一种基于双层语义的Android原生库安全性检测方法

叶益林 吴礼发 颜慧颖
(解放军理工大学指挥信息系统学院 南京 210007)

摘 要 原生代码已在Android应用中广泛使用,为恶意攻击者提供了新的攻击途径,其安全问题不容忽视。当前已有Android恶意应用检测方案,主要以Java代码或由Java代码编译得到的Dalvik字节码为分析对象,忽略了对原生代码的分析。针对这一不足,提出了一种基于双层语义的原生库安全性检测方法。首先分析原生方法Java层语义,提取原生方法函数调用路径,分析原生方法与Java层的数据流依赖关系以及原生方法函数调用路径的入口点。对于原生代码语义,定义了数据上传、下载、敏感路径读写、敏感字符串、可疑方法调用5类可疑行为,基于IDA Pro和IDA Python对原生代码内部行为进行自动分析。使用开源机器学习工具Weka,以两层语义作为数据特征,对5336个普通应用和3426个恶意应用进行了分析,最佳检测率达到92.4%,表明所提方法能够有效检测原生库的安全性。

关键词 Android应用,恶意应用检测,语义,原生库,机器学习
中图法分类号 TP393.08 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.06.027

Two-layer Semantics-based Security Detection Approach for Android Native Libraries

YE Yi-lin WU Li-fa YAN Hui-ying
(Institute of Command Information System, PLA University of Science and Technology, Nanjing 210007, China)

Abstract Native code has been widely used in Android applications, providing a new attack vector for attackers, which raises increasing security concerns. Existing Android malware detection approaches mainly focus on the analysis of Java code or the Dalvik code compiled from Java code, ignoring the native code used in Android applications. To combat this emerging threat, this paper proposed a novel two-layer semantics-based security detection method for Android native libraries. To begin with, on the base of native method call paths, the semantics of native method in Java layer is extracted by analyzing the data dependence between native methods and Java methods and the type of the entry points of native method call paths. For semantics of native code in native layer, five kinds of suspicious behaviors are defined, including data uploading, data downloading, reading or writing in sensitive system paths, sensitive strings, suspicious calling of Java methods. More specifically, IDA Pro and IDA Python are utilized to analyze the behaviors of native code mentioned above. Experiments are evaluated using the open source machine learning tool Weka with 5336 benign Android applications and 3426 Android malware, the results of which show that the best accuracy achieves 92.4%. It proves that our method can effectively detect the security of native libraries used in Android applications.

Keywords Android application, Malware detection, Semantics, Native library, Machine learning

1 引言

360互联网安全中心最新发布的《2015中国手机安全状况报告》[1]显示全年累计截获1874万个Android平台新增恶意程序,分别是2013和2014年的27.9倍和5.7倍。面对日益严峻的Android应用安全态势,学术界提出了多种恶意应用检测方案,这些方案可分为静态分析和动态分析两大类。基于静态分析的Android恶意应用检测方案主要以基于特征和基于机器学习两类为主。前者的基本思路是提取Android应用源码中的某些特征,并基于这些特征对Android应用进行分类,典型的代表有Kirin[2],DroidChecker[3],Apposcopy[4]。相比基于特征的检测方案,基于机器学习的检测方案从更广的视角对Android应用特征进行细粒度的分析,提取多样应用特征,通过机器学习训练分类器对Android应用进行分类,典型的代表有APIMiner[5],Drebin[6]。相比静态分析,动态分析是一种重量级的分析方法,通常需要修改Android系统并在其中置入监控模块,或者将Android模拟器运行于某个受控的仿真环境中,根据Android应用的运行时行为特征判断Android应用恶意与否。当前已有的基于动态分析的Android恶意应用检测典型方案有DroidScope[7],AppsPlay-

到稿日期:2016-05-13 返修日期:2016-10-08 本文受江苏省自然科学基金项目(BK20131069)资助。
叶益林(1987—),男,博士生,主要研究方向为移动应用安全,E-mail:my_etsi@163.com;吴礼发(1968—),男,教授,博士生导师,主要研究方向为网络安全;颜慧颖(1993—),女,硕士生,主要研究方向为移动应用安全。

ground[8]，AASandbox[9]等。上述方案都将 Java 代码或由 Java 代码编译后的 Dalvik 字节码作为分析对象，缺乏对原生代码的安全性分析。

Google 推出的 NDK(Native Developer Kit)原生代码开发组件，使得使用原生代码(C/C++)开发 Android 应用成为可能。凭借 JNI 技术，Android 应用能够直接复用已有原生库，提高开发效率。对于计算密集型任务，如游戏引擎、视频、音频处理和物理仿真等，使用原生代码来实现能够提升 Android 应用程序的性能[10]。与此同时，使用原生代码也为 Android 恶意应用检测带来了新的问题和挑战。原生代码以二进制指令保存在原生库中，相较 Java 代码具有更高的反编译难度，内部行为通常更难以分析，能够达到隐藏功能逻辑实现细节的目的。这一特点已被恶意应用制作者利用，其通过将恶意逻辑隐藏在原生层，以对抗已有的检测技术。危害极大的“百脑虫”、“蜥蜴之尾”等木马[11]在原生库中实现其恶意逻辑，而已有的 Android 恶意应用检测方案难以判别其恶意与否。当前已有 Android 恶意检测方案都将分析对象聚焦在 Java 层代码或源于 Java 代码的 Dalvik 字节码，因而只能分析 Java 层语义信息。由于缺乏对原生代码语义的分析，仅通过 Java 层语义难以构建完整的程序语义，因此无法准确反映程序的行为特性，导致无法检测将恶意逻辑隐藏于原生库实现的 Android 恶意应用。

针对已有 Android 恶意应用检测方案无法对原生库文件的安全性进行检测的不足，本文提出一种基于双层语义的 Android 原生库安全性检测方案，同时分析了 Java 层和原生代码层语义信息。该方案以原生方法在 Java 层的调用点为切入点，首先分析 Java 层语义。首先提取原生方法函数调用路径，对其进行数据流分析，判断原生方法是否与 Java 层存在敏感数据依赖关系。当存在敏感数据依赖关系时，分析原生方法函数调用路径入口点，判断原生方法函数调用路径是否由用户输入触发。在分析 Java 层语义的基础上，利用静态分析工具 IDA Pro 和 IDA Python 分析原生方法内部行为，具体通过分析原生方法内部函数交叉引用和数据交叉引用，来判断原生方法内部是否包含执行敏感 shell 命令、调用敏感 Java 方法等敏感行为。最后综合两层语义，提取原生库基本特征，利用机器学习方法判断原生库的安全性。综上所述，本文的主要贡献有：1)提出一种基于双层语义的 Android 原生库安全性检测方法；2)实现了原型系统，并对 5336 个普通应用和 3426 个恶意应用进行分析，最佳检测率能够达到 92.4%，表明本文提出的方法能够有效检测 Android 原生库文件的安全性。

本文第 2 节详细阐述提出的方法；第 3 节阐述实验设置与结果分析；第 4 节为相关工作对比；最后总结全文，并指明下一步工作。

2 方法描述

针对当前已有方案的不足，本文提出了一种基于双层语义的 Android 原生库检测方法，方法流程如图 1 所示，分为预处理和双层语义分析两个阶段。预处理阶段完成原生库文件过滤和 Native 方法提取两个功能，在双层语义分析阶段完成 Java 层语义和 Native 层语义的提取和分析。

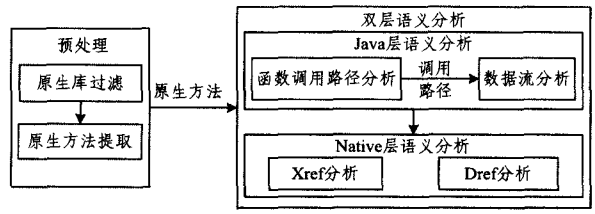

图 1 基于双层语义的 Android 原生库检测方法流程

2.1 预处理

预处理阶段分为第三方原生库过滤和原生方法提取两个步骤。

1)第三方原生库过滤

Android 应用程序通常包含一些商用或免费第三方原生库，这些原生库的安全性已知，为消除这些原生库造成的不必要的干扰，在预处理阶段，对 Android 应用程序中的原生库文件进行过滤，过滤掉已知的商用或免费第三方原生库，未知原生库留待后续分析。

2)原生方法提取

原生方法是将原生库暴露给 Java 层以调用其功能的函数接口，是原生库的功能实体，原生库文件的安全性可由原生方法探知。基于原生库的 Android 恶意应用，恶意逻辑通常置于原生方法内部实现，因而以原生方法作为分析原生库文件安全的入口点。

在原生方法提取阶段，需要提取 Android 应用程序中全部原生方法和原生方法调用点处对应的 Java 方法。在 Android 应用内部，原生代码以 SO(Shared Object)库的格式保存在 lib 目录下。Android 系统通过 System 类的静态方法即 loadLibrary 方法来加载原生库，并通过 native 关键字声明原生方法。由此可以根据方法签名是否包含 native 关键字，得到全部原生方法。得到 Android 应用程序中的所有原生方法后，再对 Java 源码或 Dalvik 字节码进行函数调用分析，可以提取原生方法调用点处的 Java 方法。

2.2 双层语义分析

在双层语义分析阶段，分别对原生方法的 Java 层和原生层语义进行分析。本文从数据依赖关系的角度完成原生方法 Java 层语义分析，通过对原生方法内部敏感行为进行分析来完成原生代码层语义分析。

2.2.1 Java 层语义分析

首先提取原生方法在 Java 层的函数调用路径，过滤其中的安全无关路径，对敏感路径进行数据流分析，确定原生方法与 Java 层的数据依赖关系。与此同时，对敏感调用路径进行入口点分析，判断调用路径入口点类型。最后根据数据流依赖关系和入口点类型判别原生方法 Java 层语义的安全性。

(1)原生方法函数调用路径敏感性分析

应用程序的运行时行为由执行路径上对应函数的行为决定，原生方法在 Java 层的函数调用路径能够反映程序行为语

义信息。通过分析程序行为语义信息，可以判断原生方法在 Java 层的执行路径是否属于异常行为。本节通过分析函数调用路径上的数据依赖来判定调用路径对应的程序行为语义是否异常。

本文直接对 Android 应用程序 APK 文件中的 dex 字节码文件进行分析，利用 Androguard[12] 中的 dvm 模块获取所有原生方法对象和 analysis 模块分析方法间的调用关系，在此基础上提取原生方法函数调用路径。通常情况下，原生方法函数调用路径数量较多，包含许多安全无关路径，只需分析其中的敏感路径，利用算法 1 将滤安全无关路径。算法 1 将全部原生方法函数调用路径作为输入，判断每条调用路径上的方法节点是否包含权限方法（算法第 4 行），或包含敏感广播事件，如地理位置改变（算法第 7 行），或随应用程序启动而被调用的方法（是否由 MainActivity 的 onCreate 方法调用，算法第 10 行），或运行在后台服务的方法（算法第 13 行）。当满足上述条件之一时，将调用路径标记为敏感路径，否则为安全无关路径。若原生方法某条函数调用路径 Patha 中包含其他原生方法（算法第 16 行），需要对被包含原生方法对应的函数调用路径进行过滤，判断其中是否包含敏感路径（算法第 17 行，递归调用 containsSensitivePath 过程）。若包含，则 Patha 标记为敏感路径；若不包含，则分析 Patha 中其他 Java 方法是否包含权限方法和敏感广播事件。

算法 1　原生方法调用路径过滤算法

```
native_method_callsites←get_native_method_callsite()
Procedure containsSensitivePath (native_method_callsite)
1. native_method_calltraces ← extract_calltrace(ncallsite)
2.   for clist in ncalltraces do
3.    for method in clist do
4.      if isPermMethod(method)
5.          flag←True
6.          break
7.      else if isSensitiveBroadcast(method)
8.          flag←True
9.          break
10.     else if isBootup(method)
11.         flag←True
12.         break
13.     else if isInservice(method)
14.             flag←True
15.             break
16.     else if isNative(method)
17.       if containsSensitivePath(method) is not None
18.             flag←True
19.             break
20.     else
21.         continue
22.     end if
23.   end for
24.   if not flag:
25.       native_method_calltraces. delete(clist)
26.     end if
27.   end for
28. output native_method_calltraces
```

（2）敏感调用路径数据流分析

在对敏感调用路径进行数据流分析前，需要判定原生方法与调用处 Java 方法是否存在数据流依赖。通常数据流依赖分为参数依赖、返回值依赖两类。具体地，参数依赖关系包含以下几种情形：1）依赖 Java 方法所属类（包含父类）的成员变量；2）依赖 Java 方法内部其他 Java 方法的返回值；3）依赖 Java 方法形参；4）依赖 Java 方法内部的局部变量；5）上述依赖关系的两种或多种；6）无依赖关系，原生方法无参数。

返回值与 Java 层的依赖关系包含以下几种情形：1）返回值改变 Java 方法所属类的字段的值、全局变量；2）返回值作为其他 Java 方法的参数；3）返回值改变 Java 方法局部变量的值；4）无依赖关系，原生方法返回值为空。

本文只对原生方法参数与 Java 层之间的数据依赖关系进行分析，对于返回值与 Java 层依赖关系的分析将在未来工作中完成。首先对原生方法调用处 Java 方法进行过程内数据流分析，确定原生方法参数与调用处 Java 方法的数据流依赖关系。当原生方法与调用处 Java 方法存在数据流依赖关系时，对原生方法的敏感调用路径进行数据流分析。若原生方法参数依赖 Java 方法所属类的成员变量，则需要对该类的成员变量进行活跃变量分析。本文采用保守的分析策略，只对原生方法函数调用路径中方法节点对类成员变量数据依赖关系的影响进行分析，结合过程内数据流分析和过程间数据流分析，确定类成员变量的值在作为实参传入原生方法前，最后一次改变时的数据依赖关系，从而确定原生方法参数的数据依赖关系。实际应用程序中，部分敏感路径无关方法可能会影响该类成员变量的数据依赖关系。可以使用静态污点分析方法，确定敏感路径无关方法通过类成员变量对原生方法参数数据依赖的影响，这部分工作将放在未来工作中完成。若原生方法参数依赖 Java 方法内部其他 Java 方法的返回值，由于采用静态分析方法无法确定该 Java 方法的返回值，本文采用一种启发式的分析方法，对该 Java 方法的实参进行分析，将实参的数据依赖源作为原生方法的数据依赖源。当原生方法参数依赖调用处 Java 方法形参时，对原生方法函数调用路径进行过程间数据流分析，确定调用处 Java 方法形参的数据依赖源。当原生方法参数依赖调用处 Java 方法的局部变量时，对局部变量进行活跃变量分析，确定局部变量在传入原生方法前最后一次值改变的数据依赖关系。

完成原生方法参数数据流依赖分析后，判断其是否满足以下依赖：1）原生方法参数依赖系统敏感数据，如联系人、地理位置信息、短信息等；2）原生方法参数依赖的数据源与敏感行为有关，如加解密操作、网络下载等。

当满足上述依赖之一时，对函数调用路径进行入口点分析，根据函数调用路径入口点类型判断函数的执行路径是否合乎用户意图。所谓入口点是指触发函数调用路径执行的外界条件。对于基于事件驱动的 Android 应用程序，函数调用

路径入口点类型分为以下两种:用户输入事件(数据输入或动作操作)和系统事件。当用户输入事件如用户点击按扭、滑动屏幕发生时,将直接触发与之关联的执行路径。当系统事件如位置变更、接收短信、来电等发生时,Framework 层调用对应的回调函数,触发与之关联的执行路径。

本文采用文献[13]提出的入口点生成算法,获取原生方法的函数调用路径的全部入口点;然后使用算法 2 对原生方法函数调用路径进行分类,以原生方法敏感函数调用路径作为参数。对于其中的每条路径,分别提取其入口点和数据依赖类型。若入口点为用户输入事件且存在敏感数据依赖,则将原生方法标记为风险原生方法;若入口点不为用户输入事件且存在敏感数据依赖,则将原生方法标记为高危原生方法;其他情况,将原生方法标记为普通原生方法。

算法 2 原生方法函数调用路径分类算法

```
Calltrace:原生方法函数调用路径
SensitiveDepen:敏感数据依赖类型
UserEventEntries:用户事件入口点
SystemEventEntries:系统事件入口点
Normal←Φ:普通原生方法函数调用路径
Risky←Φ:风险原生方法函数调用路径
Dangerous←Φ:高危原生方法函数调用路径
for ct in Calltraces do
  entry←Extract_EntryPointer(ct)
  depen←Analyze_DataDenpendence(ct)
  if entry belongs to UserEventEntries and depen belongs to Sensi-
  tiveDepen
      Risky. append(ct)
  else if depen belongs to SensitiveDepen
      Dangerous. append(ct)
  else
      Normal. append(ct)
  endif
end for
output Normal,Risky,Suspicious
```

2.2.2 原生代码层语义分析

由于原生方法以二进制数据的形式保存于原生库文件中,难以直接对其进行分析,因此本文借助静态分析工具 IDA Pro 分析原生方法是否包含以下几类敏感行为。

(1)数据上传

原生层数据上传相比 Java 层数据上传更隐密,这一特点常被 Android 恶意应用程序利用,在原生层上传敏感数据、泄露用户隐私。本文采用如下方法判断原生方法内部是否存在数据上传行为。首先使用 IDA Pro 对原生库进行分析,然后利用 IDA Python 编写脚本程序提取原生方法对应的 Xref(内部函数调用交叉引用)和 Dref(内部数据交叉引用),分析原生方法的 Xref 引用是否包含 C/C++网络数据上传的相关 API,如 socket,send 等。若包含,则标记原生方法内部存在数据上传行为。

(2)数据下载

与数据上传一样,原生层数据下载同样具有隐密性。依赖原生库的 Android 恶意应用程序通常在原生层连接服务器,下载恶意代码或接收恶意指令,进而实施恶意攻击,其常见的攻击模式为:运行原生方法收集终端信息,回传服务器。服务器根据终端类型,发回对应的攻击指令,客户端根据指令下载对应的恶意代码,运行恶意代码并实施恶意攻击。分析原生方法内部是否存在数据下载行为的基本思路为:提取原生方法的所有 Xref 引用,分析各个 Xref 引用是否调用 C/C++网络下载的相关API,如 socket,recv 等。若调用了相关网络下载的相关 API,则标记原生代码存在网络下载行为。

(3)敏感路径读写

某些依赖原生库实现恶意逻辑的 Android 恶意应用程序运行后,会篡改系统文件,在系统敏感目录中(/system、/system/sbin)创建文件或目录。如流行甚广的"舞毒蛾"木马启动后,会在/system/bin 及/data 目录下创建多个文件和目录,并篡改系统文件/system/etc/install-recovery. sh。

判断原生方法内部是否存在敏感路径读写的方法为:提取原生方法包含的所有 Xref 引用,分析每个 Xref 是否调用 open 或 write 方法。若调用,则分析该 Xref 引用对应的 Dref 引用,判断 Dref 引用中是否包含系统敏感路径,若包含则说明存在敏感路径读写行为。

(4)调用敏感 Java 方法

JNI 技术支持 Java 代码和原生代码相互调用,原生库中 C/C++代码调用 Java 代码需要经过类映射、新建类对象、映射类方法、调用类方法 4 个步骤。在原生代码层调用 Android Framework 相关 API 与在 Java 层直接调用相比具有极强的隐蔽性。恶意应用可以利用这一特性,在原生代码层调用敏感 Java 方法,实施恶意行为。一种可能的攻击方式为:在原生层调用相关 API 获取用户隐私信息后,连接网络上传至服务器,窃取用户隐私信息。

原生代码调用 Java 方法需要将该 Java 所属的全路径类名作为参数,通过 FindClass()方法获取类引用;将 Java 方法名作为参数,通过 GetMethodID()方法获取对应 Java 方法对象引用。根据这一特性,采用了一种启发式的方法来判断原生方法是否调用 Java 方法。首先编写 Java 类、方法对应的正则表达式;然后提取原生方法的 Dref 引用,利用正则表达式匹配,判断原生的 Dref 中是否包含 Java 类、方法对应的字符串。若正则匹配不为空,再利用 PScout[14] 提供的 Android 权限映射关系确定 Java 方法是否为权限方法。

(5)敏感字符串

Android 恶意应用程序在原生代码层执行恶意逻辑,实施各类恶意行为,如 root 设备、静默安装其他可疑应用、主动连接控制服务器、泄露用户隐私,其通常使用许多常量字符串,如高危 shell 命令对应的字符串、用户隐私数据对应的字符串。根据这一特点,本文定义了原生库中 4 类敏感字符串,从敏感字符串的角度推测潜在的恶意行为,具体的实现方法为获取原生方法的 Dref 引用及其 Xref 引用对应的 Dref 引用,利用正则表达式匹配判断 Dref 引用中是否包含以下 4 类敏感字符串。

1）高风险 shell 命令

高风险 shell 命令包括/system/bin、/system/xbin 下的 shell 命令，如用户切换命令 su、安装应用命令 pm install 等。

2）敏感信息相关字符串

敏感信息相关的字符串包括用户相关的隐私信息，如 IMEI、联系人号码、地理位置信息，以及一些网络相关的字符串，如 URL、IP 地址、电子邮箱等。

3）敏感文件后缀名相关字符串

敏感文件后缀名类字符串主要包括 exe，sh，so，jar，zip，rc 6 种类型。

4）安卓模拟器相关字符串

之所以将安卓模拟器相关的字符串列为敏感串字符串，是由于某些 Android 恶意应用会检测运行环境，而常用的手段为读取系统配置文件的相关字段，与模拟器环境同类字段进行对比，根据比较结果判别是否运行在模拟器中。这些典型的字符串有 sdk，golden fish 等。

完成原生代码层语义分析后，将分析结果与 Java 层语义分析结果进行综合，基于表 1 所列的规则确定原生方法语义安全的风险等级。

表 1　原生方法语义安全判定规则表

敏感行为 \ 风险等级	普通	风险	高危
数据上传/下载	风险	风险	高危
敏感路径读写	高危	高危	高危
调用敏感 Java 方法	风险	高危	高危
敏感字符串	风险	高危	高危

3　实验分析

原型系统由语义特征提取、机器学习两大模块组成。语义特征提取模块由 Python 实现，其中 Java 层语义分析子模块基于 Androguard 实现，对 Androguard 内部 apk，dvm，analysis 3 个核心模块进行了拓展；原生代码语义分析子模块基于 IDA Pro 实现，使用 IDA Python 实现具体业务逻辑。机器学习模块由 Java 实现，利用开源机器学习工具 Weka[15] 提供的机器学习算法完成 Android 原生库安全性的检测功能。

Android 应用样本集全部通过网络获取，当前已有 16116 个普通应用和 14448 个恶意应用，前者通过网络爬虫 AppChina[16]、安智[17]等应用市场获取，后者从在线恶意软件 VirusShare[18] 中获取。利用 Androguard 对全部样本进行预分析，发现其中包含原生库的普通应用、恶意应用分别为 5336 个和 3426 个，将包含原生库的普通应用和恶意应用作为实验数据集，并随机分成 3 份，从中选取 2 份，以 3557 个普通应用和 2284 个恶意应用作为训练集，剩下 1779 个普通应用和 1142 个恶意应用作为测试集。

3.1　实验环境

全部实验在 ThinkPad W540 上完成，其基本配置为：16G 内存、处理器为 Intel（R）Core（TM）i7-410MQ CPU @ 2.5GHz，操作系统为 Ubuntu 14.04，IDA Pro 版本为 6.4、Weka-3-8-0。

Weka 数据文件特征由 Java 层和原生代码层两类特征组成。Java 层特征包括：原生方法调用处 Java 方法是否随应用程序启动而运行（boot_up）、原生方法是否与 Java 层有敏感数据依赖关系（data_flow_dependency）、原生方法调用路径是否依赖敏感广播（sensitive_broadcast）、原生方法是否运行在后台服务内（run_within_service）、原生方法调用路径上是否包括敏感权限方法（perm_method）。原生代码层特征包括：模拟器相关字符串（emulator_string）、网络地址类字符串（network_address）、文件后缀名（file_extension）、隐私类字符串（privacy_string）、shell 命令类字符串（shell_command）、网络行为（network_behavior）、可疑行为（suspicious_behavior）、敏感 Java 方法调用（java_method_called）。

```
@relation android. so_security. detection
@attribute boot_up {true,false}
@attribute data_flow_dependency {normal,dangerous,risky}
@attribute emulator_string numeric
@attribute file_extension {exe,jar,rc,sh,so,zip}
@attribute java_method_called {permissions…}
@attribute network_address numeric
@attribute network_behavior {download,upload}
@attribute perm_method {permissions…}
@attribute privacy_string numeric
@attribute run_within_service numeric
@attribute shell_command numeric
@attribute suspicious_behavior {true,false}
@attribute sensitive_broadcast {true,false}
@attribute class {benign,malicious}
```

3.2　评价指标

（1）True Positive Rate（TPR）：用于衡量正确检测恶意应用的能力，其值为正确检测恶意应用的个数与恶意应用总数之比，如式（1）所示。

$$TPR=\frac{TP}{TP+FN} \tag{1}$$

其中，TP 表示正确检测出的恶意应用的个数，FN 为恶意应用漏报为普通应用的个数。

（2）True Negative Rate（TNR）：用于衡量实验正确检测普通应用的能力，其值为正确检测的普通样本的个数与普通样本总数之比，如式（2）所示。

$$TNR=\frac{TN}{TN+FP} \tag{2}$$

其中，TN 表示检测出的普通应用的个数，FP 为普通应用误报为恶意应用的个数。

（3）Accuracy：用于衡量实验整体的检测率，其值为正确检测的样本个数与样本总数之比，如式（3）所示。

$$Accuracy=\frac{TP+TN}{TP+FN+TN+FP} \tag{3}$$

3.3　实验结果

3.3.1　数据分析

本文对全部数据集进行分析，对两类应用中原生库使用频次、平均原生方法的个数、原生方法调用路径平均 Java 方

法个数进行了统计，具体如表 2 所列，两类应用的原生方法调用路径包含的 Java 方法数相差无几，而平均原生方法大相径庭。普通应用调用的原生方法个数明显多于恶意应用，这是由于普通应用大量复用已有第三库（游戏引擎、3D 渲染等）和使用原生代码实现其核心业务逻辑。对于恶意应用，其使用原生库的目的更多的是完成恶意逻辑规避查杀，功能较为单一，从而致使其内部平均原生方法数明显小于普通应用内部原生方法数。

表 2 原生库相关数据统计

应用类型	使用频次	平均原生方法	调用路径平均原生方法
普通应用	75841	78.3	5.8
恶意应用	60587	33.9	6.9

对数据集中 Android 应用包含的原生库进行分析，发现了大量同名原生库，其中许多为商用或开源第三方库，表 3 列出了重名次数排名前 20 的原生库。计算重名原生库 MD5 值，发现许多 Android 应用出于不同的功能需求，对原始第三方库进行了定制改动，使得同名原生库 MD5 的值各不相同。

表 3 原生库重名信息

原生库名	重名次数
libbspatch	1606
liblocSDK5	1494
libweibosdkcore	1174
libgif	1077
libtpnsWatchdog	1054
libsecmain	999
libtpnsSecurity	902
libentryex	767
libmsc	756
liblocSDK3	683
libunity	640
libexec	625
libmono	607
libsmssdk	596
libgetuiext	489
liblocSDK4d	475
libexecmain	474
libsecexe	466
libumeng_opustool	459
libvinit	425

3.3.2 检测结果

提取两层语义特征后，选取了 3 种不同的机器学习算法来对比其性能，这 3 种机器学习算法分别为 RandomForest，LibSVM，NaiveBayes。实验利用 Weka 自带的 10 折交叉确认验证（10-fold across validation）方法训练测试集来获取分类器，。10 折交叉确认验证是指将样本分成 10 份，每次不重复地选择其中一份作为测试集，余下数据作为训练集，利用机器学习算法对训练集数据进行训练得到分类器，再利用分类器对测试集进行检测，如此重复 10 次，取 10 次测试集检测结果的均值作为测试结果。为了提高检测精度，每种机器学习算法都进行 10 次 10 折交叉确认验证，取其中精度最高的分类器作为该算法在测试阶段的分类器。

进行了 3 组对比实验，第一组以 Java 层语义作为数据特征，第二组以 Native 层语义作为数据特征，第三组以双层语义作为数据特征，实验结果分别如图 2—图 4 所示。以单层语义作为数据特征时，原生代码语义的检测性能优于 Java 层语义的检测性能。这是由于 Native 层语义直接描述原生库的安全性，偏向于恶意应用。具有 Native 层语义的 Android 应用的原生库具有典型的恶意特征。而 Java 层语义特征对于使用原生代码的 Android 应用而言具有普适性，良性应用和恶意应用都可以具有 Java 层语义特征。而且 Java 层语义特征只能通过原生方法在 Java 层调用上下文来判别原生库的安全性，仅以此为特征将导致较高的误报和漏报。将两层语义结合起来后，双层语义能够同时刻画原生库在 Java 层和 Native 层的可疑行为，以此作为数据特征能够全面描述原生库安全性的本质，由此检测性能明显提升。

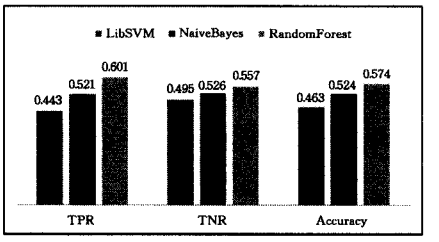

图 2 Java 层语义特征的检测性能

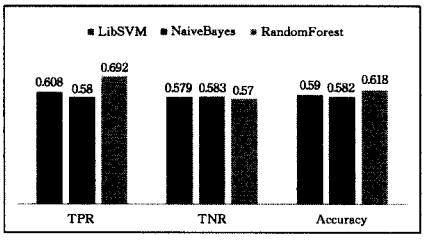

图 3 Native 层语义特征的检测性能

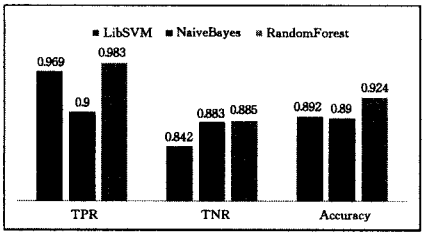

图 4 双层语义特征的检测性能

在所有机器学习算法中，NaiveBayes 的分类效果最差，检测率为 0.882；RandomForest 分类结果最佳，检测率为 0.924。NaiveBayes 基于古典贝叶斯概率模型，其实现简单，但由于假定应用特征之间相互独立，导致 NaiveBayes 算法容易引入误差，检测率偏低。RandomForest 算法训练得到的分类器包含多个决策树，最后的分类结果由决策树判决结果的众数决定，这一特性使得 RandomForest 性能突出。3 种机器学习算法的检测结果中，TPR 均高于 TNR，表明训练得到的分类器正确判别恶意原生库的能力强于判别普通原生库。TNR 低于 TPR，分类器容易将普通应用库误判为恶意应用原生库，表明普通应用原生代码的程序语义具有模糊性，其原生代码相关程序语义与恶意应用原生代码相关程序语义存在类似性，具有某些恶意特征，导致训练得到的分类器存在偏差，致使实验结果 TNR 低于 TPR。

3.4 时间性能分析

时间性能主要受两个因素影响：Java 层语义分析时间和原生代码层敏感行为分析时间。Java 层语义分析时间主要与原生方法的个数及原生方法函数调用路径上包含方法节点的个数有关。原生方法的个数决定了原生方法的函数调用路径数，调用路径上的方法节点的个数影响了 Java 层数据流和调用路径入口点分析时间。原生代码层敏感行为分析时间由原生库文件大小和原生方法个数决定。原生库大小决定 IDA Pro 预处理分析所需的时间，原生方法个数影响原生方法内部行为语义分析时间。

样本平均分析时间为 79.1s，图 5、图 6 分别显示了样本分析时间与原生库文件大小、原生方法个数的关系。图 5 中原生库文件大小与样本分析时间大致成正相关关系。当原生库增大时，分析时间也随之增加。图 5 中原生库大小主要集中在 0.5～5M。如图 6 所示，总体上样本分析时间随原生方法个数增加而增长。当原生方法个数小于 30 时，分析时间大部分在 45s 以内；当原生方法个数大于 30 时，分析时间明显增加。

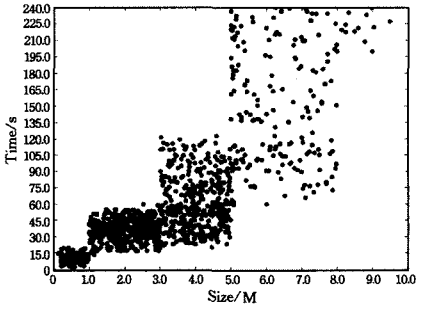

图 5　原生库文件大小与解析时间的关系

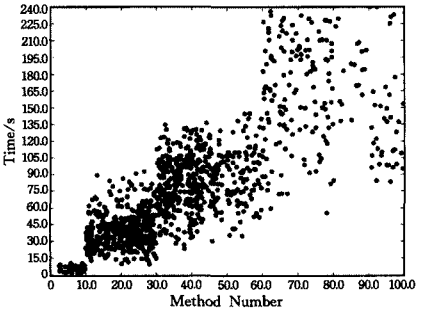

图 6　原生方法个数与解析时间的关系

4 相关工作对比

Siefers 等[19]针对 Java 应用程序内部原生代码的安全问题进行了分析，基于 SFI(Software-based Fault Isolation)技术提出了框架 Robusta，并将其集成至 JVM，使得 JVM 能够将原生代码进程与 Java 代码进程隔离，使原生代码运行在隔离的沙箱中，限制原生代码读写 Java 应用程序进程空间数据的能力，从而确保整个 Java 应用程序进程的安全性。Siefers 提出的方案并未针对 Android 应用内部原生代码的安全性进行分析，而本文提出的方法对此进行了深入的分析。

NativeGuard[20]从开发者使用未知内部实现逻辑第三方原生库可能产生的未知安全风险的角度，分析了第三方原生库的安全问题。NativeGuard 的核心思路与 Robusta 相同，提出了一种基于进程隔离的安全方案，对原生库与其他 Android 组件进行进程隔离，将 Android 应用程序的原生库与其他 4 大组件隔离到不同的进程中运行，从而剥离原生代码能够访问应用程序的地址空间的能力。与此同时，限制原生库可以使用的 Android 权限，确保原生代码只获取完成其功能所需的最小权限。

Vitor 等[21]对流行 Android 应用使用原生库的现状进行了分析，发现原生代码存在安全风险，提出了原生代码安全增强方案。针对进程隔离方案的不足，采用减少原生代码攻击面的安全策略，限制原生代码能够发起的系统调用和 Java 方法，并采用动态分析技术监控原生代码运行期间的动态行为，收集系统调用、原生代码与 Java 层的交互行为。

NativeGuard 和 Vitor 提出的方案都是从原生库使用的角度对原生库进行安全分析，都依靠进程隔离的手段来确保原生库运行时的安全性。与此不同，本文是从原生库检测的角度对原生库的安全性进行分析。采用静态分析方法对原生代码的两层程序语义进行分析，并在此基础上判别原生代码安全与否。相比之下，本文提出的方法不需要动态分析 Android 应用，能够先行检测原生库的安全性。

结束语　针对当前已有 Android 恶意应用检测方案对原生代码安全问题分析不足的问题，本文提出了一种基于双层语义的原生库安全性检测方法，通过提取原生方法在 Java 层和原生代码层的语义信息，利用机器学习算法对原生库文件的安全性进行分类，实验表明本文提出的方法能够以较高的准确率检测原生库文件的安全性。

下一步工作将采用静态污点方法分析原生方法敏感路径之外的 Java 方法对原生方法 Java 层数据流依赖的影响，以及分析原生方法函数与 Java 层的控制流依赖的关系，将数据流与控制流分析相结合，更加精准地解析 Java 层语义，解决良性应用和恶意应用原生代码程序语义存在的模糊性问题；使用 IDA Pro 动态调试技术监测原生代码动态行为，解决原生代码中可能出现的代码混淆、程序加壳等对抗静态分析的问题。

参 考 文 献

[1] 360：2015 年度中国手机安全状况报告[EB/OL]. http://useit.baijia.baidu.com/article/313267.

[2] ENCK W, ONGTANG M, MCDANIEL P, et al. On lightweight mobile phone application certification[C]// Computer and Communications Security. 2009：235-245.

[3] CHAN P P, HUI L C, YIU S M, et al. DroidChecker：analyzing android applications for capability leak[C]// Wireless Network Security. 2012：125-136.

[4] FENG Y, ANAND S, DILLIG I, et al. Apposcopy：semantics-based detection of Android malware through static analysis[C]// Foundations of Software Engineering. 2014：576-587.

（下转第 173 页）

crypto system in ad hoc networks[C]// Wireless Communications and Mobile Computing. 2007:909-917.

[7] DENG H, AGRAWAL D P. TIDS: threshold and identity-based security scheme for wireless ad hoc networks[J]. Ad Hoc Networks, 2004, 2(3): 291-307.

[8] LI J F, WEI D W, KOU H Z. Identity-based and threshold key management in mobile ad hoc networks[C]// International Conference on Wireless Communications, Networking and Mobile Computing 2006WiCOM 2006). 2006: 1-4.

[9] ZHANG C R, ZHANG Y Q, LI F G, et al. New signcryption algorithm for secure communication of ad hoc networks[J]. Journal on Communications, 2010, 31(3): 19-24. (in Chinese)
张申绒，张玉清，李发根，等. 适于 ad hoc 网络安全通信的新签密算法[J]. 通信学报，2010，31(3)：19-24.

[10] ZHOU L D, HASS Z J. Securing ad hoc networks[J]. IEEE Network, Special Issue on Network Security, 1999, 13(6): 24-30.

[11] LIU Z Y, MAO S L. A new secure group key management scheme for ad hoc networks[J]. Control & Automation, 2006, 22(12): 3-4. (in Chinese)
刘知远，毛胜利. 一个新的 ad hoc 安全组密钥管理方案[J]. 微计算机信息，2006，22(12)：3-4.

[12] ZHANG Q Y, MIAO F M, YUAN Z T, et al. Identity-base group key management scheme in ad-hoc[J]. Journal on Communication, 2009, 30(10A): 85-92. (in Chinese)
张秋余，苗丰满，袁占亭，等. 基于身份的 Ad Hoc 组密钥管理方案[J]. 通信学报，2009，30(10A)：85-92.

[13] ZHANG Y, DU R Y, CHEN J, et al. Analysis and improvement of an identity-based signcryption[J]. Journal on Communications, 2015, 36(11): 174-179. (in Chinese)
张宇，杜瑞颖，陈晶，等. 对一个基于身份签密方案的分析与改进[J]. 通信学报，2015，36(11)：174-179.

[14] BONEH D, FRANKIN M. Identity based encryption from Weil pairing [C] // Kilian JCRYPTO2001. Berin: SpringerVerlag, 2001: 213-229.

[15] MALONE-LEE J. Identity based signcryption[EB/OL]. http://eprint. iacr. org/2002/098.

[16] CHEN L, MALONE-LEE J. Improved identity-based signcryption[M]// Public Key Cryptography-PKC 2005. Springer Berlin Heidelberg, 2005: 362-379.

[17] LIBERT B, QUISQUATER J. A new identity based signcryption scheme from pairings [C] // IEEE Information Theory Workshop. 2003: 155-158.

[18] BONEH D, FRANKLIN M. Identity-based encryption from the Weil pairing[J]. SIAM Journal on Computing, 2003, 32(3): 586-615.

(上接第 167 页)

[5] AAFER Y, DU W, YIN H. DroidAPIMiner Mining API-Level Features for Robust Malware Detection in Android[M]// Security and Privacy in Communication Networks. Springer International Publishing, 2013: 86-103.

[6] ARP D M, SPREITZENBARTH M, HUBNER M. Drebin: Effective and explainable detection of android malware in your pocket [C]// Network and Distributed System Security Symposium, NDSS 2014. San Diego, USA.

[7] KWONGYAN L, YIN H. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis [C]// Proceedings of the 21st USENIX Conference on Security Symposium. 2012: 29.

[8] RASTOGI V, CHEN Y, ENCK W. AppsPlayground: Automatic Security Analysis of Smartphone Applications [C]// Conference on Data and Application Security and Privacy. ACM, 2013: 209-220.

[9] BLASING T, BATYUK L, SCHMIDT A. An Android Application Sandbox System for Suspicious software Detection [C]// 5th International Conference on Malicious and Unwanted Software. 2010.

[10] Android NDK [EB/OL]. https://developer. android. com/tools/sdk/ndk/index. html.

[11] 盘点 2015 年度 10 大安卓手机系统级病毒[EB/OL]. (2016-2-19). http://bobao. 360. cn/learning/detail/2750. html.

[12] Androguard [EB/OL]. https://github. com/androguard/androguard.

[13] ZHANG M, DUAN Y, YIN H, et al. Semantics-aware android malware classification using weighted contextual api dependency graphs[C]// Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2014: 1105-1116.

[14] AU K W Y, ZHOU Y F, HUANG Z, et al. Pscout: analyzing the android permission specification[C]// Proceedings of the 2012 ACM Conference on Computer and Communications Security. ACM, 2012: 217-228.

[15] Weka [EB/OL]. http://www. cs. waikato. ac. nz/ml/weka.

[16] Appchina [EB/OL]. http://www. appchina. com.

[17] Anzhi [EB/OL]. http://www. anzhi. com.

[18] Virus share [EB/OL]. http://www. virusshare. com.

[19] SIEFERS J, TAN G, MORRISETT G. Robusta: Taming the native beast of the JVM[C]// Proceedings of the 17th ACM Conference on Computer and Communications Security. ACM, 2010: 201-211.

[20] SUN M, TAN G. Nativeguard: Protecting android applications from third-party native libraries[C]// Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks. ACM, 2014: 165-176.

[21] VITOR A, ANOTONIO B, YANICK F, et al. Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy[C]// Symposium on Network and Distributed System Security(NDSS 2016). Diego CA, USA.