Symbolic Execution

Theory, Limitations, Tests, Concolic Testing

Yegor Bugayenko

Lecture #8 out of 10
90 minutes

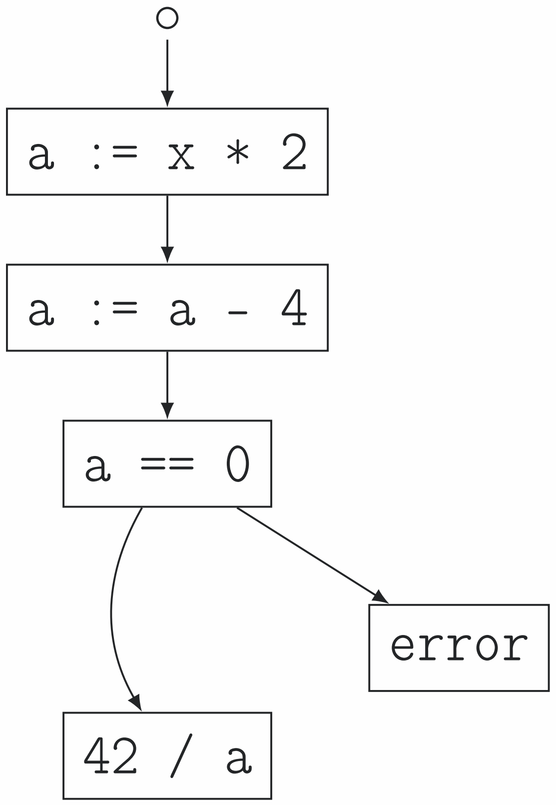
All videos are in this YouTube playlist.

Chapter #1:

In Theory

Control Flow Graph

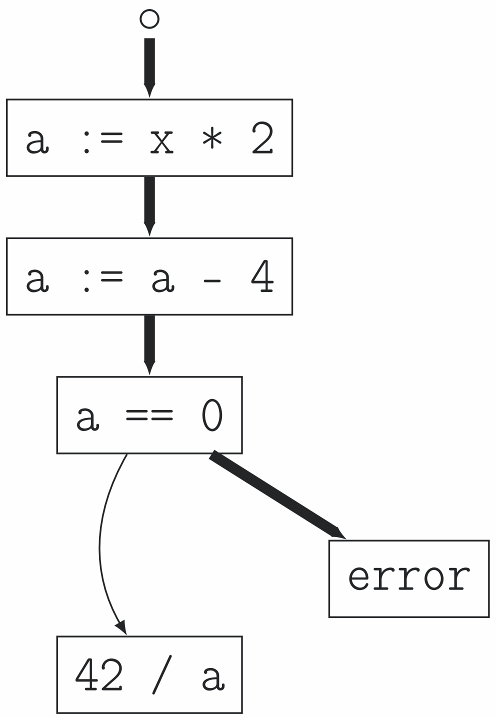
```
int f(int x) {
  int a = x * 2;
  a = a - 4;
  if (a == 0)
    error("Div by zero!");
  return 42 / a;
}
```



Path Feasibility

A path is feasible if there exists an input $\mathcal{I}$ to the program that covers the path; i.e., when program is executed with $\mathcal{I}$ as input, the path is taken.

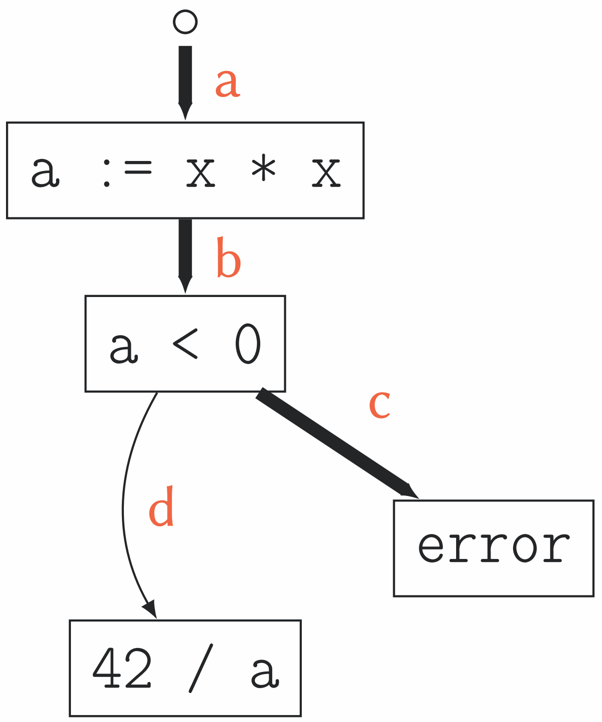
```
int f(int x) {
  int a = x * 2;
  a = a - 4;
  if (a == 0)
    error("Div by zero!");
  return 42 / a;
}
```



Infeasible Path

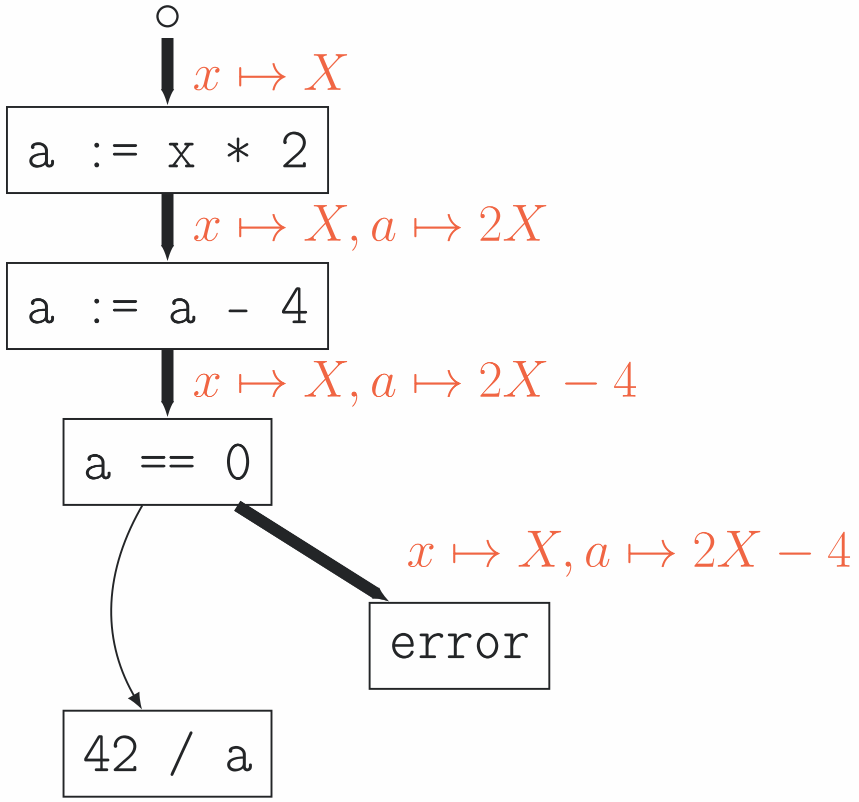
A path is infeasible if there exists no input $\mathcal{I}$ that covers the path.

```
int f(int x) {
  int a = x * x;
  if (a < 0)
    error("Too small!");
  return 42 / a;
}
```



Symbols

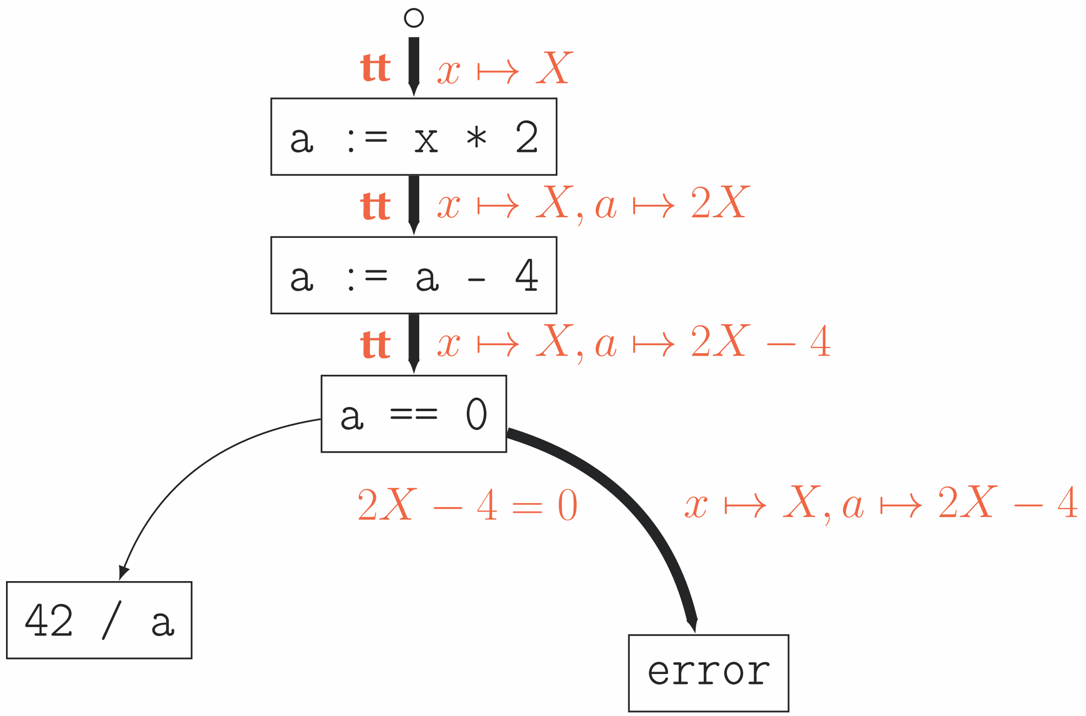
```
int f(int x) {
  int a = x * 2;
  a = a - 4;
  if (a == 0)
    error("Div by zero!");
  return 42 / a;
}
```



Path Conditions

Path condition is a condition on the input symbols such that if a path is feasible its path-condition is satisfiable.

```
int f(int x) {
  int a = x * 2;
  a = a - 4;
  if (a == 0)
    error("Div by zero!");
  return 42 / a;
}
```



Constraint Solver

A constraint solver is a tool that finds satisfying assignments for a constraint, if it is satisfiable.

A solution of the constraint is a set of assignments, one for each free variable that makes the constraint satisfiable.

Constraint:

$$x \mapsto X, \;\; a \mapsto 2X - 4$$
$$2X - 4 = 0$$

Solution:

$$X = 2$$

Chapter #2:

In Practice

SAT Solvers

SAT solver is a computer program which aims to solve the Boolean satisfiability problem: whether the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE.

Examples:

$$
\begin{aligned}
a \wedge b &\rightarrow \ldots \\
a \wedge b \wedge \neg a &\rightarrow \ldots \\
a \vee b \vee \neg a &\rightarrow \ldots \\
a \wedge (\mathbf{ff} \vee \mathbf{tt}) &\rightarrow \ldots
\end{aligned}
$$

All expressions are in Boolean logic.

SMT Solvers

SMT solver is a computer program which aims to solve the satisfiability modulo theories: determine whether a mathematical formula is satisfiable.

Examples:

$$
\begin{aligned}
a < 5 \wedge a > 3 &\to \ldots \\
a < 5 \wedge f(a) > 42 &\to \ldots \\
a < 5 \vee a > 10 \vee \neg a &\to \ldots \\
a \wedge \mathbf{ff} \wedge x = 7 &\to \ldots
\end{aligned}
$$

SMT solvers: Z3, cvc5, Yices, and many more...

Unsolvable Constraints

Symbolic execution cannot handle unsolvable or almost unsolvable constraints.

```
void enter(String p) {
  int h = sha256(p);
  if (!h.endsWith("68f728")) {
    error("Access denied!");
  }
  // You are welcome!
}
```

Path constraint:

$$p \mapsto P$$
$$H \mapsto \texttt{sha256}(P)$$
$$\texttt{endsWith}(H) = \mathbf{tt}$$

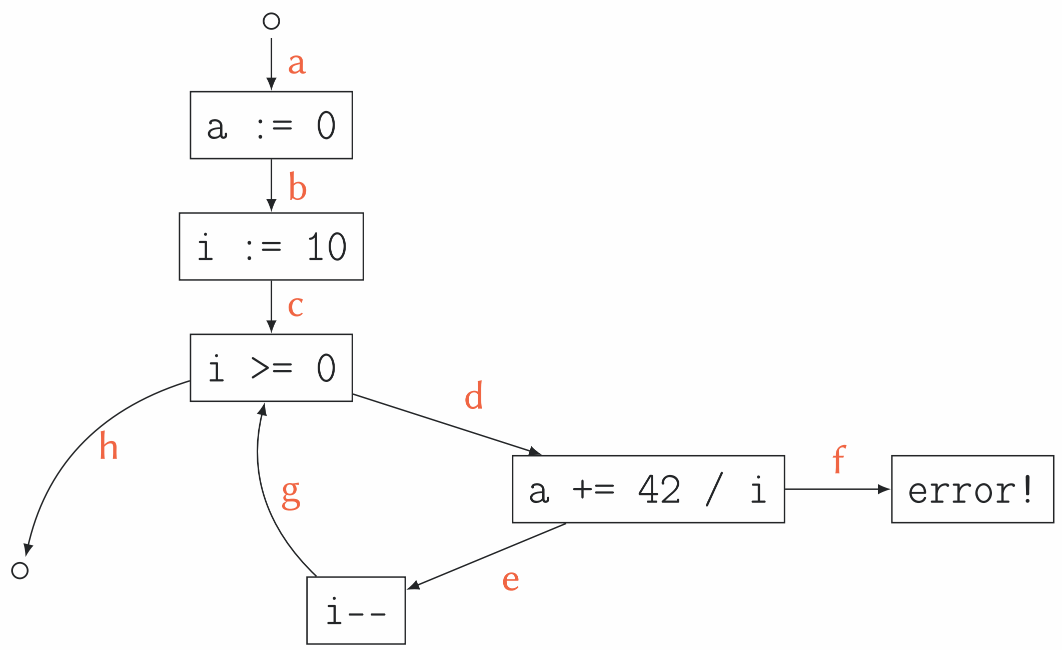
Path Explosion

Path explosion refers to the fact that the number of control-flow paths in a program grows exponentially with an increase in program size and can even be infinite in the case of programs with unbounded loop iterations.

```
int a = 0;
for (int i = 10; i >= 0; i--) {
  a += 42 / i;
}
```

Path:

$$(a, b, c, d, e, g, d, e, g, d, e, g, d, \ldots, e, g, d, f)$$



Clang Tidy vs. ChatGPT

```
/code/tmp/cpp$ cat a.cpp
int f() {
  int a = 0;
  for (int i = 3; i >= 0; i--) {
    a += 42 / i;
  }
  return a;
}/code/tmp/cpp$ clang-tidy a.cpp --
1 warning generated.
/code/tmp/cpp/a.cpp:4:13: warning: Division by zero [clang-analyzer-core.DivideZero]
    a += 42 / i;
         ~~~^~~
/code/tmp/cpp/a.cpp:3:3: note: Loop condition is true.  Entering loop body
  for (int i = 3; i >= 0; i--) {
  ^
/code/tmp/cpp/a.cpp:3:3: note: Loop condition is true.  Entering loop body
/code/tmp/cpp/a.cpp:3:3: note: Loop condition is true.  Entering loop body
/code/tmp/cpp/a.cpp:3:27: note: The value 0 is assigned to 'i'
  for (int i = 3; i >= 0; i--) {
                          ^~~
/code/tmp/cpp/a.cpp:3:3: note: Loop condition is true.  Entering loop body
  for (int i = 3; i >= 0; i--) {
  ^
/code/tmp/cpp/a.cpp:4:13: note: Division by zero
    a += 42 / i;
         ~~~^~~
```

Do you see any problems with this C++ code:

```
int a = 0;
for (int i = 10; i >= 0; i–) {
  a += 42 / i;
}
```

There is a problem with this code.

The loop condition is `i >= 0`, which means the loop will run 11 times (i.e., for `i` values of 10, 9, 8, ..., 0). When `i` reaches 0, the expression `42 / i` will cause a division by zero error, which is undefined behavior in C++.

To fix this issue, you can change the loop condition to `i > 0` instead of `i >= 0`. This will ensure that the loop runs only 10 times, for `i` values of 10, 9, 8, ..., 1.

Here's the corrected code:

```css
int a = 0;
for (int i = 10; i > 0; i--) {
  a += 42 / i;
}
```

This code will execute without any issues and will correctly calculate the sum of the expression `42 / i` for `i` values of 10, 9, 8, ..., 1.

Clang Static Analyzer

```
$ cat a.cpp
int main() {
  int a = 0;
  for (int i = 10; i >= 0; i--) {
    a += 42 / i;
  }
  return a;
}
$ clang-tidy a.cpp --
$ clang --analyze -Xclang -analyzer-constraints=z3 \
  -Xclang -analyzer-max-loop -Xclang 5 a.cpp
$ clang --analyze -Xclang -analyzer-constraints=z3 \
  -Xclang -analyzer-max-loop -Xclang 15 a.cpp
a.cpp:4:13: warning: Division by zero [core.DivideZero]
    a += 42 / i;
         ~~~^~~
1 warning generated.
```

Chapter #3:

Test Case Generation

Symbolic Input

```
#include <climits>
#include "stdlib.h"
int f(int x) {
  int a = x * 2;
  a = a - 4;
  if (a == 0)
    exit(-1);
  return 42 / a;
}
int main(int argc, char** argv) {
  int x = atoi(argv[1]);
  return f(x);
}
```

```
#include <climits>
#include "stdlib.h"
#include "klee/klee.h"
int f(int x) {
  int a = x * 2;
  a = a - 4;
  if (a == 0)
    exit(-1);
  return 42 / a;
}
int main(int argc, char** argv) {
  int x;
  klee_make_symbolic(&x, sizeof(x), "x");
  return f(x);
}
```

Compile to LLVM Bitcode

```
$ clang -I /opt/homebrew/Cellar/klee/2.3\_4/include -c -g \
  -emit-llvm -O0 -Xclang -disable-O0-optnone a.cpp

$ klee a.bc
KLEE: output directory is "/code/tmp/cpp/klee-out-2"
KLEE: Using STP solver backend
KLEE: done: total instructions = 38
KLEE: done: completed paths = 2
KLEE: done: partially completed paths = 0
KLEE: done: generated tests = 2

$ ls -al klee-out-0/*.ktest
-rw-r--r--   1 yb  staff   46 Apr  7 17:30 test000001.ktest
-rw-r--r--   1 yb  staff   46 Apr  7 17:30 test000002.ktest

$ llvm-bcanalyzer --dump a.bc
...
```

Test Cases

```
#include <climits>
#include "stdlib.h"
#include "klee/klee.h"
int f(int x) {
  int a = x * 2;
  a = a - 4;
  if (a == 0)
    exit(-1);
  return 42 / a;
}
int main(int argc, char** argv) {
  int x;
  klee_make_symbolic(&x, sizeof(x), "x");
  return f(x);
}
```

```
$ ktest-tool klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['a.bc']
num objects: 1
object 0: name: 'x'
object 0: size: 4
object 0: data: b'\x02\x00\x00\x00'
object 0: hex : 0x02000000
object 0: int : 2
object 0: uint: 2
object 0: text: ....
```

```
$ ktest-tool klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['a.bc']
num objects: 1
object 0: name: 'x'
object 0: size: 4
object 0: data: b'\x00\x00\x00\x00'
object 0: hex : 0x00000000
object 0: int : 0
object 0: uint: 0
object 0: text: ....
```

Replaying Test Cases

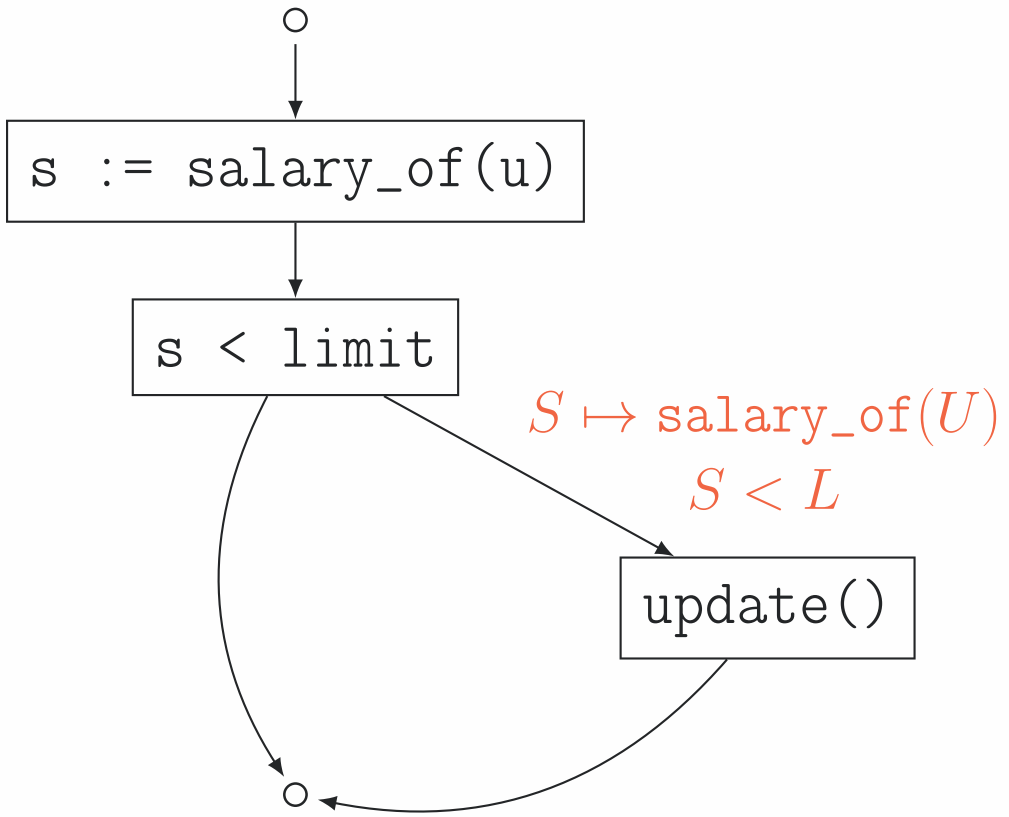
```
$ export LD_LIBRARY_PATH=/opt/homebrew/Cellar/klee/2.3_4/lib:$LD_LIBRARY_PATH

$ clang -I /opt/homebrew/Cellar/klee/2.3_4/include -L/opt/homebrew/Cellar/klee/2.3_4/lib \
  -lkleeRuntest -Xclang -disable-O0-optnone a.cpp

$ KTEST_FILE=klee-last/test000001.ktest ./a.out ; echo $?
255

$ KTEST_FILE=klee-last/test000002.ktest ./a.out ; echo $?
246
```

Chapter #4:

Concolic Testing

Motivating Example

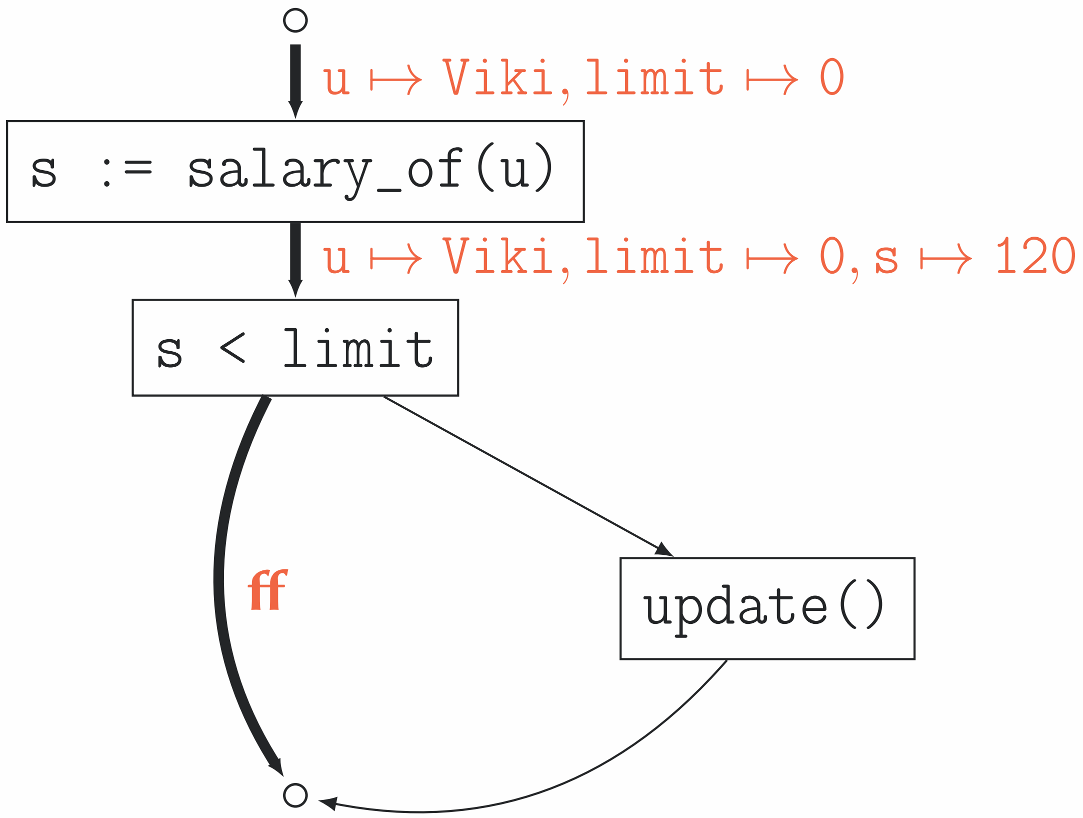
```
enum user { Viki, Peter, Jeff, Sarah };

int salary_of(user u) { ... }

void raise(user u, int limit) {
  int s = salary_of(u);
  if (s < limit)
    update(u, limit);
}

// Viki    120
// Peter   180
// Jeff     50
// Sarah    70
```



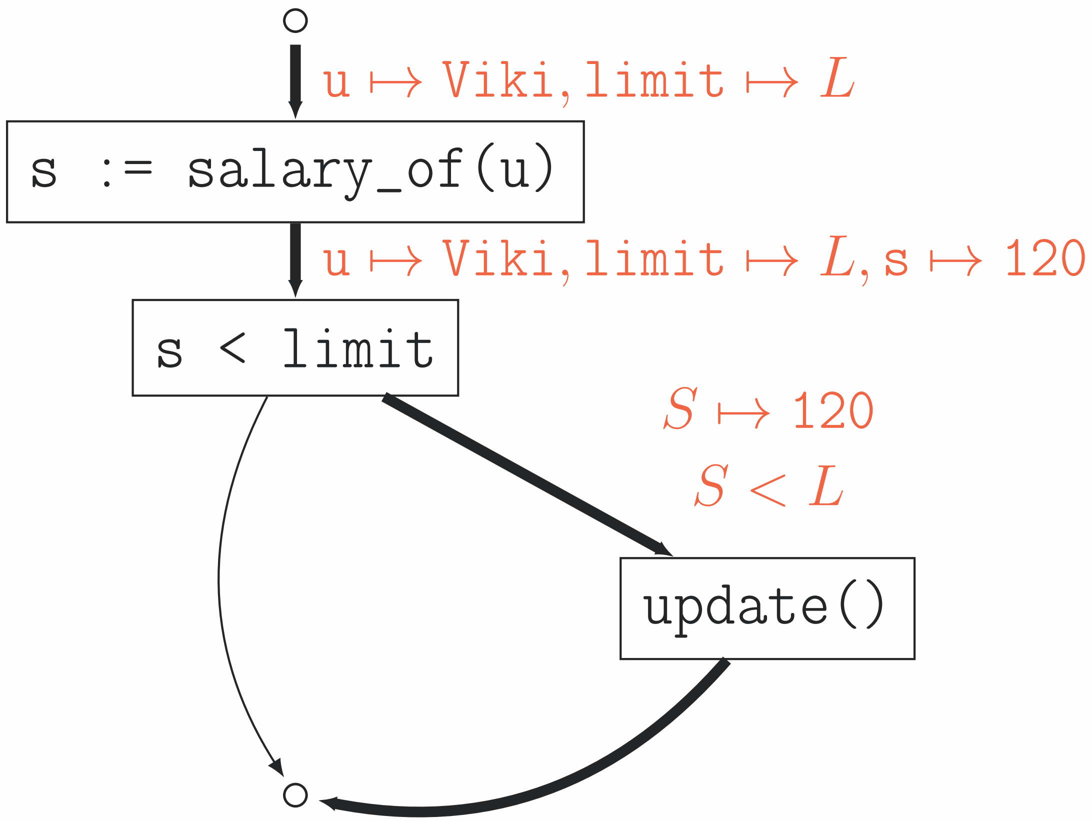
How to find test values of `u` and `limit` for `raise()`? It's impossible :(

Two Steps

1. Concrete (w/random input):



2. Symbolic (w/neglected condition):



Chapter #5:

Further Reading/Watching

Check this GitHub repo: ksluckow/awesome-symbolic-execution