《软件安全漏洞分析与发现》

第7章 符号执行技术

闫佳
中国科学院软件研究所
2025年4月30日

提纲

- 一、经典符号执行（原理）
- 二、动态符号执行（过程）
- 三、并行符号执行（思想）
- 四、选择符号执行（思想）

提纲

- 一、经典符号执行（原理）
- 二、动态符号执行（过程）
- 三、并行符号执行（思想）
- 四、选择符号执行（思想）

1.1 产生的背景—程序测试

- 程序测试无处不在
- 人工测试费用高昂，在程序开发费用中占比超过30%

1.1 程序测试

TCA

- 示例

Example [edit]

Consider the program below, which reads in a value and fails if the input is 6.

```
y = read()
y = 2 * y
if (y == 12)
    fail()
print("OK")
```

✓ 随机测试为何无法有效构造触发漏洞的测试用例？

✓ 怎样提高测试的效率？

1.2 符号执行

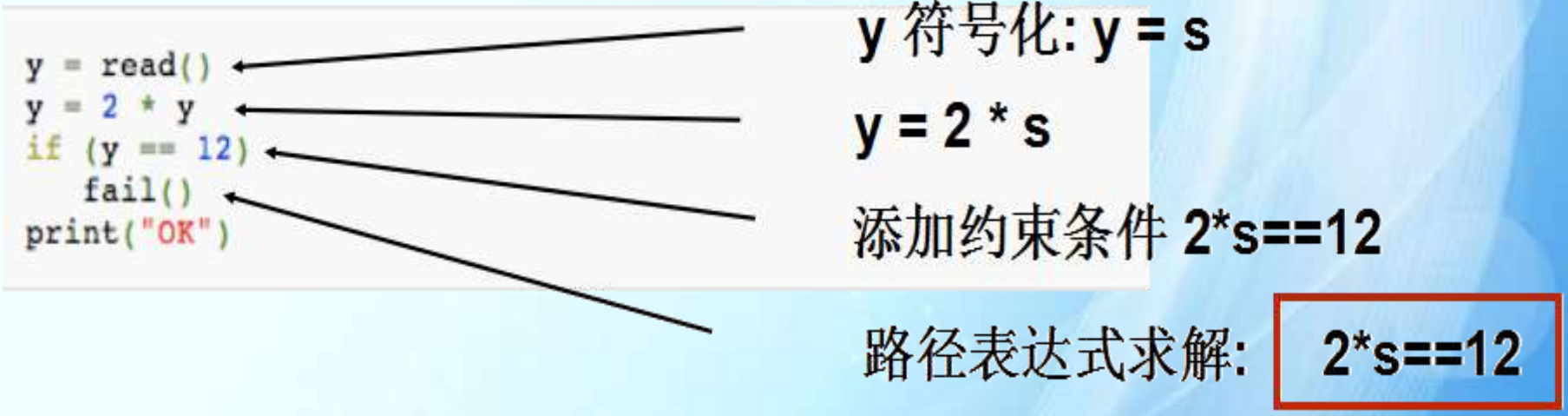
- 用于软件测试的静态程序分析方法
  - Jame C.King，Symbolic Execution and Program Testing, CACM 1976
- 基本思想
  - 符号执行使用输入变量的符号值模拟执行程序，各种指令的操作都在符号上进行： x = symA, y = symB
    - 符号变量表示一个范围内的取值，如$a_i \in [0, 2^{32} - 1]$
    - 符号执行过程中变量的值是由符号和常量组成的表达式：f(symA, symB, C)
  - 针对不同的程序分支分别进行符号执行，遍历程序中的可达路径
    - 针对不同的程序分支分别进行符号执行，同时记录分支路径约束，并通过路径约束求解判断路径是否可达
  - 程序分析：漏洞检测&软件测试&恶意代码分析
    - 堆溢出漏洞：
      - malloc(symA) & memcpy(dst, src, symB)
      - symA < symB？

1.2 符号执行

- 符号执行过程
  - 使用符号变量替换具体值作为程序输入
    - 符号变量表示一个范围内的取值，如$a_i \in [0, 2^{32} - 1]$
  - 非条件跳转语句：程序执行过程中生成并记录每个符号的表达式
    - $a_i = 100 \times \mathrm{b}_i + 20$
  - 条件跳转语句：符号执行分化出两个对应的分支，分别对每个分支继续进行符号执行，同时每个分支添加对应条件约束到对应路径约束
    - if (a > 10)
      - 分支1约束：a>10 -> 更新路径约束：当前约束∧（a>10）
      - 分支2约束：a<=10 ->更新路径约束：当前约束∧（a<=10）

1.2 符号执行

- 主要价值
  - 将程序测试问题转换为变量表达式的求解问题，有效提高了路径覆盖效率
    - 如下图，随机测试命中if分支的概率为$\frac{1}{2^{32}}$,符号执行命中的概率为1，并且只用构造一个样本



1.2 符号执行

- 思考：

```
1：    x = read()
2：    y = read()
3：    if ((x+y)==12)
4：    {
5：        fail();
6：    }
```

A. 路径约束是什么？

B. 有怎样的解？

1.2 符号执行

- King提出的符号执行理想使用场景：
  - 只处理整型数据
    - 实际程序指令中包含多种类型数据，例如，浮点类型，SIMD（single instruction, multiple data）类型等
    - 考虑到约束求解器的处理性能及相关指令的复杂程度，理想场景下不对这些复杂类型数据符号化
  - 路径爆炸：可以探索的潜在程序路径太多
    - 循环、递归等造成分支路径规模无穷扩展
  - 实际指令集中部分复杂指令很难进行符号执行操作
    - 例如，SIMD、SSE指令集中的部分指令
      - pmaxub 把源存储器与目的寄存器按字节无符号整数比较,大数放入目的寄存器对应字节

1.2 符号执行

- 经典符号执行：程序语言限定范围
  - 程序变量类型
    - 只包含整型数据
  - 程序语句类型
    - 变量声明语句，例如，a=3
    - IF条件语句，要求条件语句中的表达式可以转换成expr≥0的形式
      - 例如，if(a≤1) → 1-a≥0
    - 无条件跳转语句，例如，goto语句
    - 变量操作语句，例如，读写操作，变量数学运算（+、-、*）

1.2 符号执行

- 符号输入对程序语义的影响
  - 程序语义
    - 即程序指令序列的执行语义，决定程序的执行流程和行为
      - eg : if (x < 1){x = x + 1;}else{}
        - » 当变量x为具体值时，程序执行到if条件语句、运算语句处，指令的行为是确定的（ 假设x = 5，程序选择false分支执行）
  - 符号输入
    - 即用符号变量替换具体值作为程序的输入
      - eg : sum(x , y), input : x = a, y = b
    - 程序的语义是否还确定？

1.2 符号执行

- 符号输入对程序语义的影响
  - 程序语义的变化
    - **数据对象**
      - 用a_i替换函数输入中的整型变量
    - **程序语句**
      - 数学运算，r = 1 + 4 → r = x + y
      - 数据读写操作， read (address， size)
        - » 当address由输入参数控制时，read函数返回符号值
      - 无条件跳转语句， goto label
      - 条件跳转语句， if (a) {} else {}
        - » **因为符号变量a取值的不确定性，导致在分析条件语句时无法决定程序控制流的走向**

1.2 符号执行

- 符号输入对程序语义的影响
  - 程序语义的变化
    - **具体执行时程序执行状态**
      - 程序状态通常包括程序变量的取值，指令计数等信息，使用这些信息可以决定程序执行的状态
    - **符号执行时程序执行状态**
      - 程序取值未知，IF语句的分支选择未知，使用原有状态信息无法决定程序执行状态及控制流。
  - 针对不同分支建立各自的**Path Condition（PC）**“路径条件”
    - 唯一的确定一条执行路径，消除符号值引入导致程序执行的不确定

1.2 符号执行

- 路径条件（Path Condition）
  - 符号执行过程中，在“if分支”处无法决定程序执行哪条分支，这就需要符号执行引擎主动选择并记录程序的走向，**PC辅助完成该工作**
  - 例如
    - 程序执行路径上分别经过if_1、if_2、if_3三个分支
      - if_1：$a_1 \geq 0$
      - if_2：$a_1+2*a_2 \geq 0$
      - if_3：$a_3 \geq 0$
    - 假设符号执行引擎在三个分支处的选择分别是：
      - if_1：true，if_2：true，if_3：false
    - 程序当前的路径表达式为：

$$pc = (a_1 \geq 0) \wedge (a_1 + 2 * a_2) \wedge \neg (a_3 \geq 0)$$

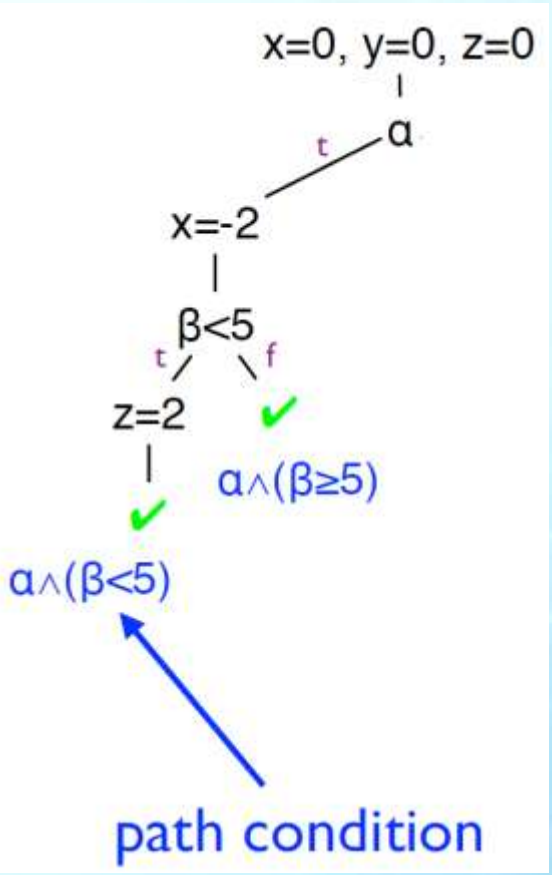
1.2 符号执行

- 路径条件(Path Condition)
  - 在每个与符号变量相关的条件语句处更新路径条件表达式
    - 将约束表达式及其对应的真值添加到PC中
    - 例如：if-else语句中的判定条件为C
      - 如果选择走 if 分支则 PC = PC ∧ C
      - 如果选择 else 分支则 PC = PC ∧¬C

1.2 符号执行

- 路径条件(Path Condition)
  - 示例

```
1. int a = α, b = β, c = γ;
2.                  // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.   x = -2;
6. }
7. if (b < 5) {
8.   if (!a && c)  { y = 1; }
9.   z = 2;
10.}
11.assert(x+y+z!=3)
```



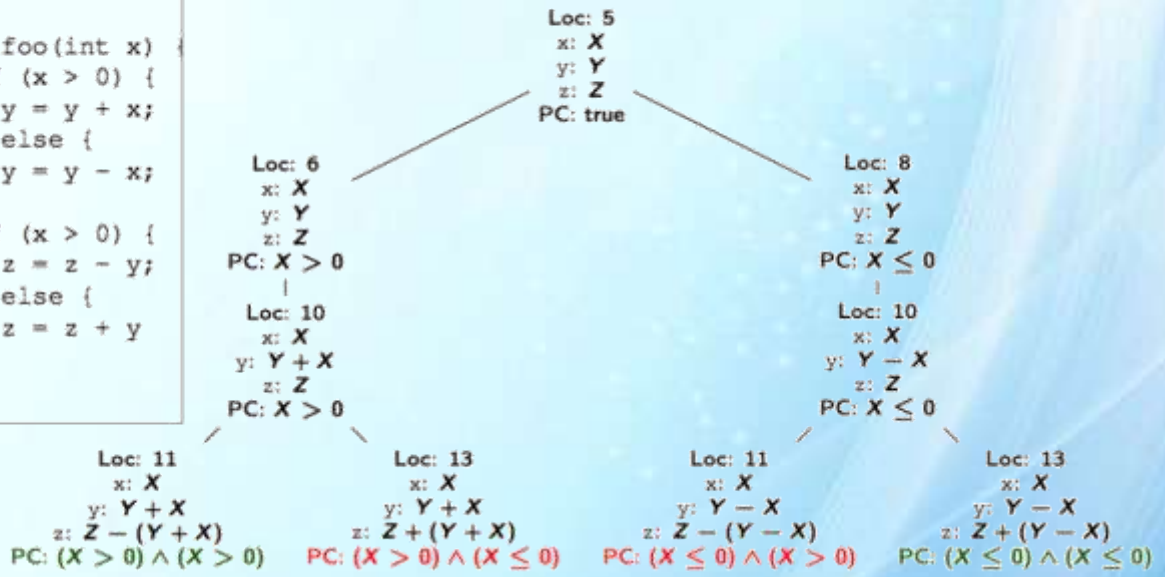
1.2 符号执行

- 路径表达式求解的应用
  - 路径可达性分析：路径遍历
    - 路径表达式记录了路径执行需要满足的约束条件，如果表达式无可满足约束解，则说明路径不可达，即该路径无效（PC＝＝False）
  - 软件漏洞检测
    - 执行过程中，满足漏洞特定的约束条件，且对应的路径表达式存在可满足约束解，则说明该漏洞确实存在
      - 例如：缓冲区溢出漏洞，PC∩（a > 256）==True
  - 测试用例自动生成
    - 使用约束求解器对可达路径的路径表达式进行求解，将表达式中符号变量的可行解构造成测试用例。
    - 例如，solve(PC) ➔ result : x=0,y=0 ➔ input(x=0,y=0)

1.3 执行树

- 执行树基本概念
  - 执行树是描述程序执行路径的树形结构
    - 每一个节点对应一条程序语句
    - 每一条边对应语句间的跳转关系
    - 执行树中还可包含变量状态等信息

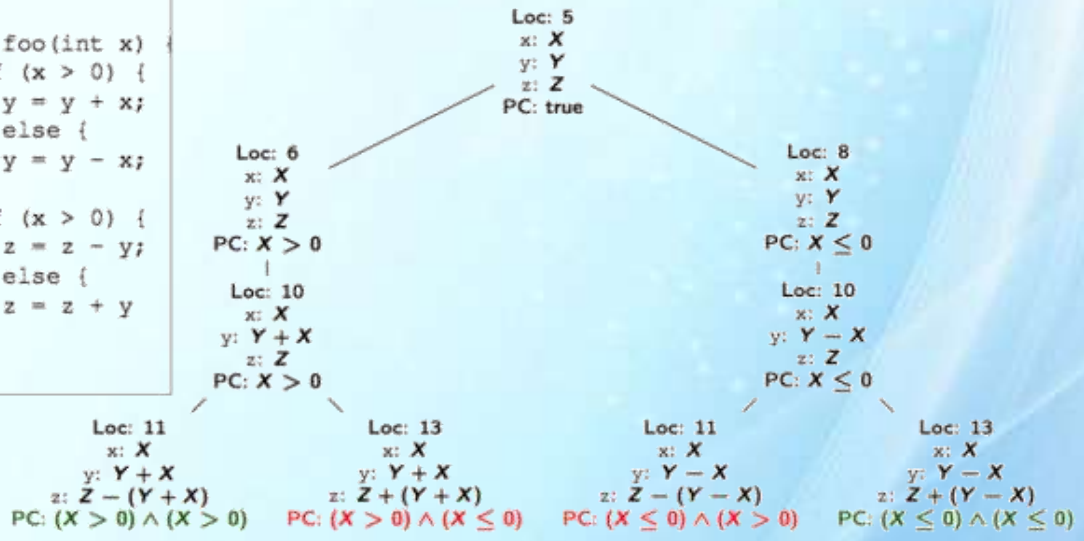
```
1  int y;
2  int z;
3  ...
4  int foo(int x) {
5    if (x > 0) {
6      y = y + x;
7    } else {
8      y = y - x;
9    }
10   if (x > 0) {
11     z = z - y;
12   } else {
13     z = z + y
14   }
15 }
```



1.3 执行树

- 执行树的性质
  - 每个叶节点对应程序的一组**输入集合**
    - 例如，下图中最左边的叶节点对应的输入集合中的一个元素为 x=1，y=0，z=0
  - 任意两个叶节点对应的路径都至少拥有一个公共节点

```
1  int y;
2  int z;
3  ...
4  int foo(int x) {
5    if (x > 0) {
6      y = y + x;
7    } else {
8      y = y - x;
9    }
10   if (x > 0) {
11     z = z - y;
12   } else {
13     z = z + y
14   }
15 }
```



1.3 执行树

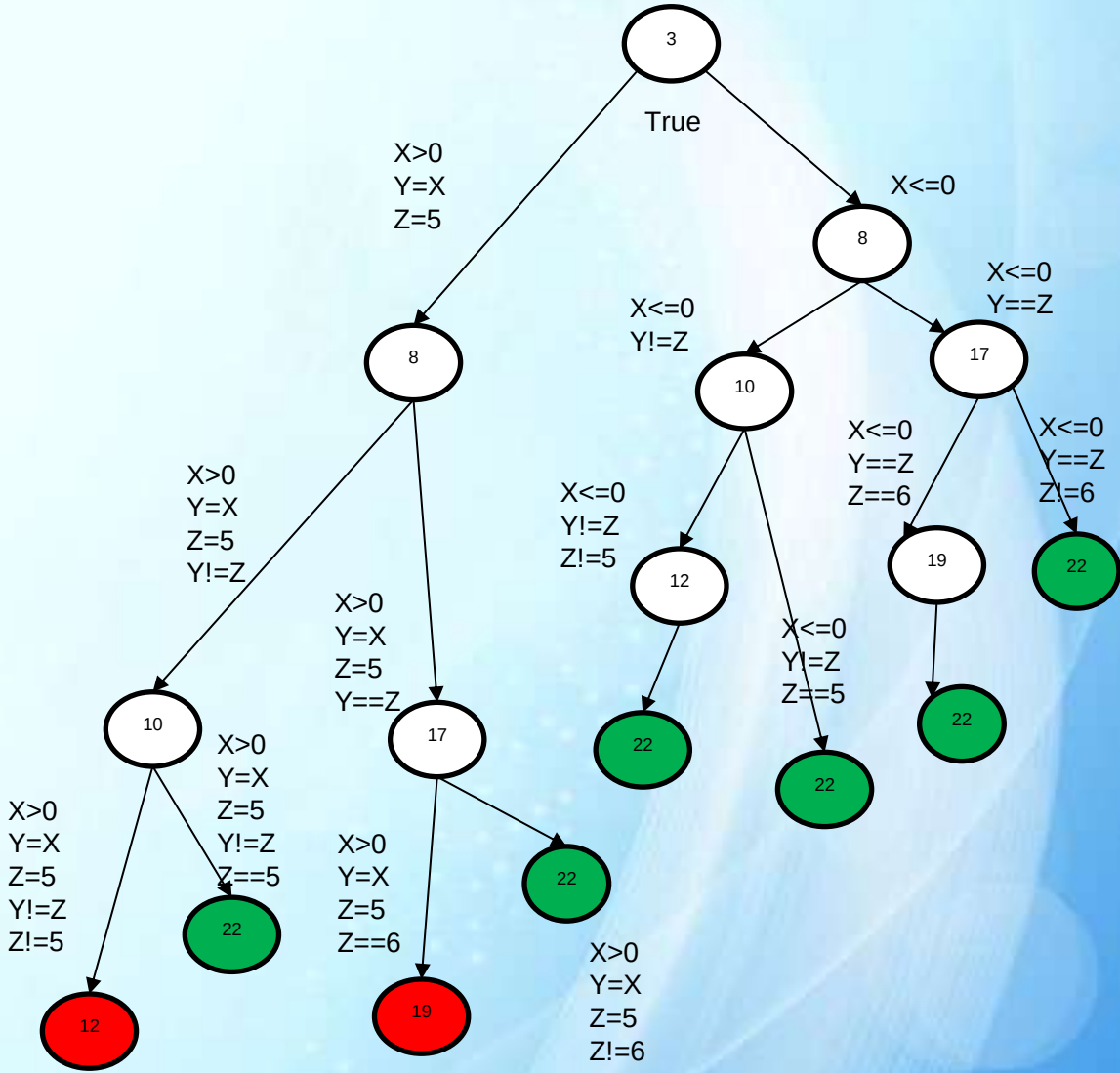
- 执行树的性质

```
1.  func(x, y, z)
2.  {
3.      if(x>0)
4.      {
5.          y = x;
6.          z = 5;
7.      }
8.      if(y!=z)
9.      {
10.         if(z!=5)
11.         {
12.             printf(“Never Say Hello!”)
13.         }
14.     }
15.     else
16.     {
17.         if(z==6)
18.         {
19.             printf(“Oh, My God!”)
20.         }
21.     }
22.     printf(“Done!”)
23. }
```

1.3 执行树

- 执行树的性质

```
1.  func(x, y, z)
2.  {
3.      if(x>0)
4.      {
5.          y = x;
6.          z = 5;
7.      }
8.      if(y!=z)
9.      {
10.         if(z!=5)
11.         {
12.             printf("Never Say Hello!")
13.         }
14.     }
15.     else
16.     {
17.         if(z==6)
18.         {
19.             printf("Oh, My God!")
20.         }
21.     }
22.     printf("Done!")
23. }
```

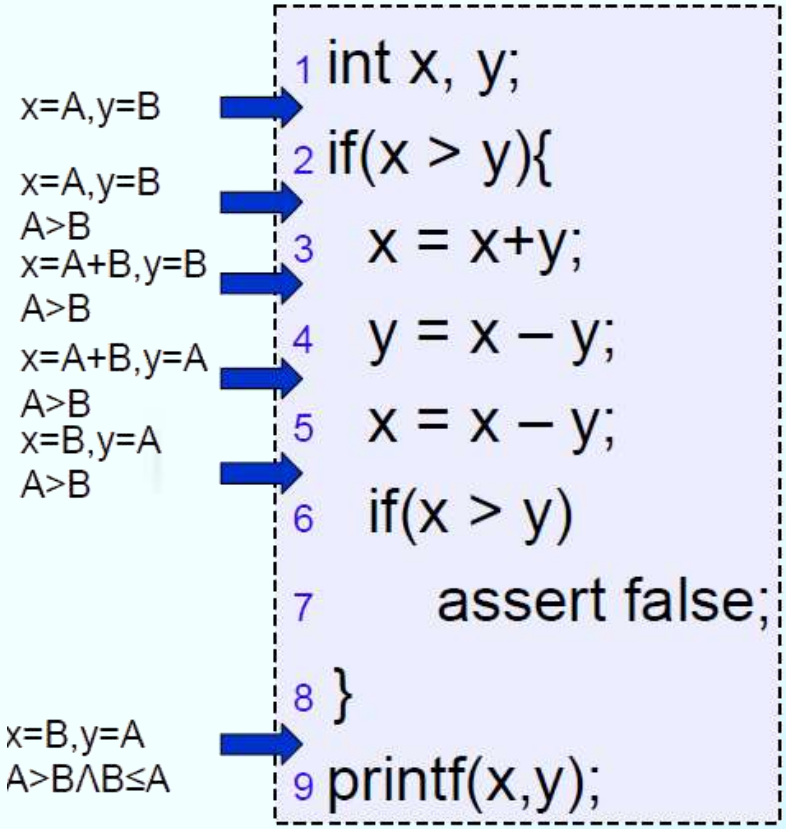


1.4 流程及示例

- 符号执行流程
  - 符号化函数输入
  - 模拟执行程序
    - 未遇到条件分支时，根据指令语义进行处理
    - 遇到条件分支时，分出新的模拟执行进程执行另外一条分支，并分别为两个执行过程更新路径条件表达式
  - 单路径模拟执行完成时，根据分析目标进行处理
    - 例如目标为路径可达性分析时，则对路径表达式进行求解
  - 当完成对所有路径的模拟执行时，符号执行测试结束

1.4 流程及示例

- 符号执行示例



inputs that cover ***else*** branch at stmt. 2:

x = 3 y = 4

inputs that cover ***then*** branch at 2 and ***else*** at 6:

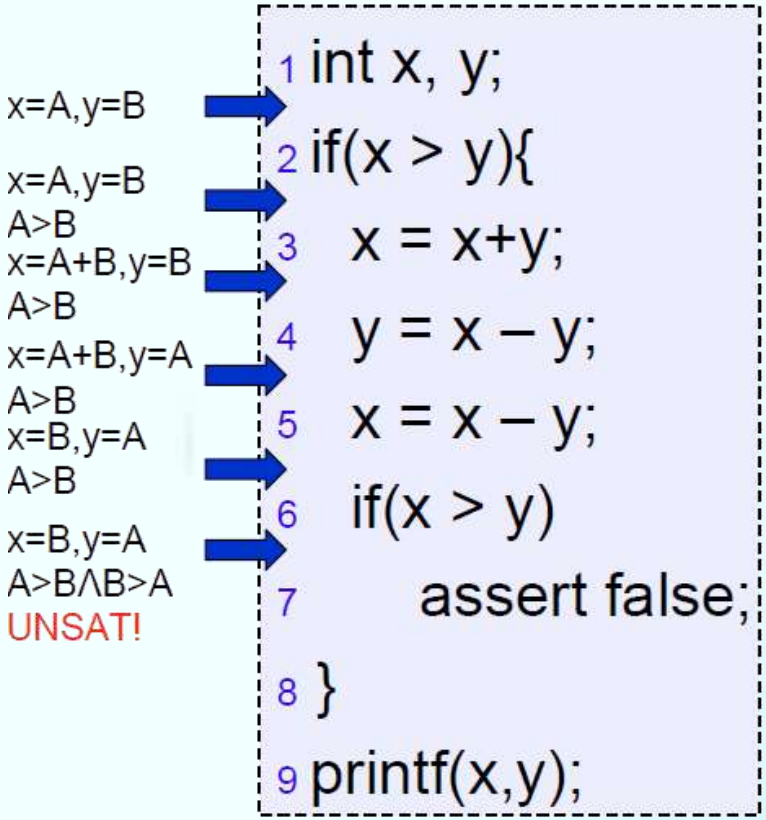
x = 5 y = 1

One solution of the constraint A>B ∧ B≤A is

A = 5, B = 1

1.4 流程及示例

- 符号执行示例



inputs that cover ***else*** branch at stmt. 2:

x = 3 y = 4

inputs that cover ***then*** branch at 2 and ***else*** at 6:

x = 5 y = 1

inputs that cover ***then*** branch at 2 and ***then*** at 6:

Does not exist!

1.5 约束求解

- 约束求解问题
  - 表示为一个三元组P = <V,D,C>
  - V表示变量的集合，$V=\{v_1,v_2,\ldots,v_n\}$
  - D表示变量可能取值的有限集合
  - C表示约束条件集合
- 约束求解
  - 目标：在变量的取值范围中找到满足所有约束条件的一组解
    - 若P至少有一组解，则称P是可满足的，反之为不可满足的

1.5 约束求解

- 约束求解的应用
  - 通过抽象建模，形式化描述，很多问题可转化为约束求解问题
    - N-皇后问题
      - 在国际象棋棋盘上放置八个皇后，使得任两个皇后都不能处于同一条横行、纵行或斜线上
      - 约束条件：
        - » 1、$x_i = 1，2，...，8$（x_i为第i个皇后在棋盘的横坐标）
        - » 2、不同列：$x_i \neq x_j$
        - » 3、不同对角线：$|i-j| \neq |x_i - x_j|$
    - 地图着色问题
    - 作业调度

1.5 约束求解

- 约束求解技术
  - 约束求解技术中通常会使用两种方法：搜索和推理
  - 搜索
    - 回溯法：先为约束满足问题中的部分变量赋值，在此基础上通过相容性技术为其余的变量赋值，并反复操作将部分解扩展为完整解
    - 随机搜索策略，由局部最优解逐渐向全局最优解靠拢，例如爬山法，模拟退火等算法
  - 推理
    - 相容性技术：用于约束求解的预处理阶段，通过相容性判断为搜索空间剪枝，降低搜索的复杂度

1.5 约束求解

- 约束求解器模型
  - SAT = 布尔可满足性理论模型
    - 对于布尔变量集合所构成的布尔函数，求解是否存在变量集合的一种分布使得布尔函数的取值为1
    - SAT只能求解布尔表达式
    - 处理能力有限，只能分析整数等简单的数据类型
  - SMT = Satisfiability modulo theory = SAT++
    - 为了对包含实数、数组、列表以及复杂函数的合取表达式进行判定，研究人员在SAT的基础上添加了对位向量和数组等模块理论的支持

1.5 约束求解

- SMT工具介绍
  - SMT工具列表

SMT 求解器	支持的操作系统	支持的求解理论
ABsolver[28]	Linux	线性计算、非线性计算
Beaver[29]	Linux/Windows	位向量
Boolector[30]	Linux	位向量、数组
CVC4[31]	Linux/Mac OS	线性计算、数组、位向量、未解释函数、符号、有理数与整数、元组
MathSAT[32]	Linux	空理论、线性计算、位向量、数组
MiniSmt[33]	Linux	非线性计算
OpenSMT[34]	Linux/ Mac OS/Windows	空理论、线性计算、位向量
SMT-RAT[35]	Linux/Mac OS	线性计算、非线性计算
STP[36]	Linux/OpenBSD/ Windows/Mac OS	位向量、数组
UCLID[37]	Linux	空理论、线性计算、位向量
Yices[38]	Linux/ Windows/Mac OS	
Z3	Linux/Mac OS/ Windows/FreeBSD	空理论、线性计算、非线性计算、位向量、数组、多种数据类型、量化

1.5 约束求解

- STP求解器
  - 由Stanford大学的Vijay Ganesh在2005年至2012年间主持开发和维护
    - 目前托管在guthub上，最后更新2024.6
    - STP可作为独立的求解工具，也可作为其他工具的求解组件
    - 提供了C，python，SMT-LIB接口
  - STP性能优越，被众多分析工具所使用
    - KLEE: Cadar, Dunbar, Engler
    - EXE : Bug Finder by Dawson Engler, Cristian Cadar and others at Stanford
    - MINESWEEPER: Bug Finder by Dawn Song and her group at CMU

1.5 约束求解

- Z3求解器
  - 微软团队设计研发
    - 使用C++实现，提供了python、C、SMT-LIB、.NET等语言的调用接口
  - Z3致力于解决软件验证和分析中的问题，支持大量理论模型
    - 相比于STP、Yices等求解器，其性能优势明显，在2007~2011年的各SMT竞赛中取得了多项第一
  - Python接口举例
    - >>> a = Int('a')
    - >>> solve(a > 0, a < 2)
    - [a = 1]

    Int('a'）函数创建了一个整数变量，并将该变量命名为a，solve函数对括号中的约束条件集合进行求解

1.6 符号执行技术分类

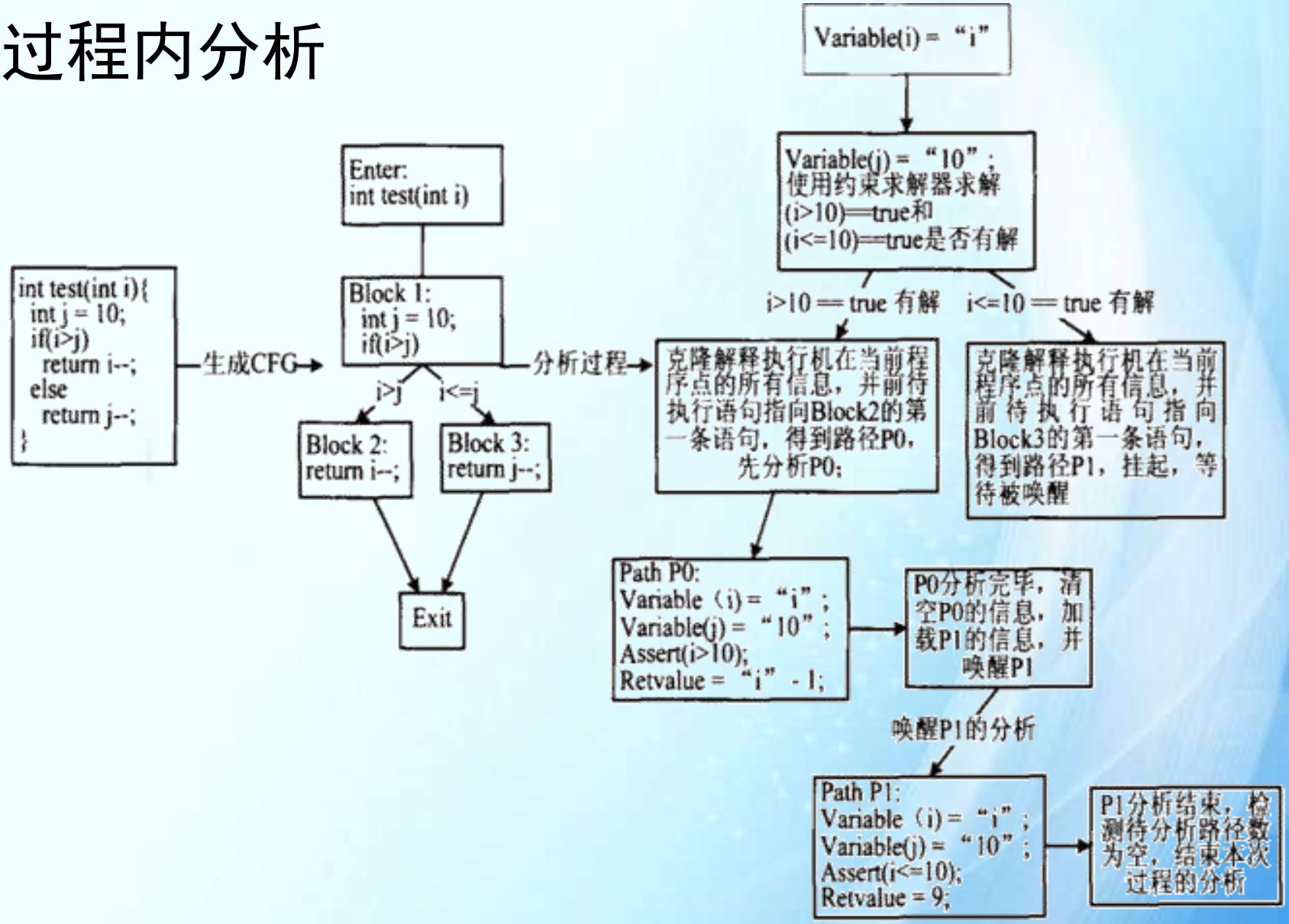
- 过程内分析
- 过程间分析

1.6 符号执行技术分类

- 过程内分析
  - 只对单个过程或函数的代码进行分析
  - 分析流程
    - 对过程或函数构建控制流图（CFG）
    - 符号执行过程从CFG入口点开始

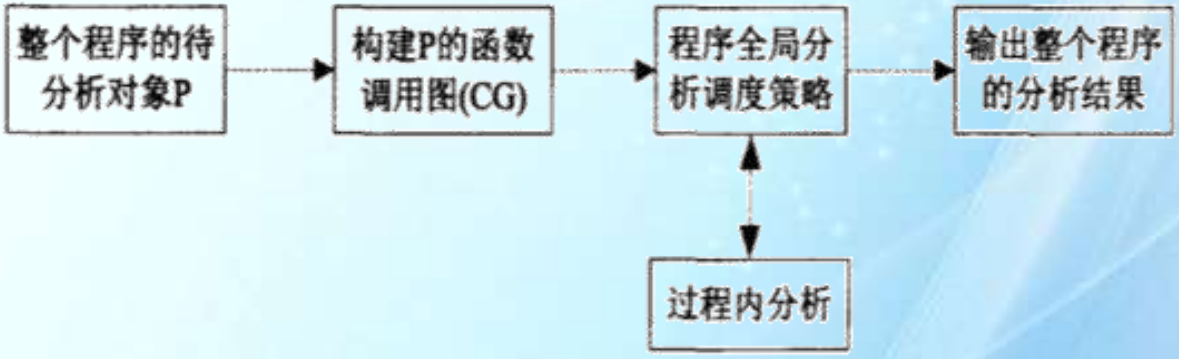
1.6 符号执行技术分类

- 过程内分析



1.6 符号执行技术分类

- 过程间分析
  - 对整个软件代码进行上下文敏感的分析
  - 上下文敏感，指在当前函数入口点，要考虑当前的函数间调用信息和环境信息等
  - 分析流程
    - 对整个程序构建函数调用图（CG）
    - 根据预设的分析策略对CG进行遍历
    - 进入函数后需按照过程内分析方法对函数进行分析



1.6 符号执行技术局限性

- 符号执行确实有用吗

软件漏洞利用经典工作给出的评价：“AEG: Automatic Exploit Generation” (T. Avgerinos et al.):

> [...] KLEE is a state-of-the-art forward symbolic execution engine, but in practice is limited to small programs such as /bin/ls.

1.6 符号执行技术局限性

- 符号执行成功案例

微软提出方案 -> SAGE: Whitebox Fuzzing for Security Testing

IMPACT OF SAGE

Since 2007, SAGE has discovered many security-related bugs in many large Microsoft applications, including image processors, media players, file decoders, and document parsers. Notably, SAGE found roughly *one-third* of all the bugs discovered by file fuzzing during the development of Microsoft's Windows 7. Because SAGE is typically run last, those bugs were missed by everything else, including static program analysis and blackbox fuzzing.

1.6 符号执行技术局限性

- 符号执行需要重点关注的问题
  - 路径爆炸
    - 条件分支太多，导致需要探索的潜在路径规模超出已有计算能力
  - 路径约束难以求解
    - 现有约束求解器能力有限，并不一定总能完成求解
  - 符号的内存模型如何选择
    - 需要依据使用场景和分析目标进行权衡
  - 执行过程中的环境交互
    - 用户输入交互、程序与系统和环境的交互（API）

1.7 局限性

- 路径爆炸问题
  - 符号执行技术在理论上面临着路径状态空间爆炸的问题
    - 每一个条件分支语句都可能会使当前的路径再分支出一条新的路径，造成路径的指数级增长
  - 通过多路径规约，或剪枝的方法可缓解路径爆炸问题，但效果有限
  - 路径爆炸示例

```
public void main(string s){
    bool a = contains(s, "Hello");
    bool b = contains(s, "World");
    bool c = contains(s, " at ");
    bool d = contains(s, "GeorgiaTech");
    if (a && b && c && d)
        throw new Exception("found it");
}
```

```
static bool contains(string s, string t){
  if (s == null || t == null)  return false;
  for (int i = 0; i < s.Length-t.Length+1; i++)
    if (containsAt(s, i, t))  return true;
  return false;
}

static bool containsAt(string s, int i, string t){
  for (int j = 0; j < t.Length; j++)
    if (t[j] != s[i + j]) return false;
  return true;
}
```

如果规定输入字符串长度为30，则可能的状态路径数量？

提纲

- 1、符号执行基础知识
- 2、动态符号执行技术
- 3、并行符号执行技术
- 4、选择符号执行技术

2 动态符号执行技术

- 定义
- 基本原理
- 流程及示例
- 关键问题及解决方案
- 工具介绍

2.1 定义

- 动态符号执行
  - **conc**rete + symb**olic** = **concolic**
    - Patrice Godefroid等在论文“DART: Directed Automated Random Testing”中首次提出
- 目标
  - 为了解决静态符号执行效率低，系统开销大的问题
  - 结合符号执行与具体执行的优势，在保证测试精度的前提下对程序执行树进行遍历

2.2 基本原理

- 原理
  - 动态符号执行以具体的数值作为输入执行程序代码，形成程序执行路径（Trace）
  - 在程序实际执行路径的基础上，用符号执行技术对路径进行分析，提取路径的输入数据约束表达式
  - 根据路径搜索策略（深度，广度）对部分分支的约束表达式进行翻转，求解翻转后的表达式并生成新的测试用例
  - 基于新的测试用例对新的执行路径进行遍历

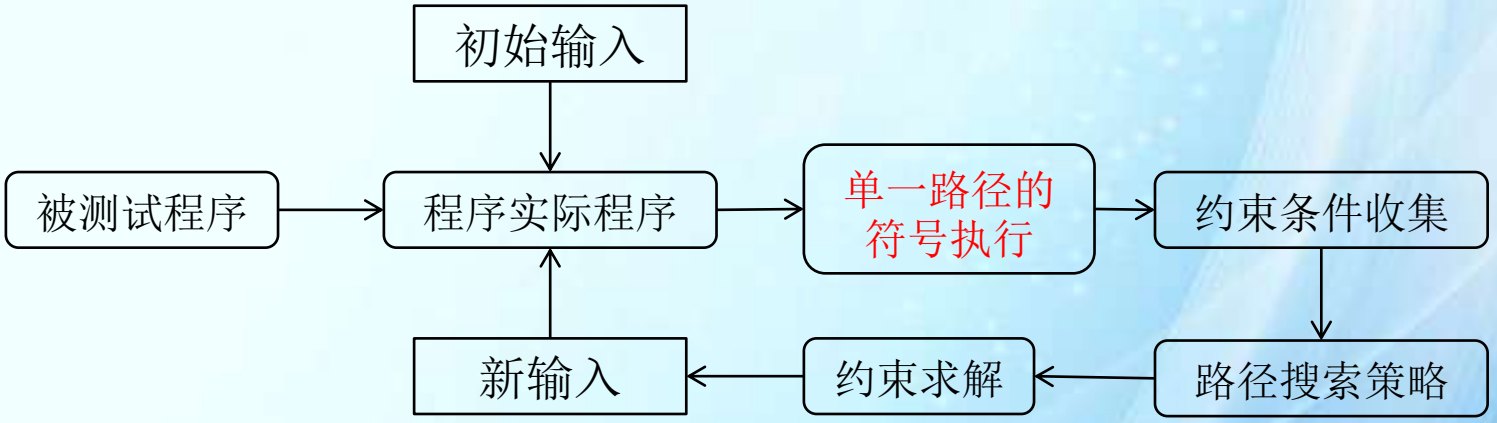
2.3 流程

1. 使用随机用例执行程序

2. 程序执行过程中收集路径条件

3. 使用路径搜索策略对路径条件中的约束进行取反，求解并生成新用例

4. 重复步骤1~3



2.3 示例

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

2.3 示例

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- Concrete input:
  $x = 10, y = 20$
  - $z = 20 \rightarrow x \neq z$

2.3 示例

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- Concrete input:
  $x = 10, y = 20$
- Initially:

$$-2^{31} \leq x \leq 2^{31} - 1$$
$$\wedge -2^{31} \leq y \leq 2^{31} - 1$$

2.3 示例

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- Concrete input:
  $x = 10, y = 20$
- After line 2:

$$-2^{31} \leq x \leq 2^{31} - 1$$
$$\wedge -2^{31} \leq y \leq 2^{31} - 1$$
$$\wedge z := y$$

2.3 示例

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- Concrete input:
  $x = 10, y = 20$
- After line 3:

$$
\begin{aligned}
& -2^{31} \leq x \leq 2^{31} - 1 \\
& \wedge -2^{31} \leq y \leq 2^{31} - 1 \\
& \wedge z := y \\
& \wedge c_1 := (x = z)
\end{aligned}
$$

2.3 示例

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- Concrete input:
  $x = 10, y = 20$
- After line 3:

$$
\begin{aligned}
& -2^{31} \leq x \leq 2^{31} - 1 \\
& \wedge -2^{31} \leq y \leq 2^{31} - 1 \\
& \wedge z := y \\
& \wedge c_1 := (x = z)
\end{aligned}
$$

- Path constraint: $\neg c_1$

2.3 示例

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- After line 3:

$$-2^{31} \leq x \leq 2^{31} - 1$$
$$\wedge\ -2^{31} \leq y \leq 2^{31} - 1$$
$$\wedge\ z := y$$
$$\wedge\ c_1 := (x = z)$$

- Old constraint: $\neg c_1$

2.3 示例

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- After line 3:

$$-2^{31} \leq x \leq 2^{31} - 1$$
$$\wedge -2^{31} \leq y \leq 2^{31} - 1$$
$$\wedge z := y$$
$$\wedge c_1 := (x = z)$$

- Old constraint: $\neg c_1$
- New constraint: c_1

2.3 示例

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- Logic formula:

$$
\begin{aligned}
& -2^{31} \leq x \leq 2^{31} - 1 \\
& \wedge -2^{31} \leq y \leq 2^{31} - 1 \\
& \wedge z := y \\
& \wedge c_1 := (x = z) \\
& \wedge c_1
\end{aligned}
$$

2.3 示例

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- Logic formula:

$$
\begin{aligned}
& -2^{31} \leq x \leq 2^{31} - 1 \\
& \wedge -2^{31} \leq y \leq 2^{31} - 1 \\
& \wedge z := y \\
& \wedge c_1 := (x = z) \\
& \wedge c_1
\end{aligned}
$$

- Satisfying assignment: $x = 0 \wedge y = 0$

2.3 示例

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- Concrete input: $x = 0, y = 0$

2.3 示例

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- Concrete input: $x = 0, y = 0$
- `c1 = x == z = 1`

2.3 示例

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- Concrete input: $x = 0, y = 0$
- c1 = x == z = 1
- t2 = x + 10 = 10

2.3 示例

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- After line 6:

$$
\begin{aligned}
& 2^{31} \leq x \leq 2^{31} - 1 \\
& \quad \wedge\, 2^{31} \leq y \leq 2^{31} - 1 \\
& \quad \wedge\, z := y \\
& \quad \wedge\, c_1 := (x = z) \\
& \quad \wedge\, t_2 := x + 10 \\
& \quad \wedge\, c_2 := y = t_2
\end{aligned}
$$

2.3 示例

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- After line 6:

$$2^{31} \leq x \leq 2^{31} - 1$$
$$\wedge\, 2^{31} \leq y \leq 2^{31} - 1$$
$$\wedge\, z := y$$
$$\wedge\, c_1 := (x = z)$$
$$\wedge\, t_2 := x + 10$$
$$\wedge\, c_2 := y = t_2$$

- Path constraint: $c_1 \wedge \neg c_2$

2.3 示例

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- New constraint: $c_1 \wedge c_2$

2.3 示例

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- New constraint: $c_1 \wedge c_2$
- Logic formula:

$$
\begin{aligned}
& 2^{31} \leq x \leq 2^{31} - 1 \\
& \wedge\, 2^{31} \leq y \leq 2^{31} - 1 \\
& \wedge\, z := y \\
& \wedge\, c_1 := (x = z) \\
& \wedge\, t_2 := x + 10 \\
& \wedge\, c_2 := y = t_2 \\
& \wedge\, c_1 \wedge c_2
\end{aligned}
$$

2.3 示例

```
1  int f(int x, int y) {
2    int z = y;
3    bool c1 = x == z;
4    if (c1) {
5      int t2 = x + 10;
6      bool c2 = y == t2;
7      if (c2) {
8        abort();
9      }
10   }
11 }
```

- New constraint: $c_1 \wedge c_2$
- Logic formula:

$$
\begin{aligned}
& 2^{31} \leq x \leq 2^{31} - 1 \\
& \wedge\, 2^{31} \leq y \leq 2^{31} - 1 \\
& \wedge\, z := y \\
& \wedge\, c_1 := (x = z) \\
& \wedge\, t_2 := x + 10 \\
& \wedge\, c_2 := y = t_2 \\
& \wedge\, c_1 \wedge c_2
\end{aligned}
$$

- Unsatisfiable! (The error is unreachable)

2.4 关键问题

- 外部函数调用
  - 外部函数的内部细节对调用程序来说是不可见的，无法使用符号执行引擎对其内部控制流进行跟踪
  - 如何避免外部函数调用造成的路径分析终止？
- 缓解方案
  - 具体值替换
    - 使用外部函数的具体执行的返回值带入符号分析过程中，以此继续对该路径进行分析
    - DART,CUTE等工具使用的是该方法
  - 外部函数建模
    - 对外部函数的行为进行建模，辅助符号执行工具“获取”外部函数内的路径约束
    - KLEE为POSIX中40多个系统函数进行建模

2.4 关键问题

- 循环问题
  - 循环造成过于复杂的路径约束条件，存在大量冗余条件

```
1   void main(int x) { // x is an input
2     int c = 0, p = 0;
3     while (1) {
4       if (x<=0) break;
5       if (c == 50) abort1(); /* error1 */
6       c = c + 1;
7       p = p + c;
8       x = x - 1;
9     }
10    if (c == 30) abort2();  /* error2 */
11  }
```

  - 当x的初始值为10时，实际执行路径对应的约束

$$pc_0 = (x_0 > 0) \wedge (x_0 - 1 > 0) \wedge \cdots \wedge (x_0 - 9) > 0 \wedge (x_0 - 10 \leq 0)$$

2.4 关键问题

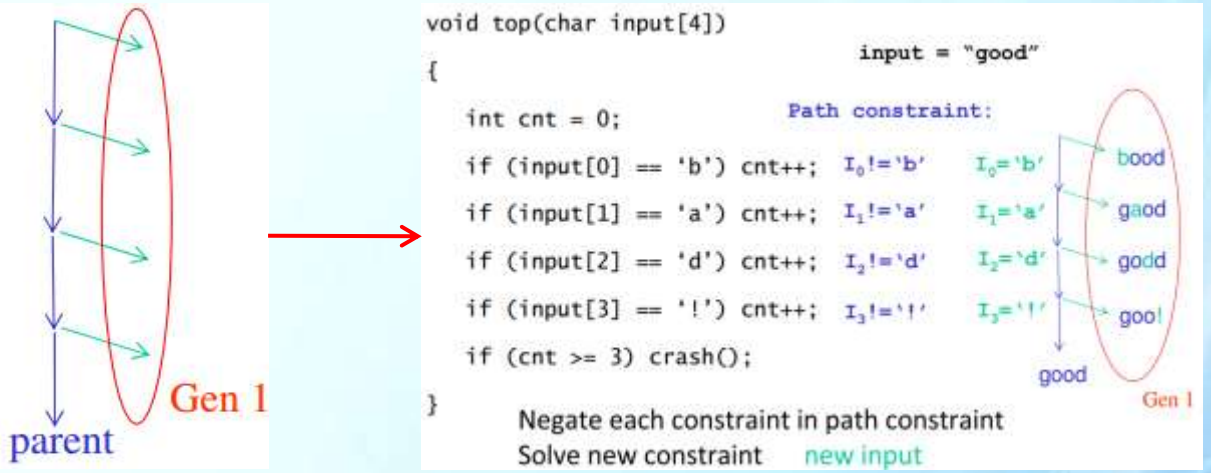
- 缓解方案
  - 约束优化：对由循环造成的冗余约束条件进行规约优化，相当于剪枝处理
    - 使用该策略的工具：SAGE
  - 限制循环执行次数：限制循环语句的展开次数，或按照固定次数对循环语句进行展开
    - 使用该策略的工具：IntScope等
  - 循环摘要：对循环片段建立归纳变量与控制执行流程退出循环的变量之间的函数关系，避免对循环内部逻辑的展开分析，直接对循环内部所有路径进行遍历
    - 使用该策略的工具：LESE等

2.5 工具介绍

- 开源工具
  - **KLEE**
    - Cadar, Dunbar, Engler， Usenix 2008
  - DART
    - Godefroid 、 Sen, PLDI 2005
  - EXE
    - Cadar, Ganesh, Pawlowski, Dill, Engler, CCS 2006
- 商用工具
  - **SAGE**
    - Godefroid 、 NDSS 2008

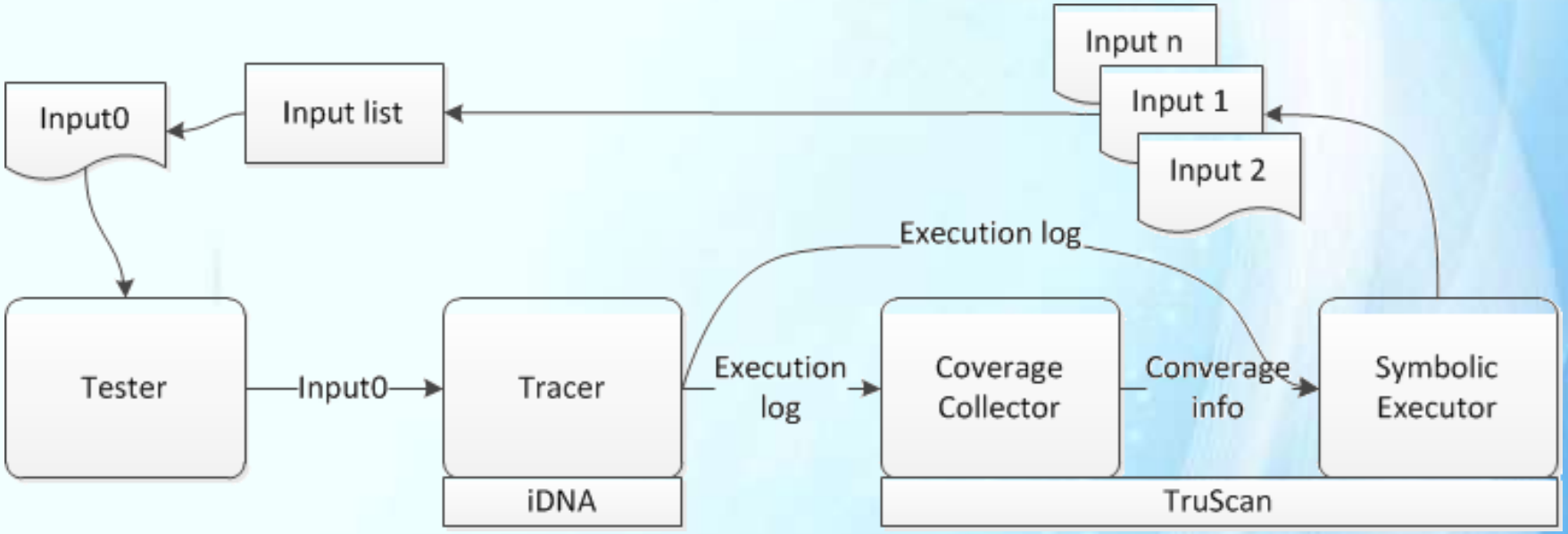
2.5 工具介绍

- SAGE
  - 微软内部测试工具,基于x86指令集
  - 主要对文件解析类的软件进行测试
    - 例如，Word，PowerPoint
  - 分代路径搜索算法
  - 基于路径执行记录的离线分析



2.5 工具介绍

- SAGE
  - 基本架构



2.5 工具介绍

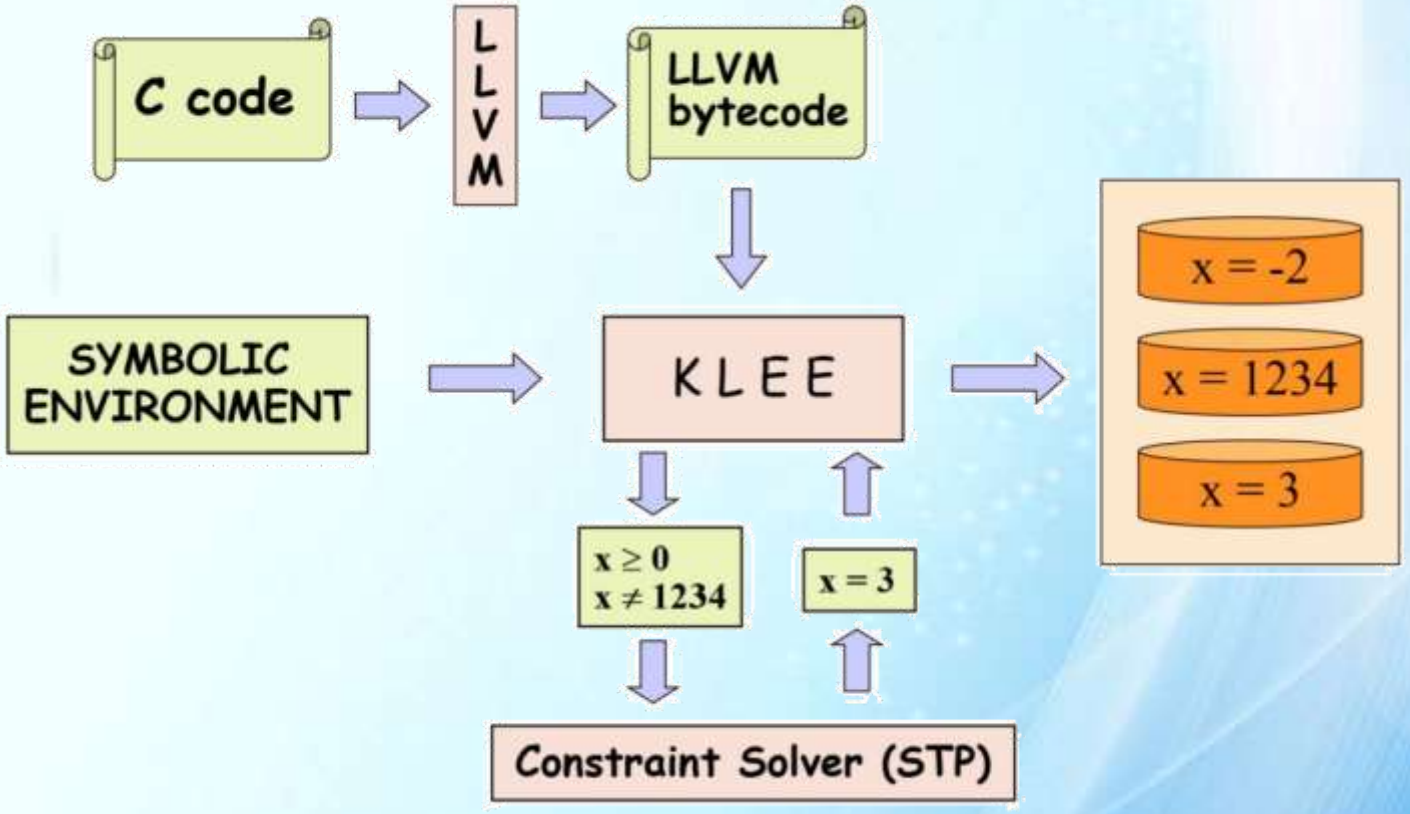
- KLEE

  - 基于LLVM
    - LLVM 是 Illinois 大学发起的一个开源项目，和JVM 以及 .net Runtime这样的虚拟机不同，这个虚拟系统提供了一套中间代码和编译基础设施，并围绕这些设施提供了一套全新的编译策略（使得优化能够在编译、连接、运行环境执行过程中，以及安装之后以有效的方式进行）
    - LLVM荣获2012年ACM软件系统奖
  - 对文件系统、网络通信、系统函数等进行建模
  - 路径搜索策略，约束求解等都进行了优化

2.5 工具介绍

- KLEE
  - 基本架构

KLEE is one of the best frameworks for SE
http://klee.github.io/



2.5 工具介绍

- Angr: a python framework for analyzing binaries

http://angr.io

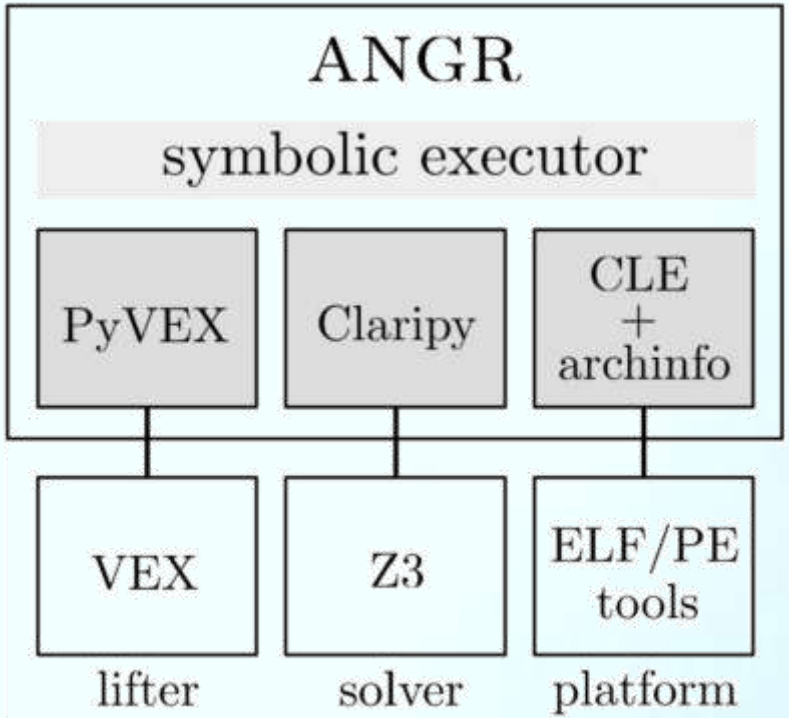
主要功能

- CFG 重构
- 静态符号执行
- 静态污点分析、切片等
- 二进制程序分析

angr 是美国DARPA Cyber Grand Challenge 竞赛中参赛队MechaPhish研发的工具
Computer Security Lab @ UC Santa Barbara

2.5 工具介绍

- Angr架构



- PyVEX: lifter based on VEX (valgrind) for reasoning on binary code. Makes angr architecture-independent!
- Claripy: solver abstraction. Support for Microsoft Z3 solver.
- CLE/archinfo: ELF/PE loaders. It handles platform details.

2.5 工具介绍

- 基于Angr的符号执行

一、探索目标设置

1) [start target] 符号执行开始的位置

2) [find target] 符号执行结束的位置

3) [avoid targets] 哪些路径要绕过（主动剪枝）

二、定义符号输入

寄存器、内存、栈

2.5 工具介绍

- 基于Angr的符号执行

```
int main(int argc, char* argv[]) {
  char buffer[9];

  printf("Enter the password: ");
  scanf("%8s", buffer);

  for (int i=0; i<LEN_USERDEF; ++i) {
    buffer[i] = complex_function(buffer[i], i);
  }

  if (strcmp(buffer, USERDEF)) {
    printf("Try again.\n");
  } else {
    printf("Good Job.\n");
  }
}
```

2.5 工具介绍

- 基于Angr的符号执行

逆向分析定位我们希望执行到的位置：(0x804867a)

2.5 工具介绍

- 基于Angr的符号执行

```
path_to_binary = ???
project = angr.Project(path_to_binary)

initial_state = project.factory.entry_state()

simulation = project.factory.simgr(initial_state)

print_good_address = ???
simulation.explore(find=print_good_address)

if simulation.found:
  solution_state = simulation.found[0]
  print solution_state.posix.dumps(sys.stdin.fileno())
else:
  raise Exception('Could not find the solution')
```

← 程序文件路径

← 创建项目

← 从main()开始符号执行

← 希望符号执行找到的位置

← 开始符号执行

← 符号执行结束并找到目标

← 获取第一个找到的目标

← 获取对应的输入

2.6 小结

- 小结
  - 研究人员将符号执行与具体执行的优势进行结合，提出了动态符号执行
    - 解决静态符号执行效率低，系统开销大的问题，在保证一定测试精度的前提下快速遍历执行树
  - 动态符号执行基本思想：在单路径上进行符号执行和具体执行分析，并使用路径搜索算法指导符号执行引擎对程序执行树进行遍历
  - 动态符号执行应用：
    - 包括微软等厂商正尝试使用动态符号执行替代模糊测试技术
  - 局限性：外部函数调用，程序循环等问题，影响分析的效率和准确性

提纲

- 1、符号执行基础知识
- 2、动态符号执行技术
- 3、并行符号执行技术
- 4、选择符号执行技术

3 并行符号执行技术

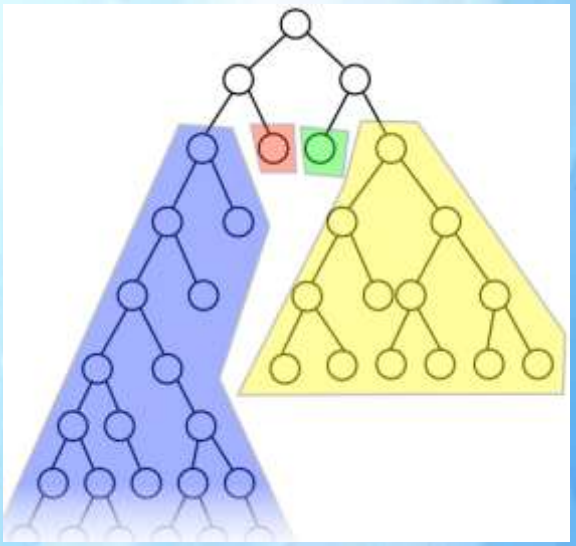
- 定义
- 关键问题
- 工具介绍

3.1 定义

- 传统符号执行
  - 路径爆炸
    - 单点执行模式下，时间消耗长，内存消耗易超出负荷
    - 对于逻辑复杂的程序无法有效测试
- 并行符号执行
  - 可看成符号执行技术与并行计算技术的融合
  - 设计初衷
    - 通过计算集群可无限扩展的内存空间和CPU资源缓解符号执行过程中的路径爆炸

3.2 关键问题

- 分布式环境下的路径搜索策略
  - 如何避免各节点重复遍历部分路径
- 负载均衡问题
  - 如何划分任务集合
    - 程序的执行树并非平衡二叉树，
    - 开始分析时执行树的整体结构未知，
    - 动态调整 or 静态划分？

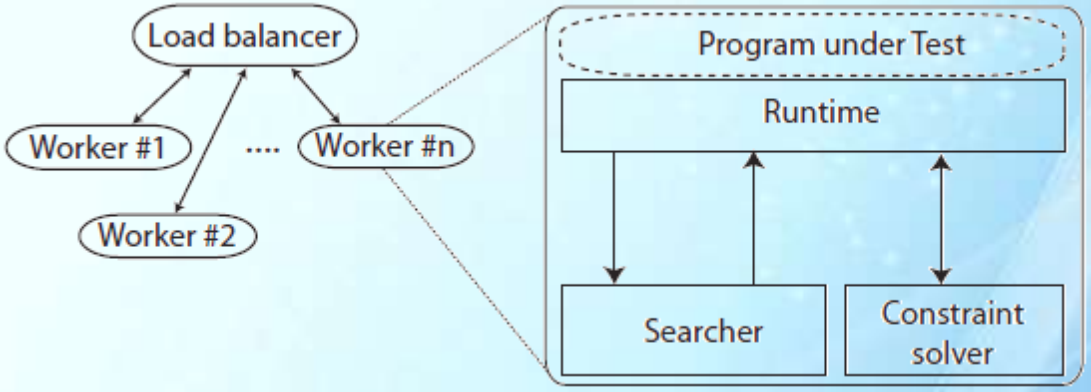


3.3 工具介绍

- Cloud9
- SCORE
- SAGEN

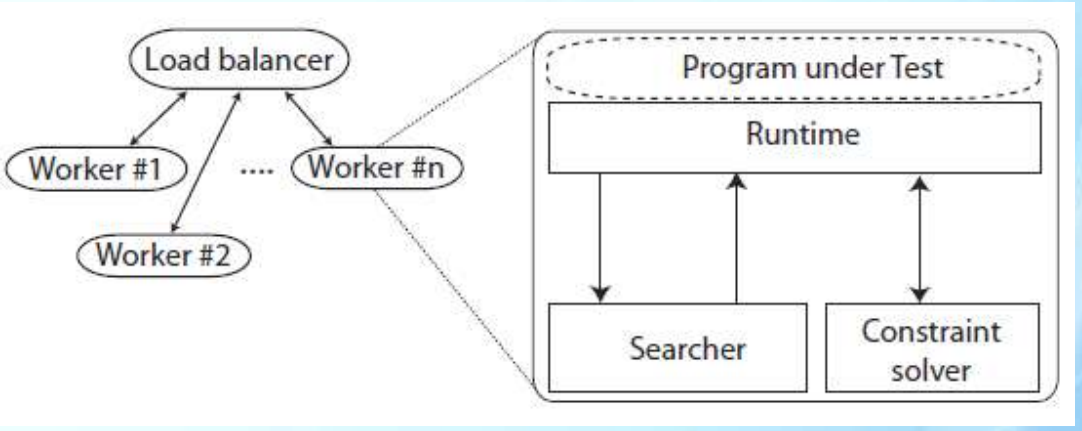
3.3 Cloud9

- 系统架构设计
  - 包含Load balancer和Worker两种节点
  - Worker
    - 各Worker节点独立的探索程序执行树的中的一个子树
      - 各Worker节点上部署了一个基于KLEE实现的符号执行引擎，包括Runtime、Constraint solver、Searcher三部分
    - 符号执行每当遇到条件分支时并不会直接探索，而是记录当前的程序状态、程序当前路径上的条件分支等信息，并用这些信息初始化一个状态结构，使用该结构可以将程序恢复到执行当前分支时的状态

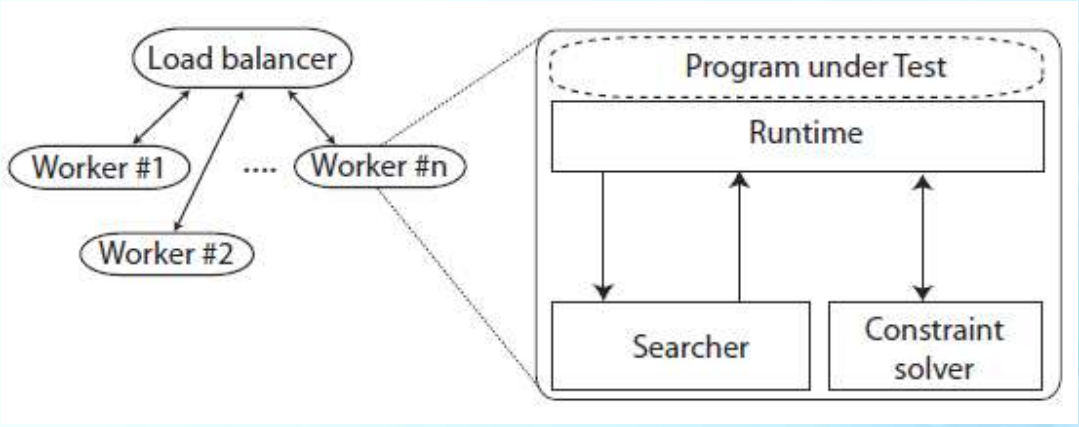


3.3 Cloud9

- 系统架构设计
  - 包含Load balancer和Worker两种节点
  - Worker
    - Runtime将生成的状态放入worker的工作队列
    - Searcher是Cloud9的路径搜索模块，用来决定下一个需要搜索的路径
    - 当从Searcher模块得到预期探索路径的约束表达式后，Runtime模块就会调用Constraint solver模块对表达式进行求解，通过求解结果初步判断该路径是否可达。如果有解则Runtime使用新用例遍历对应的路径，否则继续向Searcher发出请求。



- 系统架构设计
  - 包含Load balancer和Worker两种节点
  - Load balancer
    - 负载均衡模块
    - 监控各Worker节点的状态并动态调度



3.3 Cloud9

- 系统执行流程
  - Cloud9首先启动Load balancer，然后启动worker节点簇
  - 当第一个worker节点W1启动
    - 从LB获取种子任务，并开始对程序执行树进行探索
  - 当第二个节点W2启动
    - LB会安排其去分担W1节点的任务，即控制W1将部分未探索的子任务发送到W2节点
  - 以后每当有新的节点加入到worker节点簇
    - LB都会主动平衡系统内部节点的负载。worker节点簇也会定时将其节点状态、负载情况以及代码覆盖情况进行编码并发送到LB
  - LB能够主动进行均衡处理
    - 根据各节点的负载状况，控制节点进行任务转送

3.3 Cloud9

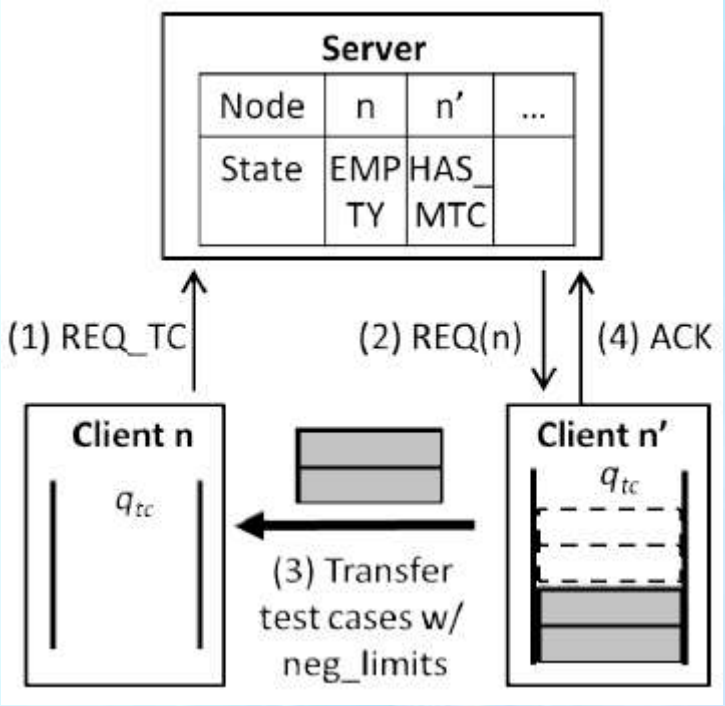
- 关键问题解决方案
  - 负载均衡算法
    - Worker主动向LB发送当前的运行状态及任务队列的状态
    - LB根据各节点的状态动态平衡负载
      - 只有当最高负载的节点的状态超过阈值时，LB才会认为系统处在不平衡状态，并作出调整
  - 路径搜索算法
    - Cloud9使用了多种路径搜索算法
    - 以测试用户的最终目的为导向选择路径算法，各节点可以使用不同的搜索策略算法，但都向最终目标靠拢
    - 可能会产生冗余用例

3.3 SCORE

- Cloud9
- SCORE
- SAGEN

3.3 SCORE

- **系统架构设计**
  - 系统中包含两种类型的节点，server和client
  - 系统中有七种类型的协议包在server和client节点间传输来实现各节点的协同工作
  - Server节点维护各节点的状态
  - client节点维护自己的工作队列，并和server交互获取任务



3.3 SCORE

- **系统执行流程**
  - 系统选择任意client节点作为起始节点，执行初始用例，生成测试用例集合
  - 新加入的client节点n向server发送任务申请报文，server节点查询内部维护的client节点任务状态表，从中选择最合适的节点m
  - m将部分任务传输至n的任务队列，n开始任务
  - 重复上面的步骤知道各节点的任务队列都为空

Input:
startup: a flag to indicate whether a current node n is a startup node or not.
Output:
TC: a set of test cases generated at a current node n (i.e., tcs of line 24)

```
1  DstrConcolic(startup) {
2  q_tcp = ∅; // a queue containing (tc, neg_limit)s
3  TC = ∅; // a set of generated test cases
4  if startup then
5      tc = random value; // initial test case
6      Add (tc, 1) to q_tcp;
7  else
8      Send a request for test cases to another node n';
9      Receive (tc, neg_limit)s from n';
10     Add (tc, neg_limit)s to q_tcp;
11 end
12 while true do
13     while | q_tcp |> 0 do
14         Remove (tc, neg_limit) from q_tcp;
15         // Execute target program on tc
16         path = an execution path on tc;
17         // Obtain a symbolic path formula φ
18         φ = a symbolic path formula of path (i.e.,
           c1 ∧ c2 ∧ ... ∧ c|φ|) ;
19         j =| φ |;
20         while j >= neg_limit do
21             // Generate ψ for the next input values
22             ψ = c1 ∧ ... ∧ cj−1 ∧ ¬cj ;
23             // Select the next input values
24             tc = Solve(ψ);
25             if tc is not NULL then
26                 Add (tc, j + 1) to q_tcp;
27                 TC = TC ∪ {tc};
28             end
29             j = j − 1;
30         end
31     end
32     if there is a test case in another node n'' then
33         Send a request for test cases to another node
           n'';
34         Receive (tc, neg_limit)s from n'';
35         Add (tc, neg_limit)s to q_tcp;
36     else
37         Halt; // no test cases exist in all nodes
38     end
39 end
40 }
```

Algorithm 1: Distributed concolic algorithm

3.3 SCORE

- 关键问题解决方案
  - 负载均衡算法
    - Client主动向Server发送当前的运行状态及工作队列的状态
    - Server维护各Client节点的工作队列状态表
    - Server不会主动调节负载，只有当Client节点工作队列为空时才会主动向Server节点申请任务，Server此时才会参与负载调节
  - 路径搜索算法
    - 基于SAGE的分代搜索算法进行的改进
    - 理论证明该算法可以保证在并行环境下不会生成冗余用例

3.3 SCORE

- 分代搜索算法（Generation search）

```
1  Search(inputSeed){
2    inputSeed.bound = 0;
3    workList = {inputSeed};
4    Run&Check(inputSeed);
5    while (workList not empty) {//new children
6      input = PickFirstItem(workList);
7      childInputs = ExpandExecution(input);
8      while (childInputs not empty) {
9        newInput = PickOneItem(childInputs);
10       Run&Check(newInput);
11       Score(newInput);
12       workList = workList + newInput;
13     }
14   }
15 }
```

```
1  ExpandExecution(input) {
2    childInputs = {};
3    // symbolically execute (program,input)
4    PC = ComputePathConstraint(input);
5    for (j=input.bound; j < |PC|; j++) {
6      if((PC[0..(j-1)] and not(PC[j]))
                    has a solution I){
7        newInput = input + I;
8        newInput.bound = j;
9        childInputs = childInputs + newInput;
10   }
11   return childInputs;
12 }
```

```
void top(char input[4])
{
  int cnt = 0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 3) crash();
}
```

input = "good"

Path constraint:

$I_0 != 'b'$ $I_0 = 'b'$

$I_1 != 'a'$ $I_1 = 'a'$

$I_2 != 'd'$ $I_2 = 'd'$

$I_3 != '!'$ $I_3 = '!'$

bood

gaod

godd

goo!

good

Gen 1

Negate each constraint in path constraint

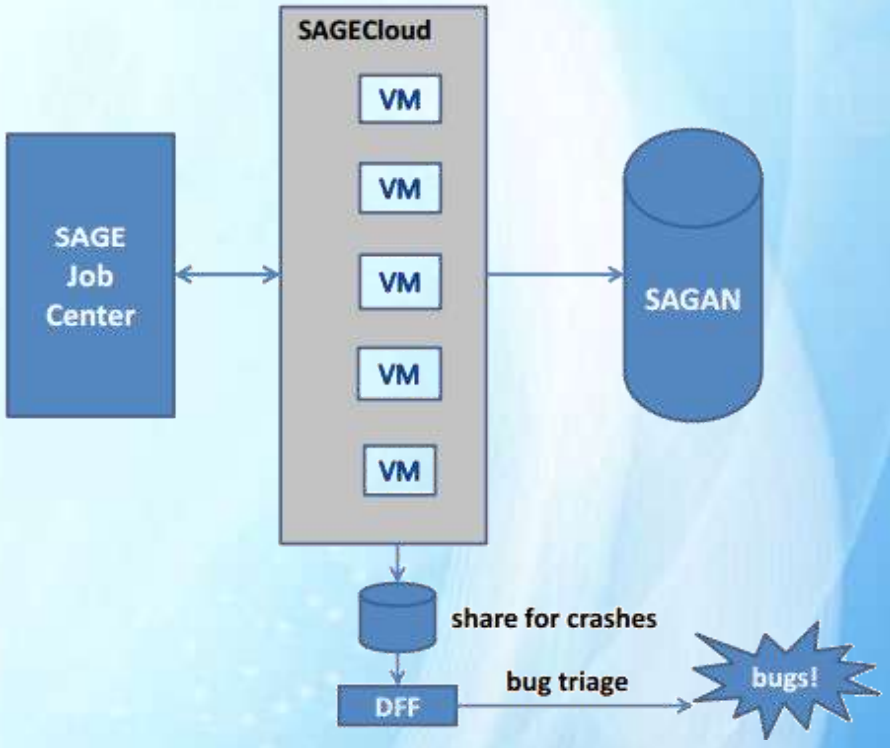
Solve new constraint new input

3.3 SCORE

- Cloud9
- SCORE
- SAGEN

3.3 SAGEN

- **系统架构设计**
  - 整个系统包含三个部分
    - JobCenter（中心控制模块）
    - SAGAN（系统监控模块）
    - SAGECloud（SAGE节点集群）
  - 各模块功能
    - SAGECloud：SAGE集群，集群中的每个虚拟机（VM）中可以部署一个SAGE系统
    - SAGEN：系统监控模块，协同SQLserver数据库为各节点维护详细的日志信息，辅助异常节点的快速恢复
    - JobCenter：任务分配模块，实时监测各节点的任务执行状态，并分配新任务



3.3 Cloud9

- 关键技术
  - 不同点
    - SAGEN并不能算普通意义上的并行系统，不同于前面的介绍的系统，SAGEN在各节点上运行不同的SAGE任务，这属于多任务并行，而不是相同任务的并行，所以不存在用例冗余的问题
  - 系统错误修复
    - 在分布式环境下，节点出现异常的情况很普遍
    - 为了解决这个问题，SAGEN通过详细的日志信息记录维护帮助节点快速恢复
  - 漏洞唯一性问题
    - 实际测试中大量的测试用例可能导致一个相同的异常
    - 为了解决这个问题，SAGEN为每个已经发现的crash都建立崩溃发生时的堆栈哈希及时间戳，通过比对哈希值就可以排除大量冗余的用例，节省存储空间

3.4 小结

- 小结
  - 通过并行技术提升动态符号执行效率
    - 通过计算集群可无限扩展的内存空间以及CPU资源在多节点中同步处理不同符号执行任务，改变了传统符号执行序列化处理任务的工作模式
  - 高效的路径搜索算法和负载均衡算法是并行符号执行的核心
    - 针对三个代表性工作介绍了设计原则和实现方案
  - 执行效率显著提升，但并没有从根本上解决路径爆炸问题

提纲

- 1、符号执行基础知识
- 2、动态符号执行技术
- 3、并行符号执行技术
- 4、选择符号执行技术

4 选择符号执行技术

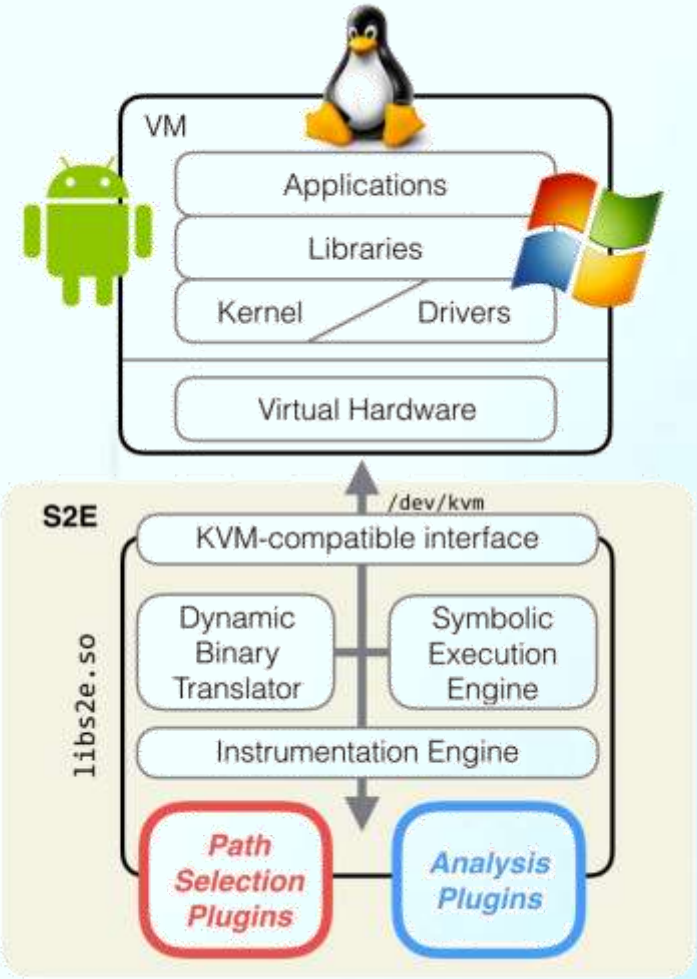
- 定义
- 基本原理
- 执行流程

4.1 定义

- 选择符号执行定义
  - 只对程序路径中的一段区域，或者只对部分路径进行符号执行分析就称之为选择符号执行
    - 传统符号执行可看做是全局符号执行
- 设计初衷
  - 各类路径搜索算法和并行技术难以满足需求
    - 效率提升仍然不足实现全路径遍历
  - 一个程序的执行树中，并不是每条路径都有价值，可能只有部分子树与最终分析目标相一致
    - 瑞士洛桑理工大学的Vitaly Chipounov等最早提出

4.1 定义

- 选择符号执行代表性工作 – S2E



S2E: complex but very powerful full stack analysis

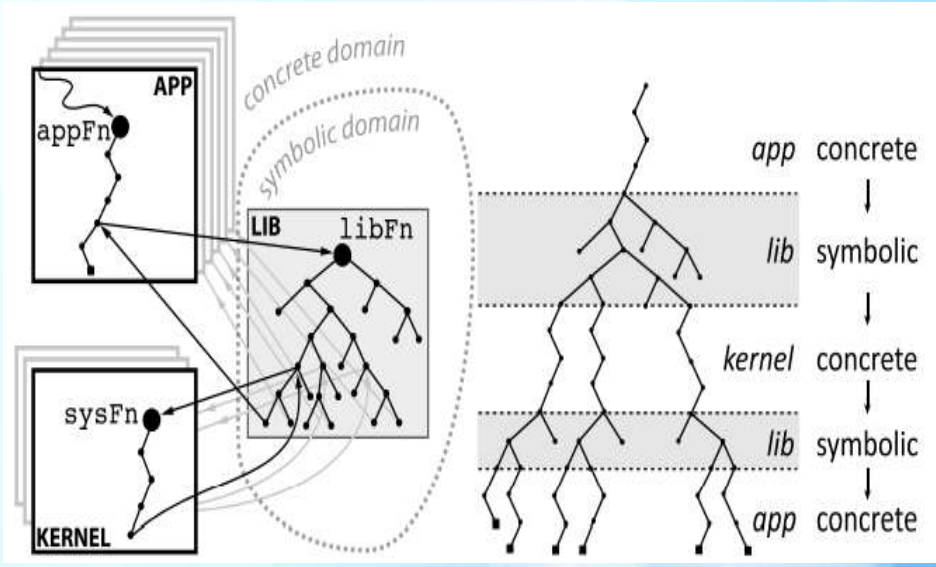
https://s2e.systems/

4.2 基本思想

- 基本思想
  - 当测试程序执行到感兴趣的程序段时，对代码进行符号执行分析
    - 当程序执行到外部调用函数时，S2E会对代码进行单路径具体执行
  - S2E对程序执行树的探索是弹性的，其不以全路径遍历为目的，而是以感兴趣的路径片段为目标
    - 将需要分析的执行树规模缩减到最小，这样做在某种程度来说使用更加巧妙的方式缓解了路径爆炸问题

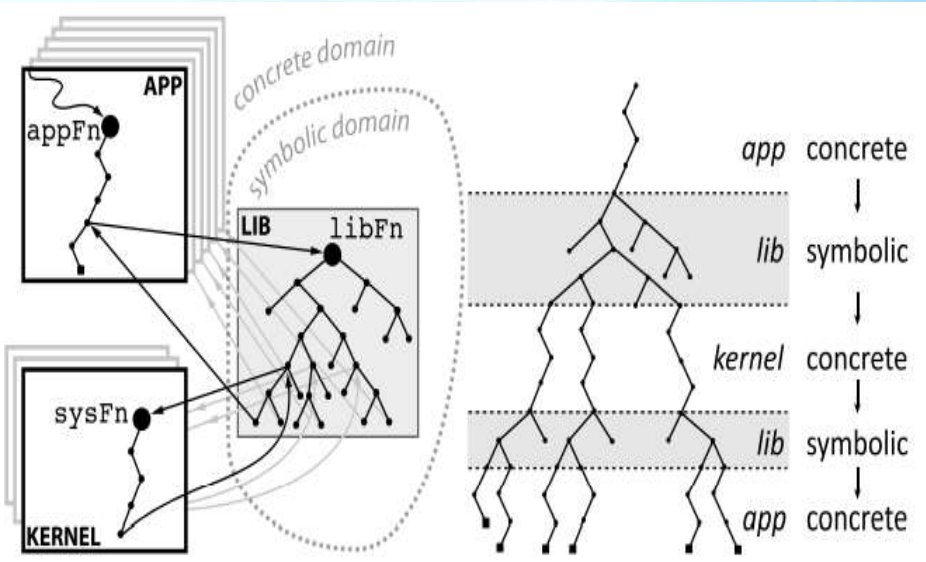
4.3 执行流程

- **总体流程**
  - 程序app由单路径具体执行模式（concrete）开始自上而下执行
  - 进入分析所关心的lib库函数时，S2E使其进入多路径执行模式（symbolic）
  - 当程序离开lib库函数进入内核kernel代码时，S2E再次切换到单路径模式
  - 如此反复，其核心只有一个，即对所关心的代码段、函数或程序片段，使用符号化执行进行分析，对不关心的代码进行具体执行



4.3 执行流程

- 关键流程
  - 具体执行至符号执行的转换
    - appFn调用libFn时就需要进行这样的模式切换
    - 将libFn函数的输入参数从具体值转换成符号值(libFn(10))转换成libFn(a)
    - 对libFn函数同时进行符号执行和具体执行
  - 符号执行到具体执行的转换
    - 惰性实例化：S2E按照需求对符号变量进行实例化，只有在具体执行片段需要访问变量x的值时才进行实例化操作
    - 根据符号约束求解得到符号对应的具体值



4.4 小结

- 小结
  - 通过减小符号分析区间，对执行路径的局部目标段进行分析，缓解路径爆炸问题
  - 选择符号执行的基本思路
    - 当测试程序执行到感兴趣的程序段时，对代码进行符号执行分析
    - 当程序执行到外部调用函数时，只对程序进行单路径具体执行

End
&
Thanks