Jadx plugin report of PoRE

概述

在Jadx上实现Task3，实现的功能包括：

- 常见字符串解码反混淆
- 重命名反混淆
- RGB函数反混淆
- Java反射反混淆

> Jadx的插件API只支持对输入文件进行操作，无法对IR层面进行操作，因此使用jadx的lib库来编写了这个插件
>
> 其中App.java相当于一个添加了自定义插件功能的jadx，可以在里面设置输入的apk和要开启的插件

常见字符串解码反混淆

添加自定义**DecodeVisitor**类，将其添加到jadx反汇编过程的**passes**列表中对反汇编过程中生成的AST树进行遍历，获取其中的加密字符串并解码，然后以注释的形式输出解码信息

获取代码中的字符串

在smali中，所有字符串都由**const-str**指令进行传递，因此通过遍历所有指令，找出其中的const-str指令即可获取字符串

在Jadx中，使用**ConstStringNode**类对**const-str**指令进行包装，通过其`getString()`即可获取指令中字符串

首先遍历AST树中的**ClassNode**节点，然后再遍历节点中的**MethodNode**，获取其中包装的指令**InsnNode**的集合

`getStringFromMethod`方法遍历**MethodNode**中的所有指令，通过是否为**InsnType.CONST_STR**类型来判断是否是const-str指令；然后获取方法中所有的字符串集合并返回

解码字符串

目前支持的字符串编码方式有

- base64
- url
- Unicode

解码逻辑

1. `decode(ArrayList<String> strings)`方法遍历方法中获取的字符串集合
2. 在`decodeString`方法中对每个字符串使用**正则表达式**进行匹配，如果符合三种支持的编码方式之一，则调用Java库中相应的解码函数进行解码；解码的结果保存在**DecodeNode**结构中，其中包含了解码的字符串和编码的方式

3. 对解码出的字符串，再递归调用`decodeString`方法，以此来处理**多次编码**的情况，返回DecodeNode类型的数组表示字符串的解码过程

通过注释添加字符串解码信息

将一个方法中的所有解码出的字符串保存在**HashMap<String, ArrayList<DecodeNode>>**结构中，其中的**key**是解码前的字符串，**values**是解码过程得到的每个字符串的解码过程列表

在Jadx中使用**JadxCommentsAttr**属性来为类或者方法和域添加注释，使用该属性以注释的方式添加字符串的解码信息

如果解码的字符串来自**构造函数**，说明字符串来自类中的静态域，将注释属性添加到**ClassNode**节点，也就是在类声明处添加注释；否则将注释添加到所属的**MethodNode**节点中，也就是函数声明处

反混淆效果

```
package com.example.decode;

import android.os.Bundle;
import androidx.appcompat.app.AppCompatActivity;

// Strings Decode at constructor:
 // Vmtkb2NHTjVRbkJqZVVKb1NVZEthR015V1RKT1EwSjZaRWhLY0dKdFl6MD0= --Base64-> VkdocGN5Qn8jeUJoSUdKaGMyVTJOQ0J6ZEhKcGJtYz0= --Base64-> VGhpcyBpcyBhIGJhc2U2NCBzdHJpbmc= --Base64-> This is a base64 string
public class MainActivity extends AppCompatActivity {
    private final String s_String = "Vmtkb2NHTjVRbkJqZVVKb1NVZEthR015V1RKT1EwSjZaRWhLY0dKdFl6MD0=";

    // Strings Decode at method onCreate:
     // VGhpcyBpcyBhIHN0cmluZw== --Base64-> This is a string
     // %57%65%6C%63%6F%6D%65%20%E4%B8%AD%E6%96%87%20%3C%62%72%3E%21%3C%2F%62%72%3E --Url-> Welcome ������ <br>!</br>
     // Vmtkb2NHTjVRbkJqZVVKb1NVZEthR015V1RKT1EwSjZaRWhLY0dKdFl6MD0= --Base64-> VkdocGN5Qn8jeUJoSUdKaGMyVTJOQ0J6ZEhKcGJtYz0= --Base64-> VGhpcyBpcyBhIGJhc2U2NCBzdHJpbmc= --Base64-> This is a base64 string
     // \u0054\u0068\u0069\u0073\u0020\u0069\u0073\u0020\u0061\u0020\u0073\u0074\u0072\u0069\u006e\u0067 --Unicode-> This is a string
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        System.out.println("VGhpcyBpcyBhIHN0cmluZw==");
        System.out.println("Vmtkb2NHTjVRbkJqZVVKb1NVZEthR015V1RKT1EwSjZaRWhLY0dKdFl6MD0=");
        System.out.println("%57%65%6C%63%6F%6D%65%20%E4%B8%AD%E6%96%87%20%3C%62%72%3E%21%3C%2F%62%72%3E");
        System.out.println("Vmtkb2NHTjVRbkJqZVVKb1NVZEthR015V1RKT1EwSjZaRWhLY0dKdFl6MD0=");
        System.out.println("\\u0054\\u0068\\u0069\\u0073\\u0020\\u0069\\u0073\\u0020\\u0061\\u0020\\u0073\\u0074\\u0072\\u0069\\u006e\\u0067");
    }
}
```

Rename反混淆

Jadx已经提供了效果比较好的重命名反混淆，通过设置**alias**属性，能够自定义一些名称的别名；因此使用这种方法在Jadx的反混淆基础上进行了一些优化

自定义**MyRenameVisitor**类来遍历AST树中的IR节点，判断节点中的类、方法和域名是否进行了混淆并进行名称反混淆

判断是否为混淆名称

通过`isObfuscationName`方法判断是否为混淆名称

判断逻辑:

- 不是android或者java包中的类
- 只有一个字符的名称
- 以数字开头的名称
- 包含特殊字符
- 存在3个连续的相同字符（不区分大小写）
- 在一个长度大于10的名称中，某一个字符（不区分大小写）出现的次数大于名称长度的三分之一

类名反混淆

在`renameClass`方法中实现

实现逻辑：

1. 获取原始类名的前3个字符作为原始类名的标识添加到**newName**中

   > 如果一个长度较长的混淆名称并不符合自定义混淆名称判断规则，则可以通过缩短长度提高其辨识度；
   >
   > 同时还可以尽量避免拥有相同继承关系的类被重命名成相同的名称

2. 如果类继承自其他类，递归向上遍历类的继承类名称

   在`visitSuperTypes`方法中进行递归，如类名符合混淆规则，则获取其super类，并判断是否为混淆名称

   如果继承类的名称不是混淆名称，则将其名称添加到newName后；如果继承类的名称也是混淆名称，则继续递归对超类的名称进行反混淆

   > 对超类名称反混淆结束后直接返回，避免多次继承使得反混淆名称中重复出现相同的超类名称

3. 如果类没有继承，则递归向上遍历其接口类

   `visitInterfacesType`方法，和使用继承关系进行重命名方式相同，通过接口类进行重命名

方法和域名反混淆

根据Java的**重载**规则来进行重命名反混淆

- 类名通过在原始名称后添加参数类型字符串来得到新的名称，如果没有参数添加“_null"
- 域名通过在原始名称之后添加类型字符串来得到新的名称

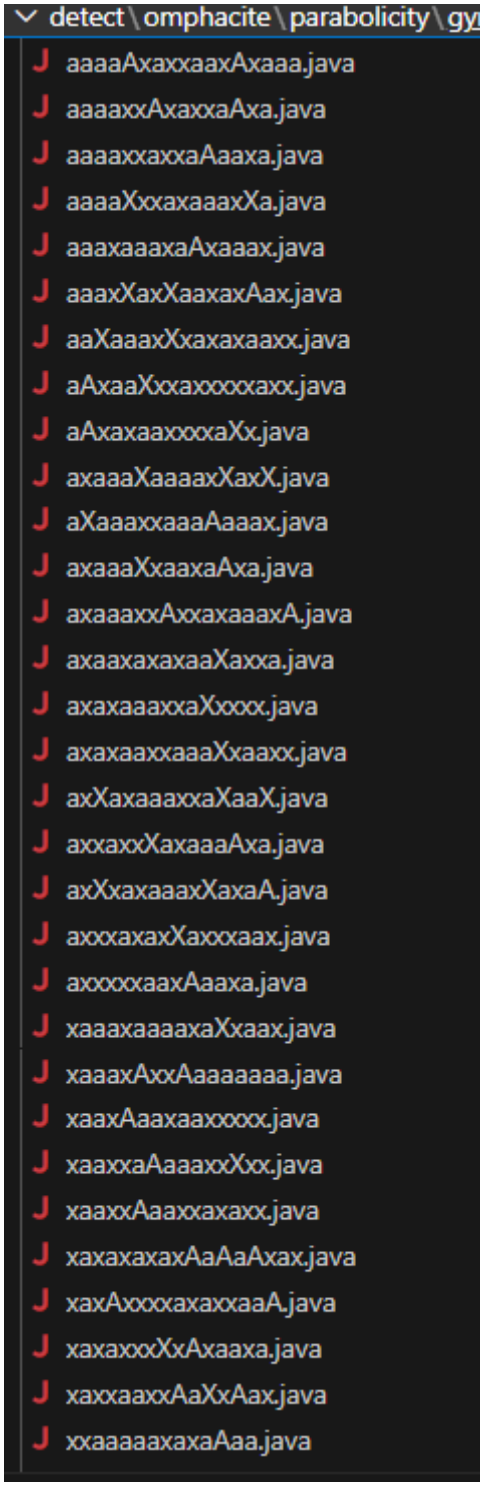
修改名称

使用`setAlias`方法来设置别名，jadx在反汇编过程中回读取这个别名并进行重命名

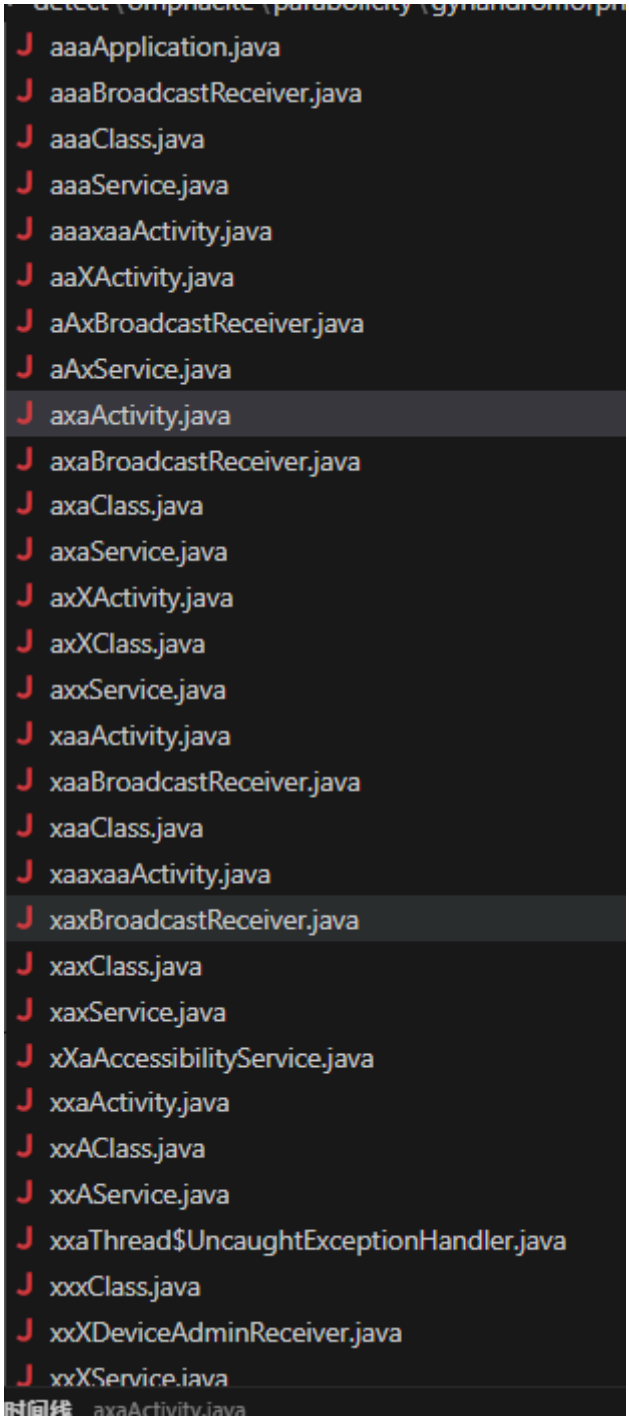
反混淆效果

使用之前lab3中的一个命名混淆过的apk进行反混淆

- 反混淆前


detect \ omphacite \ parabolicity \ gy
aaaaAxaxxaaxAxaaa.java
aaaaaxxAxaxxaAxa.java
aaaaaxxaxxaAaaxa.java
aaaaXxxaxaaaxXa.java
aaaxaaaxaAxaaax.java
aaaxXaxXaaxaxAax.java
aaXaaaxXxaxaxaaxx.java
aAxaaXxxaxxxxaxx.java
aAxaxaaxxxxaXx.java
axaaaXaaaaxXaxX.java
aXaaaxxaaaAaaax.java
axaaaXxaaxaAxa.java
axaaaxxAxxaxaaaxA.java
axaaxaxaxaaXaxxa.java
axaxaaaaxxaXxxxx.java
axaxaaxxaaaXxaaxx.java
axXaxaaaaxxaXaaX.java
axxaxxXaxaaaAxa.java
axXxaxaaaxXaxaA.java
axxxaxaxXaxxxaax.java
axxxxxaaxAaaxa.java
xaaaxaaaaxaXxaax.java
xaaaxAxxAaaaaaaa.java
xaaxAaaxaaxxxxx.java
xaaxxaAaaaxxXxx.java
xaaxxAaaxxaxaxx.java
xaxaxaxaxAaAaAxax.java
xaxAxxxxaxaxxaaA.java
xaxaxxxXxAxaaxa.java
xaxxaaxxAaXxAax.java
xxaaaaaxaxaAaa.java

- 反混淆后



RGB函数反混淆

参考示例中rgb函数反混淆，在Jadx中实现对rgb函数的优化

自定义**RGBVisitor**类对AST树进行遍历，并对Rgb函数进行优化

实现逻辑：

1. 遍历AST树中的所有指令，获取其中的rgb函数调用指令

   在jadx中，调用指令使用**InvokeNode**类进行包装，通过这个类获取invoke指令调用的方法

2. 获取到rgb函数后，通过InvokeNode的`getArg`方法获取参数值，计算出常量后将原来的invoke指令替换为const指令
   - 通过Jadx中的**LiteralArg**常量参数包装类获取rgb函数中的常量参数
   - 计算出rgb函数的返回值

- 创建一条**CONST**类型的InsnNode作为替换指令，设置指令参数为rgb函数计算出的值，设置返回的值为原来rgb函数的返回
- 使用**BlockUtils**中的**replaceInsn**方法进行指令替换

反混淆效果

- 反混淆前

```
public String get() {
    int rgb = 256 - (Color.rgb(1, 1, 1) & 17);
    return a("ifmmp", rgb) + " " + Integer.toHexString(rgb);
}
```

- 反混淆后

```
public String get() {
    int i = 256 - (65793 & 17);
    return a_String_int("ifmmp", i) + " " + Integer.toHexString(i);
}
```

> jadx中只对调用关系进行了内联，并没有继续计算出常量表达式，因为时间有限我也没有继续进行优化

Java反射反混淆

Jadx通过分析SSA和DU-UD链得到一个方法中所有指令的调用关系，然后使用**InsnWrapArg**将一条指令打包成参数作为另一条指令的参数，从而得到一条由多个打包指令作为参数的指令

在Java反射调用中，存在先获取类的信息，再调用类的构造函数实例化一个类或者获取类的方法进行调用以及获取域这样的先后调用关系

通过解包指令，判断指令是否为Java反射方法的调用，然后获取其中的信息，就可以构造出一条Java反射指令的**调用关系链**；通过分析这个关系链来构造出可以进行替换的非反射指令

自定义**ReflectionVisitor**类来遍历AST树并进行反射指令替换

判断反射指令

在`isReflection`方法中判断一条被打包的指令是否包含java反射指令调用关系

判断逻辑：

1. 因为Java反射指令都是方法，所有每个被Jadx打包的反射指令一定是INVOKE类型的，对该指令解包获取其中调用的方法名称，判断是否为反射指令调用；因为时间有限，目前只对几条常见的反射指令进行了判断：
    - froName
    - getMethod
    - invoke
    - getConstructor
    - newInstance
2. 如果解包出的调用函数是Java反射指令，则获取其参数列表
3. 遍历参数如果参数还是一个被包装的指令，说明还要可能是一条反射指令的调用，因此递归调用`isReflection`方法进行判断

4. 如果参数不是被打包的指令，则说明是反射指令的参数；获取反射指令的参数和返回值，将其保存在`ReflectionNode`类型中，这是一个关系链的节点类
5. 还要注意的是，对forName指令的调用应该被视为一个调用关系的起始节点（在目前支持反射指令中），其参数只有一个，也就是反射的类的名称，获取这个类的名称以及调用的返回值保存在`ReflectionRoot`结构中，这是一个调用链的根节点类，其中还保存了反射指令调用的类的ClassNode

构造调用链

`isReflection`方法执行完后，可以得到一个**ReflectionNode列表**，其中包含包装指令中的所有被打包的反射指令的数据（参数和返回值）

定义`ReflectionGraph`类，这是一个有向无环图，其中包含多个以forName指令作为根节点的调用关系链

> 每个节点的后继节点可能不止一个，比如一条forName指令可能或被多个getConstructor指令调用来实例化多个类，或者一条getMethod指令会被多个invoke指令调用来实现方法的多次调用，并且一个方法中会有多个forName方法来反射多个类；因此使用图来保存调用关系链

对获取的ReflectionNode列表，使用`addNode`方法构造出调用关系链

构造方法：

1. 遍历ReflectionNode列表，如果第一个节点是forName指令，说明列表是一个调用关系链，则将第一个节点添加到根节点列表**roots**中，然后将剩余节点使用`addSuccessor`方法依次将它们连接；如果列表的第一个节点不是forName，说明这个反射调用的节点所反射的类可能来自某个根节点；使用**深度优先搜索**来构造调用链
2. `isParent`方法用来判断节点间的调用关系，利用Jadx已经完成的SSA分析，通过比较待连接的节点的第一个参数是否和遍历到的节点的返回值相同，相同则说明两者间存在调用关系并进行连接

优化反射调用

因为反射最终是希望能够访问到类的方法或者属性，因此通过判断graph中是否有invoke指令并且有根节点来确定是否要进行优化

使用`optimizeReflection`方法来优化反射指令

优化逻辑：

1. 遍历根节点，依次对调用链进行分析
2. 使用`buildInstance`方法构造类的实例化方法
    - 获取类的构造信息，对每个需要被实例化的类，其构造信息保存在newinstance指令中，指令的参数是实例化参数，指令的返回值包含实例化的名称
    - 根据获取的实例化信息，调用`makeNewInstanceInsn`方法，该方法利用Jadx中的**ConstructorInsn**类来包装一条实例化指令，返回反射的替换指令
    - 利用Jadx的`BlockUtils.replaceInsn`方法，进行指令替换，然后使用`InsnRemover.remove`方法删除原来的getConstructor方法调用
3. 使用`buildInvoke`方法来替换invoke反射指令：
    - 在invoke指令中包含了方法调用的参数，在对应的getMethod方法中包含了调用的返回信息
    - 获取调用指令的信息，调用`makeInvokeInsn`方法来构造一条替换的调用指令
    - 利用Jadx的`BlockUtils.replaceInsn`方法，进行指令替换，然后使用`InsnRemover.remove`方法删除原来的getMethod方法调用
4. 替换完成后可以删除forName指令的调用

反混淆效果

- 反混淆前

```
public void reflection() throws Exception {
    Apple apple = new Apple();
    apple.setPrice(5);
    Class clz = Class.forName("com.example.decode.Apple");
    Method setPriceMethod = clz.getMethod("setPrice", Integer.TYPE);
    Constructor appleConstructor = clz.getConstructor(new Class[0]);
    Object appleObj = appleConstructor.newInstance(new Object[0]);
    setPriceMethod.invoke(appleObj, 14);
    Object p = Class.forName("com.example.decode.Apple").getMethod("getPrice", new Class[0]).invoke(appleObj, new Object[0]);
    System.out.println(p);
}message:
```

- 反混淆后

```
public void reflection() throws Exception {
    Apple apple = new Apple();
    apple.setPrice(5);
    Object appleObj = new Apple();
    appleObj.setPrice(14);
    Object p = appleObj.getPrice();
    System.out.println(p);
}
```