电子科技大学

University of Electronic Science and Technology of China

学 士 学 位 论 文

BACHELOR DISSERTATION

论文题目Android平台APK特征分析

学生姓名 党标

学　　号 2010063040035

专　　业 信息安全

学　　院 计算机科学与工程学院

指导教师 王勇

指导单位 电子科技大学

2014年5月1日

摘 要

Android 操作系统是当下市场占有率最高的智能手机操作系统。以.apk为扩展名的 Android Packet(APK)文件即是 Android 平台应用程序的安装包。近几年，第三方电子市场层出不穷，研究人员对 Android 应用的兴趣也越来越浓厚。伴随着Android 应用数量的激增，对 Android 应用进行自动化归类的需求应运而生。

本文利用反编译技术，对 Android 平台的应用程序安装包，即 APK 文件，进行反编译并从中提取静态特征。进而利用机器学习技术对提取的静态特征进行学习建模，最后达到了对 APK 进行自动化分类的目的。所提取的静态特征主要来自于 Android 应用的 Java 字节码文件和 XML 文件。除此之外，本文还对如何计算两个 APK 文件之间的相似性做了研究。通过分析从 APK 文件中反编译出的 smali 代码，我们从中提取出 APK 文件调用的 API 及函数的执行逻辑。结合从 APK 中提取的这些特征，我们分别研究了余弦相似度算法和NCD算法在 APK 相似性计算上的应用，并通过实验对算法的性能做了验证和对比。

关键词：机器学习，安卓，静态分析，分类，相似性

ABSTRACT

Android is kind of operating system developed for smartphone, which has the highest market share. Each application for Android is packaged in an .apk archive, which is similar to standard Java .jar files and comprise of both code and resources. Android Package (APK) files encapsulate valuable information that can help in understanding an application’s behavior. In recent years, the third party electronic market emerge in endlessly, the researchers become more and more interested in APK files. With the sharp increase in the number of Android application, the demand for classification of Android applications arises.

In this thesis we apply Machine Learning (ML) techniques on static features that are extracted from APK files for the classification of Android applications. Features are extracted from Android’s Java byte-code and other file types such as XML-files. Finally we achieved the purpose of classifying Android applications using Machine Learning technique. In addition, we have done some research on how to calculate the similarity between two APK files. By analyzing the smali code extracted from the APK file, we can get the APIs called by the APK and the method behavior. Combining these features extracted from APK, we respectively applied the cosine similarity algorithm and NCD algorithm on APK similarity calculation, and performed a series of experiments to evaluate the performance.

Keywords: Machine Learning, Android, Static analysis, Classifying, Similarity

目 录

第1章 引言 1
1.1 课题研究背景 1
1.2 相关课题国内外研究现状 1
1.3 课题研究内容、价值和意义 2
1.4 论文的组织结构 3
第2章 APK 结构分析及自动化归类 4
2.1 APK 结构分析 4
2.2 自动化分类的应用 5
2.3 机器学习与分类 6
2.4 用 KNN 算法对 APK 分类 7
2.5 本章小结 8
第3章 用机器学习的技术实现APK的自动化分类 10
3.1 种类的确定 10
3.2 分类特征选取 11
3.2.1 特征表示模型和选择算法 11
3.2.2 特征提取方式 12
3.3 模型的训练与测试 13
3.4 性能评估与分析 15
3.4.1 评估的目的 15
3.4.2 评估方法选择 15
3.4.3 数据结果与分析 16
3.4.4 分类效果展示 18
3.5 本章小结 19
第4章 APK 相似性算法研究 20
4.1 余弦相似度算法计算APK间相似性 20
4.1.1 余弦相似度计算 20
4.1.2 APK 文件的宏观相似 21
4.1.3 算法的优缺点分析 24

4.2 NCD 算法计算 APK 间相似性 25
4.2.1 APK 文件的微观相似 25
4.2.2 函数的序列化 25
4.2.3 压缩算法的选择 32
4.2.4 处理流程及结果计算 34
4.2.5 算法优缺点分析 36
4.3 算法性能对比 36
4.4 本章小结 38
第5章 结束语 39
参考文献 41
致 谢 43
外文资料原文 44
外文资料译文 53

第1章 引言

1.1 课题研究背景

Android 系统是由 Google 公司研发的适用于移动设备的操作系统。随着人们对移动设备的需求量不断增长，Android 系统也越来越受欢迎。短短时间，Android 系统的市场占有率便跃升到了第一位。早在2013年9月，Google 高级副总裁向外界透露 Android 设备的激活总量已经超过10亿台，而且还在以日激活量超百万的速度呈爆发式增长。面对规模如此巨大的用户市场，Android 应用开发者蜂拥而至。据估计，Google play 中的 Android 应用数量即将超过苹果 App store 的85万而成为第一大应用市场。在用户活跃时期，Google play 每月的应用下载量更是多达25亿次。Android 系统具有开放性，允许安装第三方应用市场。于是，近几年来第三方市场层出不穷。另一方面，由于 Android 系统开放源代码，这一点深受研究人员的青睐。因此，越来越多的研究人员选择 Android 系统的相关课题。

1.2 相关课题国内外研究现状

对 Android 系统进行相关研究的门槛相对较低。不管是研究 Android 系统的框架还是研究 Android 应用程序，我们都可以再 Google 的官方网站中找到相关的文档。研究者甚至还可以下载 Android 系统的源代码，参与到 Android 系统的开发中。就像有很多人研究 PC 平台的 PE 一样，也有很多研究人员或者工程师在研究 Android 平台的应用程序。Android 平台的应用程序是以归档文件的形式存在，即 Android Package(APK) 文件。不同研究人员对 APK 文件的研究的方向各有不同，研究主要中在对 APK 文件的加密、混淆、反编译和恶意软件的检测等领域。尤其是随着移动社交和移动支付的发展，越来越多的人开始关心个人隐私和数字支付的安全问题。恶意 APK 的检测与查杀的相关研究越来越受欢迎。自从2010年8 月，卡巴斯基病毒实验室发现 Android 系统首个木马程序。此后，Android 恶意代码的检测的研究迅速出现，相继有大量论文被发表。不管是研究人员做 APK

相关的研究，还是第三方电子市场对 Android 应用的管理，都难免会遇到对 APK 进行分类或者分析 APK 的家族属性的需求。

在 APK 的分类问题上，早在2010年，Asaf[1] 等人便进行了 APK 分类的相关研究。该文中，作者同样是利用反编译技术从 APK 文件中提取大量特征，并利用机器学习的算法试图对 APK 文件进行分类。遗憾的是，该文的目的并不是对 APK 进行分类。作者实际目的是想通过这种方法实现恶意软件的检测。作者试图证明文中提出的算法可以将 APK 分成良性和恶意软件，但苦于当时没有足够的恶意样本，于是想到了将 APK 分成工具和游戏两类。虽然文中实验结果拥有较高的准确率，但这却无法保证在将 APK 文件分为多个类的情况下，其依然准确。在2012年的时候，Borja[2] 等人发表了一篇 APK 分类的相关研究。文中提出了 PUMA 的概念，即根据 Android 应用的配置文件中所请求的权限分类。AndroidManifest.xml 文件是 Android 应用的配置文件。通过解码该文件，从中提取 uses-permission 标签中的内容便可以获得 Android 应用所请求的权限。权限控制是 Android 系统的一种安全机制，一个应用要想进行敏感操作或者访问硬件资源就必须要申请相应的权限，否则会出错。关于机器学习算法，文中选择了朴素贝叶斯等算法，并没有选择 KNN 算法。最后的性能对比也没有足够的说服力。

而在 APK 的家族属性的分析上，Google code 上有一个开源项目 Androguard。该项目是一系列的工具集,主要功能集中在对 APK 文件的静态分析上。除此之外，该项目集成了一个恶意代码分析工具。该工具可根据从APK文件中提取的特征码大致分析出 APK 所属的恶意代码家族，缺点是识别率不高。支持的种类不足。

1.3 课题研究内容、价值和意义

如上文所说，在做 APK 相关的研究时，难免都会遇到需要将收集到的 APK 文件分类的问题。当 APK 文件数量不多时，我们尚且可以手工分类，但是若 APK 数量巨大我们便会迫切需要能够批量自动化分析并归类 APK 的方法。而本课题，首先要解决的即是这个问题。国外研究者利用机器学习的技术将 APK 文件分为了工具和游戏两类，但其分类范围太过宽泛，无法满足很多情况下的研究需求。本课题中我们将考虑将 APK 的分类细化，并使用 KNN 方法进行分类。

除了需要对 APK 进行分类之外，研究者们往往也对某些 APK 之间的家族关系非常感兴趣。一方面，如果知道了某恶意代码属于特定的恶意代码家族，研究者们就可以使用和分析该家族其它成员相似的方法对其进行分析，这将大大提高

工作效率。另一方面，Android 应用存在严重的抄袭问题。通过分析 APK 之间的家族属性，有利于帮助我们判断两个 APK 之间是否存在抄袭问题。从而有效的保护 Android 开发者的利益。两个 APK 之间的相似度越高，就意味着两个 APK 之间的家族属性越紧密。为了分析 APK 的家族属性，这里我们首先需要解决的就是如何计算出两个 APK 文件之间的相似度的问题。在这个问题上，现有的开源软件虽然有类似的功能，但准确度却无法令人满意。有时会将两个完全没有家族关系的文件判断为高度相似，在本课题中，将着重解决准确度问题。

1.4 论文的组织结构

本文的组织方式分为五章，各章的内容安排如下：

第一章，指明了论文的研究背景、课题内容、价值和意义、国内外研究现状以及论文的结构安排。

第二章，分析了 APK 文件的结构，介绍了机器学习技术在自动化分类中的应用，并描述了如何将 KNN 算法应用到 APK 的分类问题中。

第三章，阐述了如何用机器学习技术对 APK 进行分类，并对分类效果做了评估与分析。

第四章，提出了如何将余弦相似度算法和 NCD 算法应用到 APK 相似性的计算问题上。并详细的描述了计算过程和细节。

第五章，作为结束语总结本课题所涉及的相关理论、实践开发中遇到的问题以及解决方案和课题的最终成果以及需要进一步改进之处。

第2章 APK 结构分析及自动化归类

2.1 APK 结构分析

Android 是基于 Linux 内核，主要用于移动设备的操作系统，在架构上可分为5部分，从下至上分别是 Linux Kernel、Android Runtime、Libraries、Application Framework 以及 Application。Android 应用程序以 APK 文件的形式发布，APK 文件实际上是一个压缩包，其结构[3]如图2–1所示。其中 META-INF 文件夹存放应用程序的签名信息，用来保证 APK 包的完整性；res 文件夹存储资源文件，包括图片、字符串、UI 布局文件等；AndroidManifest.xml 是应用程序的配置文件，其中声明了应用程序的包名、SDK 版本、权限、组件等信息；classes.dex 则是 Java 字节码文件，可运行于 Android 虚拟机 Dalvik 上。本文所提取的静态特征主要来自于 Java 字节码文件和 XML 文件。

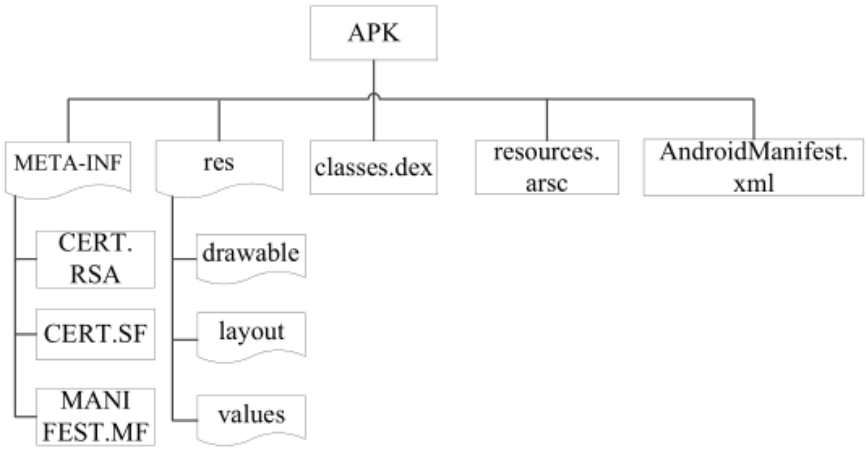

图 2–1 APK 文件的结构

Android 系统具有严格的权限管理机制，应用程序使用特定功能时，须在 AndroidManifest.xml 文件中使用 uses-permission 标签声明相应的权限，例如访问网络时，需要声明“android.permission.INTERNET”，否则在程序运行时会触发安全异常。

2.2 自动化分类的应用

分类问题是我们日常生活中较为常见的问题。当物品或信息量多到一定程度时，分类可以帮助人们有效的管理和认识这些物品或信息。因此我们会看到购物网站里的商品会被分类；新闻网站的新闻会被分类；视频网站里的视频内容也会被分类。当然，Android 应用市场中的应用程序也会被分类。对于应用市场来说，将 Android 应用进行分类是非常有必要的。有效合理的分类可以更快的帮助用户找到自己需要的应用程序，提升用户体验。对研究 APK 的研究者来说，分类更是能提要研究效率。例如，如果知道了某恶意代码属于特定的恶意代码家族，研究者们就可以使用和分析该家族其它成员相似的方法对其进行分析，这将大大提高工作效率。

对 APK 进行分类是一个实实在在的需求，那么当下各大应用市场和 APK 研究者是如何实现对 APK 的分类的呢？对于这个问题，笔者特地在知乎提了一个问题，并邀请了当下国内市场占有率较大的豌豆荚相关技术人员来解答。豌豆荚创始团队成员丁吉昌做了如下解答：

“一般情况下，Android 应用市场的应用分类第一步基本上都是由开发者自己勾选的。这里就涉及到一个问题，最初的分类是怎么来的？这里面首先会根据现有的市场（无论是购物网站还是其他的一些统计工具，都能得到很多分类信息），这些构成最初的来源，在经过产品人员或审核人员的 review 和调整后（比如图像、工具、系统、安全等是不是要合并？怎么合并），会再根据这些分类的下载量或者应用数量再做一些调整，尽量使得热门的分类不被隐藏，相关的分类可以合并到一起。对于搜索引擎来说，全部靠人工或者开发者标注的方法不太靠谱，还有一个方法，是运用应用的描述和 title 等，用机器学习的方法抽取标签，再根据这些标签的分布决定是否建立对应的分类，再将应用划入这些分类。同时，来自不同应用商店的分类方法也不一致，这个也需要通过机器的方法对不同商店的分类进行再归类，然后进行合并。”

我们得知，当下应用市场的主要的分类方式为开发者自己标注，同时用机器学习的方法做辅助。但是这里所提到的机器学习却局限于应用的描述和 title 。那么我们能不能利用反编译等技术手段对 APK 进行更深层次的解析并从中提取出内在的特征，然后用机器学习的技术对其学习建模，最后达到自动分类的目的呢？答案当然是肯定的。这也正是本文所研究的内容。

2.3 机器学习与分类

事实上关于机器学习，并没有一个统一的定义，这里列举两个比较著名的定义。第一个是1959年，Arthur Samuel[4] 认为，机器学习是在不需要被显示的程序化的情形下，给予计算机学习的能力（Field of study that gives computers the ability to learn without being explicitly programmed.）。另外一个是1998年，Tom Mitchell[5] 认为，机器学习是一个适定的学习问题：计算机程序可以就一些任务 T 和性能测量 P 来从经验 E 进行学习，如果其在 T 上，由 P 测量的性能能够提高经验 E ，我们认为这就是机器学习（A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T as measured by P improves with experience E.）。对于机器学习，一般而言，我们将其分为四类：监督学习、学习理论、非监督学习以及增强学习。

监督学习之所以称为监督学习是因为它需要被某个算法监督。换句话说，给定某个算法，我们需要这个算法能够学习输入与输出之间的某种关联，从而对于给定的数据能够得到更多的正确的分类结果。

对于监督学习问题，因为对于每一个已给出的输入数据都给出了其正确的结果，所以监督学习的目标是使得算法能够对其他新加入的数据产生更多的正确的结果。

学习理论可以让我们了解学习算法是如何并且为什么要这么做，从而我们我可以将这些学习算法运用的更加高效。探究学习理论可以让我们明白哪个算法更好的接近不同的函数和需要多少训练数据。学习理论的核心是，应该使用什么工具，以及如何利用这些工具来更好的完成机器学习。

非监督学习的问题是对所获得的数据进行分类整理，粗略的说就是给定一个数据集，但是不知道任何一个数据的正确分类结果，你所要做的是如何找到数据集中的一些感兴趣的结构，而这恰好是非监督学习要讨论的问题，即在未给定任何正确结论的前提下对数据集进行分类整理。对于非监督学习，训练集并未给出任何正确的结论，给出的仅仅是一部分训练数据而已。

增强学习涉及到不止做一个简单的决策之类的问题。在增强学习中，通常需要在一段时间里面做出一系列的决策。增强学习算法的一个基本思想是报酬函数(reward function)，即首先做出具体的指定，指定什么是“好”行为，什么是“坏”行为；然后给这个学习算法算出如何使“好”行为的报酬信号最大和最小化“坏”行为的惩罚。

在权衡了各种学习方式的利弊和适用情况之后，本课题中，我们将采用监督学习的方式。事实上，机器学习在不知不觉中已经被广泛应用到了各种领域，例如 Google news 就是利用机器学习的相关技术对从海量信息中检索出的新闻进行归类。在本课题中，我们将研究的 APK 自动化分类和文本分类并没有本质的区别。两者都需要提取特征向量，并利用分类算法对特征向量进行匹配。当然，要做到完全匹配不太可能。因此，分类算法会给出最为接近的结果。而机器学习中的分类算法多种多样，比较著名的有：决策树、贝叶斯、人工神经网络、KNN、SVM 和基于关联规则的分类等。本课题中，我们将采用 KNN 算法进行实验。

2.4 用 KNN 算法对 APK 分类

KNN 算法[6]是机器学习里面比较简单的一个分类算法，整体思想比较简单：计算一个点 A 与其它所有节点之间的距离，取出与该点最近的 K 个节点，然后统计这 K 个节点里面所属分类比例最大的，则点 A 属于该分类。具体步骤如下：

1. 用特征向量描述训练集中的样本；
2. 在样本到达后根据特征表确定样本的向量表示；
3. 在训练样本集中选出与新样本最近的 K 个样本；
4. 在新样本的 K 个邻居中依次计算每类别权重；
5. 将新样本指派给权重最大的类别。

举例说明，假如我们用与图形有关的 API 数量和与系统有关的 API 数量来界定软件类型，与图形有关 API 数量多的是游戏类型的，而与系统有关的 API 多的是工具类型的。现在已知数据如表2–1所示，还有一个名字未知（这里名字未知是为了防止能从名字中猜出软件类型），该软件中图形有关的 API 数量为18，与系统有关的 API 数量为90的软件，它到底属于哪种类型的软件呢？下面我们用 KNN 算法来解决这个问题。

KNN 算法要做的，就是先用与图形有关的 API 数量和与系统有关的 API 数量为的坐标，然后计算其他六款软件与未知软件之间的距离，取得前 K 个距离最近的软件，然后统计这 K 个距离最近的软件里，属于哪种类型的软件最多，比如游戏类最多，则说明未知的软件属于游戏类型。

在 KNN 算法的使用中，我们需要解决三个问题。第一，用哪种方式计算两者间距离。第二，训练样本要均匀。假设样本中，类型分布非常不均，比如游戏类软件有200种，但是工具类软件只有20种，这样计算起来，即使不是游戏类的软

表 2–1 软件类型与所调用的API关系表

软件名称	系统类API数量	图形类API数量	软件类型
时空猎人	3	104	游戏
天天飞车	2	100	游戏
植物大战僵尸	1	81	游戏
猎豹清理大师	101	10	工具
省电宝	99	5	工具
扫描全能王	98	2	工具
未知	18	90	未知

件，也会因为游戏类的样本太多，导致 K 个最近邻居里有不少游戏类的软件。第三， K 值的选取。我们规定需要检验的最近相邻元素（最近邻居）的数量，此值叫做k。图2–2显示了不同的k值对新样本进行分类的影响。 当$k=5$时，新样本将

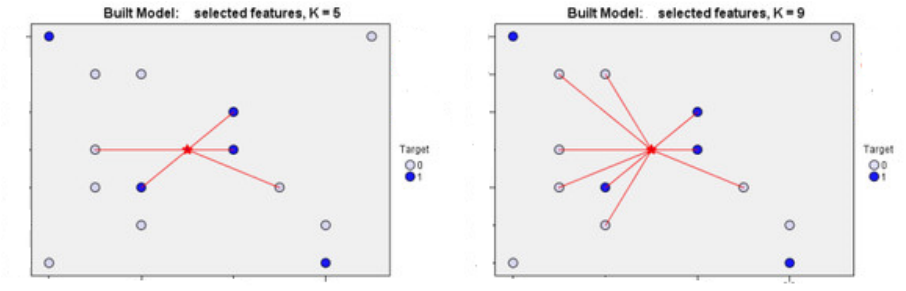

图 2–2 更改k 对分类的影响

被置于游戏类别中，因为大多数最近相邻元素属于游戏类别。但当$k=9$ 时，新样本将被置于工具类别中，因为大多数最近相邻元素属于工具类别。

具体到本课题所研究的 APK 分类问题。在参考了诸多文献和 KNN 的各种用例之后，我们决定选择使用欧几里得距离。关于样本均匀问题，属于可控因素。在构造训练集时，我们尽量使各个种类的样本数量均匀，避免此因素带来较大的误差。而 K 值的选取上，由于不同的问题最优 K 值差别较大。甚至于相同的问题，不同应用场景最优 K 值都有明显不同。因此，本课题中，我们将分别选取不同的 K 值做实验，并对比实验结果，选取出最优 K 值。

2.5 本章小结

本章简单介绍了 APK 文件的结构及各个部分的作用。然后用一些常见的例子引出了 APK 的分类问题。接着介绍了机器学习在分类问题上的应用。最重要

的，本章我们详细阐述了如何使用 KNN 算法对 APK 文件进行分类。为接下来的章节奠定了理论基础。

第3章 用机器学习的技术实现APK的自动化分类

用机器学习的方法对 APK 文件进行分类大体分为两个阶段：训练和测试。与之对应的，我们分别称相应阶段所使用的 APK 样本为训练集和测试集。其中，训练集和测试集的样本没有交集。

在第一阶段，我们先对训练集中的 APK 样本分类，并根据预先定义的特征表从每个样本中提取特征向量。然后把从训练集中提取的特征向量和样本的正确类别作为学习算法的输入，即监督学习。通过分析和处理这些数据，学习算法会生成一个对应的分类模型。

在第二阶段，我们将使用训练阶段生成的分类模型对测试集中的 APK 样本进行分类。同训练集的操作方式一样，我们从测试集中的APK样本提取特征向量。分类器会基于这些特征向量对测试集中的样本分类。通过对比测试样本的分类结果和测试样本的真实类别，便可以评估分类器的性能。因此，我们也需要知道测试样本的真实分类。

3.1 种类的确定

研究 APK 的自动化分类，确定要分为哪些类是一个比较基础的问题。考虑到我们是通过机器学习的方式对 APK 进行自动分类，为了提高实验的成功率，我们尽量选择特征较为明显的分类。关于特征明显，最容易想到的也就是调用了特定的 API 或申请了特定权限的种类。例如，区分照相机类应用和短信类应用应该成功率会比较高。因为照相机类应用会向系统申请访问摄像头的权限，短信类应用会向系统申请发送短信的权限；相反，区分新闻类应用和浏览器类应用成功率会比较低。因为这两种类别的应用都没有申请特殊的权限。

最终，我们参考了电子市场对 APK 的分类和西安交通大学智能网络与网络安全实验室所做的 APK 分析系统中对 APK 的分类，结合种类数对准确率的可能影响，确定了如下分类：电话短信、地图、相机、电子书、系统、网络、其它。

3.2 分类特征选取

在机器学习的应用中，过量的特征会带来一系列的问题，例如：误导学习算法、过度拟合、增加模型复杂度。在准备阶段进行特征筛选可以帮助学习算法生成更加有效的，速度更快的分类器。不过，减少特征应该以保持分类器有较高的准确率为前提。

3.2.1 特征表示模型和选择算法

有关表示一个 APK 样本的问题主要集中在表示模型和特征选择算法上。关于样本表示模型，我们采用一个N维特征向量来代表一个 APK 样本。其中 N 代表特征的个数。特征向量的每一个元素都对应一个特征，对于可连续取值的特征，特征值取实数类型。对于不可连续取值的特征，若该样本存在这样的特征，则对应元素取值为1，否则取0。在不加筛选的情况下，我们可以从一个 APK 样本中提取出上千个特征。但是，如上文所说，过量的特征反而会给我们带来麻烦。因此，我们需要对特征向量进行降维，也就是特征抽取。

特征抽取的主要功能是在不损伤样本核心信息的情况下尽量减少要处理的特征数，以此来降低向量空间维数，从而简化计算，提高样本处理的速度和效率。通常根据某个特征评估函数计算各个特征的评分值，然后按评分值对这些特征进行排序，选取若干个评分值最高的作为特征。这里的评估函数就是需要我们选择的特征选择算法。

特征选择算法有很多，常见的有卡方检验[7]、Fisher得分法[8]和信息增益[9]。信息增益是信息论中的一个重要概念，它表示了某一个特征的存在与否对类别预测的影响，定义为考虑某一特征在样本中出现前后的信息熵之差。某个特征的信息增益值越大，贡献越大，对分类也越重要。Asaf[1] 等人在研究对 APK 的分类时，曾对这三种特征选择算法进行了比较全面的评估。我们参考其评估结果，最终决定选择信息增益的方法来完成特征选择的任务。

为了最大化的提高分类器的性能，使之耗费相对短的时间并获得相对高的准确率，我们应该保留多少个特征？为了解答这个问题，我们在实验中分别采用尝试了使用30,50,100,200,300,500个特征进行学习建模，并对实验结果进行了记录和分析。

3.2.2 特征提取方式

APK 文件的本质是一个 ZIP 格式的压缩包，却也不是普通的压缩包。APK 打包工具在打包 APK 文件前对其中的 XML 文件进行了编码，对其中的 Java 字节码进行了再编译。所以，从 APK 中提取特征实际上是一个反编译的过程。反编译的前提当然是要对 APK 以及 APK 中打包的所以内容有充分的了解。这包括对其中各种类型的文档的编码格式和其主要作用的熟知。Android 是开源系统，在其官方网站也有齐全的文档可以查阅。更幸运的是，随着 Android 反编译的技术越来越成熟，apktool 等开源软件相继被开发出来。在本课题的研究中，我们就是采用了开源工具 apktool 来完成了对 APK 特征的提取工作。通过 apktool 我们可以解码 APK 文件中的XML文件和资源文件，还可以反编译 APK 文件中 DEX 文件，获得应用程序的源代码。因此，我们从 APK 中提取的特征，主要来自于以下三方面：

1. 来自 APK 文件的特征。

   APK 文件是一种 ZIP 格式的压缩包，其中压缩了应用程序所需要可执行文件和一些图片音频等资源。既然 APK 是 ZIP 格式的压缩包，我们便也可以从 ZIP 格式的文件特性中提取到特征。例如文件大小、Zip entry 数量、子文件夹数量、各类型文件的数量。

2. 来自XML文件的特征

   一个 APK 文件中一般都包含几个 XML 文件。在程序的安装过程中，Android 系统会读取这些文件以完成应用程序安装。例如，AndroidManifest.xml 里面就有关于该应用的组件信息以及该应用所申请的权限信息。我们自己实现了一个 XML 文件分析器，读取反编译工具解码出的 XML 文件，并分析出以下特征：

   (a) XML 元素的数量，属性、特定字符串的数量。

   (b) 各属性的特征。

   (c) 各类型属性的数量。

   (d) AndroidManifest.xml 中的权限请求。

3. 来自 DEX 文件的特征。

   Android 系统的可执行文件是以 DEX 为扩展名结尾的文件。简单的，我们可以理解为 DEX 文件是对 Java 字节码文件的再编译。这里，我们借用了

开源工具 dex2jar 作为辅助，对 DEX 文件进行了分析。分析主要集中在，DEX 文件中所包含的字符串，域，类，调用的 API 等。

在预处理阶段，我们对从文件中提取的特征进行了初步的筛选。第一，去除了无用的特征。例如，在所有文件中特征值不变的特征。第二，去除了那些可以用来识别特定文件的特征。例如，特征只有两个值，只在一个文件有一个值，在所有其它文件中是另一个值。做了这些工作之后，总共余下不到1000个特征。接下来，我们利用信息增益算法对这些特征做了评估。分别取出了前30,50,100,200,300,500个特征。

3.3 模型的训练与测试

本课题中，我们使用 WEKA(Waikato Environment for Knowledge Analysis) 平台来进行机器学习的模型训练。WEKA 是新西兰怀卡托大学开发的一款开源的机器学习软件。该软件实现了机器学习中常用的分类、聚类算法，当然也实现了我们所要用到的 KNN 算法。利用这款软件，我们可以十分方便的进行模型训练和测试。

如同常见的数据分析软件一样，WEKA所处理的数据集是 ARFF(Attribute-Relation File Format) 格式的二维的表格。表格里的一个横行称作一个实例，相当于统计学中的一个样本，或者数据库中的一条记录。竖行称作一个属性，相当于统计学中的一个变量，或者数据库中的一个字段。这样一个数据集，便可以呈现各属性之间的潜在关系。我们只需将从 APK 样本提取的特征向量组织成 ARFF 格式的数据集，输入到 WEKA 中，选择分类算法即可训练模型。

举例说明，假设我们现在选取 INTERNET、RECEIVE SMS、READ CONTACTS、CAMERA 和 ACCESS GPS 五个权限信息作特征向量，要将 APK 分为地图、相机和短信三个类别，并提供6个样本作为训练集（实际情况下6个样本是远远不够用的）。那么我们要向 WEKA 提供的数据集如下：

```
01 % ARFF file for APK classification
02 @relation 'Andriod-Class'
03 @attribute 'INTERNET' {0,1}
04 @attribute ' RECEIVE_SMS ' {0,1}
05 @attribute 'READ_CONTACTS' {0,1}
06 @attribute 'CAMERA' {0,1}
```

```
07 @attribute 'ACCESS_GPS' {0,1}
08 @attribute 'CLASS' {'map','camera','sms'}
09 @data
10 % 6 instances
11 1, 0, 0, 0, 1, map
12 1, 0, 0, 1, 1, map
13 0, 0, 0, 1, 0, camera
14 1, 0, 0, 1, 0, camera
15 0, 1, 1, 0, 0, sms
16 1, 1, 1, 0, 0, sms
```

这里的 @relation 是关系声明，这个可以根据数据集的内容自定义。@attribute 是属性声明，指明了我们用于训练模型的特征名称。@data 后的数据便是我们真正的数据集，其中一行代表一个样本。数据顺序与@attribute 的声明顺序一致。

模型训练成功后，WEKA 会生成一个以 .model 为扩展名的模型文件。对该模型的测试过程和训练过程大同小异。只需从测试样本中提取出特征向量，并将其组织成和训练数据集相同格式的 ARFF 文件。用 WEKA 先后载入模型文件和测试数据即可获得测试结果。

目前国际上没有通用的用于 APK 自动分类研究的实验样本，本文采用了从 Google Play 上获取的一些 APK 样本。并人工将这些实验样本分成了七类：电话短信、地图、相机、电子书、系统、网络、其它。为了保持各种类样本数量的均匀，我们又对其进行了增补。另一方面，为了观察训练样本的个数对分类器性能的影响，我们特地减少了电子书类的 APK 样本。最终，实验用各样本的数量如表3–1所示。 其中，各类别中的样本不重叠。训练集和测试集中的样本不重叠。

表 3–1 训练集样本种类和数量

类型	短信	地图	相机	电子书	系统	网络	其它
训练集	300	300	300	150	300	300	300
测试集	50	50	50	50	50	50	50

3.4 性能评估与分析

3.4.1 评估的目的

性能评估的主要目的是找到一个性能较好的 APK 分类模型。具体的说，性能评估的主要目的是回答如下几个问题：

1. 用机器学习的相关技术来实现 APK 的自动分类的准确率如何？
2. 选用 KNN 算法进行分类时，k值选取多少较为合适？
3. 选择多少个特征较为合适？
4. 这些特征具体是什么？

为了回答这个问题，我们首先需要有一个能够对 APK 分类器性能进行评估的方法。

3.4.2 评估方法选择

因为 APK 分类从根本上说是一个映射过程。所以评估 APK 分类系统的标志是映射的准确程度和映射的速度。映射的速度取决于映射规则的复杂程度。而评估映射准确程度的参照物是通过人工思考判断后对样本的分类结果。这里假设人工分类完全正确并且排除个人思维差异的因素。与人工分类结果越相近，分类的准确程度就越高。为了评估 APK 分类系统和确定最佳k值，我们用到了如下四个指标准确率、召回率、F1 值和平均准确率。这里我们采用微平均的方式计算前三个指标，因此前三个指标可以用来评价分类系统在具体类别上的性能。最后一个指标用来评价分类系统的总体性能。

准确率是分类器分到该类别中的正确样本数占分到该类别的样本总数的比率。其数学公式为：

$$准确率 = \frac{该类别中正确样本数}{该类别中实际样本数} \tag{3–1}$$

召回率是分类器分到该类别中的正确样本数占测试样本中该类别的实际总数的比率。其数学公式为：

$$召回率 = \frac{该类别中正确样本数}{测试样本中该类别的实际总数} \tag{3–2}$$

准确率和召回率反映了分类质量的两个不同方面。二者必须综合考虑，不可偏废。因此，存在一种新的评估指标即 F1 测试值。其数学公式为：

$$F1 = \frac{召回率 \times 准确率 \times 2}{召回率 + 准确率} \tag{3–3}$$

3.4.3 数据结果与分析

KNN 算法中一个重要的参数就是k值，目前k值的确定并没有特定的公式或者方法。主要凭经验和实验确定。为了确定一个合适的k值，我们首先试探性的以100个特征作为特征向量。并在其它条件固定不变的情况下，改变k的值。分别使k取1,20,40,60,80,100,，并算出各个参数下的平均准确率。结果如图3–1所示。

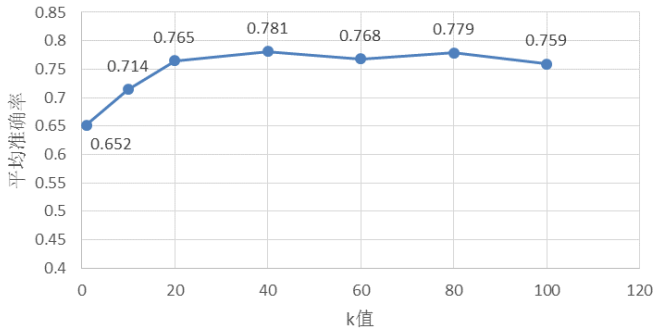

图 3–1 平均准确率与k值的关系

通过分析图3–1的数据，我们得知，在k取值很小的时候，分类的准确率会比较低。随着k的增大，平均准确率会有所增加。在20以后，平均准确率的值趋于稳定。当我们把特征的数量调整到200、300、500时，我们发现其平均准确率随k值的变化趋势大体相同。可见，在20以后，k值的大小对平均准确率的影响变的非常有限。在经过了多次实验并对实验结果进行分析后，我们决定将k的值定位20。这样即可以缩短处理时间，简化实验，又不会致使实验结果出现过大的偏差。

在确定了k值之后，我们需要通过实验来分析特征个数和分类器性能之间的关系。在特征数分别取30,50,100,200,300,500的情况下，我们取得的 F1 数据如表3–2所示。

表 3–2 特征个数与实验结果的F1值对应表

	类别	30	50	100	200	300	500
F1值	短信	0.728	0.739	0.727	0.744	0.75	0.748
	地图	0.703	0.709	0.717	0.754	0.743	0.741
	相机	0.734	0.737	0.735	0.739	0.736	0.733
	电子书	0.652	0.678	0.681	0.69	0.695	0.694
	系统	0.697	0.702	0.714	0.718	0.721	0.715
	网络	0.731	0.733	0.739	0.74	0.739	0.744
	其它	0.699	0.702	0.72	0.724	0.735	0.734

从表3–2中我们可以明显的看到，电子书的 F1 值在各个情况下表现的都比其它种类低。不难证明，这个是由于我们之前在训练集中取的电子书类样本过少的缘故。由于其样本少，在其分类的过程中，导致某些本该属于电子书类的样本被分到了其它类别。这会致使电子书类的召回率降低，从而 F1 值降低。

另一方面，数据表明，随着特征数的不断增多各个分类的 F1 值都呈上升的趋势，既性能越来越好。当然，处理样本的时间也越来越长。这里我们没有对处理时间进行定量的测量，不再给出具体数据。然而，当特征数量从300变为500时，各个分类的 F1 值反而有所减小。可以看出，这是由于特征个数过多，对分类算法产生了干扰。由此得出，在k值取20的情况下，特征个数分别取30,50,100,200,300,500的分类模型中，300个特征的模型性能最好。这300个特征是我们通过信息增益算法筛选出的前300个。

部分特征如下：

01 used-permission-INTERNET
02 used-permission-READ_PHONE_STATE
03 used-permission-RECEIVE_SMS
04 used-permission-WRITE_SMS
05 used-method-android-content-ContextgetResources
06 used-permission-ACCESS_FINE_LOCATION
07 used-permission-VIBRATE
08 apk-folder-res-raw
09 used-type-java-util-Random
10 used-permission-ACCESS_COARSE_LOCATION
11 used-permission-WAKE_LOCK
12 used-method-android-view-MotionEventgetAction
13 used-permission-READ_CONTACTS
14 used-permission-CAMERA
15 used-permission-CALL_PHONE
16 used-permission-ACCESS_GPS
17 used-permission-RECEIVE_BOOT_COMPLETED
18 used-permission-ACCESS_WIFI_STATE
19 used-permission-SEND_SMS
20 used-method-android-view-MotionEventgetX

```
21 used-permission-RESTART_PACKAGES
22 used-permission-READ_SMS
23 used-method-java-util-RandomnextInt
24 used-permission-WRITE_CONTACTS
25 used-permission-READ_LOGS
26 ......
```

以上列出的均是在 APK 分类中起重要作用的特征。经过观察，这些特征大部分是权限类特征。例如，短信类应用必须要用到的短信权限 used-permission-RECEIVE-SMS，地图类应用经常用到的 GPS 权限 used-permission-ACCESS-GPS。为什么是这些特征，而不是 APK 的大小等属性呢？其实也很容易理解。因为这些特征带有很强的类别属性。

经过以上的实验，我们得出用 KNN 算法对 APK 分类的准确率大致在70%至80%之间。在训练样本为2000个的情况下，KNN 算法中的k值取20，特征个数取300较为合适。这样即可以节约分类器的处理时间，在准确率上又不会有太大的损失。上文中已经列举了部分重要的特征，这里不再重复。

3.4.4 分类效果展示

经过了前面的实验及结果分析，我们从中选取了一个效果相对最好的模型。为了将分类效果可视化，以方便更好的展示成果，我们用程序将分类的结果画成了饼状图。同时还列出了 APK 文件中所包含敏感权限，以让人对 APK 有一个直观的了解。这里我们以一个系统工具软件绿色守护为例。绿色守护是一个可以管理后台应用及自启动的APP，属系统类应用。用我们的分类模型对绿色守护进行分类，结果如图3–2所示。

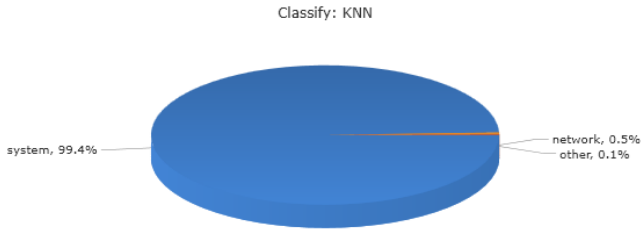

图 3–2 分类器对“绿色守护”的分类结果

从图3–2我们可以清晰的看到。分类器认为绿色守护为系统类应用的可能有99.4%，即分类器对绿色守护的分类结果同人工分类吻合。

3.5 本章小结

本章我们首先确定了将 APK 分为电话短信、地图、相机、电子书、系统、网络和其它，共七大类。然后介绍了我们在研究过程中如何从 APK 中提取用于分类的特征。接着阐述了用 WEKA 平台进行训练和测试的过程。最后，我们对测试的结果评估和分析并确定了本课题中 KNN 算法的k值、特征个数和具体的特征等重要的参数。

第4章 APK 相似性算法研究

前文中，我们阐述了如何用机器学习的技术对 APK 进行分类。然而，有时候这种分类并不能满足我们的需求，我们可能会想知道 APK 文件的家族属性。例如，同一个应用的两个版本，我们可以认为它们属于同一家族。同一恶意代码的不同变种，我们可以认为它们属于同一家族。换句话说，APK 的分类是在寻找 APK 间的相关性，而 APK 的家族属性分析则是在寻找 APK 间的相似性。

然而，我们如何验证两个文件相似呢？这不同于验证两个文件完全相同。验证两个文件完全相同很容易，只需要逐个对比两个文件的每个字节就可以确定。但若验证两个文件相似，我们就需要定义一个指标来衡量两个文件之间的相似性。

总体来说，APK 的相似性问题可以从两个维度考虑。一方面，从宏观的维度考虑，例如根据 APK 中调用的不同种类 API 的数量判断 APK 是否相似。另一方面，从微观的维度考虑，例如比对 APK 中的各个函数，根据两个 APK 中相同或相似函数的数量来判断 APK 是否相似。接下来，我们将详细介绍从宏观维度看问题的余弦相似度算法和从微观维度看问题的 NCD 算法。并对两个算法的优缺点进行对比分析。

4.1 余弦相似度算法计算APK间相似性

4.1.1 余弦相似度计算

余弦相似度[10]算法是利用两个文档的特征向量在空间中的夹角余弦值来表示相似度的一种方法。在二维空间中，向量夹角的余弦值越小，就代表两个向量越相似。在多维空间中，这种情况依然成立。余弦相似度的值在0到1间变化。0代表两特征向量成直角正交，两文档中没有任何相同的特征项，可以认为两文档相似度为零。1代表特征向量重合，两文档具有完全相同的特征项，而且出现的频率完全相同，可以认为两文档高度相似。

在向量空间模型中，文档泛指各种机器可读的记录，用D表示。特征项是指出现在文档D中，且能够代表该文档内容的基本单位，用D表示。特征项

主要是指关键词或者短语。文档可以用特征项集合表示为$D(T_1,T_2,...,T_n)$，其中T_k是特征项，$1 \leqslant k \leqslant n$。例如一篇文档中有 a、b、c、d 四个特征项，那么这篇文档就可以表示为$D(T_a,T_b,T_c,T_d)$。对含有n个特征项的文本而言，通常会给每个特征项赋予一定的权重表示其重要程度。即$D(T_1,W_1;T_2,W_2;...,T_n,W_n)$，简记为$D(W_1,W_2,...,W_n)$，我们把它叫做文档$D$的向量表示。其中$W_k$是$T_k$的权重，$1 \leqslant k \leqslant n$。假设a、b、c、d的权重分别为30，20，20，10，那么该文档的向量表示为$D(30,20,20,10)$。在向量空间模型中，两个文档D_x和D_y之间的内容余弦相似度$Sim(D_x,D_y)$可用如下公式计算：

$$Sim(D_x,D_y)=cos\theta=\frac{\sum_{k=1}^{n}W_{xk}\times W_{yk}}{\sqrt{(\sum_{k=1}^{n}W_{xk}^2)(\sum_{k=1}^{n}W_{yk}^2)}} \tag{4–1}$$

其中，W_{xk},W_{yk}分别表示文本D_x和D_y第k个特征项的权值，$1 \leqslant k \leqslant n$。

例如,文档D_x含有特征项T_a,T_b,T_c,T_d，它们权值分别为30,20,20,10，文档D_y含有特征项T_a,T_c,T_d,T_e，它们的权值分别为40,30,20,10，则文档D_x可以用向量表示为$D_x(30,20,20,10,0)$，文档D_y可以用向量表示为$D_y(40,0,30,20,10)$，根据余弦相似度公式，计算得$Sim(D_x,D_y)=0.86$。

4.1.2 APK 文件的宏观相似

用余弦相似度算法计算两个 APK 间相似性的关键工作就是从 APK 文件中提取出用于计算相似性的特征向量。在上一章的 APK 分类中，我们同样做了从 APK 中提取了特征向量的工作。这次的特征向量是否和上次的一样呢？显然是不同的。在 APK 的分类中，提取特征的标准是大而全，更加注重的是各 APK 间的相关性。而相似性的计算中，提取特征的标准是精而准，更加主要两 APK 间的内在相似性，即家族属性。要知道，属于同一类的两个 APK 并不一定属于同一家族。

从宏观的角度分析 APK ，指的是以 APK 调用的API及调用频率基础构建特征向量。我们认为，在特征 API 选择合适的情况下，如果两个 APK 调用的 API 种类及调用频率很相似，那么很大可能这两个 APK 也是相似的。之所以说是宏观角度，是因为在这里我们忽略了 API 的调用顺序和调用位置，只统计调用种类和调用频率。

在传统的余弦相似度中，各项特征值的权重一般根据 TF-IDF 算法得出。其中 TF 代表词频，IDF 代表逆向词频。由于这里没有语料库，所以这里不存在逆向词频。我们以各 API 的调用频率作为权重。

举例说明，这里有三个样本 opfake-01.apk、opfake-02.apk 和 TextReader.apk。其中，opfake-01 和 opfake-02 分别是 opfake 家族的两个变种，而 TextReader 是一个正常的 txt 阅读软件。它们哈希值依次为：

H1：b79106465173490e07512aa6a182b5da558ad2d4f6fae038101796b534628311

H2：b906279e8c79a12e5a10feafe5db850024dd75e955e9c2f9f82bbca10e0585a6

H3：01f6f6379543f4aaa0d6b8dcd682f4e2b106527584b3645eb674f1646faccad5

我们选取发送短信的接口 sendDataMessage 和 sendMultipartTextMessage 以及获取地理位置的接口 getLastKnownLocation 以及发起网络请求的接口 HttpPost，还有文件操作接口 createNewFile 为特征 API。经过统计，三个 APK 中各函

表 4–1 各样本调用相应 API 的频率

	Opfake-01	Opfake-02	TextReader
sendDataMessage	10	11	1
sendMultipartTextMessage	3	3	0
getLastKnownLocation	0	1	0
HttpPost	5	6	0
createNewFile	1	1	6

数的调用频率如表4–1所示。根据上文的描述分别构建三个样本的特征向量：$D_{OP1}=[10,3,0,5,1]$，$D_{OP2}=[11,3,1,6,1]$，$D_{TR}=[1,0,0,0,6]$。

根据公式计算得：$Sim(D_{OP1},D_{OP2})=0.960$，$Sim(D_{OP1},D_{TR})=0.226$。

根据余弦相似度的理论，值越接近1表示两个样本相似度越高，相反则表示样本相似度越低。本例，我们分析的三个样本中，前两个同属一个恶意代码家族，从计算结果看出其余弦相似度高达96%。而第三个样本是一个正常的 txt 文件阅读器，和前两个样本没有家族关系，其余弦相似度只有22.6%。

至此，还有两个问题需要去解决。如何从 APK 文件中提取 APK 调用的 API 呢？应该把哪些 API 作为特征纳入统计呢？

如何从 APK 文件中提取 API 是一个相对比较简单的问题。这里利用开源的用开源的 APK 逆向工具 baksmali 来完成 API 的提取任务。在用 baksmali 对 APK 进行反编译后会生成 Davlik 虚拟机语言表述的程序文件（以 .smali 为扩展名）。它是类似于汇编语言的低级语言，只需做简单的汇编就能转化为虚拟机可执行字

节码。正是由于此种特性，也就决定了其语法和格式的严谨性，语义上无二义性，可以作为比较 Android 应用相似性的基础文本。

以下是一段反编译后的代码：

```
01 # direct methods
02 .method constructor <init> (Lcld/navi/main-
03 frame/MainActivity;)V
04 .locals 0
05 .parameter
06 .prologue
07 .line 1
08 iput- object p1, p0, Lcld/navi/mainframe/Main-
09 Activity$1;- >this$0:Lcld/navi/mainframe/MainActivity;
10 .line 766
11 invoke- direct {p0}, Ljava/lang/Object;- ><init>( )V
12 return- void
13 .end method
```

通过对 smali 语法的操作符进行分析，大部分操作符只是对寄存器赋值、简单运算、判断跳转。还有一部分语句是函数调用类的语句。这些函数都是利用 invoke 关键字调用。因此可以从文件中提取出以 invoke 开头的所有函数调用行。通过字符串解析分析出被调用的函数。

例如：

```
01 invoke- virtual{v4,v0,v1,v2,v3},Test2.method5:(IIII)V
```

此条指令中，Test2.method5:（IIII）V 既是被调用的函数。

将哪些 API 纳入特征统计是个棘手的问题。API 的种类和数量的选择类似机器学习问题中的特征选择。种类过多会干扰到到相似度的计算。比如，如果将几乎所有的应用都需要用到的 Android 组件 API 纳入统计。那么所有样本的特征向量中，与 Android 组件有关的分量值都会很大。最后导致的就是，本来没有任何家族关系的样本会有很高的余弦相似度。相反，如果纳入统计的 API 的种类太少，则会导致提取的特征向量不能很好的代表特定的样本。计算结果肯定也是不准确的。

本课题中对相似性的研究，出发点是为分析 APK 文件的家族属性奠定基础。所以，我们希望有这样一组 API，这组 API 被某一家族的APK高频调用。例如，恶意代码家族 opfake 会高频调用与短信有关的敏感 API。当以这组 API 为特征统计 APK 样本的特征向量时，与该家族样本有关的计算会呈很高的准确率。还是以 opfake 家族为例，在已知一个 opfake 家族样本的情况下，当一个未知样本到来时，通过计算未知样本和已知样本的相似性就可以准确的判断出未知样本是否属于 opfake 家族。

通过阅读 opfake 家族样本的分析报告，我们从中选择了 sendDataMessage, sendMultipartTextMessage, getLastKnownLocation, HttpPost, createNewFile 等30个 API 作为特征。利用从 Google 的开源恶意代码库中获取的 opfake 样本，我们对余弦相似度算法的性能做了验证。计算结果表明，任意两个 opfake 样本的余弦相似度均在80%以上。而 opfake 样本与非 opfake 家族的样本的余弦相似度几乎都在60%以下。实验数据及结果分析将在下文中给出。

4.1.3 算法的优缺点分析

余弦相似度算法的理论简单，计算量小。该算法原是用于文本的相似性检测。用其来检测 APK 间的相似性实际上是将 APK 看作文本，调用的 API 看作文本的关键词。算法的主要工作量在特征权重、向量乘积和向量模的计算上。而本课题中，我们直接采用 API 的调用频率做权重，省去了特征权重的计算量。与此同时，通过统计特定家族高频调用的 API 频率，余弦相似度算法可以以极高的准确率识别出样本是否属于该家族。

但是，该算法的缺点也很明显。两个相似的 APK 样本，某些 API 的调用频率肯定也是相似的。而某些 API 调用频率相似的样本，却不一定真正的相似。这两者并不是充分必要条件。理论上，该算法的误报率会较高。另一方面，该算法的实现过程中，在 API 种类的选择问题上没有通用性。前面我们说过，统计特定家族高频调用的 API 可以提高在该家族样本上的准确率。但这也造成在非该家族样本的准确率大大降低。当然，在实际使用过程中我们可以通过设定多个 API 选择策略来缓解这个问题。

4.2 NCD 算法计算 APK 间相似性

4.2.1 APK 文件的微观相似

如上文所说，余弦相似度算法是从宏观上计算两个 APK 文件的相似性。虽然余弦相似度算法实现简单，速度快，但是其误报率却比较高。为了降低误报率，提高相似性计算的准确性，本文提出了另一种计算方法，即从微观的角度计算相似性。从微观角度计算，指的是计算两个 APK 文件中各个函数的相似性。最后根据相似或相同函数的个数得出一个能够反映两个 APK 文件相似性的值。这里我们采用 NCD 算法计算函数间的相似性。

NCD 是一种用来测量两个序列之间相似性的方式[11]。其应用范围[12, 13]很广，可以用于任何可以序列化的对象之间的相似性检测，例如文件、邮件、音频、视频、图片。因此，我们只需将 APK 文件中函数序列化，便可以用 NCD 算法计算其相似性。

对于给定的两个序列x和y，并且假设相连接后的序列为S，使用的压缩算法为$Comp$，序列压缩后的序列长度表示为$L(Comp(s))$，序列的长度[14]可以通过压缩后字符串的字节数或者字符个数等表示。因此，用 NCD 算法计算序列x和y之间的距离$d_{NCD}(x,y)$，表示为：

$$d_{NCD}(x,y)=\frac{L(Comp(S))-min\{L(Comp(x)),L(Comp(y))\}}{max\{L(Comp(x)),L(Comp(y))\}} \tag{4–2}$$

$d_{NCD}(x,y)$返回值在0.0到1.0之间。$d_{NCD}(x,y)$的值越小表示x和y的相似性越高。0.0代表完全相同，1.0代表完全不同。

4.2.2 函数的序列化

从微观角度计算 APK 文件的相似性的第一步是对 APK 进行反编译。这里我们使用开源工具 baksmali，该工具以 APK 中的可执行文件 classex.dex 为输入，以扩展名 smali 的文件为输出。输出的 smali 文件相当于是 Android Dalvik 虚拟机的汇编代码。从这些文件里我们几乎可以获得函数的任何信息，例如参数、变量、执行逻辑、调用的 API 等。

为了用 NCD 算法计算两个函数的相似性，我们必须要将函数序列化。将函数进行序列化的方式有如下几种：

1. 将函数看做是由0和1组成的比特序列。

2. 将函数看做事由字符组成的字符序列。

3. 从函数中抽取一个可以代表其执行逻辑的字符序列。

第一种方式最简单，不用对函数做任何处理。但如果这么做的话，相似性的识别率一定很低。可想而知，我们只需在函数中多加些空格便可以改变整个函数的比特序列。此时，虽然修改前和修改后的函数内容没有改变。但其计算结果的相似性却会很低。

第二种方式需要将函数中的空格换行等多余的格式字符过滤，仅保留一长串没有语义的字符序列。此种方法避开了第一种方式的缺点，却也存在问题。我们知道，Android 应用采用 Java 为编写语言。因此 APK 文件天生就具有容易被反编译的弱点。当然，这也是本文研究的基础。软件开发者们为了防止自己的软件被反编译，从而保护自己的版权，便会对自己的代码进行混淆。例如原本名叫 start() 的函数，经过混淆可能变成了 a()。当然，函数的执行逻辑基本不会改变。试想，我们对一个函数进行混淆。混淆前和混淆后的函数应该是极为相似的。但是用第二种方式对函数进行序列化，并用 NCD 方法计算相似性，结果可想而知。计算所得的相似度会很低。显然这不是我们想要的。

实际上我们之所以认为两个函数是相似的，是因为两个函数的执行逻辑相似，所完成的功能相似。因此对函数进行序列化的最好方式应该是第三种，即从函数中抽取其执行逻辑并用一个字符序列代表它。

那么我们怎样才能从函数中抽取执行逻辑呢。这是我们想到了编译原理中的相关知识—文法，用一个大写字母来代表一种关键的执行逻辑。例如我们用 G 来代表一条 goto 跳转语句。接下来函数逻辑的提取我们将基于 smali 的文法规则进行。smali 语言的文法[15]如下：

```
01 Procedure ::= StatementList
02 StatementList ::= Statement | Statement StatementList
03 Statement ::= BasicBlock | Return | Goto | If | Field |
04 Package | String
05 Return ::= 'R'
06 Goto ::= 'G'
07 If ::= 'I'
08 BasicBlock ::= 'B'
09 Field ::= 'F'0 | 'F'1
```

```
10 Package ::= 'P' PackageNew | 'P' PackageCall
11 PackageNew ::= '0'
12 PackageCall ::= '1'
13 PackageName ::= Epsilon | Id
14 String ::= 'S' Number | 'S' Id
15 Number ::= \d+
16 Id ::= [a-zA-Z]\w+
```

这里我们不考虑代码中具体的参数和变量，而只关心函数的执行逻辑。例如,对于如下一段代码：

```
01     mov X, 4
02     mov Z, 5
03     add X, Z
04     goto +50
05     add X, Z
06     goto -100
```

提取的序列将是“B[G]B[G]”。

对于真实的 smali 代码，这个序列要更加复杂一些。例如，假设有如下代码：

```
01 [...]
02 call [ meth@ 22 Ljava/lang/String; valueOf['(I)', ↩
03 'Ljava/lang/String;'] ]
04 goto 50
```

提取的序列将是“B[P1Ljava/lang/String; valueOf (I)Ljava/lang/String;G]”。

提取执行序列的过程就像绘制控制流图一样，但是这里我们只关心 if，goto 等关键指令。稀疏的 switch 语句也会被翻译成 goto 语句，并忽略掉细节。与此同时我们也可以在序列中保存了一些关键的包名、类名和系统 API 等信息。当然，你也可以选择记录更加详细的信息。但是，在后期的实验中，我们发现更详细的信息实际是多余的。因为这里我们关心的是更高层次的相似性而不是细节上的一致性。

这里我们预定义了四种序列：

1. 0型，包含该函数所调用的包名、域名等信息。
2. 1型，在0型的基础上过滤掉 Android 的包名、域名等信息。
3. 2型，在0型的基础上过滤掉 Java 的包名、域名等信息。

4. 3型，在0型的基础上过滤掉 Android 和 Java 的包名、域名等信息。

这样，在我们的实验过程中便可以动态的去调整提取序列的策略。

举个例子，下面是我们用 baksmali 对某 APK 进行反编译，得到其中一个函数的代码：

```
01 testCFG-BB@0x0 :
02        0(0) const/4 v0 , [ #+ 1 ] // {1}
03        1(2) const/4 v1 , [ #+ 1 ] // {1}
04        2(4) const/4 v2 , [ #+ 1 ] // {1}
05        3(6) const/4 v3 , [ #+ 1 ] // {1} [ testCFG-BB@0x8 ]
06
07 testCFG-BB@0x8 :
08        4(8) iget-boolean v4 , v7 , [ field@ 14 ↩
09        Ltests/androguard/TestIfs;
10             Z P ]
11        5(c) if-eqz v4 , [ + 77 ] [ testCFG-BB@0x10  testCFG-↩
12        BB@0xa6 ]
13
14 testCFG-BB@0x10 :
15        6(10) move v1 , v0
16        7(12) iget-boolean v4 , v7 , [ field@ 15 ↩
17        Ltests/androguard/TestIfs;
18              Z Q ]
19        8(16) if-eqz v4 , [ + 70 ] [ testCFG-BB@0x1a  testCFG-↩
20        BB@0xa2 ]
21
22 testCFG-BB@0x1a :
23        9(1a) const/4 v3 , [ #+ 2 ] // {2} [ testCFG-BB@0x1c ]
24
25 testCFG-BB@0x1c :
26        10(1c) add-int/lit8 v2 , v2 , [ #+ 1 ] [ testCFG-↩
27        BB@0x20 ]
28
```

```
testCFG-BB@0x20 :
      11(20) sget-object v4 , [ field@ 0 Ljava/lang/System;
            Ljava/io/PrintStream; out ]
       12(24) new-instance v5 , [ type@ 25 ↩
       Ljava/lang/StringBuilder; ]
       13(28) invoke-static v0 , [ meth@ 22 ↩
       Ljava/lang/String; valueOf
              ['(I)', 'Ljava/lang/String;'] ]
       14(2e) move-result-object v6
       15(30) invoke-direct v5 , v6 , [ meth@ 25 ↩
       Ljava/lang/StringBuilder;
              ['(Ljava/lang/String;)', 'V'] ]
       16(36) const-string v6 , [ string@ 5 ',' ]
       17(3a) invoke-virtual v5 , v6 , [ meth@ 31
              Ljava/lang/StringBuilder; append ['(↩
              Ljava/lang/String;)',
              'Ljava/lang/StringBuilder;'] ]
       18(40) move-result-object v5
       19(42) invoke-virtual v5 , v1 , [ meth@ 28
              Ljava/lang/StringBuilder; append ['(I)',
              'Ljava/lang/StringBuilder;'] ]
       20(48) move-result-object v5
       21(4a) const-string v6 , [ string@ 5 ',' ]
       22(4e) invoke-virtual v5 , v6 , [ meth@ 31
              Ljava/lang/StringBuilder; append ['(↩
              Ljava/lang/String;)',
              'Ljava/lang/StringBuilder;'] ]
       23(54) move-result-object v5
       24(56) invoke-virtual v5 , v2 , [ meth@ 28
              Ljava/lang/StringBuilder; append ['(I)',
              'Ljava/lang/StringBuilder;'] ]
```

```
60        25(5c) move-result-object v5
61        26(5e) const-string v6 , [ string@ 5 ',' ]
62        27(62) invoke-virtual v5 , v6 , [ meth@ 31
63               Ljava/lang/StringBuilder; append ['(
64               Ljava/lang/String;)',
65               'Ljava/lang/StringBuilder;'] ]
66        28(68) move-result-object v5
67        29(6a) invoke-virtual v5 , v3 , [ meth@ 28
68               Ljava/lang/StringBuilder; append ['(I)',
69               'Ljava/lang/StringBuilder;'] ]
70        30(70) move-result-object v5
71        31(72) invoke-virtual v5 , [ meth@ 32 
72        Ljava/lang/StringBuilder;
73               toString ['()', 'Ljava/lang/String;'] ]
74        32(78) move-result-object v5
75        33(7a) invoke-virtual v4 , v5 , [ meth@ 8 
76        Ljava/io/PrintStream;
77               println ['(Ljava/lang/String;)', 'V'] ] [ 
78               testCFG-BB@0x80 ]
79
80 testCFG-BB@0x80 :
81        34(80) iget-boolean v4 , v7 , [ field@ 16
82               Ltests/androguard/TestIfs; Z R ]
83        35(84) if-eqz v4 , [ + 4 ] [ testCFG-BB@0x88  testCFG-
84        BB@0x8c ]
85
86 testCFG-BB@0x88 :
87        36(88) add-int/lit8 v3 , v3 , [ #+ 4 ] [ testCFG-
88        BB@0x8c ]
89
90 testCFG-BB@0x8c :
```

```
91        37(8c) iget-boolean v4 , v7 , [ field@ 17
92               Ltests/androguard/TestIfs; Z S ]
93        38(90) if-eqz v4 , [ + -8 ] [ testCFG-BB@0x94  ↩
94        testCFG-BB@0x80 ]
95
96 testCFG-BB@0x94 :
97        39(94) add-int/lit8 v0 , v0 , [ #+ 6 ]
98        40(98) iget-boolean v4 , v7 , [ field@ 18
99               Ltests/androguard/TestIfs; Z T ]
100        41(9c) if-eqz v4 , [ + -74 ] [ testCFG-BB@0xa0  ↩
101        testCFG-BB@0x8 ]
102
103 testCFG-BB@0xa0 :
104        42(a0) return-void
105
106 testCFG-BB@0xa2 :
107        43(a2) const/4 v3 , [ #+ 3 ] // {3}
108        44(a4) goto [ + -68 ] [ testCFG-BB@0x1c ]
109
110 testCFG-BB@0xa6 :
111        45(a6) add-int/lit8 v2 , v2 , [ #+ 2 ]
112        46(aa) goto [ + -69 ] [ testCFG-BB@0x20 ]
```

对该函数提取0型序列。结果为：

“B[]B[I]B[I]B[]B[]B[P0Ljava/lang/StringBuilder;
P1Ljava/lang/String;valueOf(I)Ljava/lang/String;
P1Ljava/lang/StringBuilder;(Ljava/lang/String;)V
P1Ljava/lang/StringBuilder;append(Ljava/lang/String;)Ljava/lang/StringBuilder;
P1Ljava/lang/StringBuilder;append(I)Ljava/lang/StringBuilder;
P1Ljava/lang/StringBuilder;append(Ljava/lang/String;)
Ljava/lang/StringBuilder;
P1Ljava/lang/StringBuilder;append(I)Ljava/lang/StringBuilder;

P1Ljava/lang/StringBuilder;append(Ljava/lang/String;)
Ljava/lang/StringBuilder;
P1Ljava/lang/StringBuilder;append(I)Ljava/lang/StringBuilder;
P1Ljava/lang/StringBuilder;toString()Ljava/lang/String;
P1Ljava/io/PrintStream;println(Ljava/lang/String;)V]
B[I]B[]B[I]B[I]B[R]B[G]B[G]”。
提取3型序列，结果为：
“B[]B[I]B[I]B[]B[]B[P0P1P1P1P1P1P1P1P1P1]B[I]B[]B[I]B[I]B[R]B[G]B[G]”。

从中提取0型序列，我们可以看到每个执行块中的具体信息。而从中提取3型序列更容易看出函数的执行逻辑。

比较有趣的是，我们发现即使程序基本块的顺序发生变化，NCD 算法仍然会计算出较高的相似性。这可能是由于NCD算法是基于柯氏复杂性的特点。由此可见，柯氏复杂性天然的绕过了混淆程序对函数执行逻辑的混淆。

4.2.3 压缩算法的选择

解决了函数的序列化的问题后，我们回到 NCD 算法本身。NCD 算法的一个关键部分就是压缩算法。一个好的压缩算法可以大大提高计算的准确度，但是压缩算法的时间性能却也是我们不得不考虑的问题。如果选择 LZMA[16]这样的压缩算法，恐怕就只能用来计算 Hello World 这样的简单问题了。

文献[12]指出，如果想让 NCD 算法发挥其计算相似性的作用，压缩算法的选择应该满足如下几个特性：

1. 幂等性 $Comp(xx) = Comp(x)$
2. 单调性 $Comp(xy) \geq Comp(x)$
3. 对称性 $Comp(xy) = Comp(yx)$
4. 分散性 $Comp(xy) + Comp(z) \leq Comp(xz) + Comp(yz)$

为了评估压缩算法的时间性能以及测试其是否满足以上特性，我们进行了一个对比实验。如果压缩算法需要耗费很多的时间代价，那么它在这里几乎就可以被放弃了。为了更贴近真实的应用场景，这里我们采用随机的函数逻辑序列做实验。实验结果如表4–2所示。

从表4–2的数据我们可以看出，SNAPPY 算法的性能似乎更加出色。无论是从时间性能考虑还是从满足 NCD 算法要求的特性考虑，我们都应该选用 SNAPPY

表 4–2 各压缩算法的实验结果

属性	满足属性的个数	压缩结果总大小（B）	时间（s）
LZMA			
幂等性	0/9	1167	1.82118797
单调性	72/72	13258	9.40736294
对称性	72/72	17380	9.32561111
分散性	504/504	214466	133.6742709
BZ2			
幂等性	0/9	1947	0.00075889
单调性	72/72	18248	0.00626206
对称性	72/72	221744	0.00735211
分散性	504/504	279944	0.09816098
ZLIB			
幂等性	0/9	1073	0.00033116
单调性	72/72	11850	0.0022459
对称性	72/72	15348	0.00276113
分散性	504/504	190386	0.0346849
XZ			
幂等性	0/9	1900	0.55278206
单调性	72/72	17544	4.41346812
对称性	72/72	21008	4.35566306
分散性	504/504	269864	61.70975709
VCBLOCKSORT			
幂等性	0/9	8129	0.00140786
单调性	72/72	86960	0.0169549
对称性	10/72	115168	0.02190304
分散性	504/504	1414896	0.21149492
SNAPPY			
幂等性	0/9	1153	0.00009203
单调性	72/72	12952	0.00057387
对称性	72/72	17184	0.00059295
分散性	504/504	210952	0.01117182

算法。这里值得注意的是，无论哪个算法都无法满足幂等性。经过实验我们发现，虽然幂等性没有被完全满足，但实际上是近似满足的。因此，我们的压缩算法依然有效。

当我们用 SNAPPY 为压缩算法，计算字符序列“B[G]B[G]”和“B[G]B[G]”的 NCD 值时，我们得到$d = 0.027692307978868484$（如果为0，则说明幂等性满足）。结果不是0，却也很接近，因此我们认为其近似满足幂等性。

SNAPPY 压缩算法的压缩速度是最快的，压缩率却很一般。该问题中压缩率并不重要，我们更加关心的是压缩速度和是否满足 NCD 要求的特性，因此我们最终还是决定选用 SNAPPY 算法。

4.2.4 处理流程及结果计算

NCD 算法返回的结果为0.0到1.0之间的值。其中0.0代表完全相同，1.0代表完全不同。因此，这里我们用返回结果与1.0之间的差值来代表两个函数的相似性。例如序列A与序列B的 NCD 值为0.1，则序列A与序列B的相似性为0.9，即90%。

上一小节中，我们看到实际应用中幂等性并没有得到满足。这意味着，两个完全相等的序列的 NCD 不为0。为了解决这个问题，我们想到了哈希。在计算 NCD 之前，先对比两个序列的哈希是否相等，若相等，则可以直接判断两个序列相同即，相似度100%。

至此，APK 中的一个函数将由以下两个元素代表：

1. 一个代表函数执行逻辑的字符序列
2. 一个哈希值

其中，哈希值是为了快速判断两个函数是否相同。字符序列是为了计算两个函数的相似性。

这里我们将 APK 看作是函数的集合，因此对比两个 APK 的相似性的问题可以描述为：从集合A和集合B中找出相同函数的子集、相似函数的子集和不同函数是子集。

算法的输入和输出如下：

1. 输入：集合A、集合B
2. 输出：集合I、集合S、集合D和集合Sk

输入的集合A和集合B代表安装包A和安装包B的函数集。输出中，集合I代表集合A和集合B中相同的子集；集合S代表集合A和集合B中相似的子集；集合D代表集合A和集合B中不同的子集；集合Sk则代表算法直接跳过的函数集。由于并不是分析所有的函数都有益于相似性的计算，所以我们会跳过某些没有对计算相似性没有帮助的函数。例如，某些函数非常简短，甚至是空函数，这类函数无法反映 APK 的处理行为，我们会选择跳过处理。

算法的处理流程如图4–1所示。

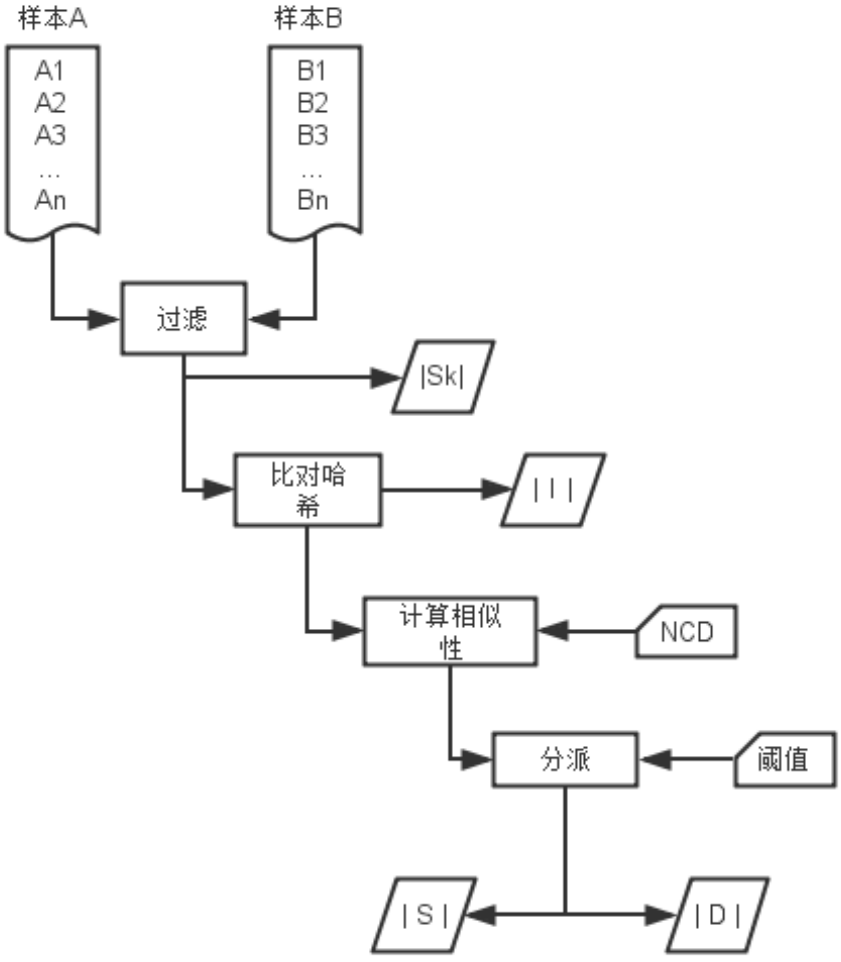

图 4–1 NCD算法计算相似性流程图

最后，我们将根据集合A和集合B中相同函数的个数和相似函数的 NCD 值计算出一个可以反映 A 和 B 相似性的值（0.0%至100.0%）。具体的计算公式如下：

$$Score = \frac{|I| + \sum_{i \in S}(1 - d_{NCD,i})}{|I| + |S| + |D|} \times 100\% \tag{4–3}$$

其中$|I|,|S|,|D|$分别表示各集合所含元素的个数。假设，A和B的计算结果为：$|I|=34,|S|=5,|D|=|Sk|=0$。且其中集合S的每个元素的 NCD 值均为0.2。
则有：

$$Score=\frac{34+(1-0.2)\times 5}{34+5+0}\times 100\%=97.4\%$$

4.2.5 算法优缺点分析

不同于余弦相似度算法，NCD 算法具有较强的通用性。它不需要根据不同的家族做出特征选择上的调整。不管处理的样本是属于哪个家族，它的性能都不会受到太大的影响。由于NCD算法是从微观上对 APK 进行了分解，若样本内在相似性不高，NCD 算法计算所得的结果也不会有很高的相似性。换句话说，NCD 算法的误报率会比余弦相似度算法低。

另一方面，NCD 算法也有一个很大的缺点。NCD 算法比较复杂，相对余弦相似度算法，其计算量要大很多。这里的计算量主要是在函数的序列化、哈希值计算以及相似函数的查找过程。现在的游戏类应用动辄几十M，用 NCD 算法处理就显的很吃力，低效。

4.3 算法性能对比

余弦相似度算法的实现中，我们取 opfake 恶意代码家族调用频率较高的30个 API 构建特征向量。NCD 算法的实现中，我们对所有 APK 样本都存在的函数进行了过滤。下面我们通过实验来对比这两个算法的性能。

实验一，取opfake家族的一个 APK 做参照样本。另取20个 opfake 家族的 APK 和20个 foncy 家族的 APK 做测试样本。用余弦相似度算法和NCD算法分别计算每个测试样本与参照样本间的相似性。实验结果如表4–3所示。

表 4–3 以 opfake 样本为参照样本的实验结果

	余弦相似度算法		NCD算法	
数值范围	Opfake样本	foncy样本	Opfake样本	foncy样本
0.8 - 1.0	20/20	0/20	18/20	0/20
0.5 - 0.8	0/20	3/30	2/20	0/20
0.0-0.5	0/20	17/20	0/20	20/20
平均处理时间	0.23112454s		109.42562621s	

表4–3的数据表明，当参照样本为 opfake 样本时，余弦相似度算法和 NCD 算法都能很好的识别出与参照样本相似的 opfake 测试样本。20个 opfake 测试样本与 opfake 参照样本的余弦相似值均在0.8以上，18个的 NCD 值在0.8以上，2个在0.5至0.8的范围内。可见NCD有一定的漏报率。20个 foncy 样本与 opfake 参照样本的NCD值都在0.5以下，17个余弦相似值在0.5以下，3个在0.5至0.8的范围。可见余弦相似性算法有表现出误报的趋势。

实验二，将参照样本替换为 foncy 家族的样本。其它条件同实验一。实验结果如表4–4所示。

表 4–4 以 foncy 样本为参照样本的实验结果

	余弦相似度算法		NCD算法	
数值范围	foncy样本	Opfake样本	foncy样本	Opfake样本
0.8 - 1.0	6/20	0/20	17/20	0/20
0.5 - 0.8	11/20	4/20	3/20	1/20
0.0-0.5	3/20	16/20	0/20	19/20
平均处理时间	0.29312873s		108.63764824s	

表4–4的数据表明，当参照样本为 foncy 样本时，NCD 算法依然能够很好的识别出与 foncy 参照样本相似的 foncy 测试样本。而余弦相似算法的表现却十分糟糕。20个 foncy 测试样本与 foncy 参照样本的余弦相似值没有像预期的那样集中在0.8以上，而是集中在0.5至0.8的范围内。也就是说余弦相似无法很好的计算出 foncy 家族样本间的相似性。事实上，这是由于这里用于计算余弦相似性的特征 API 大部分是 opfake 家族常用的，在 foncy 中并不常用。因此从中提取的特征向量无法反映 foncy 的真实特性，计算结果自然不准确。而NCD算法在本轮实验的数据则比较稳定。

时间性能上，两者相差巨大。实际上这是由算法本身的复杂性决定的。余弦相似度算法从宏观考虑，特征向量构建完成后只有一次相似性的计算。而 NCD 算法则是从微观角度将 APK 分解成了函数集合，在两个集合中寻找相同和相似的函数需要不断的遍历。综上，余弦相似算法和 NCD 算法各有优势和缺点。余弦相似度简单，计算迅速，计算的准确率也很高。但是特征的选取没有通用性。而 NCD 算法则具有较强的通用性，它根据代码执行逻辑判断 APK 的相似，没有特征选择的问题。但是由于 NCD 算法复杂的处理过程，其时间性能很差。甚至于，在有大量 APK 需要处理的情况下，它所耗费的时间是让人不可接受的。

4.4 本章小结

本章我们研究了用于分析 APK 家族属性的相似算法。分别从宏观和微观角度提出了用余弦相似度算法和 NCD 算法计算APK相似性的理论，详细的阐述了算法的处理过程及参数的选择，分析了算法的优缺点。并通过两个实验对算法的性能做了对比。

第5章 结束语

本文主要阐述了如何通过分析 APK 的特征来确定 APK 的类别及家族属性的理论及方法。本课题来源于APK相关研究工作的实践过程。经多方调研，课题的研究价值得到了肯定，最终该课题被确定为毕业设计题目。

文中先是对课题的背景、意义和国内外的相关研究工作进行了详细介绍和探讨。接着分别从理论和应用方法两个层次阐述了如何将机器学习的方法运用到 APK 文件的自动化分类以及如何用余弦相似度算法和 NCD 算法计算 APK 文件间的相似性。

在 APK 的分类问题上，我们吸取了前人在类别的确定和特征筛选方面的经验。与此同时我们也创新性的将 KNN 方法应用到了其中。更重要的，本文提出了一套APK分类器的性能评估方法，并通过多次实验确定了K值、特征个数和特征内容等重要参数。实验数据表明，文中训练出的模型的分类准确率不够高，效果并没有达到让人十分满意。机器学习离不开大量的训练和重复的实验，文中所进行的实验次数和训练数据时远远不够的。在今后的研究里，我们的分类器还有很大的改进空间。

而对于 APK 相似性计算的研究，我们的最终目的是通过计算相似性获取 APK 的家族属性。文中我们分别提出了从宏观角度分析APK的余弦相似度算法和微观角度分析 APK 的 NCD 算法。无论是算法理论，还是算法对 APK 的处理过程，文中都进行了细致的描述。为了对比两个算法的性能，文中设计了两个实验。实验数据表明余弦相似度算法和NCD算法各有优缺点。余弦相似度算法实现起来简单方便，计算量小。但是在特征选择问题上，有较大的局限性。NCD算法的优点是通用性强，没有特征选择上的局限性。但是 NCD 算法的实施步骤较为复杂，计算量非常大。在时间消耗上和余弦相似度算法相差三个数量级。要想将余弦相似度应用于家族属性分析系统，还有更多的工作等着我们去做。

本文的工作只是我们在 APK 特征分析方向进行更深入研究的奠基石。要想APK自动化分类和家族属性分析功能应用于工业界，我们还需进行下面的工作。

第一，对 APK 分类系统进行更多训练和并不断根据实验效果进行参数的调整。文中在算法选择和参数确定问题上，处理方法都太过粗犷。通过更多的实验和更细致的理论分析会有利于提高分类器的准确率。

第二，文中的 APK 相似算法未成熟。下一步的工作将尝试通过建立家族 API 库来解决特征 API 选择问题上的局限性问题。至于 NCD 算法，将考虑在相似函数的查找过程中引入聚类算法，以减少计算量。

参考文献

[1] A. Shabtai, Y. Fledel, Y. Elovici. Automated static code analysis for classifying android applications using machine learning[C]//Computational Intelligence and Security (CIS), 2010 International Conference on. .[S.l.]: [s.n.] , 2010:329–333.

[2] B. Sanz, I. Santos, C. Laorden, et al. Puma: Permission usage to detect malware in android[C]//International Joint Conference CISIS’12-ICEUTE´12-SOCO´12 Special Sessions. .[S.l.]: [s.n.] , 2013:289–298.

[3] 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).

[4] A. L. Samuel. Programming computers to play games[J]. Advances in computers, 1960, 1:165–192.

[5] J. R. Anderson, R. S. Michalski, R. S. Michalski, et al. Machine learning: An artificial intelligence approach[M]. Vol. 2.[S.l.]: Morgan Kaufmann, 1986.

[6] 张宁, 贾自艳, 史忠植. 使用 KNN 算法的文本分类[J]. 计算机工程, 2005, 31(8).

[7] I. Imam, R. Michalski, L. Kerschberg. Discovering attribute dependence in databases by integrating symbolic learning and statistical analysis techniques[C]//Proceeding of the AAAI-93 Workshop on Knowledge Discovery in Databases, Washington DC. .[S.l.]: [s.n.] , 1993.

[8] T. R. Golub, D. K. Slonim, P. Tamayo, et al. Molecular classification of cancer: class discovery and class prediction by gene expression monitoring[J]. science, 1999, 286(5439):531–537.

[9] C. E. Shannon. A mathematical theory of communication[J]. ACM SIGMOBILE Mobile Computing and Communications Review, 2001, 5(1):3–55.

[10] 张振亚, 王进, 程红梅,等. 基于余弦相似度的文本空间索引方法研究[J]. 计算机科学, 2005, 32(9):160–163.

[11] M. Li, P. M. Vitányi. An introduction to Kolmogorov complexity and its applications[M].[S.l.]: Springer, 2009.

[12] A. N. Kolmogorov. Three approaches for defining the concept of information quantity[J]. Problems of Information Transmission, 1965, 1(1):3–8.

[13] D. Sankoff, J. B. Kruskal. Time warps, string edits, and macromolecules: the theory and practice of sequence comparison[J]. Reading: Addison-Wesley Publication, 1983, edited by Sankoff, David; Kruskal, Joseph B., 1983, 1.

[14] R. Cilibrasi, P. M. Vitányi. Clustering by compression[J]. Information Theory, IEEE Transactions on, 2005, 51(4):1523–1545.

[15] S. Cesare, Y. Xiang. Classification of malware using structured control flow[C]//Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing-Volume 107. .[S.l.]: [s.n.] , 2010:61–70.

[16] M. Cebrian, M. Alfonseca, A. Ortega, et al. Common pitfalls using the normalized compression distance: What to watch out for in a compressor[J]. Communications in Information & Systems, 2005, 5(4):367–384.

致 谢

经过了两个多月的不懈努力，终于完成了毕业设计课题。回顾这两个多月来的研究过程，我得到了太多的恩惠和帮助，在此一并致谢。

在此论文完成之际，谨向我尊敬的指导老师——王勇老师致以诚挚的谢意和崇高的敬意。王勇老师深厚的学术功底，丰富的实践经验，对学生的严格要求和耐心指导，灵活的创新思想和富有激情的人生态度让我受益匪浅。

感谢这篇论文所涉及到的各位学者。本文引用了数位学者的研究文献，如果没有各位学者的研究成果的帮助和启发，我将很难完成本篇论文的写作。

感谢所有开源代码贡献者，你们无私的开源精神是科技进步的有力辅助，没有你们的代码我根本无法在这么短的时间内完成文中设计的实验。

感谢我所有在我毕业设计过程中提供帮助的老师、师兄师姐以及同学，你们的帮助大大推进了本课题的进展速度。

最后向牺牲自己宝贵时间对我的毕设论文予以检查、指导和评审及参加我毕业设计答辩的老师们表示由衷的感谢。

Automated Static Code Analysis for Classifying Android Applications Using Machine Learning

1.1 Abstract

In this paper we apply Machine Learning (ML) techniques on static features that are extracted from Android’ s application files for the classification of the files.Our evaluation focused on classifying two types of Android applications: tools and games. Successful differentiation between games and tools is expected to provide positive indication about the ability of such methods to learn and model Android benign applications and potentially detect malware files. The results of an evaluation, performed using a test collection comprising 2,285 Android .apk files, indicate that features, extracted statically from .apk files, coupled with ML classification algorithms can provide good indication about the nature of an Android application without running the application, and may assist in detecting malicious applications. This method can be used for rapid examination of Android .apks and informing of suspicious applications.

Keywords- Mobile Devices, Machine Learning, Malware, Security, Android, Static analysis

1.2 INTRODUCTION

Smart phones have evolved from simple mobile devices into sophisticated yet compact minicomputers. Designed as open, programmable, networked devices, smart phones are susceptible to various malware threats such as viruses, Trojan horses, and worms, all of which are well-known from desktop platforms. These devices enable users to browse the Internet, receive and send emails, SMSs, and MMSs, exchange information with other devices, and activate various applications, which make these devices potential attack targets.

A compromised smart-phone can inflict severe damages to both users and the cellular service provider. Malware on a smart-phone can make the phone partially or fully

unusable; cause unwanted billing; steal private information; or infect every name in a user’s phonebook. Possible attack vectors into smart phones include: Cellular networks, Internet connections (via Wi-Fi, GPRS/EDGE or 3G network), USB and other peripherals.

Among the most significant smart-phone operating systems that have arisen recently is Google’s Android. Android 1 is a comprehensive software framework for smart mobile devices and it includes an operating system, middleware and a set of key applications. Our security assessment of the Android framework indicates that malware penetration to the device is likely to happen, not only at the Linux level but in the application level (Java) and thus exploring methods for protecting the Android framework is essential. A collection of applicable security solutions for mobile devices in general and Android in particular is presented in.

Mobile OS makers are now very much concerned with the security challenges that PCs have been facing down through the years. The increasing number of attacks on mobile platforms along with the increasing usage has led many security vendors and researchers to propose a variety of security solutions for mobile platforms. As a case in point, Symbian and Google have designed their operating systems to run applications only in specialized sandboxes, minimizing the capability of malware to spread.

Additionally, common desktop-security solutions are being downsized to mobile devices. As a case in point, Botha et al. analyzed common desktop security solutions and evaluated their applicability to mobile devices. When porting a security solution, such as antivirus software, the limited availability of resources (power source, CPU and memory) of such devices should be taken into account. In addition, most antivirus detection capabilities depend on the existence of an updated malware signature repository; therefore antivirus users are not protected whenever an attacker releases a previously unencountered malware. Some malware instances may target a specific and relatively small number of mobile devices (e.g., for extracted confidential information or track owner’s location) and will therefore take quite a time until they are discovered.

A robust application signing and certification mechanism, which relies mostly on manual code inspection, was integrated into Symbian’s operating system and was proven highly effective in reducing malware attacks. Apple also requires all applications to pass

a review process that checks amongst other things for malware. However, malware writers can still evade manual code inspection as shown in. Therefore, researchers are currently looking at various alternatives that capture application semantics in order to make smarter security decisions without relying on manual code inspection. Since Android was released, several attempts to statically investigate the code of Android applications were published. Schmidt et al. evaluated a framework for static function call analysis and performed a statistical analysis on the function calls used by native applications. Chaudhuri presented a formal language for describing Android applications and data flow among application’s components. This formal language can be used for statically analyzing Android applications and data flow between applications and comparing those with security specifications defined in the application’s manifest.

In this paper we propose the detection of unknown malware instances on Android-based mobile devices by applying Machine Learning techniques on static features that are extracted from Android’s application files. The information includes requested permissions, framework methods called up by the application, framework classes used by the application, user interface widgets, etc. The primary goal of the study is to find the optimal mix of: a classification method, feature selection method and the number of extracted features that yields the best performance in accurately detecting new malware on Android. While most of previous studies extracted features that are based on byte sequence n-grams, in this study we evaluate the use of meaningful features from the Android application files such as the requested permissions, framework methods and classes used by the application.

Two facts support the applicability of such method for detecting malicious code on Android: 1) the proposed features can be extracted from any given Android application; and 2) the vast majority of the proposed features cannot be effectively obfuscated by Android applications. This is due to the side-effects of the installation process. In particular, information from the application manifest is only used in the installation process, leaving the application unable to alter it after the installation.

The rest of the paper is organized as follows. We start in section 2 with a survey of previous relevant studies. Section 3 describes the methods we used. Next, in sections 4 we present the evaluation and the evaluation results. Finally section 5 discusses the results and future work.

1.3 RELATED WORKS

Machine Learning (ML) classification algorithms were employed to automate and extend the idea of heuristic-based detection methods to enable better detection of unknown malware. In these methods, the application file is represented by static features that are extracted from the file (e.g., byte sequence n-grams or PE header features) and classifiers are applied to learn patterns in the code in order to classify new (unknown) files. Recent studies have shown that these methods yield very accurate classification results. Applying static analysis for detecting malicious software consumes much less resources and time, compared to dynamic analysis in which features are extracted from the system while the application is running, and does not require to execute the application for the detection.

Over the past nine years, several studies have focused on the detection of unknown malware on personal computers based on its binary code content. The authors of [16] were the first to introduce the idea of applying ML methods for the detection of different malware based on their respective binary codes. Three different types of features were tested: program header (Portable Executable section), string features (meaningful plaintext strings that are encoded in programs files such as “windows”, “kernel”, “reloc” etc.) and byte sequence features, to which they applied four classifiers: a signature-based method (antivirus), Ripper – a rule-based learner, Naïve Bayes, and Multi-Naïve Bayes. The study found that all of the ML methods were more accurate than the signature-based algorithm. Abou-Assaleh et al. introduced a framework that uses the Common N-Gram (CNG) method and the k-nearest neighbor (KNN) classifier for the detection of malware. For each malicious and benign class a representative profile was constructed. A new executable file was compared with the profiles of malicious and benign classes, and was assigned to the most similar. Kotler and Maloof used a collection of 1971 benign and 1651 malicious executables files. N-grams were extracted and 500 features were selected using the Information Gain measure. The vector of n-gram features was binary, presenting the presence or absence of a feature in the file. In their experiment, they trained several classifiers: IBK (KNN), a similarity-based classifier called the TFIDF classifier, Naïve Bayes, SVM, and Decision Tree (J48). The last three of these were also boosted.

In the experiments, the four best-performing classifiers were Boosted J48, SVM, Boosted SVM and IBK.

Recently, Moskovitch et al. published the results of a study that used a collection of more than 30,000 files, in which the files were represented by byte sequence n-grams. In other studies the representation of executable files was done by OpCode sequence n-gram expressions, through streamlining an executable into a set of OpCodes.

1.4 DETECTION METHOD

The overall process of classifying unknown files using ML methods is divided into two subsequent phases: training and testing. In the first phase, a labeled set of .apk files (the training-set) is provided to the system. Each file is then parsed, and a vector representing each file is extracted based on a predetermined vocabulary. The representative vectors of the files in the training-set and real (known) labels are the input for a learning algorithm. By processing these vectors, the learning algorithm generates a classification model.

Next, during the testing phase, a test-set collection of .apks that did not appear in the training-set, are classified by that classifier. Each file in the test-set is first parsed and the representative vector is extracted (based on the same vocabulary used in the training phase). Based on this vector, the classifier will attempt to predict the class of the file. The performance of the classifier is evaluated by extracting standard accuracy measures that compare the prediction with the real label of the file. Thus, it is necessary to know the real class of the .apk files in the test-set.

Since no malicious applications are yet available for Android, we evaluated the proposed method ability to differentiate between game and tool application files. Successful differentiation between games and tools is expected to provide positive indication about the ability of such methods to learn and model an Android benign application files and potentially detect malware files.

The goal of our work was to explore methods of using data mining techniques in order to create accurate detectors for new (unseen) Android .apk files. We evaluated the following classifiers: Decision Tree (DT), Naïve Bayes (NB), Bayesian Networks (BN),

PART, Boosted Bayesian Networks (BBN), Boosted Decision Tree (BDT), Random Forest (RF), Voting Feature Intervals (VFI).

We used the filters approach for feature selection, due to its fast execution time and its generalization ability. Under a filter approach, an objective function evaluates features by their information content and estimates their expected contribution to the classification task. We evaluated three feature selection measures: Chi-Square (CS), Fisher Score (FS) and Information Gain (IG). These feature selection methods, using a specific metric, compute and return a score for each feature individually.

A problem was raised when we had to decide how many features we would choose for the classification task from the feature selection algorithms' output ranked lists. In order to avoid any bias by selecting an arbitrary number of features, we used, for each feature selection algorithm, six different configurations: 50, 100, 200, 300, 500 and 800 features that were ranked the highest out of all the featured ranked by the feature selection algorithms.

1.5 EVALUATION

1.5.1 Research Questions

We evaluated the capability of the proposed method to classify Android .apk files through a set of experiments, aimed at answering the following questions:

1. Is it possible to detect unknown instances of known Android application types?

2. Which classifier is most accurate in detecting malware on Android devices: DT, NB, BN, BBN, BDT, PART, RF or VFI?

3. Which top-selection (number of extracted features) and feature selection method yield the most accurate detection results: 50, 100, 200, 300, 500 or 800 top features selected using CS, FS or IG?

4. What are the specific features that yield maximal detection accuracy?

In order to perform the comparison between the various detection algorithms and feature selection schemes, we employed the following standard metrics: the True Positive Rate (TPR) measure, which is the proportion of positive instances (i.e., feature vectors of

games) classified correctly; False Positive Rate (FPR), which is the proportion of negative instances (i.e., feature vectors of tools) misclassified; and the total Accuracy, which measures the proportion of absolutely correctly classified instances, either positive or negative. Additionally, we used the Receiver Operating Characteristic (ROC) curve and calculated the Area-Under-Curve (AUC). The ROC curve is a graphical representation of the trade-off between the TPR and FPR for every possible detection cut-off.

1.5.2 Creating the Dataset for the Experiments

Since no standard dataset was available for this study, we had to gather our own dataset. Our evaluation focuses on classifying two types of Android applications: tools and games. We collected free .apk files available on the Android Market (a Google-proprietary service which allows browsing and downloading of applications that were published by different developers). A total of 2,285 .apk files were collected. We categorized the applications according to the Market classification which results in 407 games and 1878 tools. More than 22,000 features were initially extracted from the file collection using an .apk feature extractor which we developed specifically for this task. Extracted features included:apk features, XML features, dex features.

.....

1.5.3 Experiments and Results

In the experiment we wanted to answer the four research questions presented in section IV. According to these questions, we wanted to identify the best settings of the classification framework that is determined by a combination of: (1) the top-selection of features (50, 100,200, 300, 500 or 800); (2) the feature selection method (CS, FS or IG); and (3) the classifier (DT, NB, BN, BBN, BDT, PART, RF or VFI) to distinguish between games and tools applications. The evaluation performed in a 10-fold cross validation format repeated 5 times for all the combinations of the optional settings. Note that the files in the test-set were not included in the training set presenting unknown files to the classifiers.

Figure 1 depicts the accuracy and FPR for each classification algorithm (averaged over all iterations, folds and feature selection algorithms). The results show that the two

boosted algorithms (BDT and BBN) performed better than all other classifiers, while the Boosted Bayesian Networks outperformed all classifiers.

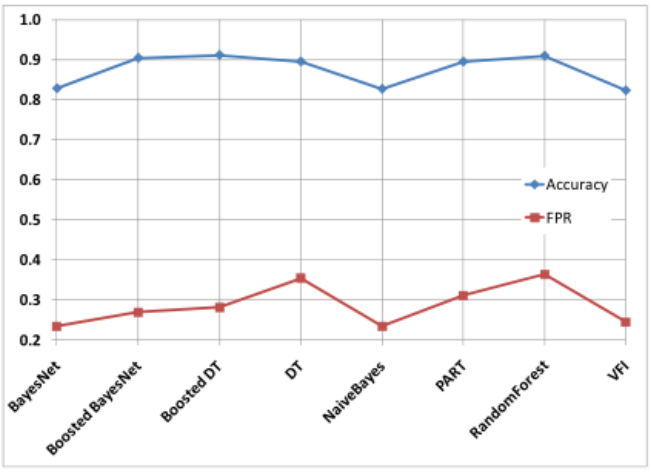

Figure A–1 Mean accuracy and FPR of each classifier

Table I depicts the accuracy, FPR, TPR and AUC for each combination of feature selection and the number of selected top features (averaged over all iterations, folds and classifiers). The results show that all three feature selection algorithms yielded similar results, where the performance was better with large number of features (i.e., 500 and 800 top features).

Table A–1

		ChiSquare	FisherScore	InfoGain
Accuracy	**50**	0.839	0.847	0.856
	100	0.861	0.868	0.871
	200	0.877	0.879	0.880
	300	0.881	0.882	0.882
	500	0.888	0.885	0.890
	800	**0.891**	0.884	**0.890**
FPR	**50**	0.336	0.349	0.308
	100	0.303	0.293	0.288
	200	0.283	0.270	0.279
	300	0.283	0.268	0.269
	500	0.283	0.269	0.279
	800	**0.254**	0.261	**0.260**

Table II depicts the averaged accuracy, FPR and AUC for the top two combinations of feature selection and the number of selected top features. Boosted BN trained on the

top 800 features, selected using Chi-Square and Information Gain outperformed all other settings.

Table A–2

Configuration	Accuracy	FPR	AUC
Boosted BN ChiSquare 800	0.922	0.190	0.945
Boosted BN InfoGain 800	0.918	0.172	0.945

1.6 DISCUSSION AND CONCLUSIONS

In this paper we used features extracted from Android application (.apk) files. The extracted data is used as features during a classification process of the applications. Since no malicious applications are yet available for Android, our evaluation focused on classifying two types of applications: tools and games. We performed an evaluation using a collection comprising 2,850 games and tools. The results show that the combination of Boosted Bayesian Networks and the top 800 features selected using Information Gain yield an accuracy level of 0.918 with a0.172 FPR.

Features, extracted statically from .apk files, coupled with Machine Learning classification can provide good indication about the nature of an .apk file without running it, and may assist in the detection of malicious applications. The most important features for the detection are extracted using our .dex file parser that can transform contents of the .dex file into standard features (e.g., strings, types, classes, methods, fields, static values, inheritance, opcodes). A more advanced usage of such tool is to merge two .dex files. This may be used in order to inject malicious payload into a benign .apk, thereby creating a Trojan horse. Consequently, an attacker can perform a Man-In-The-Middle attack by injecting malicious code while a user, connected to an un-trusted wireless network, is downloading an .apk file. We plan to further investigate the applicability of such attack.

通过自动化静态代码分析和机器学习对Android应用分类

1.1 摘要

本文我们通过分析从Android应用安装包中提取中的静态特征，应用机器学习技术实现了Android应用分为工具和游戏两类。我们的测试主要集中在将Android应用分为工具和游戏两类。工具和游戏两类应用的成功区分向我们传递了一个积极的信号。这表明我们如果以同样的方法对良性Android应用进行学习和建模以发现潜在的恶意Android应用具有一定的可行性。我们总共对2285个Android安卓包进行了测试。结果表明，从APK文件中提取的静态特征配合机器学习算法可以在不需要运行应用的情况下很好地检测出应用的内在特征。因此，这种技术很可能可以用来帮助检测恶意软件。

关键词：移动设备，机器学习，恶意软件，安全，安卓，静态分析

1.2 引言

智能手机已经从简单的移动设备演变成复杂紧凑的小型计算机。作为开放、可编程的网络设备，智能手机非常容易受到病毒、特洛伊木马和蠕虫等恶意软件的威胁。利用智能手机用户可以浏览互联网，收发电子邮件和短信，与其他设备交换信息，使用各种应用程序，然而正是这些操作使设备成为了潜在的攻击目标。

被感染的智能手机可能会同时给用户和运营商带来严重的损失。恶意软件可以让手机部分或全部的功能失效；造成不必要的计费；窃取用户隐私或感染用户的电话簿中的所有联系人。用来攻击智能手机的媒介主要包括:蜂窝网络，互联网连接(通过WFI、GPRS / EDGE或3G网络)、USB和其他外围设备。

近几年最常见的智能手机操作系统是Google的Android系统。Android智能移动设备是一个全面的软件框架，它包括一个操作系统，中间件和一组关键应用程序。我们对Android框架[5]的安全评估表明，恶意软件渗透到设备的事情不仅可能发生在Linux层，也可能发生在应用程序层(Java)，因此探索保护Android框架

的方法是至关重要的。文献中就提出了一揽子通用的移动设备安全解决方案和为Android定制的安全解决方案。

多年以来，PC端遭受到了大量的安全威胁。如今，手机系统厂商也开始担心手机系统将面临的安全挑战。伴随着移动设备的普及，针对移动平台的安全攻击越来越多。因此许多安全厂商和研究者提出了各种的移动平台安全解决方案。例如，Symbian和Google设计他们的操作系统中，应用只能在特定的沙箱中运行，以最大限度的减小恶意软件传播的能力。此外,常用的桌面安全解决方案也被移植到移动设备。例如，Botha等人分析了常见的桌面安全解决方案评估了其在移动设备的适用性。当移植一个安全解决方案到移动设备时，如杀毒软件，我们应该考虑到移动设备资源(电源、CPU和内存)的有限性。除此之外，大部分杀毒软件的能力取决于已知的恶意软件特征库；因此当攻击者放出一个新的恶意软件时，杀毒软件并不能保护使用者。还有一些恶意软件可能针对特定的、相对少量的移动设备(如，获取机密信息或记录设备所有人的位置)，因此会持续相当长的时间，直到被发现。

Symbian操作系统中集成了一个健壮的应用程序签名和认证机制。虽然此机制主要依赖手工代码检查，但事实证明它有效地减少了恶意软件的攻击。苹果还要求所有应用程序通过严格的审查程序。然而，恶意软件作者仍然可以逃避手工代码检查。因此，为了在摆脱手工代码检查，做出更明智的安全决策，研究人员正在寻找各种捕获应用程序语义的替代品。自从Android系统发布以来，几款静态分析Android代码的工具相继被开发出来。Schmidt等人[13]对一个Android代码静态分析框架进行了评估，并对本地应用的函数调用数据进行了分析。Chaudhuri提出了一个正式的语言用来描述Android应用程序和应用程序组件之间的数据流。该语言可以用来静态分析Android应用程序和应用程序之间的数据流并将其和Manifest文件中定义的安全规范做比对。

本文中，我们提出了一种Android平台上检测未知恶意软件方法，即用机器学习的方法对从Android应用中提取出来的静态特征进行学习和建模。Android应用程序是以.apk结尾的归档文件，应用程序相关的代码和资源文件即被压缩在此文件中。这些信息主要包括应用所请求的权限、调用的方法和类、用户交互界面等。此项研究的主要目标是找出分类算法、特征选取方法和特征数量的最佳组合，以尽可能的提高检测未知恶意软件的准确率。以往大多数的研究都是从字节序列中提取特征，而本文，我们试着使用Android应用中一些有意义的特征，例如，应用的请求权限、调用方法和类。

以下两个事实支持了用此种方法检测恶意代码的可行性：1）上文中所提到的特征均可以从Android的安装文件中提取；2）上文中所提到的大部分特征不会因为代码被混淆而被隐藏。尤其的，Android应用的列表文件中的内容只在安装过程中使用，程序被安装后无法自主更改列表文件的内容。

本文的结构如下：在第二章，我们对前人的相关工作进行了一些调查。第三章，我们描述了我们的算法。然后，第四章我们进行了测试，并展示了测试结果。最后，第五章，我们进行了总结并讨论了未来的工作。

1.3 相关工作

通过机器学习方法来实现自动化，通过扩展启发式的检测方法来达到检测未知恶意软件的目的。在这些方法中，我们用一些从程序中提取出来的静态特征来代表该程序。（例如，字节序列或者PE头特征）。分类器对代码特征进行学习和建模以对未知的程序文件进行分类。近期的研究表明，用这些方法进行分类可以达到非常高的准确率。相比于在程序运行时从系统中获取特征，静态分析有很大的优势。一方面，静态分析耗费更少的资源和时间；另一方面，静态分析不需要运行相应程序。

在过去的九年里，有一些研究主要集中在个人电脑平台上基于二进制代码的未知恶意软件检测。文献[16]首次提出将机器学习的方法应用到基于二进制特征码的恶意软件检测上。他们从文件中提取三种不同类型的特征：程序头（Portable Executale section）,字符串特征(程序文件中会编码一些有意义的明文字符串，例如:”Window”,”kernel”,”reloc”等。)，字节序列特征，采用了四种分类算法：基于指纹的算法（反病毒），Ripper（基于规则的），本地贝叶斯，多重贝叶斯。该研究表明，所有的机器学习中的分类算法都比基于指纹的算法准确率高。Abou-Assaleh等人介绍了一套用Common NGram (CNG)分类算法和k-nearest neighbor (KNN)分类算法进行恶意软件检测的框架。首先为每个恶意和良性的软件创建一个代表性的描述，当有新文件来临时，通过和已知类型的所有描述信息进行比对，最后将其分派给最相似的那一类。Kotler 和Maloof使用了一个包含1971个良性应用和1651个恶意应用的数据集。利用信息提取方法从文件中提取N元组合和500个特征。N元向量用二进制表示，N元组中的每个元素代表一个特征是有或者无。在他们的实验中，他们分别用IBK（KNN）、TFIDF、Naïve

Bayes、SVM和Decsion Tree（J48）进行了建模。后三种算法也进行了相应的优化。本次实验中，效果最好的四种算法是改进J48、SVM、改进SVM和IBK。

最近，Moskovitch等人公布了一项研究结果，该研究使用了超过30000个文件样本，从每个样本中提取一个基于字节序列的N元组特征。在其它研究中，样本的特征向量多是从指令序列中提取。

1.4 检测算法

算法分为训练和测试两个阶段。在第一阶段，我们会向系统提供一组已知分类的apk文件。根据我们预先定义好的特征表从每个文件中提取特征向量。然后把从训练集提取的特征向量和对应的分类作为学习算法的输入。通过处理和分析这些数据，学习算法会生成一个对应的分类模型。

接下来，在测试阶段，分类器会对一系列未知分类并且没有出现在训练集中的apk文件进行分类。测试集中的每一个文件都是第一次被分析，利用同样的规则，从每一个文件中提取特征向量。分类器会基于这些特征向量对测试集中的文件分类。我们通过分类器对测试文件的分类结果和文件的真实结果进行对比来评估分类器的性能。

由于目前还没有可用的Android平台恶意软件。我们通过用此方法区分出游戏和工具两类来证明此方法的可行性。工具和游戏两类应用的成功区分向我们传递了一个积极的信号。这表明我们如果以同样的方法对良性Android应用进行学习和建模以发现潜在的恶意Android应用具有一定的可行性。

此研究的目标是希望能够用数据挖掘的技术探索出能够精确检测未知Android文件的方法。我们对以下分类器的性能进行了评估：Decision Tree (DT), Naïve Bayes(NB), Bayesian Networks (BN), PART, Boosted Bayesian Networks (BBN), Boosted Decision Tree (BDT), Random Forest (RF), Voting Feature Intervals (VFI)。

我们使用的过滤器进行特征选择，这种方法执行时间短和泛化能力强。在过滤器方法中，目标函数会对信息内容特征进行评估，并估算它们在分类任务中的预期贡献。我们评估了三个特征选择的措施：卡方检验（CS），费舍尔分数（FS）和信息增益（IG）。这些特征选择方法，使用特定的指标，为每个特征计算并返回一个分值。

一个新的问题产生了，我们选取多少个特征来完成这个分类任务呢？为了避免随意的选择会引起过大的偏差，我们对于每个特征选择算法，使用六种不同的配置，分别选取分数排名最高的：50，100，200，300，500和800个特征。

1.5 仿真与验证

1.5.1 问题的提出

我们通过一组实验评估了文中提出的对Android进行分类的方法，旨在回答以下问题：

1.是否可以检测出未知Android应用程序的类型？

2.在检测Android设备上的恶意软件的问题上，哪些分类算法是最准确的，：DT，NB，BN，BBN, BDT, PART, RF 还是 VFI？

3.使用多少特征和哪些特征选择方法能得到最准确的检测结果：50，100，200，300，500还是800？CS, FS 还是 IG??

4.能够使检测结果最准确的具体特征是哪些？

为了对比各种检测算法和特征选择算法的性能，我们采用以下标准指标：真阳性率（TPR），假阳性率（FPR）和平均准确率来衡量。此外，我们使用受试者工作特征（ROC）曲线和曲线下面积（AUC）。ROC曲线是的TPR和FPR的一种折中和图形表示。

1.5.2 数据集的创建

由于没有可用于本研究的标准数据集，我们不得不自己收集数据。我们的评估侧重于将APK分为两种类型：工具和游戏。我们从Android Market（谷歌的专有服务，可以浏览和下载由不同开发者发布的应用程序）上总共收集了2285个免费apk文件。2285个样本中总共有407个游戏类和1878个工具类。为了这项研究，我们专门开发了APK特征提取模块。利用该模块我们从apk文件特征提取了超过22,000个初步特征以供后期筛选。提取的特征来自于apk、xml和dex文件。

……

1.5.3 实验与结果

在实验中，我们想回答第四节提出的四个研究问题。根据这些问题，我们要确定分类框架的最佳组合：（1）特征数量（50，100,200，300，500或800）；（2）特征选择算法（CS，FS还是IG）；（3）分类算法（DT，NB，BN，BBN，PART，RF 还是 VFI）。实验中，我们采用k=10的交叉验证对所以组合重复进行5次评估。另外，测试集并不包括在训练集中。

图A–1描述的是各分类算法的准确性和FPR（所以实验的平均值）。结果表明，两种改进算法（BDT和BBN）的表现比其他所有的分类器更好，而改进贝叶斯网络算法优于所有分类算法。

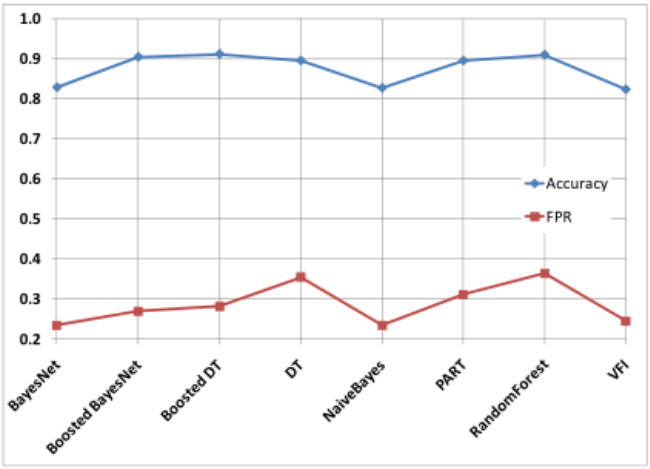

图 A–1 各分类算法的平均准确率和FPR

表A–1描述的是各特征选择算法和特征数量组合的准确性，FPR，TPR和AUC（所有实验的平均值）。结果表明，三个特征选择算法的表现类似，特征数量越多性能越好（即，500和800及以上数量的特征）。

表A–2描述了性能表现最佳的方案的平均准确度，FPR和AUC。改进BN算法加上由卡方检验和信息增益算法筛选出的前800个特征，分类效果优于其它组合。

1.6 结论与分析

在本文中，我们利用从Android应用程序（apk文件）文件中提取的特征对应用程序进行了分类处理。由于没有可用于测试的Android恶意应用，我们的测试

表 A–1 各特征选择算法和特征数量组合的平均准确性，FPR，TPR和AUC

		ChiSquare	FisherScore	InfoGain
Accuracy	**50**	0.839	0.847	0.856
	100	0.861	0.868	0.871
	200	0.877	0.879	0.880
	300	0.881	0.882	0.882
	500	0.888	0.885	0.890
	800	**0.891**	0.884	**0.890**
FPR	**50**	0.336	0.349	0.308
	100	0.303	0.293	0.288
	200	0.283	0.270	0.279
	300	0.283	0.268	0.269
	500	0.283	0.269	0.279
	800	**0.254**	0.261	**0.260**

表 A–2 最佳配置的平均准确度，FPR和AUC

Configuration	Accuracy	FPR	AUC
Boosted BN ChiSquare 800	0.922	0.190	0.945
Boosted BN InfoGain 800	0.918	0.172	0.945

集中将APK分为以下两种类型：工具和游戏。我们总共使用了2850个样本进行实验。结果表明，改进贝叶斯网络和由信息增益算法筛选出的前800个特征的组合的功能效果最好，准确度达0.918，FPR仅为0.172。 从APK中提取的静态特征，加上机器学习分类算法可以在不必运行APK的情况下很好的检测出apk文件的内在性质，也可以协助恶意程序的检测。系统中的一个重要模块是dex解析器。它将dex文件的内容转换成标准的特征（例如，字符串，类型，类，方法，字段，静态值，继承，操作码）。这些特征在分类系统中起着至关重要的作用。dex解析器的一个高级功能是可以将两个dex文件合并成一个。这意味着可以将恶意代码注入到正常的APK中，从而创建一个木马。当用户在不可信网络中下载AKP时，攻击者就可以利用中间人攻击的方式将正常APK替换为木马。 下一步的工作中，我们将对这种攻击做深入的调查研究。