Android 应用 Smali 代码混淆研究

邱子杨,付 雄
(南京邮电大学 计算机学院,江苏 南京 210003)

摘 要:近年来,互联网行业快速发展,安卓系统由于其开源的特点,成为了全球市场份额最多的操作系统。但也由于其开源性的特点,造成安卓应用软件恶意攻击的简易性,再加上目前安卓应用软件保护技术的不成熟,使得针对安卓应用软件的恶意攻击越来越多。厂商应对恶意软件攻击的方式简易,一般会在源码级别进行安全加固混淆,来防止恶意攻击,但由于 Android 中间代码产物过多,攻击者可对反编译产物 smali 代码进行随意篡改,即可在 APP 中植入恶意代码,这严重破坏了 Android 的安全性。为了解决这一问题,该文总结出一套可以防御 smali 代码篡改的混淆方法,对寄存器存储字符串类型的值进行加密混淆,并结合不透明谓词技术对其控制流进行混淆,还加入新的自定义逻辑来对 APP 的逻辑流进行混淆,让攻击者在反编译时发生异常并且无法获得正确的代码逻辑。从强度、开销、弹性三个方面对 smali 混淆方法进行有效性分析。实验结果表明,该方法可以对抗反编译的逆向分析。

关键词:smali;混淆;数据流;控制流;逻辑流
中图分类号:TP393 **文献标识码**:A **文章编号**:1673-629X(2021)07-0120-07
doi:10.3969/j.issn.1673-629X.2021.07.020

Research on Smali Code Obfuscation in Android Application

QIU Zi-yang,FU Xiong
(School of Computer,Nanjing University of Posts & Telecommunications,Nanjing 210003,China)

Abstract:In recent years,with the rapid development of the Internet industry,Android system has become the operating system with the largest market share in the world due to its open source characteristics. However,due to its open source characteristics,the Android application malicious attack is simple,coupled with the current Android application protection technology is not mature,so the malicious attack against Android application software is more and more. Manufacturers respond to malicious software attacks in a simple way. Generally,security reinforcement and obfuscation are performed at the source code level to prevent malicious attacks. However,due to the excessive intermediate code products of Android,attackers can tamper with the decompiled product smali code at will and implant malicious codes into the APP,which seriously undermines the security of Android. In order to solve this problem,we summarize a set of obfuscation methods that can prevent smali code tampering,encrypt and obfuscate the value of the register storage string type,and combine it with opacity. The predicate technology obfuscates its control flow,and also adds new custom logic to obfuscate the logic flow of the APP,so that the attacker will have an exception during decompilation and cannot obtain the correct code logic. The effectiveness of smali obfuscation method is analyzed from three aspects of strength,cost and flexibility. Experiment shows that the proposed method can resist the reverse analysis of decompilation.

Key words:smali;obfuscation;data flow;control flow;logic flow

0 引 言

根据 2016 年第一季度统计报告指出[1],Android 设备所占市场份额为 76.4%。但同时根据《腾讯移动安全实验室 2016 年上半年手机安全报告》指出[2],2016 年上半年新增安卓恶意应用安装包 918 万个,是 2014 年全年新增恶意应用包的 9.15 倍,感染用户数量已经超过 2 亿,同比去年增长了 42%。安卓用户群体的高比例以及恶意应用软件的激增使得安卓应用软件的安全防护技术日渐重要。

攻击者一般会通过对安卓应用程序进行反编译并篡改代码再对其进行重新打包并且重新签名来进行恶意行为的攻击。当攻击者攻击一个安卓应用程序文件时,并不会直接反汇编去分析其 smali 代码[3],因为 smali 代码过于复杂,难以读阅,而他们往往会把可执行文件 apk 反编译成 java 源码,然后从 java 源码中获取到关键代码的位置,然后再去对应的 smali 代码位

收稿日期:2020-09-08 **修回日期**:2021-01-11
基金项目:国家自然科学基金资助项目(61202354)
作者简介:邱子杨(1995-),男,硕士生,研究方向为移动应用安全;付 雄,博士,教授,研究方向为云计算、分布式计算。

置进行篡改[4],然后重新打包,重新签名就可以得到一个被篡改过的安卓应用程序。

因此,防止 Android 应用程序逆向分析是非常重要的。在以往的研究中,Enck 等列出了 Android 系统的安全性分析方法[5],分析了 Android 系统的安全性模型并解释其复杂性。Felt 和 Fahl 等[6-7]详细说明了 Android 系统的安全性。Bernhard 等[8]开发了 Bauhaus,基于软件安全技术和代码分析技术的 Android 系统源代码分析工具,并通过案例研究从安全认证和许可机制的角度分析了 Android 系统实施中的不一致信息。

牛豪飞[9]实现一套在 ART 模式下的安卓应用保护方案。通过对 Android 平台的体系结构、安全机制以及常见静态和动态攻击的分析,提出了动态和静态两个方向的安卓安全防御方案。尉惠敏[10]提出了一套基于 Hook 文件的自修改方案,可以应对大部分攻击手段。其加密粒度更小,加密精度更准确。实现了 So 动态库中函数运行前解密运行后加密,保证任何时刻 So 动态库都不是完整的明文。是一种针对于 Hook 攻击的动态防御机制。

对于 Java 层面的安卓应用加固,彭守镇[11]提出了基于资源文件的加密方式。对反编译后的 dex 文件进行加密混淆处理,对资源文件进行加壳处理,从而预防 apk 被反编译篡改,增强打包难度。吕苗苗[12]提出基于 Java 的安卓应用代码混淆技术。通过抽离安卓软件中的 code_item 代码,将其进行混淆然后形成混淆代码索引表,同时在 So 中进行封装。最后采取 JNI 机制注册封装后的代码,生成特定的执行环境。从而做到对安卓应用代码的混淆加固。

对于 smali 层面的安卓加固,郑琪等人[13]通过插入多余的控制流和压扁控制流对控制流进行混淆以对安卓应用进行防御。并从功能、性能两个方面做出了评价。刘方圆等人[14]同样在 smali 层面对安卓应用加固进行研究,其主要是对寄存器的值加以混淆以及通过不透明谓词对控制流进行混淆,并且在强度、弹性、开销进行的技术评价。吴林[15]提出了基于 Dalvik 字节码指令的混淆技术,从数据流和控制流两个层面对 apk 进行混淆。

综合现有的工作,大部分的工作都是针对于 Android 应用程序中源码级别 Java 代码的混淆,对于 Android 应用程序中 smali 级别的代码的安全性分析还是比较缺少。对于 Java 源代码层面的混淆目前主要是控制流和标识符混淆两个方面,可是大部分情况下厂商并不可能直接提供源码给安全公司,这样的话这种保护就无法发挥其作用。对于现有的 smali 层面的研究,主要围绕数据流和逻辑流进行混淆[16-18],且混淆方式并不考虑对原来安卓应用的额外消耗,且混淆方式单一,缺乏动态性。

针对上述代码混淆技术出现的问题,文中提出了 smali 代码的混淆保护方法,在除去数据流和控制流两个层面之外,又增加了逻辑流层面的混淆,并且对于数据流和控制流消耗方面也做了优化和把控,其主要贡献如下:

(1)设计实现 smali 代码混淆系统,不需要提供源码,但能对抗攻击者的攻击。

(2)针对之前研究的混淆方式,提出新的混淆方式逻辑流混淆用来加强混淆强度。

(3)实验与评价。用这种方法保护后的应用程序可以抵抗静态分析,并使反编译后的代码出错。同时,还评估了其对时间和空间性能的影响,以证明该方法的有效性。

1　混淆框架综述

文中提出的混淆方式不同于 Java 策略方向的安全加固,不需要源码但是完全可以对抗恶意攻击,通过对 Android 应用程序代码中间产物 smali 代码的混淆,使得攻击者无法获取正确的逻辑代码。

下面简要介绍该方法的混淆过程(见图 1):

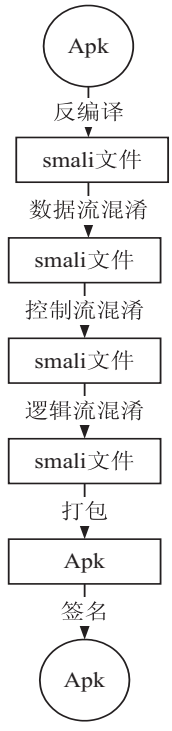

图 1　混淆方法过程

step1:反编译待保护的 apk 文件;

step2:遍历所有 smali 文件,通过数据流混淆模块进行混淆;

step3:遍历所有 smali 文件,通过控制流混淆模块进行混淆;

step4:遍历所有 smali 文件,通过逻辑流混淆模块进行混淆;

step5:将编译后的文件重新打包为 apk 文件;

step6:将 apk 文件重新签名。

2 加密策略

在数据流混淆模块中,对数据通过加密策略进行混淆,为了不影响源程序的性能,且尽量缩短加密过程但对数据能起到混淆的作用,所以文中提出了一种基于字典的简易加密方式。但由于字典加密算法有较大风险,字典一旦被破解或盗取就面临数据泄露的风险,所以又通过 RSA 对字典进行加密存储在服务器端,且字典是由系统随机生成的,所以不同 Apk 的数据加密方式是不同的,减少被破解的风险。下面介绍字典的生成和传输过程。

step1:生成两个数组 A,B,其大小为 26,且每个数组依次存放 0-25 数字;

step2:打乱 A 数组和 B 数组的内容,即为字典 D;

step3:使用混淆系统中的私钥对 A 与 B 进行加密存储,生成加密字典 RSA(D);

step4:混淆系统发送字典 RSA(D),给需要加密的 apk 端,apk 端通过公钥解密得到字典 D;

step5:使用字典 D 进行加密。

在上述过程中,小写字母 x 对应的密文为 A[index_x],其中 index_x = ascii(x) - 0;ascii(x)为 x 对应的 ascii 的码值。同样大写字母 X 对应的密文为 B[index_X],其中 index_X = ascii(X) - 0;ascii(X)为 x 对应的 ascii 的码值。

3 混淆策略

Collberg 等人[19]提出了代码混淆的概念,提出了代码混淆的基本定义,给出了第一个代码混淆算法,将代码混淆分为外形混淆、控制混淆、数据混淆和预防混淆,通过强度、执行代价、弹性来评估有效性。代码混淆其实就是一种代码功能一致性的转化,会保留与之前代码一致的有效性,通过某种混淆转换,使得变换后的程序在具有相同功能的前提下更难分析[20]。如果程序和具有一致的可观测行为,则称其为混淆转换。

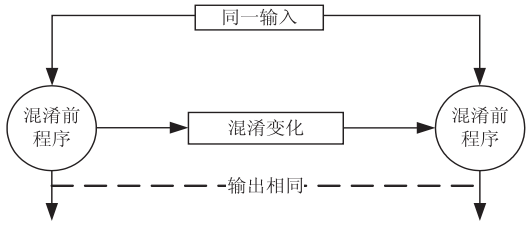

图 2 smali 代码混淆原理示意图

代码混淆的基本原理如图 2 所示,整体上说混淆方式主要有静态混淆和动态混淆两个方式。静态混淆有控制流混淆、数据流混淆等,动态混淆是通过自定义的混淆器来达到运行过程中动态的进行混淆以及解混淆的方法。

smali 代码混淆:

(1)基于字典加密的数据流混淆。

在 Dalvik 虚拟机中,由于字符串是通过明文形式显示赋值给寄存器,然后再由具体操作在寄存器内取字符串的具体值,这样就会造成代码暴露的危险性。所以可以将字符串进行加密处理,对数据进行混淆,增加代码阅读难度,降低破解代码逻辑风险。

在传统加密方式中,常用的加密方式为对称加密和非对称加密,但其加密方式繁琐且性能消耗大,但是混淆方法中,对原有程序的影响越小则混淆方法是越有效的,所以文中采用基于字典的加密方式来简化加解密过程,减少对原有程序的影响。这种加密方式即随机生成 a-z 以及 A-Z 唯一对应的字典密文后进行加密。数据流混淆过程如算法 1 所示。

算法 1:数据流混淆。

输入:待保护应用反编译之后的 smali 文件。

输出:数据流混淆之后是 smali 文件。

1. 设 smali 文件集合为 $F = (f_1, f_2, \cdots, f_n)$,其中 f_i 表示第 i 个 smali 文件。

2. 遍历集合 F,逐行读取 f_i 文件。

3. 设行内容为 f_{i_j},记录加密位置 $p_m = j$(当 f_{i_j} 满足正则\s*const-string\s+(v|p)(\d)\,\s+"(.*)"),添加到集合 P_i 中。

4. 设字典加密算法为 $D(x)$,遍历集合 P_i,获取加密字符串 $s = D(f_{i_{p_m}})$,替换 $f_{i_{p_m}} = s$。

5. 遍历集合 F,输出。

6. end

算法 1 只是对如下指令 const-string v0, "InjectLog"进行简单的修改,并且由于加密算法是基于字典进行加密,所以并不会修改字符串长度,且解密过程是基于字典式的,所以查询密钥对应原文的时间复杂度为 $O(1)$,并不会对系统性能产生很大的消耗,也不会对系统文件大小产生改变,但是能在一定程度上抵抗逆向工具的逆过程,是一种低消耗、高性能的数据流混淆方式。

图 3 为数据流混淆后的结果。

```
.method public test()V
    ......
    const-string v0, "adminqiu"
    .line 17
    .local v0, "account":Ljava/lang/String;
    const-string v1, "admin"
    ......
    const-string v3, "success"
    ......
    const-string v3, "fail"
    ......
.end method
```

(a)混淆前

```
.method public test()V
    ......
    const-string v0, "apkjngji"
    .line 17
    .local v0, "account":Ljava/lang/String;
    const-string v1, "apkjn"
    ......
    const-string v3, "yizzxyy"
    ......
    const-string v3, "qajl"
    ......
.end method
```

(b)混淆后

图 3 数据流混淆

可以看出,有意义的字符串全部被更换为了无意义的字符串,这样就使得攻击者无法从字面获取程序的意义从而进行逻辑分析,很大程度上增加了逻辑分析的难度。

(2)低消耗控制流混淆。

算法 2:控制流混淆。

输入:待保护应用反编译之后的 smali 文件。

输出:控制流混淆之后是 smali 文件。

1. 设 smali 文件集合为 $F = (f_1, f_2, \cdots, f_n)$,其中 f_i 表示第 i 个 smali 文件。

2. 设存储方法的集合为 Map 集合, $M = (m_i, c_i)$,其中 m_i 表示方法的全限定名称, c_i 表示 m_i 被调用的次数。

3. 遍历集合 F,逐行读取 f_i,如果当前调用了方法 m_i,检测 Map 是否存在 (m_i, c_i),如果存在则 c_i ++,否则添加一个新元素 (m_i, c_i)(其中 $c_i = 1$)到 Map 中,并且维持方法总量 $C = C + c_i$(C 初始值为 0)。

4. 遍历 M 集合,如果当前元素 (m_i, c_i) 中 $c_i > \frac{1}{10}C$,将其添加至位置集合 $P = (p_1, p_2, \cdots, p_l)$,其中 $p_s = c_i$。

5. 重新遍历集合 F,逐行读取 f_i,如果当前调用的方法 $m_i \in P$,则执行控制流混淆器 $D(x)$,混淆结果 $s = D(m_i)$,替换 $f_i = s$。

6. 遍历集合 F,输出。

7. end

控制流混淆相对于数据流混淆来说,是更有效的混淆方式,其原理是在程序的某些地方添加一些常用的控制 switch、if 等来隐藏原程序中真正的控制流程,从而阻止反编译攻击。并且在 smali 语法中,控制流语法复杂程度远远高于 Java 源代码,其是通过跳转指令来实现对代码的逻辑控制,所以加入一些无效控制流或者修改一些控制流,可以很大程度混淆代码,降低代码的可读性。

文中提出的低消耗控制流多注入的混淆方式,首先确定插入位置,确定插入位置的方式是将方法被调用次数作为依据,超过阈值的,将其被调用的位置确定为待插入位置。

控制流混淆不会对原有命令做修改混淆,而是在原有命令上增加无效控制流,比如常用的 if 分支,switch 分支,这增加了原有程序的复杂性,给攻击者分析 smali 代码增加了难度。

图 4 为控制流混淆的结果。

可以看出,混淆后之前的简单逻辑被加大了复杂度,出现了分支,从而干扰攻击者的逻辑分析,增大了分析难度。

(3)逻辑流混淆。

对数据流混淆和控制流混淆来说,如果攻击者通过数据分析获取加密规则并且攻击者对 smali 语言十分熟悉,这种情况下,数据流混淆和控制流混淆会显得苍白。

为加大混淆力度,文中提出一种新的混淆方式—逻辑流混淆,逻辑流是指程序的执行过程。

```
.method protected onCreate(Landroid/os/Bundle;)V
    ......
    invoke-virtual {p0}, Lcom/qiuziyang/test1/MainActivity;->test()V
    ......
    return-void
.end method
```

(a)混淆前

```
.method protected onCreate(Landroid/os/Bundle;)V
    ......
    .local v0, "a":I
    const/4 v1, 0x2
    .line 14
    .local v1, "b":I
    if-ge v0, v1, :cond_0
    .....
    invoke-virtual {p0}, Lcom/qiuziyang/test1/MainActivity;->test()V
    ......
    :cond_0
    return-void
.end method
```

(b)混淆后

图 4　控制流混淆

由于 smali 源代码所有的逻辑必然都是与 APP 本身执行过程有关的,所以当攻击者进行反编译之后,对 smali 源代码,在短时间内很有可能分析出 APP 执行过程从而对 APP 的安全造成危害。为了提高安全性,文中提出了一种逻辑流混淆的思想,让攻击者在短时间内无法分析出 APP 的逻辑流,或者让其分析无效的逻辑流。

逻辑流混淆的具体做法是在 APP 逻辑流某一节点处添加不影响 APP 逻辑流和数据流的自定义逻辑流,不仅加大 APP 逻辑流的复杂程度,也由于无关逻辑流的加入,使得攻击者对于 APP 执行过程的分析并不都是有效的,进而增加 APP 的安全性。对于自定义的逻辑流,如果过于复杂会增加 APP 对系统的消耗,过于简单又对混淆加固起不到很大作用,所以选择合适的自定义逻辑流是尤为重要的。控制流混淆过程如算法 3 所示。

算法 3:逻辑流混淆。

输入:待保护应用反编译之后的 smali 文件。

输出:控制流混淆之后是 smali 文件。

1. 设 smali 文件集合为 $F = (f_1, f_2, \cdots, f_n)$,其中 f_i 表示第 i 个 smali 文件。

2. 构建逻辑流图 $T = (t_1, t_2, \cdots, t_n)$,其中 t_i 为树节点,也就是调用的方法,其中 T 表示树的层次遍历序列,且 T 是一棵有序树。

3. 选择逻辑流节点,这里采用随机选取的方式,调用逻辑流混淆器 $D(x)$,结果 $t = D(t_i)$,替换 $t_i = t$。

4. 遍历集合 F,输出。

5. end

图 5 为逻辑流混淆的结果。

```
.method protected onCreate(Landroid/os/Bundle;)V
    ......
    invoke-virtual {p0}, Lcom/qiuziyang/test1/MainActivity;->test()V
    ......
    return-void
.end method
```

(a)混淆前

```
.method protected onCreate(Landroid/os/Bundle;)V
    ......
    new-instance v0, Lcom/qiuziyang/test1/Add1;
    invoke-direct {v0}, Lcom/qiuziyang/test1/Add1;-><init>()V
    .local v0, "add1":Lcom/qiuziyang/test1/Add1;
    invoke-virtual {v0}, Lcom/qiuziyang/test1/Add1;->t1()V
    invoke-virtual {p0}, Lcom/qiuziyang/test1/MainActivity;->test()V
    new-instance v1, Lcom/qiuziyang/test1/Add2;
    invoke-direct {v1}, Lcom/qiuziyang/test1/Add2;-><init>()V
    .local v1, "add2":Lcom/qiuziyang/test1/Add2;
    invoke-virtual {v1}, Lcom/qiuziyang/test1/Add2;->t2()V
    return-void
.end method
```

(b)混淆后

图 5 逻辑流混淆

可以看出,之前的逻辑前后不只是单纯的一个逻辑的执行,而是被加上了多余的逻辑,而这些多余的逻辑是逻辑流混淆模块自定义的,与程序本身逻辑流无关,也不会影响程序逻辑流的执行。这样的混淆既做到了安全加固,增大攻击者分析难度,也不会影响源程序的执行,且由图可看出相对数据流混淆和控制流混淆,复杂度更高。

4 评价标准

4.1 强 度

借助于软件复杂度技术,可以定义混淆后的程序 P_r' 比原始程序 P_r 更难被人理解其逻辑过程,这种混淆过程定义如下:设 T 是一种保持混淆程序 P_r' 和原始程序 P_r 功能一致的混淆过程,混淆后的程序 P_r' 与原始程序 P_r 有 $P_r \overrightarrow{T} P_r'$。设 $E(P)$ 为 P_r 的复杂度,该复杂度由程序长度、圈复杂度、面向对象度量等特征定义。令 T 对程序 P_r 的力量 $T_{\text{pot}}(P_r) = E(P_r')/E(P_r) - 1$,规定 T 是对 P_r 的一个有力的混淆变换当且仅当 $T_{\text{pot}}(P_r) > 0$。

4.2 执行代价

混淆程序 P_r' 相对于原始程序 P_r 由于混淆产生的额外开销,通过执行代价来度量。

执行代价的定义如下:

设 T 是一种保持混淆程序 P_r' 和原始程序 P_r 功能一致的混淆过程,混淆后的程序 P_r' 与原始程序 P_r 有 $P_r \overrightarrow{T} P_r'$。令 T 对程序 P_r 的执行代价 $T_{\text{cost}}(P_r)$ 取值如表 1 所示。

表 1 执行代价度量

执行代价	额外消耗的资源情况
非常昂贵	当且仅当 P_r' 的执行较之 P_r 的执行需要额外消耗的资源为指数级别
昂贵	当且仅当 P_r' 的执行较之 P_r 的执行需要额外消耗的资源为一次以上多项式级别
低廉	当且仅当 P_r' 的执行较之 P_r 的执行需要额外消耗的资源为一次多项式级别
无代价	当且仅当 P_r' 的执行较之 P_r 的执行需要额外消耗的资源为零次多项式级别

4.3 弹 性

弹性用来度量混淆程序的抵抗攻击程度,是指针对于反混淆程序的攻击,主要包括反混淆程序运行时的时间和空间代价以及攻击者编写反混淆程序的代价。

弹性定义如下:

设 T 是一种保持混淆程序 P_r' 和原始程序 P_r 功能一致的混淆过程,混淆后的程序 P_r' 与原始程序 P_r 有 $P_r \overrightarrow{T} P_r'$。令 T 对程序 P_r 的弹性 $T_{\text{res}}(P_r) = \text{Resilience}(T_{\text{De}}, T_{\text{Pe}})$,其中 T_{De} 是指反混淆程序运行时的时间和空间代价,T_{Pe} 是指攻击者编写反混淆程序的代价。

规定 T 是对 P_r 的一个单项的混淆变化当且仅当从原始程序 P_r 除去某些信息后无法通过混淆后的程序 P_r' 恢复出 P_r。

5 有效性分析

5.1 强度分析

对于文中提出的 smali 代码混淆方法从数据流、控制流、逻辑流三个层面进行混淆。在数据流混淆方面,将 constr 指定中存储的字符串进行加密存储,这种方法使得逆向分析获取的字符串值是错误的,实验结果如图 3 所示。在控制流混淆方面,在关键函数调用指令和返回值获取指令中间插入不透明谓词,使得攻击者无法获得正确的控制流程,逆向工具逆向结果图 4 所示。在此基础上文中还提出了新的混淆方式,即逻辑流混淆方式,因为 APP 在逆向之后所有的逻辑流都是和 APP 本身运行有关的,所以引入逻辑流混淆,才进行一些无用逻辑的处理和调用。逆向工具逆向结果如图 5 所示,当攻击者利用逆向工具去逆向分析时,结果不仅仅是 APP 的逻辑流还有加入混淆的逻辑流。

5.2 开销分析

对不同的混淆方式分别进行时间开销的分析。对于数据流层面的混淆只是简单的加密混淆对时间开销必然没有影响。对于控制流混淆,插入的无效控制流时间复杂度都为 $O(1)$,并且控制流个数的插入也是有限的,所以并没有影响原程序的复杂度。对于逻辑流混淆,插入的新逻辑流的时间复杂度均为 $O(1)$,这在可以容忍的范围之内。对于空间开销来说,由于混

淆仅仅是针对关键函数进行指令转换、插入定量的无效控制流、修改控制流,以及插入定量的空间复杂度为 $O(1)$ 的逻辑流,因此,smali 代码的混淆方法对于程序空间开销来说影响较小。

5.3　弹性分析

被混淆的程序,在数据流层面其字符串的值被加密混淆,在控制流层面进行的混淆,如图 4 所示,增加了程序的控制流复杂度;除此之外,对于逻辑流进行混淆,如图 5 所示,增加了许多无关程序的逻辑流,加强了程序的复杂度。可以看出文中的代码混淆方法具有较好的弹性。

6　实验结果

如图 1 所示,文中设计并实现了一个 smali 混淆系统,能够有效地抵抗逆向分析。

实验环境:Android7. 0 红米 Note,Idea 开发环境。测试用例说明如表 2 所示。

表 2　测试用例说明

测试用例	指令特点
Test1. apk	Const-str 指令
Test2. apk	函数调用
Test3. apk	多条函数调用
Test4. apk	综合以上

测试用例:本节采用四个自主编写的 Android 应用程序作为测试用例,这四种应用程序是具有代表性的,完全可以表明混淆方法的有效性。

文中从两个方面分析 smali 混淆系统的性能,一方面是通过应用程序保护方法的有效性来评估应用程序保护方法的质量,另一方面通过保护前后应用程序的性能消耗变化评估其质量。对于有效性,之前已经在强度、开销、弹性进行了分析,接下来对性能消耗变化作根据四个测试用例的实验结果进行分析,主要从三个方面分析性能消耗:保护前后的代码长度变化,时间开销变化和空间开销变化。

由于移动设备往往在内存和处理器方面是受限的,所以应用程序的大小以及 APP 启动时间很重要,这关乎到用户体验问题。因此对代码长度变化以及启动时间进行定量分析以此来证明 smali 混淆系统的有效性。

如表 3 所示,混淆程序相比于原始程序代码长度并没有显著变化,主要原因如下:

(1)对 const-str 中的字符串只是加密等长替换,所以对代码长度不会有影响;

(2)只是简单的指令替换;

(3)测试用例的代码长度本身比较小,满足混淆操作的指令并不多;

(4)逻辑流混淆插入的部分的逻辑,是新增的逻辑,由于新增逻辑本身并不大,所以对代码长度和内存消耗也并不是很大。

表 3　代码长度和开销

测试用例	smali 大小/kB		内存消耗/kB	
	混淆前	混淆后	混淆前	混淆后
Test1. apk	1 575	1 591	13 211	17 741
Test2. apk	3 478	3 501	35 672	51 796
Test3. apk	6 891	6 971	67 781	88 791
Test4. apk	10 031	10 141	99 781	134 804

为了更直观地体现混淆操作对应用程序的影响并不大,另外比较了应用程序混淆前后启动时间的变化(见表 4)。

表 4　启动时间变化

启动		Test1. apk	Test2. apk	Test3. apk	Test4. apk
第一次	前	1. 881	2. 903	3. 792	4. 978
启动/s	后	2. 710	3. 813	4. 616	5. 931
第二次	前	1. 481	2. 503	3. 392	4. 178
启动/s	后	1. 602	2. 713	3. 523	4. 321
第三次	前	1. 451	2. 553	3. 322	4. 138
启动/s	后	1. 652	2. 683	3. 593	4. 401

4 个应用程序在第一次启动时,启动时间在混淆前后有明显的变化,分析原因可能是以下原因造成的:

(1)dex 文件类加载器的替换;

(2)混淆方法的查找。但是在第一次启动之后,可以发现启动时间在混淆前后基本没有变化。

7　结束语

针对 Android 应用程序容易被反编译篡改 smali 源码并重新打包重新签名的问题,提出了一种 smali 代码混淆的应用程序保护方法,不同于常规的 Android 应用程序保护,是针对于 smali 层面做的防护,无需获得源码即可进行混淆,并且在常规的数据流和控制流层面使用了新的算法。在数据流层面采用快速加解密的字典式对其进行混淆,在控制流层面,选取特定的位置进行混淆,并且控制流的插入会选取轻量级消耗的控制流,从而减少对程序开销的影响。文中在此基础上还提出了新的混淆层面逻辑流混淆,相对于数据流和控制流来说混淆强度更大,扩展性更强,这使得对 Android 应用程序的防御有很大的提升。同时也从强度、开销、弹性三个方面对 smali 代码混淆方法进行了分析,也从代码长度和启动时间定量进行了分析,实验结果表明,smali 代码混淆方法是有效的。

总之,提出的 smali 代码混淆对于解决 Android 安

全性被破坏的问题有一定意义,但是对于自定义的逻辑流没有进行很好的选择,导致开销上有一定影响。因此下一步工作将针对逻辑流进行优化,以更好地保护代码并且减少消耗。

参考文献:

[1] Kantar. 2016 年 Q1 中国 Android 市场份额[EB/OL]. (2016 - 04 - 19). http://www. ebrun. com/20160419/172891. shtml.

[2] 腾讯移动安全实验室. 2016 年上半年手机安全报告[R/OL]. (2016-07-20). http://slab. qq. eom/news/authority/1505. html.

[3] JESUSFREKEJ. smali/backsmali:an assembler/disassembler for Android´s dex format[EB/OL]. 2011. http://code. google. com/p/samli.

[4] XU J, LI S, ZHANG T. Security analysis and protection based on smali injection for android applications[C]//Algorithms and architectures for parallel processing. Dalian, China:Springer,2014:577-586.

[5] ENCK W,OCTEAU D,MCDANIEL P,et al. A study of android application security[C]//USENIX security symposium. San Francisco, CA, United States: USENIX, 2011: 21 - 36.

[6] FELT A P,HA E,EGELMAN S,et al. Android permissions: user attention, comprehension, and behavior[C]//Proceedings of the eighth symposium on usable privacy and security. [s. l.]:[s. n.],2012:1-14.

[7] FAHL S, HARBACH M, MUDERS T, et al. Why eve and mallory love android: an analysis of android SSL (in)security[C]//ACM conference on computer & communications security. Raleigh, North Carolina, USA: ACM, 2012: 50-61.

[8] BERGER B J, BUNKE M, SOHR K. An Android security case study with Bauhaus[C]//18th working conference on reverse engineering. Limerick, Ireland: IEEE, 2011: 179 - 183.

[9] 牛豪飞. Android 应用保护方案的设计与实现[D]. 北京:北京邮电大学,2018.

[10] 尉惠敏. 面向 Android 系统的 App 安全加固技术的研究与实现[D]. 西安:西安理工大学,2019.

[11] 彭守镇. Android APP 加固方案的研究[J]. 软件工程,2019,22(6):8-12.

[12] 吕苗苗. 基于 JAVA 的安卓应用代码混淆技术研究[J]. 山东农业大学学报:自然科学版,2019,50(4):671-674.

[13] 郑 琪,徐国爱. 面向 Android 移动应用的控制流混淆[EB/OL]. [2014-12-25]. http://www. paper. edu. cn/releasepaper/content/201412-783.

[14] 刘方圆,孟宪佳,汤战勇,等. 基于 smali 代码混淆的 Android 应用保护方法[J]. 山东大学学报:理学版,2017,52(3):44-50.

[15] 吴 林. 面向 Android 应用的 dalvik 字节码混淆技术研究[D]. 成都:电子科技大学,2018.

[16] VARGAS R J G,ANAYA E A,HUERTA R G,et al. Security controls for Android[C]//2012 fourth international conference on computational aspects of social networks (CASoN). Sao Carlos, Brazil: IEEE, 2012: 212-216.

[17] 刘金梁. Android 平台软件安全防护技术的研究与实现[D]. 北京:北京邮电大学,2015.

[18] 郑 琪. 面向 Android 智能手机终端应用程序的代码混淆算法研究与实现[D]. 北京:北京邮电大学,2015.

[19] COLLBERG C, THOMBORSON C, LOW D. A taxonomy of obfuscating transformations[R]. New Zealand: The University of Auckland, 1997.

[20] KOVACHEVA A. Efficient code obfuscation for Android[C]//Advances in information technology. Bangkok, Thailand: Springer, 2013: 104-119.