《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
2024年4月27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？

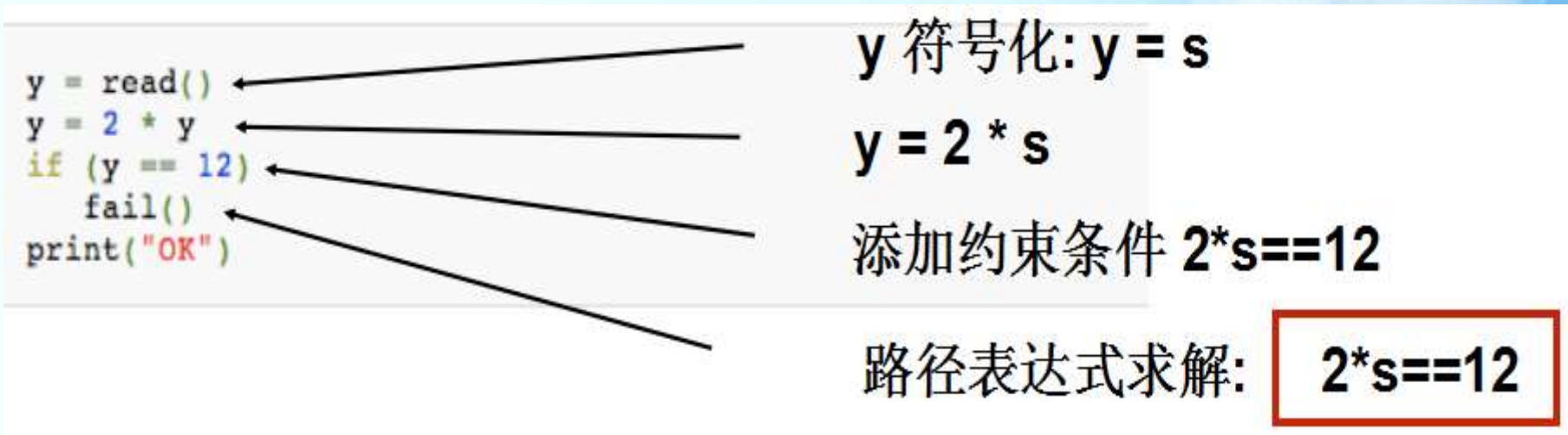
✓ 怎样提高测试的效率？

1.2 符号执行

- 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
  - 与常规执行的不同之处在于，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)
  - 基本步骤
    - 使用符号变量替换具体值作为程序输入
    - 程序执行过程中提取条件约束生成路径表达式
    - 通过分析路径表达式实现对程序执行路径安全性的评估
      - 即该路径上是否蕴含着漏洞：
        - » malloc(symA) & memcpy(dst, src, symB)
        - » var = symA+symB
        - » if(symA < symB)

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，相对随机测试，在减少测试样本数量的同时有效提高了路径覆盖率
    - 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。
  - “执行树”的规模是有限的
    - 循环、递归等造成执行树规模无穷扩展
  - 对所有IF条件语句都可进行符号执行处理
    - 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

- 程序语言定义
  - 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

- 符号输入与程序语义
  - 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

TCA

- 符号输入与程序语义
  - 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

- 符号输入与程序语义
  - 程序语义的变化
    - **程序执行状态**
      - 具体执行时，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。
      - King为程序状态新添加了**变量PC**，“路径条件表达式”，通过PC就可以唯一的确定一条执行路径，以此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，在各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：

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

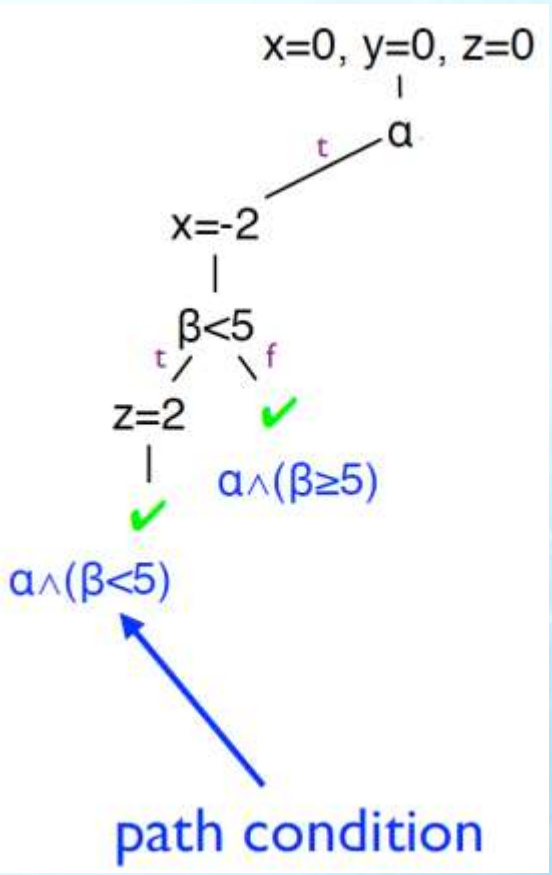
1.2 符号执行

- 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 ∩ *not* 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
    - 例如：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

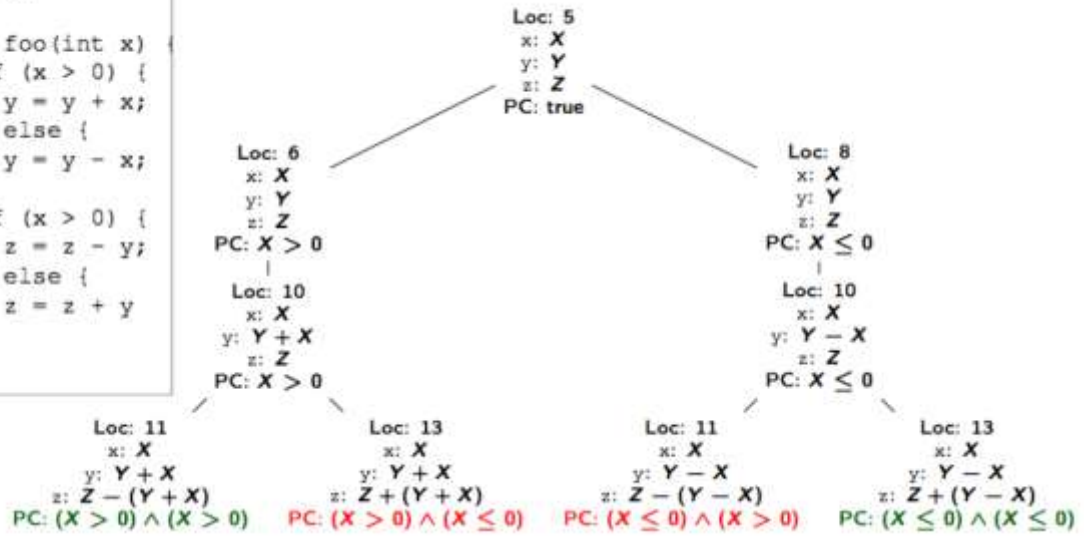
1.2 符号执行

- 路径表达式求解的应用
  - 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

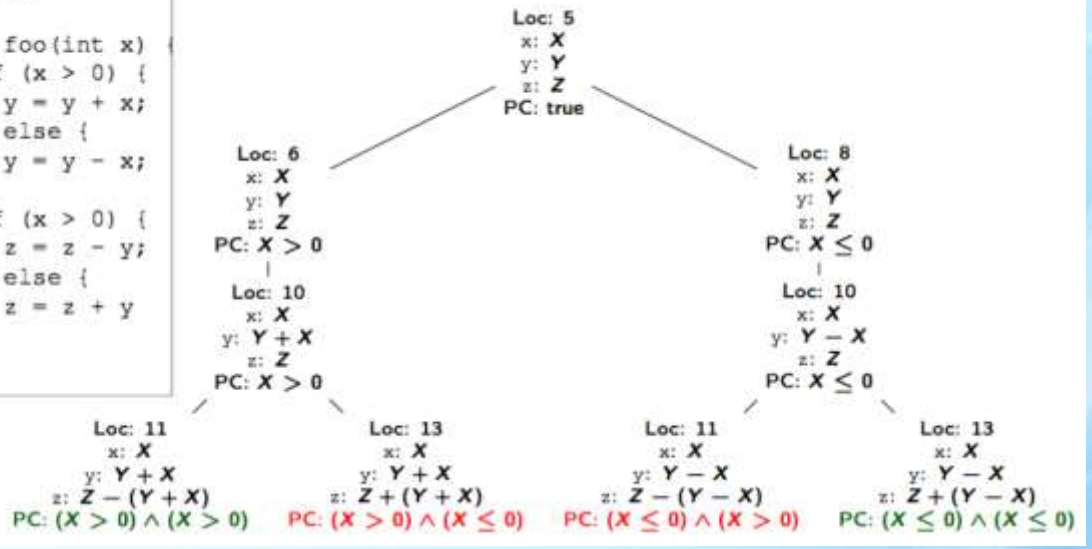
```
1  int y;
2  int z;
3  ...
4  int foo(int x) {
5    if (x > 0) {
6      y = y + x;
7    } else {
8      y = y - x;
9    }
10   if (x > 0) {
11     z = z - y;
12   } else {
13     z = z + y
14   }
15 }
```



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，如下图所示。

```
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

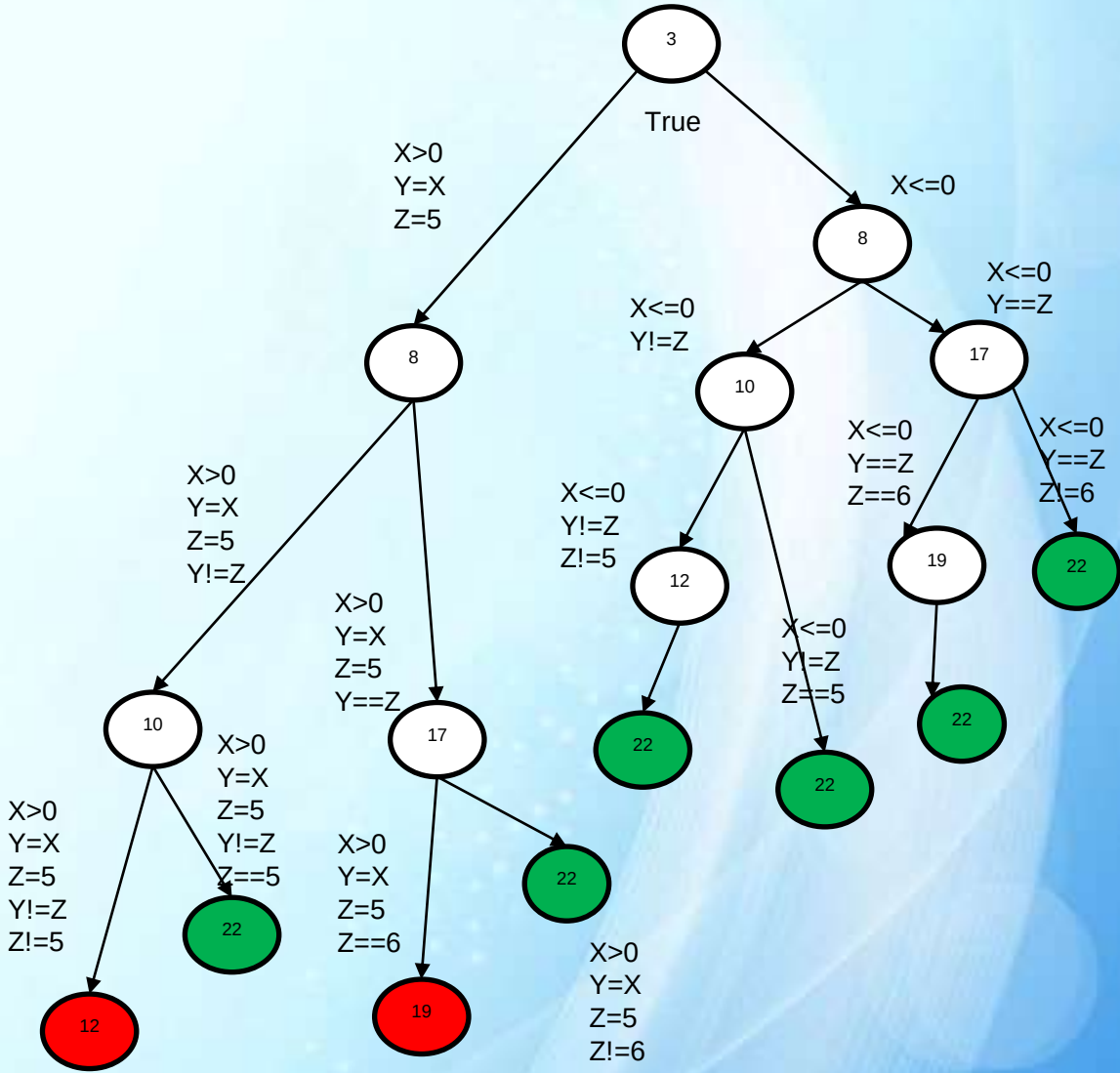
- 执行树的性质

```
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. }
```

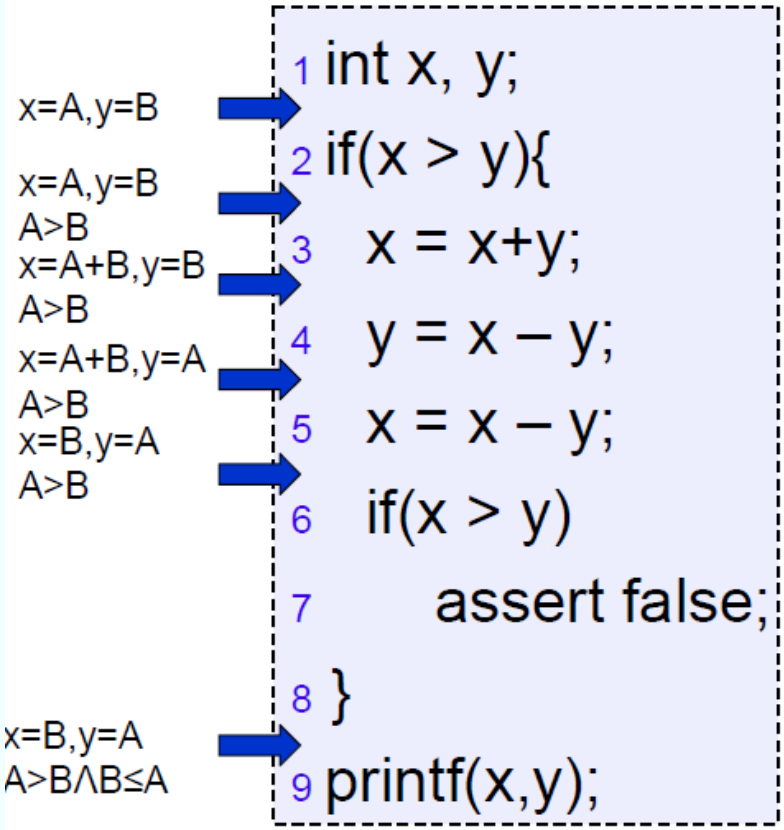


1.4 流程及示例

- 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



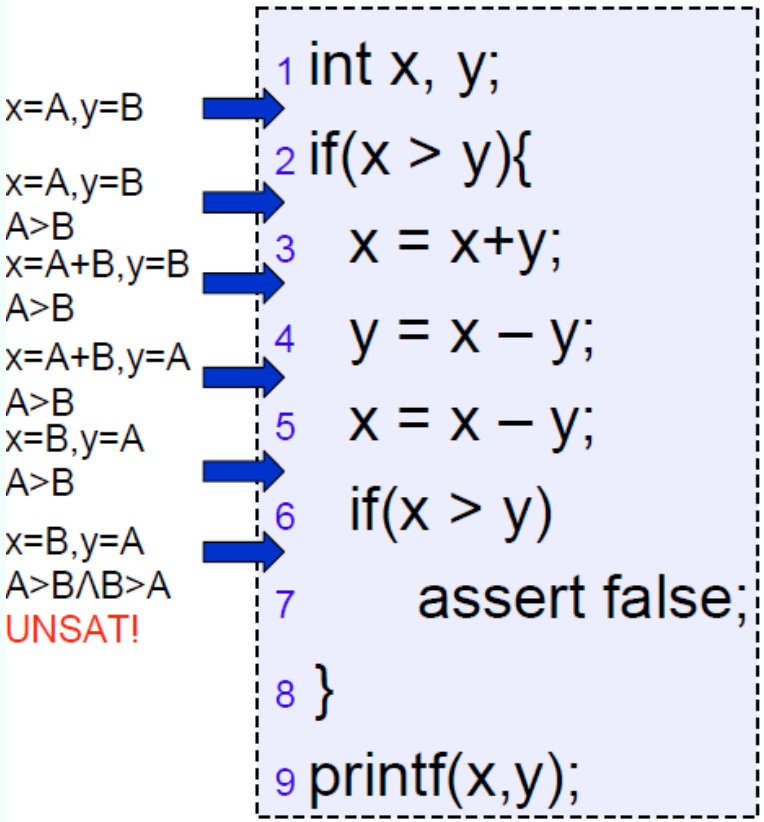
inputs that cover ***else*** branch at stmt. 2:
x = 3 y = 4

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表示论域，即变量可能取值的有限集合
  - C表示约束条件集合
- 约束求解
  - 约束求解就是在变量的论域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
    - 地图着色问题
    - 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、$x_i = 1，2，...，8$
        - » 2、不同列：$x_i \neq x_j$
        - » 3、不同对角线：$|i-j| \neq |x_i - x_j|$
    - 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，更新和维护仍然活跃
  - STP可作为独立的求解工具，也可作为其他工具的求解组件，其提供了C，python，SMT-LIB接口
  - 因为STP优越的性能，被众多分析工具所使用
    - EXE : Bug Finder by Dawson Engler, Cristian Cadar and others at Stanford
    - Klee : Cadar, Dunbar, Engler
    - MINESWEEPER: Bug Finder by Dawn Song and her group at CMU

1.5 约束求解

- Z3求解器
  - 微软研究院Leonardo de Moura主持设计，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

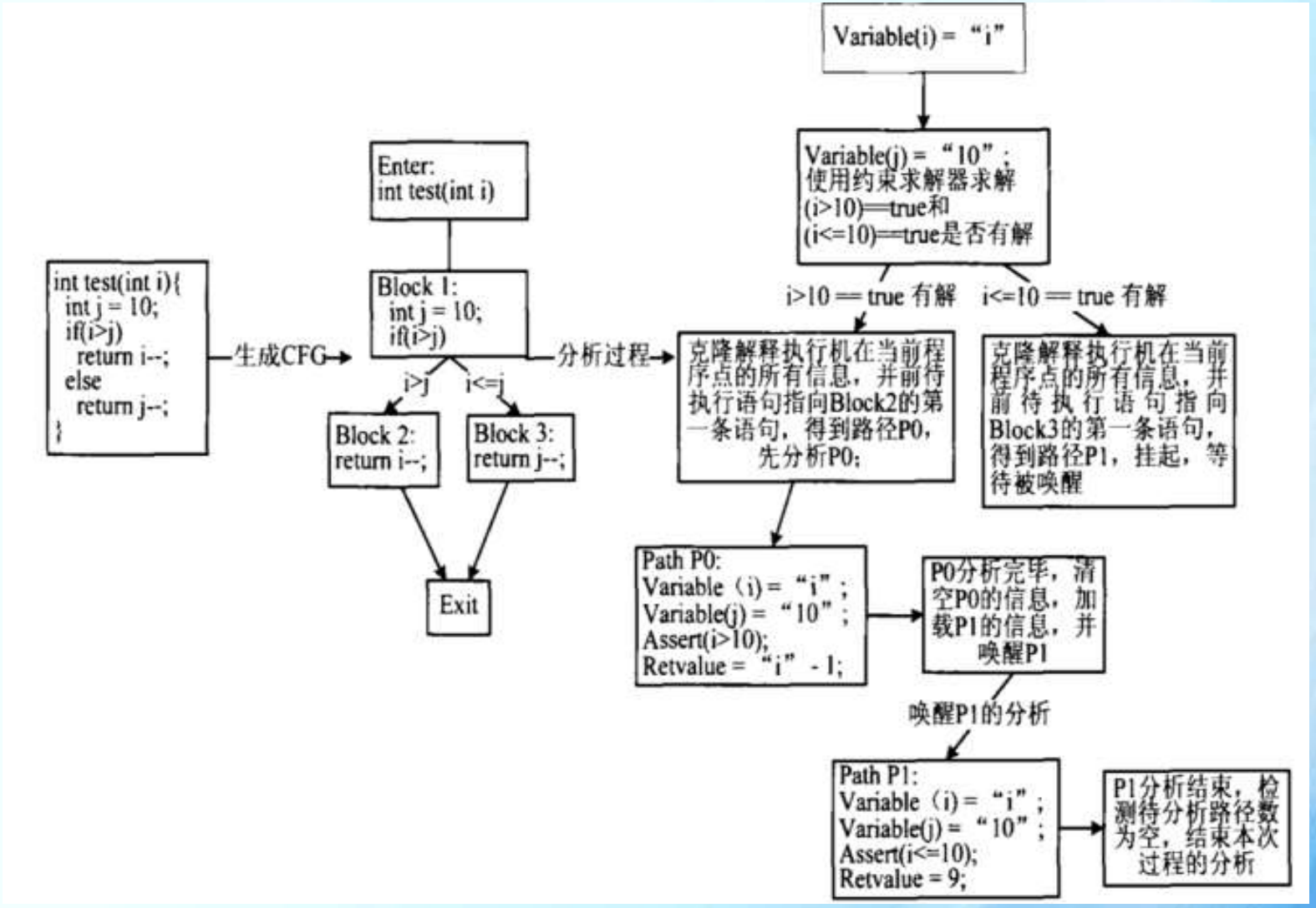
- 过程内分析
- 过程间分析

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

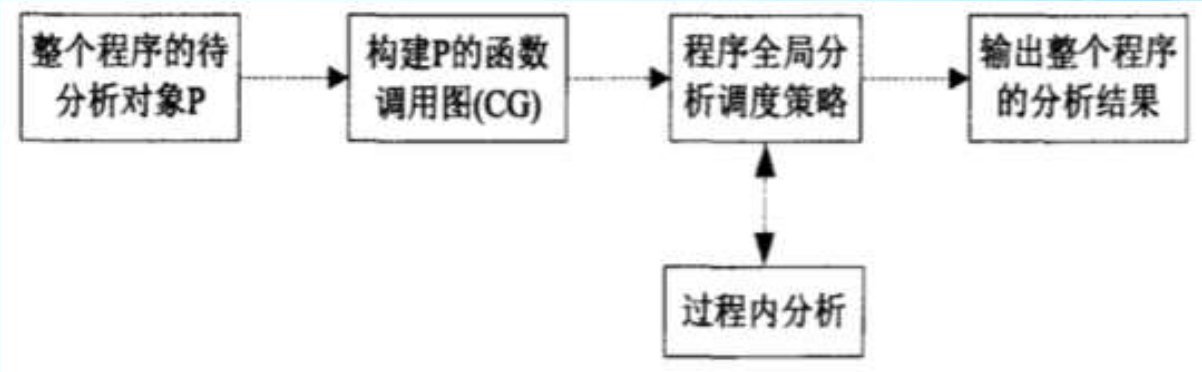
1.6 符号执行技术分类

- 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



1.7 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，主要原因是，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

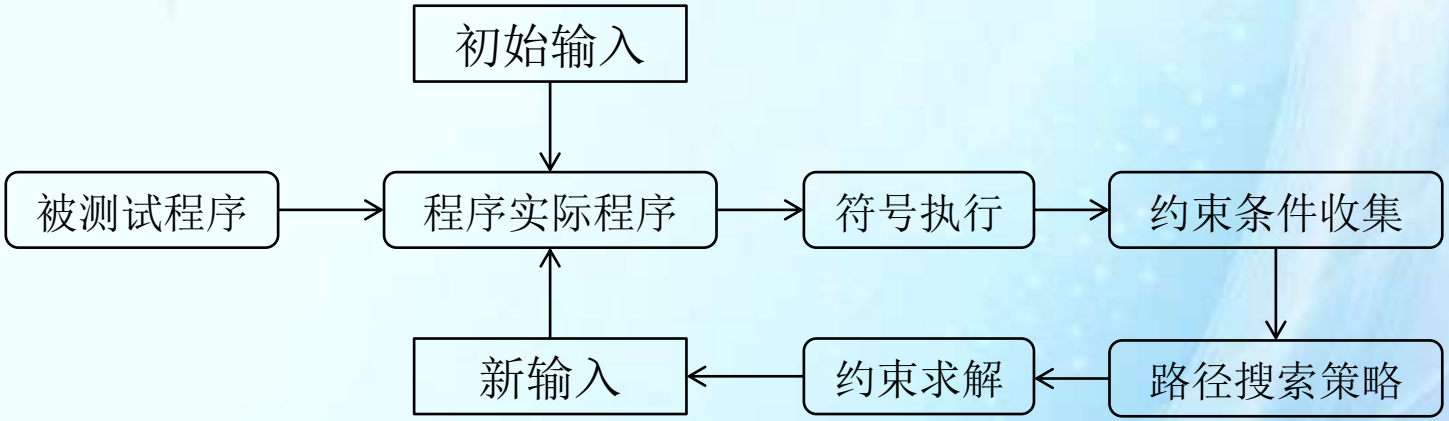
- 动态符号执行
  - **conc**rete + symb**olic** = **concolic**
  - 由Patrice Godefroid等，在《DART》中首次提出
- 设计目标
  - 为了解决静态符号执行效率低，系统开销大，误报率高的问题
  - 结合符号执行与具体执行的优势，在保证测试精度的前提下对程序执行树进行遍历

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，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，提取路径的约束表达式
  - 根据路径搜索策略（深度，广度）对约束表达式进行变形，求解变形后的表达式并生成新的测试用例

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
  - 下图就是混合符号执行中典型的循环问题

```
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

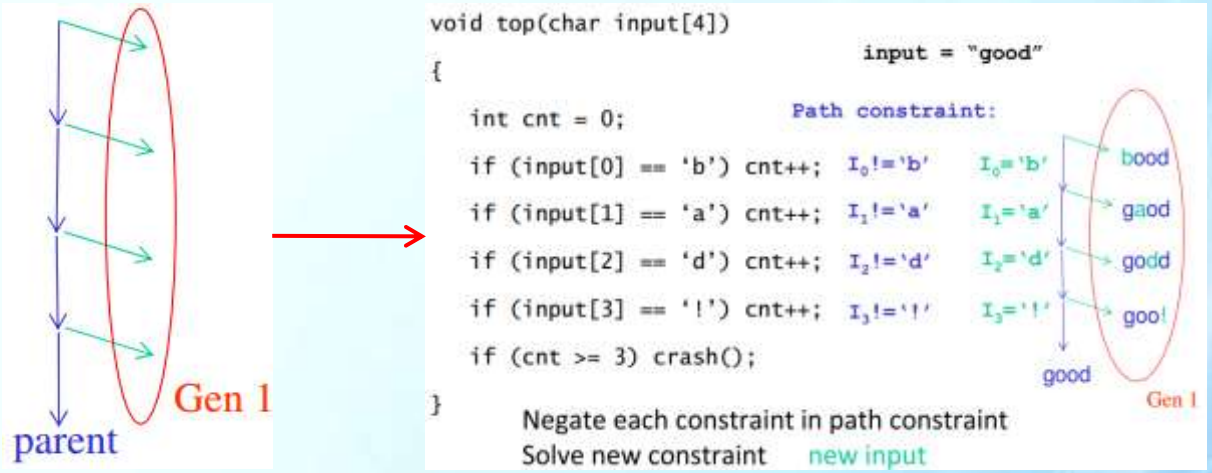
- 缓解方案
  - 约束优化：对由循环造成的冗余约束条件进行规约优化，相当于剪枝处理
    - 使用该策略的工具：SAGE
  - 限制循环执行次数：限制循环语句的展开次数，或按照固定次数对循环语句进行展开
    - 使用该策略的工具：IntScope等
  - 循环摘要：对循环片段建立归纳变量与控制执行流程退出循环的变量之间的函数关系，避免对循环内部逻辑的展开分析，直接对循环内部所有路径进行遍历
    - 使用该策略的工具：LESE等

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

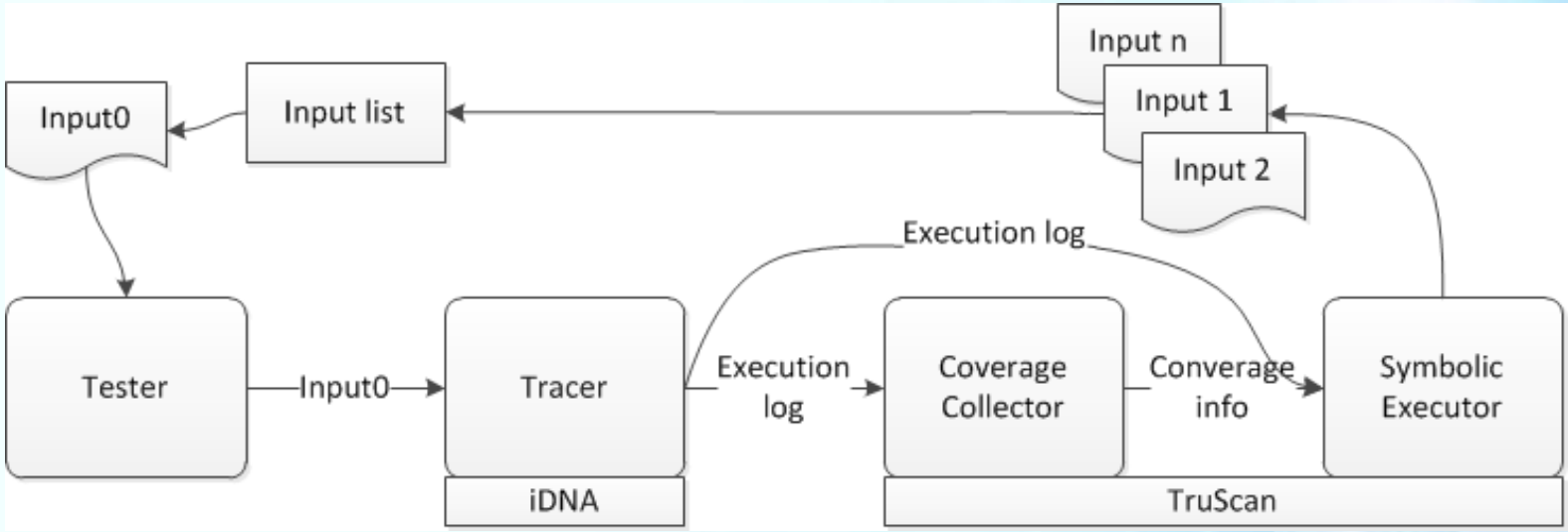
2.5 工具介绍

- 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

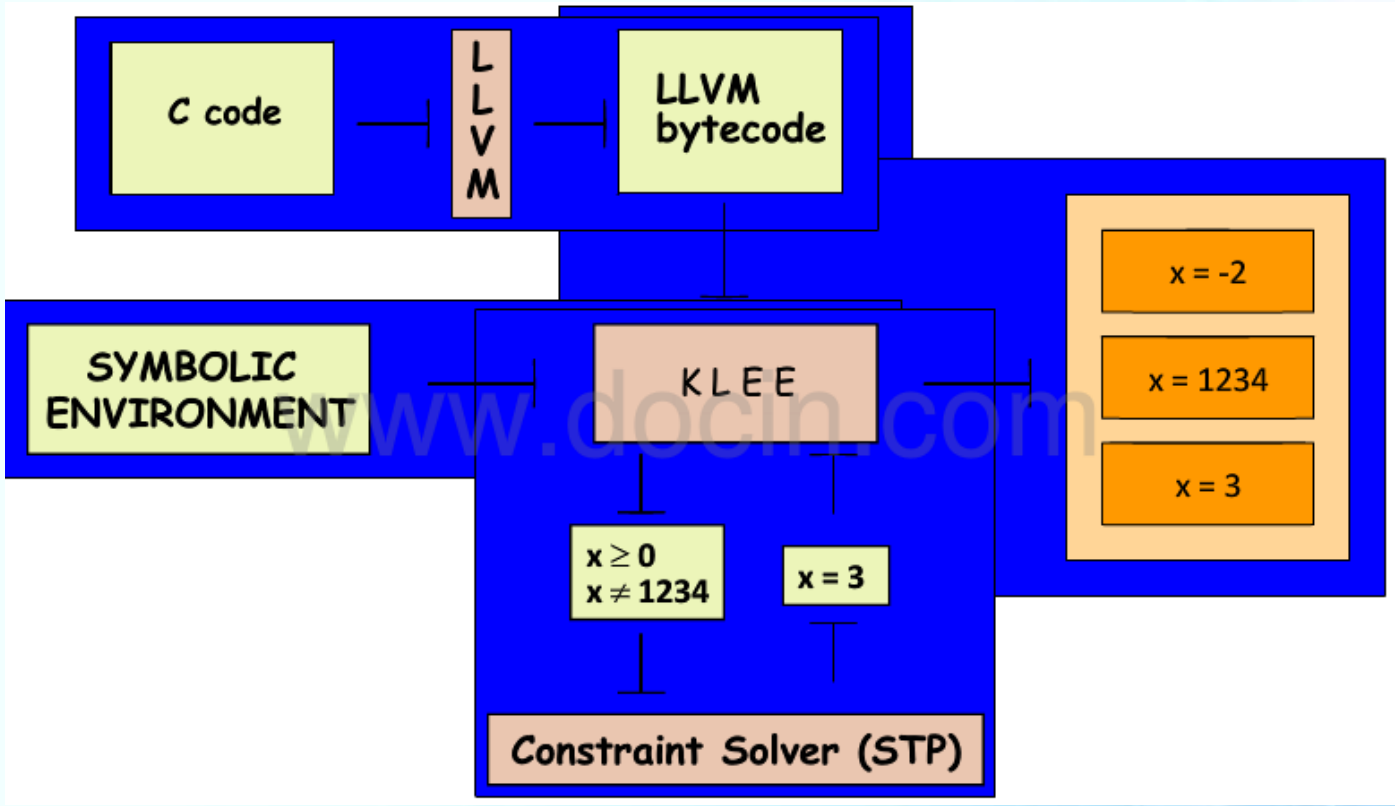


2.5 工具介绍

- 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



2.6 小结

TCA

- 小结
  - 为了解决静态符号执行效率低，系统开销大，误报率高的问题，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，在保证一定测试精度的前提下快速遍历执行树
  - 动态符号执行技术的基本思想是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
  - 动态符号执行技术近年来逐渐被重视，并在产品测试环节得到一定应用，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
  - 动态符号执行面临外部函数调用，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

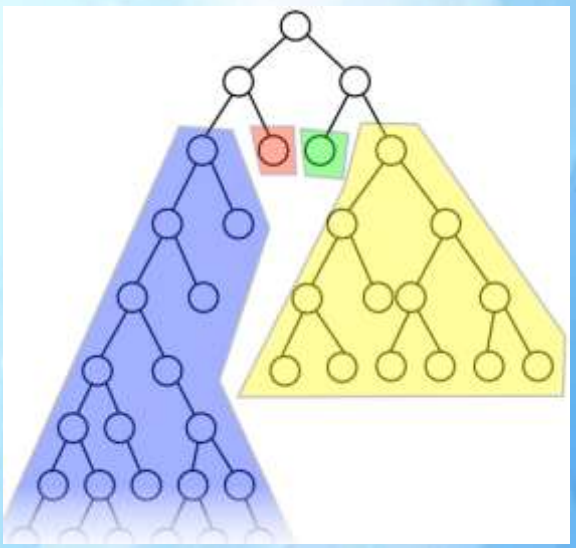
- 定义
- 关键问题
- 工具介绍

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？



3.4 小结

- 小结
  - 为了进一步提升动态符号执行技术的效率，研究人员引入并行技术，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。
  - 并行符号执行技术中，高效的路径搜索算法和负载均衡算法是系统设计的核心，本小结中介绍的三个系统分别给出了设计原则和实现方案
  - 虽然带来了执行效率上的提升，但并行符号执行技术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

TCA

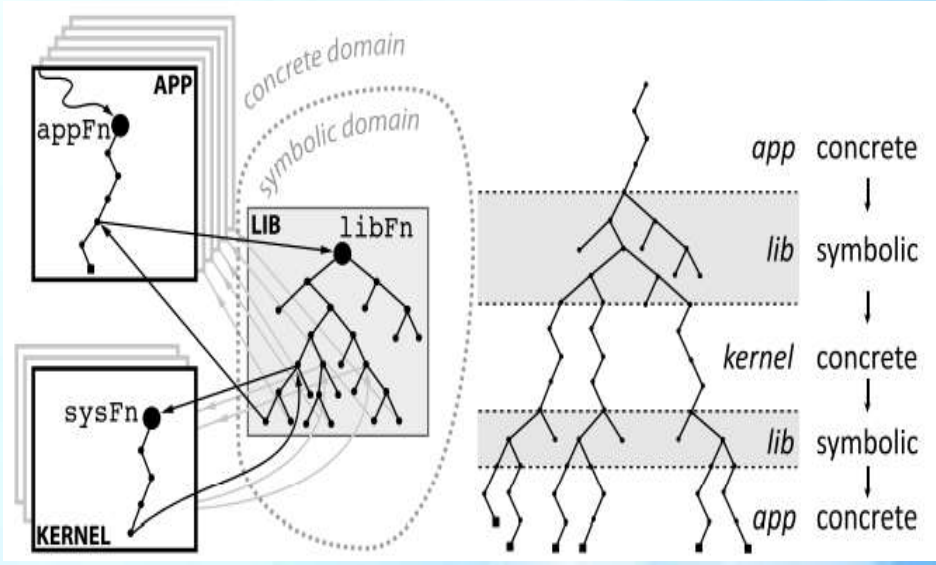
- 定义
  - 只对程序路径中的一段区域，或者只对部分路径进行符号执行分析就称之为选择符号执行，传统符号执行可看做是全局符号执行
- 设计初衷
  - 传统符号执行技术会对程序执行树的所有分支和叶子节点进行遍历分析，人们希望通过改进路径搜索算法和使用并行技术加快对执行树的遍历，但效果都并不理想
  - 瑞士洛桑理工大学的Vitaly Chipounov等人解决该问题时另辟蹊径，他们发现，一个程序的执行树中，对分析人员来说并不是每条路径都有价值，通常情况下可能只有部分子树与最终分析目标相一致

4.2 基本思想

- 基本思想
  - S2E的设计方案是，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，即对遇到的所有路径分支进行分析，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

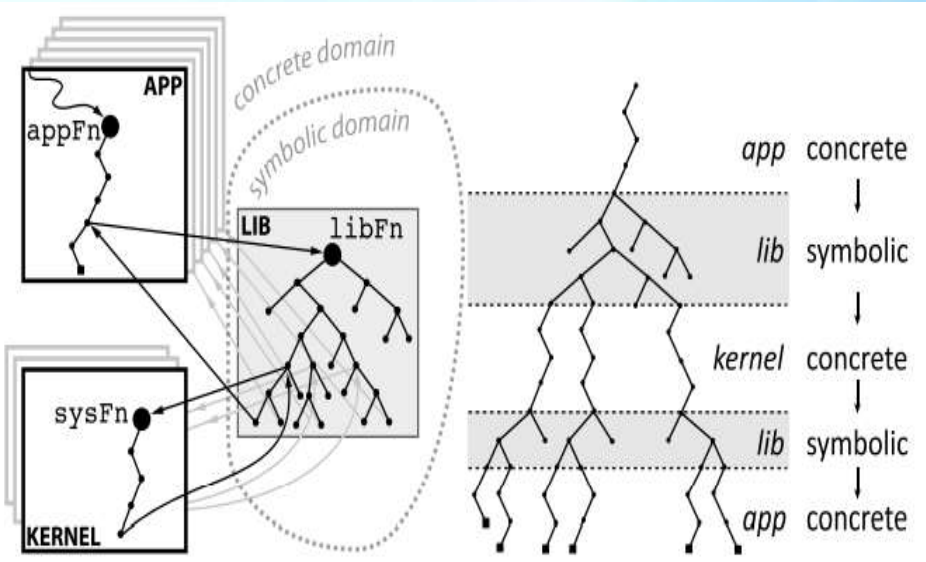
4.3 执行流程

- **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

TCA

- 小结
  - 传统符号执行技术会对程序执行树的所有分支和叶子节点进行遍历分析，人们希望通过改进路径搜索算法和使用并行技术加快对执行树的遍历，但结果对效率的提升都有限，部分研究人员提出对执行路径的局部目标段进行分析，通过减小符号分析区间来缓解路径爆炸问题
  - 选择符号执行的基本思路是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，即对遇到的所有路径分支进行分析，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
  - 因为选择符号执行的使用场景局限性较强，目前使用并不广泛

End
&
Thanks