Beijing Forest Studio

北京理工大学信息系统及安全对抗实验中心

符号执行简介

硕士研究生 柯懂湘

2019年1月1日

内容提要

- 技术起源
- 基本原理
- 面临问题&解决方案
- 实例
- 参考文献

预期收获

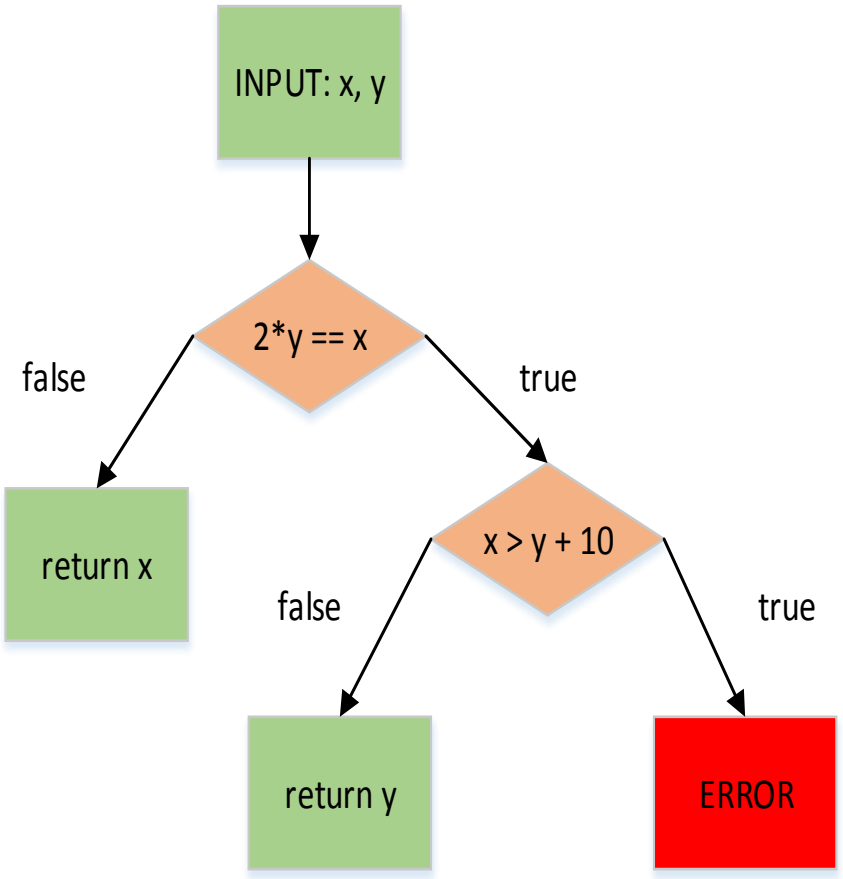
- 预期收获
  - 清楚符号执行用途
  - 了解符号执行基本原理
  - 了解符号执行面临的问题与解决方案
  - 简单使用现代符号执行引擎

技术起源

技术起源

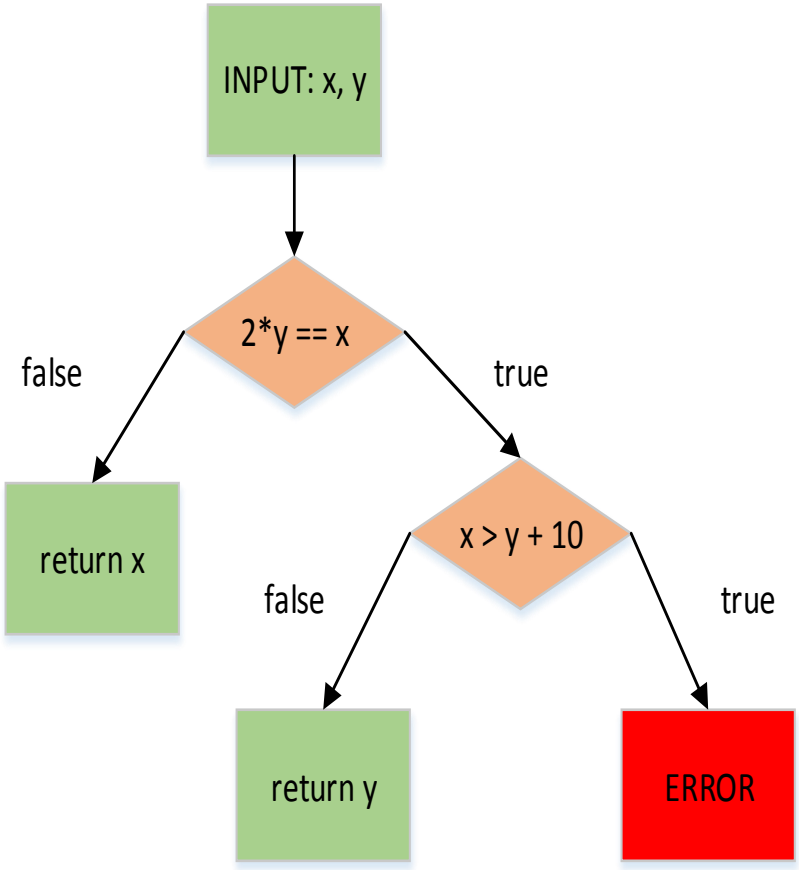
- 开发人员写出如下程序，其中包含一个错误。

```
1   int twice(int v)
2   {
3       return 2*v;
4   }
5   int testme(int x, int y)
6   {
7       int z = twice(y);
8       if (z == x) {
9           if (x > y + 10) {
10              ERROR;
11          } else {
12              return y;
13          }
14
15      } else {
16          return x;
17      }
18  }
19
20  int main()
21  {
22      int x, y;
23      cin >> x;
24      cin >> y;
25      testme(x, y);
26      return 0;
27  }
```



背景知识

- 作为测试人员，如何检查出上述错误?

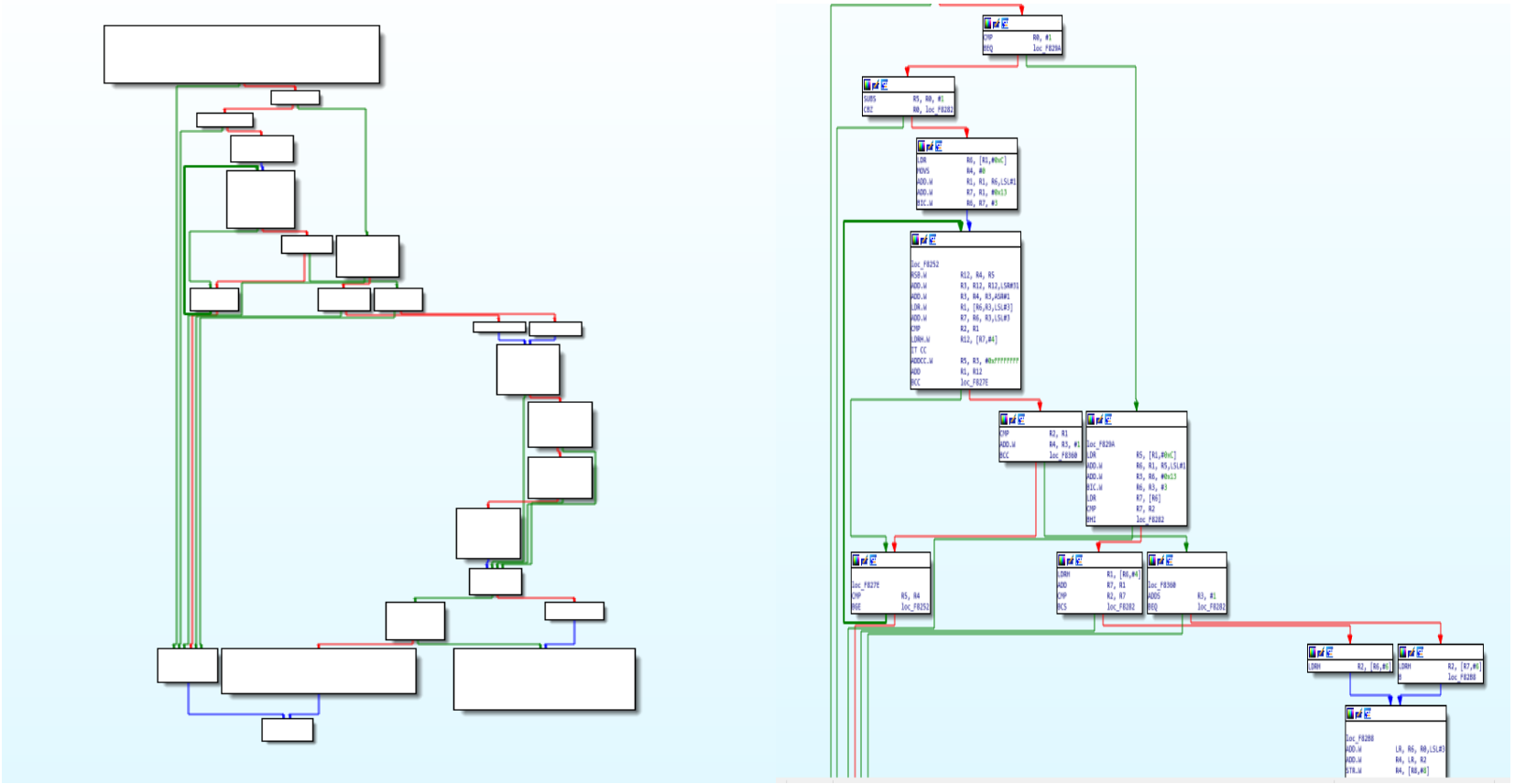


测试数据

x	y	result
4	1	4
3	2	3
10	5	5
8	4	4

技术起源

- 如何生成覆盖所有路径的测试用例？依靠人工分析代码？



基本原理

基本原理

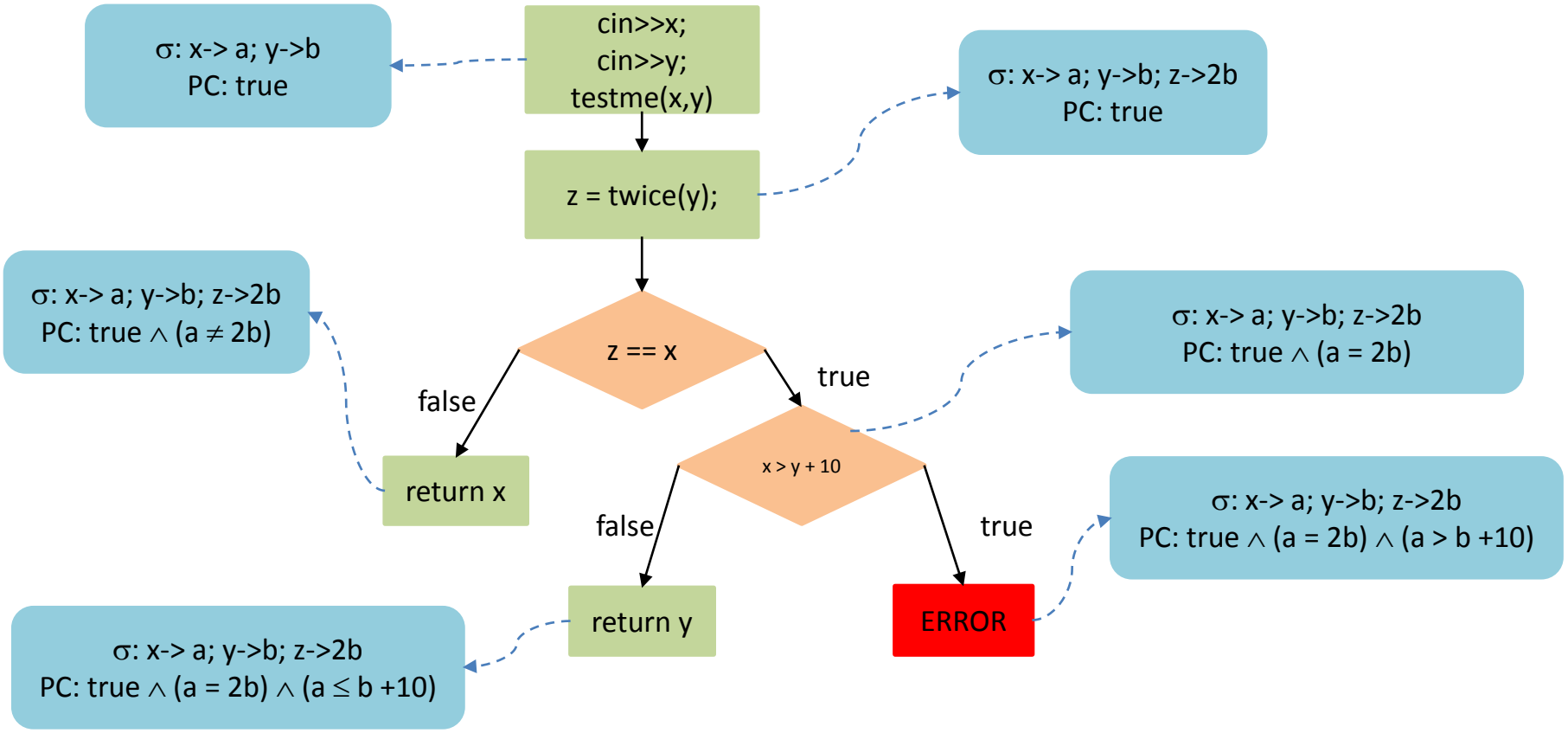
- 什么是符号执行?
  - 符号执行（Symbolic Execution）是一种程序分析技术。其可以通过分析程序来得到让特定代码区域执行的输入。
- 基本思想
  - 把程序的输入变为符号值，那么程序计算的输出值就是关于符号输入值的函数。

$$f(x,y)=\begin{cases} x & (x \neq 2y) \\ y & (x=2y \wedge x \leq y+10) \\ Error & (x=2y \wedge x > y+10) \end{cases}$$

基本原理

- 符号执行分析过程
  - 符号执行会在全局维护两个变量，符号状态σ，符号化的路径约束PC。
  - 符号状态σ，记录了程序中每个变量到符号表达式的映射。
  - 符号化路径约束PC，用来表示路径条件，初始值为true。
  - σ和PC会在符号执行过程中不断更新，当符号执行结束时，求解PC就可以得到覆盖所有路径的输入。

基本原理


σ: x-> a; y->b
PC: true
cin>>x;
cin>>y;
testme(x,y)
σ: x-> a; y->b; z->2b
PC: true
z = twice(y);
σ: x-> a; y->b; z->2b
PC: true ∧ (a ≠ 2b)
z == x
false
true
σ: x-> a; y->b; z->2b
PC: true ∧ (a = 2b)
return x
x > y + 10
false
true
σ: x-> a; y->b; z->2b
PC: true ∧ (a = 2b) ∧ (a > b +10)
return y
ERROR
σ: x-> a; y->b; z->2b
PC: true ∧ (a = 2b) ∧ (a ≤ b +10)

基本原理

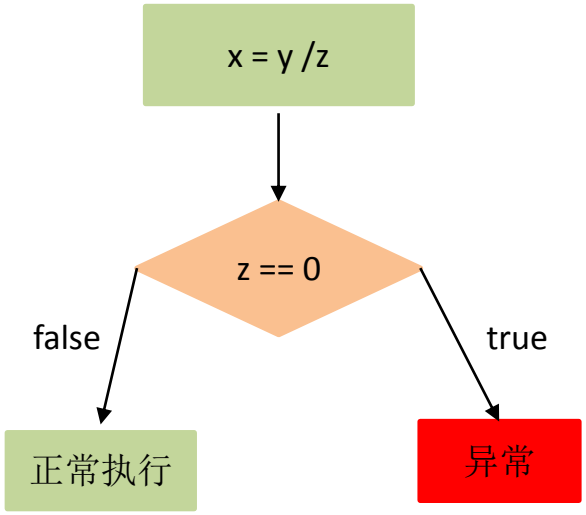
- 得到三组PC，对应三条路径
  - 路径1 PC: true $\wedge$ (a $\neq$ 2b)
  - 路径2 PC: true $\wedge$ (a = 2b) $\wedge$ (a $\leq$ b +10)
  - 路径3 PC: true $\wedge$ (a = 2b) $\wedge$ (a > b +10)

- 三组PC为三个方程组，求解方程组，即可得到该路径对输入的要求。
  - 路径1对输入的要求：a $\neq$ 2b，如 a = 3, b =1。
  - 路径2对输入的要求：b $\leq$ 10, a = 2b，如 a= 4, b =2。
  - 路径3对输入的要求：b > 10, a = 2b，如 a = 30, b = 15。

基本原理

- 如何利用符号执行发现程序Bug
  - 利用符号执行生成具体输入，使程序实际运行处理这些输入，并监控程序运行状态。
  - 在符号执行阶段发现程序Bug。

- 在符号执行阶段发现程序Bug
  - 一些敏感操作含有一些隐含路径。
    - 除法，除数为零
    - 指针解引用，数组越界读写



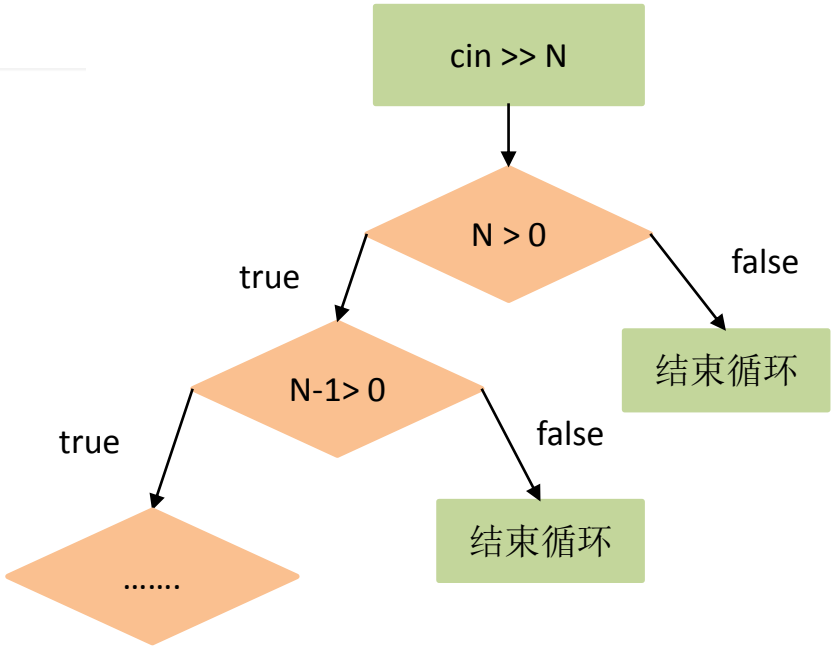
符号执行简介

面临问题&解决方法

面临问题&解决方案

- 问题1：如何处理循环和递归(循环的终止条件为符号化的)?
  - 可能存在无数条路径，任意数量的true加上一个false结尾。

```
1   void testme_inf()
2   {
3       int sum = 0;
4       int N, a = 0;
5       int buf[100] = {0};
6       cin >> N;
7       while (N > 0) {
8           buf[a++] = a;
9           N = N - 1;
10      }
11  }
12
```



面临问题&解决方案

- 解决方案：限制路径搜索
  - 限制符号执行时间
  - 限制循环迭代次数
  - 限制路径数量

面临问题&解决方案

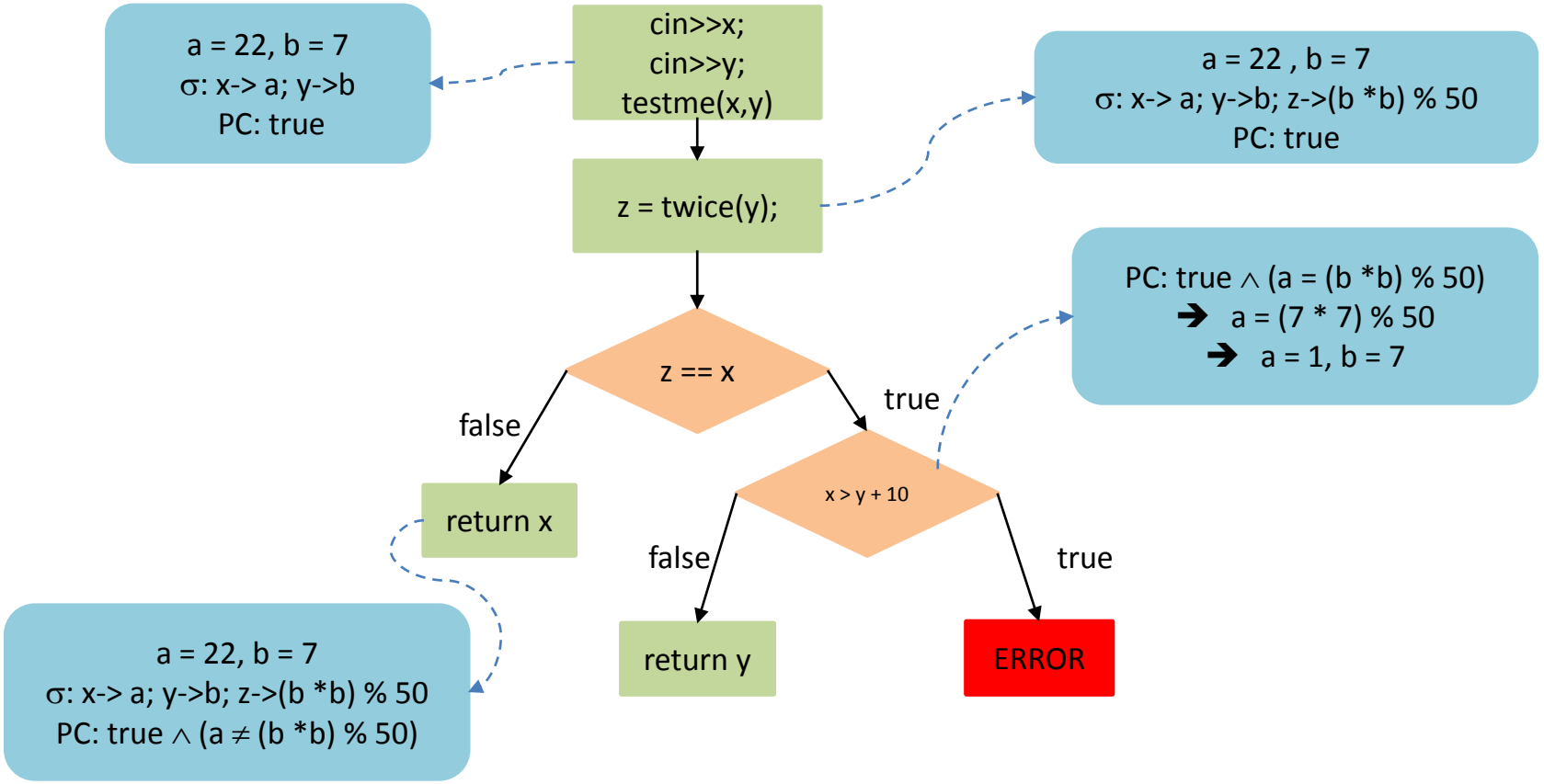
问题2：无法求解路径约束

- 路径约束为非线性约束
  - 如示例中twice函数修改为返回 (v*v) mod 50
  - 对应路径约束
    - 路径1约束条件: true $\wedge$ (a $\neq$ (b*b) mod 50)
    - 路径2约束条件: true $\wedge$ (a = (b*b) mod 50) $\wedge$ (a $\leq$ b +10)
    - 路径3约束条件: true $\wedge$ (a = (b*b) mod 50) $\wedge$ (a > b +10)

面临问题&解决方案

- 解决方案：混合实际执行和符号执行（concolic execution）
  - 首先使用一些随机的输入来执行程序，收集执行过程中条件语句对输入的符号化约束。
  - 在已有实际输入得到的路径上，对分支路径条件进行取反，得到的输入就可以使程序执行走向另外一条路径。
  - 这个过程会不断地重复，直到所有的路径都被探索，或者用户定义的覆盖目标达到，或者时间开销超过预计。

面临问题&解决方案


a = 22, b = 7
σ: x-> a; y->b
PC: true
cin>>x;
cin>>y;
testme(x,y)
a = 22 , b = 7
σ: x-> a; y->b; z->(b *b) % 50
PC: true
z = twice(y);
PC: true ∧ (a = (b *b) % 50)
➔ a = (7 * 7) % 50
➔ a = 1, b = 7
z == x
false
true
x > y + 10
return x
false
true
a = 22, b = 7
σ: x-> a; y->b; z->(b *b) % 50
PC: true ∧ (a ≠ (b *b) % 50)
return y
ERROR

面临问题&解决方案

- 问题3：如何处理代码中的系统调用?
  - 一些系统函数调用无法用数学表示，如fopen等系统调用会截断符号化赋值过程。

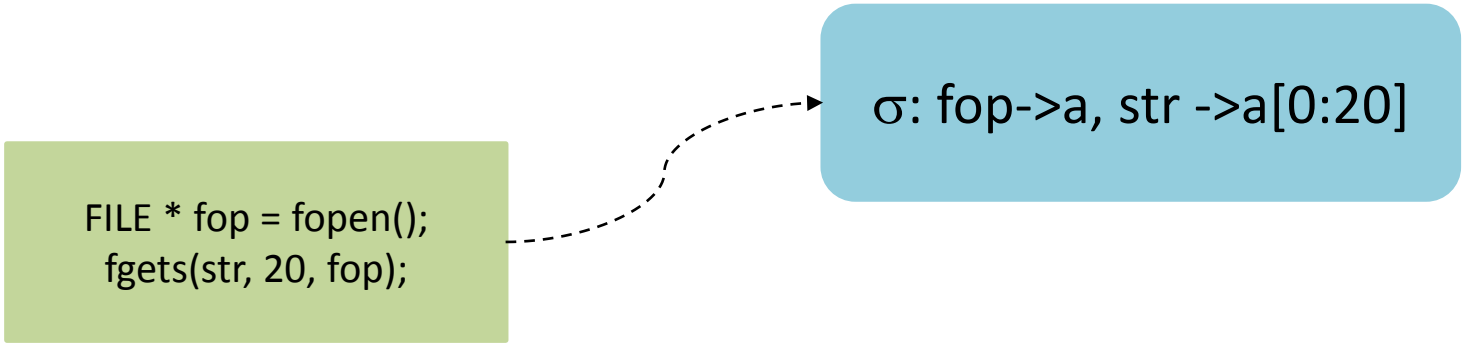
```
int main(int argc, char *argv[])
{
    if (argc == 2) {
        FILE *fop = fopen("test.txt", "r");
        char str[20];
        fgets(str, 20, fop);
        if (str == 'a') {
            ...
        }
        else {
            ....
        }
        ...

    }
    ...
}
```

如何对fop，str进行符号化赋值?

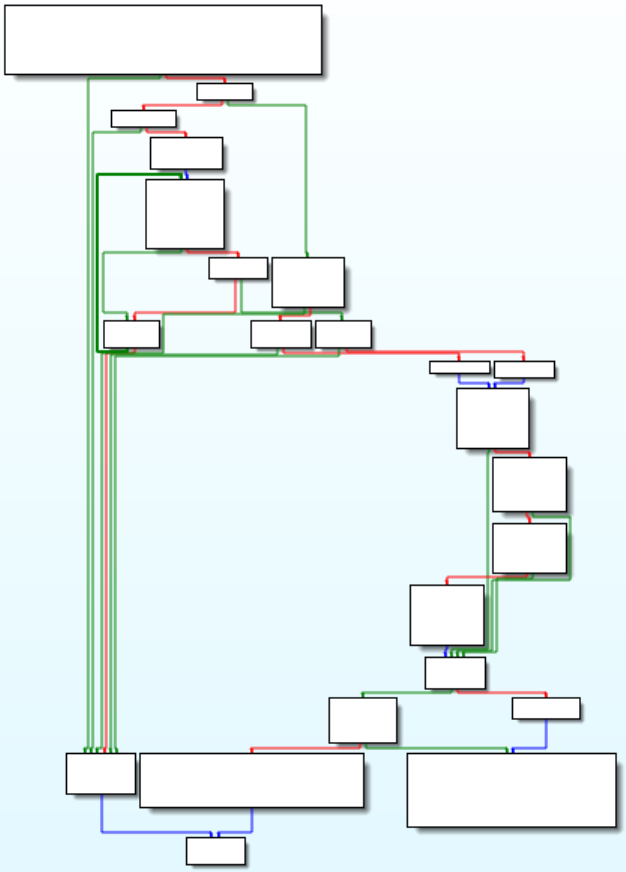
面临问题&解决方案

- 解决方案：
  - 现代符号执行引擎会对系统函数进行重新建模。
  - 比如在新模型中，fopen函数返回一个数组，fgets函数修改为从一个变量中获取内容。（实际在对系统函数进行建模的时候，会要处理更复杂的情况。）



面临问题&解决方案

- 问题4：路径选择
  - 符号执行要探索的执行路径依分支数指数增长。
  - 在时间和资源有限的情况下，应该对最相关的路径进行探索，如何选取最值得探索的路径?
- 解决方案
  - 使用静态控制流图来指导路径选择，尽量选择与未覆盖指令最多的路径。



符号执行简介

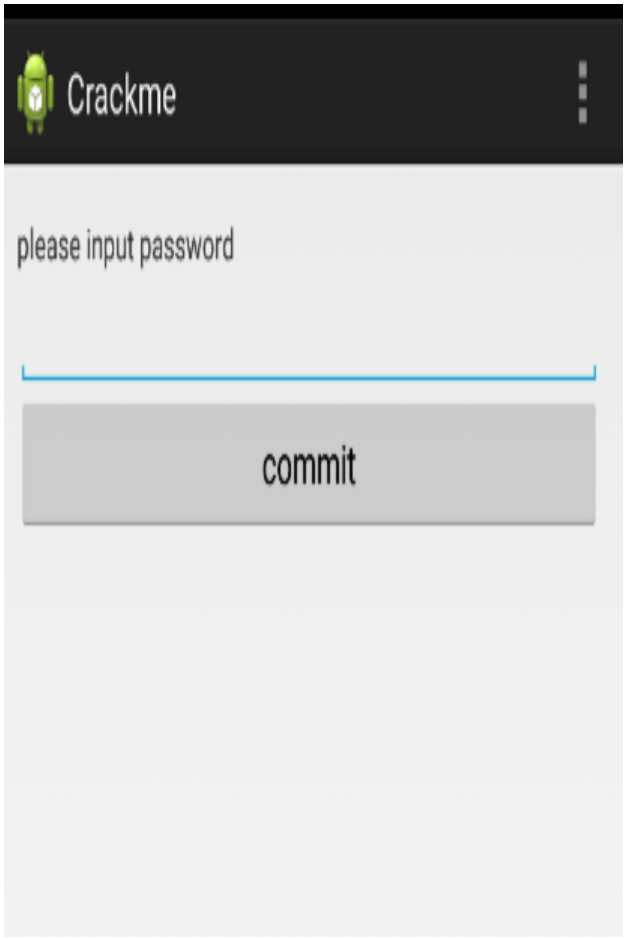
实例

Angr简介

- Angr
  - Angr是一个基于符号执行的二进制程序分析框架。
  - 使用Python实现，支持多种系统架构(X86、ARM、ARM64、MIPS）。
- 工作流程
  - 加载待分析的二进制程序
  - 将二进制程序转换为中间语言（intermediate representation, IR）。
  - 对中间语言进行分析
    - 完整的或者部分的静态分析（依赖关系分析，程序分块）
    - 符号执行

破解实例

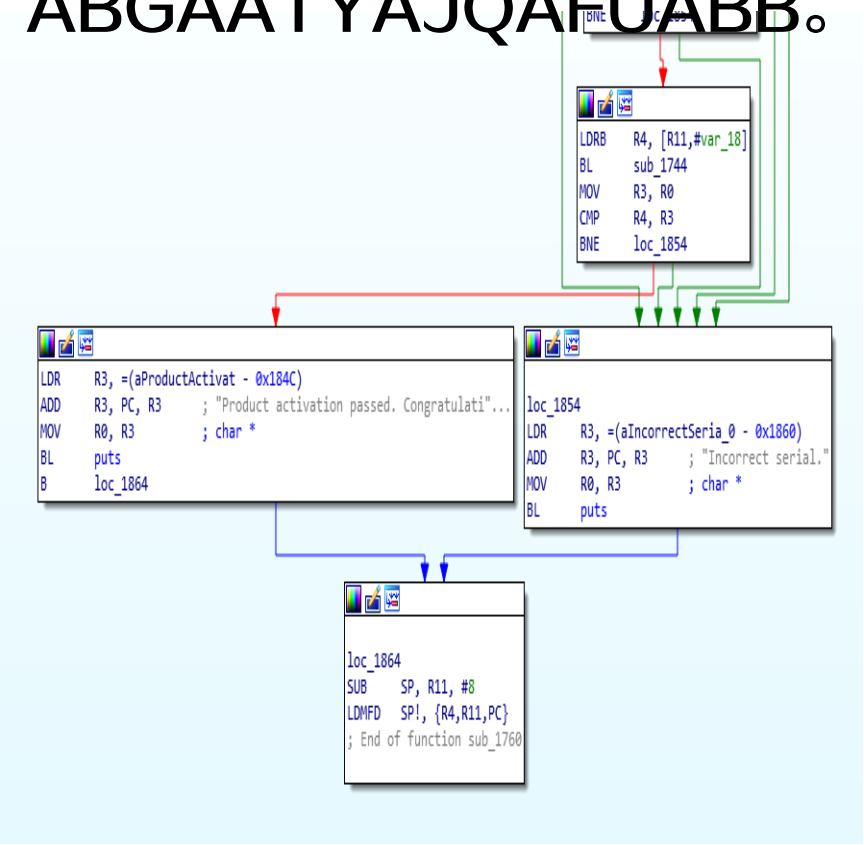
使用Angr生成正确的注册码




```
.text:00001760 sub_1760                                ; CODE XREF: main+B0↓p
.text:00001760
.text:00001760 var_20          = -0x20
.text:00001760 var_1C          = -0x1C
.text:00001760 var_1B          = -0x1B
.text:00001760 var_1A          = -0x1A
.text:00001760 var_19          = -0x19
.text:00001760 var_18          = -0x18
.text:00001760 var_14          = -0x14
.text:00001760 var_10          = -0x10
.text:00001760 var_C           = -0xC
.text:00001760
.text:00001760                 STMFD   SP!, {R4,R11,LR}
.text:00001764                 ADD     R11, SP, #8
.text:00001768                 SUB     SP, SP, #0x1C
.text:0000176C                 STR     R0, [R11,#var_20]
.text:00001770                 LDR     R3, [R11,#var_20]
.text:00001774                 STR     R3, [R11,#var_10]
.text:00001778                 MOV     R3, #0
.text:0000177C                 STR     R3, [R11,#var_14]
.text:00001780                 B       loc_17D0
.text:00001784 ; ---------------------------------------------------------------------------
.text:00001784
.text:00001784 loc_1784                                ; CODE XREF: sub_1760+78↓j
.text:00001784                 LDR     R3, [R11,#var_10]
.text:00001788                 LDRB    R2, [R3]
.text:0000178C                 LDR     R3, [R11,#var_10]
.text:00001790                 ADD     R3, R3, #1
.text:00001794                 LDRB    R3, [R3]
.text:00001798                 EOR     R3, R2, R3
.text:0000179C                 AND     R2, R3, #0xFF
.text:000017A0                 MOV     R3, #0xFFFFFFF0
.text:000017A4                 LDR     R1, [R11,#var_14]
.text:000017A8                 SUB     R0, R11, #-var_C
.text:000017AC                 ADD     R1, R0, R1
.text:000017B0                 ADD     R3, R1, R3
.text:000017B4                 STRB    R2, [R3]
.text:000017B8                 LDR     R3, [R11,#var_10]
.text:000017BC                 ADD     R3, R3, #2
.text:000017C0                 STR     R3, [R11,#var_10]
.text:000017C4                 LDR     R3, [R11,#var_14]
```

实例

利用Angr，只需要知道如果输入正确注册码，函数执行路径的起点和终点，就可以生成正确注册码：ABGAATYAJQAFUABB。




```python
import angr
import claripy
import base64

def main():
    load_options = {}
    //加载二进制程序
    b = angr.Project("./validate", load_options = load_options)
    state = b.factory.blank_state(addr=0x401760)
    code = claripy.BVS('code', 10*8)
    state.memory.store(concrete_addr, code, endness='Iend_BE')

    sm = b.factory.simulation_manager(state)

    //设置正确路径的起点与终点
    sm.explore(find=0x401840, avoid=0x401854)
    found = sm.found[0]

    # Get the solution string from *(R11 - 0x20).

    solution = found.solver.eval(code, cast_to=bytes)

    print(base64.b32encode(solution))
    return code, found
```

参考文献

- Cadar C, Sen K. Symbolic execution for software testing: three decades later[M]. ACM, 2013.
- R. Majumdar and K. Sen. Hybrid concolic testing. In *Intl.Conf. on Software Engineering*, 2007.
- C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Systems Designand Implementation*, 2008.
- 符号执行入门 https://zhuanlan.zhihu.com/p/26927127
- 使用符号执行解决 Android Crackme https://www.anquanke.com/post/id/85445

道德经

知人者智，自知者明。
胜人者有力，自胜者强。
知足者富，强行者有志。
不失其所者久，死而不亡者，寿。

谢谢！