基于分层API调用的Android恶意代码静态描述方法

魏松杰 杨 铃
(南京理工大学计算机科学与工程学院 南京 210094)

摘 要 针对Android APK的静态描述,目前主要是采用权限、数据以及API调用序列的方法,而忽视了代码本身的层级结构,因此无法有效地通过这些静态特征来揭示应用程序的可能行为和恶意属性。设计并实现了一种基于代码层次结构的系统API调用描述方法,其主要是提取APK文件中API调用在应用包、对象类、类函数层面的信息,并将这些信息以树形结构表示,通过将不同应用程序的描述树进行逐层对比来计算相似度,揭示恶意应用程序由于在类型和族群上的差异所带来的API调用特征上的区别,从而为Android应用程序的特征描述和恶意检测提供新的视角。实验采用真实多样的已知Android恶意程序来验证描述方法的正确性和系统实现的效果,分析了不同层次和检测情况下该方法的利弊以及可能的改进之处。

关键词 Android,恶意代码,静态分析
中图法分类号 TP309.5 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2015.1.036

Android Malware Characterization Based on Static Analysis of Hierarchical API Usage

WEI Song-jie YANG Ling
(School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing 210094, China)

Abstract Current static-analysis practice on Android application package (APK) mainly uses the features such as permissions, data flows, API calls, extracted from the manifest file and the code. Such features lack consideration on the APK code organizations and object hierarchy, and thus they may be ineffective in describing and predicting an APK's application behaviors and maliciousness. This research work tried to design and implement a comprehensive API-usage characterization method for Android APK on different resolutions and hierarchies, namely packages, classes, and functions. A tree structure is used to contain such hierarchical API-usage information, and a comparison algorithm is designed for cross-tree similarity, which provides extra insights in classifying and differentiating Android malware of different types and code families. The variations in API-usage on different code layers imply code functionalities and application behaviors, and thus they can be used to improve current static-analysis based malware detection and signature generation. Realistic malware packet samples of various types and families were used to validate the proposed characterization method, and results were discussed for its strength and future improvement.

Keywords Android, Malware, Static analysis

1 引言

移动互联网的发展和移动智能设备的普及,带动了移动应用程序(APP)的火爆,同时也滋生了大量的移动恶意应用程序和应用攻击方法(Malware),损害了移动互联网的资源和效率,侵犯了移动用户的数据安全和隐私。根据艾瑞咨询《2013年中国移动安全数据报告》[1],当年新增Android恶意软件65万多个,应用市场上5.6%的APK具有恶意行为。由于Android系统从推出就实行源代码开放,攻击者容易在热门Android应用程序(APK)中插入恶意代码或恶意广告,也容易把已有的恶意代码植入不同的应用程序,二次打包后形成新的APK发布。因此数量庞大的Android恶意APK往往是在有限的恶意类型内(例如木马、后门、广告、骚扰等)形成大大小小的代码族群(Family)。如何通过代码静态特征来比对多APK之间的相似程度和关联水平,进而检测出经过了重新打包、代码重用、代码混淆的同类同族恶意APK,是急需解决的问题,也是目前通用的基于代码MD5的APK恶意检测技术和杀毒程序所无法完成的任务。

Android应用程序对于敏感数据、系统资源、设备状态的使用和修改,都是通过调用相应的系统API来实现的。不同代码范围和层次的API使用情况,反映了相应模块的代码功能,更能够大致推断出该模块在运行过程中的可能行为属性和表现形式。本文提出基于分层API调用的Android APK代码静态检测和描述方法,以程序的代码组织和调用关系作为描述索引,以APK包、对象类、对象函数3层结构作为描述视角,以API的调用统计和相似程度作为描述特征,最终实

到稿日期:2014-02-17 返修日期:2014-05-17 本文受国家自然科学基金(61472189),南京理工大学紫金之星项目资助。
魏松杰(1977—),男,博士,副教授,主要研究方向为计算机网络与信息安全、网络协议与应用,E-mail:swei@njust.edu.cn;杨 铃(1992—),女,硕士生,主要研究方向为移动互联网安全、恶意代码检测,E-mail:yangling_2014@njust.edu.cn。

现精确描述 APK 行为特征并达到判断两个 APK 恶意代码类型一致、代码同族的目标。

目前根据检测流程和特征描述方法不同，Android 恶意代码的检测方法大致可分为静态特征检测和动态行为检测两种。常用的 APK 静态特征包括权限声明、代码哈希、数据依赖关系、调用关系、程序组件等。对于已知恶意 APK，可采用 MD5 描述，而对于未知 APK，可根据权限和数据传播行为来检测隐私窃取和恶意扣费行为[2]。将恶意 APK 常用权限按照危险级别分别设定权重，通过 AndroidManifest. xml 中权限声明推断未知 APK 的危险程度[3]，也可以追踪比对权限声明和 API 调用关系在恶意 APK 中的分布规则[4,5]来总结规律进行未知检测。动态行为方面，一般采用网址访问[4]、短信或电话号码[4]、API 调用序列[5,9]、数据访问[4,7,9]等。虽然基于系统 API 的调用情况已经被不同的研究者所尝试和采用，但现存方法只是针对单个 API 的简单调用过程和调用序列，没有结合程序自身的组织结构和代码模块之间的隶属关系，缺乏从程序结构的层面、从程序功能实现的角度来分析系统 API 调用情况。

本文将 APK 的系统 API 使用情况与代码组织结构相结合，在包-类-函数 3 个层面上进行 API 调用情况的统计和 APK 相似度比较，通过真实多样的已知恶意 APK 类型和代码族群来对方法的有效性进行实验验证。研究的贡献和创新性概括如下：

1)定义了在 APK 包-对象类-函数 3 个层次上对于 API 调用情况的描述方法和比较算法，将层级的代码结构与 API 调用下的代码功能相结合，为 Android 恶意代码描述比较提供了新方法；

2)设计并实现了从 APK 到 API 的解码、分析、统计、存储、比较等一系列处理流程，并通过程序将这一流程有效地加以实施；

3)采用了现实多样的恶意 APK 样本进行实验，不但验证了所提方法的有效性和合理性，同时对于该方法在实际应用中的应用步骤和参数选择做出了合理的建议和推断。

2 APK 代码特征的提取流程和描述方法

对程序的静态分析主要是分析应用程序的核心代码，Android 将 APK 的核心代码存储在 DEX 文件中，所以我们的方法主要是静态分析 DEX 文件，检测待测 APK 中 DEX 文件的相似度，再根据相似度对 APK 进行分类和比较。整个分析和比较过程分为 5 个步骤：

1)预处理 APK 文件，使用 Android SDK 自带工具以及 ApkTool[10]工具对 APK 文件进行预处理。这个过程实现 APK 文件的解压，得到 DEX 文件、AndroidManifest. xml 文件、Resource 文件等；

2)读取 DEX 字节码，遍历后生成 5 个 txt 输出文档，分别包含该 APK 所有对象类、函数、系统 API 以及它们之间的各种隶属和调用关系。

3)分析 APK 代码结构和 API 调用情况，整理对象类、函数和 API 之间的隶属包含关系，重建程序代码的树形组织结构。输出结果为一个描述 APK 的树形结构。根节点为 APK 包名及包信息，其下第一层表示各个对象类的信息的节点，第二层表示各个类中包含的函数信息的节点，第三层是各个函数中调用的 API 信息节点。

4)对于任意两个 APK 的三层 API 描述结构，分层进行匹配查询和相似度计算，进而迭代出整个 APK 在 API 调用上的相似程度。

5)对 APK 相似度的计算结果进行统计和阈值设定，根据相似程度决定 APK 的恶意类型和代码族归属。

3 系统设计与实现

下面具体介绍分析的各个步骤的具体实现方法和所用到的技术、工具和算法。

3.1 APK 信息

APK 是运行在 Android 平台上的应用程序文件的统称，实际就是一个 ZIP 格式的压缩文件，使用 Google 提供的 aapt 或者第三方反编译工具 ApkTool 解压缩文件，可以得到 AndroidManifest. xml 文件，里面包含有 APK 安装时的配置信息，包括包名、版本号、组件声明以及权限需求等信息。如果对 APK 直接进行解压，还可以得到它的二进制字节码 DEX 文件——classes. dex，它是 Android 系统为了提高运行效率而生成的可以直接在 Dalvik 虚拟机上运行的文件格式，包含 APK 文件的核心代码。

3.2 DEX 文件信息提取

DEX 文件[11]是由多个结构体组合而成的，主要分为 7 个部分：DexHeader，DexStringId，DexTypeId，DexProtoId，DexFieldId，DexMethodId，DexClassDef，还有两个存储数据的 DexData 数据区和静态链接数据区的 DexLink。另外 File Header 中包含 APK 的校验和、文件长度、字符串表中字符串的数量以及其他 6 个结构体的绝对偏移量等；Class List 存储 DEX 文件中所有类的类名字符串索引；Method Table 包含所有方法的索引；Class Definition Table 定义了类的索引和超类索引。

为了分析 DEX 文件，我们用 C++语言编写一个 dexinfo 分析器。它可以直接读取二进制 classes. dex 文件，处理后生成 5 个 txt 文档。各文档的名称以及数据内容见表 1。

表 1 各文件内容

文件名称	数据内容
classes. txt	classID, superclass, method, API
classFuncs. txt	classID, funcID
funcs. txt	funcID, funcName, params, return, API
allcalls. txt	funcID, called API
apis. txt	apiID, apiName, parameter, return

以下针对每个输出文件的格式和内容给出示例及说明：

classes. txt

39 com. flurry. android. d　java. lang. Object 2 1 2

其中，“39”表示该类在 APK 中的编号 classID；“com. flurry. android. d”则表示这个类的类名，有可能是经过了混淆后的结果；“java. lang. Object”表示这个类继承的父类名；第一个“2”表示这个类调用的 API 总次数；“1”表示这个类调用的 API 的种类数(即我们所说的个数)；第二个“2”表示这个类中调用了 API 的函数(方法)个数；

classFuncs. txt

39 980 981

其中，“39”是 classes. txt 中对应的 classID；“980”和“981”是

classID=39 的类中函数的编号，即 funcID；

funcs. txt

980 Lcom/flurry/android/d;->⟨init⟩()V

981 Lcom/flurry/android/d;->uncaught Exception (Ljava/lang/Thread;Ljava/lang/Throwable;)V

其中，“980”和“981”是函数的编号，后边跟随的是该函数的签名，包括函数名、变量、返回值；

allCalls. txt

980 1264

981 1261

其中，“980”和“981”是函数编号，而“1264”和“1261”则是 API 调用编号；

apis. txt

1261 Ljava/lang/Thread $ UncaughtExceptionHandler; -> uncaughtException (Ljava/lang/Thread; Ljava/lang/Throwable;)V

1264 Ljava/lang/Thread; -> getDefaultUncaughtExceptionHandler ( ) Ljava/lang/Thread $ UncaughtExceptionHandler;)V

其中，“1261”和“1264”是 API 的编号，后面是对应 API 的具体信息。

3.3 生成 APK 三层描述的树形结构

得到 APK 代码的上述信息之后，要对有效信息进行筛选整合，生成一棵树形结构，这就是本文所采用的描述 Android APK 中代码结构和 API 调用分布的 3 层树形结构。如图 1 所示，除根节点外其每一层节点所包含的信息如下：

• 对象类：包含该类的父类、类成员、API 使用情况。

• 类函数：包含该函数的特征值(参数、返回类型)、归属类、API 使用列表和统计。

• API：包含每个系统 API 调用的调用位置和 API 参数配置。

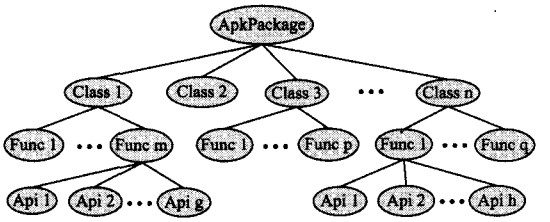

图 1 描述 APK 代码结构和 API 调用的树形结构

3.4 相似度计算

对从两个 APK 的 classes. dex 文件分析得到的两棵描述树进行匹配计算，得到相似度。对树进行逐层比对，即在各个层面上进行嵌套比较——package 层、class 层、function 层以及 API 层。除根节点外每一层的相似度都要递归计算它的子节点的相似程度。算法描述如下：

```
输入 apk1. apk 和 apk2. apk，计算两个 apk 对应树的每一层的相似度；
Similarity{
  float NameSimilarity(ApkPackage apk1,Apkpackage apk2){
    返回两个 String 类型包名的相似度
  }
  float ClassSimilarity(ApkPackage apk1,Apkpackage apk2){
    for (两个 apk 的每个类节点) {
      如果两个类节点属性中父类相同，且 apiCount 值相差小于阈值；
      则计算这两个节点每个 function 子节点的相似度并记录下最大的相似度；
    }
    返回所有 class 相似度不为 0 的最大相似度的平均值
  }
  float FuncSimilarity(ApkPackage apk1,ApkPackage apk2){
    for(两个 apk 的每个方法节点){
      如果两个方法节点的属性中返回值和参数相同且 apiCount 值相差小于阈值；
      则计算这两个节点子节点的相似度并记录下最大的相似度
    }
    返回所有 function 相似度不为 0 的最大相似度的平均值
  }
  float ApiSimilarity(ApkPackage apk1,ApkPackage apk2){
    返回 API 字符串匹配的比值
  }
}
```

1)根节点：根节点中存放有应用程序包的包名，可用来推测相同或者同族的 APK。对包名字符串以“.”为分隔符，

$m=split(PackageName1,".")$

$n=split(PackageName2,".")$

将分隔出的字符串作为一个整体进行比较：

$$NameSimilarity=\frac{|m\cap n|}{(|m|+|n|)/2}$$

2)class 节点相似度：取两个待测 APK 的 class 集合 $C_1=\{c_i\,|\,APK_1$ 中的类节点$\}$，$C_2=\{c_j\,|\,APK_2$ 中的类节点$\}$，定义 $C_{1,2}=\{c_i,c_j\,|\,c_i\in C_1,c_j\in C_2,c_i$ 和 c_j 继承的父类相同，且调用 API 的个数相差小于 10%$\}$，$CSim_{1,2}=\{\max(csim_{i,j})\,|\,(c_i,c_j)\in C_{1,2}\}$，则有：

$$ClassSimilarity=\sum_{cs\in CSim_{1,2}} cs/|CSim_{1,2}|$$

3)function 节点相似度：取两个待测 APK 的 function 集合 $F_1=\{f_i\,|\,f_i\in APK_1\}$，$F_2=\{f_j\,|\,f_j\in APK_2\}$，定义 $F_{1,2}=\{f_i,f_j\,|\,f_i\in F_1,f_j\in F_2,f_1$ 和 f_2 的返回类型 $returnType$ 相同，包含参数个数及类型相同，且 API 调用个数相差小于 10%$\}$，$FS_{1,2}=\{\max(fsim_{i,j})\,|\,(f_i,f_j)\in F_{1,2}\}$，则有：

$$FuncSimilarity=\sum_{f\in FS_{1,2}} f/|FS_{1,2}|$$

4)API 节点相似度：取两个待测 apk 中调用的系统 API 的集合，$S_1=\{s_i\,|\,APK_1$ 中调用的 API 节点的字符串$\}$，$S_2=\{s_j\,|\,APK_2$ 中调用的 API 节点的字符串$\}$，则有：

$$ApiSimilarity=\frac{S_1\cap S_2}{(|S_1|+|S_2|)/2}$$

4 实验结果及分析

为了检测本文所提出的 APK 恶意代码的描述方法与流程的检测效果与处理性能，首先选取了 Android 平台上的具有典型性的 4 个恶意代码类型的 100 个恶意代码样本，共计 100 个完整的 APK 文件，具体分布情况如表 2 所列。样本来自 Android 平台安全软件反病毒测试的国际权威机构 AV-Test，其恶意属性已经过多家安全软件测评并确认。

表 2　APK 恶意代码列表

恶意类型(Type)	代码族(Family)	样本个数
Backdoor	BaseBrid	30
Backdoor	Kmin	3
Backdoor	KungFu	27
PUA	Aveasms	3
PUA	Linux. Lotoor	6
PUA	Other	4
Trojan	FakeDoc	3
Trojan-SMS	Boxer	3
Trojan-SMS	FakeInst	17
Trojan-SMS	Jifake	1
Trojan-SMS	Opfake	3
Total		100

首先利用系统的 ApkTool 工具对每个 APK 文件进行解压缩并提取出 AndroidManifest 文件和 DEX 字节码文件，然后利用 dexinfo 程序针对每个 APK 查询程序结构和 API 调用信息，最后为每个 APK 生成 class-function-api 三层结构树形描述。以上所有步骤采用脚本语言在 CentOS Linux 平台上自动实现。接下来对于样本集合中的任意两个 APK，在三层结构上进行相似度的计算和存储。为了加速计算过程，采用分割并行的方法对 APK 样本集合进行动态划分，计算平台为一台 Xeron E5 CPU 和 8GB 内存的计算机。

任意两个恶意 APK 可能是同一类型的同一代码族(同类同族)，同一类型不同的代码族(同类异族)，或者是完全不同类型的两个恶意程序(异类异族)。在过滤掉一些明显的重复 APK 包(即只有配置文件和资源文件不同，字节码完全相同)后，3 种组合方法产生的 API 比较大致分别为 950、1150 和 2800 个。根据系统的理论依据和设计初衷，同类同族的两个 APK 由于具有相同的恶意属性和相近的恶意行为实现代码，其相似程度在 3 个描述层面上都应该具有较高的水平。同类异族的两个 APK 具有不同的代码实现，因此在 Class-Similarity 和 FuncSimilarity 层面上相差较大，但相同或者相近的恶意行为的实现需要同样的系统支持及系统 API 调用，因此仍期望得到较高的 ApiSimilarity 层面相似度。异类异族的 API 可能具有完全不同的程序恶意功能和实现，因此它们之间的相似程度会在所有描述层面上表现出较大差异。

图 2 展示了不同情况下 APK 之间在对象类(class)描述层面上的相似度的分布规律。显而易见，同类同族 APK 具有明显的相似性，超过 93%的同类同族 APK 的相似程度在 0.6 以上，75%达到 0.8 甚至更高。与此相对照，具有不同代码结构和实现方法的异族 APK 无论是同类的还是异类的，其类描述相似程度都相对较低，超过 0.8 的寥寥无几。图 3 中的类函数(function)描述层面上的比较数据展现了类似的结果。这充分说明了在检测 Android 恶意应用程序的衍生版本(repackaging 或者 refactoring)时，考虑 API 调用在类和函数范围内的相似程度，有利于提高检测的准确度。

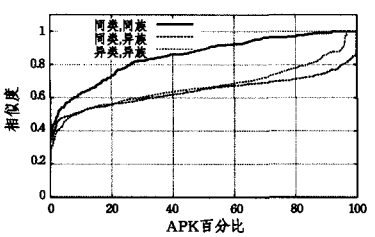

图 2　在对象类的层面描述和比较 APK 恶意代码

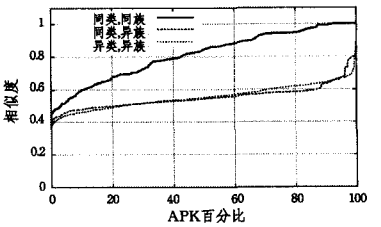

图 3　在类函数的层面描述和比较 APK 恶意代码

当采用基于整个 APK 范围的 API 调用情况来描述每一个 APK 时，APK 之间的相似水平则比较低，图 4 中超过一半以上的同类同族 APK 的相似度不足 0.6。这说明 APK 中调用 API 的情况与 APK 本身的功能需求和实现方法有关，与其恶意关联性联系并不紧密。而且从整个 APK 的范围来考虑，典型的 APK 很容易对上千个 API 调用数万次甚至更多，API 使用的组合极其复杂，无法实现程序代码和功能的高分辨率，很难有效区分正常和恶意代码，或者不同类型、族群的恶意代码。本文在 APK 全局的基础上，增加了类和函数层面上的 API 调用描述，正是这种将调用情况比较的范围缩小的方法，使比较的机会增加，从而放大了同类同族 APK 比较结果与其它情况的差异性。

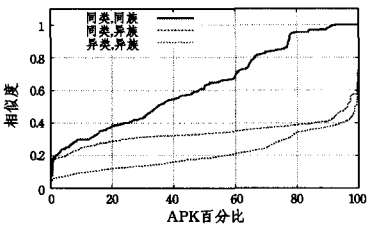

图 4　通过系统 API 调用来描述和比较 APK 恶意代码

图 5 给出了通过比较 APK 包名来度量 APK 的相似程度的方法，从而检测出代码重用、重打包、深度混淆等情况。很明显，单纯依赖包名是无法区分恶意 APK 程序的类型和代码族群的。

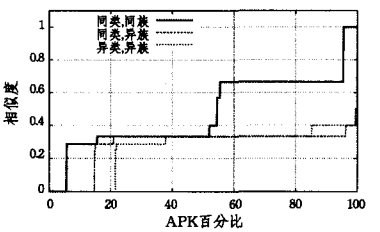

图 5　通过 APK 包名比较 APK 恶意代码

上述实验数据都是通过比较已知类型和代码族的恶意 APK 的代码特征在三层描述上的相似度而得到的。在恶意 APK 检测的实际应用中，往往是比较一个未知的 APK 与已知恶意代码库之间的匹配度，而这个未知 APK 可能是恶意的、风险的，也可能是正常可信的应用程序。为了检验本文中的代码静态 API 调用特征的三层描述结构应用到正常可信的 APK 上的效果，我们在 Android 应用市场下载了 10 个下载量领先的知名 IT 公司的应用程序 APK，分别跟已知的 100 个恶意 APK 进行比较。图 6 中的比较结果显示，它们之间相似程度符合预期：即与图 2－图 4 中的结果迥异，无论是在对象类、函数还是 API 层面，都具有较大差异性或相对较低的相似度。

(下转第 179 页)

这也是软件危机产生的原因。我们将解决危机的重点放在了人身上，通过群体智慧，实现软件开发社会化和全球化的生产。我们相信利用间层开发模型和柠檬框架将是一个行之有效的方案。

未来，我们将进一步实现间层开发模型的其他框架并完善柠檬框架，最终落实到产品上，以验证间层模型确实是解决软件危机行之有效的方案。

参 考 文 献

[1] Naur P, Randell B. Software Engineering: Report of a conference sponsored by the NATO Science Committee[R]. Germany(Garmisch): Brussels, Scientific Affairs Division, NATO, 1969

[2] Brooks F P. The mythical man-month [M]. Reading: Addison-Wesley, 1975

[3] Brown T. Modernisation or failure? IT development projects in the UK public sector[J]. Financial Accountability & Management, 2001, 17(4): 363-381

[4] Beizer B. Software Is Different[OL]. 2001[2014]. www. soft. com/News/QTN-Online/qtnapr01. html

[5] 钟志永，姚珺. 大学计算机应用基础[M]. 重庆：重庆大学出版社，2012：230-231

[6] Fitzgerald B. Software Crisis 2. 0[J]. Computer, 2012, 45(4): 89-91

[7] Tracz W. Confessions of a used program salesman: institutionalizing software reuse[M]. Addison-Wesley Longman Publishing Co., Inc., 1995

[8] Cox B. " No Silver Bullet" Reconsidered[J]. American Programmer, 1995, 8: 2

[9] What is com? [OL]. http://www. microsoft. com/com/

[10] Enterprise JaveBeans Technology [OL]. http://www. oracle. com/technetwork/java/index-jsp-140203. html

[11] CORBA[OL]. http://www. corba. org/

[12] Kang K C, Cohen S G, Hess J A, et al. Feature-oriented domain analysis (FODA) feasibility study[R]. Carnegie-mellon Univ Pittsburgh Pa Software Engineering Inst, 1990

[13] Zhu L. Model-driven architecture[M]// Mellor S J, Scott K, Uhl A, et al. Advances in Object-Oriented Information Systems. Springer Berlin Heidelberg, 2002: 290-297

[14] Mangalaraj G, Mahapatra R K, Nerur S. Acceptance of software process innovations-the case of extreme programming[J]. European Journal of Information Systems, 2009, 18(4): 344-354

(上接第 158 页)

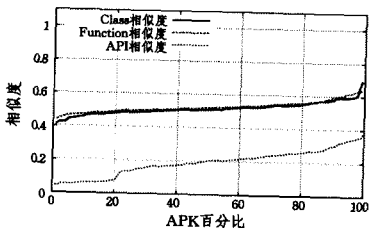

图 6 可信 APK 与恶意 APK 之间的代码比较结果

上面的实验结果分别展示了在 APK 的三层结构中 API 调用情况的比较效果，作为有效的 APK 描述和相似度度量方法，在实际应用中，完全可以将这 3 种比较结合起来，互相弥补在不同情况下的缺陷。例如，将 3 种相似度取平均值或者通过计算 3 种相似度的发散程度，来估计 APK 相似的置信程度。另外，在上面的实验中，我们并未给出具体的可以用来检测 APK 的代码，包括经过了重打包、混淆等过程的恶意代码的有效阈值，而只是展示了在设置合理阈值的前提下三层检测方法的有效性。我们相信有效阈值应该针对不同的恶意类型和检测要求，通过增加预处理和训练的样本数量来动态选择。

结束语 本文提出一种基于静态代码结构的 Android APK 的 API 调用三层描述和相似度比较方法，用以进行恶意 APK 类型和代码族的检测工作，也可用于未知 APK 的风险评估和分类。通过程序实现了整个 APK 处理过程和相似度计算方法。实验数据采用真实多样的已知恶意代码类型和族群，进行了跨类跨族的分层比较方法测试和有效性分析。本文针对以往的 APK 检测特征无法准确描述 API 调用关系和应用范围的缺陷，在 package-class-function 3 个层次上进行了 API 调用情况的统计和对比，增加了代码比较的精确度，放大了恶意代码在实现其恶意行为时无法规避的特定 API 调用需求和分布规律，为 Android 恶意代码特征提取和检测提供了新的视角。在后续研究中，我们将尝试结合目前 3 个层次的相似度，计算出统一的 APK 层面的相似度以及基于恶意相似度的未知 APK 风险程度评估；计划获取并测试更多的恶意代码样本，充分测试文中方法的准确性、适用性和稳定性，尝试针对不同的恶意代码类给出建议的检测阈值。

参 考 文 献

[1] 艾瑞咨询. 2013 年中国移动安全数据报告[EB/OL]. http://report. iresearch. cn/2103. html

[2] 秦中元，徐毓青，梁彪，等. 一种 Android 平台恶意软件静态检测方法[J]. 东南大学学报：自然科学版，2013，43(6)：1162-1167

[3] Canfora G, Mercaldo F, Corrado Aaron Visaggio. A classifier of Malicious Android Applications[C]// Proceedings of 2013 International Conference on Availability, Reliability and Security (ARES 2013). 2013: 607-614

[4] 胡文君，赵双，陶敬，等. 一种针对 Android 平台恶意代码的检测方法及系统实现[J]. 西安交通大学学报，2013，47(10)：37-43

[5] 李寅，范明钰，王光卫，等. 基于反编译的 Android 平台恶意代码静态分析[J]. 计算机系统应用，2012，21(11)：187-189

[6] 杨欢，张玉清，胡予濮，等. 基于多类特征的 Android 应用恶意行为检测系统[J]. 计算机学报，2014，37(1)：15-27

[7] Yang Zhe-min, Yang Min, Zhang Yuan, et al. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection[C]// Proceedings of the 20th ACM Conference on Computer and Communications Security. 2013

[8] Shabtai A, Kanonov U, Elovici Y, et al. Andromaly: A Behavioral Malware Detection Framework for Android Devices[J]. Journal of Intelligent Information Systems, 2012, 38: 161-190

[9] Isohara T, Takemori K, Kubota A. Kernel-based Behavior Analysis for Android Malware Detection[C]// Proceedings of International Conference on Computational Intelligence and Security (CIS). 2011: 1011-1015

[10] Android ApkTool-A Tool for Reverse Engineering Android APK[EB/OL]. http://code. google. com/p/android-apktool

[11] Dalvik Executable Format [EB/OL]. http://source. android. com/devices/tech/dalvik