

## 第八章 漏洞挖掘技术进阶

知识点六：符号执行基本原理

知识点七：Z3约束求解器

知识点八：Angr应用示例

知识点九：污点分析基本原理

知识点十：污点分析方法

# 知识点六：符号执行基本原理

## 符号执行 三个关键点

变量符号化

程序执行模拟

程序执行模拟，可以收集到哪些信息？  
程序执行模拟具体如何工作的？

约束求解

约束求解，对谁求解，得到结果是什么？

## 1. 程序执行状态

- 符号执行具体执行时，程序状态中通常包括：程序变量的具体值、程序指令计数和路径约束条件pc (path constraint)。
- pc是符号执行过程中对路径上条件分支走向的选择情况，根据状态中的pc变量就可以确定一次符号执行的完整路径。**pc初始值为true。**
- 举例来说，假设符号执行过程中经过3个与符号变量相关的if条件语句if1、if2、if3，每个条件表达式如下：  
$$if_1: a_1 \geq 0 \quad if_2: a_1 + 2 * a_2 \geq 0 \quad if_3: a_3 \geq 0$$
- 设引擎在3个if条件分支处分别选择if1: true, if2: true, if3: false, 则pc表示为：  
$$pc = (a_1 \geq 0 \wedge a_1 + 2 * a_2 \geq 0 \wedge \neg(a_3 \geq 0))$$

□ 假设if处的表达式为 $R \geq 0$ ， $R$ 是一个与符号变量相关的多项表达式，把 $R \geq 0$ 称为 $q$ ，则程序执行到if处时pc可能会表现为下面两种形式之一：

(1) pc包含 $q$       (2) pc包含 $\neg q$

□ 如果符号执行引擎选择进入**then**分支，则 $R \geq 0$ 的真值为true，pc表现为(1)的形式，且记为： $pc = pc \wedge q$

□ 如果选择**else**分支，则 $R \geq 0$ 的 false真值为，pc表现为(2)的形式，且记为： $pc = pc \wedge \neg q$

□ 要确定pc对应路径的程序输入参数，只需要使用约束求解器对pc进行求解就可以。



## 2. 符号传播

□ 符号传播主要作用是建立符号变量传播的关系，并且更新映射的关系。

在实际操作的过程中，通常是将对应内存地址的数据进行变化。

```
1 int x;  
2 int y, z;  
3 y=x*3;  
4 z=y+5;
```

符号量的内存地址	符号值
add_x	X

最初的符号映射表如表所示，addr\_x代表第1行中变量x的地址，X为对应地址应该存放的符号表达式。变量y和变量z不需要进行符号化，因为他们两个的值都取决于x

符号量的内存地址	符号值
add_x	X
add_y	$X*3$
add_z	$X*3 + 5$



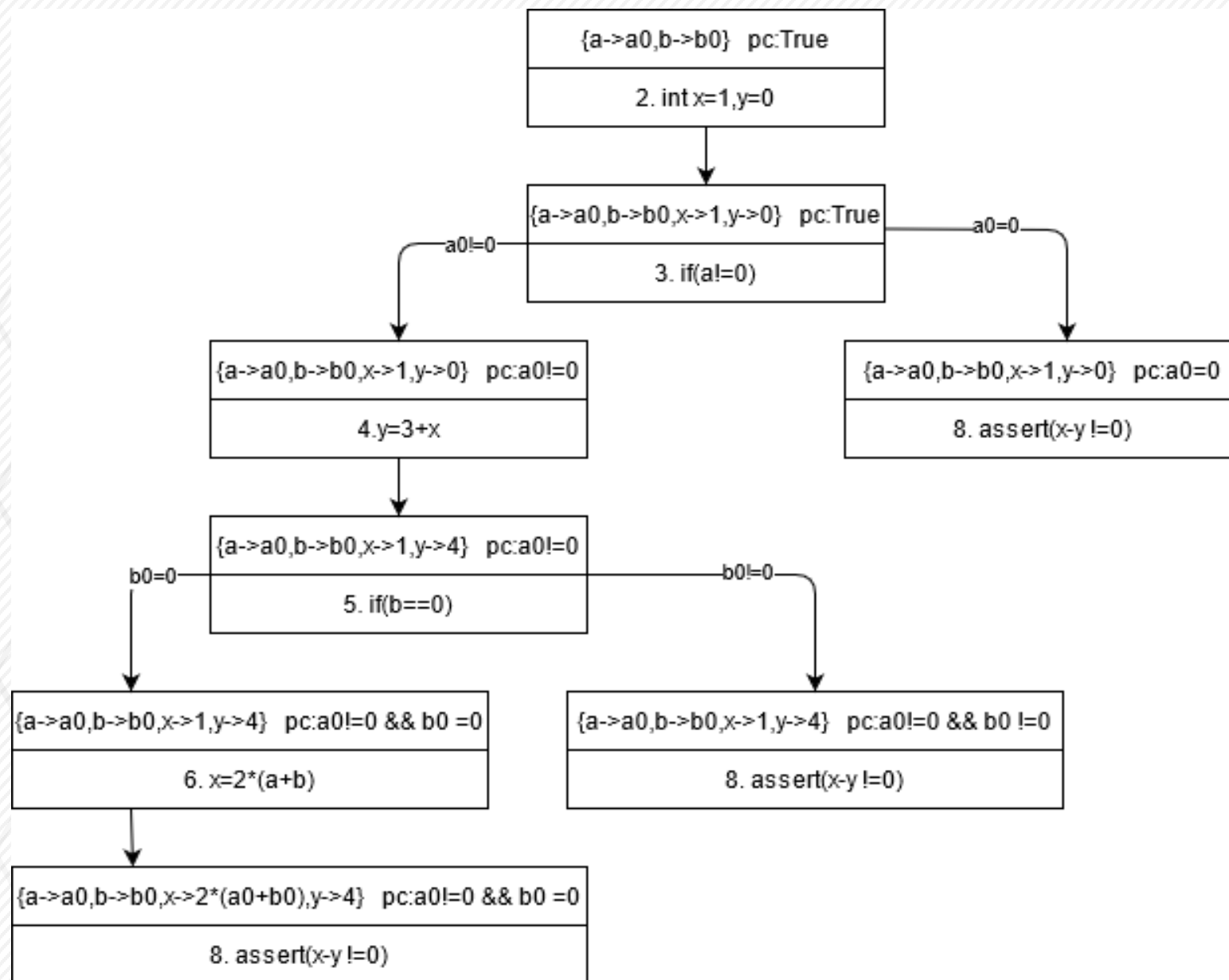
### 3. 符号执行树

- 如何形式化地表示符号执行的过程呢？程序的所有执行路径可以表示为树，叫做执行树。符号执行过程也是对执行树进行遍历的过程。
- 执行树中的一个节点对应程序中的一条语句，程序语句之间的执行顺序或跳转关系对应执行树中节点间的边，对于每个语句会有两条边与其相连，左子树对应的是if语句的true(then)分支，右子树对应if语句的false(else)分支。
- 执行树中还可以包含指令计数、pc(路径约束条件)、变量符号值等程序执行状态信息。

```

1 void foobar(int a,int b){
2   int x=1,y=0;
3   if(a != 0){
4     y = 3+x;
5     if (b ==0)
6       x = 2*(a+b);
7   }
8   assert(x-y !=0);

```



符号执行完得到三条路径，可以对路径约束条件pc进行约束求解得到到达该路径的一组输入，结合assert的约束 $x-y \neq 0$ 就可以进行求解出触发约束的输入。



## 4. 约束求解

- 符号执行得到的约束条件，可以通过约束求解器进行求解。
- 主流的约束求解器主要有两种理论模型：SAT求解器和SMT求解器。
- **SAT问题 (The Satisfiability Problem, 可满足性问题)**，求解由布尔变量集合组成的布尔表达式，对命题逻辑公式问题适用，但是当前有很多实际应用的问题，并不能直接转换为SAT问题来进行求解。
- **SMT (Satisfiability Module Theories, 可满足性模理论)**，求解范围从命题逻辑公式扩展为可以解决一阶逻辑所表达的公式。SMT包含很多的求解方法，通过组合这些方法，可以解决很多问题。
- **Z3就是一个典型的约束求解器。**

## 5. 符号执行方法分类

- **静态符号执行本身不会实际执行程序，通过解析程序和符号值模拟执行，有代价小、效率高的优点，但是存在路径爆炸、误报高的情况。**
- **动态符号执行也称为混合符号执行，它的基本思想是：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，不断迭代上面的过程，直到完全遍历程序的所有执行路径。动态符号执行结合了真实执行和传统符号执行技术的优点，在真实执行的过程中同时进行符号执行，可以在保证测试精度的前提下对程序执行树进行快速遍历。**
- **选择性符号执行可以对程序员感兴趣的部分进行符号执行，其它的部分使用真实值执行，在特定任务环境下可以进一步提升执行效率。**

# 知识点七：Z3约束求解器

## 1. Z3

- Z3是一个微软出品的SMT问题的开源约束求解器，能够解决很多种情况下的给定部分约束条件寻求一组满足条件的解的问题（可以理解为自动解方程组）。
- Z3在工业应用中常见于软件验证、程序分析等。由于Z3功能实在强大，也被用于很多其他领域：软件/硬件验证和测试、约束解决、混合系统分析、安全性、生物学（计算机模拟分析）和几何问题。著名的二进制分析框架angr也内置了一个修改版的Z3。

Z3是个开源项目，Github链接：<https://github.com/z3prover>。

## (1) Window下安装Z3

下载x64-win版：<https://github.com/Z3Prover/z3/releases>。

解压到D:\z3-4.8.10，可以看到文件夹里包含bin子文件夹，里面有可执行文件z3.exe。

**配置PATH。**打开“我的电脑”的属性窗口，选择“高级系统设置”，在“高级 环境变量”里，编辑path，添加D:\z3-4.8.10\bin。

**安装Python。**Windows 10系统中，在命令控制台里输入python3会自动弹出商店安装，也可以自己到网上下载环境进行安装。



## 2. Z3常用API

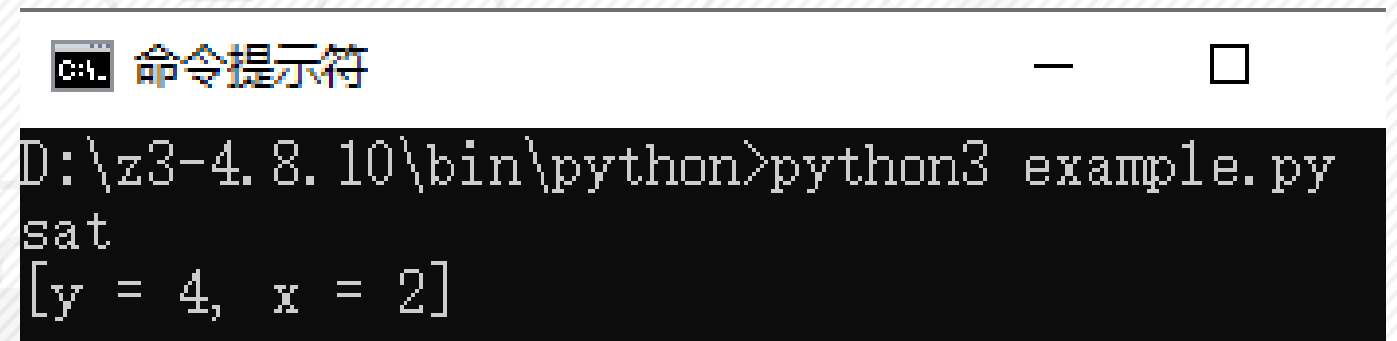
- **Solver()**: 创建一个通用求解器，创建后可以添加约束条件，进行下一步的求解。
- **add()**: 添加约束条件，通常在solver()命令之后。
- **check()**: 通常用来判断在添加完约束条件后，来检测解的情况，有解的时候会回显sat，无解的时候会回显unsat。
- **model()**: 在存在解的时候，该函数会将每个限制条件所对应的解集取交集，进而得出正解。



### 3. Z3简单示例

```
from z3 import *  
  
x = Real('x')  
y = Real('y')  
s = Solver()  
s.add(x + y > 5, x > 1, y > 1)  
print(s.check())  
print(s.model())
```

打开命令控制台，进入D:\z3-4.8.10\bin\python，  
执行example.py，如下：



```
命令提示符  
D:\z3-4.8.10\bin\python>python3 example.py  
sat  
[y = 4, x = 2]
```

# 知识点八：Angr应用示例

## 1. Angr安装

- Angr是一个基于python的二进制漏洞分析框架，它将以前多种分析技术集成进来，它能够进行动态的符号执行分析（如KLEE和Mayhem），也能够进行多种静态分析。
- Windows下安装Angr。首先安装Python3，如果安装了就忽略。可以到python官方网站下载安装版本，选择将python增加到path中。然后，打开命令控制台，使用PIP命令安装angr： `pip install angr`。
- 测试安装。输入命令python，进入python界面，然后输入 `import angr`，如果成功，则说明安装没有问题。

```
C:\Users\liuzheli>python
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import angr
>>>
```

## 2. Angr示例

- **Angr官方手册。** GitHub上有angr的开源项目<https://github.com/angr>以及相关的文档信息，建议将<https://github.com/angr/angr-doc>里的所有文档以zip方式下载到本地。
- **angr-doc里有各类Example**，展示了Angr的用法，比如cmu\_binary\_bomb、simple\_heap\_overflow等二进制爆破、堆溢出等漏洞挖掘、软件分析的典型案例。
- **以sym-write为例子，来说明angr的用法：**
  - ① 怎么使用angr?
  - ② 使用angr能解决什么问题?

```
#include <stdio.h>
char u=0;
int main(void){
    int i, bits[2]={0,0};
    for (i=0; i<8; i++) {
        bits[(u&(1<<i))!=0]++;
    }
    if (bits[0]==bits[1]) {
        printf("you win!");
    }
    else {
        printf("you lose!");
    }
    return 0;
}
```

两条路径

什么样的u可以win?

如果u二进制中1和0个数不同, lose

□ ~~变量符号化~~: 将u进行符号化

□ ~~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。

□ ~~获取路径约束条件~~

□ ~~约束求解~~

```
import angr
```

```
import claripy
```

```
def main():
```

```
    p = angr.Project('./issue', load_options={"auto_load_libs": False})
```

```
    state = p.factory.entry_state(add_options={angr.options.SYMBOLIC_WRITE_ADDRESSES})
```

```
    u = claripy.BVS("u", 8)
```

```
    state.memory.store(0x804a021, u)
```

新建一个工程，导入二进制文件，选项是选择不自动加载依赖项，不会自动载入依赖的库

初始化模拟程序状态的SimState对象state，该对象包含了程序内存、寄存器、符号信息等模拟运行时动态数据

创建符号变量u，以8位bitvector形式存在。存储到二进制文件.bss段u的地址



```
sm = p.factory.simulation_manager(state)
```

```
def correct(state):
```

```
    try:
```

```
        return b'win' in state.posix.dumps(1)
```

```
    except:
```

```
        return False
```

```
def wrong(state):
```

```
    try:
```

```
        return b'lose' in state.posix.dumps(1)
```

```
    except:
```

```
        return False
```

```
sm.explore(find=correct, avoid=wrong)
```

```
return sm.found[0].solver.eval_upto(u, 256)
```

```
if __name__ == '__main__':
```

```
    print(repr(main()))
```

创建一个Simulation Manager  
对象，管理运行得到的状态对象

定义函数：state.posix.dumps(1)  
获得所有标准输出

动态符号执行&得到想要的状态→使用  
explore函数进行状态搜寻  
也可以写成：sm.explore(find=0x80484e3,  
avoid=0x80484f5)

约束求解→获得state之后，通  
过solver求解器，求解u的值

**实验验证。**在windows 10环境下，选择填写的solve.py，点右键选择Edit with IDLE Edit with IDLE 3.9 (64 bit)，将弹出界面，选择Run run model，界面如下

### 3. 其他解法

```
import angr
```

```
import claripy
```

```
def hook_demo(state):
```

```
    state.regs.eax = 0
```

```
p = angr.Project("./issue", load_options={"auto_load_libs": False})
```

```
p.hook(addr=0x08048485, hook=hook_demo, length=2)
```

```
state = p.factory.blank_state(addr=0x0804846B, add_options={"SYMBOLIC_WRITE_ADDRESSES"})
```

**演示Hook:** 0x08048485处指令为xor eax, eax, hook一个函数, 指令长度为2, 实际并没有带来任何变化, 仅为Hook演示

**其他创建state方式:** 使用blank\_state创建状态对象, 指定了程序入口点位置为主函数第一行代码

### 3. 其他解法

```
u = claripy.BVS("u", 8)
```

```
state.memory.store(0x0804A021, u)
```

```
sm = p.factory.simulation_manager(state)
```

```
sm.explore(find=0x080484DB)
```

状态搜寻：只给定一个条件，因为是分支语句，已经足以确定唯一路径

```
st = sm.found[0]
```

```
print(repr(st.solver.eval(u)))
```

eval(u)替代了原来的eval upto，  
将打印一个结果出来

# 知识点九：污点分析基本原理



- 污点分析是信息流分析的一种实践技术：如果系统满足了用户定制的信息流策略，那么系统是信息流安全的。
- 污点分析标记程序中的数据（外部输入数据或者内部数据）为污点，通过对带污点数据的传播分析来达到保护数据完整性和保密性的目的。  
如果信息从被标记的污点数据传播给未标记的数据，那么需要将未标记的标记为污点数据；如果被标记的污点数据传递到重要数据区域或者信息泄露点，**那就意味着信息流策略被违反。**
- 污点分析被广泛地应用在隐私数据泄露检测、漏洞挖掘等实际领域。



污点分析可以抽象成一个三元组 (sources, sinks, sanitizers) 的形式:

**source即污点源**, 代表直接引入不受信任的数据或者机密数据到系统中;

**sink即污点汇聚点**, 代表直接产生**安全敏感操作(违反数据完整性)**或者泄露**隐私数据到外界(违反数据保密性)**;

**sanitizer即无害处理**, 代表通过数据加密或者移除危害操作等手段使数据传播不再对软件系统的信息安全产生危害。

□ 污点分析就是分析程序中由污点源引入的数据是否能够不经无害处理, 而直接传播到污点汇聚点。如果不能, 说明系统是**信息流安全**的; 否则, 说明系统产生了**隐私数据泄露或危险数据操作**等安全问题。

□ 可以分成3个阶段: 识别污点源和汇聚点、污点传播分析和无害处理。

## 1.识别污点源和汇聚点

**识别污点源和污点汇聚点是污点分析的前提。**目前，在不同的应用程序中识别污点源和汇聚点的方法各不相同，缺乏通用方法。

现有的识别污点源和汇聚点的方法可以大致分成3类：

- 使用启发式的策略进行标记，例如把来自程序外部输入的数据统称为“污点”数据，保守地认为这些数据有可能包含恶意的攻击数据；
- 根据具体应用程序调用的API或者重要的数据类型，手工标记源和汇聚；
- 使用统计或机器学习技术自动地识别和标记污点源及汇聚点。

## 2.污点传播分析

污点传播分析就是分析污点标记数据在程序中的传播途径。按照分析过程中关注的程序依赖关系的不同，可以将污点传播分析分为显式流分析和隐式流分析。

```
1 void foo () {  
2   int a = source() ,  
3   int b = source() ;  
4   int x, y;  
5   x = a * 2 ;  
6   y = b + 4 ;  
7   sink(x);  
8   sink(y);  
9 }
```

→ 显式污点传播

- 污点传播分析中的显式流分析就是分析污点标记如何随程序中变量之间的数据依赖关系传播。
- 左图很明显，在对sink点进行污点判定的时候，可以发现代码存在信息泄漏的问题，即通过sink点可以推测输入的值。

```

1 void foo () {
2   String X = source();
3   String Y = new String();
4   for (int i = 0; i < X.length(); i++) {
5     int x = (int)X.charAt(i);
6     int y = 0;
7     for (int j = 0; j < x; j++) {
8       y = y + 1;
9     }
10    Y = Y + (char)y;
11  }
12  sink(Y);
13 }

```

 显式污点传播   
  隐式污点传播

□ 污点传播分析中的**隐式流分析**  
 是分析污点标记如何随程序中  
 变量之间的**控制依赖关系传播**，  
 也就是分析污点标记如何从条  
 件指令传播到其所控制的语句。

□ 变量X是被污点标记的字符串类型变量，变量Y和变量X之间并没有直接  
 或间接的数据依赖关系(显式流关系)，但X上的污点标记可以经过控制  
 依赖隐式地传播到Y。最终，第12行的Y值和X值相同。但是，如果不进  
 行隐式流污点传播分析，第12行的变量Y将不会被赋予污点标记。



隐式流污点传播一直以来都是一个重要的问题，如果不被正确处理，会使污点分析的结果不精确。

**欠污染：**由于对隐式流污点传播处理不当导致本应被标记的变量没有被标记的问题称为欠污染(under-taint)问题。

**过污染：**由于污点标记的数量过多而导致污点变量大量扩散的问题称为过污染(over-taint)问题。

目前，针对隐式流问题的研究重点是尽量减少欠污染和过污染的情况。

### 3.无害处理

- 污点数据在传播的过程中可能会经过无害处理模块，**无害处理模块是指污点数据经过该模块的处理后，数据本身不再携带敏感信息或者针对该数据的操作不会再对系统产生危害。**换言之，带污点标记的数据在经过无害处理模块后，污点标记可以被移除。
- 正确地使用无害处理可以降低系统中污点标记的数量，提高污点分析的效率，并且避免由于污点扩散导致的分析结果不精确的问题。常数赋值是最直观的无害处理的方式；加密处理、程序验证等在一定程度上，可以认为是无害处理。



# 知识点十：污点分析方法

## 1. 显式流分析

### 静态分析

静态污点传播分析(简称静态污点分析)是指在不运行且不修改代码的前提下，通过分析程序变量间的**数据依赖**关系来检测数据能否从污点源传播到污点汇聚点。

静态污点分析的对象一般是程序的源码或中间表示，可以将对污点传播中显式流的静态分析问题转化为对程序中**静态数据依赖**的分析:首先，根据程序中的函数调用关系构建**调用图**(call graph, 简称CG); 然后，在函数内或者函数间根据不同的程序特性进行具体的**数据流传播分析**。

常见的显式流污点传播方式包括直接赋值传播、通过函数(过程)调用传播以及通过别名(指针)传播。

```
1 void main () {  
2   Data a = new A();  
3   int b = source();  
4   int c = b + 10;  
5   foo(a, a, c);  
6 }  
7 void foo (Data x, Data y, int z) {  
8   x.f = z;  
9   sink(y.f);  
10 }
```

直接赋值传播

函数调用传播

别名传播

显式污点传播

- 由于foo的两个参数对象x和y都是对对象a的引用(Java程序), 二者之间存在别名, 存在信息泄露。
- 利用数据流分析解决显式污点传播分析中的直接赋值传播和函数调用传播已经相当成熟, 研究的重点是如何为别名传播的分析提供更精确、高效的解决方案。

## 1. 显式流分析

### 动态分析

动态污点传播分析(简称动态污点分析)是指在程序运行过程中,通过实时监控程序的污点数据在系统程序中的传播来检测数据能否从污点源传播到污点汇聚点。

动态污点传播分析首先需要为污点数据扩展一个污点标记(tainted tag)的标签并将其存储在存储单元(内存、寄存器、缓存等)中,然后根据指令类型和指令操作数设计相应的传播逻辑传播污点标记。

**动态污点传播分析按照实现层次被分为三类：**

□ **基于硬件的污点传播分析**需要定制的硬件支持，一般需要在原有体系结构上为寄存器或者内存扩展一个标记位，用来存储污点标记。

□ **基于软件的污点传播分析**通过修改程序的二进制代码来进行污点标记位的存储与传播。基于软件的污点传播的优点在于不必更改处理器等底层的硬件，并且可以支持更高的语义逻辑的安全策略，但**缺点是使用插桩或代码重写修改程序往往会给分析系统带来巨大的开销**。基于硬件的污点传播分析虽然可以利用定制硬件降低开销，但通常不能支持更高的语义逻辑的安全策略，并且需要对处理器结构进行重新设计。

□ **混合型的污点分析**是对上述两类方法的折中。



## 如何降低分析代价？

动态污点传播分析的一个研究重点是如何降低分析代价。基于硬件的分析技术需要定制硬件的支持，基于软件的技术由于程序插桩或代码重写会带来额外的性能开销。

- 一类研究思路是有选择地对系统中的指令进行污点传播分析。例如，LIFT提出的快速路径(fast-path)优化技术通过提前判断一个模块的输入和输出是否是具有威胁的(如果没有威胁，则无需进行污点传播)以降低需要重写的代码的数量；
- 另外一类思路是使用低开销的机制代替高开销机制。例如，LIFT的快速切换(fast switch)优化使用低开销的lahf/sahf指令代替高开销的pushq/popq指令，以提高插桩代码与原始二进制文件之间的切换效率。



## 2. 隐式流分析

污点传播分析中的隐式流分析就是分析污点数据如何通过**控制依赖**进行传播，如果忽略了对隐式流污点传播的分析，则会导致欠污染的情况；如果对隐式流分析不当，那么除了欠污染之外，还可能出现过污染的情况。

### 静态隐式流分析

面临的核心问题是精度与效率不可兼得的问题。

- 精确的隐式流污点传播分析需要分析**每一个分支控制条件是否需要传播污点标记**。路径敏感的数据流分析往往会产生路径爆炸问题，导致开销难以接受。
- 简单的静态传播(标记)分支语句的污点标记方法是**将控制依赖于它的语句全部进行污点标记**，但该方法会导致一些并不携带隐私数据的变量被标记，导致过污染情况的发生。

## 2. 隐式流分析

### 动态隐式流分析

有三个问题需要解决：

□ 如何确定污点控制条件下需要标记的语句的范围？

动态执行轨迹并不能反映出被执行的指令之间的控制依赖关系

□ 由于部分泄漏导致的漏报如何解决？

指污点信息通过动态**未执行**部分进行传播并泄漏

□ 如何选择合适的污点标记分支进行污点传播？

鉴于单纯地将所有包含污点标记的分支进行传播会导致过污染的情况

## 如何确定污点控制条件下需要标记的语句的范围

```
1 x = false;
2 y = false;
3 if (document.cookie == "abc") { //source
4     x = true;
5 } else {
6     y = true;
7 }
8 if (x == false) {
9     sink(x);
10 }
11 if (y == false) {
12     sink(y);
13 }
```

污点控制条件

后支配关系标记算法：对后面这条语句进行污点标记

- 动态执行轨迹并不能反映出被执行的指令之间的控制依赖关系
- 目前的研究多采用离线的静态分析辅助判断动态污点传播中的隐式流标记范围。
- 利用离线静态分析得到的控制流图节点间的后支配关系来解决动态污点传播中的隐式流标记问题。

## 如何解决部分泄漏导致的漏报

```
1 x = false;
2 y = false;
3 if (document.cookie == "abc") { //source
4     x = true;
5 } else {
6     y = true;
7 }
8 if (x == false) {
9     sink(x);
10 }
11 if (y == false) {
12     sink(y);
13 }
```

污点控制条件

污点标记了x, 但没有标记y

有没有信息泄露?

部分泄漏是指污点信息通过动态未执行部分进行传播并泄漏。

□ 攻击者由第11行y等于false的条件能够反推出程序执行了第3行的分支条件，程序实际上存在信息泄漏的问题——**cookie不是abc**。

□ 可以对污点分支控制范围内的所有赋值语句中的变量都进行标记?? →然而，过污染!!

## 如何选择合适的污点标记分支进行污点传播

```
1 void foo () {  
2   int a = source();  
3   int x, y, z;  
4   int w = a + 10;  
5   if (a == 10) {  
6     x = 1;  
7   } else if (a > 10 & w <= 23) {  
8     y = 2;  
9   } else if (a < 10) {  
10    z = 3;  
11  }  
12  sink(x, y, z);  
13 }
```

单纯地将所有包含污点标记的分支进行传播  
会导致过污染的情况

□ 如果传播策略为只要分支指令中包含污点标记就对其进行传播，则三条分支语句后支配语句全部被标记。

- a等于10的情况：攻击者可以根据第12行泄漏的x的值直接还原出污点源处的值
- a大于10且小于或等于23的情况：攻击者只需要尝试3次就可还原出污点源处的值
- a小于10的情况：还原污点源处的值
- 根据信息泄漏范围的不同，定量地设计污点标记分支的选择策略**
- 于前两种，无需污点标记

道高一尺 魔高一丈

符号执行 污点分析

依然在路上