

看雪·第七届安全开发者峰会

# JDooop: 下一代针对Java Web应用的静态分析框架

陈浩浩&陈光义 京东安全



# 1. 引擎目标

```
// 读入不可信数据
String str = user_input();    // <= source
// 过滤
String cmd = filter(str);
// 不可信数据流向危险函数
execute(cmd);    // <= sink
```

数据流类型漏洞可以总结为两方面的问题

1. 数据流向问题: 不可信的数据流向了某些危险的函数中
2. 数据过滤问题: 过滤不可信数据时出现了遗漏

# 1. 引擎目标

```
// 读入不可信数据
String str = user_input();    // <= source
// 过滤
String cmd = filter(str);
// 不可信数据流向危险函数
execute(cmd);    // <= sink
```

注入类型漏洞可以总结为两方面的问题

1. 数据流向问题: 不可信的数据流向了某些危险的函数中
2. 数据过滤问题: 过滤不可信数据时出现了遗漏

# 1. 引擎目标

```
@RestController
@RequestMapping(value = "/api")
public class MyController {
    @GetMapping(value = "/test")
    public String test(@RequestParam String str) { // str为HTTP GET请求的参数
        return doSomething(str);
    }
}
```

Runtime.exec(...) 命令执行?

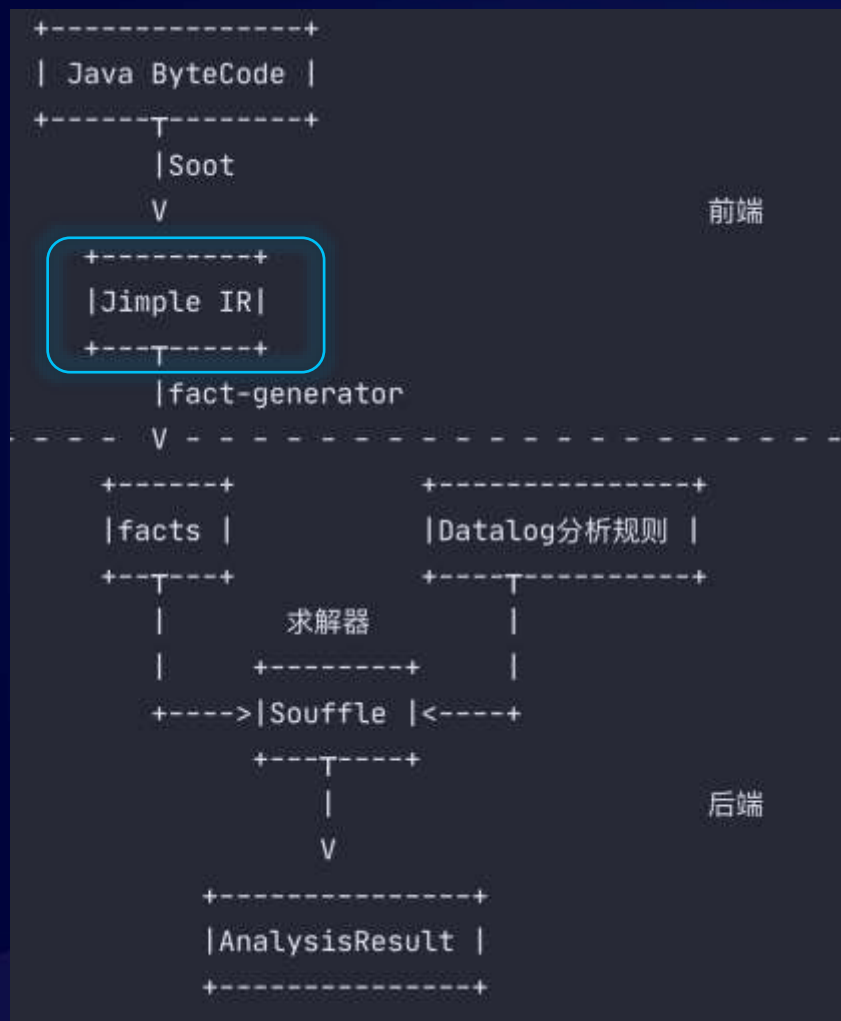
javav.sql.Statement.execute(...) SQL注入?

## 2. 引擎架构





## 2. 引擎架构



IR为jimple格式 example:

```

public void T16() throws java.lang.Exception
{
    java.lang.String $stack1;
    InformationFlowTest this#_0;

    this#_0 := @this: InformationFlowTest;

    $stack1 = staticinvoke <InformationFlowTest: java.lang.String source()>();

    staticinvoke <InformationFlowTest: void sink(java.lang.String)>($stack1);

    return;
}
  
```

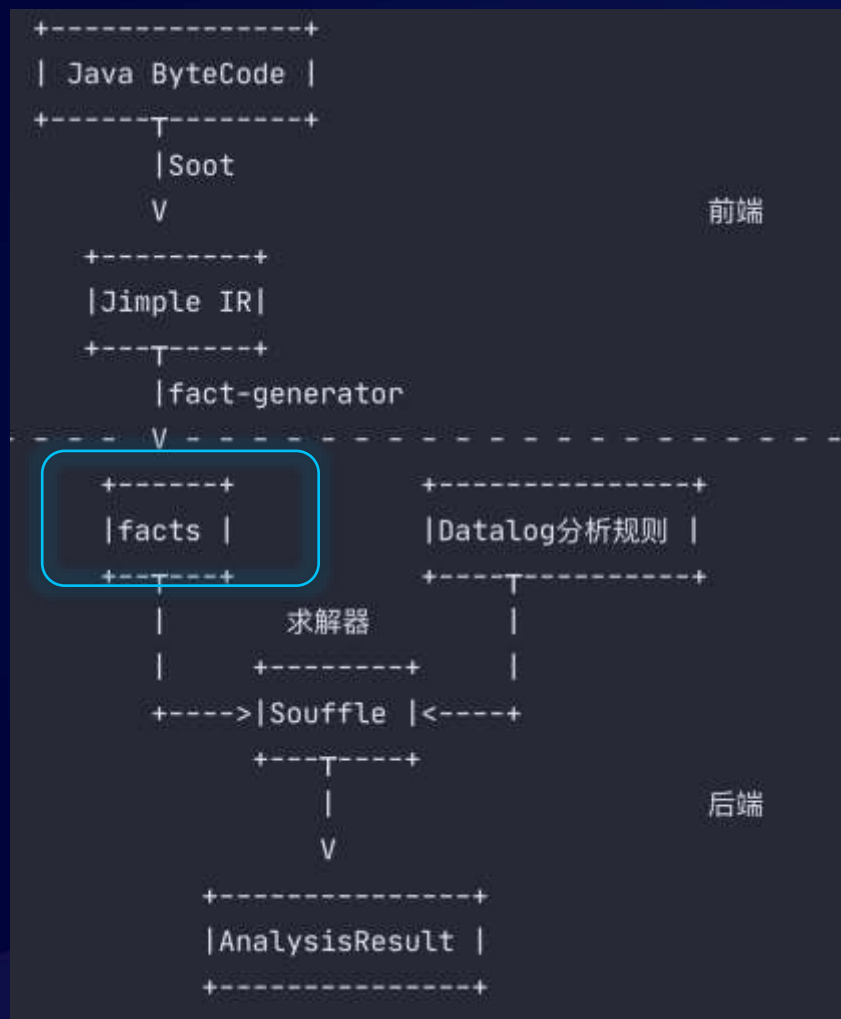
## 2. 引擎架构

facts例子:

```
1. a = new A();
2. b = a;
3. c = b;
```

Assign	
b	a
c	b

New	
a	o1



## 2. 引擎架构

facts例子:

```
1. a = new A();
2. b = a;
3. c = b;
```

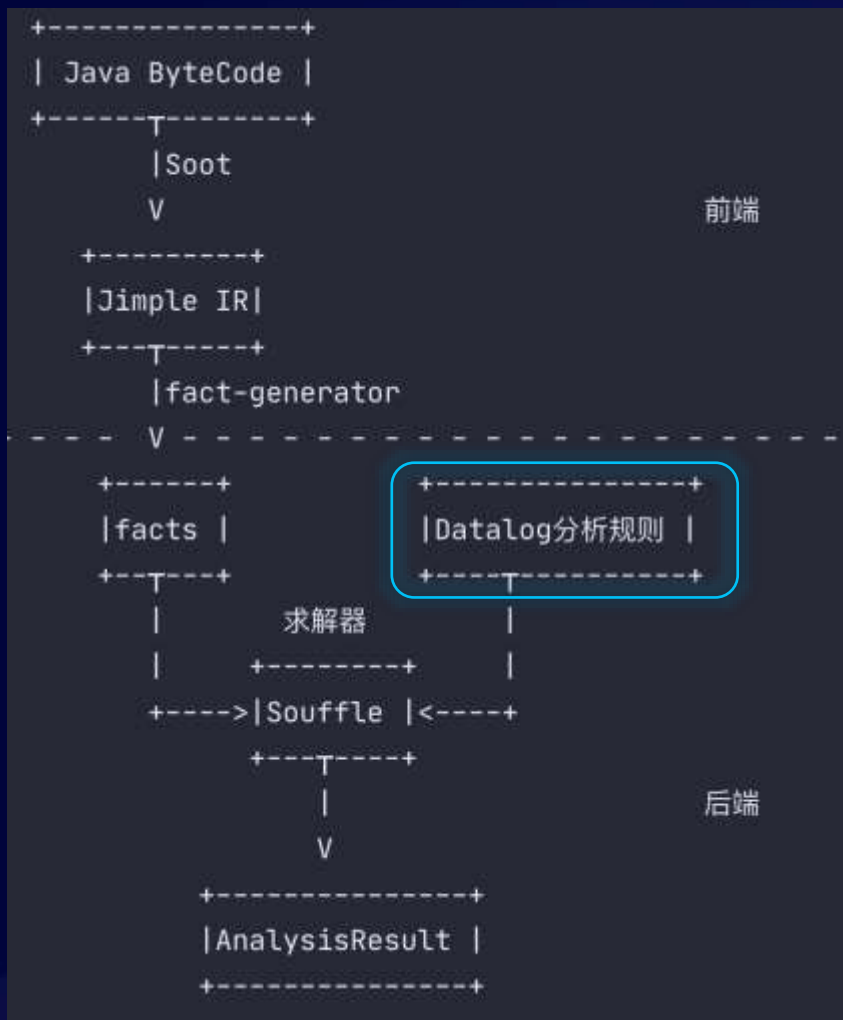
分析规则的例子:

```
VarPointsTo(p, o):-
    New(p, o).

VarPointsTo(to, o):-
    Assign(to, from),
    VarPointsTo(from,
o).
```

Assign	
b	a
c	b

New	
a	o1





## 2. 引擎架构

facts例子:

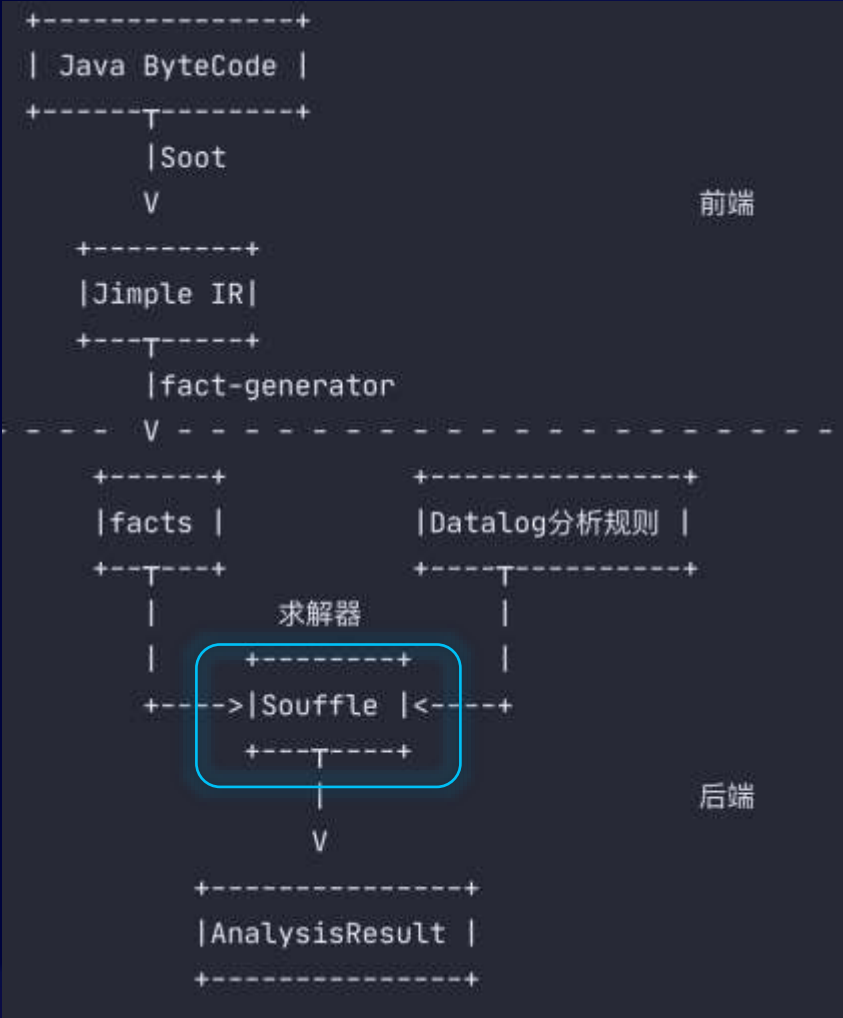
Assign	
b	a
c	b

New	
a	o1

分析规则的例子:

```
VarPointsTo(p, o):-  
  New(p, o).  
  
VarPointsTo(to, o):-  
  Assign(to, from),  
  VarPointsTo(from,  
  o).
```

VarPointsTo	
a	o1
b	o1
c	o1



## 2. 引擎架构

Why?

1. Datalog规则与指针分析规则天然契合, 高度一致, 可读性强
2. 将分析规则与求解过程进行解构, 专注于规则的编写
3. 有效利用souffle的高性能求解能力
4. 利用Dooop原有的反射处理规则.



*Soufflé*

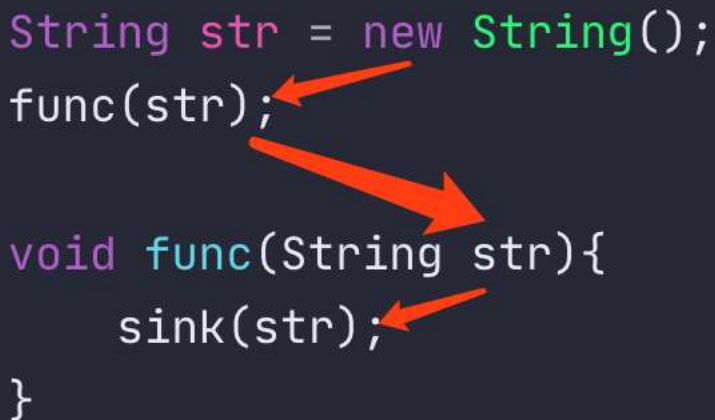


## 3.1 P/T Analysis算法

P/T Analysis核心: 为source生成污点对象, 把污点对象视为普通堆对象, 通过指针分析进行传播.

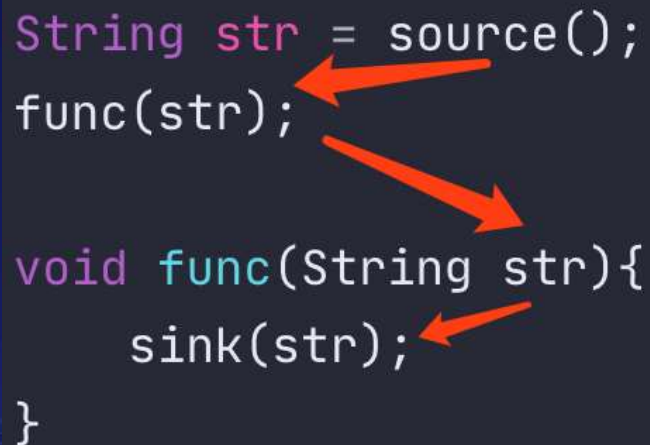
指针分析:

```
String str = new String();  
func(str);  
  
void func(String str){  
    sink(str);  
}
```



污点分析:

```
String str = source();  
func(str);  
  
void func(String str){  
    sink(str);  
}
```



## 3.1 P/T Analysis算法

P/T Analysis核心: 为source生成污点对象, 把污点对象视为普通堆对象, 通过指针分析进行传播.

优点:

1. 对指针分析框架进行少量更改即可实现P/T Analysis
2. 一次分析即可同时完成指针分析与污点分析, 效率更高
3. 污点分析可天然地复用指针分析规则



## 3.2 taint transfer

污点字符串相加的例子:

```
String str = source();

// 等价于"JDSec"+str
StringBuffer buffer = new StringBuffer();
buffer.append("JDSec");
buffer.append(str);
String res = buffer.toString();

sink(res);
```

```
class String{
    private char[] value;
    ...
}
```

```
class StringBuffer {
    private char[] value;
    private int idx;

    void append(String str){
        // char复制
        for(int i=0; i<str.length; i++){
            value[idx] = str.value[i];
            idx++;
        }
    }
}
```



## 3.2 taint transfer

污点字符串相加的例子:

```
String str = source();

// 等价于"JDSec"+str
StringBuffer buffer = new StringBuffer();
buffer.append("JDSec");
buffer.append(str);
String res = buffer.toString();

sink(res);
```

```
class String{
    private char[] value;
    ...
}
```



```
class StringBuffer {
    private char[] value;
    private int idx;

    void append(String str){
        // char复制
        for(int i=0; i<str.length; i++){
            value[idx] = str.value[i];
            idx++;
        }
    }
}
```



## 3.2 taint transfer

污点字符串相加的例子:

```
String str = source();  
  
// 等价于"JDSec"+str  
StringBuffer buffer = new StringBuffer();  
buffer.append("JDSec");  
buffer.append(str);  
String res = buffer.toString();  
  
sink(res);
```

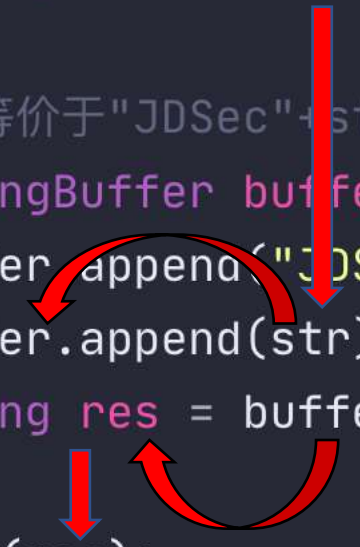
```
class String{  
    private char[] value;  
    ...  
}
```

```
class StringBuffer {  
    private char[] value;  
    private int idx;  
  
    void append(String str){  
        // char复制  
        for(int i=0; i<str.length; i++){  
            value[idx] = str.value[i];  
            idx++;  
        }  
    }  
}
```

## 3.2 taint transfer

污点字符串相加的例子:

```
String str = source();  
  
// 等价于"JDSec"+str  
StringBuffer buffer = new StringBuffer();  
buffer.append("JDSec");  
buffer.append(str);  
String res = buffer.toString();  
sink(res);
```



The diagram illustrates the flow of taints in the provided code. A red arrow points from the variable `str` in the first line to the `append(str)` call in the fifth line. Another red arrow points from the `toString()` call in the sixth line to the variable `res` in the same line. A curved red arrow also points from the `append(str)` call to the `toString()` call, indicating the intermediate state of the `StringBuffer` object.

Param To Base:

若str指向污点对象,  
则buffer指向StringBuffer类型的污点对象

Base To Ret:

若base指向污点对象  
则res指向String类型的污点对象

## 3.3 taint transfer的局限性

复杂字符串操作的例子:

```
String str = source();

// 获取字符数组
char[] arr = new char[8];
str.getChars(0, 8, arr, 0);

// 数组负责
char arr2 = new char[8];
arr2[0] = arr[0];

// 重新生成字符串
String str2 = new String(arr2);
sink(str2);
```

taint transfer生成的污点对象是一个空壳

arr指向char[]类型的污点对象  
但是arr[\*]没有指向任何对象, 导致污点断流

## 3.3 taint transfer的局限性

taint transfer可能会生产各种类型的污点对象

```
// 对于调用: list.add(ele);    如果ele指向污点对象, 则list指向LinkedList类型的污点对象  
ParamToBaseTaintTransferMethod(0, "<java.util.LinkedList: boolean add(java.lang.Object)>").  
  
// 对于调用 ele=list.get(0);    如果list指向污点对象, 则ele指向Object类型的污点对象  
BaseToRetTaintTransferMethod("<java.util.LinkedList: java.lang.Object get(int)>").
```

Iterator?  
HashMap?  
Pojo?  
Pojo.field?  
.....





## 3.4 access path问题

问题的本质: 创建污点对象时如何处理对象内部的字段?

- |                  |        |   |
|------------------|--------|---|
| 1. 不处理内部字段       | 漏报     | X |
| 2. 内部所有字段都指向污点对象 | 误报     | X |
| 3. 部分字段指向污点对象    | 规则过于复杂 | X |



## 3.4 access path问题

思路: 只生成String相关类型的污点对象, 并为其字段/索引添加指向关系

```
501 // java.sql.Connection中的sink方法
502 LeakingSinkMethodArg("sql", 0, "<java.sql.Connection: java.sql.PreparedStatement prepareStatement(java.lang.String,int[])>").
503 LeakingSinkMethodArg("sql", 0, "<java.sql.Connection: java.sql.PreparedStatement prepareStatement(java.lang.String)>").
504 LeakingSinkMethodArg("sql", 0, "<java.sql.Connection: java.sql.PreparedStatement prepareStatement(java.lang.String,java.lang.S
505 LeakingSinkMethodArg("sql", 0, "<java.sql.Connection: java.sql.PreparedStatement prepareStatement(java.lang.String,int,int,int
506 LeakingSinkMethodArg("sql", 0, "<java.sql.Connection: java.sql.PreparedStatement prepareStatement(java.lang.String,int)>").
507 LeakingSinkMethodArg("sql", 0, "<java.sql.Connection: java.sql.PreparedStatement prepareStatement(java.lang.String,int,int)>").
508 LeakingSinkMethodArg("sql", 0, "<java.sql.Connection: java.sql.CallableStatement prepareCall(java.lang.String,int,int)>").
509 LeakingSinkMethodArg("sql", 0, "<java.sql.Connection: java.sql.CallableStatement prepareCall(java.lang.String)>").
510 LeakingSinkMethodArg("sql", 0, "<java.sql.Connection: java.sql.CallableStatement prepareCall(java.lang.String,int,int,int)>").
511
```

大部分情况下:

- source()只会生成String类型的污点对象
- sink()只会接受String或者String相关泛型的污点对象

## 3.4 access path问题

思路: 只生成String相关类型的污点对象, 并为其字段/索引添加指向关系

```
String str = source();
```

```
// 获取字符数组
```

```
char[] arr = new char[8];
```

```
str.getChars(0, 8, arr, 0);
```

```
// 数组复制
```

```
char[] arr2 = new char[8];
```

```
arr2[0] = arr[0];
```

```
// 重新生成字符串
```

```
String str2 = new String(arr2);
```

```
sink(str2);
```

char[]类型的对象

数组索引指针arr[\*]

char类型的污点对象

## 3.4 access path问题

```
class LinkedList{
    String[] arr;

    public void add(String ele){
        this.arr[0] = ele;
    }

    public String get(int idx){
        return this.arr[idx];
    }
}

// 创建LinkedList对象
LinkedList list = new LinkedList();    // list->{ listObj }
list.arr = new Object[1];    // listObj.<arr> -> { strArrObj }

// 污点对象传播
String str = source();    // str指向污点对象taintObj
list.add(str);
sink(list.get(0));
```

LinkedList只是传播污点对象的桥梁  
自身不会作为污点对象



## 3.4 access path问题

总结: 分类处理

1. 限制taint transfer的适用范围: 只生成String操作相关类对象, 以简化access path问题
2. 通过mock JDK中的代码处理容器类, 平等地传播普通堆对象与污点对象.





## 4. 上下文策略改进

主流上下文敏感策略

1. Call-Site Sensitivity: 使用方法调用点作为callee的上下文元素
2. Object Sensitivity: 使用方法调用的receiver object作为callee的上下文元素
3. Type Sensitivity: 使用创建receiver object的方法所在的类作为callee的上下文元素



## 4. 上下文策略改进

实际执行时:  
整个调用栈构成了F6()的上下文

```
↩ F6:34, Test (org.example)
  F5:31, Test (org.example)
  F4:27, Test (org.example)
  F3:23, Test (org.example)
  F2:20, Test (org.example)
  F1:17, Test (org.example)
  main:38, Test (org.example)
```

静态分析时:  
只能选取有限元素作为F6()的上下文

```
[
  <F3:23>,
  <F4:27>,
  <F5:31>
]
```

*3-callsite context sensitivity*

## 4.1 更有区分性的上下文元素

应当在有限的上下文长度中有保留更有区分性的上下文元素



## 4.1 更有区分性的上下文元素

1-obj sensitivity的例子:

```
class T1{
    void doGet(HttpServletRequest req1, ...){
        Cookie cookie1 = req1.getCookie("...");
        String value1 = cookie1.getValue();
        ...
    }
}

class T2{
    void doGet(HttpServletRequest req2){
        Cookie cookie2 = req2.getCookie("...");
        String value2 = cookie2.getValue();
        ...
    }
}
```

req1 -> [immuCtx]:reqObj1

cookie1 ->  
[reqObj1]:cookieObj

getValue()的上下文:  
~~[reqObj1]~~, cookieObj]

## 4.1 更有区分性的上下文元素

1-obj sensitivity的例子:

```
class T1{
    void doGet(HttpServletRequest req1, ...){
        Cookie cookie1 = req1.getCookie("...");
        String value1 = cookie1.getValue();
        ...
    }
}
```

req1 -> [immuCtx]:reqObj1

cookie1 ->  
[reqObj1]:cookieObj  
getValue()的上下文:

~~[reqObj1]~~, cookieObj]

```
class T2{
    void doGet(HttpServletRequest req2){
        Cookie cookie2 = req2.getCookie("...");
        String value2 = cookie2.getValue();
        ...
    }
}
```

req2 -> [immuCtx]:reqObj2

cookie2 ->  
[reqObj2]:cookieObj

getValue()的上下文:

~~[reqObj2]~~, cookieObj]



## 4.1 更有区分性的上下文元素

1-obj sensitivity的例子:

```
class T1{
    void doGet(HttpServletRequest req1, ...){
        Cookie cookie1 = req1.getCookie("...");
        String value1 = cookie1.getValue();
        ...
    }
}

class T2{
    void doGet(HttpServletRequest req2){
        Cookie cookie2 = req2.getCookie("...");
        String value2 = cookie2.getValue();
        ...
    }
}
```

req1-> [immuCtx]:reqObj1

cookie1->  
[reqObj1]:cookieObj

getValue()的上下文为[cookieObj]

req2-> [immuCtx]:reqObj2

cookie2->  
[reqObj2]:cookieObj

getValue()的上下文为[cookieObj]

*confused*

## 4.1 更有区分性的上下文元素

1-obj sensitivity的例子:

```
class T1{
    void doGet(HttpServletRequest req1, ...){
        Cookie cookie1 = req1.getCookie("...");
        String value1 = cookie1.getValue();
        ...
    }
}
```

```
class T2{
    void doGet(HttpServletRequest req2){
        Cookie cookie2 = req2.getCookie("...");
        String value2 = cookie2.getValue();
        ...
    }
}
```

req1-> [immuCtx]:reqObj1

cookie1->  
[reqObj1]:cookieObj

更有区分性的上下文元素

req2-> [immuCtx]:reqObj2

cookie2->  
[reqObj2]:cookieObj

## 4.1 更有区分性的上下文元素

1-obj sensitivity的例子:

```
class T1{
    void doGet(HttpServletRequest req1, ...){
        Cookie cookie1 = req1.getCookie("...");
        String value1 = cookie1.getValue();
        ...
    }
}

class T2{
    void doGet(HttpServletRequest req2){
        Cookie cookie2 = req2.getCookie("...");
        String value2 = cookie2.getValue();
        ...
    }
}
```

req1 -> [immuCtx]:reqObj1

cookie1 ->  
[reqObj1]:cookieObj  
getValue()的上下文:

[reqObj1, ~~cookieObj~~]

## 4.1 更有区分性的上下文元素

1-obj sensitivity的例子:

```
class T1{
    void doGet(HttpServletRequest req1, ...){
        Cookie cookie1 = req1.getCookie("...");
        String value1 = cookie1.getValue();
        ...
    }
}
```

req1 -> [immuCtx]:reqObj1

cookie1 ->  
[reqObj1]:cookieObj  
getValue()的上下文:

[reqObj1, ~~cookieObj~~]

```
class T2{
    void doGet(HttpServletRequest req2){
        Cookie cookie2 = req2.getCookie("...");
        String value2 = cookie2.getValue();
        ...
    }
}
```

req2 -> [immuCtx]:reqObj2

cookie2 ->  
[reqObj2]:cookieObj  
getValue()的上下文:

[reqObj2, ~~cookieObj~~]

## 4.1 更有区分性的上下文元素

1-obj sensitivity的例子:

```
class T1{
    void doGet(HttpServletRequest req1, ...){
        Cookie cookie1 = req1.getCookie("...");
        String value1 = cookie1.getValue();
        ...
    }
}
```

req1 -> [immuCtx]:reqObj1

cookie1 -> [reqObj1]:cookieObj  
getValue()的上下文为[reqObj1]

**distinguish**

```
class T2{
    void doGet(HttpServletRequest req2){
        Cookie cookie2 = req2.getCookie("...");
        String value2 = cookie2.getValue();
        ...
    }
}
```

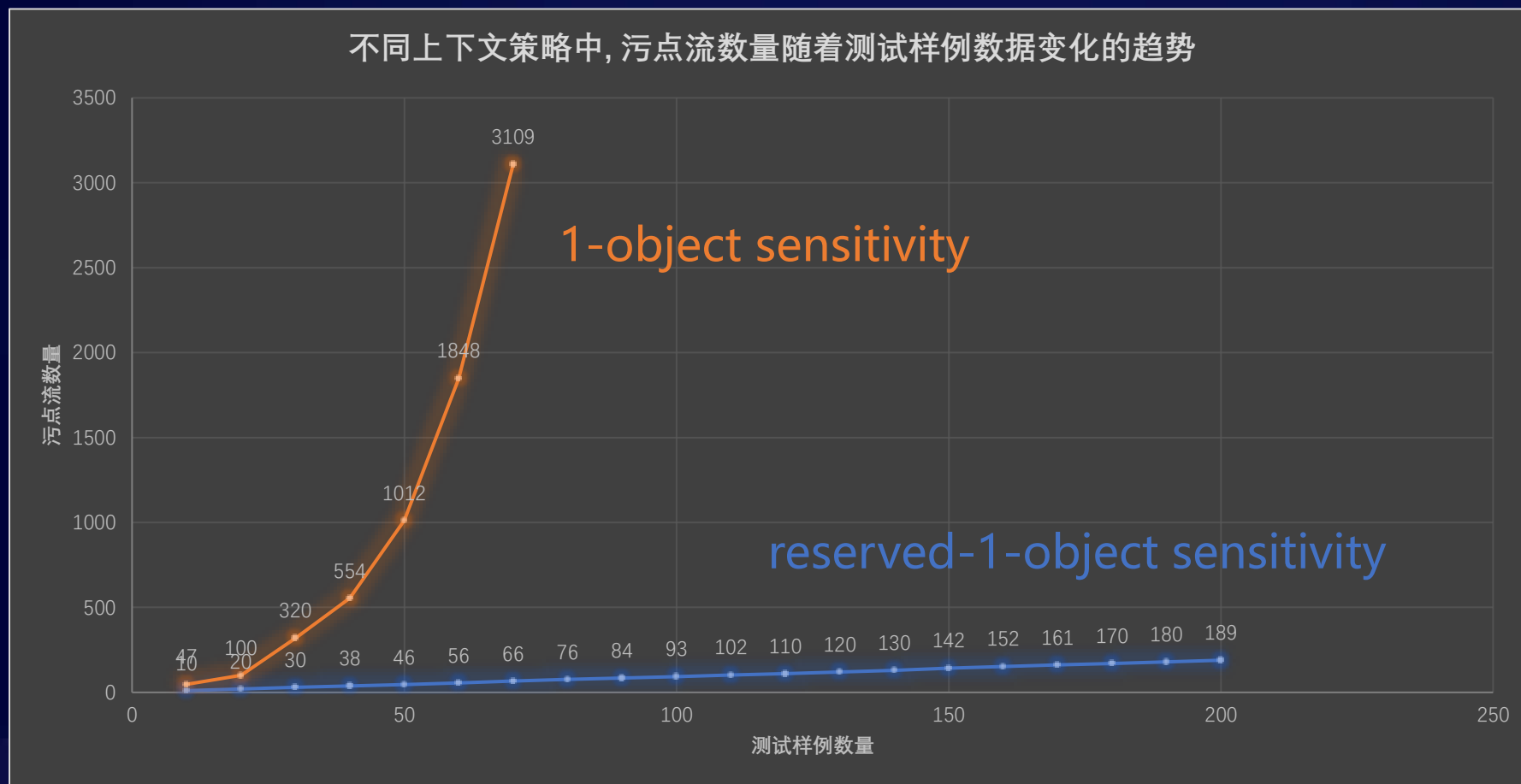
req2 -> [immuCtx]:reqObj2

cookie2 -> [reqObj2]:cookieObj  
getValue()的上下文为[reqObj2]

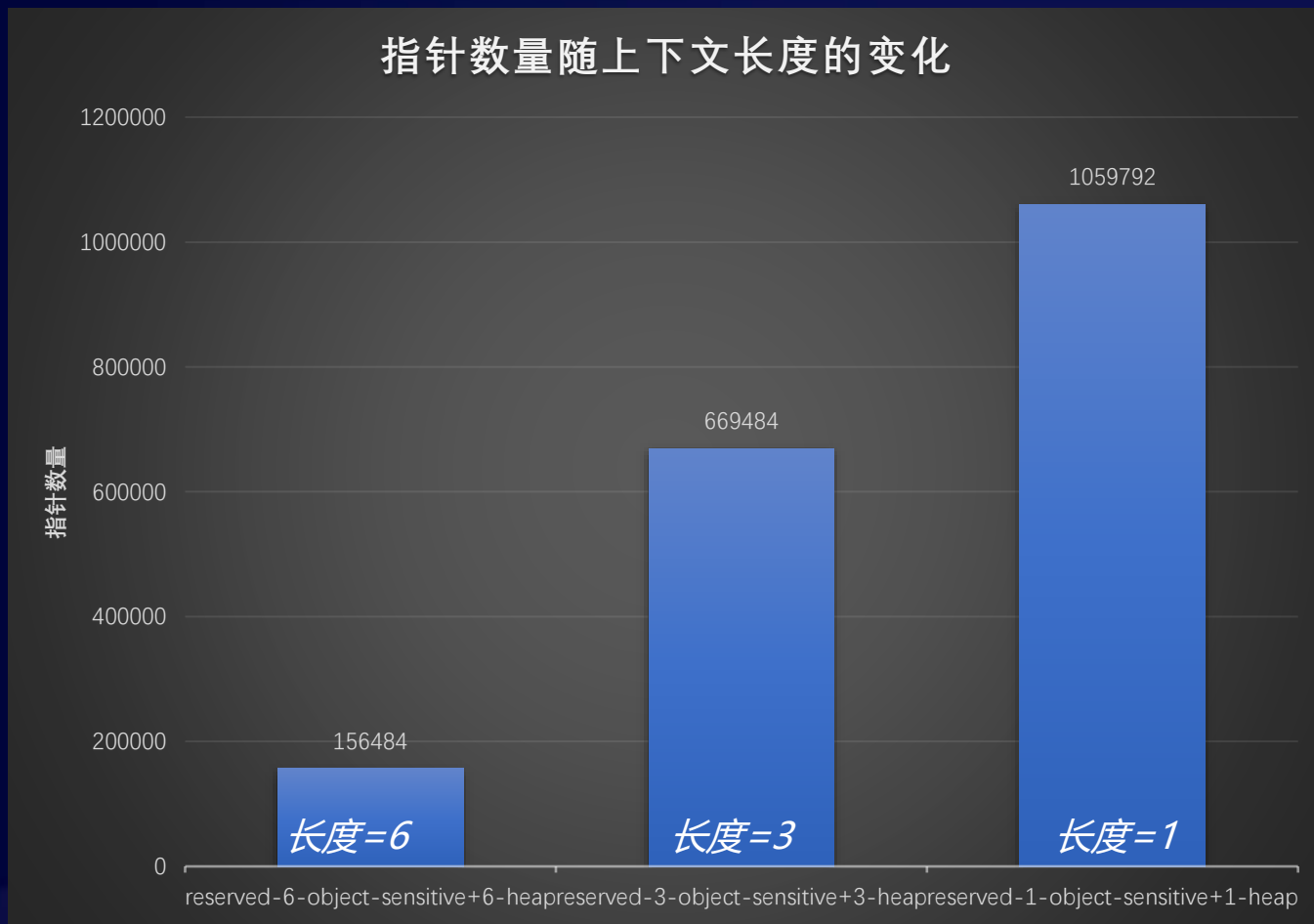


## 4.1 更有区分性的上下文元素

保留入口类相关的相关上下文元素(eg: 入口类对象, 入口方法), 可有效避免数据流混淆



## 4.2 更长的上下文长度



上下文长度

指针指向关系数量  
分析耗时  
污点流数量



## 4.2 更长的上下文长度



## 4.2 更长的上下文长度

总结:

1. “上下长度减少, 要分析的指针减少” 的成立前提是指针分析
2. 应当增加上下文长度, 尽量避免污点流混淆, 反而会减少指针数量



## 5. Spring框架适配



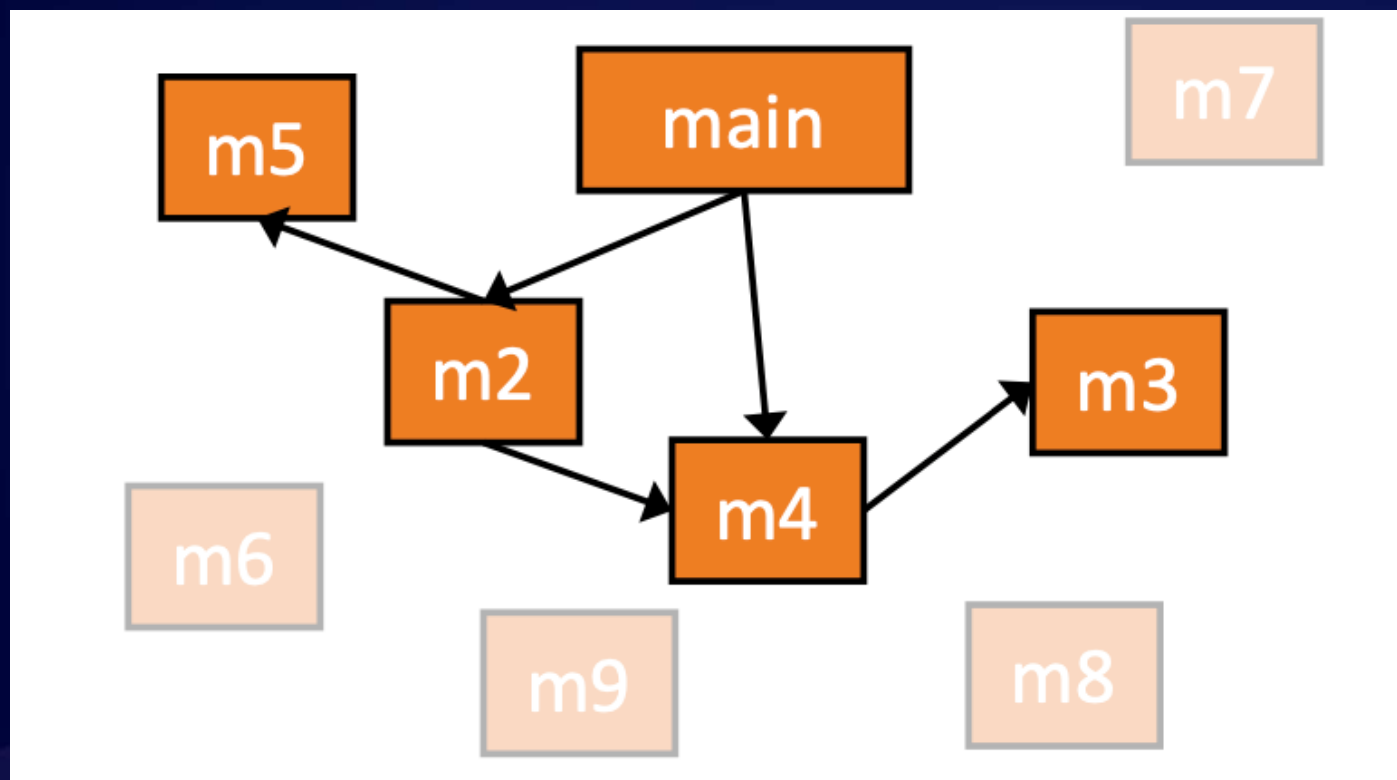
## 5.1 隐式可达方法丢失问题





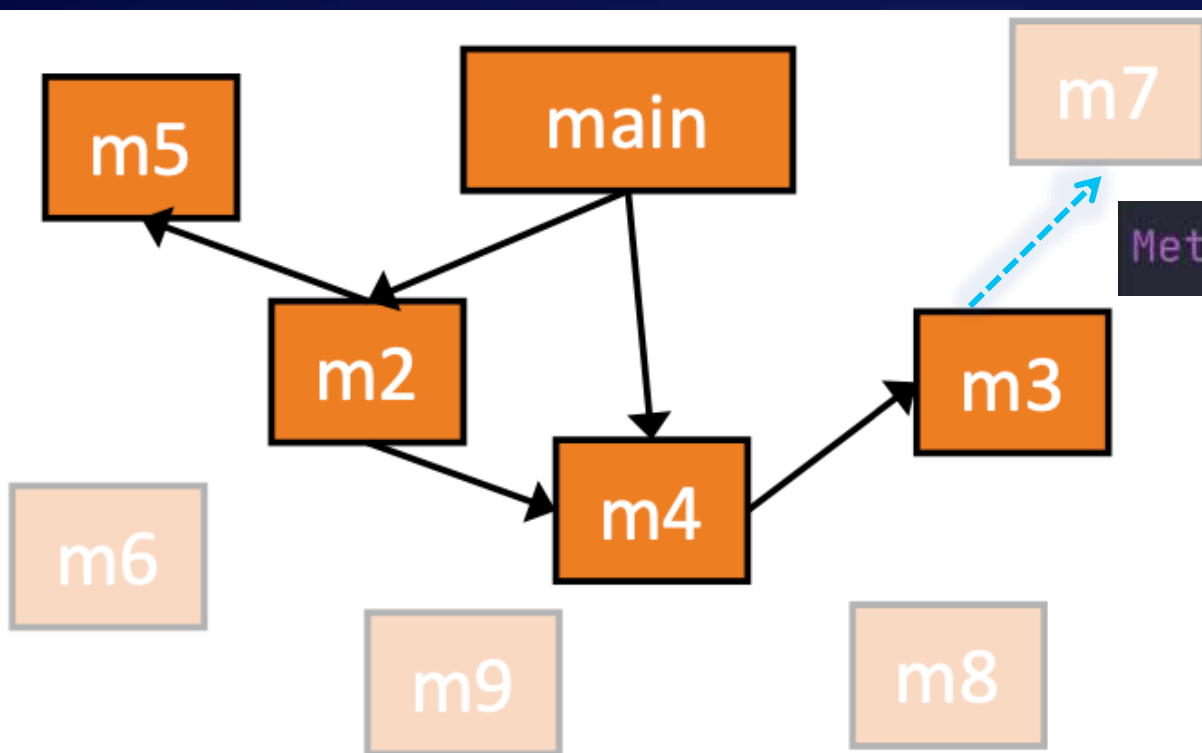
## 5.1 隐式可达方法丢失问题

soot会进行简单的指针分析, 识别出可达方法. 只为可达方法生成IR



## 5.1 隐式可达方法丢失问题

隐式可达类丢失问题: soot可能会认为某些实际可达的方法不可达

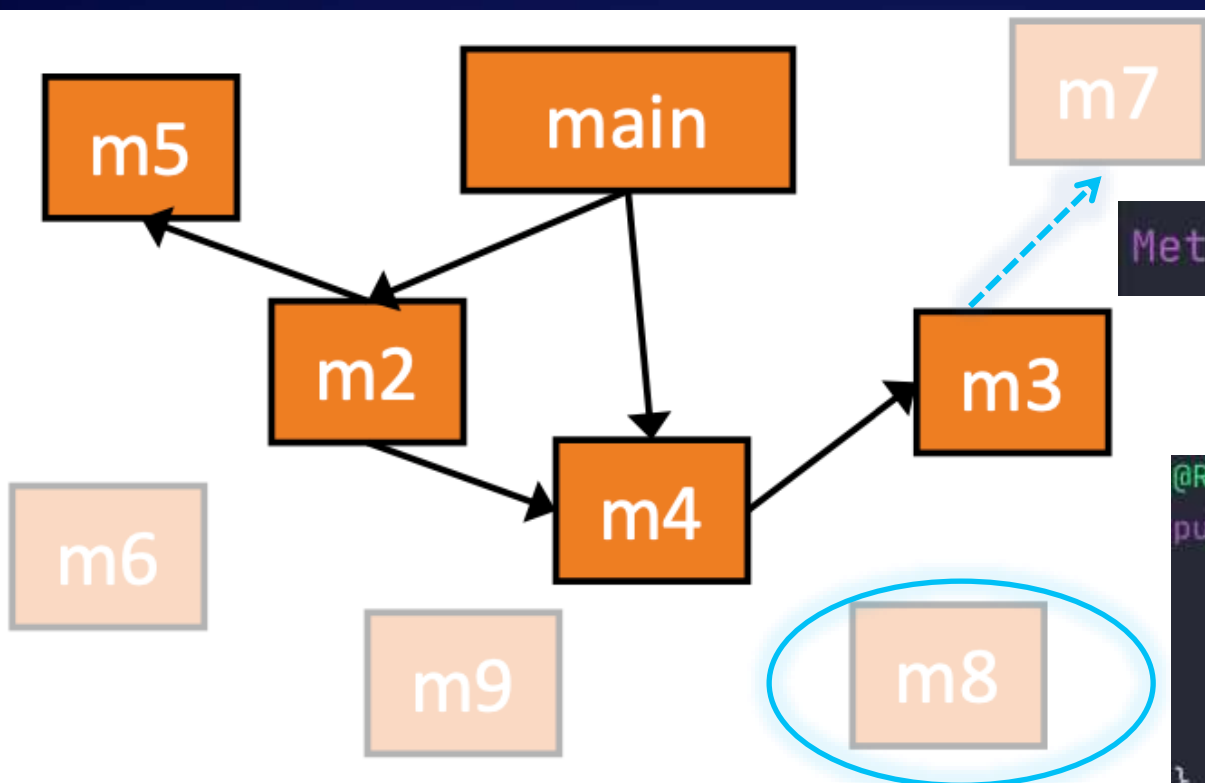


反射方法引用

```
Method m = clazz.getMethod("m7", ...);
```

## 5.1 隐式可达方法丢失问题

隐式可达类丢失问题: soot可能会认为某些实际可达的方法不可达



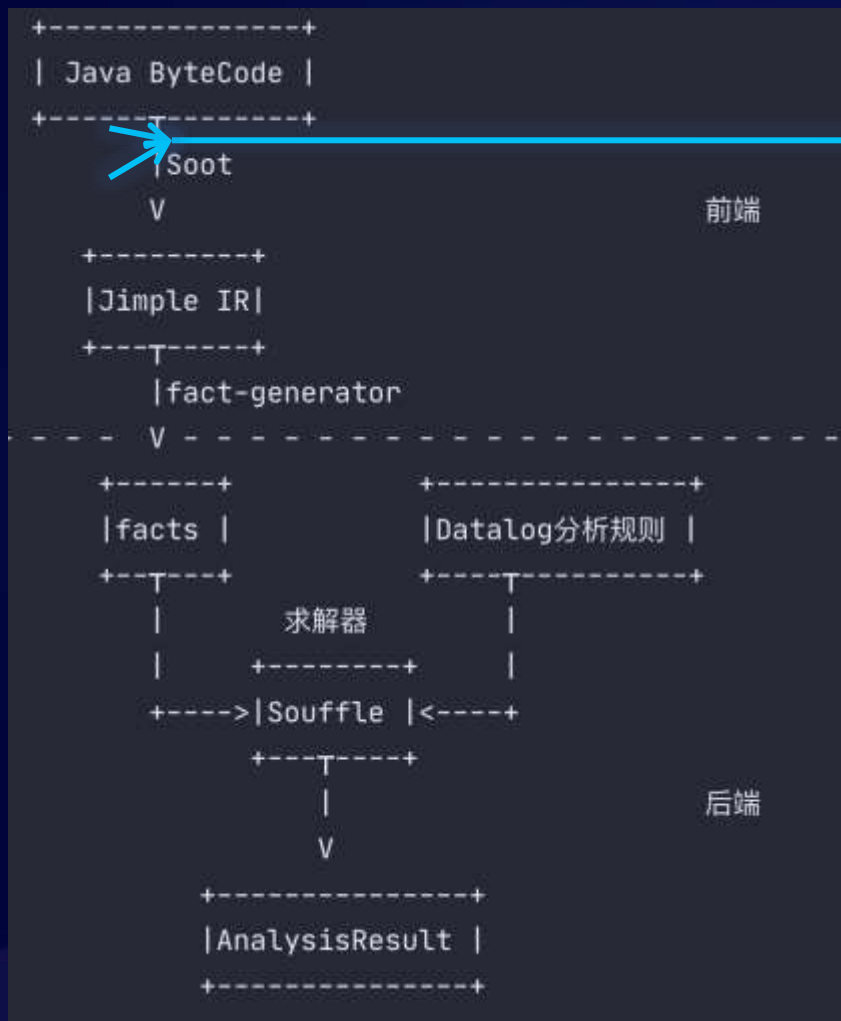
反射方法引用

```
Method m = clazz.getMethod("m7", ...);
```

spring框架入口方法

```
@RestController
public class Test{
    @PostMapping(value = "/m8")
    public String m8(@RequestParam String jsonStr) {
        ...
    }
}
```

## 5.1 隐式可达方法丢失问题



预分析的目的: 分析出隐式可达方法作为后面反编译的入口点

## 5.1 隐式可达方法丢失问题



预分析的目的: 分析出隐式可达方法作为后面反编译的入口点

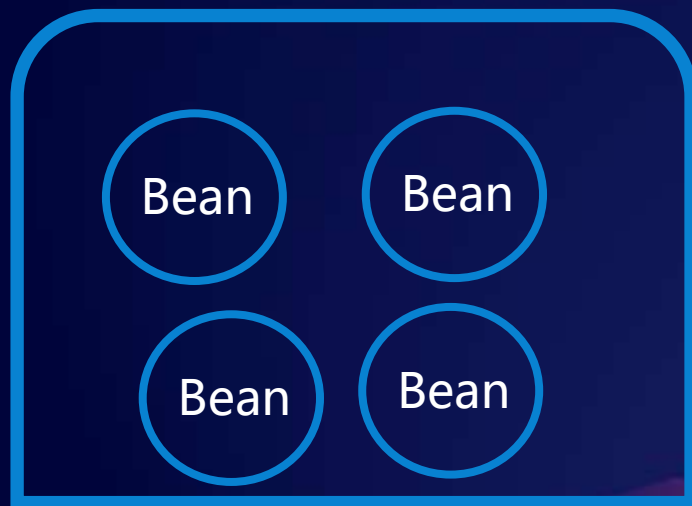
识别隐式可达类

1. spring框架访问: 所有具有spring注解的类
2. 反射访问: 所有字符串常量中指定的类

## 5.2 Bean容器

动态执行

静态分析



Spring框架

编写规则模拟Spring框架的行为:

1. 基于注解识别与创建Bean
2. 基于注解处理Bean之间的依赖注入:
  - 字段+@Autowired
  - 构造方法+@Autowired
  - setter方法+@Autowired



## 5.3 为入口参数添加污点对象

```
@RestController
@RequestMapping(value = "/api")
public class T3 {
    @PostMapping(value = "/login")
    public String login(@RequestBody User user) {
        ...
    }
}
```

```
public class User {
    public String name;
    public int age;

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

User类型的污点对象



## 5.3 为入口参数添加污点对象

```
@RestController
@RequestMapping(value = "/api")
public class T3 {
    @PostMapping(value = "/login")
    public String login(@RequestBody User user) {
        ...
    }
}
```

```
public class User {
    public String name;
    public int age;

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

String类型的污点对象

## 5.3 为入口参数添加污点对象

```
@RestController
@RequestMapping(value = "/api")
public class T3 {
    @PostMapping(value = "/login")
    public String login(@RequestBody User user) {
        ...
    }
}
```

```
public class User {
    public String name;
    public int age;

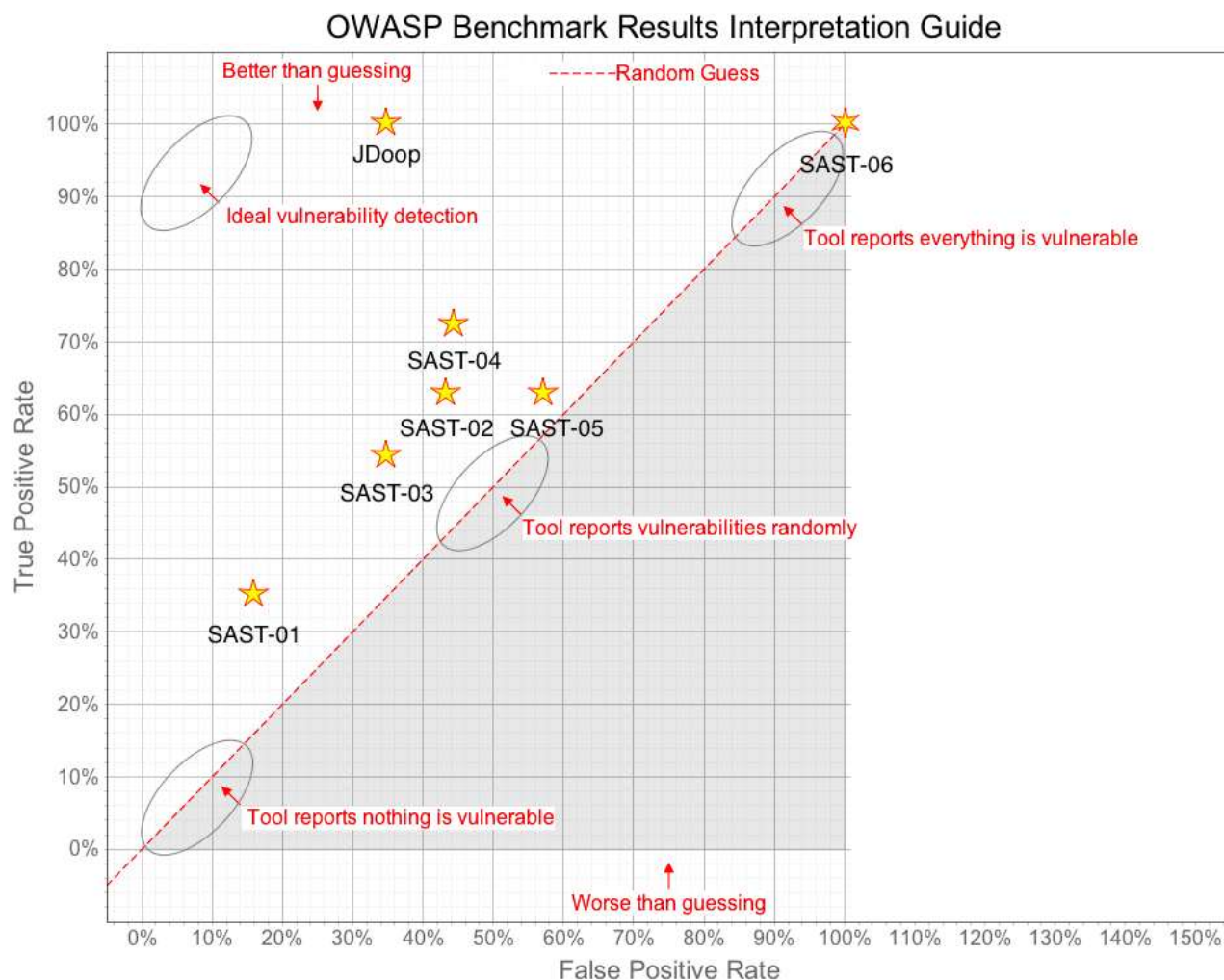
    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

String类型的污点对象

支持的入口参数类型:

- String, String[], char[]
- Map<String, String>
- List<String>
- Set<String>
- JSONObject
- Pojo

## 6. Benchmark



测试样本: Owasp benchmark V1.1命令注入样本  
共计2708个

资源消耗:

实际内存峰值使用14G

facts生成耗时: 71s

污点流分析时: 148s (24核CPU)

准确率: 84.80%

召回率: 100.00%

Score: 64.35%

# JDoop使用流程



# JDoop特点

01

源码解析 “可选”

02

易编写多实现的  
Sink点规则

03

污点流 “优化”



## 源码解析 “可选性”

```
jeecg-system-start-3.5.3.jar
├── BOOT-INF
│   ├── classes
│   └── lib
│       ├── alipay-sdk-java-3.1.0.jar
│       ├── antlr4-runtime-4.7.jar
│       ├── asm-5.0.3.jar
│       └── ...
```

```
Configuration var1 = new Configuration();
String Content = "${1+1}";
StringWriter var2 = new StringWriter();
try{
    Template t = new Template("template", new
        StringReader(Content), var1);
    t.process(null, var2);
    System.out.println(var2);
}catch (Exception e){
    //do nothing
}
```

- 支持三方依赖包的解析
- 三方依赖包的解析可选

String Content



t.process



freemarker.Execute



runtime.exec()

# 易编写多实现的Sink点规则

```
cmdi 0 <java.lang.Runtime: java.lang.Process exec(java.lang.String)>
```

## 一对一Sink规则

- 第一列标签
- 第二列污点参数
- 第三列污点函数

- 第一列标签
- 第二列污点类
- 第三列污点函数
- 第四列污点参数

## 一(多)对多Sink规则

```
cmdi java.lang.Runtime exec 0
```

# 污点流 “优化” - 反射调用

```
public void doPost(HttpServletRequest request, HttpServletResponse response) {  
    String cmd = request.getParameter("cmd");  
    try {  
        Class clazz = Class.forName("com.example.HelloServlet$Rce");  
        Rce r = (Rce) clazz.newInstance();  
        r.exec(cmd);  
    } catch (Exception e) {  
        //do nothing  
    }  
}  
  
class Rce {  
    public void exec(String command){  
        try {  
            Class clazz = Class.forName("java.lang.Runtime");  
            Method getRuntimeMethod = clazz.getMethod("getRuntime");  
            clazz.getMethod("exec", String.class).invoke(getRuntimeMethod.invoke(clazz),  
command);  
        } catch (Exception e) {  
            // do nothing  
        }  
    }  
}
```

request.getParameter("cmd")



sink: Runtime.exec()

## 污点流 “优化” -Log4j2

```
String payload = "${jndi:ldap://127.0.0.1/evil};
```



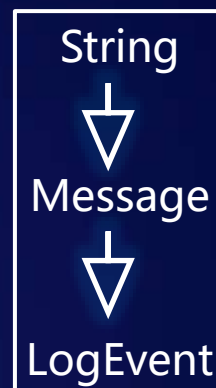
```
Message data = messageFactory.newMessage(payload);
```



```
LogEvent logEvent = createEvent(loggerName, marker, fqcn, location, level, data, (List)props, t);
```



```
this.log(logEvent, LoggerConfig.LoggerConfigPredicate.ALL);
```



# Jeecg-Boot挖掘

Sink规则: Freemarker SSTI Inject

freemarker.template.Template

<init> 1

```
public Result<?> c(@RequestBody JSONObject var1)
```



```
String var2 = var1.getString("sql");
```



```
Map var12 = this.reportDbService.parseReportSql(var2, var3, var4, var5)
```



```
(new Template("template", new StringReader(var0), var2)).process(var1, var3)
```

```
ic static String a(String var0, Map<String, Object> var1) {
    if (var0 == null) {...} else {
        Configuration var2 = new Configuration();
        var2.setNumberFormat("#.#####");
        var2.setSharedVariable( name: "func", new FunctionMethod());
        var1.put("jeecg", new FreemarkerMethod());
        var1.put("isEmpty", new NotEmptyMethod());
        var2.setClassicCompatible(true);
        StringWriter var3 = new StringWriter();

        try {
            a.debug("模板内容:{}", var0.toString());
            (new Template( name: "template", new StringReader(var0), var2)).process(var1, va
            a.debug("模板解析结果:{}", var3.toString());
        }
    }
}
```

存在[spring entry method param=>Freemarker SSTI Inject]类型的污点流:

污点对象来源: <org.jeecg.modules.jmreport.desreport.a.a: org.jeecg.modules.jmreport.common.vo.Result c(com.alibaba.fastjson.JSONObject)>/@parameter0

污点方法调用: <org.jeecg.modules.jmreport.desreport.render.utils.FreeMarkerUtils: java.lang.String a(java.lang.String,java.util.Map)>/freemarker.template.Template.<init>/0

污点方法调用参数: <org.jeecg.modules.jmreport.desreport.render.utils.FreeMarkerUtils: java.lang.String a(java.lang.String,java.util.Map)>/\$u5

存在[spring entry method param=>Freemarker SSTI Inject]类型的污点流:

污点对象来源: <org.jeecg.modules.jmreport.desreport.a.a: org.jeecg.modules.jmreport.common.vo.Result b(com.alibaba.fastjson.JSONObject)>/@parameter0

污点方法调用: <org.jeecg.modules.jmreport.desreport.render.utils.FreeMarkerUtils: java.lang.String a(java.lang.String,java.util.Map)>/freemarker.template.Template.<init>/0

污点方法调用参数: <org.jeecg.modules.jmreport.desreport.render.utils.FreeMarkerUtils: java.lang.String a(java.lang.String,java.util.Map)>/\$u5

# 使用感受

## 源码解析

有的工具不支持依赖包一起解析，有的则需要全部解析，动态选择解析有效提高了分析效率。

## 用户体验

各种操作还是基于命令行形式，无法通过点击某个具体污点流的方式跳转具体污点函数。

## 污点流传递

实现了反射调用推理的能力，及JDoop是基于先推出1和3成立再反推2实现数据流的还原，优化了access path问题。

## 计算资源

至少需要CPU 16核，运行内存64G  
对于比较大的项目需要更高的配置。