;
11   }
12   assert(x + y + z != 3);
13 }
```



Node: a conditional. Edge: the code run between two conditionals. t/f: true/false.

Example: Symbolic Execution (7/7)

```
1  void f(bool a, int b, bool c) {
2    int x = 0, y = 0, z = 0;
3    if (a) {
4      x = -2;
5    }
6    if (b > 5) {
7      if (!a && c) {
8        y = 1;
9      }
10     z = 2;
11   }
12   assert(x + y + z != 3);
13 }
```



Node: a conditional. Edge: the code run between two conditionals. t/f: true/false.

Example: Path Conditions of the Six Paths (1/4)

Recall: the **path condition** of a path is the conjunction ($\wedge$) of its branch decisions.

Path	Path condition		
1	$a_0 \wedge (b_0 > 5) \wedge (\neg a_0 \wedge c_0)$		
2	$a_0 \wedge (b_0 > 5) \wedge \neg(\neg a_0 \wedge c_0)$		
3	$a_0 \wedge \neg(b_0 > 5)$		
4	$\neg a_0 \wedge (b_0 > 5) \wedge (\neg a_0 \wedge c_0)$		
5	$\neg a_0 \wedge (b_0 > 5) \wedge \neg(\neg a_0 \wedge c_0)$		
6	$\neg a_0 \wedge \neg(b_0 > 5)$		

Example: Path Conditions of the Six Paths (2/4)

Recall: the **path condition** of a path is the conjunction ($\wedge$) of its branch decisions.

Path	Path condition	$x+y+z$	
1	$a_0 \wedge (b_0 > 5) \wedge (\neg a_0 \wedge c_0)$	1	
2	$a_0 \wedge (b_0 > 5) \wedge \neg(\neg a_0 \wedge c_0)$	0	
3	$a_0 \wedge \neg(b_0 > 5)$	-2	
4	$\neg a_0 \wedge (b_0 > 5) \wedge (\neg a_0 \wedge c_0)$	3	
5	$\neg a_0 \wedge (b_0 > 5) \wedge \neg(\neg a_0 \wedge c_0)$	2	
6	$\neg a_0 \wedge \neg(b_0 > 5)$	0	

Example: Path Conditions of the Six Paths (3/4)

Recall: the **path condition** of a path is the conjunction ($\wedge$) of its branch decisions.

Path	Path condition	$x+y+z$	Feasible?
1	$a_0 \wedge (b_0 > 5) \wedge (\neg a_0 \wedge c_0)$	1	**no**: a_0 and $\neg a_0$
2	$a_0 \wedge (b_0 > 5) \wedge \neg(\neg a_0 \wedge c_0)$	0	yes
3	$a_0 \wedge \neg(b_0 > 5)$	-2	yes
4	$\neg a_0 \wedge (b_0 > 5) \wedge (\neg a_0 \wedge c_0)$	3	yes, e.g. `a=false, b=6, c=true`
5	$\neg a_0 \wedge (b_0 > 5) \wedge \neg(\neg a_0 \wedge c_0)$	2	yes
6	$\neg a_0 \wedge \neg(b_0 > 5)$	0	yes

Example: Path Conditions of the Six Paths (4/4)

Recall: the **path condition** of a path is the conjunction ($\wedge$) of its branch decisions.

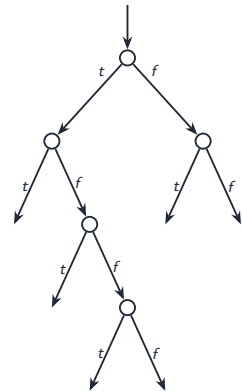
Path	Path condition	$x+y+z$	Feasible?
1	$a_0 \wedge (b_0 > 5) \wedge (\neg a_0 \wedge c_0)$	1	**no**: a_0 and $\neg a_0$
2	$a_0 \wedge (b_0 > 5) \wedge \neg(\neg a_0 \wedge c_0)$	0	yes
3	$a_0 \wedge \neg(b_0 > 5)$	-2	yes
4	$\neg a_0 \wedge (b_0 > 5) \wedge (\neg a_0 \wedge c_0)$	**3**	yes, e.g. `a=false, b=6, c=true`
5	$\neg a_0 \wedge (b_0 > 5) \wedge \neg(\neg a_0 \wedge c_0)$	2	yes
6	$\neg a_0 \wedge \neg(b_0 > 5)$	0	yes

Input `a = false, b = 6, c = true` makes the assertion fail.

Execution Tree

All possible execution paths

- Binary tree
- Nodes: **conditional statements**
- Edges: execution of a sequence of non-conditional statements
- Each **path** in the tree represents an **equivalence class of inputs**



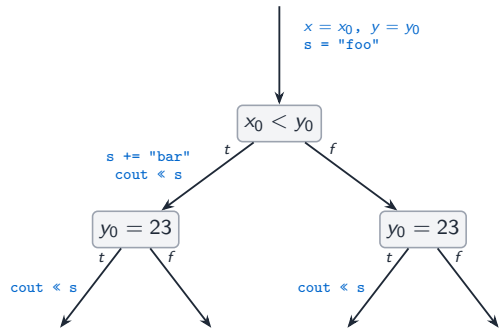
Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution

Example: Execution Tree (1/2)

```
1  void show(int x, int y) {
2    std::string s = "foo";
3    if (x < y) {
4      s += "bar";
5      std::cout « s;
6    }
7    if (y == 23) {
8      std::cout « s;
9    }
10 }
```

Example: Execution Tree (2/2)

```
1  void show(int x, int y) {
2    std::string s = "foo";
3    if (x < y) {
4      s += "bar";
5      std::cout « s;
6    }
7    if (y == 23) {
8      std::cout « s;
9    }
10 }
```



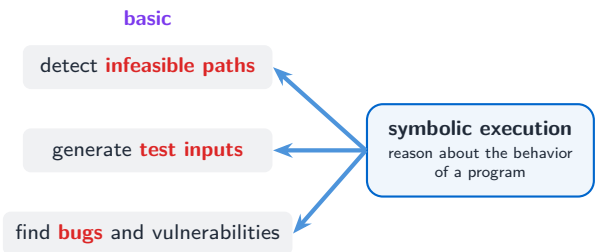
3 nodes (start and end points are not nodes)
7 edges, 4 paths

Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution

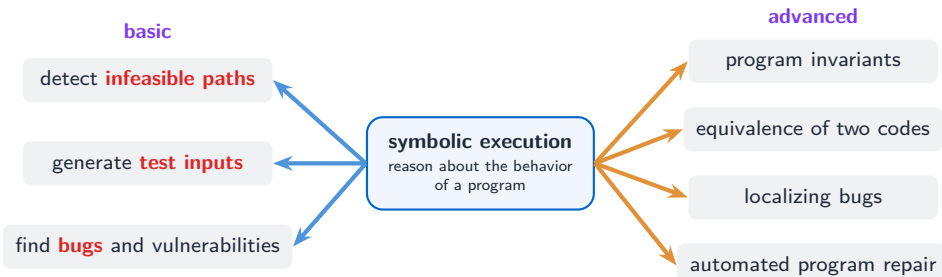
Applications of Symbolic Execution (1/3)

symbolic execution
reason about the behavior of a program

Applications of Symbolic Execution (2/3)



Applications of Symbolic Execution (3/3)



Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution

Problems of Symbolic Execution (1/2)

What goes wrong on real programs?

- Loops and recursion: infinite execution trees
- Path explosion: number of paths is exponential in the number of conditionals
- Environment modeling: dealing with native, system and library calls
- Solver limitations: dealing with complex path conditions
- Heap modeling: symbolic representation of data structures and pointers

Problems of Symbolic Execution (2/2)

What goes wrong on real programs?

- **Loops and recursion**: infinite execution trees
- **Path explosion**: number of paths is exponential in the number of conditionals
- Environment modeling: dealing with native, system and library calls
- Solver limitations: dealing with complex path conditions
- Heap modeling: symbolic representation of data structures and pointers

Path Explosion (1/4)

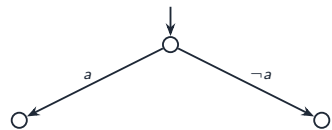
```
1 if (a) ...
2 if (b) ...
3 if (c) ...
4 ...
5 if (n) ...
```



1 path

Path Explosion (2/4)

```
1 if (a) ...
2 if (b) ...
3 if (c) ...
4 ...
5 if (n) ...
```

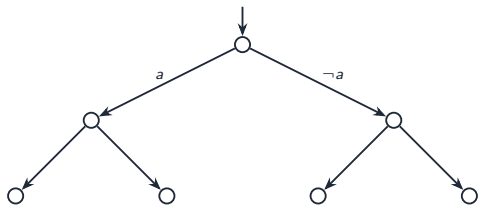


1 path

2 paths

Path Explosion (3/4)

```
1 if (a) ...
2 if (b) ...
3 if (c) ...
4 ...
5 if (n) ...
```



1 path

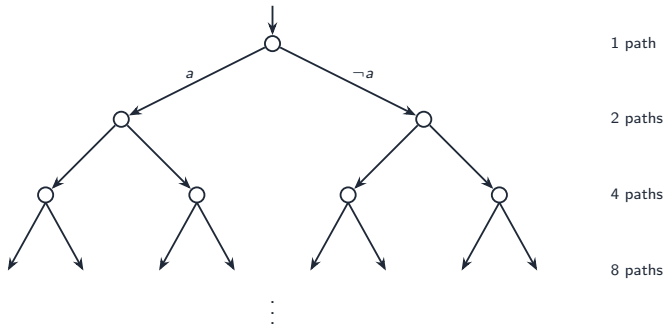
2 paths

4 paths

Path Explosion (4/4)

```
1 if (a) ...
2 if (b) ...
3 if (c) ...
4 ...
5 if (n) ...
```

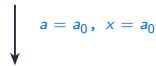
conditionals	paths
3	8
10	1,024
30	$\approx 10^9$
n	2^n



n conditionals: up to 2^n paths

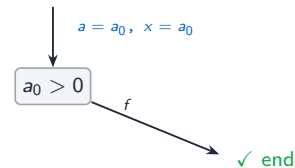
Loops and Recursion (1/4)

```
1 void count_down(int a) {
2   int x = a;
3   while (x > 0) {
4     x--;
5   }
6 }
```



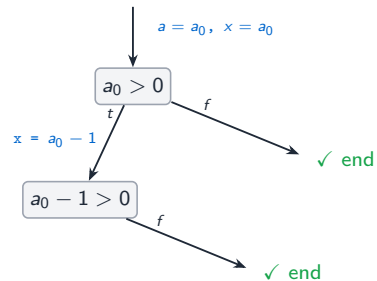
Loops and Recursion (2/4)

```
1 void count_down(int a) {
2   int x = a;
3   while (x > 0) {
4     x--;
5   }
6 }
```



Loops and Recursion (3/4)

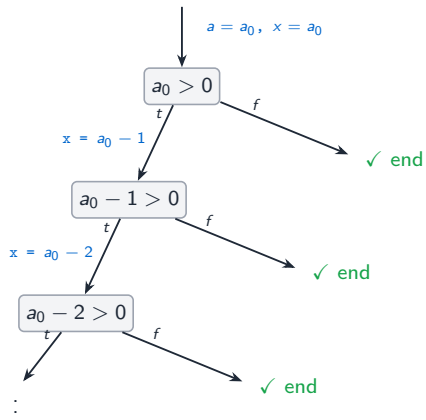
```
1 void count_down(int a) {
2   int x = a;
3   while (x > 0) {
4     x--;
5   }
6 }
```



Loops and Recursion (4/4)

```
1 void count_down(int a) {
2   int x = a;
3   while (x > 0) {
4     x--;
5   }
6 }
```

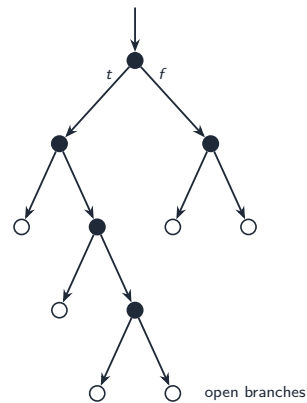
One path per number of loop iterations.
Mathematical integers: **infinite** tree.
32-bit `int`: 2^{31} paths, still far too many to explore.



Dealing with Large Execution Trees (1/5)

Heuristically select which branch to explore next

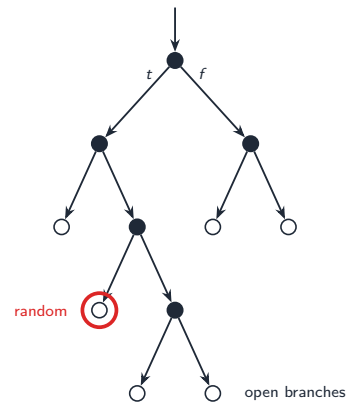
- Select at random
- Select based on coverage
- Prioritize based on distance to "interesting" program locations
- Interleave symbolic execution with random testing



Dealing with Large Execution Trees (2/5)

Heuristically select which branch to explore next

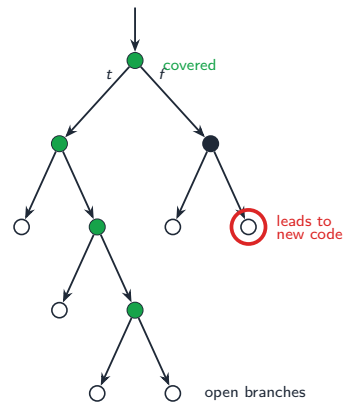
- **Select at random**
- Select based on coverage
- Prioritize based on distance to "interesting" program locations
- Interleave symbolic execution with random testing



Dealing with Large Execution Trees (3/5)

Heuristically select which branch to explore next

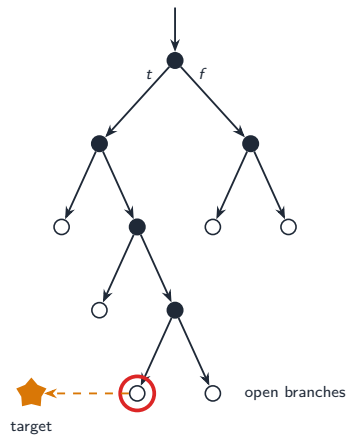
- Select at random
- **Select based on coverage**
- Prioritize based on distance to "interesting" program locations
- Interleave symbolic execution with random testing



Dealing with Large Execution Trees (4/5)

Heuristically select which branch to explore next

- Select at random
- Select based on coverage
- **Prioritize based on distance to "interesting" program locations**
- Interleave symbolic execution with random testing

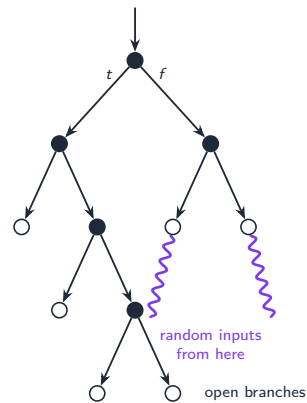


Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution

Dealing with Large Execution Trees (5/5)

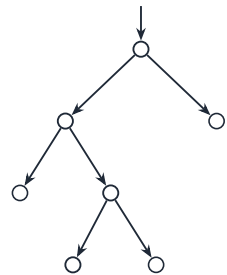
Heuristically select which branch to explore next

- Select at random
- Select based on coverage
- Prioritize based on distance to "interesting" program locations
- **Interleave symbolic execution with random testing**



Distance-Based Selection: KATCH (1/5)

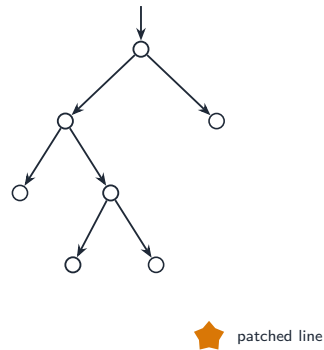
- Goal: test each new **patch**



Marinescu and Cadar, ESEC/FSE 2013

Distance-Based Selection: KATCH (2/5)

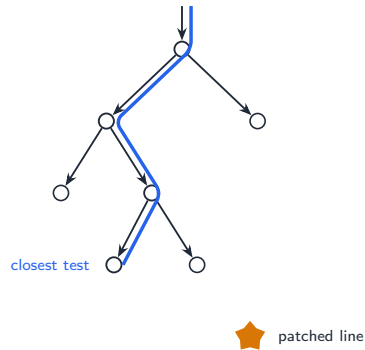
- Goal: test each new **patch**
- Target: lines the patch **changed**



Marinescu and Cadar, ESEC/FSE 2013

Distance-Based Selection: KATCH (3/5)

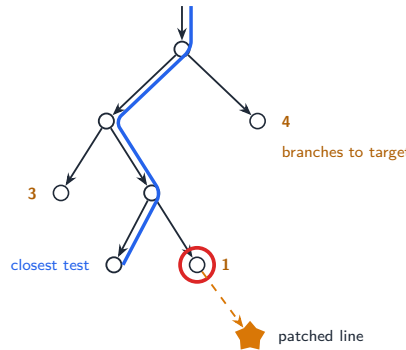
- Goal: test each new **patch**
- Target: lines the patch **changed**
- Start: the existing test **closest** to the target



Marinescu and Cadar, ESEC/FSE 2013

Distance-Based Selection: KATCH (4/5)

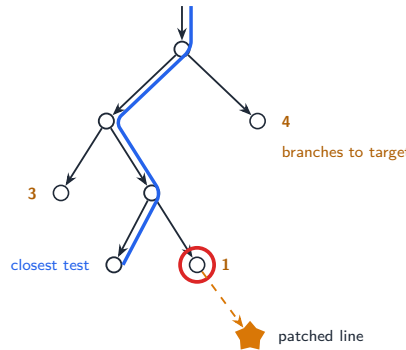
- Goal: test each new **patch**
- Target: lines the patch **changed**
- Start: the existing test **closest** to the target
- Next: the open branch with the fewest **branches** to the target



Marinescu and Cadar, ESEC/FSE 2013

Distance-Based Selection: KATCH (5/5)

- Goal: test each new **patch**
- Target: lines the patch **changed**
- Start: the existing test **closest** to the target
- Next: the open branch with the fewest **branches** to the target



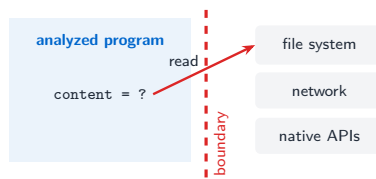
Built on KLEE. On 19 GNU programs (diffutils, binutils, findutils), ~6 years of patches: patch coverage **31%** → **52%**, and **15 distinct crash bugs**, 6 of them in the new code itself.

Marinescu and Cadar, ESEC/FSE 2013

Modeling the Environment

- Program behavior may depend on **parts of the system not analyzed** by symbolic execution
- E.g. native APIs, interaction with the network, file system accesses

```
std::ifstream in("/tmp/foo.txt");
std::string content;
std::getline(in, content);
if (content == "bar") {
  // ...
}
```



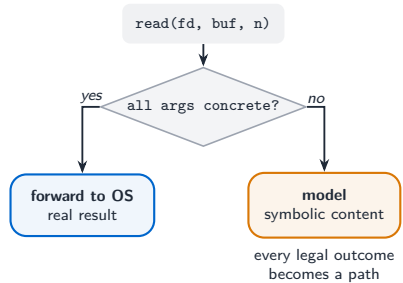
Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution

Modeling the Environment in KLEE

Solution implemented by **KLEE**[1]

- If all arguments are concrete, forward to the OS
- Otherwise (symbolic arguments, or a file declared symbolic), use **models that handle symbolic files**

  goal: explore all legal interactions with the environment




```
// model, used instead of the real read()
ssize_t read(int fd, void* buf, size_t n) {
  // does not read the actual file system, but
  // models its effects for a symbolic file:
  // the n bytes in buf become fresh symbols
}
```

[1] **KLEE**: a symbolic execution engine for C programs compiled to LLVM IR (Cadar, Dunbar and Engler, OSDI 2008).

Solver Limitations: An Example (1/3)

```
1 void test_me(int x) {
2   // pow_mod(2, x, c) = 2^x mod c
3   // c = product of two large primes
4   if (pow_mod(2, x, c) == 17) {
5     std::puts("something bad");
6     ERROR;
7   } else
8     std::puts("OK");
9 }
```

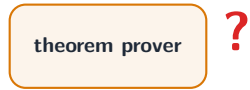
Is there x_0 with $2^{x_0} \mod c = 17$?

theorem prover

Solver Limitations: An Example (2/3)

```
1 void test_me(int x) {
2   // pow_mod(2, x, c) = 2^x mod c
3   // c = product of two large primes
4   if (pow_mod(2, x, c) == 17) {
5     std::puts("something bad");
6     ERROR;
7   } else
8     std::puts("OK");
9 }
```

Is there x_0 with $2^{x_0} \mod c = 17$?

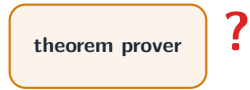


discrete logarithm: believed intractable

Solver Limitations: An Example (3/3)

```
1 void test_me(int x) {
2   // pow_mod(2, x, c) = 2^x mod c
3   // c = product of two large primes
4   if (pow_mod(2, x, c) == 17) {
5     std::puts("something bad");
6     ERROR;
7   } else
8     std::puts("OK");
9 }
```

Is there x_0 with $2^{x_0} \mod c = 17$?



discrete logarithm: believed intractable

```
gives up => "maybe"
both branches reachable
```

Symbolic execution says both branches are reachable: **False positive!**

Heap Modeling: An Example (1/5)

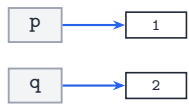
```
1 void alias(int *p, int *q) {
2   *p = 1;
3   *q = 2;
4   if (*p == 2)
5     ERROR;
6 }
```

p and q are symbolic pointers:
do they point to the same cell?

Heap Modeling: An Example (2/5)

```
1 void alias(int *p, int *q) {
2   *p = 1;
3   *q = 2;
4   if (*p == 2)
5     ERROR;
6 }
```

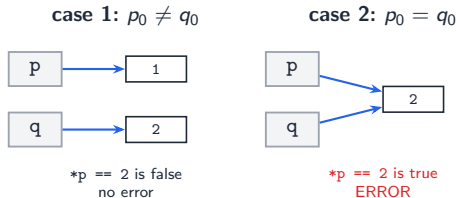
case 1: $p_0 \neq q_0$



p and q are symbolic pointers:
do they point to the same cell?

Heap Modeling: An Example (3/5)

```
1 void alias(int *p, int *q) {
2   *p = 1;
3   *q = 2;
4   if (*p == 2)
5     ERROR;
6 }
```

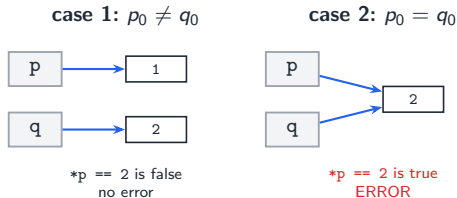


p and q are symbolic pointers:
do they point to the same cell?

Heap Modeling: An Example (4/5)

```
1 void alias(int *p, int *q) {
2   *p = 1;
3   *q = 2;
4   if (*p == 2)
5     ERROR;
6 }
```

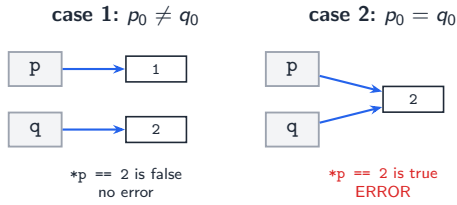
p and q are symbolic pointers:
do they point to the same cell?



Every symbolic pointer may alias others:
the cases multiply.

Heap Modeling: An Example (5/5)

```
1 void alias(int *p, int *q) {
2   *p = 1;
3   *q = 2;
4   if (*p == 2)
5     ERROR;
6 }
```



p and q are symbolic pointers:
do they point to the same cell?

Every symbolic pointer may alias others:
the cases multiply.

One approach to solver and heap limits: mix symbolic with concrete execution

Quiz: Which Problem? (1/5)

Which problem of symbolic execution does each snippet hit?

1. `for (char ch : s) if (ch == '\n') n++;`

2. `if (sha256(pwd) == STORED_HASH) login();`

3. `int fd = open(argv[1], O_RDONLY); read(fd, buf, 64);`

4. `*p = 0; *q = 1; if (*p == 1) ERROR;`

Quiz: Which Problem? (2/5)

Which problem of symbolic execution does each snippet hit?

1. `for (char ch : s) if (ch == '\n') n++;`

2. `if (sha256(pwd) == STORED_HASH) login();`

3. `int fd = open(argv[1], O_RDONLY); read(fd, buf, 64);`

4. `*p = 0; *q = 1; if (*p == 1) ERROR;`

loops / path explosion: one path per input length

Quiz: Which Problem? (3/5)

Which problem of symbolic execution does each snippet hit?

1. `for (char ch : s) if (ch == '\n') n++;`

2. `if (sha256(pwd) == STORED_HASH) login();`

3. `int fd = open(argv[1], O_RDONLY); read(fd, buf, 64);`

4. `*p = 0; *q = 1; if (*p == 1) ERROR;`

loops / path explosion: one path per input length

solver limitation: the hash cannot be inverted

Quiz: Which Problem? (4/5)

Which problem of symbolic execution does each snippet hit?

1. `for (char ch : s) if (ch == '\n') n++;`

2. `if (sha256(pwd) == STORED_HASH) login();`

3. `int fd = open(argv[1], O_RDONLY); read(fd, buf, 64);`

4. `*p = 0; *q = 1; if (*p == 1) ERROR;`

loops / path explosion: one path per input length

solver limitation: the hash cannot be inverted

environment: file contents come from the OS

Quiz: Which Problem? (5/5)

Which problem of symbolic execution does each snippet hit?

1. `for (char ch : s) if (ch == '\n') n++;`

2. `if (sha256(pwd) == STORED_HASH) login();`

3. `int fd = open(argv[1], O_RDONLY); read(fd, buf, 64);`

4. `*p = 0; *q = 1; if (*p == 1) ERROR;`

loops / path explosion: one path per input length

solver limitation: the hash cannot be inverted

environment: file contents come from the OS

heap modeling: do p and q alias?