

# 硕士学位论文

## 基于对象引用关系图的 Android 恶意代码检测的研究

### RESEARCH ON ANDROID MALWARE DETECTION BASED ON OBJECT REFERENCE GRAPH

陆亮

哈尔滨工业大学

2013 年 6 月

国内图书分类号: TP315  
国际图书分类号: 004.4

学校代码: 10213  
密级: 公开

## 工学硕士学位论文

# 基于对象引用关系图的 Android 恶意代码检测的研究

硕 士 研 究 生: 陆亮

导 师: 李东

申 请 学 位: 工学硕士

学 科: 计算机科学与技术

所 在 单 位: 计算机科学与技术学院

答 辩 日 期: 2013 年 6 月

授予学位单位: 哈尔滨工业大学

Classified Index: TP315

U.D.C: 004.4

Dissertation for the Master Degree in Science

**RESEARCH ON ANDROID MALWARE  
DETECTION BASED ON OBJECT REFERENCE  
GRAPH**

<b>Candidate:</b>	Lu Liang
<b>Supervisor:</b>	Prof Li Dong
<b>Academic Degree Applied for:</b>	Master of Engineering
<b>Speciality:</b>	Computer science and Technology
<b>Affiliation:</b>	School of Computer Science and Technology
<b>Date of Defence:</b>	June, 2013
<b>Degree-Conferring-Institution:</b>	Harbin Institute of Technology

## 摘 要

如今,智能手机行业飞速发展,尤其是 Android 智能手机更是得到了广泛普及,成为人们重要的交流工具。但是 Android 智能手机上恶意软件泛滥,给用户的信息安全带来巨大的威胁。软件检测技术是实现恶意代码检测的一种途径。已有的静态检测技术局限性较大。如何使用动态特征来实现软件检测成为了人们越来越关注的问题。本文对软件运行时内存的特征进行了一定的研究。

本文的主要研究目的是利用对象引用关系图为动态特征,实现对 Android 恶意代码的检测。本文主要的研究内容是在 Android 平台下抽取内存信息建立对象引用关系图,并研究通过子图同构匹配算法实现对象引用关系图的匹配。

首先,本介绍了 Android 系统的整体架构,分析了其安全机制和应用程序运行机制。并对 Android 平台的恶意代码进行了分类,分析了其行为特点和攻击原理。抽取堆内存中的对象引用关系是本文工作的难点,本文实现了在 Android 平台下对单个进程内存数据的抽取,并从数据中获取了对象间的引用关系。

其次,本文分析了目前广泛应用的经典图同构匹配算法,重点研究了适用于子图同构匹配的 VF2 算法,并在 VF2 算法的基础上进行修改,加入了新的参数来控制匹配的精度,使其适用于对象引用关系图的同构匹配,并详细分析了算法的时间复杂度和空间复杂度。

最后,本文搭建了 Android 恶意代码的检测系统,对该方法在恶意代码检测中的有效性进行了测试。实验表明,对象引用关系图检测方法对程序通过代码混淆方法改变自身静态特征之后的识别依然有效。通过调整修改后 VF2 算法中控制匹配精度的参数,实验测试出了该算法在不同匹配精度下检测的漏报率和误报率,验证了算法在实际应用中的可行性。

**关键词:** Android; 恶意代码; 动态检测; 对象引用关系图; VF2

## Abstract

Today the smart phone industry is developing rapidly. Especially the Android smart phone is widely used and is becoming an important communication tool. But there are too many Android malwares. It brings users great threat on information security. In essence, the key to solve the problem is software recognition technology. Static identification technology has many disadvantages. People are becoming more and more concerning about the dynamic methods to identify software. This paper makes some research on software memory feature.

The main purpose of this paper is to use object reference graph to detect Android malware. The main content is getting memory data to make object reference graph and using Subgraph Isomorphism algorithm to match object reference graphs.

Firstly, this paper describes the overall architecture of the Android system, analyzes its security and application software. This paper also classifies android malware and analyzes their behavioral characteristics and attack principle. Getting the references among objects in memory is a difficulty. This paper implements extracting memory data for a process under the Android platform. Objects and their references can be made from the data.

Secondly, this paper analyzes the widely used classical graph Isomorphism matching algorithm, especially the VF2. Some changes are made on VF2 algorithm. A new parameter is added to control the accuracy of matching which makes it suitable for object reference graphs. The time complexity and space complexity of VF2 algorithm are analyzed.

Finally, a detecting system for Android malware is built, and the effectiveness of the method for detecting malware is tested. Experimental results show that after malwares change their static characteristics via code obfuscation, the method still works. In experiment, it shows that false positive rate and false negative rate are changing with different accuracy. Thus, the feasibility of the algorithm in practice are verified.

**Keywords:** Android, Malware, Dynamic Detecting, Object Reference Graph, VF2

# 目 录

基于对象引用关系图的 Android .....	II
恶意代码检测的研究 .....	II
<b>RESEARCH ON ANDROID MALWARE DETECTION BASED ON OBJECT REFERENCE GRAPH .....</b>	<b>III</b>
摘 要 .....	I
Abstract.....	II
目录 .....	III
第 1 章 绪论 .....	1
1.1 课题背景 .....	1
1.2 国内外研究现状 .....	3
1.3 主要研究内容及意义 .....	6
1.3.1 对象引用关系图的概念 .....	6
1.3.2 主要研究内容 .....	8
1.3.3 研究意义 .....	9
1.4 本文的组织结构 .....	9
第 2 章 Android 恶意代码分析及数据抽取实现 .....	11
2.1 Android 系统平台分析 .....	11
2.1.1 Android 系统的内核结构 .....	11
2.1.2 Android 应用组件 .....	12
2.2 Android 恶意代码分析 .....	15
2.3 Android 平台下堆内存数据抽取的实现 .....	17
2.3.1 Android 进程的堆内存信息文件的获取 .....	18
2.3.2 内存信息文件格式转换的实现 .....	19
2.3.3 对象及引用关系数据的抽取 .....	22
2.4 本章小结 .....	26
第 3 章 图及其匹配算法的研究与分析 .....	27
3.1 图的同构模式 .....	27
3.1.1 图的基本概念 .....	27
3.1.2 Isomorphism 同构 .....	28
3.1.3 Isomorphism 子图同构 .....	30
3.1.4 Monomorphism 同构 .....	30
3.2 图同构算法 .....	31

3.2.1 图同构匹配算法分析 .....	31
3.2.2 图同构匹配算法性能对比 .....	32
3.3 VF2 算法实现与分析 .....	33
3.3.1 基于状态表示的 VF2 算法的策略 .....	33
3.3.2 引入精确度控制参数的 VF2 算法的实现 .....	37
3.3.3 性能分析 .....	38
3.4 本章小结 .....	40
第 4 章 检测系统与结果分析 .....	41
4.1 检测系统设计与实现 .....	41
4.1.1 Android 端设计与实现 .....	41
4.1.2 服务器端设计与实现 .....	42
4.2 测试环境 .....	47
4.2.1 Android 端运行环境 .....	47
4.2.2 PC 端运行环境 .....	47
4.3 数据来源与整理 .....	47
4.3.1 模拟恶意代码样本 .....	47
4.3.2 真实恶意代码样本 .....	48
4.4 模拟样本实验结果 .....	49
4.4.1 模拟样本检测结果与分析 .....	49
4.4.2 模拟样本的代码混淆变种检测 .....	50
4.5 真实样本测试结果 .....	51
4.5.1 VF2 算法在恶意代码检测中的效果 .....	51
4.5.2 $\lambda$ -VF2 算法在不同精确度下效果对比测试 .....	52
4.6 本章小结 .....	54
结 论 .....	55
参考文献 .....	56
攻读学位期间发表的学术论文 .....	60
哈尔滨工业大学学位论文原创性声明和使用权限 .....	61
致 谢 .....	62

# 第 1 章 绪论

## 1.1 课题背景

伴随着移动通信技术的高速发展，手机等移动终端给人们的生活带来了极大的方便，已成为人们日常生活中不可缺少的交流工具。2009 年，工信部就颁发了首批 3G 牌照，成为我国的 3G 元年。虽然和国际主流市场相比发牌时间较晚，但受到智能手机快速普及的影响，智能手机产业链不断成熟的推动，经过短短 3 年多的发展，我国 3G 用户数量增长十分迅速。仅 2012 年 3G 用户就增加了 9206.2 万户，用户总量达到了 2.20 亿户，3G 用户增长率由上一年年末的 13% 提升到 20%。据工信部表示，2013 年我国新增的 3G 用户数量将突破 1 亿户，预计年内我国将替代美国成为全球第一大 3G 用户国家<sup>[1]</sup>。

Android 智能手机操作系统是以 Linux 内核为基础的开源操作系统，主要用于便携的移动设备。Android 系统首先被应用于手机。2005 年，Google 联合多家手机制造商，对 Android 系统进行开发、改良，进而将其扩展到平板设备以及众多其他领域。2011 年第一季度，Android 系统在全球市场的占有率超过塞班系统，成为全球第一。2012 年 2 月，Android 的市场份额在全球智能手机中已经达到 52.5%，在中国市场份额的占有率是 68.4%。作为一个开放移动设备的系统平台，Android 近几年发展十分迅速，市场占有率更是不断提升，Android 系统也逐渐发展成为主流手机操作系统之一。表 1-1 来自美国市场研究公司 IDC（International Data Corporation）发布的 2013 年第一季度调查报告。从表中可以看出，Android 与 iOS 移动设备的市场份额仍然在继续扩大，两家的市场份额总和为 92.3%。排名第三的则是 Windows Phone，黑莓系统紧随其后。2013 年第一季度，Android 智能手机的出货量达到了 1.621 亿台，占据了智能手机市场 75% 的份额，增长速度惊人。

Android 智能手机广泛普及的背后带来的是严重的手机安全问题，3G 网络环境下针对 Android 平台的恶意攻击问题已经突显，并且 3G 技术和手机智能化的紧密结合更加剧了日趋恶化的移动终端安全形势。由于 Android 应用市场不规范，管理系统不完善以及各种弊端等等，众多黑客将其作为攻击的主要目标。Android 应用软件粗糙烂制，各种恶意软件、垃圾软件充斥在互联网上，给使用者带来了极大的不便与危害。

Android 智能手机广泛普及的背后带来的是严重的手机安全问题，3G 网络环境下针对 Android 平台的恶意攻击问题已经突显，并且 3G 技术和手机智能化的紧密结合更加剧了日趋恶化的移动终端安全形势。由于 Android 应用市场不规范，



管理系统不完善以及各种弊端等等,众多黑客将其作为了攻击的主要目标。Android 应用软件粗糙烂制,各种恶意软件、垃圾软件充斥在互联网上,给使用者带来了极大的不便与危害。

表 1-1 2012 年 1 季度与 2013 年 1 季度智能手机市场对比表

操作系统	13 年 1 季度 出货量(百万)	13 年 1 季度 市场份额	12 年 1 季度 出货量(百万)	12 年 1 季度 市场份额	同比增长
Android	162.1	75.0%	90.3	59.1%	79.5%
ios	37.4	17.3%	35.1	23.0%	6.6%
Win Phone	7.0	3.2%	3.0	2.0%	133.3%
BlackBerry	6.3	2.9%	9.7	6.4%	-35.1%
Linux	2.1	1.0%	3.6	2.4%	-41.7%
Symbian	1.2	0.6%	10.4	6.8%	-88.5%
Others	0.1	0.0%	0.6	0.4%	-83.3%
Total	216.2	100.0%	152.7	100%	41.6%

Android 智能手机广泛普及的背后带来的是严重的手机安全问题,3G 网络环境下针对 Android 平台的恶意攻击问题已经突显,并且 3G 技术和手机智能化的紧密结合更加剧了日趋恶化的移动终端安全形势。由于 Android 应用市场不规范,管理系统不完善以及各种弊端等等,众多黑客将其作为了攻击的主要目标。Android 应用软件粗糙烂制,各种恶意软件、垃圾软件充斥在互联网上,给使用者带来了极大的不便与危害。

恶意软件是具有窃取信息、破坏系统等恶意行为软件的统称,包括病毒、木马、蠕虫等<sup>[2]</sup>。恶意软件给用户带来的安全隐患是极大的,用户在使用智能手机时面临着各种安全问题。针对 Android 平台的恶意软件行为各异,有的是盗取数据,骗取流网络流量;有的是通过 GPS 获得用户的位置信息达到追踪用户的目的;还有的是嵌入内部广告骗取点击量或者安装隐蔽的拨号软件来悄悄增加保险费率;此外还有窃取银行信息造成财产损失。因此,我们在享受 Android 平台带来的快捷与方便的同时,必须采取防范措施解决随之而来的安全问题。

将程序动态特征应用于 Android 恶意代码的检测是一种新的思路。这在技术上和代码抄袭检测是一致的,因为本质上都是对已知代码的识别。随着互联网的快速发展,代码抄袭问题也日益严重。对于许多 IT 公司,软件的源代码是公司的核心知识产权,一旦遭受抄袭就会给公司的利益带来巨大的损失。例如,在 2005 年,经州法院判决,IBM 因其某些程序使用了 Compuware 公司的代码,向 Compuware 支付了 1.4 亿美元的许可费和 2.6 亿美元的服务费<sup>[3]</sup>。为了解决代码抄袭的问题,

各种各样的软件保护技术被提出。数字水印技术是最早被使用的。它通过向软件中加入水印来证明代码的所有权<sup>[4]</sup>。缺点是水印易受攻击<sup>[5]</sup>并且加入水印给发开者带来了额外的工作。取而代之的是代码混淆技术，它虽然能通过打乱代码使人难以理解代码逻辑<sup>[6]</sup>，然而却不能阻止代码被直接抄袭。一个较为新颖的方法是软件胎记技术。软件胎记技术不需要向软件内添加新的代码，它完全依靠软件的固有特性来识别软件。通过实验表明，即使使用代码混淆技术将软件内的水印破坏掉，软件胎记技术依然能够识别出软件<sup>[7]</sup>。软件胎记分为静态软件胎记和动态软件胎记<sup>[8]</sup>。静态胎记是根据软件内代码的句法结构来抽取<sup>[9]</sup>，而动态胎记是根据软件的运行时特点来抽取<sup>[10]</sup>。一般情况下，破坏软件胎记的方法是通过代码混淆来打乱软件的句法结构。因此，动态胎记比静态胎记具有更好的健壮性。已有的动态胎记技术利用的是控制流的追踪或者 API 函数调用次序。基于控制流的方法面临的问题是无法有效的应对代码混淆的攻击，如打乱循环结构。而基于 API 函数调用次序的方法面临的问题是 API 函数可能不足，无法建立动态胎记。

Android 恶意代码检测和软件抄袭检测都需要使用更为有效的动态特征作为检测依据。本文基于内存中对象引用关系图，对程序运行时的动态特征进行了研究。

## 1.2 国内外研究现状

Android 病毒检测的方法主要分为两种：静态分析方法和动态分析方法。研究人员首先开始关注和研究的是静态的分析方法。相关领域的研究人员对其内容的分析与研究较为成熟，静态分析方法在很多领域也得到了广泛应用。

静态分析方法是指在程序不运行的状态下，对程序的可执行文件进行分析，寻找代码字符序列的特征。

Julia 是一种 Android 平台下 JAVA 的字节码静态分析工具，但是它无法解析 XML 文件映射生成的类。文献[11]通过改进 Julia，使其能够使对 DVM(Dalvik Virtual Machine)字节码进行分析，第一次将静态分析方法应用于 Android 程序。

Indus 是 JAVA 代码的静态分析工具，为了能够使其分析 Android 程序。文献[12]针对窃取隐私类的恶意程序提出了字节码转换工具，使其能够对 DVM 字节码进行转换，这使得 Indus 能够分析 Android 程序。

文献[13]对 DVM 字节码的依赖性进行了深入研究。文章充分利用了现有工具 dex2jar 和 FindBugs 进行分析。DVM 字节码可以通过 dex2jar 转换为 JAVA 字节码。FindBugs 是 JAVA 字节码漏洞静态分析器，作者通过它实现了对 Android 程序控制

流图的遍历, 可以获得 Intent 对象之间功能依赖以及直接依赖的统计数据。

以上三篇文献对 DVM 字节码的分析进行了深入探索, 具有很大的参考价值, 但由于使用的是现有工具, 局限性较大。文献[14]在他们的基础上进行了改进。文章采用了模式分类的方法, 将程序的检测分为了两个步骤: 训练和测试。作者首先设计了 DEX 文件的解析器, DEX 文件是 DVM 可执行文件格式。通过解析器获得训练样本的标准特征, 比如类, 成员, 字符串等等。然后使用各种聚类算法, 如朴素贝叶斯等生成模型。这样在测试阶段, 通过输入测试样本, 使用分类器进行分类。文章中实验部分采用的是普通的 Android 应用程序, 并未使用真实的恶意代码, 因此其有效性还有待验证。

上面四篇文献都是基于 DVM 字节码进行检测分析的, 文献[15]和[16]采用分析 ELF (Executable and Linkable Format) 文件的方法, 分析了文件的信号信息, 从而获得 Android 软件运行时函数的调用列表, 然后再采用朴素贝叶斯等方法进行训练, 建立模型。其中协同入侵在文献[15]中被到, 这是一种较为新颖的方法。由于实验采用的也不是真实的恶意软件程序, 因此其实际的入侵检测效果也有待验证。

文献[17]中作者提出了反汇编的方法, 对 Android 的恶意程序进行反汇编, 分析出恶意代码部分, 对其二进制代码进行修改, 最终实现分离恶意代码的目的。反汇编的方法对于没有经过处理的恶意程序是有效的, 但对于经过代码混淆处理的恶意程序就无法处理了。

文献[18]对窃取隐私类的代码进行了研究。文献以敏感数据的访问为基础, 对访问的行为特征进行了定义, 这些特征的内容是由一系列特定权限构成的。通过分析程序请求的权限特征, 进而与已定义的特征对比, 来判断程序是否是恶意软件。此方法的局限性较大, 首先它不能够检测出利用 Android 漏洞的恶意代码; 其次由于 Android 内部 API 的使用没有权限限制, 因而也无法检测出这类恶意代码。

静态分析方法的缺点十分明显, 其健壮性较弱, 恶意代码通过代码混淆、花指令等防检测技术就可以轻易躲避检测。而动态分析方法的健壮性相对来说就要好很多, 通过分析恶意程序在运行时的动态特征, 达到对入侵软件的检测, 可显著提高检测的有效性。同时, 一些动态特征也可以有效识别一些未知的恶意程序。不同于常规的检测方式。

文献[19]采用了内核级别的监控方式, 对 Android 程序的系统调用和信息进行了记录。文章作者在系统内核中加入了日志记录与分析模块, 可过滤掉特定的事件, 同时记录下特殊信息如 sim 卡号等。该方法对于恶意程序的系统调用的记录

较为有效，作者将其用于信息窃取的类的恶意程序的监控，并没有对其它类恶意程序进行分析，有效性有待进一步研究。

文献[20]在此基础上进行了进一步的研究，将监控划分了三个层次：Android 应用层，系统应用层和系统内核层。思路十分创新，值得深入研究。但是作者没有提供有效的实验测试，可行性还需要验证。

文献[21]中设计了基于异常行为检测的 Crowdroid 系统，该系统是基于异常行为检测的分类器，采用的是轻量级的 C/S 架构。其客户端部署于 Android 平台，使用现有的 Strace 程序监控系统调用，创建记录文件，上传至服务器。服务器端对文件进行解析，建立程序的系统调用特征，进而构造模型，最后利用 K-Means 算法对其进行分类。Crowdroid 系统有两个创新点：一是使用了 C/S 架构；二是采用分布式方式。安装客户端的用户分别上传了各自的监控状态，使得服务器获得的数据十分广泛。由此带来的问题是数据量过大，数据安全的保护较为困难。攻击者容易伪造攻击数据，对结果进行干扰。此外，由于数据较多，系统的网络流量开销也比较大，使用成本也是一个必须解决的问题。

文献[22]提到了一种新的动态分析技术—沙箱技术。这是分析 Android 恶意代码新的发展方向，有很大的研究空间。但是目前沙箱技术还不够成熟，应用不多，同时还有很多问题没有解决。文献[23]构建沙箱系统采用的方法是，通过在内核中加入 LKM (Loadable Kernel Module)，从内核底层分析系统调用，形成日志文件[24]。然而对 Linux 内核底层的修改，会导致运行的不稳定性增加，而且用户的交互过程是通过自动工具进行模拟实现的，并没有真实的操作。

代码抄袭检测技术也有一定的借鉴意义，早期的代码抄袭检测技术是静态的。静态的检测技术主要分为两类。第一类静态检测方法是属性计数 (AC) 为基础。这种方法主要考虑的静态特征是程序的属性统计数据，而不关心代码的结构。另一类静态检测方法是结构度量为基础的。它考虑的不仅仅是程序的属性，还考虑整个程序的结构。由于该方法考虑了更多的程序信息，因而其检测的准确性和 AC 方法相比较。而采用动态胎记技术来识别软件是近几年新兴的技术。首先提出动态胎记的是 G. Myles and C. Collberg<sup>[25]</sup>，他们利用程序在运行过程中完整的控制流来识别软件，在应对保留程序语义的攻击时，控制流方法比静态胎记技术更加有效。但如果程序受到代码混淆攻击，该技术就会失效。而且由于程序控制流庞大，对于较大的程序很难实现。Tamada et al.<sup>[26]</sup>提出了两种基于 API 函数调用建立动态胎记的方法<sup>[27]</sup>。采用 API 调用是考虑到抄袭者很难替换掉 API 函数，并且编译器不会优化 API 函数，因此较为稳定。通过分析 API 调用的顺序和调用频率

来建立动态胎记。这些胎记可以区分实现相同功能的不同程序，不会造成误判。Schuler et al.提出了一种给 JAVA 程序建立动态胎记的方法。它依赖于程序如何使用 JAVA 标准 API 提供的对象<sup>[28]</sup>。首先分析程序中每个 JAVA 标准 API 对象所接收到的函数调用序列，然后再将这些调用序列按照紧凑程度分割成若干段，这样就易于比较。通过实验表明，该方法可以准确识别出程序并有较好的区分度，比 G. Myles and C. Collberg 提出的 WPP 胎记方法具有更好的规模性和准确性<sup>[29]</sup>。Wang et al.提出了基于系统调用依赖关系建立动态胎记的方法。通过程序运行时系统调用之间的依赖关系建立一张系统调用依赖关系图（SCDG）<sup>[30]</sup>。在 SCDG 中，每一个系统调用作为节点，系统调用之间的依赖关系（即存在数据交流）作为边。SCDG 胎记作为整个 SCDG 的一个子图来识别程序。通过实验测试，该方法对于不同的编译选项，不同编译器以及代码混淆的攻击都具有很好的健壮性。这两种方法最大的问题是需要足够多的 API 调用，所以当程序的 API 数量不足时，该方法就无法建立有效的动态胎记，因此有较大的局限性。

### 1.3 主要研究内容及意义

基于国内外研究现状的分析，寻求一种更加有效的运行时软件特征是动态检测技术的迫切需要。目前已经有部分的研究人员将软件运行时堆内存作为特征来创建动态胎记。本文的研究方向就是以此为切入点，分析堆内存中分配的对象之间的引用关系，进而建立对象引用关系图<sup>[31]</sup>（ORG，Object Reference Graph），然后再以 ORG 为特征建立对象引用关系图胎记（ORGB，Object Reference Graph birthmark）来识别程序。

#### 1.3.1 对象引用关系图的概念

为了能够更加清晰地说明所要研究的主要内容，先给出重要概念的形式化定义。前面已经提到，动态胎记是由软件运行时的动态特征抽取建立的，可以作为软件识别的固有特征。下面给出动态胎记的形式化定义。

定义（动态胎记）：假设  $p$  和  $q$  为两个程序或者程序的组件， $I$  为  $p$  和  $q$  的输入。令  $f(p, I)$  为以  $I$  为输入执行  $p$  时抽取的一系列特征， $f(p, I)$  是  $p$  的动态胎记当且仅当以下两个条件同时成立：

- （1）当以  $I$  为输入执行  $p$  时， $f(p, I)$  仅仅是从  $p$  自身抽取的；
- （2）如果  $q$  是  $p$  的一部分，那么必有  $f(p, I) = f(q, I)$ 。

ORG 是对象引用关系图的简称。ORG 是一个有向图，图中的节点代表对象，

边代表对象之间存在引用关系。从同一个类产生的若干对象由一个节点代表，节点之间的多次引用仅由一条边来表示，一个对象发起的引用由代表这个对象的节点的出边来表示，同时忽略对象的自引用。下面是给出 ORG 的形式化定义。

定义 (ORG: Object Reference Graph): 对象引用关系图是一个二元组  $ORG = (N, E)$ ,  $N$  是图中节点的集合,  $N$  中的每一个元素表示的是产生对象的类。  $E \in N \times N$ , 是对象之间引用关系的集合。图 1-1 是一个由对象引用关系生成 ORG 的示例图。

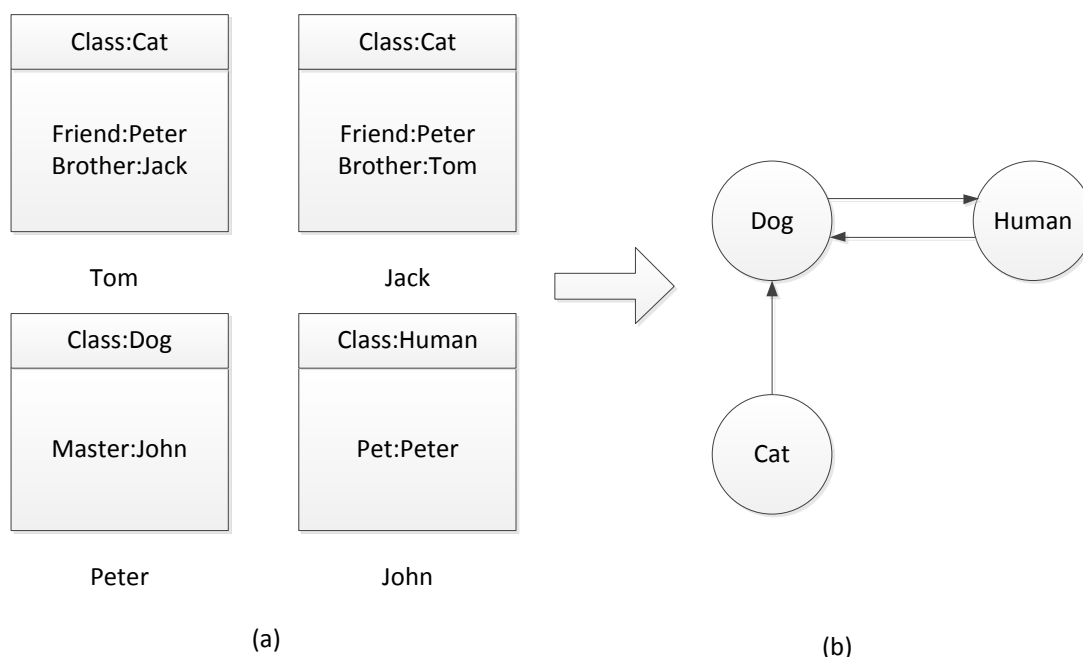


图 1-1 对象引用关系图示例

在图 1-1 (a) 中, 共有 Cat, Dog 和 Human 3 个类, 其中 Cat 有两个实例 Tom 和 Jack, Dog 类有一个对象 Peter, Human 类有一个对象 John。生成对象引用关系图如图 1-1 (b) 所示。虽然有 4 个对象, 但是因为仅仅有 3 个类, 故生成 ORG 图中有三个节点。因为对象 Peter 和 John 之间互有引用关系, 因此 ORG 图中代表这两个类的节点之间有两边。因为对象 Tom 引用了对象 Peter, 故代表 Tom 的节点到代表 Peter 的节点有一条出边, 注意虽然 Jack 和 Peter 同样有引用关系, 但是仅仅有一条边来表示。还应该注意的是 Tom 和 Jack 之间虽然也有引用关系, 但是因为同属一个节点, 是自引用关系, 因此忽略此边。

有了动态胎记和 ORG 的定义之后, 就可以定义对象引用关系图胎记 (ORGB: Object Reference Graph Birthmark)。

定义 (ORGB: Object Reference Graph Birthmark) 令  $p$  和  $q$  为两个程序或者程

序的组件，令  $I$  为  $p$  和  $q$  的输入。 $ORG_p$  和  $ORG_q$  是以  $I$  为输入， $p$  和  $q$  运行时各自的对象关系引用图。令  $ORGB_p$  为  $ORG_p$  的子图。 $ORGB_p$  是  $p$  的动态胎记当且仅当以下条件得到满足：

- (1) 如果  $q$  是  $p$  的一部分，那么必有  $ORGB_p$  子图同构于  $ORG_q$ ；
- (2) 如果  $q$  不是  $p$  的部分，那么必有  $ORGB_p$  不子图同构于  $ORG_q$ 。

$ORGB$  实际上是  $ORG$  的一个子图，程序在运行时，会在内存中会创建所有用到的对象。但是不能把所有对象作为程序识别的特征，只能够对其中有代表意义的对象建立对象引用关系图，形成程序的动态特征。这样产生的就是  $ORGB$ 。

### 1.3.2 主要研究内容

本文主要的研究内容是如何通过  $ORGB$  来识别 Android 恶意程序。具体过程如图 1-2 所示。

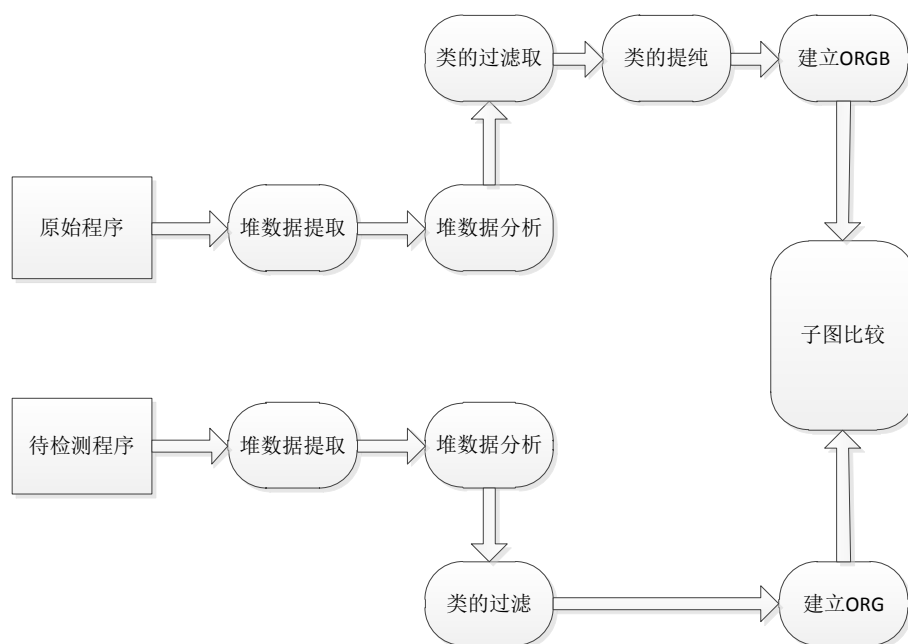


图 1-2 ORGB 识别程序的过程

首先运行原始程序，从堆内存中提取创建的对象和对象之间的引用关系，分析创建的对象，抽取其中的特征类，也就是选取一部分具有代表性的对象来建立  $ORGB$ 。然后运行待检测的程序，从内存中提取对象及引用关系，与  $ORGB$  不同的是，它需要使用所有的对象来建立  $ORG$ 。这样就形成了特征图  $ORGB$  和待检测图  $ORG$ 。于是，程序的识别抽象成为了两张有向图的子图同构比较，若  $ORGB$  与  $ORG$  是子图同构关系，则表明在待检测程序中发现了原始程序。

本文对  $ORG$  的研究面向 Android 平台，内容包括两个方面。一方面是在 Android

平台下如何提取出程序运行时内存中的对象信息。另一方面是设计子图同构的算法，使算法适合对象引用关系图的子图同构匹配。

### 1.3.3 研究意义

代码剽窃和恶意程序的检测归根结底是对程序的准确识别问题。静态的识别技术以静态特征为依据，以字符串识别匹配算法为基础，对程序进行识别。但是随着反检测技术的不断发展，这种检测方式逐渐失效，于是程序的动态检测技术应运而生。针对以上亟待解决的应用问题，本文的主要研究内容是基于堆内存建立软件动态胎记，通过程序运行时的动态特征，达到对软件识别的目的。这是一种新的建立软件动态胎记的技术。基于堆内的对象引用关系可以建立软件运行时的动态特征，以此作为软件的标记，达到对软件识别的目的。与数字水印、代码混淆、静态胎记和已有的动态胎记技术相比，由于它采用的堆内对象引用关系作为软件胎记，可以对变种的程序提高识别的准确性、有效性和健壮性，其应用范围十分广泛。

首先，该方法可以应用于恶意软件的检测。当前恶意软件泛滥，各银行、政府等重要部门受到严重影响，PC 和智能手机用户也深受其害，该技术在恶意软件检测方向提出了一个新的方法。对已知的恶意软件建立胎记库，再对可疑进程的堆内存抽取动态胎记，通过与胎记库比对，可实现对恶意软件的准确检测。

其次，可以应用该技术解决前文中提到的代码剽窃的问题。通过对原程序和涉嫌抄袭的程序分别抽取动态胎记并加以比对，就可以在多种防抄袭检测的攻击下，判断程序是否抄袭。同时，该技术的另一个特点是可以检测出程序的部分抄袭。例如抄袭程序虽然仅仅使用了其他程序部分的代码，但是在堆内的对象引用中依然会有体现，因此该方法也可以识别出这种部分抄袭的情况。

## 1.4 本文的组织结构

第 1 章绪论部分介绍了课题的研究背景，详细说明了目前代码剽窃和 Android 恶意软件泛滥的情况。对这两个问题的国内和国外研究现状进行了总结。最后介绍了课题的主要研究内容以及论文的结构。

第 2 章首先对 Android 系统平台进行了介绍，详细说明了系统的框架结构，对结构的各个模块进行了分析，描述了 Android 系统 4 个重要组件的功能。然后分析了 Android 平台下恶意代码，对其分类和攻击方式进行了详细的阐述。最后提出了 Android 平台下提取应用程序进程内存信息文件的技术，说明了提取实现的原理并



分析了数据格式及内容。

第 3 章详细分析了图的同构匹配算法。首先对图的基本概念，图同构的模式进行了严格定义。然后分析了常用的同构算法的特点，对比各个算法的优缺点，指出算法的适用场景。接着着重研究了本文中采取的 VF 算法，分析了其算法主要思想，适用情况以及性能。最后对算法进行了修改，加入控制匹配精确度的参数，使其适用于 ORG 图的同构匹配、

第 4 章首先给出了恶意代码检测系统的设计，包括 Android 客户端和服务端分析端。然后采用 VF2 算法以及  $\lambda$ -VF2 算法分别针对模拟样本进行了检测实验，验证 ORG 检测技术的有效性。使用  $\lambda$ -VF2-Monomorphism 算法对真实样本进行了测试，通过调整  $\lambda$  取值，使用检测的漏报率和误报率，分析了  $\lambda$  的合理取值。

## 第 2 章 Android 恶意代码分析及数据抽取实现

### 2.1 Android 系统平台分析

#### 2.1.1 Android 系统的内核结构

Android 是一种以 Linux 为基础的开放源码的智能移动平台系统。它广泛应用于智能手机和平板电脑。Android 系统的开发得益于谷歌公司。与 Linux、Windows 操作系统类似，Android 采用的也是分层架构的操作系统。Android 系统架构图如图 2-1 所示，Android 系统架构包含 4 层，最接近用户的是应用程序层，这一层运行着用户日常使用的各种应用程序；然后是应用程序架构层；再往下是系统运行库层；最底层是系统的内核层。

(1) 处于架构最底层的是 Linux 内核层。2.6 版本的 Linux 内核是 Android 系统内核的基础，Android 很多重要的服务都是基于该内核实现的，比如驱动程序、网络协议等等。谷歌公司对驱动层和硬件抽象层进行了一些修改，以适应 Android 智能手机的操作系统环境。Android 的内核层以驱动程序为主。例如 WIFI 驱动、电源管理驱动、相机驱动等等。

(2) 位于内核层之上的是 Android 核心函数库和运行环境。核心函数库中包含了很重要的函数，提供给应用程序开发者和 Android 组件使用。例如 C 函数库、多媒体库等等。这些函数库的使用降低了应用开发的难度，提高了开发效率。Android 运行时环境分为核心库和 Dalvik 虚拟机 (DVM)。类似于 JAVA 的程序运行于 JAVA 虚拟机 (JVM)，Android 的应用程序运行于 DVM。DVM 针对 Android 进行了特别优化，有很多优点。比如，由于 DVM 是基于寄存器实现的，因此它在代码的执行效率上更高。每一个应用程序的运行都对应一个 DVM 实例，实现了进程隔离，使进程之间不互相影响，提高了应用程序的安全性和系统的稳定性。

(3) 位于库和运行环境之上的是应用程序框架层。Android 的应用程序开发便是以此框架为基础的。该层为应用程序的开发提供了各种 API 接口，是 Android 开发的基础。同时该层提供了很多重要的组件，例如：通知栏管理器组件可以使应用程序在通知栏显示本应用的消息；位置管理器组件可以为应用程序提供位置信息等等。同时，应用程序框架层还对组件的使用进行了简化，使开发者可以快速开发出功能丰富的个性化应用程序。

(4) 框架的顶层是应用程序层。它是与用户距离最接近的一层。Android 上的应用软件是使用 JAVA 语言编写的。这一层包含的是用户在使用手机时安装的各

种第三方应用软件和系统自带的核心应用。常见的应用包括：通讯录，浏览器，拨号程序等等。

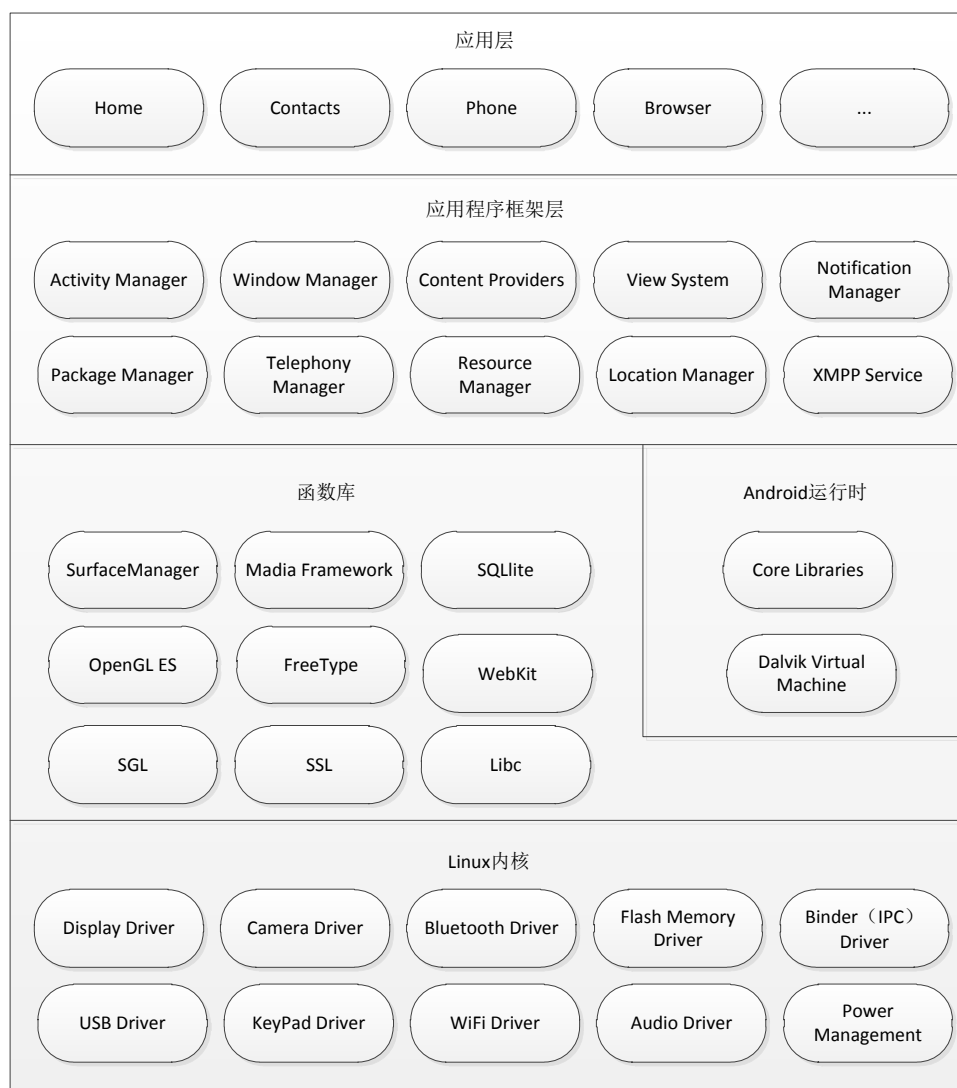


图 2-1 Android 系统架构

## 2.1.2 Android 应用组件

Android 系统下的应用是由不同的应用程序组件按照功能需求组合而成的。Android 系统提供的应用组件共有 4 类: Activities (活动), Services (服务), Content providers (内容提供) 以及 Broadcast receivers (广播接收)。在 Android.content 当中定义了 Content provider 和 Broadcast receiver 类<sup>[32]</sup>, 而 Activity 和 Service 的类定义在 android.app 中<sup>[33]</sup>。它们有着各自的功能和生命周期, 组件的产生和回收由本身的生命周期决定。下面对这四种组件一一进行详细分析<sup>[34]</sup>。

### 2.1.2.1 Activity

Activity 是 5 个组件中最常用的。程序中 Activity 通常的表现形式就是我们看到的应用程序的界面，一个 Activity 表示一个界面，界面带有用户接口。每个 Activity 都是一个单独的类，它扩展实现了 Activity 基础类。这个类显示为一个由 Views 组成的用户界面，并响应事件。大多数程序有多个 Activity。例如，一个照相机程序有这么几个界面：取景界面，参数设置界面，图片显示界面等等。每个界面都是一个 Activity，界面之间的切换实际上就是 Activity 之间的切换。Activity 之间可以传递数据，例如登录界面的 Activity 会把用户填写的用户名、密码以及验证码等等传递给验证界面的 Activity。打开一个新的界面后，前一个界面就被暂停，并放入历史堆栈中（界面切换历史栈）。使用者可以回溯前面已经打开的存放在历史堆栈的界面。也可以从历史堆栈中删除没有使用价值的界面。Android 会自动保存程序运行时的所有界面，从第一个界面一直到当前打开的界面。

当 Activity 被创建或者销毁的时候，它们就会入栈或者出栈，在整个过程中 Activity 会存在 4 种可能的状态。Activity 的状态转换图如图 2-2 所示。

(1) active: 在栈顶的 Activity 是可见的，是当前具有焦点的 Activity，用来响应用户的输入。系统会首先满足它的活跃性，在资源不够的情况系统会消灭更靠下的 Activity。当有新的 Activity 入栈的时候，它就会变为 paused。

(2) paused: 可见但不具有焦点的 Activity 的状态就是 paused。当前具有焦点的 Activity 如果不是全屏的，那么下面 Activity 的状态就是 paused。但此时，该 Activity 还是被视为 active 的，只是不能与用户进行交互。在特殊情况下，系统会销毁处于 paused 状态的 Activity 来恢复资源给 active 的 Activity。当一个 Activity 完全不可见的时候，它就变成 stopped。

(3) stopped: 不可见 Activity 的状态。这个 Activity 里所有的状态和成员信息会在内存里保留，但是当系统需要内存的时候，它就会被销毁。当一个 Activity 处于 stopped 的状态时，保存数据和当前 UI 状态十分重要。一旦 Activity 被销毁，它就会变成 inactive。

(4) inactive: 当一个 Activity 被销毁的时候，它就会变成 inactive，被从栈中删除，再次使用的时候需要重新启动。

### 2.1.2.2 Service

Service 是一个没有用户界面，运行于后台且常住系统的程序。我们以 Android 手机上的 FM 收音机为例对 Service 的功能进行说明。在收音机应用程序中，会创建一个 activity 让用户选择频率，当用户选择完毕，收音机开始播放之后，用户可

能就去浏览照片或者阅读小说而切换页面了。这时收音机通过调用 Context.startService()来启动一个 Service，这个 Service 在后台运行以保持继续播放广播。通过 Context.bindService()方法连接到 Service，就可以通过接口与 Service 进行数据传输。对于收音机来说，可以实现控制广播的暂停，播放等功能。

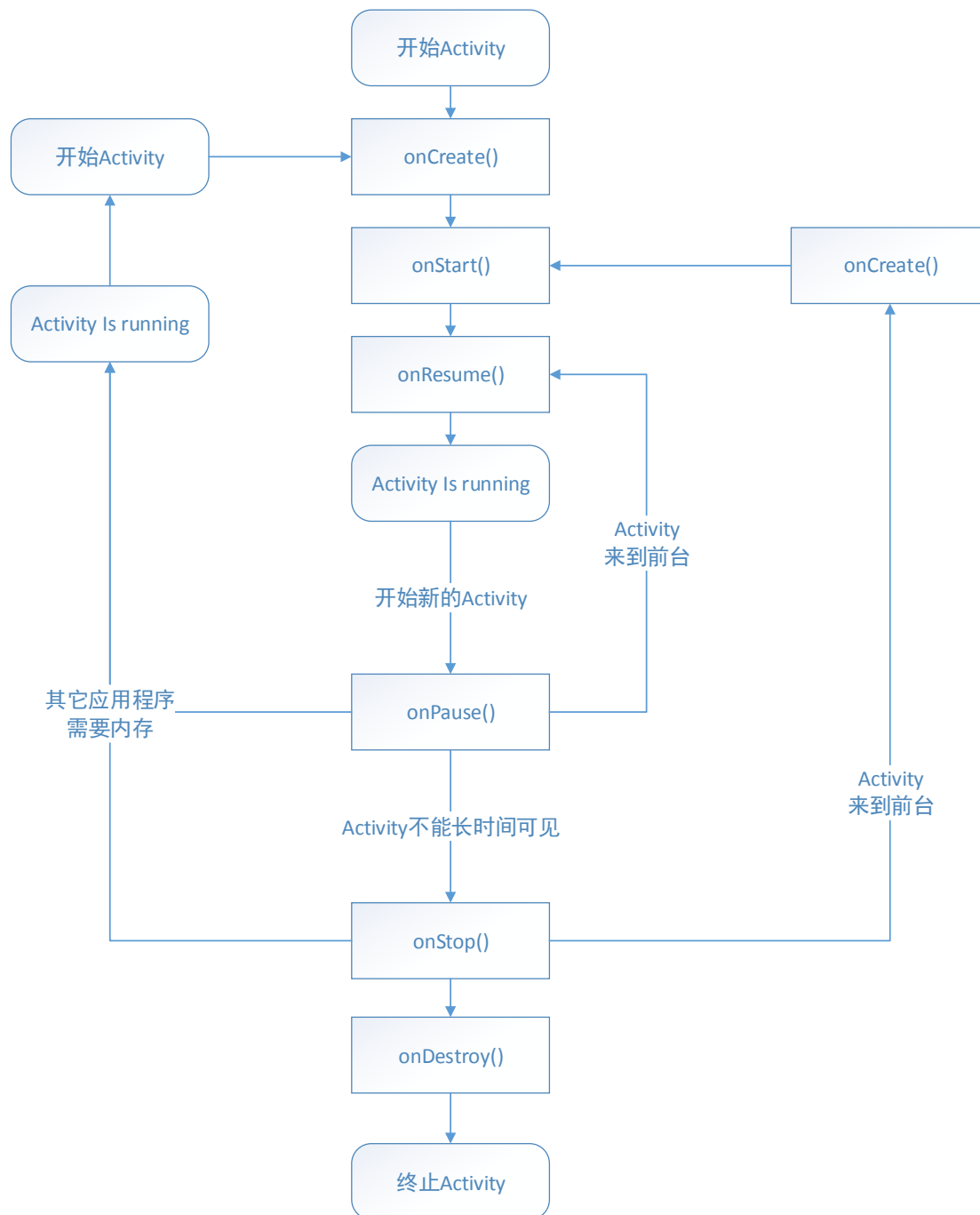


图 2-2 Activity 状态转换图

### 2.1.2.3 Content provider

Android 的应用程序之间也需要数据共享，Content provider 就是实现这一功能的重要组件。Android 应用程序产生的数据可存储于 SQLite 数据库、web 或者其他有效地设备当中。Content provider 是一个实现了一系列标准方法的类，这个类使其他程序能存储、读取某种 Content provider 可处理的数据。例如，很多应用程序都需要访问手机里的联系人信息，联系人信息的共享就是通过在系统运行一个 Content provider 组件来实现的。这样，只要应用程序获得了相应的权限就可以访问这些数据内容了。

#### 2.1.2.4 Broadcast receiver

在 Android 系统中，应用程序之间的消息传递是通过广播机制实现的。而实现这一功能的组件就是 Broadcast receiver。开发者的应用程序可以通过 Broadcast receiver 组件选择监听自己需要的广播消息，做出处理。系统或者应用程序都可以产生广播消息。对系统来说，当手机的电量不足，接收到短信、电话，手机屏幕的关闭时会产生一个消息广播。对于应用程序来说，当下载完毕时会产生一个消息广播。通过调用 Context.sendBroadcast()，应用程序可以实现消息广播。另外值得注意的是，BroadcastReceiver 组件本身无法启动新的交互界面，当它接收到广播之后，可以根据自身程序的功能需要作出行动。

## 2.2 Android 恶意代码分析

随着 Android 的普及，Android 智能手机的用户数量也在飞速增长。目前，针对 Android 手机的恶意软件越来越多，传播的范围也十分广泛，严重影响着用户的手機安全。不同于计算机病毒，Android 恶意程序的攻击行为有着自身的特点，恶意代码的类型覆盖了多个层面。各种恶意软件类型所占的比例如图 2-3 中所示<sup>[35]</sup>。

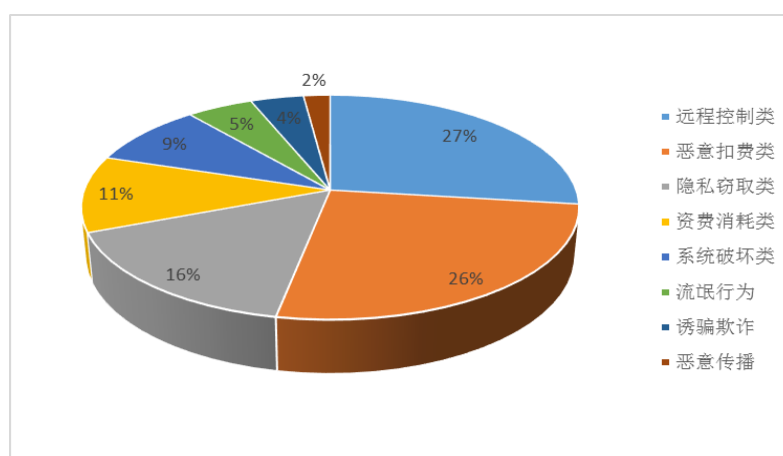


图 2-3 恶意软件分类比例

下面详细分析 Android 恶意程序的各个类型。

#### (1) 远程控制类

远程控制类的恶意软件在 PC 上就层出不穷，但是对于 Android 系统而言却是刚刚兴起的类型。早期恶意代码造成的威胁是确定的、静态的，而远程控制造成的威胁是不确定的、动态的。当系统感染了远程控制类的恶意软件后，软件会在运行于后台，用户难以发现。通过网络等悄悄与相应的服务器端进行连接，接收服务器发送的指令实现攻击者的远程操作。

#### (2) 恶意扣费类

恶意扣费类恶意软件被安装进入手机后，会在后台隐蔽执行，在用户不知情的情况下进行扣费操作，对用户造成一定经济损失。扣费的实现方式多种多样，例如，联接网络造成用户流量损失，发送彩信、短信、拨打电话，强制订阅 SP 服务。此类软件可以产生较大的经济利益，因此成为 Android 恶意软件中占取比例较高的一类。

#### (3) 隐私窃取类

隐私窃取类恶意软件的目标是窃取用户手机上的隐私信息，包括 GPS 地理位置、短信内容、通话记录、联系人信息甚至手机的上文件和图片等等。攻击者利用恶意软件将这些隐私内容在用户无法察觉的情况下，偷偷发送到对应的服务器，再将这些信息汇总、出售，获得经济利益。这类软件不会给用户造成直接的经济损失，但是却严重影响了用户的个人隐私。

#### (4) 系统破坏类

系统破坏类恶意软件的目的是系统无法正常运行，导致崩溃。由于软件在攻击的过程中会使用较高权限，因此会首先获取系统的最高权限，root 权限。这样，恶意软件就可以随意进行系统操作，更改系统设置，删除系统应用，对用户数据进行更改，给用户带来严重的、无法修复的损失。

#### (5) 流氓软件类

和 PC 机上的流氓软件类似，Android 上的流氓软件也是通过捆绑等手段强制安装，用户难以卸载。为了不让用户对软件进行卸载，一般流氓软件会在安装完成之后，将数据更改为只读模式，使程序无法删除。一些流氓软件采取了更极端的方式，即使被用户卸载之后，也能通过某些方法实现复位，重新安装运行。攻击者编写流氓软件的目的绝大多数都是为了在软件中嵌入广告，推广商品。流氓软件给手机用户带来了很大困扰，除此之外，强行安装的软件一般并不稳定，运行时容易崩溃，对系统的稳定性也造成一定的影响。

### (6) 诱骗欺诈类

诱骗欺诈类恶意软件通过伪造虚假信息欺骗用户，对用户直接造成经济利益的损失，具有极大的危害性。该类恶意程序感染系统后，主要通过两种方式对用户实行诈骗。一是向通讯录中的联系人发送伪造的求助信息，骗取联系人向指定的银行账号汇款等等；二是在手机中制造虚假短信，内容多为银行消息、中奖通知、恶意链接等等。

### (7) 恶意传播类

恶意传播类软件在没有经过用户授权，在用户不知情的情况下，将自身或其它恶意程序传播扩散，感染其它设备。恶意传播类软件经常采取的方式是在短信中键入恶意链接、通过无线网络传播恶意代码和感染手机内应用程序等等。

## 2.3 Android 平台下堆内存数据抽取的实现

一般情况下，现有恶意代码会被嵌入到正常 APK 安装文件中。当 APK 被安装进入 Android 手机，应用程序启动后，恶意代码也会随之运行，在内存中与应用程序共用一个进程。整个进程在运行过程中，会创建对象，对象之间会有相互的引用。那么，进程创建了哪些对象，每一个对象的引用者和被引用者是谁，就是需要抽取的内存数据。

对象引用关系的获取全部是在 Android 平台下完成的。这样做的主要原因是导出的原始内存文件过大。Android 自带的“计算器”应用程序，其内存文件就有 10M 左右。一台手机上有若干个进程，每个进程产生一个大于 10M 的文件传输给服务器分析，数据传输量过大。因此，必须在 Android 平台下分析出原始文件中的有效信息，提取之后传给服务器，减少传输数据量。然而 Android 系统并没有提供现成的工具来进行文件的分析。因此需要分析文件格式，编写运行于 Android 平台的分析程序。

Android 平台下获得引用关系分为 3 个步骤。

第一步导出原始内存文件。首先要确定指定进程的 ID，然后通过 `dumpheap` 工具导出该进程的内存信息文件。第一步较为简单，但是产生的文件还远远不能作为可用的数据。

第二步是对内存信息文件进行格式转换。转换是通过编写格式转换工具 `Conventor` 实现的，详细的实现过程在下文 2.3.2 小节中说明。格式转换的目的是为了第三步分析工作的方便。

第三步对文件进行分析，找出对象及引用关系相关内容，抽取出来。这一步



的最终结果是产生仅仅包含对象引用关系数据的文件。其实现是通过编写分析工具 AHAT 实现的，具体过程在 2.3.3 小节会说明。

### 2.3.1 Android 进程的堆内存信息文件的获取

Android 的 SDK 提供了功能丰富的内存监视工具，其中包括对堆数据监控的 dumpheap 工具。但是并非所有的 Android 版本都支持 dumpheap 工具，只有 android2.3 版本以上的系统才能够通过 dumpheap 获取堆信息。

这些信息以二进制的形式存储于文本当中，是内存里的原始数据，无法直接使用。PC 平台下可以使用 JDK 的 JHAT 对二进制文件进行分析，但是在 Android 系统中直接分析数据需要重新实现，这个工作在下一小节中会详细介绍。下面首先讲解数据的提取过程。

#### (1) 数据提取环境

表 2-1 Android 内存数据的提取环境

PC 操作系统	使用软件	虚拟 android 环境
PC 操作系统 Windows7	ADT (Android Developer Tools)	4.0.3 版本

#### (2) 提取过程

为了便于分析，实验中，在 PC 使用 AVD 虚拟了 Android 4.0.3 版本的系统，使用 dumpheap 工具成功提取了指定进程的堆数据信息，后续为了方便将 dumpheap 集成进入了 Android 下的分析应用程序。首先在 PC 端的 CMD 中，使用 sdk 的 adb 命令进入虚拟设备的 shell，切换到 androidsdk 的 platform-tools 文件夹。显示设备的序号，从图 2-4 可以看到当前与 PC 相连的 android 设备只有一个，设备编号为“emulator-5554”，这是用 AVD 创建的虚拟 android 设备。

```
D:\android-sdk-windows\platform-tools>adb devices
List of devices attached
emulator-5554    device
```

图 2-4 已接连的 Android 设备

然后通过 adb 命令获取“emulator-5554”的 shell，实验中在虚拟设备打开了系统自带的“计算器”应用程序，实验的目的是获取计算器应用进程对应的堆内存信息。

Dumpheap 需要进程的 ID 号作为参数，因此首先需要获得“计算器”的进程 ID。图 2-5 显示了当前虚拟设备上运行的所有进程，其中 PID 为 857 的进程就是计算器的进程，进程名是“com.android.calculator2”。有了进程的 ID 号，就可以通过 dumpheap 来导出该进程的内存数据文件。

```
# ps -x
```

USER	PID	PPID	VSZ	RSS	WCHAN	PC	NAME
root	1	0	276	192	c0099f1c	000086e8	S /init (u:4, s:550)
root	2	0	0	0	c004df64	00000000	S kthreadd (u:0, s:1)
root	3	2	0	0	c003fa28	00000000	S ksoftirqd/0 (u:0, s:3)
root	4	2	0	0	c004abc0	00000000	S events/0 (u:0, s:49)
root	5	2	0	0	c004abc0	00000000	S khelper (u:0, s:0)
root	6	2	0	0	c004abc0	00000000	S suspend (u:0, s:0)
root	7	2	0	0	c004abc0	00000000	S kblockd/0 (u:0, s:0)
root	8	2	0	0	c004abc0	00000000	S cqueue (u:0, s:0)
root	9	2	0	0	c0178c7c	00000000	S kseriod (u:0, s:0)
root	10	2	0	0	c004abc0	00000000	S kmcd (u:0, s:3)
root	11	2	0	0	c006efa8	00000000	S pdfflush (u:0, s:0)
root	12	2	0	0	c006efa8	00000000	S pdfflush (u:0, s:13)
root	13	2	0	0	c0073480	00000000	S kswapd0 (u:0, s:0)
root	14	2	0	0	c004abc0	00000000	S aio/0 (u:0, s:0)
root	24	2	0	0	c01764ac	00000000	S mtdblockd (u:0, s:0)
root	25	2	0	0	c004abc0	00000000	S kstripped (u:0, s:0)
root	26	2	0	0	c004abc0	00000000	S hid_compat (u:0, s:0)
root	27	2	0	0	c004abc0	00000000	S rpciod/0 (u:0, s:0)
root	28	2	0	0	c0193fd0	00000000	S mmcd (u:0, s:17)
root	29	1	252	156	c0099f1c	000086e8	S /sbin/ueventd (u:6, s:25)
system	30	1	776	292	c019ff30	40010690	S /system/bin/servicemanager (u:3, s:28)
root	31	1	3964	772	ffffffff	40015c94	S /system/bin/vold (u:4, s:9)
root	34	1	640	280	c01abd48	4000cf98	S /system/bin/debuggerd (u:0, s:1)
radio	35	1	5440	768	ffffffff	40010c94	S /system/bin/rild (u:48, s:94)
system	36	1	28768	11664	ffffffff	40036690	S /system/bin/surfaceflinger (u:6919, s:703)
root	39	1	788	332	c0207be4	40010458	S /system/bin/installld (u:0, s:1)
keystore	40	1	1680	548	c01abd48	40010f98	S /system/bin/keystore (u:0, s:2)
root	41	1	776	308	c00b72d0	400113c0	S /system/bin/qemud (u:0, s:3)
shell	44	1	704	308	c014ea24	4000c458	S /system/bin/sh (u:1, s:0)
root	45	1	4416	208	ffffffff	0000825c	S /sbin/adbd (u:5, s:24)
media	451	1	20500	5528	ffffffff	4004f690	S /system/bin/mediaserver (u:49, s:8)
root	452	1	6300	1016	ffffffff	40040c94	S /system/bin/netd (u:2, s:4)
root	454	1	148376	38052	ffffffff	400107b4	S zygote (u:1864, s:142)
system	469	454	216456	47088	ffffffff	40010690	S system_server (u:3302, s:1555)
system	522	454	165824	45548	ffffffff	400113c0	S com.android.systemui (u:396, s:107)
radio	551	454	173172	35828	ffffffff	400113c0	S com.android.phone (u:1628, s:1166)
app_13	565	454	176120	58572	ffffffff	400113c0	S com.android.launcher (u:1195, s:292)
app_21	581	454	162388	31912	ffffffff	400113c0	S com.android.calendar (u:41, s:26)
app_34	612	454	159664	31752	ffffffff	400113c0	S com.android.deskclock (u:36, s:25)
app_11	625	454	159376	32036	ffffffff	400113c0	S com.android.providers.calendar (u:75, s:32)
app_1	645	454	186300	54504	ffffffff	400113c0	S android.process.acore (u:186, s:111)
system	665	454	167616	40108	ffffffff	400113c0	S com.android.settings (u:56, s:44)
app_1	688	454	164344	34864	ffffffff	400113c0	S com.android.contacts (u:55, s:24)
app_7	722	454	160536	32756	ffffffff	400113c0	S android.process.media (u:82, s:26)
app_17	740	454	162996	31208	ffffffff	400113c0	S com.android.exchange (u:26, s:20)
app_27	754	454	170720	35956	ffffffff	400113c0	S com.android.email (u:81, s:40)
app_28	775	454	159412	32044	ffffffff	400113c0	S com.android.mms (u:39, s:33)
app_6	795	454	158700	34628	ffffffff	400113c0	S com.android.inputmethod.latin (u:67, s:30)
app_16	857	454	160612	36320	ffffffff	400113c0	S com.android.calculator2 (u:1293, s:237)
root	872	45	704	332	c003d800	4000d284	S /system/bin/sh (u:1, s:1)

图 2-5 运行进程列表

dumpheap 命令的格式是“am dumpheap ID 存储路径”。命令执行完毕后将进程 857 的堆数据保存在了 HeapData 文件中。这样就在 android 平台下导出了一个完整的堆内存原始信息的文件。此文件为二进制的文件，无法直接读取其中内容。需要对二进制文件的格式进行分析。

### 2.3.2 内存信息文件格式转换的实现

本文编写的分析工具 AHAT 是以 JHAT 为基础的，因而要求内存文件的格式与 JAVA 的保持相同，于是需要转换二进制文件格式。Dumpheap 生成的二进制内存文件的版本是 1.0.3，而 JHAT 可以分析的版本是 1.0.2。因此需要在 Android 平台下对文件格式从 1.0.3 转换为 1.0.2。

为此，编写了文件格式转换工具 Converter。Converter 的功能与 SDT 中的 HprofConv 工具实现的功能类似，区别在于 Converter 运行于 Android 平台，而

HprofConv 运行于 PC 平台。

首先要对两个版本的格式进行分析。Dumpheap 产生的二进制文件格式如图 2-6 所示。二进制文件的格式是固定的，以版本字符串开头，通常是“JAVA PROFILE 1.0.2”，然后是 4 字节的 ID 信息，接着是 8 字节的文件创建日期信息。之后是文件的主体，内存数据。

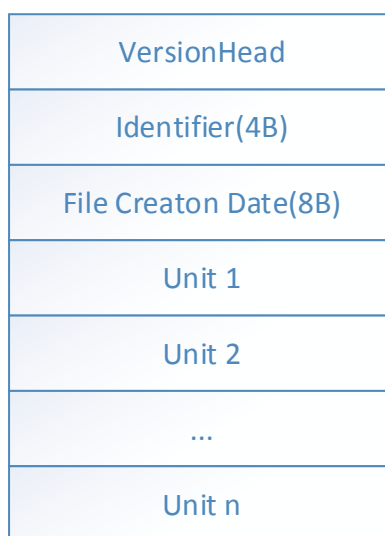


图 2-6 Dumpheap 文件格式

数据是由 Unit 单元组成的，每个单元的格式相同，内容不同。其中每个记录单元代表了一个 JAVA 对象的信息。Unit 格式如图 2-7 所示，首先是 1 字节的类型，然后是 4 字节的时间戳，然后是 4 字节的数据长度 n，最后是 n 字节的对象信息。

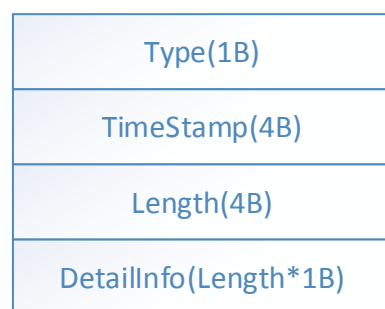


图 2-7 Unit 格式

内存文件 1.0.2 版本和 1.0.3 版本的主要区别在于 Unit 数据单元中 Detail Info 字段的类型数量是不相同的。在 1.0.2 版本的内存文件中，表 2-2 列出了 Detail Info 中 13 种可能的片段种类。

表 2-2 1.0.2 版本 Unit 类型

类型名称	16 进制标志
HPROF_ROOT_UNKNOWN	0xff,
HPROF_ROOT_JNI_GLOBAL	0x01,
HPROF_ROOT_JNI_LOCAL	0x02,
HPROF_ROOT_JAVA_FRAME	0x03,
HPROF_ROOT_NATIVE_STACK	0x04,
HPROF_ROOT_STICKY_CLASS	0x05,
HPROF_ROOT_THREAD_BLOCK	0x06,
表 2-2（续表）HPROF_ROOT_MONITOR_USED	0x07,
HPROF_ROOT_THREAD_OBJECT	0x08,
HPROF_CLASS_DUMP	0x20,
HPROF_INSTANCE_DUMP	0x21,
HPROF_OBJECT_ARRAY_DUMP	0x22,
HPROF_PRIMITIVE_ARRAY_DUMP	0x23,

在 1.0.3 版本中，新增了 9 种 Unit 类型，具体见表 2-3。

表 2-3 1.0.3 版本 unit 新增类型

类型名称	16 进制标志
HPROF_HEAP_DUMP_INFO	0xfe,
HPROF_ROOT_INTERNED_STRING	0x89,
HPROF_ROOT_FINALIZING	0x8a,
HPROF_ROOT_DEBUGGER	0x8b,
HPROF_ROOT_REFERENCE_CLEANUP	0x8c,
HPROF_ROOT_VM_INTERNAL	0x8d,
HPROF_ROOT_JNI_MONITOR	0x8e,
HPROF_UNREACHABLE	0x90,
HPROF_PRIMITIVE_ARRAY_NODATA_DUMP	0xc3,

可以看到，1.0.3 版本的内存文件包含的内容类型更多，导致 AHAT 无法识别。转换的思路是对每个 unit 的类型进行判断，只保留原始的类型，过滤掉新增类型。转换程序的流程如图 2-8 所示。

Unit 类型作为 Convertor 类的成员变量，在分析的过程中会被用于判断标志，决定该片段是否要写入输出文件，最后将文件按照 1.0.2 版本的格式重新组织文件。

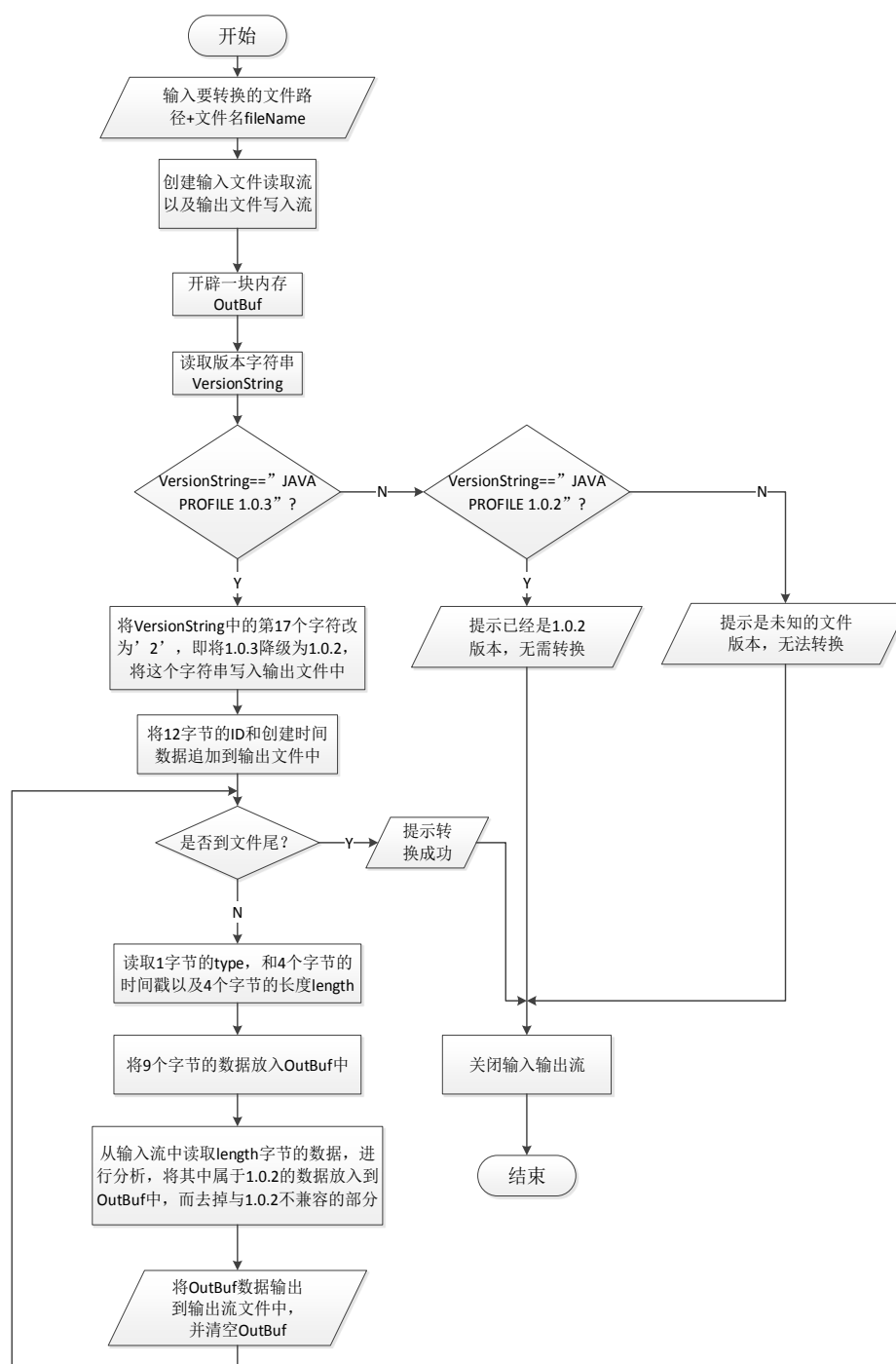


图 2-8 转换程序流程图

## 2.3.3 对象及引用关系数据的抽取

### 2.3.3.1 AHAT 整体结构

AHAT 的结构有四大块：Model、Parser、Util 以及外界调用接口。四个模块之

间的关系如图 2-9 所示。

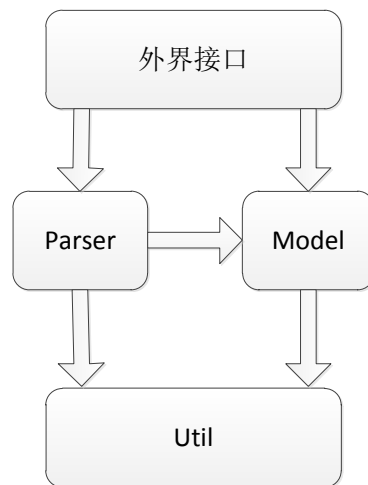


图 2-9 AHAT 结构

(1) **Model**: 定义了可能涉及到的所有对象的类型（数据结构），这些数据结构的对象组成了一个模型。共有 29 个类，对应着 JAVA 中的对象类型，其中最重要的类是 **Snapshot**，是内存快照模型的最大单元。

(2) **Parser**: 负责读取二进制文件，分析数据并将之填充到模型对象中，构建一个模型。共有 7 个类，最主要的类是 **HprofReader**，负责读取堆二进制文件。

(3) **Util**: 常用的工具包。

(4) 外界调用接口: **AHAT** 的框架，负责调用各个模块，使之正常工作起来。在 **Android** 中与用户交互的是 **Activity** 类，因此这块主要的类是 **MainActivity** 类，还有用于获取类引用关系的 **QueryClassInfo** 类。

### 2.3.3.2 AHAT 分析过程

根据 **Jhat** 的工作过程，**AHAT** 分析文件分为以下 4 个步骤。

(1) **Create**: **AHAT** 首先创建一个 **Snapshot**，用来准备存储数据。

(2) **Read**: **HprofReader** 类读取二进制文件内容，获取必要的数，进行数据填充，构建 **Snapshot** 对象。

(3) **Resolve**: **Snapshot** 对象根据读取的数据，初始化数据结构，计算对象信息，包括类之间的引用关系。

(4) **Query**: 根据构建好的模型，获取类引用关系，写入文件中。

### 2.3.3.3 重要数据结构和方法。

(1) **Snapshot** 类: 它代表某一时刻 **JVM** 中的 **JAVA** 对象的快照，里面存储了所有程序运行时 **JAVA** 对象的信息，以及互相之间的引用关系，这是从单个内存文件中读取的最大的模型（**Model**）对象单元。其包含的数据结构如表所示。这其中

涉及到的数据结构都是在 Model 模块定义的。

(2) HprofReader 类：它有一个方法 read(), 用于读取二进制文件, 并返回一个 Snapshot 对象。HprofReader 的 read 方法主要就是将这些信息读取到内存中, 将每个单元里的对象信息填充到 Snapshot 对象中。通过 HprofReader 的 read() 方法读取完信息之后, 得到 Snapshot 对象, Snapshot 中的 resolve 方法用于初始化数据结构, 计算每个对象的具体信息, 包括包名、类名、类的 ID、类的成员变量、类之间的引用关系等等。这个方法是获取类之间的引用关系的核心方法。

(3) QueryClassInfo 类：经过 resolve 之后, 需要的信息已经计算出来, 最后就需要将这些信息获取, 并写入文件中。QueryClassInfo 类的功能就是将 Snapshot 对象中的类之间的引用关系提取出来。其 process 方法中的 referrersStat 变量是一个 Hash map, 用于存储该类的被引用信息, 而 refereesStat 变量用于存储类的引用信息, 在该方法中, 通过 Snapshot 的 getClasses 方法获取内存快照中的所有类。

(4) PlatformClasses 类：在获取对象引用关系时, 通过 getClasses 方法获得的类一般都有上千个, 而这些类中大多数都是平台提供的类, 如 JAVA 标准 API 中的类, Android 系统提供的 API 类等等。这些类和要分析的问题没有关系, 如果加入分析行列的话, 还可能冲淡真正关键的类之间的关系。因此, 需要把这些类屏蔽。PlatformClasses 的作用就是去掉平台提供的类, 这些平台提供的类是定义在 platform\_names.txt 文件中的, 文件内容如图 2-10 所示。

boolean	JAVA.	JAVAx.sql.
char	JAVAx.accessibility	JAVAx.swing.
float	JAVAx.crypto.	JAVAx.transaction.
double	JAVAx.imageio.	JAVAx.xml.parsers.
byte	JAVAx.naming.JAVAx.net	JAVAx.xml.transform.
short	JAVAx.print.	org.ietf.jgss.
int	JAVAx.rmi.	org.omg.
long	JAVAx.security.	org.w3c.dom.
sun.	JAVAx.sound.	org.xml.sax.

图 2-10 抽取过程中需要过滤的类

该文件存放在 asserts 文件夹下, 作为 AHAT 程序的资源文件。当 QueryClassInfo 获取到一个类时, 通过 PlatformClasses 来判断该类名是否包含定义在 platform\_classes.txt 文件中的前缀, 如果包含的话, 就忽略该类的分析。这样可以随意添加和删除所要过滤的类, 列表的维护较为方便。

#### 2.3.3.4 AHAT 运行结果

AHAT 的运行平台要求是 Android4.0 以上。AHAT 的测试运行于实际的硬件开发设备当中，选用的 Android 手机是 Google 的 Galaxy nexus 3，具体实验环境见表 2-4。

表 2-4 AHAT 运行环境

设备名称	Galaxy nexus 3
Anroid 系统版本	Android4.1.2
手机 RAM	1GB
CPU	德州仪器 OMAP4460，双核，频率 1228MHz

在 Android 上运行 AHAT 之后，选取 Dumpheap 文件，对已经选中的文件进行数据分析。

分析的过程包括 dumpheap 文件的读取，二进制数据解析，类引用关系分析，以及结果文件的创建。数据结果文件存放在 SD 卡的 dumpheap 文件夹下。

共有2563个类

```

Package ts.NetTraffic
class ts.NetTraffic.ViewService
[0x4129be60]
  Referrers by Type
    java.lang.Object[] 11
    android.app.
LoadedApk$ReceiverDispatcher 4
  java.util.HashMap$HashMapEntry 2
  ts.NetTraffic.p 1
  ts.NetTraffic.n 1
  ts.NetTraffic.q 1
  ts.NetTraffic.o 1
  ts.NetTraffic.s 1
  ts.NetTraffic.j 1
  ts.NetTraffic.m 1
  android.app.ContextImpl 1
  ts.NetTraffic.k 1
  ts.NetTraffic.l 1
  Referees by Type
    android.app.Application 1
    ts.NetTraffic.p 1
    ts.NetTraffic.n 1
    ts.NetTraffic.m 1
    android.app.ContextImpl 1
    ts.NetTraffic.k 1
    java.text.DecimalFormat 1
    ts.NetTraffic.q 1
    java.lang.Class 1
    ts.NetTraffic.o 1
    android.app.ActivityManagerProxy 1
  
```

图 2-11 形成的关系文件

创建的文件格式如图 2-11 所示。第一行列出的是包的名字，每个包都以 Package 开头，实验示例显示的是 ts.NetTraffic 包。接下来是包中所包含的类，每个类都以 class 开头。图中显示的类是 ts.NetTraffic.ViewService，其内存地址为 0x4129be60。Referrers by Type 显示的是引用该类的类，而 Referrees by Type 表示的是被该类所引用的类。至此完整的引用关系数据已经导出，数据抽取的工作成功实现。



## 2.4 本章小结

本章主要介绍与 Android 平台相关的内容。2.1 节对 Android 系统平台绍了系统框架结构进行了详细说明,介绍了各个模块的功能,分析了 Android 的 4 个重要组件重要作用。2.2 节阐述当前 Android 平台下恶意代码的种类和特点,对其攻击方式进行了分析。2.3 节提出了 Android 平台下提取应用程序进程内存文件的方法,详细介绍了抽取工具实现的过程,并进行了实验。

## 第 3 章 图及其匹配算法的研究与分析

在过去的几十年里，图形匹配一直是计算机科学里众多研究的主要课题之一。大体上来讲，图匹配算法被分为两大类：精确匹配算法和非精确匹配算法<sup>[36]</sup>。精确匹配要求匹配的两张图之间具有严格的一致性，或者至少要求子图与模式图之间具有严格的一致性；而非精确匹配算法，也称为容错匹配算法，则放宽了这一条件，允许目标存在误差和噪音。当两张图在结构上有一定程度不一致的情况下，依然可以判定它们为匹配。这两类算法在实际应用中都有广泛的应用。

### 3.1 图的同构模式

图精确匹配的特点是这样的：两张图节点之间的映射必须具有边保留性质，即如果第一张图的两个节点之间是存在边的，那么映射到第二张图的两个节点之间必须存在边。最严格的图精确匹配的形式是 **graph Isomorphism**，它要求上述条件从两个方向上都是满足的，两张图上的节点和边的映射是一个双射。也就是说，第一张图的每一个节点和第二张图的每一个节点都具有一对一的相关性。弱一点的图精确匹配形式是 **Subgraph Isomorphism**，它要求一张图的子图和另一张图成为 **graph Isomorphism**。更为弱一些的匹配形式是在 **Subgraph Isomorphism** 的基础上去掉了边保留的双向要求，也就是说两个节点的边可以不是一个双射。这样的映射被称为 **Monomorphism**。它要求第一张图的每一个节点都能映射到第二张图的不同节点，同时第一张图的每一条边能够和第二张图上的边对应上；然而，第二张图允许存在多余的边和节点。更弱的匹配是 **homomorphism**，它不要求第一张图的每一个节点都映射到第二张图上不同的节点，因此这是一个多对一的映射。还有一种图的匹配方式是，两张图的子图是 **Isomorphism** 匹配。这样的结果并不唯一，通常算法的目标是找出最大的子图匹配，这就是“最大公共子图问题”（**MCS, maximum common Subgraph**）。

#### 3.1.1 图的基本概念

各种文献当中对图的定义是不尽相同的，这主要取决于具体的应用领域。而经过大量的实例，下面给出的定义是具有广泛适用性的。

定义：（图）令  $L_V$  和  $L_E$  分别是节点和边的有穷或无穷字母集合，图  $g$  是一个满足下列条件的四元组  $g=(V,E,\mu,\nu)$ ：

- （1） $V$  是有限的顶点集

- (2)  $E \subseteq V \times V$  是边集
- (3)  $\mu: V \rightarrow L_V$  是顶点属性映射函数
- (4)  $\nu: E \rightarrow L_E$  是边属性映射函数

这个图的定义是普遍适用的，特别要说明的是这个定义加入了图顶点和边的属性。 $|g|$ 表示的是图中顶点的个数。图 3-1 列出了四个不同类型的图，均符合以上定义。其中，(a)无属性的无向图,(b)无属性的有向图,(c)顶点带属性的无向图,(d)顶点和边均带属性的有向图。有了不受限制的属性映射函数，我们可以对任何结构图进行形式化处理。例如，赋予给节点和边的属性可以是一组整数、向量空间或者是字母符号。在定义当中，没有属性的图是  $L_V$  和  $L_E$  为空集的特殊情况。图的边是通过顶点对  $(u,v)$  的形式给出的，在有向图中  $u \in V$  表示的是源节点，而  $v \in V$  表示的目标节点。通常来说，被边  $(u,v)$  连接起来的两个顶点被称作是相邻的。一个图被称为完全图，如果图中的任意两个顶点都是相邻的。

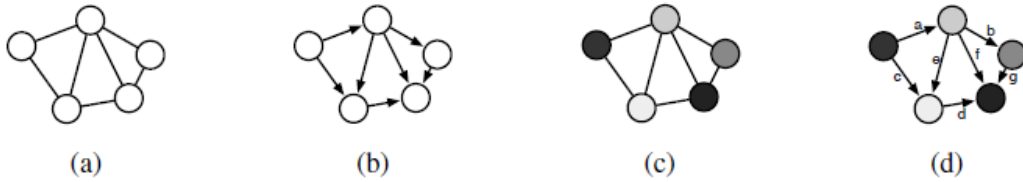


图 3-1 不同类型的图

以图的定义为基础，下面给出子图的定义。

定义：（子图）令  $g_1=(V_1,E_1,\mu_1,\nu_1)$ ,  $g_2=(V_2,E_2,\mu_2,\nu_2)$ ,  $g_1$  是  $g_2$  的子图，表示为  $g_1 \subseteq g_2$ ，如果下列条成立：

- (1)  $V_1 \subseteq V_2$ ;
- (2)  $E_1 \subseteq E_2$ ;
- (3)  $\mu_1(u) = \mu_2(u)$ , 对任意  $u \in V_1$ ;
- (4)  $\nu_1(e) = \nu_2(e)$ , 对任意  $e \in V_1$ .

### 3.1.2 Isomorphism 同构

精确的图匹配算法的目标是判定两张图或其一部分在结构和属性方面均具有完全的一致性。图结构的通用表示方法一般是邻接矩阵。例如图  $g=(V,E,\mu,\nu)$  可表示为  $A=(a_{ij})_{n \times n} (|g|=n)$ 。在矩阵  $A$  中，如果顶点  $v_i$  和顶点  $u_j$  之间存在边  $(v_i, u_j) \in E$ ，那么  $a_{ij}$  的值为 1，否则为 0。

通常来说，一张图的顶点和边是没有严格的次序规定。因此，由  $n$  个顶点构

成的图，有  $n!$  种邻接矩阵的表示形式存在。那么，判定两张图结构的一致性，不能简单的通过比较两张图的邻接矩阵来实现。图  $g1$  和  $g2$  的一致性通常由一个映射函数来建立，被称作 Graph Isomorphism，将图  $g1$  映射到  $g2$ 。

定义：（Isomorphism 同构）令  $g1=(V_1,E_1,\mu_1,v_1)$ ， $g2=(V_2,E_2,\mu_2,v_2)$ 。Graph Isomorphism 类型的同构是一个双射， $f: V_1 \rightarrow V_2$ ，当且仅当下列条件成立：

- (1)  $\mu_1(u) = \mu_2(f(u))$ ，对任意  $u \in V_1$ ；
- (2) 对任意  $e_1 = (u,v) \in E_1$ ，存在边  $e_2 = (f(u),f(v)) \in E_2$  且  $v_1(e_1) = v_2(e_2)$ ；
- (3) 对任意  $e_2 = (u,v) \in E_2$ ，存在边  $e_1 = (f^{-1}(u),f^{-1}(v)) \in E_1$  且  $v_1(e_1) = v_2(e_2)$ 。

从定义中可以看出，Isomorphism 图的同构要求两张图在属性和结构上具有完全的一致性，也就是说对于第一张图的每一个节点和第二张图的每一个节点之间要具有一对一的对应关系，同时保证边结构保留完整，以及顶点和边的属性一致。

图 3-2 中的两张图 G 和 H 就是 Isomorphism 同构的两张图，虽然两图直观上并不相同，当经过映射之后，两张图的结构完全一致的。

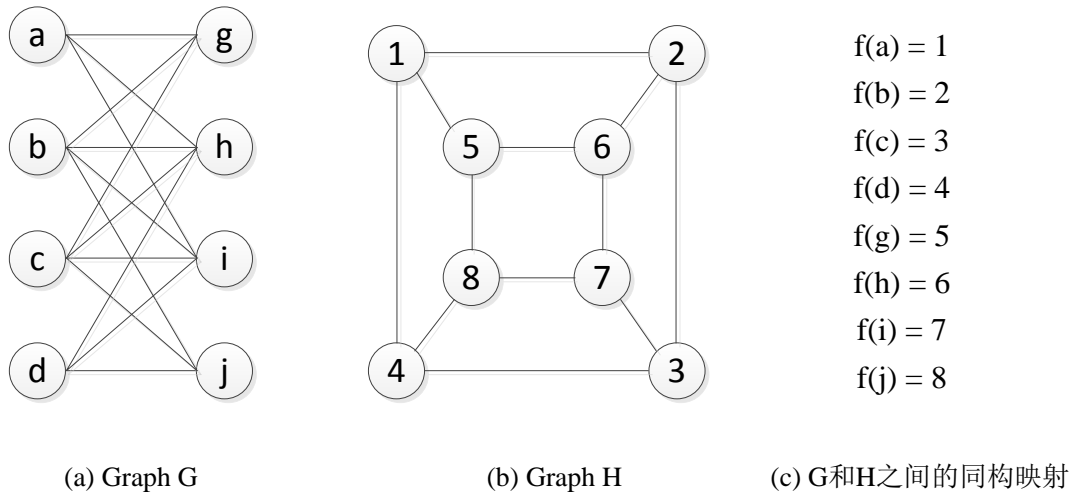


图 3-2 Isomorphism 同构

在计算复杂性中 Graph Isomorphism 本身是一个 NP 问题。目前为止，对于一般性的图还没有人提出多项式时间的算法可以解决 Graph Isomorphism，同时也不能证明其为 NPC 问题<sup>[37]</sup>。

也就是说，在最坏的情况下，目前针对 Graph Isomorphism 可行的算法，都是指数级的计算复杂度，与两图的顶点个数相关。然而实际应用中，大多数的场景都不是最坏的计算复杂度的情况，而且边和节点的属性往往可以降低搜索的复杂性，因此实际的时间开销是可以接受的。对于一些特殊的图，也有很多多项式时间的算法被提出。特殊的图包括树、有序图、平面图、节点具有唯一性属性的图

等等。

大部分针对图精确匹配的算法都是基于带回溯的树搜索策略。算法的基本思想是部分匹配，开始的时候是空集，然后迭代的向里面加入配对的节点。加入的节点要满足一些必要条件保证能够根据已经配对的节点匹配类型的兼容，通常也会使用一些启发式的条件来是尽早减掉不合适的搜索分支。最终，算法要么是找到了一个完整的匹配，要么是因为匹配条件的限制无法再加入新的匹配对。对于后者发生的情况，算法开始回溯，去除最后加入的一对节点直到找到可选的替代方式进行扩展。如果满足限制条件的所有匹配都已经试过，那么算法就会停止。这类算法已经有几个不同的实施策略被实现，他们的区别在于访问分支的顺序。其中最简单的方式是深度优先搜索，它比其它的搜索方式需要的内存更少，并且很易于递归设计，这就是著名的分支限界法。这类算法有一个很好的特点：它能够很容易适应边和节点带属性的匹配，并且对属性的类型没有限制。这对模式识别的应用非常有用，因为在应用中会经常利用属性的限制来大幅缩短匹配的时间。

### 3.1.3 Isomorphism 子图同构

定义：（Isomorphism 子图同构）令  $g_1=(V_1, E_1, \mu_1, v_1)$ ,  $g_2=(V_2, E_2, \mu_2, v_2)$ , 从  $g_1$  到  $g_2$  的一个单射  $f: V_1 \rightarrow V_2$  称作是 Isomorphism 子图同构，当且仅当存在子图  $g \subseteq g_2$  使得  $f$  是  $g$  与  $g_1$  的一个 Isomorphism 同构。

针对 Isomorphism 同构的算法，基于树搜索策略，还有基于决策树的算法都可以被用于解决 Isomorphism 子图同构问题<sup>[38]</sup>。和 Isomorphism 同构问题不同的是，Isomorphism 子图同构问题是一个 NP 完全问题<sup>[39]</sup>。事实上，Isomorphism 子图同构问题要比 Isomorphism 同构问题更难解决，因为 Isomorphism 同构问题只需要判断  $g_1$  和  $g_2$  是否同构，而 Isomorphism 子图同构问题需要判断  $g_1$  是否与  $g_2$  中某个和  $g_1$  大小相同的子图是 Isomorphism 同构的。

### 3.1.4 Monomorphism 同构

Monomorphism 同构是也是子图同构的一种形式，与 Isomorphism 子图同构相比，它同构的条件更为宽松。Monomorphism 同构允许这样的情况存在：在基图中存在的某条边，而在模式图中不存在。这在 Isomorphism 子图同构是不允许的。下面来看其严格定义。

定义：（Monomorphism 同构）令  $g_1=(V_1, E_1, \mu_1, v_1)$ ,  $g_2=(V_2, E_2, \mu_2, v_2)$ 。Monomorphism 同构是一个单射， $f: V_1 \rightarrow V_2$ ，当且仅当下列条件成立：

- (1)  $\mu_1(u) = \mu_2(f(u))$ , 对任意  $u \in V_1$ ;
- (2) 对任意  $e_1 = (u, v) \in E_1$ , 存在边  $e_2 = (f(u), f(v)) \in E_2$  且  $v_1(e_1) = v_2(e_2)$ 。

条件 (1) 保证的是对应的顶点具有一致的属性。这里要注意的是条件 (2), 它要求在在模式图  $g_1$  存在的边在基图中必须存在, 而且边属性也要保持一致。但是并没有要求在基图  $g_2$  中存在的边在  $g_1$  中也要存在, 但是 Isomorphism 子图同构却要求两条边同时存在。因此说 Monomorphism 同构是比 Isomorphism 子图同构条件更弱。

## 3.2 图同构算法

### 3.2.1 图同构匹配算法分析

在各种同构模式中, 除了 Graph Isomorphism 之外都是 NP 完全问题。现在还没有证明 Graph Isomorphism 是否为 NP 完全问题<sup>[40]</sup>, 目前多项式时间的算法都是针对特殊类型的图进行匹配, 还没有针对一般图的通用多项式时间的算法。因此精确图匹配算法在最坏情况下时间复杂度是指数级别的。然而在实际问题中, 所用的时间基本上是可以接受的。有两个原因: 一是实际问题中遇到的图的类型是往往不是最差的情况; 二是节点和边带有属性, 使用这些属性可以大大降低搜索的时间。

图的同构匹配问题是图论中一个非常经典的问题, 在不同的应用场景下使用的算法也不尽相同。在上面提到的匹配问题中, Graph Isomorphism 是很少被使用的, 因为在图的建立过程中, 获得的数据不可避免的会出现一些干扰, 这样得到的图就会失去一些节点和边, 或者增加一些额外的节点和边。Subgraph Isomorphism 和 Monomorphism 是经常使用的方式, 它们在实际问题中往往更加有效, 对于这两个问题已经有了许多成型的算法。目前精确的图匹配算法对于节点数目较少的情况较为有效, 寻找 MCS (Max Common Subgraph) 目前受到越来越多的关注。

#### 3.2.1.1 Ullmann 算法

图匹配算法中最重要的算法之一就是 1976 年提出的 Ullmann 算法<sup>[41]</sup>。虽然它被提出的时间很早, 但目前仍然是广泛使用的同构算法之一。它可以解决包括 Isomorphism, Subgraph Isomorphism 和 Monomorphism 在内的同构问题。同时算法的作者还提供了一种方法来处理最大匹配检测, 因此它也能够用来解决 CMS 问题。为了减掉不好的匹配分支, Ullmann 算法中提出了预测方程, 以此来控制回溯过程, 显著降低了搜索空间的规模, 提高了算法性能。

#### 3.2.1.2 Ghahraman 算法

另一个基于回溯的 Monomorphism 算法是 Ghahraman 在 1980 年提出的<sup>[42]</sup>。为了减小搜索空间，文中使用了一种类似于关联图的技术。匹配的搜索是在 netgraph 矩阵上进行的。这个矩阵是由被匹配的两个图的节点之间的笛卡尔乘积产生的。两张图的 Monomorphism 匹配和 netgraph 的某个子图相关。作者找到了部分匹配能够产生最终匹配结果的两个必要条件。从这些条件中衍生出两个算法的版本。算法的一个主要缺点是，netgraph 的存储至少需要一个  $N^2 \times N^2$  大小的矩阵， $N$  代表的是较大的图的节点数目。因此，这种算法比较适合节点数目较少的图。

#### 3.2.1.3 VF 与 VF2 算法

一个较新的能够同时适用于 Isomorphism 和 subgraph Isomorphism 是 VF 算法<sup>[43]</sup>，算法的提出者是 Cordella。作者定义了一种启发式的搜索条件，基于对已经配对节点的临近节点的分析。这种启发式条件在很多情况下比 Ullman 和其他算法有显著的改善。在 2002 年作者的一篇文章中，对这个算法进行了改进，称为 VF2 算法<sup>[44]</sup>。VF2 将空间复杂度从  $O(N^2)$  降到了  $O(N)$ ， $N$  是图中节点的数目。这样使该算法能够适用于大图的匹配。

#### 3.2.1.4 Nauty 算法

在不基于回溯的树搜索算法之中最著名的就是 Nauty 算法<sup>[45]</sup>。Nauty 算法只能处理 Isomorphism 同构问题，被认为是目前处理 Isomorphism 问题最快的算法。它是基于群论实现的。特别是，它利用群论的结论有效的创建了每个输入图的自同构群。一个自同构群会产生一个规范标签，这样就为每一个自同构群的等价类引入了唯一的节点顺序。两个图的同构比较就变成了规范标签的邻接矩阵的比较。比较的时间复杂度为  $O(N^2)$ ，但是建立规范标签在最坏的情况下是指数级别的时间开销。在大多数情况下，Nauty 算法的时间性能表现是可以接受的。由于规范标签的建立是可以独立进行，与被匹配的图无关，因此它比较适合一个图在一个规模较大的图库内的比较，因为图库的规范标签是可以提前建立的。

### 3.2.2 图同构匹配算法性能对比

现在与本文提出的问题背景相结合，对这几个经典的算法进行分析，选择出合适对象引用关系图的同构算法。3.2 节提出的算法针对不同类型的图有各自的特点与优势。图的主要类型有界价图 (bounded Valence Graph)、二维网格图(2D Mesh Graph)和随机连接图(Randomly Connected Graph)。对这三种不同类型的有向图，Foggia<sup>[46]</sup>对以上算法进行了测试与分析。对象引用关系图与随机连接图较为相似，与另外两种有规律的图差别较大。因而只考虑对随机连接图的分析。Foggia 在实

验中用使用多组数据进行测试和评估，其中包括不同的节点数以及不同的边密度。实验结果显示，对于随机连接图，Ullmann 算法不如 VF (2) 算法和 Nauty 算法。在图的顶点数目和边密度不同的情况下，VF2 优于 VF 算法。VF2 算法与 Nauty 算法相比较，在图的规模较小，边较为稀疏的情况下，其效果更好。Nauty 算法对于规模较大的稠密图有更好的处理能力。

本文针对的图同构问题是对象引用依赖关系图之间的子图匹配。关系图中，一个节点代表一个类，一条有向边代表着一个类对另一类的引用。根据对样本分析，对象引用关系图的节点数基本在 100 个以内，且类之间的引用关系并不频繁，是规模较小的稀疏矩阵。基于以上分析，本文采用的算法以 VF2 算法为基础。

### 3.3 VF2 算法实现与分析

#### 3.3.1 基于状态表示的 VF2 算法的策略

VF2 算法同时适用于 Isomorphism, Subgraph-Isomorphism 和 monomorphism 匹配模式。VF2 算法具有普遍的有效性，因为它没有对图的拓扑结构有任何要求。算法在匹配的过程中引用状态空间的概念 (SSR)，同时也提出了 5 个可行的规则进行剪枝来缩小搜索空间。和 VF2 算法的前一版本 VF 算法相比，最显著的改善是在便利搜索空间的过程中，采用的数据结构明显降低了内存需求，这使得算法可以适用于上千个节点的图。

下面我们来介绍算法的主要思想。假设已知图  $G1(N1, B1)$  和  $G2(N2, B2)$  现在我们要寻找这两图之间的同构映射  $M$ 。通常映射  $M$  被描述为节点对  $(n, m)$ ，表示的是图  $G1$  中的节点  $n$  和图  $G2$  中的节点  $m$  之间的对应关系。

寻找映射  $M$  的过程是通过状态空间表示 (SSR) 来描述的。匹配过程中的每一个状态  $s$  都是一个局部映射，用  $M(s)$  来表示。 $M(s)$  是  $M$  的一个子集。 $G1(s)$  表示映射  $M(s)$  与  $G1$  相关的子图， $G2(s)$  表示映射  $M(s)$  与  $G2$  相关的子图。 $N1(s)$  和  $N2(s)$  分别表示  $G1(s)$  和  $G2(s)$  中的顶点集， $E1(s)$  和  $E2(s)$  分别表示  $G1(s)$  和  $G2(s)$  中的边集。

在图 3-3 中给出了两张图  $G1$  和  $G2$ ，通过实例来说明 SSR 及其它基本概念。 $G1$  和  $G2$  同构映射为  $M$ ，中间状态为  $sp$ 。

$$M = \{(A1, B2), (A2, B1), (A3, B3), (A4, B6), (A5, B4), (A6, B5)\}$$

$$M(sp) = \{(A1, B2), (A2, B1), (A3, B3), (A4, B6)\}$$

$$V1(sp) = \{A1, A2, A3, A4\}$$

$$V2(sp) = \{B2, B1, B3, B6\}$$

$$E1(sp) = \{\langle A1, B2 \rangle, \langle A2, B3 \rangle, \langle A3, B4 \rangle\}$$



$$E2(sp) = \{ \langle A2, B1 \rangle, \langle A1, B3 \rangle, \langle A3, B6 \rangle \}$$

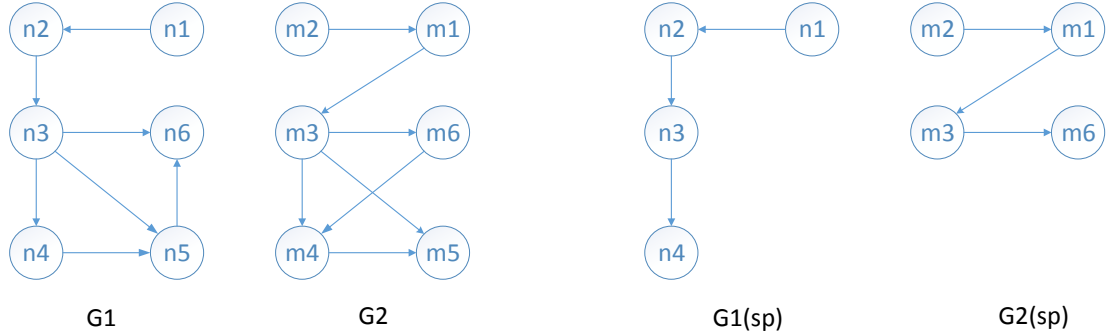


图 3-3 SSR 实例图

VF2 算法在匹配过程中会产生多个状态，从一个状态  $s$  转换为另一个状态  $s'$ ，实际上在  $s$  的基础上加入一对新的匹配节点。通过加入不同的节点对，状态  $s$  会转换为多个状态。这样，不断产生的新的状态空间可以使用树的结构来描述。父亲节点表示原来的状态，孩子节点表示加入新节点后产生的新状态。以图 3-3 所示的  $sp$  状态为例，当加入节点对  $(A5, B4)$  之后，状态转换为  $sq$ ，如图 3-4 所示。在 (a) 中可以看出，加入节点对  $(A5, B4)$  转换为  $sq$  状态只是多种可能性之一，还可以加入其他的节点对转换为状态  $sr, ss, st$  等等，这就需要回溯来选择合适的转换状态。从图 (b) 可以看出，加入  $(A5, B4)$  成功转换为新的状态  $sq$ 。

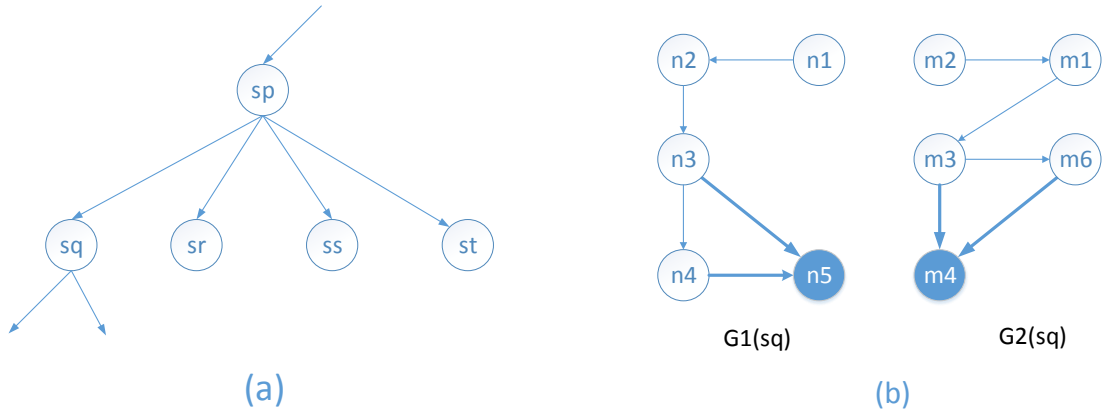


图 3-4 SSR 状态转换图

这样可以通过搜索所有的 SSR 来找到最终的同构映射  $M$ 。但是在搜索过程中，VF2 算法引入了一些规则，以通过剪枝来缩小搜索空间，降低时间复杂度。每个状态  $s$  可以加入的节点对是有限的，而且并不是所有的节点对都需要一一搜索。根据引入的规则可以去除一部分节点对，剩余的节点对则是需要遍历继续搜索的，这些节点对称作是状态  $s$  的候选节点对集，记为  $H(s)$ 。算法 3-1 是对 VF2 算法的详细描述。

VF2 算法采用的深度优先搜索方式，对于每一个节点，首先计算出该节点对

应的中间状态  $s$  的候选节点对，然后递归的对每个节点对进行搜索。候选节点对集合  $H(s)$  的计算是 VF2 算法的难点。

---

**算法 3-1: VF2 算法**

---

输入:  $G1, G2$ , 状态  $s$ , 初始的状态为  $s0$ ,  $M(s0)$  为空集

输出: 两图的同构映射  $M$

```

01  PROCEDURE  VF2Match( $s$ )
02      IF   $|M(s)| = |G2|$   THEN
03          Successful Match
04      ELSE
05          Find  $H(s)$  which is the set of possible pairs for  $M(s)$ 
06          FOREACH   $h$   in   $H(s)$ 
07              IF  all rules are satisfied for  $h$  added to  $M(s)$   THEN
08                   $s' =$  put  $h$  into  $M(s)$ 
09                  CALL VF2Match( $s'$ )
10              ENDIF
11          ENDFOREACH
12      Restore data
13      ENDIF
14  END  PROCEDURE  VF2MATCH

```

---

先给出以下几个定义。

(1)  $T1^{out}(s)$  : 表示的是  $G1$  中一个顶点集合, 集合中的顶点不属于  $G1(s)$ , 但是  $G1(s)$  中顶点的后继结点。

(2)  $T2^{out}(s)$  : 表示的是  $G2$  中一个顶点集合, 集合中的顶点不属于  $G2(s)$ , 但是  $G2(s)$  中顶点的后继结点。

(3)  $T1^{in}(s)$  : 表示的是  $G1$  中一个顶点集合, 集合中的顶点不属于  $G1(s)$ , 但是  $G1(s)$  中顶点的前驱结点。

(4)  $T2^{in}(s)$  : 表示的是  $G2$  中一个顶点集合, 集合中的顶点不属于  $G2(s)$ , 但是  $G2(s)$  中顶点的前驱结点。

$H(s)$  选取的步骤如下所示:

- (1) 如果  $T1^{out}(s)$  和  $T2^{out}(s)$  不是空集, 那么  $P(s) = T1^{out}(s) \times T2^{out}(s)$ ;
- (2) 如果  $T1^{out}(s)$  和  $T2^{out}(s)$  都是空集, 但是  $T1^{in}(s)$  和  $T1^{in}(s)$  不是空集, 那么

$$P(s) = T1^{in}(s) \times T2^{in}(s);$$

(3) 如果  $T1^{out}(s)$ ,  $T2^{out}(s)$ ,  $T1^{in}(s)$  和  $T2^{in}(s)$  都是空集, 那么  $P(s) = (V1-V1(s)) \times (V12-V2(s))$ ;

(4) 其他情况, 对状态  $s$  进行剪枝。

情况 4 是指如果  $T1^{out}(s)$  和  $T2^{out}(s)$  之一为空集, 或者  $T1^{in}(s)$  和  $T2^{in}(s)$  之一为空集, 那么就表明状态  $s$  不能发展为最后的匹配映射, 因而可以将状态  $s$  的搜索剪枝。

对于当前状态  $s$ , 针对候选的节点对  $(m,n)$ , 需要判断其加入的可行性。VF2 算法中对节点对的判断是通过可行性函数  $F(s,n,m)$  来实现的。其中  $s$  表示当前状态,  $n$  表示图  $G1$  的一个顶点,  $m$  表示图  $G2$  的一个顶点。  $F(s,n,m)$  的返回值为布尔型, 返回值为 **true** 则表示节点对可行。返回值为 **false** 则表示节点对不可行。如果判断出节点对不可行, 则剪枝去掉该搜索路径。

可行性规则分为语法可行性规则和语义可行性规则。语法可行性规则针对的是图的拓扑结构, 而语义可行性规则针对的是图的顶点和边的属性。由于本文研究的对象引用关系图不带有边和顶点的属性。因此, 只考虑语法可行性规则, 用  $F_{syn}(s,n,m)$  来表示。VF2 算法中定义了 5 个语法可行性规则。其中  $R_{pred}$  和  $R_{succ}$  考虑的是当前状态  $M(s)$  加入候选节点对  $(m,n)$  转换为  $s'$  后一致性问题而  $R_{in}$ 、 $R_{out}$  和  $R_{new}$  考虑的是对搜索空间的剪枝。

$$F_{syn}(s,n,m) = R_{pred} \wedge R_{succ} \wedge R_{in} \wedge R_{out} \wedge R_{new}$$

用  $Pred(G,n)$  代表图  $G$  中  $n$  的前驱节点的集合, 用  $Succ(G,n)$  表示图  $G$  中  $n$  的后继节点的集合。同时使用  $T1(s) = T1^{in}(s) \vee T1^{out}(s)$ , 并使用  $N'(s) = N1(s) - M1(s) - T1(s)$ 。  $T2(s)$  和  $N2'(s)$  的定义类似。

规则 1:  $R_{pred}(s,n,m)$

$$(\forall n' \in M1(s)) \cap Pred(G1,n) \exists m' \in Pred(G2,m) | (n',m') \in M(s)) \wedge$$

$$(\forall m' \in M2(s)) \cap Pred(G2,n) \exists n' \in Pred(G1,n) | (n',m') \in M(s))$$

规则 2:  $R_{succ}(s,n,m)$

$$(\forall n' \in M1(s)) \cap Pred(G1,n) \exists m' \in Succ(G2,m) | (n',m') \in M(s)) \wedge$$

$$(\forall m' \in M2(s)) \cap Pred(G2,n) \exists n' \in Succ(G1,n) | (n',m') \in M(s))$$

规则 3:  $R_{in}(s,n,m)$

$$(Card(Succ(G1,n) \cap T1^{in}(s)) \geq Card(Succ(G2,m) \cap T2^{in}(s))) \wedge$$

$$(Card(Pred(G1,n) \cap T1^{in}(s)) \geq Card(Pred(G2,m) \cap T2^{in}(s)))$$

规则 4:  $R_{out}(s,n,m)$

$$(\text{Card}(\text{Succ}(G1, n) \cap T1^{\text{out}}(s)) \geq \text{Card}(\text{Succ}(G2, m) \cap T2^{\text{out}}(s))) \wedge$$

$$(\text{Card}(\text{Pred}(G1, n) \cap T1^{\text{out}}(s)) \geq \text{Card}(\text{Pred}(G2, m) \cap T2^{\text{out}}(s)))$$

规则 5:  $R_{\text{new}}(s, n, m)$

$$(\text{Card}(N1'(s) \cap \text{Pred}(G1, n)) \geq \text{Card}(N2'(s) \cap \text{Pred}(G2, n))) \wedge$$

$$(\text{Card}(N1'(s) \cap \text{Succ}(G1, n)) \geq \text{Card}(N2'(s) \cap \text{Succ}(G2, n)))$$

以上 5 条规则是针对 Subgraph-Isomorphism 的。如果是 Isomorphism，则前两条规则  $R_{\text{pred}}$  和  $R_{\text{succ}}$  不变，而  $R_{\text{in}}$ 、 $R_{\text{out}}$  和  $R_{\text{new}}$  中的“ $\geq$ ”要变为“ $=$ ”。当新加入的节点对符合这 5 条可行性规则后，则继续对新加入节点对后的状态进行搜索，否则就进行剪枝，不加入该节点对。通过这 5 条规则，可以降低搜索空间的大小，提高算法的执行效率

### 3.3.2 引入精确度控制参数的 VF2 算法的实现

VF2 算法针对的是 Isomorphism 和 Subgraph-Isomorphism。但是针对本文研究对象引用依赖关系图，即使是 Subgraph-Isomorphism 匹配，在实际情况中，模式图也很难与原图的子图相匹配。主要原因是，带有恶意代码的正常程序启动后，在进程中运行的时间不够充分，导致恶意代码对象的创建和对象之间的引用不完全，从而致使源图中的特征不足，可能无法与恶意代码的特征的模式图完全匹配。因此 Subgraph-Isomorphism 在实际中并不实用。为此需要对算法进行调整，放松匹配条件使其适合对象引用关系图的匹配。

通过对 VF2 算法运行过程的分析，可以发现其匹配的过程是在现有的状态  $s$  上加入新节点对来不断回溯，直到找到合适的匹配。也就是说算法的终止条件是状态  $s$  覆盖了模式图中所有的顶点。为了放松匹配条件，不要求模式图中所有的顶点都匹配上才终止算法，而是达到模式图定点数量的一定比例即可。数值的大小作为算法的输入参数，由使用者自己确定。这个新引入的输入参数为  $\lambda$ ， $\lambda$  的取值范围是 (0,1] 之间的任意小数。其表示的含义是当模式图中匹配上的顶点数量与模式图中总数量之比大于等于  $\lambda$  时终止算法，表示匹配成功。如果遍历完所有的搜索空间仍然无法匹配，则表示匹配失败。

为此，需要对原始的 VF2 算法进行修改，增加判断条件，改变算法的终止条件。称改变后的 VF2 算法为  $\lambda$ -VF2 算法。原始的 VF2 算法结束条件为“ $M(s)=|G2|$ ”，将其改为“ $|M(s)| \geq \lambda |G2|$ ”。当条件成立后，即匹配成功。修改后的算法如下所示。

---

**算法 3-2:  $\lambda$ -VF2 算法**

---

输入:  $G1, G2$ , 状态  $s$ , 初始的状态为  $s_0$ ,  $M(s_0)$  为空集,  $\lambda$  为精度控制参数

输出: 两图的同构映射  $M$

```

01  PROCEDURE  VF2Match(s)
02      IF   $|M(s)| \geq \lambda|G2|$  THEN
03          Successful Match
04      ELSE
05          Find  $H(s)$  which is the set of possible pairs for  $M(s)$ 
06          FOREACH  $h$  in  $H(s)$ 
07              IF  all rules are satisfied for  $h$  added to  $M(s)$  THEN
08                   $s' = \text{put } h \text{ into } M(s)$ 
09                  CALL VF2Match( $s'$ )
10              ENDIF
11          ENDFOREACH
12      Restore data
13      ENDIF
14  END  PROCEDURE  VF2MATCH

```

---

### 3.3.3 性能分析

加入  $\lambda$  的 VF 算法, 其时间和空间开销与  $\lambda$  正相关。即随着  $\lambda$  的增加, 算法时间开销和空间开销都会增加。反之亦然。 $\lambda$  为输入参数, 与算法本身无关。在分析的过程中可考虑将其认为是 1, 计算其最坏情况下的开销。

#### 3.5.3.1 时间复杂度

VF2 算法是以状态空间表示为基础的图同构算法。它的时间复杂度的分析包含两部分: 遍历的所有的状态的时间和每个状态处理的时间。

##### (1) SSR 规模

最好的状况是, 所有的状态都只有一个符合规则的节点对可以加入, 即状态的转换是固定的, 算法不需要进行回溯。这样需要处理的总状态数就是图中节点的数目  $N$ 。

最坏的情况是, 对于所有的状态  $s$ , 存在的后继状态均满足条件。这时搜索的状态数量最大。状态搜索树的深度为节点的个数  $N$ 。最坏情况下, 每一个深度的

状态数量是可以计算的。如果深度为  $d+1$ ，那么这一层的状态数量就是  $N(N-1)(N-2)\cdots(N-d)$ 。总数量就是各层状态数量之后：

$$\begin{aligned}
 & 1 + N + N(N-1) + N(N-1)(N-2) + \cdots + N(N-1)(N-2) + \cdots + 2 \\
 &= 1 + \frac{N!}{(N-1)!} + \frac{N!}{(N-2)!} + \frac{N!}{(N-3)!} + \cdots + \frac{N!}{1!} \\
 &= 1 + N! \sum_{d=1}^{N-1} \frac{1}{d!}
 \end{aligned}$$

$$\sum_{d=1}^{N-1} \frac{1}{d!} \text{ 是小于 } 2 \text{ 的, 因此总数量的规模是 } O(N!)$$

### (2) 每个状态的处理时间

每个状态的处理时间包括三个方面：候选状态集合  $P(s)$  的计算时间  $T_p$ ，可行函数  $F(s, n, m)$  的计算时间  $T_F$  以及新状态相关数据的计算时间  $T_{new}$ 。单个状态处理的时间  $T = T_p + T_F + T_{new}$ 。

$T_p$ ：候选状态集的一个节点的检验需要常数时间，节点数目最多为  $N$  个，因此  $T_p$  为  $O(N)$ 。

$T_F$ ：可行函数  $F(s, n, m)$ ，对每条边的处理时间的一个常量，边数在最坏的情况下是节点和剩余其它所有节点都连接，因此  $T_F = O(B)$ 。

$T_{new}$ ：新状态  $s'$  相关集合的计算时间包括  $M(s'), V1^{in}(s'), V1^{out}(s'), V2^{in}(s'), V2^{out}(s')$ 。其中  $M(s')$  只需加入新节点对，时间为常量。而剩余四个集合都需要遍历新加入节点的连接边，最坏情况下为  $O(B)$ 。

$B$  是一个节点的连接边数，顶点个数为  $N$  的有向图，当顶点全部互为前驱和后继的时候，单个顶点的边数达到最大值，为  $2 * (N - 1)$ ，因此  $O(B)$  在最坏情况下为  $O(N)$ 。

综上， $T = T_p + T_F + T_{new} = O(N) + O(N) + O(N) = O(N)$ 。也就是说 VF2 算法对每个状态的处理时间是顶点个数  $N$  的常量倍。

### (3) 最终时间复杂度

经过上述分析，得到了遍历所有的状态和对处理每个状态的时间复杂度，那么 VF2 算法的时间复杂度就是这两部分之积。

最好情况下为： $O(N) * O(N) = O(N^2)$ ;

最坏情况下为： $O(N) * O(N!) = O(N * (N!))$ ;

### 3.5.3.2 空间复杂度

由于 VF2 算法中采取的是数据结构共享的形式，使每个状态需要的存储空间是一个常量。在算法搜索的过程中，采用的是 Depth-First（深度优先）的搜索方式，最大深度不会超过结点个数  $N$ ，VF2 算法在运行过程中需要保存的状态数目也就不会超过  $N$ ，因此需要的空间开销为  $O(N)$ 。

## 3.4 本章小结

本章主要的研究内容是图的同构匹配算法。3.1 节定义了图的基本概念，阐述了图同构的模式及其特点。3.2 节介绍了常用的同构算法，并进行了简单比对，分析了各个算法的优缺点。3.4 节着重介绍了本文中采取的 VF2 算法，分析了其算法流程，适用情况以及性能。最后针对本文的实际情况，对算法进行了修改，加入了参数  $\lambda$  来控制匹配的精确程度。并分析了算法的时间和空间复杂度。

## 第 4 章 检测系统与结果分析

### 4.1 检测系统设计与实现

检测系统主要包括两部分，Android 手机端和服务端。

Android 手机端负责抽取数据提取，然后将数据传递给服务器端。数据的抽取是通过编写程序 `MalwareDetection` 实现的。其主要功能 AHAT 的 `Conventor` 的实现过程已经在第二章第 3 节介绍了。

服务器端有三大功能，建立 ORG 图、建立 ORGB 图以及图的匹配。Android 端提取的数据内容是进程在内存中运行时创建的类及其引用关系。ORG 图，对象引用关系图，是针对所有的类创建的有向图。ORGB 图，对象引用关系图胎记，是只针对恶意代码类创建的有向图，是恶意代码的动态特征。它的建立需要一个恶意代码类的列表来过滤所得到的类。一个恶意代码库就是 ORGB 的集合。而图的匹配就是一个 ORG 在 ORGB 库的检测。

#### 4.1.1 Android 端设计与实现

Android 端模块的主要功能是提取某一个进程的对象引用关系数据。在第二章中已经介绍了 AHAT 和 `Conventor` 两个工具的实现。现通过加入一些其它简单的功能将其整合为以完整的模块 `MalwareDetection`。

`MalwareDetection` 程序是运行在 Android 平台上的手机客户端程序，它的功能是分析当前正在运行的非系统程序，导出其当前时刻的堆信息文件，经过分析之后获取其在运行过程中类与类之间的引用关系。

`MalwareDetection` 主要组成部分有：前台界面、活动进程查询器、Shell 命令执行器、`Conventor`、AHAT，其抽取流程如图 4-1 所示。

(1) 前台界面：显示程序的操作功能。

(2) 活动进程查询器：负责显示当前系统中运行的非系统进程，当在系统中安装了包含恶意代码的应用程序后，它相应的进程就会出现在列表中。

(3) Shell 命令执行器：`am dumpheap` 命令用来导出进程的原始内存文件。Shell 命令执行器调用了此命令完成原始的二进制数据文件导出功能。

(4) `Conventor`：负责二进制文件格式的转换，使 AHAT 工具能够对其分析。

(5) AHAT：二进制的文件数据中包含了大量的内存信息，ORG 图的建立仅仅需要对象之间的引用关系。AHAT 通过对文件的分析，抽取了相应数据，并组织



成新的文本文件。这个文本文件就是服务器端最后需要处理的数据。

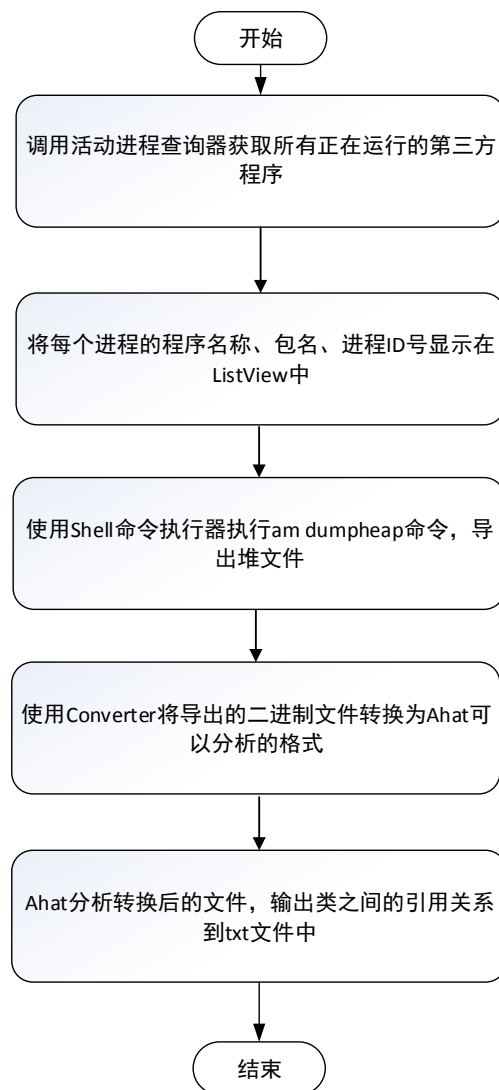


图 4-1 MalwareDetection 检测流程图

#### 4.1.2 服务器端设计与实现

服务器端主要有 3 个模块，建立 ORG 图模块，建立 ORGB 图模块和检测模块。整体架构如图 4-2 所示。图的建立分为两种，一种是针对恶意代码建立特征的 ORGB 图，另一种是针对待检测的应用程序建立的 ORG 图。图建立完毕后以文本文件的格式传送给检测模块。检测模块的功能是实现图的匹配，其核心部分是图的同构算法，VF2 算法。

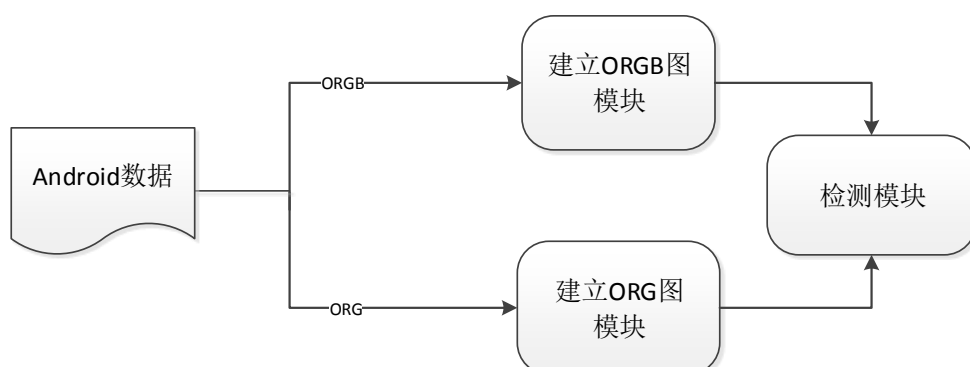


图 4-2 服务器整体架构图

#### 4.1.2.1 Android 数据格式

AHAT 将数据存储于文本文件，存放在二进制文件所在目录下。类是按照包来进行划分的，首先是包名，以 **Package** 开始。包下面是类，每个类以 **class** 开始，跟着是类名，最后是类的 ID。类下面是引用关系，分别是被引用的类和引用的类。引用的类跟在 **Referrers by Type** 后面，每个被引用的类，以类名开始，接着是引用的次数。被引用的类跟在 **Referees by Type** 后面，每个被引用的类，以类名开始，接着是被引用的次数。具体格式如图 4-3 所示。

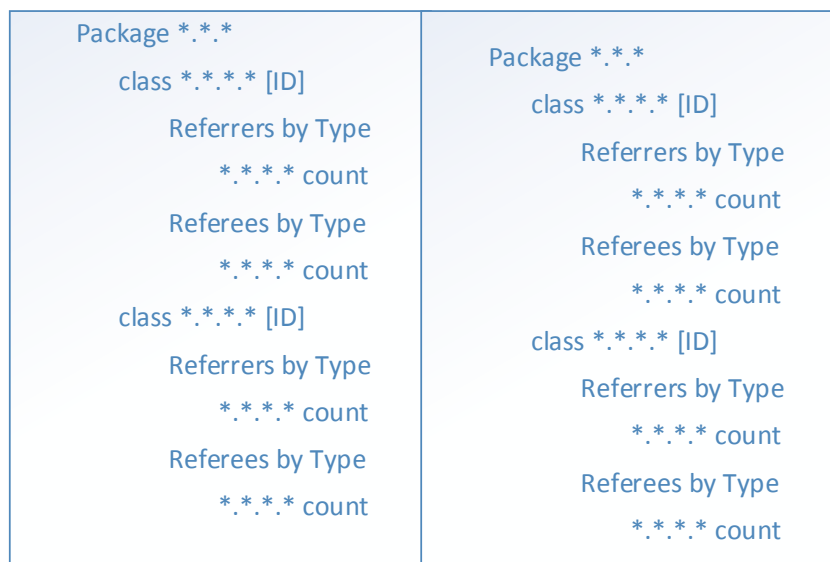


图 4-3 对象引用关系文件格式

#### 4.1.2.2 建立 ORG 图模块

ORG 图的建立是对整个应用程序的非系统类建立的，以数据中的类为节点，以类之间的引用关系直接建立有向图即可。该模块的程序流程图如图 4-4 所示。

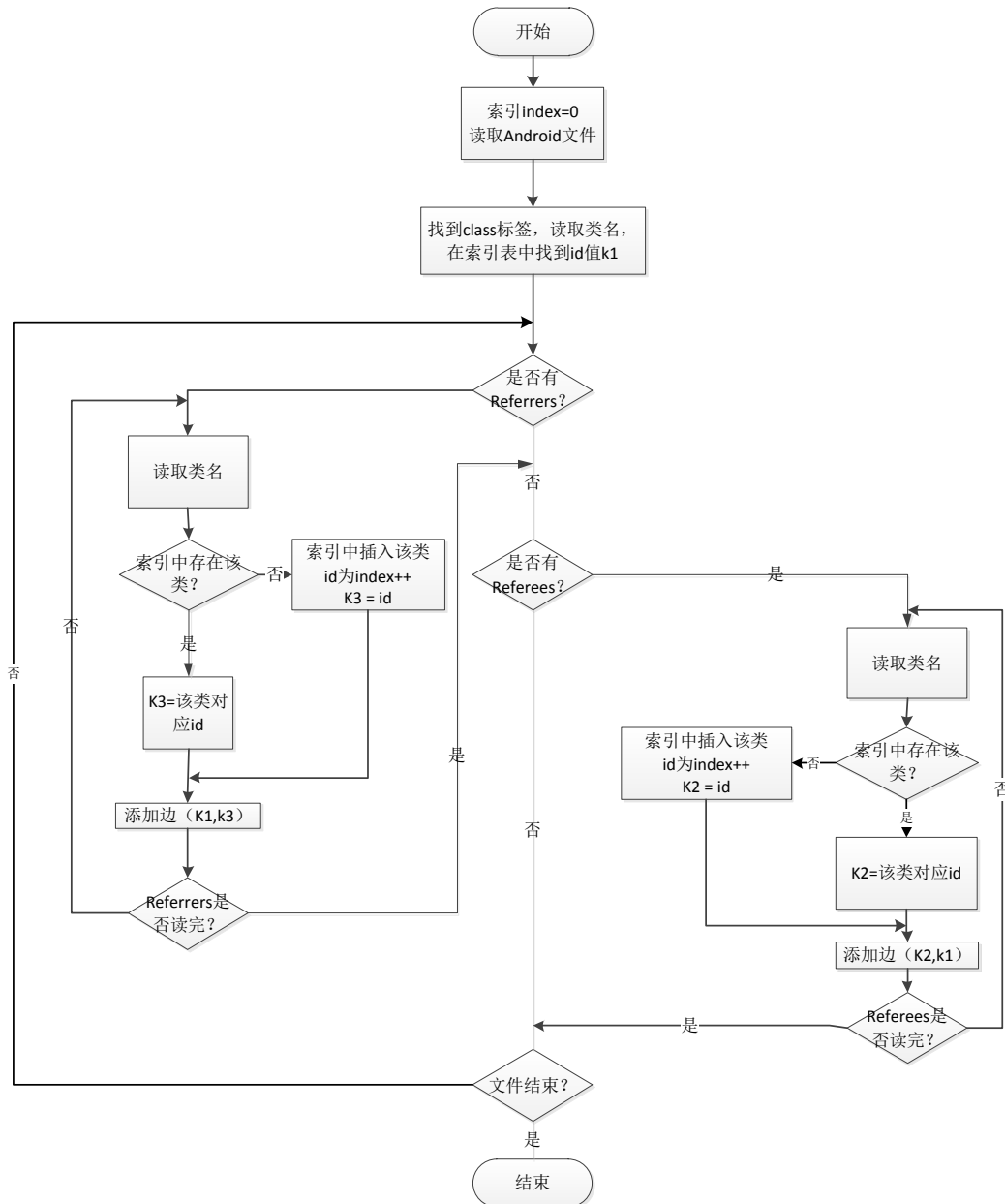


图 4-4 ORG 建立流程图

算法中节点的 ID 为数字, 需要将文件中的类名转 ID。于是建立索引文件, 为每一个类分配 ID, ID 从 0 开始, 每一个新类 ID 值加 1。当程序遇到“class”时, 读取类名。在索引文件中寻找该类。若找到则获取其对应 ID, 否则在索引文件中插入该类, 并以当前索引值加 1 作为其 ID。这个过程相当于在图中添加新节点。当程序遇到“Referrers by type”时, 依此读取所包含的引用类。先执行节点操作, 获取 ID。然后添加从该节点指向当前 class 的边。当程序遇到“Referees by type”时, 依此读取所包含的被引用类。先获取 ID, 然后添加当前 class 指向该节点的边。

#### 4.1.2.3 建立 ORGB 图模块

ORGB 图的建立仅仅选取恶意代码的类作为节点。而恶意代码类列表的获得是通过人工手动分析得到的。建立 ORGB 的流程图如图 4-5 所示。

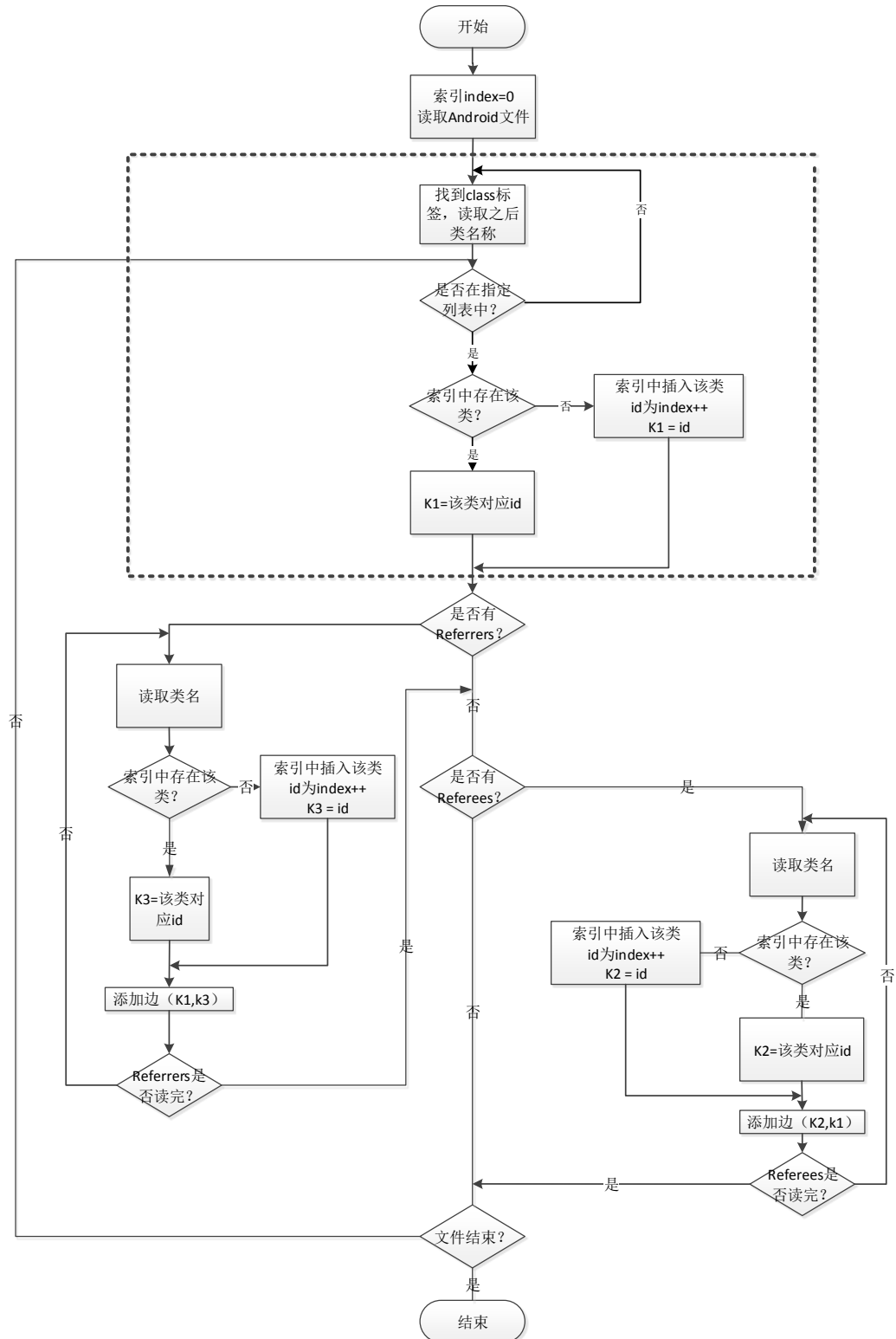


图 4-5 ORGB 建立模块

#### 4.1.2.4 检测模块

检测中使用匹配算法是修改的 $\lambda$ -VF2算法,当 $\lambda$ 值为1时, $\lambda$ -VF2算法就退化为原始的VF算法。实验过程中,通过设定 $\lambda$ 为不同的值来产生不同的结果进行比较。检测模块的程序流程如图4-6所示。程序首先设定 $\lambda$ 值,然后载入ORG图,接着依次载入ORGB库中的图,进行比较。一旦匹配成功,则终止程序,输出匹配结果。如全部匹配失效,则表示未检测出恶意代码。ORG及ORGB的图都是采用二进制文件存储,不记录节点和边的属性,适合实验的情况。

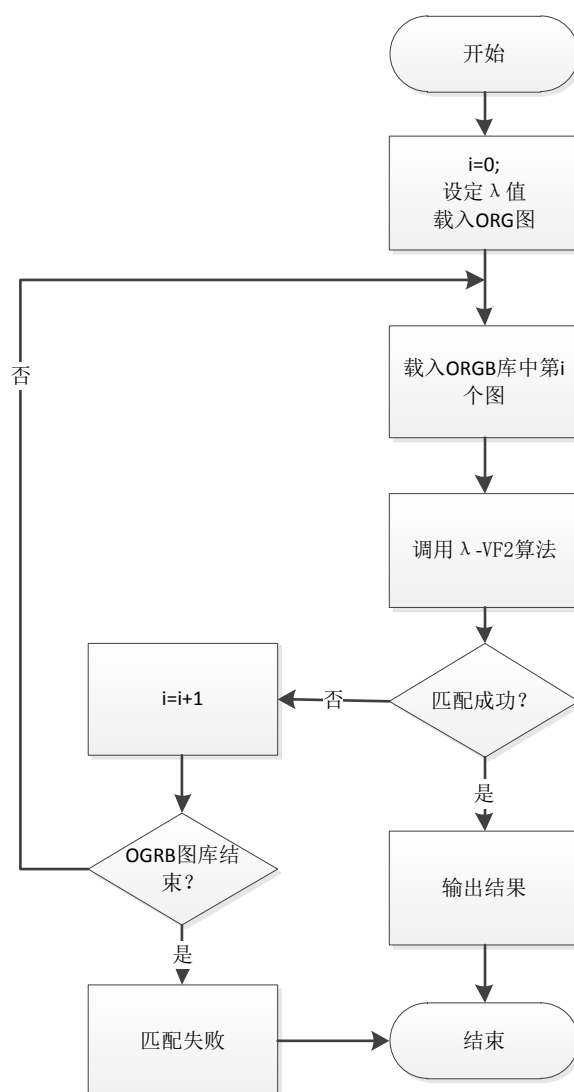


图 4-6 检测过程模块

## 4.2 测试环境

实验过程中，在 Android 端运行 Android 应用程序，并使用自制的工具提取原始数据。在服务器端建立 ORG 图，组织样本库进行比对工作。

### 4.2.1 Android 端运行环境

内存数据的提取是在实际的 Android 开发机上完成的。表 4-1 列出了 Android 端具体测试环境。

表 4-1 Android 端测试环境

设备名称	Anroid 系统版本	CPU	手机内存
Galaxy nexus 3	Android4.1.2	德州仪器 OMAP4460，双核， 频率 1228MHz	1GB

首先需要将样本程序安装到开发机上，然后使样本程序运行，最后使用在 Android 端实现的工具对样本程序的进程抽取内存文件。

### 4.2.2 PC 端运行环境

Android 端将 ORG 图上传给服务器，匹配工作是在服务器端完成的，表 4-2 列出了服务器端的测试环境。

表 4-2 PC 端测试环境

操作系统	CPU	内存
Windows 7	Xeon E3-1200 v2，4 核 8 线程，3300MHZ	1GB

## 4.3 数据来源与整理

实验采用的数据有两类：手动编写的模拟样本和真实的恶意代码样本。

### 4.3.1 模拟恶意代码样本

对于模拟的测试样本，每一个手动编写的测试样本在代码结构上分为两个包。一个包里的类代表正常应用程序的代码，另一个包的类代表恶意代码。每一类模拟样本中，不同的测试样本中代表恶意代码的包是相同的，而代表正常应用程序的类不同。这样就模拟了一类感染相同恶意代码的应用程序。模拟样本的好处是，可以控制恶意代码的规模和运行，方便清晰的分析结果。实验中，精心编写了一组模拟样本，共 10 个。这 10 个样本的恶意代码类型是基本相同的，但是为了测

试出 VF2-Isomorphism , VF2-Monomorphism ,  $\lambda$ -VF2-Isomorphism ,  $\lambda$ -VF2-Monomorphism 不同的效果, 对其中的恶意代码部分进行了调整, 模拟了恶意代码反检测的攻击。各种类型模拟样本的数量见表 4-3。

表 4-3 模拟样本数量

Origin	Extra Reference	Extra Class	Class Replacement
4	2	2	2

(1) Origin 样本: 原始的恶意代码样本, 恶意代码部分完全相同, 正常 APK 程序不同。

(2) Extra Reference 样本: 恶意代码的类相同, 但是和原始恶意代码相比, 类之间增加了新的引用关系, 模拟恶意代码在类之间增加没有意义的无效引用以逃避检测的情况。

(3) Extra Class 样本: 在原始恶意代码的类的基础之上, 增加了新的类。模拟恶意代码增加新的功能, 发生变种的情况。

(4) Class Replacement 样本: 模拟恶意代码变种后, 删除一部分类, 增加一部分类的情况。

### 4.3.2 真实恶意代码样本

另一类是真实的恶意代码样本。只有当含有恶意代码的应用程序在运行的时候才能够获取其对象引用关系图。本文搜集了两类带有恶意代码的 Android 的应用程序安装包。见表 4-4。恶意代码嵌入在 APK 安装文件中, 程序安装后, 恶意代码依附于应用程序进程一同运行。

表 4-4 真实恶意代码样本

ADRD	Bgserv
22	16

为了提取同一类恶意代码的对象引用关系图胎记作为该类恶意代码的动态特征, 从每一类恶意代码中随机选取几个样本, 对其进行手动分析。

APK 文件是经过 dx 工具打包生成的, 通过使用 dex2jar 工具可以将打包的 APK 文件还原为之前的 JAVA 文件。在经过 JD 反汇编工具就可以反编译 JAVA 的 class 文件, 这时就能看到 apk 应用程序的类了。通过比对, 分析出其中的恶意代码类。在 APK 安装文件中, 恶意代码的类一般是存在于单独的包中, 这使得手动识别恶意代码类成为了可能。图 4-7 中显示了两个含有 ADRD 恶意代码的 APK 反汇编出代码结构, 通过比较发现, 两个 APK 含有恶意代码的类包 “xxx.yyy”, 由此可

以确定 ADRD 恶意代码类的列表。

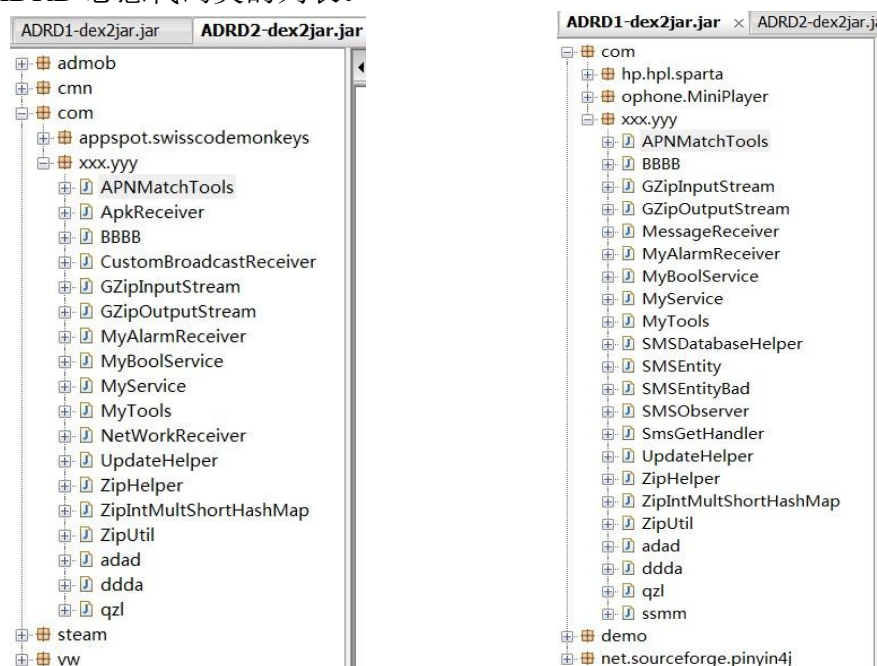


图 4-7 恶意代码类比较

## 4.4 模拟样本实验结果

### 4.4.1 模拟样本检测结果与分析

实验过程中首先从 Origin 样本中建立原始恶意代码的 ORGB，作为恶意代码的识别特征。然后对 10 个样本一一建立完整 ORG 图。最后分别使用了 4 种类型的 VF2 算法在所有 ORG 中对 ORGB 进行了检测，实验结果如表 4-5 所示。

实验中  $\lambda$  取值 0.8。

表 4-5 模拟样本检测结果

算法名称	Origin	Extra Reference	Extra Class	Class Replacement
VF2 Subgraph Isomorphism	4/4	0/2	2/2	0/2
VF2 Monomorphism	4/4	2/2	2/2	0/2
$\lambda$ -VF2 Subgraph Isomorphism	4/4	1/2	2/2	1/2
$\lambda$ -VF2-Monomorphism	4/4	2/2	2/2	2/2

从表中可以看到，所有算法能够检测出所有原始恶意代码和在恶意代码的基础上增加新类的情况。这是因为在恶意代码中增加的新类导致 ORG 图中只是增加了新的节点，原始恶意代码 ORGB 依然是 ORG 的一个子图，因而都可以检测出来。这表明，当恶意代码增加了新的类发生变种的时候，对象引用关系图方法依然有



效。

VF2-Subgraph Isomorphism 算法无法检测出 Extra Reference 类型的攻击。这是因为这种攻击在恶意代码类之间增加了一些没有意义的引用，造成 ORG 图中恶意代码部分增加了新的边，而 Subgraph Isomorphism 要求边是一一对应的，即 ORG 和 ORGB 中边要同时存在，所以匹配失效。

注意到  $\lambda$ -VF2 Subgraph Isomorphism 算法对于 Extra Reference 和 Class Replacement 的检测结果是不完全的。原因是新增加的引用关系只能部分的通过不精确匹配消除掉，不能完全不受影响，这是算法匹配条件过强的结果。

$\lambda$ -VF2-Monomorphism 匹配的条件最弱，在以上四种检测中都成功匹配了。在实际的恶意代码检测中，同一类恶意代码并不完全相同，而且在运行过程中创建的对象也会有所差别，考虑到这些因素以及恶意代码的变种和防检测攻击， $\lambda$ -VF2-Monomorphism 是最佳的选择。其有效性还需要进一步在真实恶意代码中进行验证。

#### 4.4.2 模拟样本的代码混淆变种检测

代码混淆是恶意代码常用的一种变种技术。代码混淆的主要手段是通过使用简短的、无意义的字母替换代码中类名、方法名和变量名。通过代码混淆，恶意代码能够轻易改变自身的代码特征，迅速产生多种变种，以躲避基于静态特征的检测技术。

Proguard 是著名的开源混淆代码工具，被集成进入 Android 的开发环境。在使用的时候只需要将工程下 project.properties 文件结尾加入“proguard.config=\${sdk.dir}/tools/proguard/proguard-android.txt:proguard-project.txt”，在 release 模式下生成项目时就启用了 ProGuard。

实验中使用了 ProGuard 工具对 4 个 Oringin 模拟样本进行了代码混淆。然后重新对其生成 ORG 图，依然使用原始的恶意代码 ORGB 图对其进行检测。匹配采用的是  $\lambda$ -VF2-Monomorphism 算法。实验结果是 4 个 ORG 全部匹配成功。

恶意代码混淆变种是 ORG 动态检测技术的重要优点之一，模拟实验验证了这一点。虽然混淆之后，代码的结构发生了变化，但是运行过程中创建的对象，以及对象之间的引用关系是不变的。ORG 检测技术正是利用了这一特征实现了恶意代码的检测。

模拟样本的实验表明，基于 ORG 的检测技术可以应对多种恶意代码的攻击和变种类型，其中以采用  $\lambda$ -VF2-Monomorphism 算法最为有效。

## 4.5 真实样本测试结果

### 4.5.1 VF2 算法在恶意代码检测中的效果

VF2 算法是精确的子图匹配算法。它要求模式图与基图的子图完全匹配。匹配精度十分高，一旦匹配上，误判的可能性会很低。但问题是，模式图与基图都会一定程度上受到噪声的影响，导致完全匹配上的可能性会很低。因此，实际应用中的可行性有待检测。本小结就是针对这个问题进行的实验。

实验中，对恶意代码样本 ADRD, Bgserv 分别使用了各类型 VF2 算法进行了测试，其中  $\lambda$  值设定为 0.8。匹配的数据结果如表 4-6 所示。匹配的对比如图 4-8 所示。

表 4-6 真实样本检测结果

算法名称	ADRD	Bgserv
VF2 Subgraph Isomorphism	1/22	1/16
VF2 Monomorphism	2/22	2/16
$\lambda$ -VF2 Subgraph Isomorphism	12/22	9/16
$\lambda$ -VF2-Monomorphism	16/22	11/16

测试结果数据如表 4-6。从表中可以看出，VF2 Subgraph Isomorphism 和 VF2 Monomorphism 匹配成功的比例很小，主要原因分析如下。

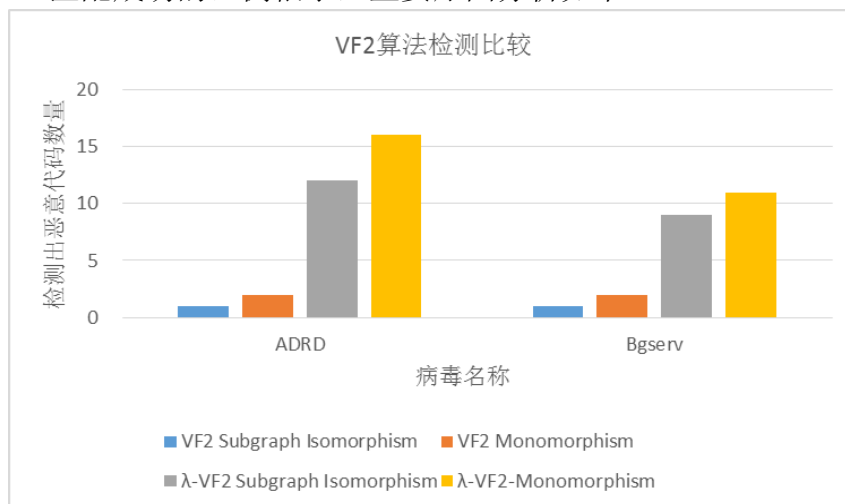


图 4-8 模拟样本检测对比图

(1) 由于每一类恶意代码内的样本不是完全相同，导致作为恶意代码的特征的 ORGB 图提取的不够典型。

(2) 在应用程序运行的过程中，恶意代码会随之运行并产生动作，在这一过

程中，恶意代码会动态的创建和销毁类，导致应用程序的 ORG 图中，恶意代码相关部分可能会不够完全，导致匹配失效。

(3) 这两算法都是求精确匹配，而以上两个原因导致 ORGB 无法与 ORG 精确匹配。

$\lambda$ -VF2 Subgraph Isomorphism 和  $\lambda$ -VF2-Monomorphism 两个算法的匹配率相对来说较高。这表明匹配精确度的降低可以消除一定噪音的影响，提高匹配准确率。

#### 4.5.2 $\lambda$ -VF2 算法在不同精确度下效果对比测试

$\lambda$ -VF2-Monomorphism 算法在真实恶意代码测试中较为实用，其中  $\lambda$  的选取对匹配结果会产生较大的影响。如果  $\lambda$  的值减小，在其值趋近于 0 的过程中，匹配的精度会越来越低，误判的可能性就会逐渐增加，也就是可能会把本不是恶意代码的应用程序误判为恶意代码；如果  $\lambda$  的值减小，在其值趋近于 1 的过程中，匹配的精度会越来越低，漏判的可能性就会逐渐增加，也就是会把恶意代码程序判断为正常应用程序。本小结的实验就是来研究  $\lambda$  的不同取值对结果的影响。

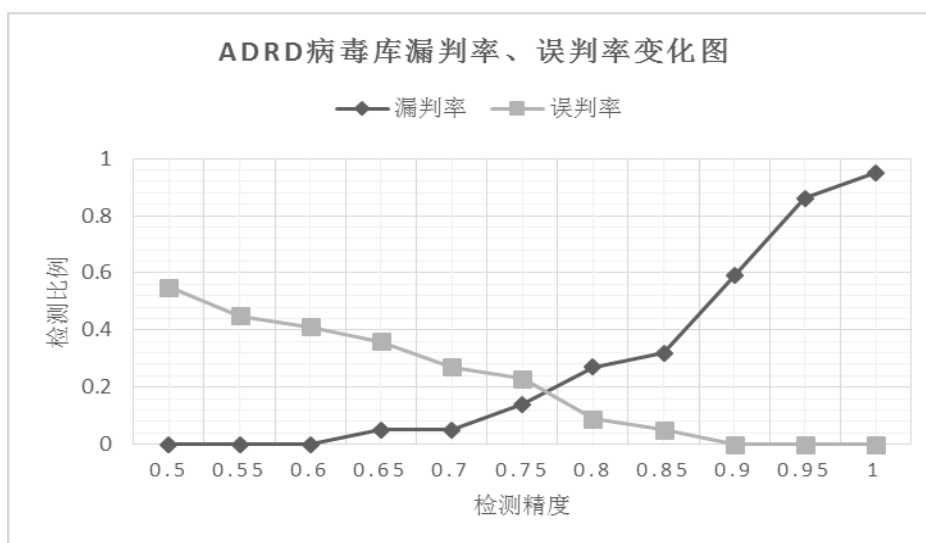
为了能够看出  $\lambda$  值降低后的误判率，实验中又加入了一组正常应用程序，不含有恶意代码。在测试的时候分组进行，每组包含一个恶意代码类库和一个正常的应用程序库，两个库的样本数量相同。 $\lambda$  从 0.5 开始，步进为 0.05。这样随着  $\lambda$  值的不断变化，可以从恶意代码库的判断结果得到漏判率，从正常应用程序库的结果中得到误判率。

第一组：ADRD 恶意代码库。实验结果数据如下所示。

表 4-7 随着  $\lambda$  变化 ADRD 病毒库测试结果

$\lambda$ 值	0.5	0.55	0.6	0.65	0.7	0.75	0.8	0.85	0.9	0.95	1
总数	22	22	22	22	22	22	22	22	22	22	22
ADRD	22	22	22	21	21	19	16	15	9	3	1
正常	12	10	9	8	6	5	2	1	0	0	0
漏判率	0.00	0.00	0.00	0.05	0.05	0.14	0.27	0.32	0.59	0.86	0.95
误判率	0.55	0.45	0.41	0.36	0.27	0.23	0.09	0.05	0.00	0.00	0.00

从表中可以看出当  $\lambda$  取值 0.9 时，漏判率已经高达 0.5，已经无法满足实际需求了。而当  $\lambda$  取到 0.75 时误报率高达 0.23，也不符合实际需求了。因此  $\lambda$  在 0.75 到 0.85 之间较为合适。图 4-9 可以直观看到漏判率和误判率的变化。

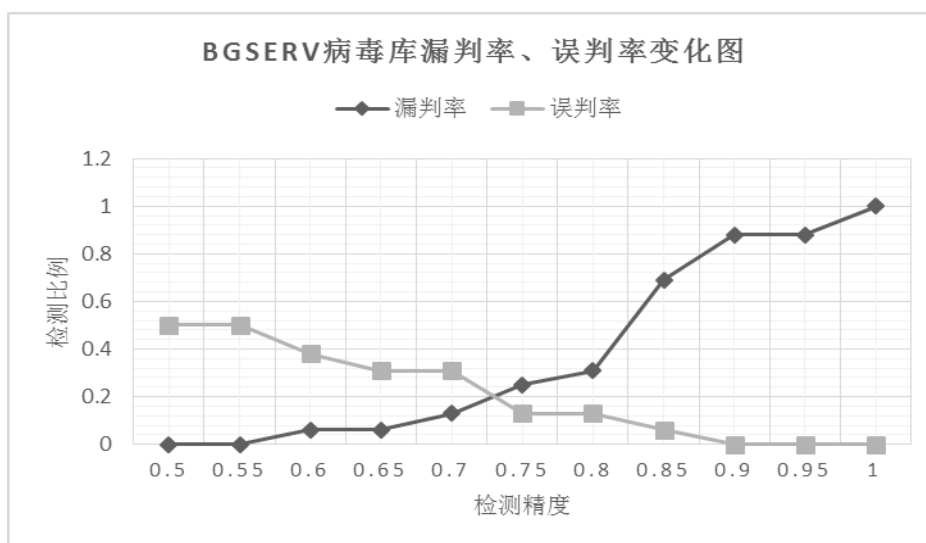

 图 4-9 随着  $\lambda$  变化 ADRD 病毒库测试结果

第二组：Bgserve 恶意代码库。实验结果数据如下所示。

 表 4-8 随着  $\lambda$  变化 Bgserve 病毒库测试结果

$\lambda$ 值	0.5	0.55	0.6	0.65	0.7	0.75	0.8	0.85	0.9	0.95	1
总数	16	16	16	16	16	16	16	16	16	16	16
ADRD	16	16	15	15	14	12	11	5	2	2	0
正常	8	8	6	5	5	2	2	1	0	0	0
漏判率	0.00	0.00	0.06	0.06	0.13	0.25	0.31	0.69	0.88	0.88	1.00
误判率	0.50	0.50	0.38	0.31	0.31	0.13	0.13	0.06	0.00	0.00	0.00

从表中可以看出当  $\lambda$  取值 0.85 时，漏判率已经高达 0.69，已经无法满足实际需求了。而当  $\lambda$  取到 0.7 时误报率高达 0.31，也不符合实际需求了。因此  $\lambda$  在 0.7 到 0.8 之间较为合适。图 4-10 可以直观看到漏判率和误判率的变化。


 图 4-10 随着  $\lambda$  变化 Bgserve 病毒库测试结果

通过以上两组实验可以看出,随着  $\lambda$  的增加,恶意代码的漏判率在不断提高,而误判率在不断降低。这两个参数的值是此消彼长的关系,具体应用中可根据测试要求选取  $\lambda$  的取值,以保证误判率和漏判率能同时满足要求。综合上面试验情况来看, $\lambda$  在 0.85 左右得到的结果较好。

## 4.6 本章小结

本章介绍了 Android 端 MalwareDetection 数据抽取的流程,并说明了主要模块:前台界面、活动进程查询器、Shell 命令执行器、Convertor、AHAT 的功能。然后设计并实现了服务器端对对象引用关系图的建立和匹配。最后设计并进行了实验,验证在模拟样本中 VF2 算法和  $\lambda$ -VF2 算法的表现,并在真实恶意代码库上测试了不同  $\lambda$  值对误判率和漏判率的影响。

## 结 论

本文主要研究了以对象引用关系图为基础的软件动态胎记,并在 Android 系统上以恶意代码入侵检测为背景,验证了 ORG 作为软件的动态特征在软件识别上的有效性。

建立 ORG 图首先要解决的问题是应用程序对象引用关系的获取,而内存数据的提取和分析是本文在研究过程中的一个难点。本文通过自己开发的工具 MalwareDetection 实现了在 Android 系统下,对应用程序的原始内存数据的导出、分析并最终获得对象引用关系数据的功能。

ORG 图的匹配算法是本文研究的重点。本文分析了多种图同构匹配的算法,比较了各类算法优缺点。ORG 图的特点是节点规模不大,边较为稀疏。根据 ORG 图这一特点本文选取了 VF2 算法作为匹配的基础算法。为了能够在实际测试中,有效进行同构匹配,本文修改了 VF2 算法,加入了控制匹配精度的参数  $\lambda$ 。

模拟样本的测试实验表明基于 ORG 的检测技术可以有效的应对各种恶意代码变种的检测,尤其是对混淆代码类型的变种有着明显优势。

实验中,分别测试了 VF2 算法和  $\lambda$ -VF2 算法。由于噪音的干扰, VF2 算法匹配的效果较差,恶意代码的检测率较低。而  $\lambda$ -VF2 算法通过设定精确度  $\lambda$  的值,能够有效检测出恶意代码。

本文在实验中测试了不同的  $\lambda$  值对匹配结果的影响,给出了在  $\lambda$  值的变化过程中,误判率和漏判率的变化,分析了实际应用中  $\lambda$  的合理取值。

本文只是在 ORG 图的动态软件识别技术上进行了一个初步的研究,还有很多相关工作要做,如 ORG 图建立的优化,匹配算法性能的改进等等。

## 参考文献

- [1] 尤肖虎, 曹淑敏, 李建东. 第三代移动通信系统发展现状与展望[J]. 电子学报, 2013, 27(11): 3-8.
- [2] Idika N, Mathur A P. A survey of malware detection techniques[J]. Purdue University, 2007: 48.
- [3] Wang X, Jhi Y C, Zhu S, et al. Behavior based software theft detection[C]//Proceedings of the 16th ACM conference on Computer and communications security. ACM, 2009: 280-290.
- [4] Collberg C, Thomborson C. Software watermarking: Models and dynamic embeddings[C]//Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1999: 311-324.
- [5] Collberg C, Carter E, Debray S, et al. Dynamic path-based software watermarking[J]. ACM SIGPLAN Notices, 2004, 39(6): 107-118.
- [6] Collberg C, Thomborson C, Low D. A taxonomy of obfuscating transformations[R]. Department of Computer Science, The University of Auckland, New Zealand, 1997.
- [7] Myles G, Collberg C. Detecting software theft via whole program path birthmarks[M]//Information security. Springer Berlin Heidelberg, 2004: 404-415.
- [8] H. Tamada, K. Okamoto, M. N. A. M., and ichi Matsumoto, K. Detecting the theft of programs using birthmarks.//Tech. rep., Graduate School of Information Science, Nara Institute of Science and Technology, 2003.
- [9] Collberg C, Thomborson C. Software watermarking: Models and dynamic embeddings[C]//Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1999: 311-324.
- [10] Tamada H, Okamoto K, Nakamura M, et al. Dynamic software birthmarks to detect the theft of windows applications[C]//International Symposium on Future Software Technology. 2004, 2004..
- [11] Payet É, Spoto F. Static analysis of Android programs[J]. Information and Software Technology, 2012, 54(11): 1192-1201.
- [12] Kui Luo. Using static analysis on Android applications to identify private information leaks:[Master Dissertation]. Kansas: Kansas State University, 2011,

1-2.

- [13]Dienst S, Berger T. Static Analysis of App Dependencies in Android Bytecode[J].
- [14]Shabtai A, Fledel Y, Elovici Y. Automated static code analysis for classifying android applications using machine learning[C]//Computational Intelligence and Security (CIS), 2010 International Conference on. IEEE, 2010: 329-333.
- [15]Schmidt A D, Bye R, Schmidt H G, et al. Static analysis of executables for collaborative malware detection on android[C]//Communications, 2009. ICC'09. IEEE International Conference on. IEEE, 2009: 1-5.
- [16]童振飞, 杨庚. Android 平台恶意软件的静态行为检测[J]. 江苏通信, 2011, 5(1): 35-39.
- [17]Batyuk L, Herpich M, Camtepe S A, et al. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications[C]//Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on. IEEE, 2011: 66-72.
- [18]Di Cerbo F, Girardello A, Michahelles F, et al. Detection of malicious applications on android os[M]//Computational Forensics. Springer Berlin Heidelberg, 2011: 138-149.
- [19]Isohara T, Takemori K, Kubota A. Kernel-based Behavior Analysis for Android Malware Detection[C]//Computational Intelligence and Security (CIS), 2011 Seventh International Conference on. IEEE, 2011: 1011-1015.
- [20]Schmidt A D, Bye R, Schmidt H G, et al. Monitoring android for collaborative anomaly detection: A first architectural draft[J]. Technische Universität Berlin-DAI-Labor, Tech. Rep. TUB-DAI, 2008, 8: 08-02.
- [21]Burguera I, Zurutuza U, Nadjm-Tehrani S. Crowdroid: behavior-based malware detection system for android[C]//Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices. ACM, 2011: 15-26.
- [22]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(1): 161-190.
- [23]Yan Q, Li Y, Li T, et al. Insights into malware detection and prevention on mobile phones[M]//Security Technology. Springer Berlin Heidelberg, 2009: 242-249.
- [24]Blasing T, Batyuk L, Schmidt A D, et al. An android application sandbox system for



- suspicious software detection[C]//Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on. IEEE, 2010: 55-62.
- [25]Wang X, Jhi Y C, Zhu S, et al. Behavior based software theft detection[C]//Proceedings of the 16th ACM conference on Computer and communications security. ACM, 2009: 280-290.
- [26]Tamada H, Okamoto K, Nakamura M, et al. Design and evaluation of dynamic software birthmarks based on api calls[J]. Info. Science Technical Report NAIST-IS-TR2007011, ISSN, 2007: 0919-9527.
- [27]Schuler D, Dallmeier V, Lindig C. A dynamic birthmark for java[C]//Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. ACM, 2007: 274-283.
- [28]Sosonkin M, Naumovich G, Memon N. Obfuscation of design intent in object-oriented applications[C]//Proceedings of the 3rd ACM workshop on Digital rights management. ACM, 2003: 142-153.
- [29]Wang X, Jhi Y C, Zhu S, et al. Behavior based software theft detection[C]//Proceedings of the 16th ACM conference on Computer and communications security. ACM, 2009: 280-290.
- [30]辛天卿. 基于序列匹配的代码克隆分析系统设计与实现 [D][D]. , 2008.
- [31]Conte D, Foggia P, Sansone C, et al. Thirty years of graph matching in pattern recognition[J]. International journal of pattern recognition and artificial intelligence, 2004, 18(03): 265-298.
- [32]Google Inc. Android Developer Documents android-app [EB/OL]. [2011-12-30].<http://developer.android.com/reference/android/app/package-summary.html>.
- [33]Google Inc. Android Developer Documents android-content [EB/OL]. [2011-12-30].<http://developer.android.com/reference/android/content/package-summary.html>.
- [34]Idika N, Mathur A P. A survey of malware detection techniques[J]. Purdue University, 2007: 48.
- [35]2011 年中国大陆地区手机安全报告 .[EB/OL] <http://www.netqin.com/upload/File/baogao/20120112.pdf>.
- [36]Chan P P F, Hui L C K, Yiu S M. Dynamic software birthmark for java based on

- heap memory analysis[C]//Communications and Multimedia Security. Springer Berlin Heidelberg, 2011: 94-107.
- [37]M.R. Garey and D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman and Co., 1979.
- [38]Messmer B T, Bunke H. A decision tree approach to graph and Subgraph Isomorphism detection[J]. Pattern recognition, 1999, 32(12): 1979-1998.
- [39]M.R. Garey and D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman and Co., 1979.
- [40]Fortin S. The graph Isomorphism problem[R]. Technical Report 96-20, University of Alberta, Edmonton, Alberta, Canada, 1996.
- [41]Ullmann J R. An algorithm for Subgraph Isomorphism[J]. Journal of the ACM (JACM), 1976, 23(1): 31-42.
- [42]Ghahraman D E, Wong A K C, Au T. Graph optimal monomorphism algorithms[J]. Systems, Man and Cybernetics, IEEE Transactions on, 1980, 10(4): 181-188.
- [43]Cordella L P, Foggia P, Sansone C, et al. Fast graph matching for detecting CAD image components[C]//Pattern Recognition, 2000. Proceedings. 15th International Conference on. IEEE, 2000, 2: 1034-1037.
- [44]Cordella L P, Foggia P, Sansone C, et al. An improved algorithm for matching large graphs[C]//3rd IAPR-TC15 workshop on graph-based representations in pattern recognition. 2001: 149-159.
- [45]McKay B D. Practical graph Isomorphism[M]. Department of Computer Science, Vanderbilt University, 1981.
- [46]Foggia P, Sansone C, Vento M. A performance comparison of five algorithms for graph Isomorphism[C]//Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition. 2001: 188-199.

## 攻读学位期间发表的学术论文

陆亮, 李东, 王科怀, 支持 Ajax 的网络爬虫研究与实现 智能计算机与应用 录用待发表。

## 哈尔滨工业大学学位论文原创性声明和使用权限

### 学位论文原创性声明

本人郑重声明：此处所提交的学位论文《基于对象引用关系图的 Android 恶意代码检测的研究》，是本人在导师指导下，在哈尔滨工业大学攻读学位期间独立进行研究工作所取得的成果，且学位论文中除已标注引用文献的部分外不包含他人完成或已发表的研究成果。对本学位论文的研究工作做出重要贡献的个人和集体，均已在文中以明确方式注明。

作者签名：  日期：2013 年 7 月 1 日

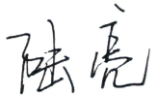
### 学位论文使用权限


学位论文是研究生在哈尔滨工业大学攻读学位期间完成的成果，知识产权归属哈尔滨工业大学。学位论文的使用权限如下：

(1) 学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文，并向国家图书馆报送学位论文；(2) 学校可以将学位论文部分或全部内容编入有关数据库进行检索和提供相应阅览服务；(3) 研究生毕业后发表与此学位论文研究成果相关的学术论文和其他成果时，应征得导师同意，且第一署名单位为哈尔滨工业大学。

保密论文在保密期内遵守有关保密规定，解密后适用于此使用权限规定。

本人知悉学位论文的使用权限，并将遵守有关规定。

作者签名：  日期：2013 年 7 月 1 日

导师签名：  日期：2013 年 7 月 1 日

## 致 谢

衷心感谢研究生学习生活中所有给予我帮助的老师 and 同学！

感谢我的导师李东教授。从入学以来，李老师在学习和生活中一直给予了我无微不至的关怀。平时，李老师在百忙之中经常主动联系我，对我谆谆教诲，让我受益匪浅。导师在生活中也给了我很多的帮助，最令我印象深刻的是，在我崴脚期间，李老师两次亲自来到宿舍为我送饭，关心我的恢复情况，令我倍受感动。导师勤勉努力、乐观豁达的性格更让我学习到一个人对待生活一定要有一个积极向上的态度。

感谢张宏莉教授，张老师在生活和学习上给了我很多的关怀和帮助，她给了我许多宝贵的意见，让我深受启迪，获益匪浅。

感谢张伟哲教授，张老师在课题研究上给予了我很多指导和帮助，拓宽了我的思路，让我在课题科研方面有了很大进步。他严谨的科研态度，热情的探索精神，敏锐的思维一直是我学习的榜样，督促着我不断努力。

感谢余翔湛副教授、何慧副教授、宋颖慧副教授，刘家辉副教授、张宇老师、叶琳老师，他们在我做论文期间给我提出了宝贵的意见，感谢所有实验室的老师对我的教诲和帮助。

感谢魏欣、欧阳显雅、吴世山、王东、王科怀、薛建、苏银杰等各位同学在学业和课余生活给予的热情鼓励和真诚帮助。

感谢我的父母，在我读研期间，他们的关心一直伴随着我的成长，他们的理解和支持是我不断进步的信心和动力。

# 基于对象引用关系图的Android恶意代码检测的研究

作者: [陆亮](#)

学位授予单位: [哈尔滨工业大学](#)

## 参考文献(2条)

1. [尤肖虎, 曹淑敏, 李建东](#) 第三代移动通信系统发展现状与展望[期刊论文]-[电子学报](#) 1999 (Z1)
2. [辛天卿](#) [基于序列匹配的代码克隆分析系统设计与实现](#)[学位论文]硕士 2008

引用本文格式: [陆亮](#) [基于对象引用关系图的Android恶意代码检测的研究](#)[学位论文]硕士 2013