

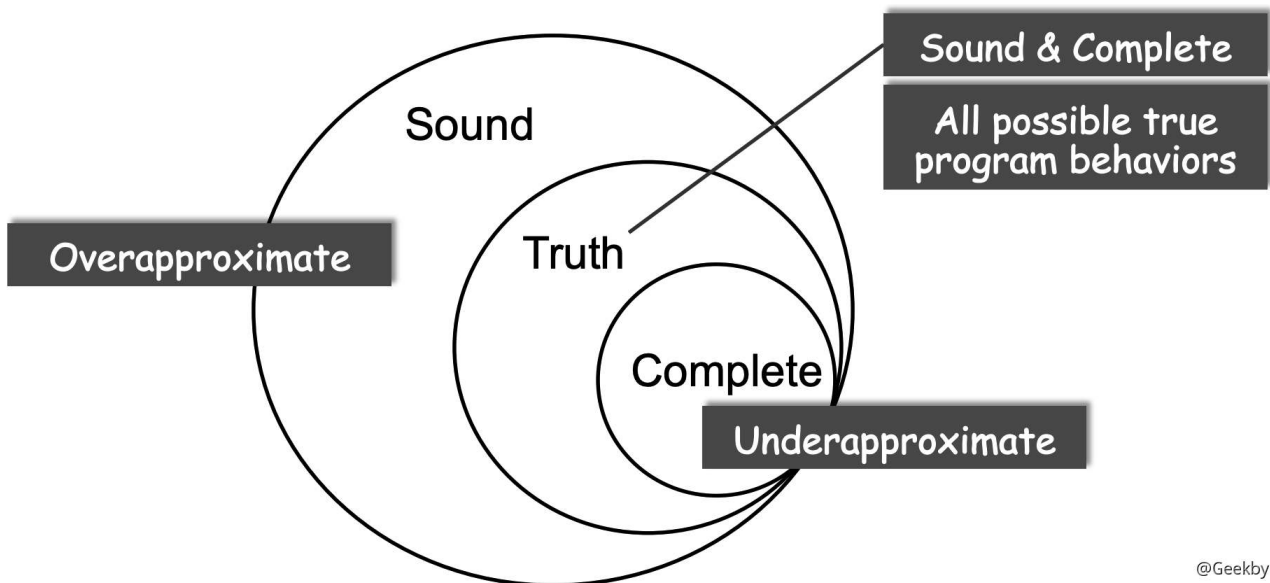
静态分析

给定一个输入程序，输出程序的属性。

Rice 定律

在一个即递归可枚举(recursively enumerable)语言中，任何程序行为的 **non-trivial** 属性都是不可解释的。**non-trivial** 属性指的是那些与程序运行行为有关的属性。

即不存在完美的静态分析。



国内课程：[北大熊英飞](#)、[南大](#)及南大B站的[视频](#)。

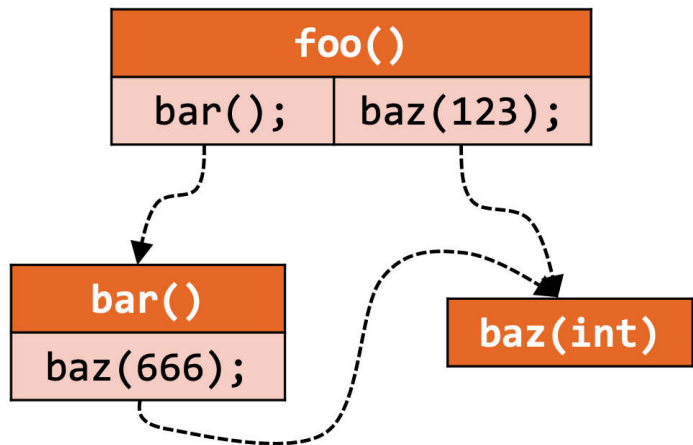
调用图

一个调用图就是从调用点到目标方法的一系列调用边。

```
void foo() {  
    bar();  
    baz(123);  
}
```

```
void bar(int x) {  
    baz(666);  
}
```

```
void baz() { }
```



算法

CHA

论文[地址](#)。

定义

- 需要首先获得整个程序的类继承关系图
- 通过接收变量的声明类型来解析 Virtual call
 - 接收变量的例子：在 `a.foo()` 中，`a` 就是接收变量
- 假设一个接收变量能够指向 `A` 或 `A` 的所有子类

具体过程

通过 CHA 算法寻找到某个程序调用点对应的可能的目标函数实体。

```
Resolve(cs)  
   $T = \{\}$   
   $m = \text{method signature at } cs$   
  if  $cs$  is a static call then  
     $T = \{ m \}$   
  if  $cs$  is a special call then  
     $c^m = \text{class type of } m$   
     $T = \{ \text{Dispatch}(c^m, m) \}$   
  if  $cs$  is a virtual call then  
     $c = \text{declared type of receiver variable at } cs$   
    foreach  $c'$  that is a subclass of  $c$  or  $c$  itself do  
      add  $\text{Dispatch}(c', m)$  to  $T$   
  return  $T$ 
```

@Geekby

- call site(cs) 就是调用语句，m(method) 就是对应的函数签名。
- T 集合中保存找到的结果

static call

静态方法调用前写的是类名，而非静态方法调用前写的是变量或指针名。静态方法调用不需要依赖实例，所以直接加到集合 T 中。

special call

`special call` 主要分为三种情况。

第一种使用 `super` 类的调用方法。`foo()` 虽然在当前类有定义，但是实际使用的是父类的 `foo()`，因此需要使用 `Dispatch` 函数。其中的 `foo()` 的签名 `m` 由编译器返回信息可知是父类 `B` 的，那么获取 `foo()` 返回值的 `c` 也指向 `B`，也就相当于在父类中查找了。

virtual call

这是 CHA 区别于其他算法的主要之处。该算法会对此方法做一个 `Dispatch(c,m)` 并将 `c` 的所有子集以及子集的子集全都做一次 `Dispatch(c', m)`。直观来看，可以分为两步，第一步是对本身做一次 `Dispatch`，看看当前类中是否有 `foo()`，没有的话就到父类中递归地找；第二步是在当前类地所有子集中找到所有的 `foo()`，然后将这些 `foo` 同第一步找到的 `foo` 全都加入 `T` 中。

Resolve(cs)

$T = \{ \}$ T: 目标函数集合

$m = \text{method signature at } cs$ m: 取出cs调用点的函数签名

if cs **is a** static call **then**

$T = \{ m \}$

(1) 若cs是调用静态函数，
则T就是该类A中的静态函数

if cs **is** special call **then**

$c^m = \text{class type of } m$

(2) 若cs是调用构造函数、
私有函数或者父类函数，则
递归查找父类（函数实际所
在的类）

$T = \{ \text{Dispatch}(c^m, m) \}$

if cs **is a** virtual call **then**

$c = \text{declared type of receiver variable at } cs$

foreach c' **that is a subclass of** c **or** c **itself** **do**

add $\text{Dispatch}(c', m)$ to T

return T (3) 若cs是调用virtual call，则根据变量c的声明类C，对C
和C所有子类递归查找

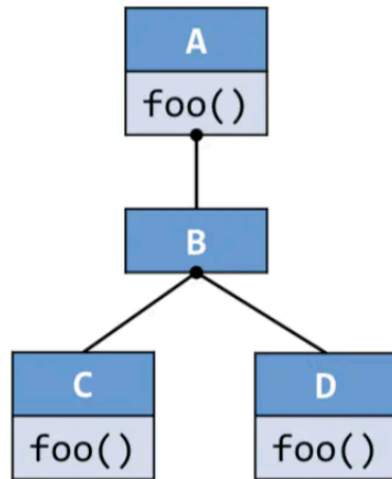
一个🍎:

```

class A {
    void foo() {...}
}
class B extends A {}

class C extends B {
    void foo() {...}
}
class D extends B {
    void foo() {...}
}

```



```

void resolve() {
    C c = ...
    c.foo();

    A a = ...
    a.foo();

    ➔ B b = new B();
    b.foo();
}

```

c的声明类型是C，C没有子类

$\text{Resolve}(c.\text{foo}()) = \{C.\text{foo}()\}$

a的声明类型是A，A和A所有的子类

$\text{Resolve}(a.\text{foo}()) = \{A.\text{foo}(), C.\text{foo}(), D.\text{foo}()\}$

B中没有foo()，要到B的父类A中查找

$\text{Resolve}(b.\text{foo}()) = \{A.\text{foo}(), \underline{C.\text{foo}()}, D.\text{foo}()\}$

Spurious call targets

CHA 的特征

1. 只考虑类继承结构，所以很快
2. 因为忽略了数据流和控制流的信息，所以不太准确

SPARK

论文[地址](#)。

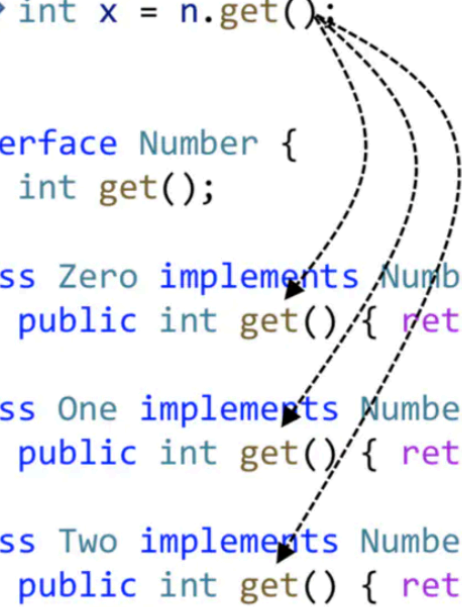
通过使用 `PointsTo Analysis` 找到变量的实际类型来准确地处理这种情况。与 `CHA` 相比，`SPARK` 去除了许多虚假边缘；但是，它要慢得多，并且可能会漏掉一些真正的边。

指针分析

指针分析要解决的问题

Problem of CHA

```
void foo() {  
    Number n = new One();  
    → int x = n.get();  
}  
  
interface Number {  
    int get();  
}  
  
class Zero implements Number {  
    public int get() { return 0; }  
}  
  
class One implements Number {  
    public int get() { return 1; }  
}  
  
class Two implements Number {  
    public int get() { return 2; }  
}
```



CHA: based on
class hierarchy

- 3 call targets

Constant propagation

- $x = \text{NAC}$

CHA根据声明类型来找可能的调用目标，根据Number找到3个，导致常量传播错误将x识别为非常量。实际上n仅指向One()，只会调用One.Number()。

程序中的指针指向哪个内存的问题，Java语言中的指针分析指的是一个指针指向程序中的哪个对象（Object）的问题。通常指针分析是一个 may-analysis，分析的结果通常是一个指针可能指向哪些对象。

指针分析的应用

- 可以用来计算其他基本信息（别名分析，调用图...）。
- 编译优化。
- 找Bug。
- 安全性分析。
-

指针分析是最基础的静态分析之一，也是很多其他分析的基础。

Soot

soot 有很多 Option 配置，可以参考[这里](#)或者[这里](#)。

IR

Jimple

介于 Java 和 Java 字节码之间，是一个基于语句的、类型化的（每个变量都有一个类型）和 3-addressed（每条语句最多有 3 个变量）的中间表示。

三地址码，一条指令的右侧最多只有一个运算符，如 $x+y*z$ 表示为：

```
t1 = y*z;
t2 = x + t1;
x = t2;
```

调用方法：

- `specialinvoke`：用于调用构造方法、父类方法、私有方法。
- `virtualinvoke`：用于调用普通的成员方法，进行 `virtual dispatch`。
- `interfaceinvoke`：用于调用继承的接口的方法，不能做优化，需要检查是否实现了接口中的方法。
- `staticinvoke`：用于调用静态方法。

例子：

```
(1) r0 := @this: FizzBuzz
(2) i0 := @parameter0: int
(3) $i1 = i0 % 15
(4) if $i1 != 0 goto $i2 = i0 % 5
(5) $r4 = <java.lang.System: java.io.PrintStream out>
(6) virtualinvoke $r4.<java.io.PrintStream: void println(java.lang.String)>("FizzBuzz")
// 尖括号内的是方法签名
(7) goto [?= return]
(8) $i2 = i0 % 5
(9) if $i2 != 0 goto $i3 = i0 % 3
(10) $r3 = <java.lang.System: java.io.PrintStream out>
(11) virtualinvoke $r3.<java.io.PrintStream: void println(java.lang.String)>("Buzz")
(12) goto [?= return]
(13) $i3 = i0 % 3
(14) if $i3 != 0 goto $r1 = <java.lang.System: java.io.PrintStream out>
(15) $r2 = <java.lang.System: java.io.PrintStream out>
(16) virtualinvoke $r2.<java.io.PrintStream: void println(java.lang.String)>("Fizz")
(17) goto [?= return]
(18) $r1 = <java.lang.System: java.io.PrintStream out>
(19) virtualinvoke $r1.<java.io.PrintStream: void println(int)>(i0)
(20) return
```

Shimple

与 Jimple 基本相同，是 Jimple 静态单任务形式的中间表示。

Baf

是流线型的基于栈的字节表示。将 `java` 字节码转为基于栈的代码。

Grimp

与 `Jimple` 类似，比 `Jimple` 更接近于 `java` 源码。容易阅读，方便人工阅读。

数据结构

基本的数据结构如下：

- `Scene`，是一个单例类，表示所分析的环境。
- `SootClass`，用于表示 `Scene` 中的类。
- `SootMethod`，用于表示 `SootClass` 中的方法。
- `Body`，使用 `Body` 来访问 `SootMethod` 中的各种信息，每个 `Body` 里面有三个主链，分别是 `Units` 链、`Locals` 链、`Traps` 链。
 - `Local`，方法内的局部变量。
 - `Trap`，方法内的异常处理。
 - `Unit`，方法体内的语句。

还有四种 `Body`，对应四种中间表示：`BafBody`、`JimpleBody`、`ShimpleBody`、`GrimpBody`，`Soot` 中的默认 IR 是 **`Jimple`**（Java Simple）。

Unit (Jimple中是Stmt)

表示 `Unit` 语句。主要有以下几种：

- 核心语句：`NopStmt`，`DefinitionStmt` (`IdentityStmt`，`AssignStmt`)。
 - `IdentityStmt`：通常指的是对变量赋值，这个变量既可以是显式的也可以是隐式的。
 - 一个 `IdentityStmt` 将特殊值，如参数、`this` 或被捕获的异常，分配给一个 `Local`。
 - 所有正常的赋值，例如从一个 `Local` 到另一个 `Local`，或者从一个 `Constant` 到一个 `Local`，都是用 `AssignStmt` 表示的。
- 负责过程内控制流的语句：`IfStmt`，`GotoStmt`，`TableSwitchStmt`，`LookupSwitchStmt`。
- 负责过程间的控制流语句：`InvokeStmt`，`ReturnStmt`，`ReturnVoidStmt`。
- 监控语句：`EnterMonitorStmt`，`ExitMonitorStmt`。
- `ThrowStmt`，`RetStm`。

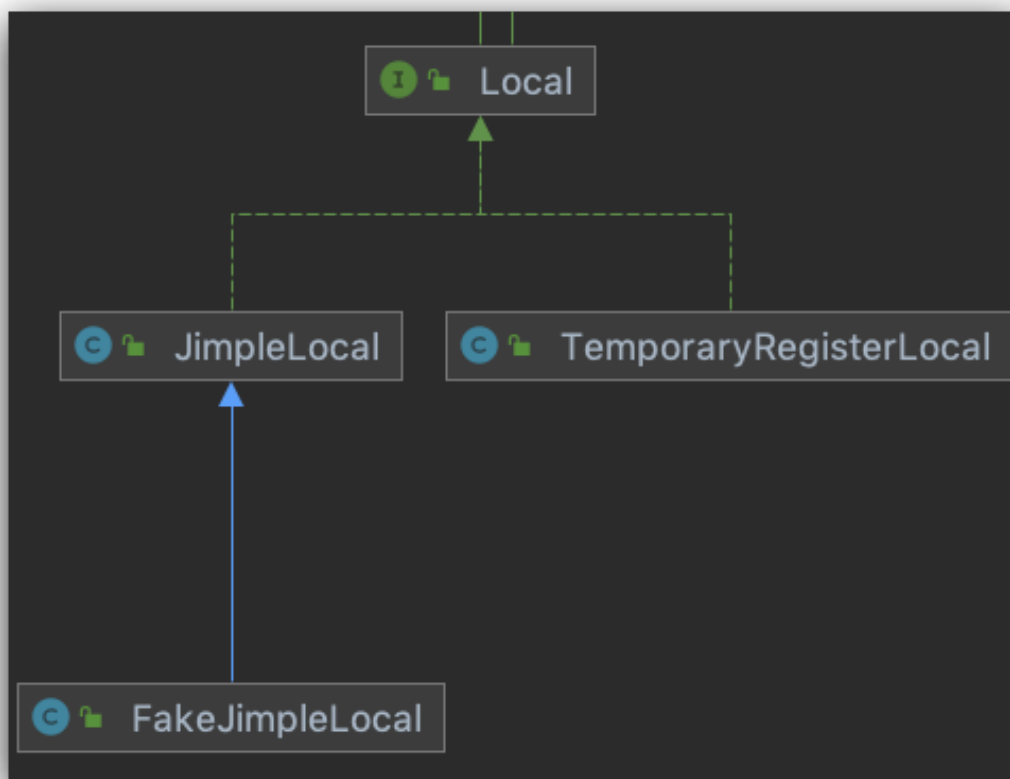
Value

- `Local`
- `Constant`
- `Ref`
- `Expr`

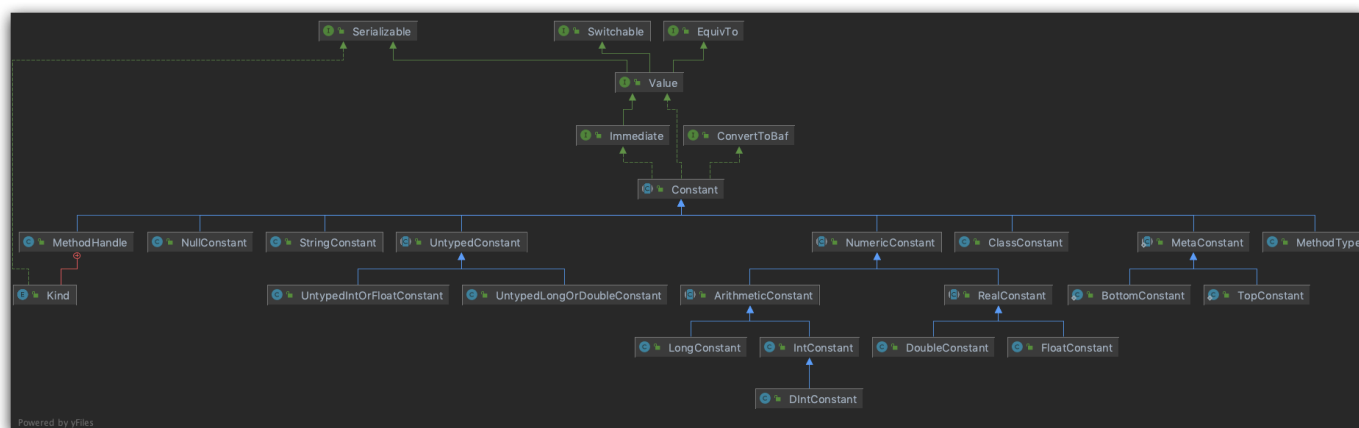
Local

JimpleLocal: local 变量

TemporaryRegisterLocal: \$开头的临时变量



Constant

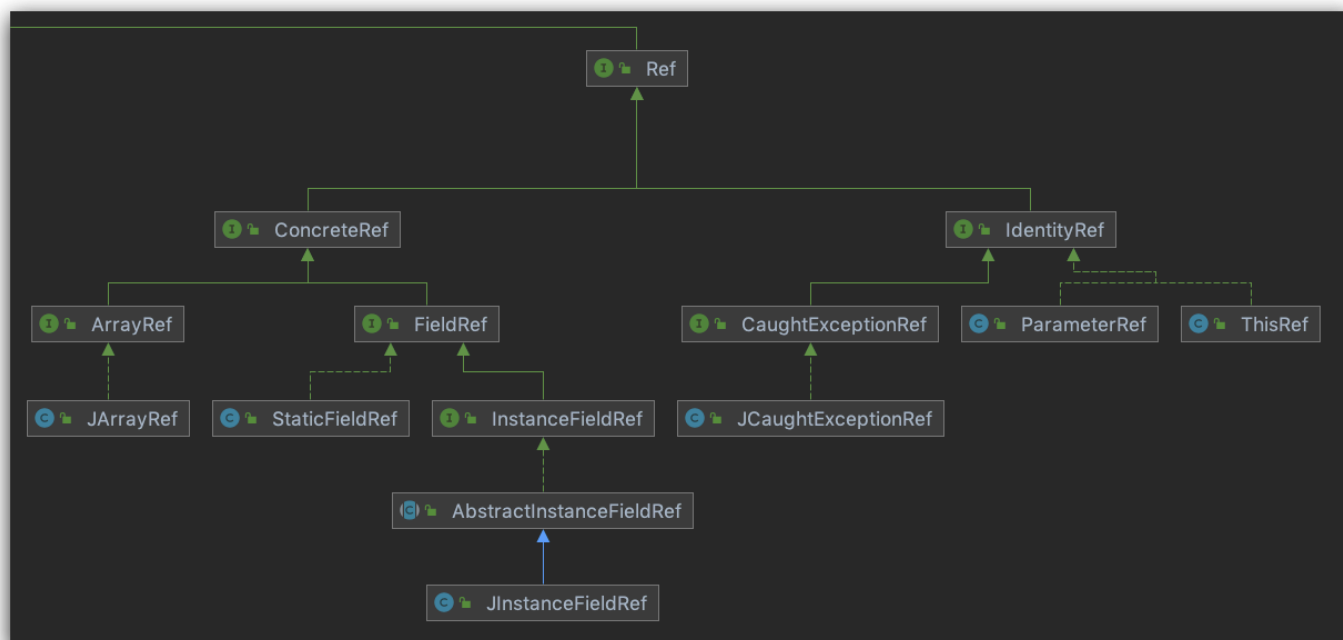
**Ref**

```
ConcreteRef :
```

- `ArrayRef`：指向数组。
- `FieldRef`：指向 `field`。
 - `StaticFieldRef`：静态 `field` 的引用。
 - `InstanceFieldRef`：指向的 `field` 是一个对象实例。

IdentityRef :

- CaughtExcrptionRef : 指向捕获到的异常的引用
- ParameterRef : 函数参数的引用
- ThisRef : this 的引用



Expr

一般来说，一个 Expr 可以对若干个 Value 进行一些操作并且返回另一个 Value。在 Jimple 中，强制要求所有的 Value 最多包含一个表达式。

Stmt 和 Expr 的区别： Stmt 没有返回值， Expr 有返回值。

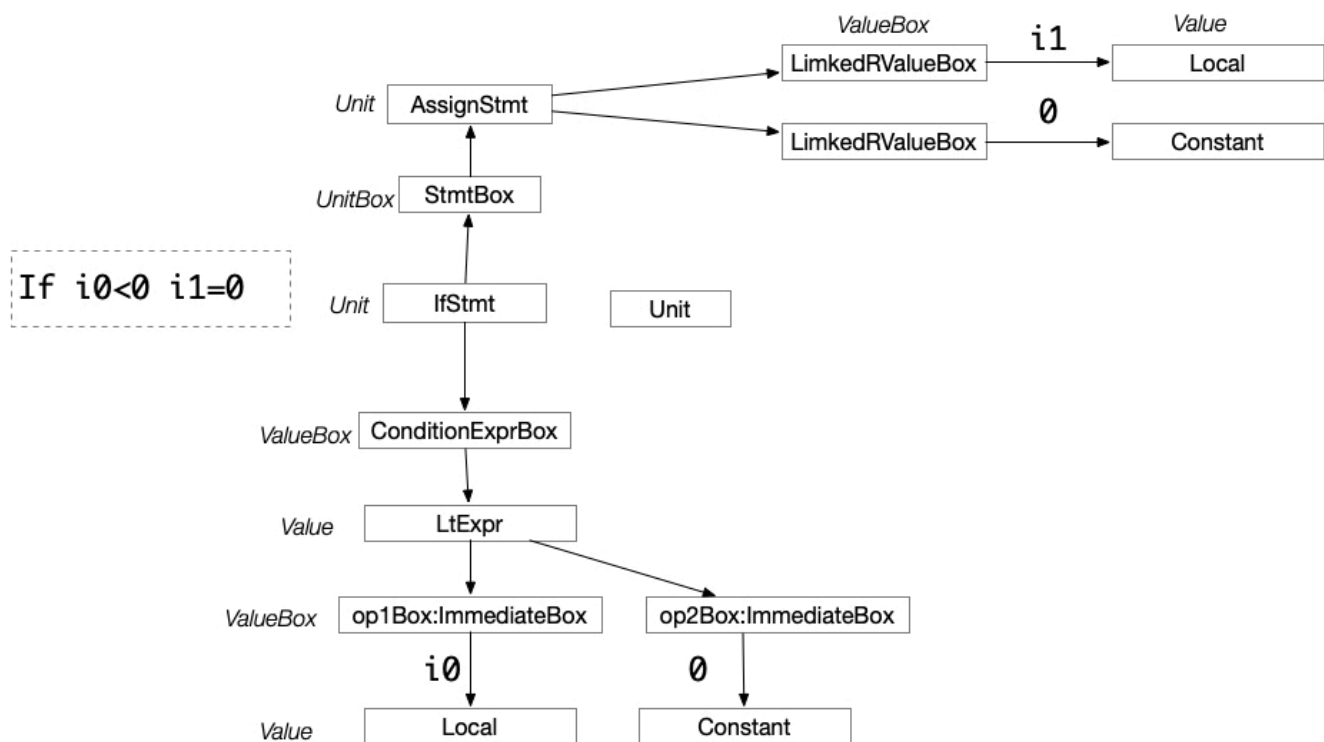
Box

Box 是指针，提供了对 Soot 对象的间接访问。一个 Box 提供了一个间接访问 soot(Unit, Value) 的入口，类似于 Java 的一个引用。当 Unit 包含另一个 Unit 的时候，需要通过 Box 来访问，Soot 里提供了两种类型的 Box，一个是 ValueBox 一个是 UnitBox。

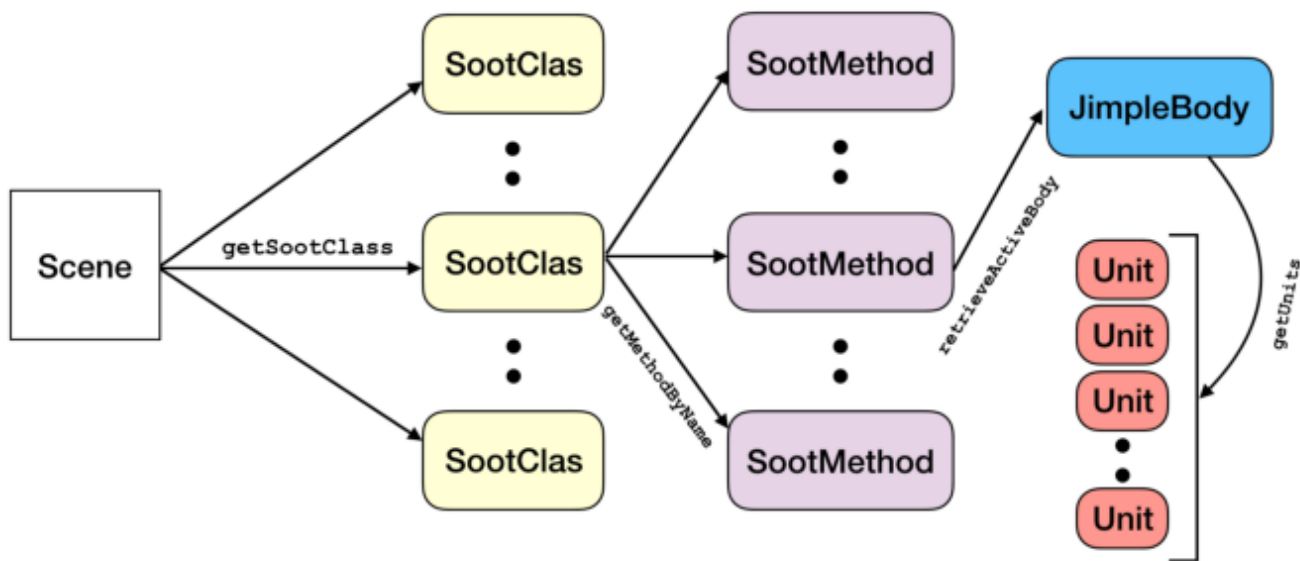
- ValueBox，指向 Values：
 - 对于一条 Unit 来说，他的 ValueBox 存储的是在这条语句内部所用到的和所定义的语句。
- UnitBox，指向 Units：
 - 以 goto 语句为例，UnitBox 其实存的就是 goto 所指的下一跳节点。
 - switch 语句，则会包含很多 boxes。

还需要知道以下的规则：

- 一个 Unit 可以有多个 UnitBox，但是每个 UnitBox 只能指向一个 Unit。GotoStmt 需要知道目标的 Unit 是什么，所以一个 Unit 会包含其它的 UnitBox，通过 UnitBox 获取下一个 Unit。
- 一个 Value 可以对应多个 ValueBox，但是一个 ValueBox 只能对应一个 Value，对于一个 Unit，可以得到很多个 ValueBox，包含着这条语句内部的所用到的和所定义的语句。



基本API



SootClass

获取一个类的信息，如果类是在一个包里，则应该包含完整的包名：

```
circleClass = Scene.v().getSootClass("Circle")
```

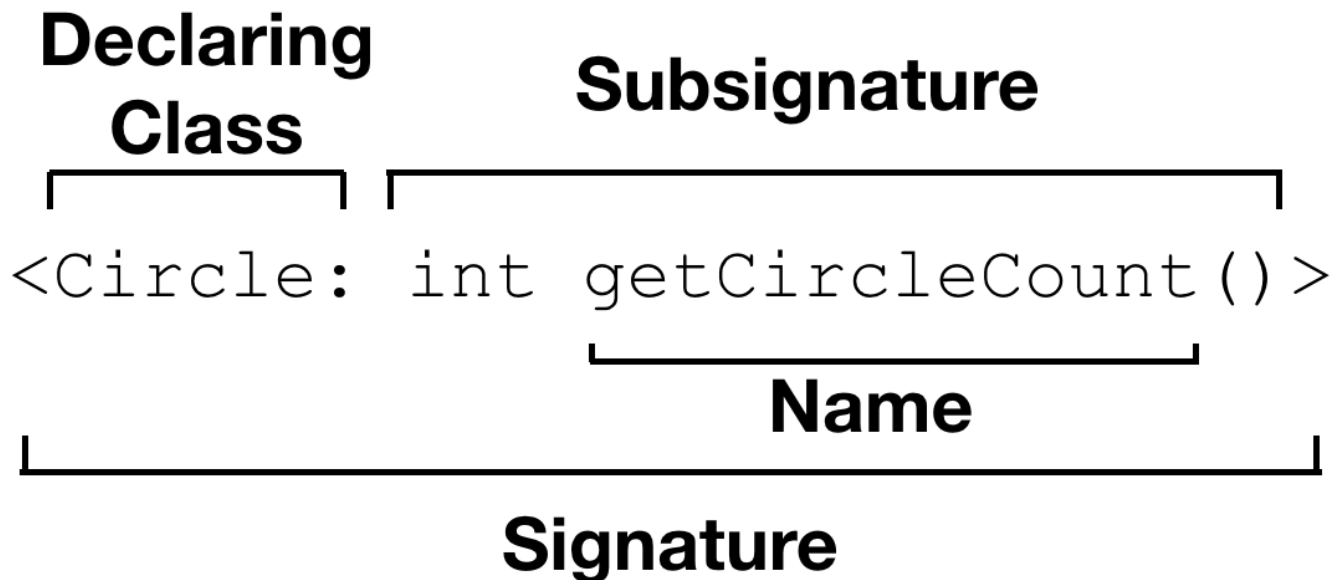
如果分析的类不再 `Scene` 中，则 `getSootClass` 将返回一个 `Phantom` 类或者是异常。`Phantom` 类不影响分析的进行，这在不关注其他模块的代码时很有用。如果要确保查询的类存在，可以通过 `getSootClassUnsafe(className, false)`（如果不存在，则结果为空）检索它或者使用 `circleClass.isPhantom()` 判断它是不是为 `Phantom` 类。

SootField

类中包含字段和方法，通过名称和类型来查找它们：

```
SootField radiusField = circleClass.getField("radius", IntType.v())
```

SootMethod



一般情况下使用 `getMethodByName` 即可：

```
circleClass.getMethodByName("getCircleCount")
```

但是对于方法重载，需要使用 `getMethod` 方法，并提供 `Subsignature`：

```
circleClass.getMethod("int area(boolean)")
```

Modifier

类、方法和字段的访问模式：`public`、`private`、`protected`、`final`、`abstract` 等。这些信息保存在 `Modifier` 中，比如，查询一个方法是否为静态的：

```
Modifier.isStatic(method.getModifiers())
```

Body

在 `Body` 中有 `Units`，`Values`，`Traps`。

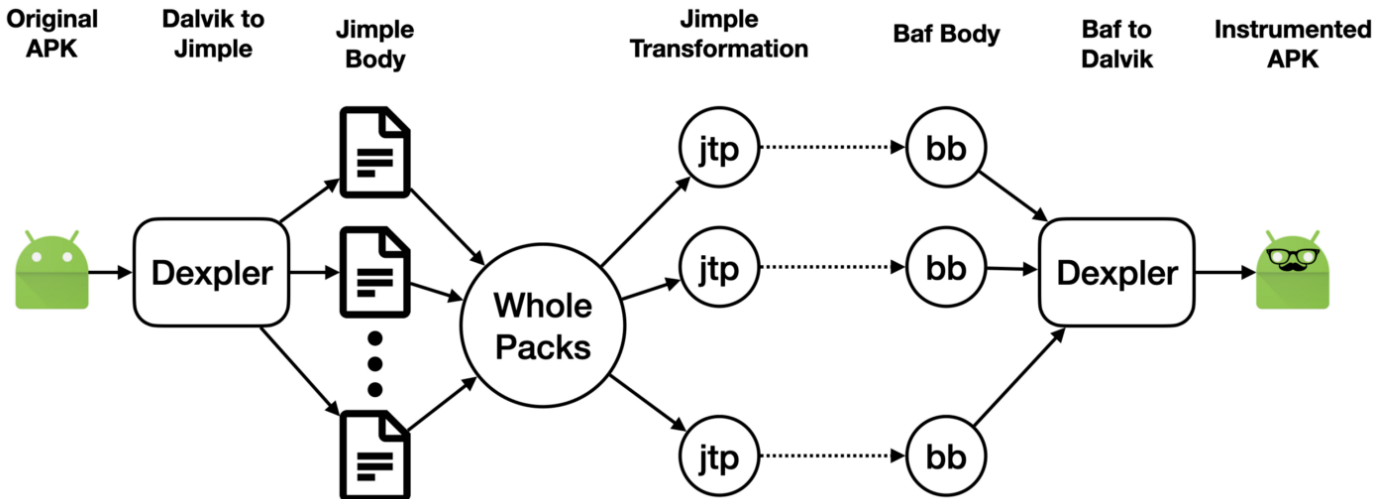
修改代码

Soot 可以修改方法的 `Body`。然后可以使用 `validate` 检查修改是否正确。

```
stmt.apply(new AbstractStmtSwitch() {
    @Override
    public void caseIfStmt(IfStmt stmt) {
        stmt.setTarget(body.getUnits().getSuccOf(stmt));
    }
});
body.validate();
```

APK分析

首先，soot 使用 [Dexpler](#) 将 Dalvik 字节码转换为 Jimple Body。然后可以对整个程序进行转换、优化和注释。接下来，Jimple 转换包将在每个 Jimple Body 上运行。最后，Soot 将所有 Jimple Body 转换为 Baf（Soot 中的低级中间表示），并使用 [Dexpler](#) 将整个代码编译成 APK。



- `jtp`, `jimple` 转换包。
- `jop`, `jimple` 优化包。
- `jap`, `jimple` 注释包。

关于包的具体信息可以看[这里](#)。

Soot设置

关于 `Option` 的详细信息在[这里](#)。

分析 APK 之前，Soot 还需要进行一些设置：

```
public static void setupSoot(String androidJar, String apkPath, String outputPath) {
    // Reset the Soot settings (it's necessary if you are analyzing several APKs)
    G.reset();
    // Generic options
    Options.v().set_allow_phantom_refs(true);
}
```

```

Options.v().set_whole_program(true);
Options.v().set_prepend_classpath(true);
// Read (APK Dex-to-Jimple) Options
Options.v().set_android_jars(androidJar); // The path to Android Platforms
Options.v().set_src_prec(Options.src_prec_apk); // Determine the input is an APK
Options.v().set_process_dir(Collections.singletonList(apkPath)); // Provide paths
to the APK
Options.v().set_process_multiple_dex(true); // Inform Dexpler that the APK may
have more than one .dex files
Options.v().set_include_all(true);
// Write (APK Generation) Options
Options.v().set_output_format(Options.output_format_dex);
Options.v().set_output_dir(outputPath);
Options.v().set_validate(true); // Validate Jimple bodies in each transformation
pack
// Resolve required classes
Scene.v().addClass("java.io.PrintStream", SootClass.SIGNATURES);
Scene.v().addClass("java.lang.System", SootClass.SIGNATURES);
Scene.v().loadNecessaryClasses();
}

```

案例一 插入指令

在每个方法中加入日志输出：`System.out.println("Beginning of method: " + METHOD_NAME)`。jimple 代码：

```

$r1 = <java.lang.System: java.io.PrintStream out>
virtualinvoke $r1.<java.io.PrintStream: void println(java.lang.String)>("
<SOOT_TUTORIAL> Beginning of method METHOD_NAME")

```

Body转换：

```

PackManager.v().getPack("jtp").add(new Transform("jtp.myLogger", new BodyTransformer()
{
    @Override
    protected void internalTransform(Body b, String phaseName, Map<String, String>
options) {
        // First we filter out Android framework methods
        if(InstrumentUtil.isAndroidMethod(b.getMethod()))
            return;
        JimpleBody body = (JimpleBody) b;
        UnitPatchingChain units = b.getUnits();
        List<Unit> generatedUnits = new ArrayList<>();
        // The message that we want to log
        String content = String.format("%s Beginning of method %s", InstrumentUtil.TAG,
body.getMethod().getSignature());
        // In order to call "System.out.println" we need to create a local containing
"System.out" value

```

```

        Local psLocal = InstrumentUtil.generateNewLocal(body,
RefType.v("java.io.PrintStream"));
        // Now we assign "System.out" to psLocal
        SootField sysOutField = Scene.v().getField("<java.lang.System:
java.io.PrintStream out>");
        AssignStmt sysOutAssignStmt = Jimple.v().newAssignStmt(psLocal,
Jimple.v().newStaticFieldRef(sysOutField.makeRef()));
        generatedUnits.add(sysOutAssignStmt);
        // Create println method call and provide its parameter
        SootMethod printlnMethod = Scene.v().grabMethod("<java.io.PrintStream: void
println(java.lang.String)>");
        Value printlnParamter = StringConstant.v(content);
        InvokeStmt printlnMethodCallStmt =
Jimple.v().newInvokeStmt(Jimple.v().newVirtualInvokeExpr(psLocal,
printlnMethod.makeRef(), printlnParamter));
        generatedUnits.add(printlnMethodCallStmt);
        // Insert the generated statement before the first non-identity stmt
        units.insertBefore(generatedUnits, body.getFirstNonIdentityStmt());
        // Validate the body to ensure that our code injection does not introduce any
problem (at least statically)
        b.validate();
    }
}));

```

案例二 FlowDroid

FlowDroid 是一款用于 Android 应用程序和 Java 程序的数据流分析功能，项目[地址](#)。

如果是使用 Maven，编辑 pom.xml 即可：

```

<dependencies>
    <dependency>
        <groupId>de.fraunhofer.sit.sse.flowdroid</groupId>
        <artifactId>soot-infoflow</artifactId>
        <version>2.10.0</version>
    </dependency>
    <dependency>
        <groupId>de.fraunhofer.sit.sse.flowdroid</groupId>
        <artifactId>soot-infoflow-summaries</artifactId>
        <version>2.10.0</version>
    </dependency>
    <dependency>
        <groupId>de.fraunhofer.sit.sse.flowdroid</groupId>
        <artifactId>soot-infoflow-android</artifactId>
        <version>2.10.0</version>
    </dependency>
</dependencies>

```

如果只使用命令行，则在[Release](#)里下载 soot-infoflow-cmd-jar-with-dependencies.jar 即可。

如果不使用 `pom.xml`，使用 `jar` 包的方式，需要去[Release](#)下载 `soot-infoflow-android-classes.jar` 和 `soot-infoflow-classes.jar`，再去Soot的[仓库](#)下载包含 `heros` 与 `jasmin` 的 `sootclasses-trunk-jar-with-dependencies.jar`，将上述三个包加入项目依赖便完成了 `FlowDroid` 的配置。

未来工作

1. 参照 `Jandroid` 的思路，先做一个模版解析，然后多分析APP漏洞多总结多思考。
2. `Java` 不会，这在编程工作中会有问题。
3. 调用图的话，有没有无所谓，主要还是漏洞思路。
4. 还得补一下静态分析的基础知识，否则对于静态分析能做什么其实是不清楚的，也就不知道应该怎么继续开发。
5. 需要对 `soot` 的了解更多一些

端午节

