Program Testing and Analysis:

Symbolic and Concolic Testing (Part 2)

Prof. Dr. Michael Pradel

Software Lab, TU Darmstadt

Warm-up Quiz

What does the following code print?

```
var sum = 0;
var array = [11, 22, 33];
for (x in array) {
  sum += x;
}
console.log(sum);
```

112233 **0012** **66** **Something else**

Warm-up Quiz

What does the following code print?

```
var sum = 0;
var array = [11, 22, 33];
for (x in array) {
  sum += x;
}
console.log(sum);
```

112233 **0012** **66** Some JS engines **Something else**

Warm-up Quiz

What does the following code print?

```
var sum = 0;
var array = [11, 22, 33];
for (x in array) {
  sum += x;
}
console.log(sum);
```

Arrays are objects

For-in iterates over object property names (not property values)

112233 | **0012** | **66** | Some JS engines **Something else**

Warm-up Quiz

What does the following code print?

```
var sum = 0;
var array = [11, 22, 33];
for (x in array) {
  sum += x;
}
console.log(sum);
```

For arrays, use traditional for loop:

```
for (var i=0;
     i < array.length;
     i++) ...
```

112233 | **0012** | 66 | Some JS engines: **Something else**

Outline

1. Classical Symbolic Execution
2. Challenges of Symbolic Execution
3. Concolic Testing
4. Large-Scale Application in Practice

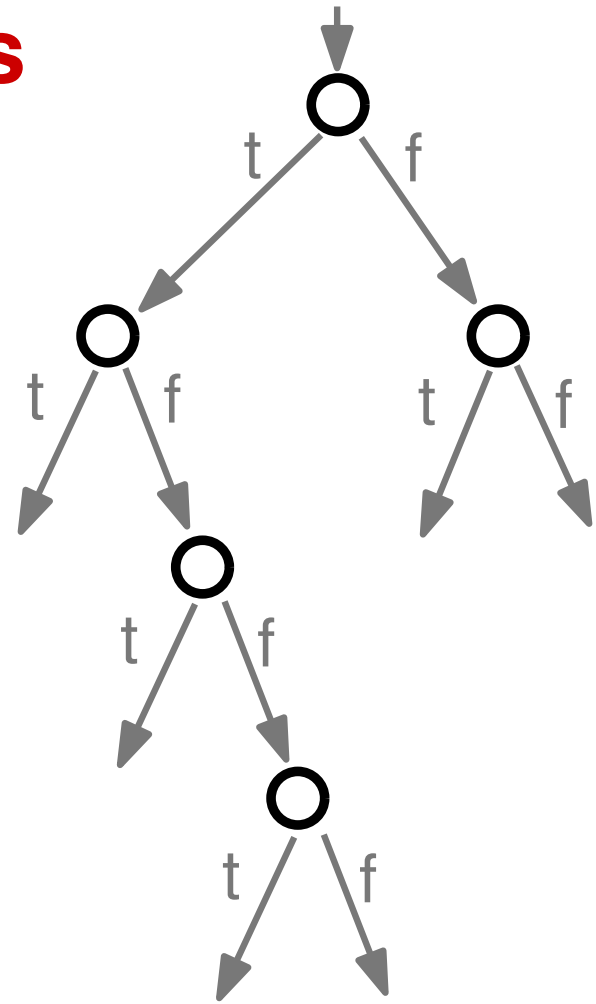
Mostly based on these papers:

- *DART: directed automated random testing*, Godefroid et al., PLDI'05
- *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*, Cadar et al., OSDI'08
- *Automated Whitebox Fuzz Testing*, Godefroid et al., NDSS'08

Execution Tree

All possible execution paths

- Binary tree
- Nodes: Conditional statements
- Edges: Execution of sequence on non-conditional statements
- Each path in the tree represents an equivalence class of inputs



Path Conditions

Quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far

```
function f(x, y) {
  var z = x + y;
  if (z > 0) {
    ...
  }
}
```

Path Conditions

Quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far

```
function f(x, y) {
  var z = x + y;
  if (z > 0) {
    ...
  }
}
```

Path condition:

$x_0 + y_0 > 0$

Applications of Symbolic Execution

- General goal: Reason about behavior of program
- Basic applications
  - Detect infeasible paths
  - Generate test inputs
  - Find bugs and vulnerabilies
- Advanced applications
  - Generating program invariants
  - Prove that two pieces of code are equivalent
  - Debugging
  - Automated program repair

Outline

1. Classical Symbolic Execution
2. Challenges of Symbolic Execution 
3. Concolic Testing
4. Large-Scale Application in Practice

Mostly based on these papers:

- *DART: directed automated random testing*, Godefroid et al., PLDI'05
- *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*, Cadar et al., OSDI'08
- *Automated Whitebox Fuzz Testing*, Godefroid et al., NDSS'08

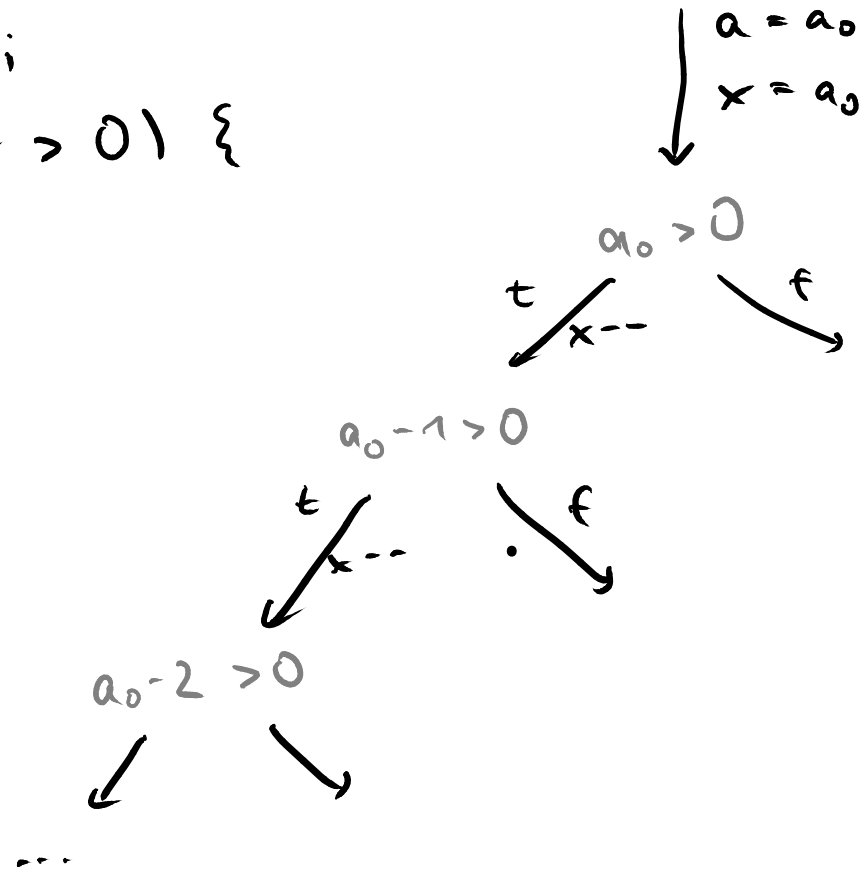
Problems of Symbolic Execution

- Loops and recursion: Infinite execution trees
- Path explosion: Number of paths is exponential in the number of conditionals
- Environment modeling: Dealing with native/system/library calls
- Solver limitations: Dealing with complex path conditions
- Heap modeling: Symbolic representation of data structures and pointers

Problems of Symbolic Execution

- Loops and recursion: Infinite execution trees
- Path explosion: Number of paths is exponential in the number of conditionals
- Environment modeling: Dealing with native/system/library calls
- Solver limitations: Dealing with complex path conditions
- Heap modeling: Symbolic representation of data structures and pointers

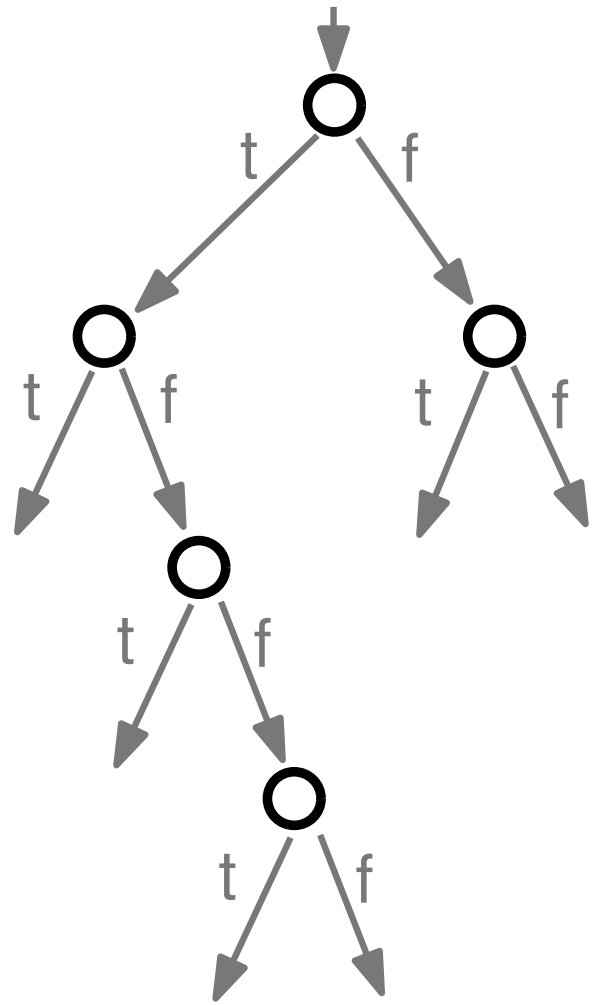
```
function f(a) {
    var x = a;
    while (x > 0) {
        x--;
    }
}
```



Dealing with Large Execution Trees

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



Problems of Symbolic Execution

- Loops and recursion: Infinite execution trees
- Path explosion: Number of paths is exponential in the number of conditionals
- Environment modeling: Dealing with native/system/library calls
- Solver limitations: Dealing with complex path conditions
- Heap modeling: Symbolic representation of data structures and pointers

Problems of Symbolic Execution

- Loops and recursion: Infinite execution trees
- Path explosion: Number of paths is exponential in the number of conditionals
- Environment modeling: Dealing with native/system/library calls
- Solver limitations: Dealing with complex path conditions
- Heap modeling: Symbolic representation of data structures and pointers

Modeling the Environment

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

```
var fs = require("fs");
var content = fs.readFileSync("/tmp/foo.txt");
if (content === "bar") {
  ...
}
```

Modeling the Environment (2)

Solution implemented by KLEE

- If all arguments are concrete, forward to OS
- Otherwise, provide models that can handle symbolic files
  - Goal: Explore all possible legal interactions with the environment

```
var fs = {
  readFileSync: function(file) {
    // doesn't read actual file system, but
    // models its effects for symbolic file names
  }
}
```

Problems of Symbolic Execution

- Loops and recursion: Infinite execution trees
- Path explosion: Number of paths is exponential in the number of conditionals
- Environment modeling: Dealing with native/system/library calls
- Solver limitations: Dealing with complex path conditions
- Heap modeling: Symbolic representation of data structures and pointers

Problems of Symbolic Execution

- Loops and recursion: Infinite execution trees
- Path explosion: Number of paths is exponential in the number of conditionals
- Environment modeling: Dealing with native/system/library calls
- Solver limitations: Dealing with complex path conditions
- Heap modeling: Symbolic representation of data structures and pointers

Approach to address these problems:

Mix symbolic with concrete execution

Outline

1. Classical Symbolic Execution
2. Challenges of Symbolic Execution
3. Concolic Testing 
4. Large-Scale Application in Practice

Mostly based on these papers:

- *DART: directed automated random testing*, Godefroid et al., PLDI'05
- *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*, Cadar et al., OSDI'08
- *Automated Whitebox Fuzz Testing*, Godefroid et al., NDSS'08

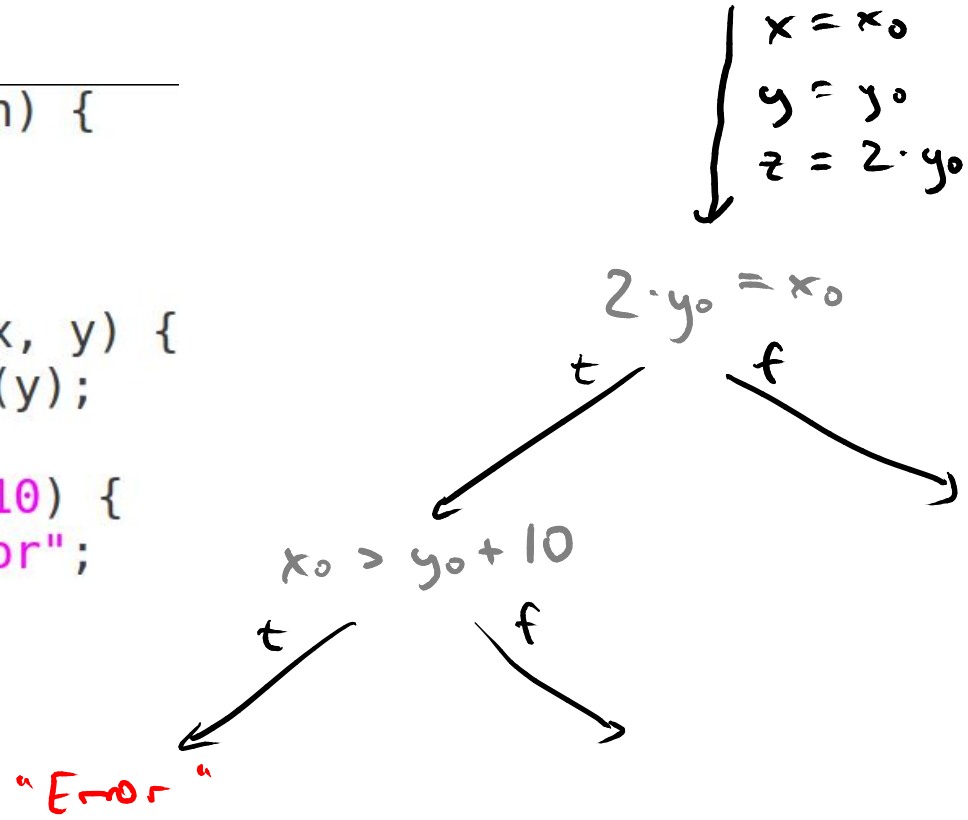
Concolic Testing

Mix concrete and symbolic execution = ”concolic”

- Perform concrete and symbolic execution side-by-side
- Gather path constraints while program executes
- After one execution, negate one decision, and re-execute with new input that triggers another path

```
function double(n) {
  return 2 * n;
}

function testMe(x, y) {
  var z = double(y);
  if (z === x) {
    if (x > y + 10) {
      throw "Error";
    }
  }
}
```



Execution 1:

Concrete execution	Symbolic execution	Path conditions
$x = 22,\ y = 7$	$x = x_0,\ y = y_0$	
$x = 22,\ y = 7,$ $z = 14$	$x = x_0,\ y = y_0,$ $z = 2 \cdot y_0$	
$x = 22,\ y = 7,$ $z = 14$	$x = x_0,\ y = y_0,$ $z = 2 \cdot y_0$	$2 \cdot y_0 \neq x_0$

```
function double(n) {
  return 2 * n;
}

function testMe(x, y) {
  var z = double(y);
  if (z === x) {
    if (x > y + 10) {
      throw "Error";
    }
  }
}
```

Solve: $2 \cdot y_0 = x_0$

Solution: $x_0 = 2,\ y_0 = 1$

Execution 2

```
function double(n) {
  return 2 * n;
}

function testMe(x, y) {
  var z = double(y);
  if (z === x) {
    if (x > y + 10) {
      throw "Error";
    }
  }
}
```

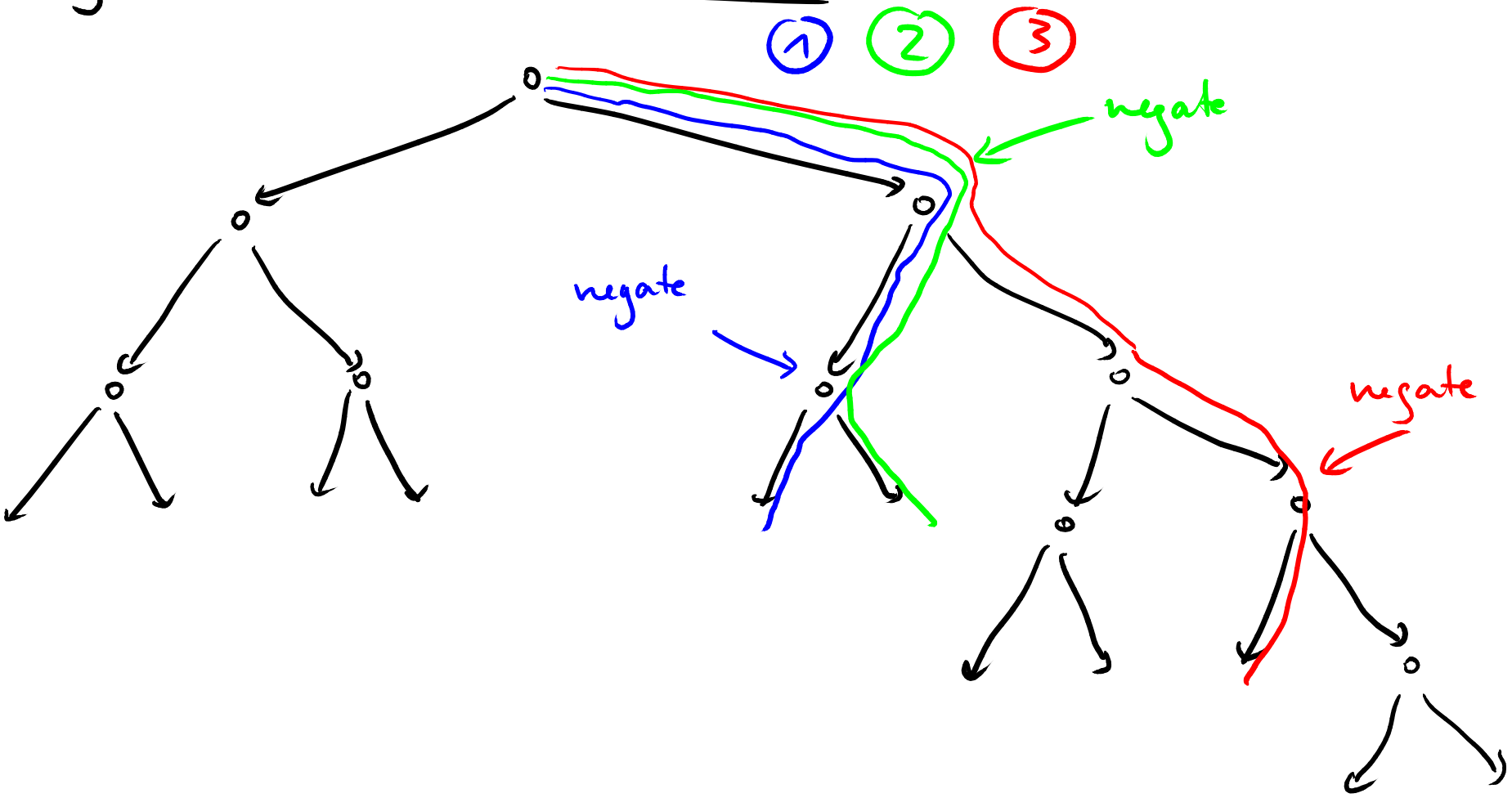
Concrete execution	Symbolic execution	Path constraints
$x = 2, y = 1$	$x = x_0$ $y = y_0$	
$x = 2, y = 1$ $z = 2$	$x = x_0, y = y_0$ $z = 2 \cdot y_0$	
— " —		$2 \cdot y_0 = x_0$
— " —		$2 \cdot y_0 = x_0 \wedge$ $x_0 \leq y_0 + 10$

Solve: $2 \cdot y_0 = x_0 \wedge x_0 > y_0 + 10$

Solution: $x_0 = 30, y_0 = 15$

Will hit "Error"

Exploring the execution tree



Algorithm

Repeat until all paths are covered

- Execute program with concrete input i and collect symbolic constraints at branch points: C
- Negate one constraint to force taking an alternative branch b': Constraints C'
- Call constraint solver to find solution for C': New concrete input i'
- Execute with i' to take branch b'
- Check at runtime that b' is indeed taken
  Otherwise: ”divergent execution”

Divergent execution: Example

```
function f(a) {
  if (Math.random() < 0.5) {
    if (a > 1) {
      console.log("took it");
    }
  }
}
```

First execution

$a = 0$

branch taken

branch not taken

path constraint:

$a_0 \leq 1$

Solver:

$a_0 = 2$

Second execution

$a = 2$

branch not taken

→ Divergent execution

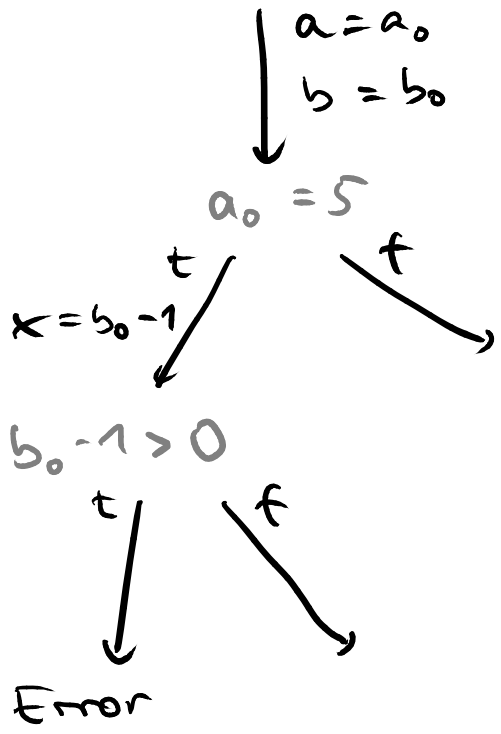
Quiz

After how many executions and how many queries to the solver does concolic testing find the error?

Initial input: `a=0, b=0`

```
function concolicQuiz(a, b) {
  if (a === 5) {
    var x = b - 1;
    if (x > 0) {
      console.log("Error");
    }
  }
}
```

Quiz:



Exec. 1

$a_0 = 0$, $b_0 = 0$

Solve : $a_0 = 5$ → $a_0 = 5$

Exec. 2

$a_0 = 5$, $b_0 = 0$

Solve : $a_0 = 5 \wedge b_0 - 1 > 0$

→ $a_0 = 5$, $b_0 = 2$

Exec 3

→ reach error

⇒ 3 executions, 2 queries

Benefits of Concolic Approach

When symbolic reasoning is impossible or impractical, fall back to concrete values

- Native/system/API functions
- Operations not handled by solver (e.g., floating point operations)

Outline

1. Classical Symbolic Execution
2. Challenges of Symbolic Execution
3. Concolic Testing
4. Large-Scale Application in Practice ←

Mostly based on these papers:

- *DART: directed automated random testing*, Godefroid et al., PLDI'05
- *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*, Cadar et al., OSDI'08
- *Automated Whitebox Fuzz Testing*, Godefroid et al., NDSS'08

Large-Scale Concolic Testing

- SAGE: Concolic testing tool developed at Microsoft Research
- Test robustness against unexpected inputs read from files, e.g.,
  - Audio files read by media player
  - Office documents read by MS Office
- Start with known input files and handle bytes read from files as symbolic input
- Use concolic execution to compute variants of these files

Large-Scale Concolic Testing (2)

- Applied to hundreds of applications
- Over 400 machine years of computation from 2007 to 2012
- Found hundreds of bugs, including many security vulnerabilties
  - One third of all the bugs discovered by file fuzzing during the development of Microsoft's Windows 7

Details: Bounimova et al., ICSE 2013

Summary: Symbolic & Concolic Testing

Solver-supported, whitebox testing

- Reason symbolically about (parts of) inputs
- Create new inputs that cover not yet explored paths
- More systematic but also more expensive than random and fuzz testing
- Open challenges
  - Effective exploration of huge search space
  - Other applications of constraint-based program analysis, e.g., debugging and automated program repair