基于异常利用的安卓应用重打包对抗技术

周立博, 梁 彬, 游 伟, 黄建军, 石文昌

中国人民大学信息学院 北京 中国 100872

摘要 应用重打包是安卓生态中的一种严重的安全威胁。借助应用重打包技术, 攻击者可以向原始应用插入恶意代码以实现不同的恶意功能, 如窃取用户隐私数据、发送收费短信及替换应用广告 SDK 等。有研究表明, 85%以上的恶意应用通过应用重打包的方式产生。对抗安卓重打包攻击, 主要有三种防御方式: 一是在应用开发过程中, 由开发者对应用进行加固, 实施重打包防御策略;二是在应用上传到应用市场时, 进行静态应用重打包检测; 三是在终端设备上进行动态重打包应用检测。其中, 利用重打包工具解析安卓应用程序安装包的缺陷对应用进行加固来提高攻击者生成重打包应用的技术门槛被证明是一种有效的缓解措施。但距今为止, 已有工作并未提出一种系统化的方法来发现可用于保护应用的重打包工具缺陷。本文提出了一种系统化的面向重打包对抗的重打包工具可利用缺陷检测方法。首先, 我们通过代码扫描定位重打包工具中的潜在异常点; 其次, 使用模糊测试的方式来尝试触发被定位的异常; 最后, 监测触发异常的变异应用在目标安卓设备上的运行情况, 并进行进一步的模糊测试来最终构建能被用于对抗重打包攻击的异常触发向量。在以应用广泛的重打包工具 Apktool 为实验对象的测试中, 我们总共发现了 12 个未知的可利用的缺陷, 这些缺陷都已被证明可用于实际应用来对抗重打包攻击。

关键词 安卓应用重打包; 模糊测试; 异常

中图法分类号 TP311 DOI 号 10.19363/J.cnki.cn10-1380/tn.2022.07.04

Countering Android Application Repackaging Attacks via Exception Exploitation

ZHOU Libo, LIANG Bin, YOU Wei, HUANG Jianjun, SHI Wenchang

School of Information, Renmin University of China, Beijing 100872, China

Abstract Android ecosystem has faced a serious security threat of repackaging. With application repackaging, an attacker can insert malicious codes into the original application to implement various malicious functions, such as stealing the user privacy data, sending messages to a number that charged a premium fee, replacing the advertising SDK, and so on. An existing study shows that more than 85% of malicious applications are generated through Android application repackaging. There are three main defense methods against Android application repackaging: implementing the self-protection mechanisms to reinforce the target application by the developer during the development process; performing the static repackaging detection in the application market; enforcing the dynamic repackaging detection in the end devices. Among the above methods, exploiting APK parsing defects of repackaging tools to reinforce application to increase the threshold of application repackaging has been proved to be effective. However, there is lack of a systematic method to discover defects of repackaging tools that can be used to protect applications. In this paper, we propose a systematic method to detect the exploitable implementation defects of the repackaging tool in parsing APK files. First, code scanning is performed to locate potential exception points in the repackaging tool. Secondly, a fuzzing test is employed to trigger the located exception points. Finally, the execution of the mutation application that has triggered the exception points during the fuzzing test on the target Android device is monitored. And further mutation tests to eventually construct an exception trigger vector that can be used to combat repackaging attacks are conducted. In the test of using the repackaging tool Apktool as the experimental object, a total of 12 unknown exploitable defects have been found. All these defects have been proved to be useful in practical applications to counter android application repackaging attacks.

Key words android application repackaging; fuzzing; exception

通讯作者: 梁彬, 博士, 教授, Email: liangb@ruc.edu.cn。

本课题得到国家自然科学基金(No.U1836209, No. 61802413, No. 62002361)资助。

收稿日期: 2021-05-14; 修改日期: 2021-07-16; 定稿日期: 2022-05-11

1 引言

根据国际数据公司 IDC 发布的统计资料, 近五年来, 安卓系统的市场份额稳定在 85%以上[1], 安卓应用生态一片繁荣。但与此同时, 安卓应用也面临着严重的重打包威胁。Zhou 等人[2]的研究显示, 有 85%以上的恶意应用是重打包应用。

为了有效降低恶意重打包应用所带来的危害, 如图 1 安卓应用重打包威胁模型所示, 研究人员从开发者、应用市场和应用安全服务商三个角度出发, 提出了不同的重打包攻击对抗方式: ①开发者实施重打包防御策略, 对应用进行加固, 提高攻击者构建重打包应用的难度。常见的策略有防篡改检查[3-4]、代码混淆[5-7]、应用反调试和信息隐藏等。这种方式部署灵活, 开发者可以根据需要配置不同的防御策略, 保护通过不同渠道发布的应用。②应用市场对应用进行静态重打包应用检测, 通过重打包应用与原应用的相似性[8-17]进行检测, 限制重打包应用通过应用市场传播。③应用安全服务商在应用安装和执行时收集特征信息来进行重打包应用检测。常见的检测方式有安装时签名校验, 运行时 UI 相似度检测[18-20]等。

在上述对抗方式中, 开发者利用安卓应用重打包工具的实现缺陷来加固应用, 中断图 1 中的反编译环节, 是一种有效的重打包防御策略。但目前仅有少数工作对其进行研究, Tim Strazzere[21]指出可以利用反编译工具解析 dex 文件的一个特定缺陷来中断重打包工具对 APK 的解析操作。尽管该缺陷目前已被修复, 但证明了该思路的可行性。自然地, 我们想要知道, 如何系统化地发现更多重打包工具中的缺陷?

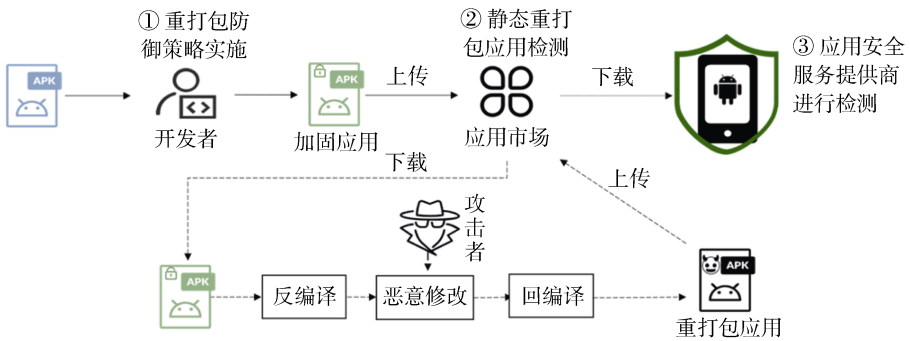


图 1 安卓应用重打包威胁模型

Figure 1 Android application repackaging threat model

针对这一问题, 我们提出了一种面向重打包对抗的重打包工具可利用缺陷检测方法。如图 2 所示, 本方法能够尽可能地检出重打包工具中的可利用缺陷。开发者可以利用这些缺陷对应用进行加固, 加固后的应用可以在目标安卓设备中正常运行而无法被重打包工具正常解析。与其它由开发者实施的重打包攻击对抗技术(如防篡改检查、代码混淆、应用反调试和信息隐藏等)相比, 本文提出的方法具有易于实施、运行时开销小等优势。同时, 该方法与其它加固方法互补, 能够与其他加固方法共同使用, 进一步提高应用抗重打包的能力。

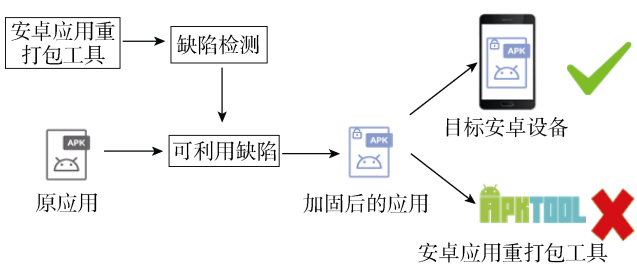


图 2 方法作用示意图

Figure 2 The Schematic diagram of the method

本方法利用重打包工具中缺陷与异常的联系将对缺陷的检测转换到对重打包工具中潜在异常点的研究上来。这一转换基于我们的以下观察: ①重打包工具中的文件解析缺陷能够导致其对 APK 的解析失败, 使得重打包工具不能输出正确的反编译结果。这一特性可用于对安卓应用进行加固。②重打包工具中的文件解析缺陷主要由程序执行过程中未经处理的异常引起。现有工作[21]中对重打包工具解析缺陷的利用证明了我们观察的合理性。此外, 我们在研究过程中发现, 一些知名的应用程序(如作业帮、Instagram)也基于类似观察进行应用加固。

本工作的主要贡献有以下两点:

1) 提出了一种面向重打包对抗的重打包工具解析 APK 文件的实现缺陷检测方法, 以发现可利用的缺陷来加固应用。该方法利用重打包工具文件解析缺陷与异常的联系, 把对缺陷的检测转换为对重打

包工具中潜在异常点的定位、触发和可利用性确认。

2) 我们以当下最流行的重打包工具 Apktool[22] 作为实验对象, 在 Apktool 2.4.1 中找到了 12 个未知的可利用的解析缺陷。并利用这些缺陷对真实应用进行了加固, 加固后的应用不能被 Apktool 重打包, 证明了方法的实际价值。

2 背景知识

为了对应用进行重打包, 攻击者可能会使用多种重打包工具对 APK(Android package)文件进行解析。常见的重打包工具有 Apktool、Jadx、dex2Jar 等。在众多重打包工具中, Apktool 使用最为广泛,是本文的重点研究对象。为了便于读者对本文检测方法的理解, 本节对 Apktool 和重打包工具中的文件解析异常进行简要介绍。

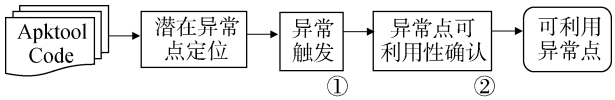
Apktool 是主要由 Ryszard Wiśniewski 与 Connor Tumbleson 开发和维护的开源安卓重打包工具。Apktool 功能完善, 能够反编译 resources.arsc、dex 文件、png 格式的图像文件和二进制的 xml 文件(包括 AndroidManifest.xml 文件与 res 目录下压缩后的 xml 资源文件等)并在之后对其进行回编译。其使用 Java 语言实现对 APK 的反编译, 利用安卓官方提供的打包工具 aapt 或 aapt2 进行回编译生成未签名的重打包应用。

APK 文件本质上是 ZIP 格式的压缩文件, 重打包功能提供对压缩文件中部分或全部文件的解析功能。重打包工具对文件的解析通常是顺序进行的, 即一次解析压缩文件中一个或一组文件。当重打包工具中存在文件解析缺陷时, 重打包工具可能会因为某些文件解析失败而终止运行导致 APK 解析失败。而重打包工具的文件解析过程通常基于特定的文件格式。当遇到不符合工具开发者设定格式的文件时, 程序通常会产生异常, 若开发者认为该异常无法被处理, 则重打包工具则会因为该异常而终止执行。但如果此时安卓操作系统能够对该应用正常解析, 我们就可以认为重打包工具在实现上具有缺陷。由这一思路出发, 本文提出了重打包工具解析缺陷的检测方法, 第三节是对这一方法的具体介绍。

3 方法设计

本方法的指导思想是通过检测重打包工具解析 APK 文件的实现缺陷, 以发现能够用于加固应用的可利用缺陷。我们设计了包含二次模糊测试的缺陷检测方法, 通过寻找可利用的异常点来检测重打包工具中的缺陷。我们以当下最流行的重打包工具 Apktool 为研究对象, 对本方法的具体实现进行介绍。

图 3 是本方法的总体流程图。如图所示, 本方法分可主要分为潜在异常点定位、基于模糊测试的异常触发和包含二次模糊测试的异常点可利用性确认三个阶段。首先, 在异常定位阶段, 我们使用静态分析的方法对应用代码进行扫描, 确定潜在的异常点; 其次, 在异常触发阶段, 我们使用模糊测试的方式来生成测试应用, 根据 Apktool 解析测试应用的日志判断异常是否被触发; 最后, 在异常点可利用性确认阶段, 确认潜在异常点能否用于应用加固, 并对部分无法直接确认可利用性的异常点进行第二次模糊测试。



①: 第一次模糊测试 ②: 第二次模糊测试

图 3 总体流程图

Figure 3 An overview of the method

下面, 我们对本方法的每个环节的具体实现进行详细介绍。

3.1 潜在异常点定位

在进行异常定位前, 我们首先要了解 Apktool 中的异常。由前文我们对 Apktool 的介绍可知, Apktool 进行 APK 解析部分的代码由 Java 语言编写。Java 语言提供了良好的错误处理机制, 其使用 Throwable 类的众多子类来描述各种不同的异常, 为开发者提供了 try, catch 语句捕获和处理异常。

在本方法中, 我们将 Java 语言中的潜在异常点分为两类: **显式异常点**和**隐式异常点**。显式异常点指使用 throw 关键字抛出异常的语句, 如图 4 中第 12 行和第 18 行所示。隐式异常点指的是, 开发者没有主动进行异常处理的可能导致运行时异常的代码块。图 4 中的 *array* 函数包含了一个隐式异常点实例, 在第 23 行, 参数 *x* 没有经过检查就被当作数组初始化的下标使用。当参数 *x* 为负值时, 程序会在运行时抛出异常 Java.lang.NegativeArraySizeException, 同时没有异常处理语句对此异常进行捕获处理。在本工作中, 我们仅关注那些可能导致重打包工具解析终止的异常点, 过滤掉被已被捕获处理的异常点。图 4 中 *test1* 函数仅在 *invoke* 函数中被调用(第 3 行), 其抛出的异常会在 *invoke* 函数中被捕获处理(4~6 行), 因此我们将忽略 *test1* 函数中的异常点。

```
1.    public static void invoke (int x, int y) throws
      AndrolibException{
2.        try {
3.          test1 (x, y);
4.        } catch (AndrolibException e) {
5.          System.out.println ("ignore exception in test1");
6.        }
7.        test2 (x);
8.        array (x);
9.    }
10.   public static int test1 (int a, int b) throws
      AndrolibException {
11.       if (a == 0&&a + b > 0) {
12.         throw new AndrolibException();
13.       }
14.       return a + b;
15.   }
16.   public static int test2 (int size)throws AndrolibException{
17.       if (size ! = 0) {
18.         throw new AndrolibException();
19.       }
20.       return size;
21.   }
22.   public static int array (int x){
23.       String offset[] = new String[x];
24.       System.out.println (offset.length);
25.       return x;
26.   }
```

图 4　3 个 Java 异常实例

Figure 4　Three exception examples in the Java code

为了进行潜在异常点的定位, 我们实现了基于 Soot[23]的代码扫描工具。其工作流程如图 5 所示。首先, 扫描器将输入的 Java 字节码转换为 Jimple 中间代码, 生成程序的函数调用图(Call Graph, CG)和控制流图(Control Flow Graph, CFG)。在生成的 CFG 中包含从 throw 子句到 catch 模块的边和从隐式异常点的前驱节点到异常处理语句的边。我们可以在遍历过程中获得可能的异常点。然后, 如算法 1 所示, 通过尝试匹配异常点与其 try 模块来确认其是否被捕获处理, 过滤掉那些已经被捕获处理的异常点。

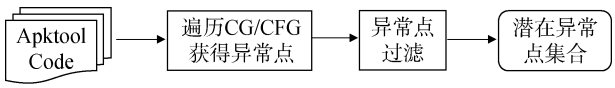


图 5　潜在异常点定位流程

Figure 5　The process of locating potential exceptions

算法 1　异常过滤

输入: CG, 异常点发生语句 *TS*

输出: True:该异常已被处理, 过滤该异常;

False:该异常未被处理, 保留该异常点

```
FUNCITION ifKeep(TS):
IF TS is wrapped by the try:
RETURN False
IF TS.getMethod().getSimpleName()
.equals('Main'):
RETURN True
For caller in CG.getCaller(TS.getMethod()):
InvokeStmt = caller.find(TS)
IF InvokeStmt is wrapped by the try:
CONTINUE
ELSE:
tmp=CG.
    getCaller(InvokeStmt.getMethod())
RETURN ifKeep(tmp.find(InvokeStmt))
RETURN False
```

3.2　基于模糊测试的异常触发

由于不是所有异常点都能够被触发, 我们需要通过异常触发测试筛选出可触发的潜在异常点。如图 6 所示, 这一阶段的主要步骤包括: ①通过重打包工具插桩和测试获得异常可能被触发时所处的解析阶段和对应的解析文件类型; ②根据待检测的异常和其对应解析文件类型生成变异应用; ③使用 Apktool 解析测试应用, 确认异常触发情况。

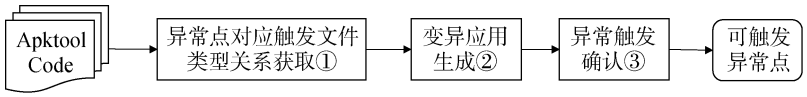


图 6　异常触发流程

Figure 6　The process of the exception triggering

异常点对应触发文件类型关系获取

本阶段的目的是获得 APK 中不同的文件类型可能触发的异常点, 以此加速后续的模糊测试。在整个方案中, 该步骤只需进行一次。我们首先对 Apktool 正常解析 APK 获得的日志进行分析, 发现 Apktool 对 APK 中不同部分的解析相对独立, 每一段解析开始时均会输出特定的日志。可以根据日志区分 Apktool 开始解析不同类型文件的顺序, 在该日志输出处插入相应的开关变量, 标识相应的解析流程。Apktool 对不同类型文件的解析顺序如表 1 所示。

接下来我们依据 Apktool 的解析流程和已获得的异常点进行代码插桩操作。图 7 展示了我们的插桩结果: 在 Apktool 中加入 Phase 类, 使用静态域 *now* 表明当前属于哪个分析阶段。在异常点所在控制流的第一个条件跳转语句前添加日志输出语句, 记录可能触发的异常。

表 1　Apktool 对安卓应用中不同类型文件的解析顺序
Table 1　The order of parsing files of different types in Apks with Apktool

次序	资源类型	说明
1	*resources.arsc*	记录资源文件和相关资源 ID 之间的关系, 可用于查找特定资源。
2	*AndroidManifest.xml*	应用的配置文件
3	*Res*	Android 资源文件
4	*.dex*	由 Java 或 Kotlin 代码编译产生的字节码
5	*Assets*	应用所需的原生资源
6	*Libs*	程序依赖的 native 库

```
AndroidlibResources.java
public ResPackage loadMainPkg(ResTable resTable,
     ExtFile  apkFile) throws AndrolibException {
          Phase.now = 1;
          LOGGER.info("Loading resource table...");
          ……
}

StringBlock.java
private static final int[] getUtf8(byte[] array, int offset) {
       if(pre_getUtf8 != Phase.now) {
          System.out.println( Phase.now +
               " StringBlock getUtf8:line326:" + offset);
          pre_getUtf8 = Phase.now;
       }
       ……
}
```

图 7　代码插桩实例
Figure 7　An example of instrumentation Code

插桩工作完成后, 我们使用插桩后的 Apktool 对测试应用进行解析, 获得输出日志之后就可以根据日志信息确定对应异常触发与 APK 中文件类型的对应关系。图 8 展示了插桩后的 Apktool 解析多看阅读应用的实例, 根据日志片段, 我们可以看到 StringBlock 类中的 *getUtf8* 与 *getUtf16* 函数中存在的异常可能会在 AndroidManifest.xml 和 res 目录下资源文件解析的过程中被触发。

• 变异应用生成

由于 APK 本质上是一个 ZIP 格式的压缩文档, 而 ZIP 文件中存在大量文件有效性检查。例如, ZIP 文件的本地文件头中的 CRC-32 校验值, 核心目录头中存储的其相对本地目录头的偏移量, 对这些变量的修改会导致 APK 无法通过 ZIP 格式校验。

```
I: Decoding AndroidManifest.xml with resources...
... ...
2 StringBlock getUtf8:line326: 141
... ...
2 StringBlock getUtf16:line355: 130
I: Decoding file-resources...
3 StringBlock getUtf8:line326: 176
```

图 8　插桩 Apktool 的解析日志
Figure 8　The parser log of instrumented Apktool

为此, 我们采用变异解压后的文件, 再重新压缩, 签名的方式生成用于测试的变异应用。本质而言, 我们的模糊测试是一种以字节为单位进行变异的针对特定目标文件和特定异常点的定向模糊测试。其中的关键环节包括变异文件选择和具体的文件变异方式。

1) 变异文件选择

首先根据需要探测的异常点和 APK 中文件类型之间的对应关系, 确定可能触发异常点的文件类型。接着遍历 APK 内的文件, 从满足类型且在之前的测试中没有被选中的文件中, 找到体积最小的文件作为输出结果。通过变异文件的选择, 我们可以减小每次测试过程中进行变异的选择范围, 提高模糊测试效率。

2) 具体的文件变异方式

如图 9 所示, 对于每一个文件, 我们以单字节为基本单位对文件进行变异。这主要是因为重打包工具通常以字节作为单位进行读取, 因此以字节为基本单位进行变异是一个自然的选择。从文件头开始, 逐字节将原始位置变异为 0x00-0xFF 之间的一个随机值。每变异一个字节生成一个新文件, 用“源文件名_文件名_位置_变异值”的方式命名变异后的文件。在后续的测试过程中, Apktool 测试输出的日志同样按照此规律进行命名, 根据日志文件名就可以确定对应测试应用变异的字节, 避免因存储大量变异应用而带来的存储开销。

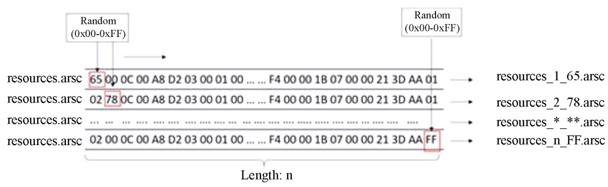


图 9　文件变异方式
Figure 9　An overview of file mutation

• Apktool 异常触发确认

生成变异应用以后, 我们使用 Apktool 对测试应用进行解析, 生成对应的日志文件。此时, 若日志文件中存在报错信息且应用解析提前停止, 我们就认为异常被触发, 可以从日志文件中得到异常的相关信息。否则, 我们认为这次变异没有导致异常被触发。

3.3 异常点可利用性确认

为了保证利用缺陷加固的应用在使 Apktool 解析出错的同时能够在目标安卓设备上正常运行, 我们对应用进行异常点可利用性确认。确认流程如图 10 所示, 包含两个主要流程: 执行监测和第二次模糊测试。下面对这两阶段进行详细介绍。

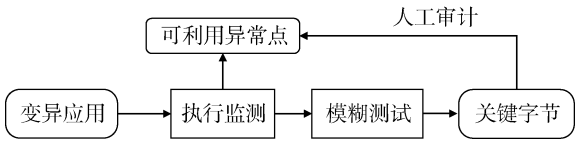


图 10 异常点可利用性确认流程

Figure 10 The validation process of availability of exceptions

• 执行监测

在应用执行监测环节, 我们需要判断变异应用能否在目标安卓系统上正常运行。因此, 我们需要对应用的正常运行状态进行刻画。考虑到变异应用可能会影响原始应用的正常功能, 而从市场下载的应用正常功能对于测试而言是未知的。同时可能存在其他反重打包措施导致变异应用无法直接运行。因此我们考虑直接开发一个测试应用作为两次模糊测试的种子应用。为了使种子应用能够触发尽可能多的解析异常, 我们根据合法 APK 所可能包含的内容, 在应用中添加对应资源, 如在 res/目录下添加 menu, assets 等非必须资源项, 并在相应的 Activity 进行加载和调用。同时为了便于测试, 我们将对不同资源的访问分布在不同的 Activity 下, 通过配置 Activity 的<intent-filter>属性, 使得应用更易于用 adb 进行调试。我们对 Activity 的配置如图 11 所示, 我们将所有的 Activity 都设置为可通过显示意图直接启动, 之后就可以使用脚本来启动 Activity, 加载对应的资源文件, 收集程序运行时的日志信息, 来监测应用的执行状况。

```
<activity android:name=".MainActivity"><intent-filter>
    <action android:name="android.intent.action.1" />
    <category android:name="android.intent.category.LAUNCHER"
    /></intent-filter></activity>
<activity android:name=".MainActivity1"><intent-filter>
    <action android:name="android.intent.action.2" />
    <category android:name="android.intent.category.LAUNCHER"
    /></intent-filter></activity>
<activity android:name=".MainActivity2"><intent-filter>
    <action android:name="android.intent.action.3" />
    <category android:name="android.intent.category.LAUNCHER"
    /></intent-filter></activity>
```

图 11 测试应用的活动配置

Figure 11 Activity configuration of the test application

• 进一步模糊测试

在执行检测过程中, 我们发现一些异常能够使 Apktool 在解析 APK 时崩溃, 同时对应变异应用无法在安卓手机上正常执行。图 12 展示了一个此类待确认异常。

```
325.  private static final int[] getUtf8 (byte[] array, int offset) {
326.      int val=array[offset];
327.      int length;     ...
}
```

图 12 待确认异常实例

Figure 12 An example of exception to be confirmed

在对测试应用中 res/layout 目录下一个布局文件进行变异后, 该异常被触发并导致 Apktool 崩溃, 但变异应用无法在安卓系统上正常运行。图 13 展示了其在实际执行中产生的崩溃日志。

```
AndroidRuntime: FATALEXCEPTION: main
AndroidRuntime: Process: com.example.simpletest, PID: 31241
AndroidRuntime: java.lang.RuntimeException:
Unable to start activity ComponentInfo
{com.example.simpletest/com.example.simpletest.MainActivity2}:
    android.view.InflateException: Binary XML file line #2: null
```

图 13 测试应用崩溃日志

Figure 13 Crash logs of the test application

我们认为对于此类异常已经体现了重打包工具与安卓系统在解析 APK 上的差异, 通过后续人工分析造成差异的原因可能将此类异常利用起来。接着我们将此变异测试应用作为模糊测试的种子, 对该文件进行变异。根据日志分析判断对应的测试应用所变异的字节与待确定异常是否相关。

4 实验评估

在本章, 我们将用实验来评估我们的方案在 Apktool 上的具体实现。首先, 我们对实验中的环境和使用的数据做简要介绍(4.1 节); 接着, 我们对通过实验找到的可利用异常进行介绍, 并对其中的部分实例进行解析(4.2 节); 然后, 我们对本方法的性能开销进行分析(4.3 节); 最后, 我们以 LinkedIn 为例, 展示了两个可利用异常对实际应用进行加固的过程和效果(4.4 节)。

4.1 实验环境及数据

我们在 Java 环境中实现了对 Apktool 字节码的静态分析和插桩工作, 使用 Python 语言编写脚本进行模糊测试和应用在安卓设备上运行情况监测。实验用到的 PC 配置为 16GB 内存, 6 个 2.1GHz 内核和

1TB 硬盘, 安卓手机型号是小米 6, 安卓系统版本为 9.0。

我们的实验主要有以下两个数据来源:

1) 测试应用: 测试应用集合来自豌豆荚与 Apkpure。其中包含豌豆荚应用分类排行榜 14 个分类中, 每类的前 50 个应用, Apkpure 中分类排行榜 32 个分类中每类前 10 个应用, 总测试应用共计有 1020 个。在异常点对应触发文件类型关系获取阶段, 我们使用插桩后的 Apktool 对测试应用进行解析, 获得异常和 APK 压缩包中文件类型之间的对应关系。

2) 种子应用: 我们在进行模糊测试时使用自己开发的测试应用作为种子应用, 以避免应用市场中包含重打包检测, 代码混淆和资源混淆的应用对实验结果造成影响。

4.2 实验结果与实例分析

通过实验与人工分析, 在 Apktool 2.4.1 上共发现 343 个异常, 总共确认了 12 个未知的可利用的缺陷, 与缺陷相关的可利用异常点的详细信息如表 2 所示, 下面我们对找到的异常及其抛出这些异常的解析缺陷进行分析。

表 2 可利用异常点

Table 2 Available exception points

序号	文件名	方法名	行号	类型
1	ExtDataInput.Java	skipCheckShort	53	显式
2	ExtDataInput.Java	skipCheckByte	63	显式
3	ARSCDecoder.Java	readTableHeader	77	隐式
4	ARSCDecoder.Java	readEntrData	27	显式
5	ResValueFtory.Java	Fatory	35	显式
6	ResValueFactory.Java	bagFatory	101	显式
7	DexUtil.Java	verifyDexHeader	86	显式
8	StringBlock.Java	decodeString	294	隐式
9	StringBlock.Java	getUtf8	328	隐式
10	StringBlock.Java	getUtf16	351	隐式
11	ExtDataInput.Java	skipCheckInt	45	显式
12	ExtDataInput.Java	skipCheckChuTypeInt	75	显式

• 抛出显式异常的解析缺陷

图 14 展示了 ExtDataInput 类中被确认的显式异常。在异常定位阶段, 我们能够通过对代码的解析, 定位到该 throw 语句, 并逐级检查该函数抛出的异常有没有被调用者用 catch 语句捕获。最终我们能够从找到从 *main* 函数到该函数的路径, 确认在 *main* 函数抛出的异常中可能包含此异常, 因此将其作为可能的异常。之后在变异 arsc 文件的过程中得到的变异应用触发了该异常, 且能够在目标设备上正常运行。由此我们确认其是一个可利用异常。

```
50.  public void skipCheckShort (short expected) throws IOException {
51.      short got = readShort();
52.      if (got != expected) {
53.          throw new IOException (String.format(
54.          "Expected: 0x%08x, got: 0x%08x", expected, got));
55.  }}
```

图 14 显式异常实例

Figure 14 An example of explicit exception

• 抛出隐式异常的解析缺陷

图 15 展示了 ARSCDecoder 类中的一个隐式异常, 该隐式异常同样与数组操作相关。图中 *packageCount* 变量从文件中获取, 后续作为数组的大小在初始化时被使用。与 StringBlock 类中的例子类似, 该值同样可能为负值, 导致抛出 Java.lang.NegativeArraySizeException 异常。

```
72. private ResPackage[] readTableHeader() throws IOException,
AndrolibException {
73.         nextChunkCheckType (Header.TYPE_TABLE);
74.         ResPackage[] packages = new ResPackage[packageCount];
75.         nextChunk();
76.         for (int i= 0; i < packageCount; i++) {
77.             mTypeIdOffset = 0;
78.             packages[i] = readTablePackage();
79.         }
80.         return packages;
81.}
```

图 15 隐式异常实例

Figure 15 An example of implicit exception

但 Android 系统去解析 APK 时, 并不关心 *packageCount* 对应的字节。该解析差异导致此异常被触发的同时, 应用能够在手机上正常运行。

• 需要进一步探测的解析缺陷

以 3.3 节我们讨论的异常为例, 在使用种子应用进行模糊测试测试的过程中我们注意到, 若在第一轮探测中当变换 res/layout/activity_main.xml 中第 64 个字节为 0xFF 时, 会导致第 318 行 *offset* 值为负值, 触发 Java.lang.NegativeArraySizeException 异常。但该应用无法在手机上正常运行。于是我们将此变异测试应用作为模糊测试的种子进行第二轮 fuzz 操作。

通过这种方式, 我们可以发现异常的关键输入, 如图 16 所示。确认了 arsc 文件中与其对应的关键字节, 通过观察可以得知其储存着值 *res/layout/activity_main.xml*, 即我们变异的文件。经过进一步分析, 我们可以从中获得一种加固应用的新方法: 向 res/layout 目录下插入无意义的损坏布局文件, 同时在对应的索引文件中添加相应的索引值。这样由于

Apktool 会依据索引值对 APK 中的文件进行解析, 重打包工具将会在解析到我们添加的损坏布局文件时出错。而无意义的布局文件不会在系统中被解析, 使得加固后的安卓应用能够在目标安卓设备上正常执行。

图 16 模糊测试确定的异常相关字节

Figure 16 Exception related bytes confirmed by fuzzing

4.3 性能开销

本文方案在实现上可以大致分为三个阶段: 潜在异常点定位、异常触发和异常可利用性确认。在潜在异常点定位阶段, 我们主要对 Apktool 进行静态分析, 这一步骤需要对重打包应用的 CG 和 CFG 进行遍历。静态分析阶段在整个方案中仅需执行一次, 即使消耗时间较长也在可接受范围内。在异常触发阶段, 会进行大量变异操作并使用 Apktool 进行解析, 消耗时间较长。在异常可用性确认阶段, 我们能够利用异常触发阶段的结果, 跳过部分可能导致异常的变异探测, 从而加速关键字节的探测过程。每个阶段的详细耗时情况如表 3 所示。

从表 3 可以看出, 异常查找花费时间最少, 平均时间在 1.05 s 左右, 异常触发花费时间最长, 平均每个异常被触发需要十分钟以上。在异常利用阶段因为已经获得触发阶段的数据, 平均每个异常耗时 600 s 要低于异常触发阶段, 符合我们的预期。

表 3 各阶段耗时统计

Table 3 Time consuming statistics of each stage

	异常查找	异常触发	异常利用
总消耗时间(s)	362	9800	1800
异常数量	343	N/A	3
触发异常数目	N/A	12	N/A
平均耗时(s)	1.05	817	600

4.4 实例验证

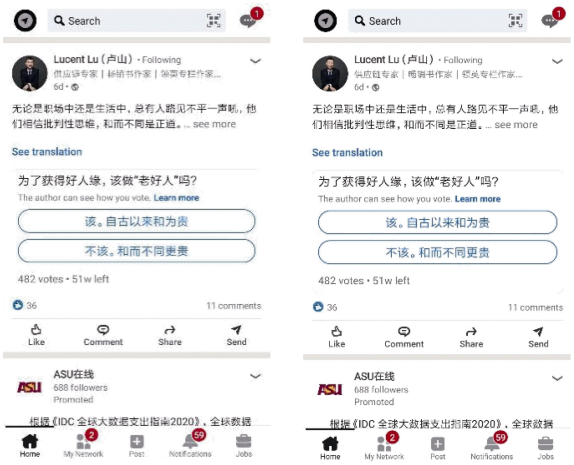
在实例验证阶段, 我们从豌豆荚和 Apkpure 中挑选了 9 个真实应用, 使用已检测到的缺陷进行应用加固验证。9 个应用可以按大小分为三组, 1~3 号小型应用(1~20MB), 4~6 号中型应用(20~100MB), 7~9 号大型应用。每组应用从来自豌豆荚中的测试应用中随机选出, 为了体现 Apkpure 商店与豌豆荚的差异, 我们使用 Apkpure 特有的应用替换部分测试应用。得到了最终的真实应用列表, 如表 4 所示。

表 4 真实应用列表

Table 4 Real application list

应用编号	应用包名	应用版本	应用来源
1	guangdiangtong.suanming1	5.0	豌豆荚
2	com.yoyo.aimao	3.0.1	豌豆荚
3	com.laji.esports	1.0.8	豌豆荚
4	com.whatsapp	2.19.168	Apkpure
5	com.linkedin.Android	6.0.125	豌豆荚
6	com.telegram	5.7.1	Apkpure
7	com.imangi.templerun2	5.13.1	豌豆荚
8	com.knight.union.wdj	3.1.0	豌豆荚
9	jp.naver.line	9.7.0	Apkpure

接下来我们以常见的 LinkedIn 应用为例, 展示我们利用 4.2 节已分析的部分异常点对实际应用的加固效果和加固细节。如图 17 所示, 在应用加固前后, 应用功能和用户界面没有发生变化, 加固后应用能够在目标安卓设备上正常运行。



(a) 应用加固前 (b) 应用加固后

图 17 应用加固前后界面对比

Figure 17 Comparison of the interface before and after application reinforcement

下面介绍应用的加固细节。*skipCheckShort* 函数中异常对应的解析缺陷是 Apktool 在解析 arsc 文件时对 Entry 资源项数值读取的额外检查。进行检查的语句为 ARSCDecoder 类 *readValue* 方法中的 *mIn.skipCheckShort((short) 8)*, 该语句检查读入的数值是否为 0x08, 当检测到的值不是 0x08 时, 抛出显式异常。由此, 我们首先对 LinkedIn 应用中 arsc 文件的部分 Entry 项进行修改, 将对应的 0x08 替换成 0xFF,如图 18 所示。

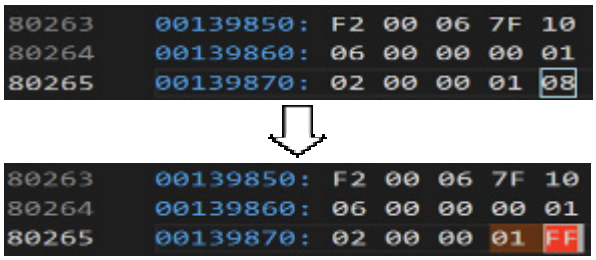


图 18　skipCheckShort 函数中异常的利用实例
Figure 18　Exploitation of the exception in the function skipCheckShort

接着用修改后的 arsc 文件替换原 arsc 文件生成 LinkedIn 的重打包应用 com.linkedin.android_1.apk。重打包应用能够在手机上正常运行，使用 Apktool 无法正常进行解析，在解析过程中抛出异常：brut. androlib.AndrolibException: Could not decode arsc file。同样的，我们也可以利用隐式异常使得 Apktool 抛出运行时异常。图 19 展示了对 *readTableHeader* 函数中隐式异常的利用。

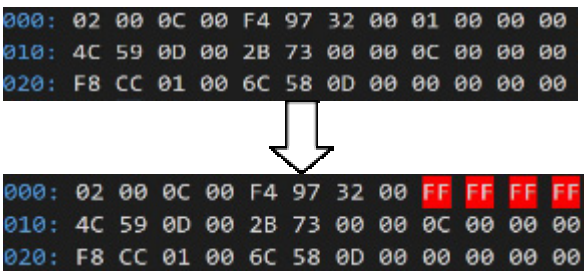


图 19　readTableHeader 函数中异常点的利用实例
Figure 19　Exploitation of the exception in the function readTableHeader

通过变异 arsc 文件起始的第 9~12 个字节，使预期对应的 *packageCount* 值为–1，最终导致 Apktool 在解析过程中抛出异常：Exception in thread "main" Java.lang.NegativeArraySizeException。

类似地，我们可以利用其他缺陷对应用进行加固。加固后的应用在目标设备上正常运行地同时能够使 Apktool 解析异常。限于文章篇幅，其他异常点的利用情况不再一一赘述。

5　讨论

文件变异方式：在本文中，我们以单字节为基本单位对文件进行变异。这主要是因为重打包工具通常以字节作为单位进行读取，因此以字节为基本单位进行变异是一个自然的选择。另一个可能的选择是以比特为单位进行变异，这种变异方式可能会提高触发异常的可能性，但是在模糊测试阶段会带来较大的性能开销。在未来工作中，我们将深入探索不同变异策略的异常触发效果及性能开销。

方法自动化程度提升：本方法在执行检测环节，根据日志判断应用是否在目标系统上正常运行。目前，应用不能正常运行的具体原因需要人工对日志内容进行分析确认。在实验过程中，我们发现从日志中提取出的关键字能够反映出应用不能正常运行的原因。在未来工作中，我们拟设计启发式规则，根据日志内容提取出的关键字，确定应用崩溃原因。

方法有效性：本工作提出了一种新的应用自我保护策略来对抗安卓应用重打包技术——利用安卓应用重打包工具解析 APK 的实现缺陷对应用进行加固。尽管目前在本方案中发现的 Apktool 解析缺陷可能在后续版本中被修复，但这一思路能够被应用加固厂商借鉴，将其扩展到其他重打包工具如 jadx, dex2jar 等，并发现新版本中可能存在的缺陷。对于攻击者来说，通过本方案加固的应用能够增加其重打包应用的难度。即使攻击者已经获得了可能会导致重打包工具出错的异常点，也要对重打包工具或加固后的应用进行修改。提高了其进行攻击的时间成本，在一定程度上能够起到提高重打包攻击技术门槛的效果。

6　相关工作

对于如何抵抗安卓重打包攻击，目前主要有三种防御方法：一是开发者对应用进行加固，在应用中实施自我保护策略；二是在应用市场方面进行的静态重打包应用检测；三是在应用安装或执行时由可靠的应用安全服务提供商进行动态重打包应用检测。

应用重打包对抗的一种主流方式是开发者进行应用加固，既在应用中实施应用自我保护策略，常见的策略有防篡改检查、代码混淆、应用反调试和信息隐藏等。防篡改检查是指在应用中添加代码检测应用自身是否被篡改。为了加强防篡改检查代码的安全性，Luo 等人[3]在应用中分散插入大量检测代码来使得攻击者难以定位所有检测代码，Zeng 等人[4]在方案 BombDroid 中创造性地利用经常在恶意软件中使用的逻辑炸弹来检测攻击者对应用的修改。代码混淆技术通过对应用的控制流[5]、程序中涉及到的标识符和字符串进行混淆[6-7]，通过提高攻击者分析混淆代码难度来对应用进行保护。应用反调试则通过在应用中添加调试检测代码来进行判断应用执行状态，阻碍攻击者通过调试进行动态分析。提高攻击者重打包应用的难度。信息隐藏技术是指通过加密

等方式隐藏部分关键代码, 提高攻击者对代码进行分析的难度。本质而言, 本文提出的方法属于由开发者实施的重打包攻击对抗技术。相比于已有的防篡改检查、代码混淆、应用反调试等策略, 本文提出的方法具有易于实施、运行时开销小等优势。同时, 该方法与其它加固方法互补, 能够与其他加固方法共同使用, 进一步提高应用抗重打包的能力。

重打包应用检测技术以重打包应用与原应用之间的相似性为基础, 通过特征选取, 相似性比较来判断待检测应用是否为重打包应用。根据选取特征的方式, 重打包应用检测技术可以进一步分为两种类型: 静态重打包应用检测和动态重打包应用检测。静态重打包应用检测对 APK 进行静态分析, 选取不同的应用特征如应用代码、API 调用序列、代码中包含的控制流信息、应用功能、资源文件等作为相似性度量的基础。Zhou 等人[8]在安卓发展的早期就实现了一款名为 DroidMoss 的相似性度量系统。该系统从 dex 文件中抽取指令序列作为待测应用的初步特征, 接着计算指令序列的模糊哈希(Fuzzy Hashing)值作为该应用的指纹信息, 最后通过两两计算应用指纹之间的编辑距离来确定第三方市场中是否存在重打包应用。文献[9-10]发现大量第三方库的使用会导致基于dex文件的相似性检测系统产生误报, 提出了两阶段的检测方案: 在进行特征提取前首先对应用使用的库进行检测, 移除第三方库代码后再进行特征提取操作。Chen 等[11]的团队与 Marastoni 等[12]团队对代码做进一步处理, 将代码之间的控制流信息作为相似性比较的基础。Marastoni 等人为了获得更好的检测效果, 还额外提取了 API 的调用情况作为第二特征来刻画应用。Abdurrahman 等人[13]与 Fan 等人[14]利用机器学习技术, 从程序的 API 调用情况出发训练分类器进行重打包应用的检测。文献[15-17]基于重打包应用与原应用 UI 的高度相似性, 提出不同的检测方法, 对重打包应用进行检测。

应用安全服务提供商常用动态重打包检测来进行重打包应用检测。他们可以在应用执行时收集应用特征来进行重打包应用检测。文献[18-20]收集应用运行时产生的UI信息来进行相似性检测, 文献[24]提出了一种收集程序运行时 API 调用信息作为检测依据的方法, 文献[25]提出了一种向应用中添加水印信息并在运行过程中进行检查的重打包应用检测方式。

7 总结与展望

本文提出了一种面向重打包对抗的重打包工具可利用缺陷检测方法, 能够发现可用于加固应用的重打包工具文件解析缺陷。首先, 我们使用静态代码分析的方式对重打包应用进行代码扫描, 定位可能的异常点; 其次, 使用模糊测试技术触发异常, 获得异常对应的关键输入; 然后, 监测触发异常的变异应用在目标安卓设备上的运行情况, 对异常的可用性进行确认, 并进行进一步的模糊测试来辅助研究者构建能被用于对抗重打包攻击的异常触发向量。在以重打包工具 Apktool 2.4.1 为实验对象的测试中, 我们总共发现了 12 个未知的可利用的缺陷。所有这些缺陷都已被证明可用于实际应用中, 能够有效对抗重打包攻击, 证明了本工作的可用性。

尽管已经可以利用本方法检测出 Apktool 中未知的可利用缺陷, 但本工作仍存在一定不足和需要改进的地方。

一是在其他重打包工具上进行推广验证。当前我们针对 Apktool 进行实验验证, 未来我们将在后续工作中将本方法推广应用到其他重打包工具上。

二是增加对其他重打包阶段的考虑。应用的重打包过程通常需要经历反编译与回编译两个阶段。在本文的工作中, 我们主要考虑 Apktool 在反编译方面可能引起的异常, 缺少对回编译阶段的分析。为此, 我们要在将来的后续工作中加入对回编译阶段异常的利用, 实现对重打包工具的全流程覆盖。

参考文献

[1] Os-share. https://www.idc.com/promo/smartphone-market-share/os. April. 2021.

[2] Zhou Y J, Jiang X X. Dissecting Android Malware: Characterization and Evolution[C]. *2012 IEEE Symposium on Security and Privacy*, 2012: 95-109.

[3] Luo L N, Fu Y, Wu D H, et al. Repackage-Proofing Android Apps[C]. *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016: 550-561.

[4] Zeng Q, Luo L N, Qian Z Y, et al. Resilient Decentralized Android Application Repackaging Detection Using Logic Bombs[C]. *The 2018 International Symposium on Code Generation and Optimization*, 2018: 50-61.

[5] Junod P, Rinaldini J, Wehrli J, et al. Obfuscator-LLVM — Software Protection for the Masses[C]. *2015 IEEE/ACM 1st International Workshop on Software Protection*, 2015: 3-9.

[6] DexGuard. https://www.guardsquare.com/en/tags/dexguard. April. 2021

[7] Aonzo S, Georgiu G C, Verderame L, et al. Obfuscapk: An Open-Source Black-Box Obfuscation Tool for Android Apps[J]. *SoftwareX*, 2020, 11: 100403.

[8] Zhou W, Zhou Y J, Jiang X X, et al. Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces[C]. *The second ACM conference on Data and Application Security and*

Privacy, 2012: 317-326.

[9] Glanz L, Amann S, Eichberg M, et al. CodeMatch: Obfuscation Won't Conceal your Repackaged App[C]. *The 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017: 638-648.

[10] Linares-Vásquez M, Holtzhauer A, Poshyvanyk D. On Automatically Detecting Similar Android Apps[C]. *2016 IEEE 24th International Conference on Program Comprehension*, 2016: 1-10.

[11] Chen K, Liu P, Zhang Y J. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets[C]. *The 36th International Conference on Software Engineering*, 2014: 175-186.

[12] Marastoni N, Continella A, Quarta D, et al. GroupDroid: Automatically Grouping Mobile Malware by Extracting Code Similarities[C]. *The 7th Software Security, Protection, and Reverse Engineering / Software Security and Protection Workshop*, 2017: 1-12.

[13] Pektaş A, Acarman T. Deep Learning for Effective Android Malware Detection Using API Call Graph Embeddings[J]. *Soft Computing*, 2020, 24(2): 1027-1043.

[14] Fan M, Liu J, Wang W, et al. DAPASA: Detecting Android Piggybacked Apps through Sensitive Subgraph Analysis[J]. *IEEE Transactions on Information Forensics and Security*, 2017, 12(8): 1772-1785.

[15] Zhauniarovich Y, Gadyatskaya O, Crispo B, et al. FSquaDRA: Fast Detection of Repackaged Applications[C]. *Data and Applications Security and Privacy XXVIII*, 2014: 130-145.

[16] Zhang F F, Huang H Q, Zhu S C, et al. ViewDroid: Towards Obfuscation-Resilient Mobile Application Repackaging Detection[C]. *The 2014 ACM conference on Security and privacy in wireless & mobile networks*, 2014: 25-36.

[17] Lyu F, Lin Y P, Yang J F, et al. SUIDroid: An Efficient Hardening-Resilient Approach to Android App Clone Detection[C]. *2016 IEEE Trustcom/BigDataSE/ISPA*, 2016: 511-518.

[18] Yue S T, Feng W Z, Ma J, et al. RepDroid: An Automated Tool for Android Application Repackaging Detection[C]. *2017 IEEE/ACM 25th International Conference on Program Comprehension*, 2017: 132-142.

[19] Hu Y Y, Xu G S, Zhang B W, et al. Robust App Clone Detection Based on Similarity of UI Structure[J]. *IEEE Access*, 2020, 8: 77142-77155.

[20] Soh C, Kuan Tan H B, Arnatovich Y L, et al. Detecting Clones in Android Applications through Analyzing User Interfaces[C]. *2015 IEEE 23rd International Conference on Program Comprehension*, 2015: 163-173.

[21] Tim Strazzere. Dex education: Practicing safe dex[C]. *Blackhat USA*, 2012.

[22] Apktool. https://ibotpeaches.github.io/Apktool/, April. 2021.

[23] Soot. https://github.com/soot-oss/soot. April. 2021.

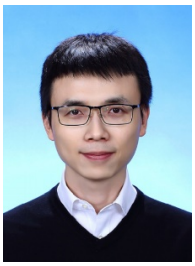
[24] Kim D, Gokhale A, Ganapathy V, et al. Detecting Plagiarized Mobile Apps Using API Birthmarks[J]. *Automated Software Engineering*, 2016, 23(4): 591-618.

[25] Zhou W, Zhang X W, Jiang X X. AppInk: Watermarking Android Apps for Repackaging Deterrence[C]. *The 8th ACM SIGSAC symposium on Information, computer and communications security*, 2013: 1-12.

周立博 于 2018 年在中国人民大学信息安全专业获得学士学位。现在中国人民大学信息安全专业攻读硕士学位。研究兴趣包括软件安全、移动安全等。Email: zhoulibo@ruc.edu.cn

梁彬 于 2004 年在中国科学院软件研究所计算机软件与理论专业获得博士学位。现任中国人民大学信息学院教授。研究领域为信息安全。研究兴趣包括: 软件安全性分析、信息安全攻防对抗及系统软件安全机制等。Email: liangb@ruc.edu.cn

游伟 于 2016 年在中国人民大学信息学院获得博士学位。现任中国人民大学信息学院副教授。研究领域为信息安全。研究兴趣包括安全漏洞挖掘、恶意程序分析及移动安全等。Email: youwei@ruc.edu.cn



黄建军 于 2017 年在普渡大学计算机科学专业获得博士学位, 现为中国人民大学信息学院计算机系讲师, 研究兴趣包括: 移动安全、区块链安全、软件安全分析。Email: hjj@ruc.edu.cn

石文昌 于 2002 年在中国科学院软件研究所计算机软件与理论专业获得博士学位。现任中国人民大学信息学院教授。研究领域为信息安全。研究兴趣包括: 信息安全、可信计算和数字取证等。Email: wenchang@ruc.edu.cn