基于Android平台的SO文件壳检测与脱壳技术研究

专业

研究生 指导老师

随着Android操作系统的不断发展，Android安全加壳技术也逐步普及，越来越多的Android应用程序开始使用加壳保护自己。加壳服务能够有效保护程序的真实逻辑和完整性。但是这些特点同时很容易被恶意程序利用，有数据表明随着加壳技术的流行，加壳恶意程序的数量也越来越多。传统的恶意代码检测在加壳应用的逆向分析环节比较薄弱，难以对加壳应用进行检测，这给Android的安全发展带来了严峻的挑战。

本文的研究对象是Android应用程序中的常见可执行文件——SO（动态链接库）文件，研究目的是寻找一种Android平台下的SO文件壳检测与脱壳方法。本文的主要工作可以归纳为：

首先，本文系统地研究了Android JNI机制，对SO文件格式进行了深入研究，研究内容包括ELF头、节相关内容、段相关内容。为后续壳检测和脱壳方法的研究做准备。

其次，本文通过对现有各种流行的加壳技术进行研究，发现未加壳文件与加壳文件在SO文件格式上存在较大的差异。在以上研究基础上，采用了一种基于SO文件结构的静态特征抽取方法。该方法通过对SO文件结构进行解析，初步抽取出SO文件的特征。

接着，本文针对初选特征中包含了大量与加壳SO文件无关的属性的问题，采用了一种基于线性判别分析（LDA）的特征变换方法，该方法通过使用LDA算法，剔除了初选特征中的无用和冗余特征，达到降维的目的，以提高分类器的检测效果。

然后，本文借鉴恶意代码检测的相关方法，将机器学习的相关理论应用到壳检测领域中，采用了一种基于朴素贝叶斯分类器的壳检测方法。该方法首先提取训练样本的特征用于训练分类器，最后利用训练的分类器对测试样本进行壳检测。

最后，本文借鉴Window平台加壳PE文件的脱壳方法，并结合对Android平台SO文件加载机制的研究，采用了一种基于soinfo内存转存的脱壳方法。该方法分为内存dump和文件修复两个阶段。首先通过研究SO文件的加载机制确定OEP地址，从而实现自动化内存dump，完成初步脱壳。最后通过解析soinfo结构体信息，对初步脱壳后的SO文件的头部、段地址、重定位表以及节头进行修复，实现完全脱壳。完全脱壳后的SO文件可以同时被静态分析和动态调试。

实验表明，本论文提出并实现的基于Android平台的SO文件壳检测与脱壳技术有着良好的实际效果，具有一定的理论参考价值和实际应用价值。

关键词：Android，SO文件，壳检测，脱壳

**Research on Detection and Unpacking Technology of Packed SO File Based on Android Platform**

**Presented for MSc Degree**

**Subject: Security of Information System**

**Postgraduate: Supervisor:**

**Abstract:** With the development of the Android operating system. Android security packing technology has been gradually popularized, more and more Abdroid applications begin to use packing to protect themselves. Packing services can effectively protect the real logic and the integrity of the program. However, these features could easily be used by malicious programs, data show that with the popularity of the packing technology, the numbers of packed malicious programs are also increasing. In the reverse analysis of packing application, traditional malicious detection is relatively weak, it is difficult to detect the packing application, which has brought serious challenges to the development of Android security.

The objectives of this thesis is SO(shared object) file which is the main executable file format of Android application. The aim is to find a method for SO file detection and unpacking based on Android platform. The main work and contributions of this thesis are summarized as follows:

Firstly, the Android JNI mechanism is systematically studied in this thesis, ELF header is described in detail, sections and segments of SO file are thoroughly analyzed. For further propose of packed SO file detection and unpacking.

Secondly, through the study of various popular packing technology, this thesis find that there is a difference between the packed file and the unpacked file in the SO file format. A static feature extraction method based on SO file structure is given by this thesis based on the above research. This method helps to get primary features by parsing SO file structure.

Subsequently, according to the problems of the useless and redundant features in the primal features, a feature transformation method based on linear discriminant analysis is adopted in this thesis. This method eliminates the useless and redundant features in the primary features by using LDA-based algorithm. The aim of this method is to reduce the dimension and improve the detection effect of the classifier.

Then, by using related methods of malicious code detection, the machine learning theory is applied to the field of detection of packed file, a packed SO file detection method base on Native Bayes classifier is proposed in this thesis. The detailed process is first extract key features of sample to train the classifier, and then classify test samples with the classifier.

Finally, referring to the upacking method of PE file based on Windows platform and the study on the loading mechanism of SO file on Android platform, a packed SO file unpacking method based on soinfo memory dump is proposed in this thesis. This method is divided into two stages, namely memory dump stage and SO file restoration stage. Firstly, through the study of the loading mechanism of SO file to determine the OEP address, so as to realize the automation of memory dump, then preliminary unpacking is completed. Finally, through the analysis of the soinfo structure information, ELF header, segment address, relocation table and section header of the SO file which is preliminary unpacking are be repaired, then fully unpacking is completed. The SO file which is fully unpacking could be simultaneously static analysis and dynamic debugging.

Experiments show that packed SO file detection and unpacking method based on the Android platform in this thesis is able to a good practical effect. This thesis has a certain theoretical reference value and practical value to further research.

**Keywords** Android, SO file, detection, unpacking

**目 录**

第1章 绪论 1

1.1 研究背景 1

1.2 国内外研究现状 3

1.3 研究内容 4

1.4 论文主要工作及组织结构 5

第2章 Android平台与SO文件格式 6

2.1 Android系统概要 6

2.1.1 Android系统简介 6

2.1.2 Android系统架构 8

2.2 Android应用的组成结构 9

2.3 JNI机制与NDK 9

2.4 ELF文件格式概述 10

2.4.1 ELF文件类型 10

2.4.2 文件格式 11

2.4.3 数据表示形式 12

2.5 ELF头 12

2.6 节 14

2.6.1 节头 14

2.6.2 特殊节 17

2.6.3 字符串表节 18

2.6.4 符号表节 19

2.6.5 重定位节 21

2.6.6 动态节 23

2.6.7 初始化节 24

2.7 段 24

2.7.1 程序头 25

2.7.2 可加载段 27

2.7.3 动态段 27

2.8 本章小结 27

第3章 基于朴素贝叶斯分类器的壳检测技术 28

3.1 加壳文件 28

3.1.1 壳的概念 28

3.1.2 流行的加壳技术 28

3.1.3 未加壳文件与加壳文件的差异 29

3.2 壳检测技术的框架设计 31

3.2.1 设计思想 31

3.2.2 总体框架 33

3.3 SO文件格式检测 34

3.4 特征集 35

3.4.1 特征抽取 35

3.4.2 特征变换 36

3.5 朴素贝叶斯分类方法 38

3.5.1 贝叶斯定理 38

3.5.2 朴素贝叶斯分类 38

3.5.3 训练与分类的过程 39

3.6 本章小结 40

第4章 基于soinfo内存转存的脱壳技术 41

4.1 加载机制 41

4.1.1 SO文件的加载机制 41

4.1.2 壳的加载机制 43

4.2 脱壳技术的框架设计 43

4.2.1 设计思想 43

4.2.2 总体框架 45

4.3 寻找OEP 45

4.4 soinfo结构体 46

4.5 内存dump 47

4.6 重构SO文件 48

4.6.1 头部修复 48

4.6.2 段地址修复 49

4.6.3 重定位表修复 50

4.6.4 重建节头 51

4.6.5 清除壳入口 56

4.7 本章小结 57

第5章 实验结果与分析 58

5.1 壳检测实验 58

5.1.1 WEKA平台概述 58

5.1.2 实验环境与数据 58

5.1.3 评价指标 58

5.1.4 实验结果与分析 59

5.2 脱壳实验 60

5.2.1 实验环境与数据 60

5.2.2 实验结果与分析 60

5.3 本章小结 63

第6章 总结与展望 64

6.1 研究工作总结 64

6.2 研究展望 65

参考文献 67

作者在读期间科研成果简介 69

致 谢 70

声 明 71

学位论文版权使用授权书 72

# 第1章 绪论

## 1.1 研究背景

由于人们对于便捷性和即时性的需求，传统互联网已经无法满足人们的要求。移动互联网应运而生并迅猛发展，根据中国互联网络信息中心公布的第 34 次《中国互联网络发展状况统计报告》显示，在2014年6月，中国网民上网设备中，手机使用率首次超越传统PC使用率[[[1]](#endnote-1)]，这标志着中国正式跨入了移动互联网时代。

伴随着移动互联网的发展，Android操作系统也发展迅猛，Android凭借其强大的开放性和丰富的硬件支持，一经推出就得到了广大用户的喜爱，自2008年发行第一版Android操作系统以来，采用Android作为操作系统的智能机的市场占有率不断攀升。图1.1是IDC发布的2014年智能手机出货量数据，单位为百万部[[[2]](#endnote-2)]。

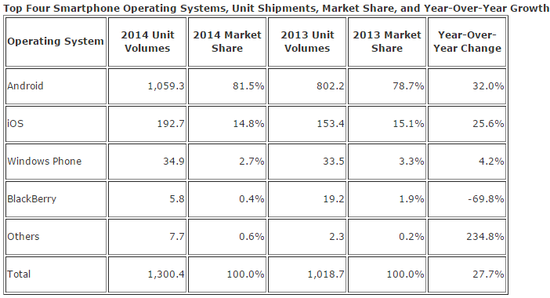


图1.1 2014年智能手机出货量

从图1.1可知，采用Android操作系统的智能手机出货量明显领先，总出货量超过十亿大关，甚至超过了2013年全部智能手机的总出货量，其出货量占比达到了81.5%。

于此同时，随着Android的广泛应用，其安全问题也逐渐暴露，且已经严重影响了Android的进一步发展[[[3]](#endnote-3)]。从2010年8月发现的首个Android木马程序“Trojan-SMS.Android – OS.FakePlayer”以来，Android下的恶意软件的数量急速扩张，严重威胁了用户的安全。图1.2是近几年Android恶意代码数量变化情况[[[4]](#endnote-4)]。

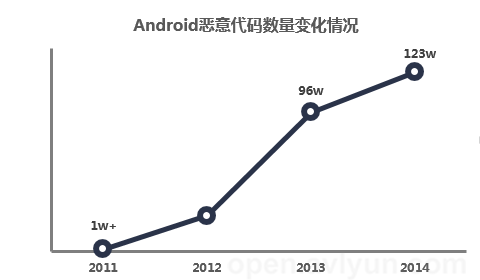


图1.2 Android恶意代码数量变化情况

从图1.2可知，从2011年到2014年，Android恶意代码数量每一年都增长迅速，其中2014年Android恶意代码总量已增至123万。更加令人担忧的是，更多的恶意软件采用加壳技术躲避安全软件的查杀，相比2013年，2014年加壳恶意软件的数量增长了约18倍，如图1.3所示[4]。

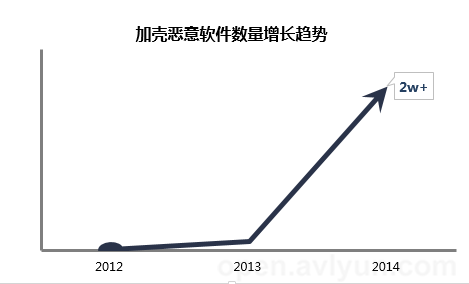


图1.3 加壳恶意软件数量增长趋势

经过加壳后的恶意软件更加难以识别与分析，给杀毒软件的杀毒工作造成了巨大的困难。

Android包含两种可执行文件，分别是以Java语言编写的可执行文件Dex（Dalvik VM executes）[[[5]](#endnote-5)]和以C/C++语言编写的共享目标文件SO（Shared Object）[[[6]](#endnote-6)]。因此，对Android的脱壳技术研究自然也分为Dex脱壳技术和SO脱壳技术。

目前国内外在Dex脱壳方面的研究已经取得了比较明显的成果，出现了数种Dex文件脱壳工具，其中较为常用的有ZjDroid[[[7]](#endnote-7)]和DexHunter[[[8]](#endnote-8)]，两者均具有较高的脱壳效率。然而另一方面，在SO脱壳方面的研究才刚刚起步，大多数安全从业者仍采取手动的手段通过静态分析与动态调试进行脱壳，其效率不高，脱壳的成功率也取决于从业者的技术水平。因此迫切需要对SO壳检测与脱壳技术进行研究。

## 1.2 国内外研究现状

目前，在Window平台上的壳检测与脱壳技术已经相当成熟，在壳检测技术方面主要有三种，分别为基于签名的壳检测技术，基于熵的壳检测技术和基于PE头部属性的壳检测技术[[[9]](#endnote-9)]。

基于签名的壳检测技术，主要用于检测已知壳。其基本思想是为每种已知壳产生一个专属的签名特征，并存入特征数据库。检测的时候，将待测文件的签名在特征数据库中进行搜索，若匹配成功，则判定为加壳文件，否则为未加壳文件。典型的工具有PEID与FileInfo[[[10]](#endnote-10)]。

基于熵的壳检测技术，壳程序通常都会对原文件的内容进行压缩或加密，经过压缩或加密的加壳文件的内容会更加丰富，包含的信息的无序化程度更高，也就是相比原文件，加壳文件的熵值更高。典型的工具为Bintropy[[[11]](#endnote-11)]。

基于PE头部属性的壳检测技术，其原理是由于加壳文件和未加壳文件在PE头部上具有较大的差异性，检测时通过分析PE头信息，从而判断是否加壳。典型的工具为PHAD[[[12]](#endnote-12)]。

基于签名的壳检测技术具有实现简单，对已知壳检测率高的优点，然而其检测率的高低依赖于特征数据库的完善程度，无法检测变种壳和未知壳，具有一定的滞后性。与基于签名的壳检测技术不同，基于熵的壳检测技术对已知壳、变种壳和未知壳都有着相同的检测效果，但是由于熵值的阈值难以确定，造成其有较高的误报率。Choi在2008年提出的基于PE头部属性的壳检测技术的查壳软件PHAD，取得了比上述两种技术更好的检测效果。

壳检测技术的发展也刺激了脱壳技术的研究，常见的脱壳技术主要有三种，基于工具的脱壳技术，基于静态分析的脱壳技术，基于动态分析的脱壳技术。

基于工具的脱壳技术，即使用已有的脱壳工具对加壳文件进行自动脱壳。如UPX：UPX是一种支持多平台可执行文件加壳的压缩壳工具，支持对Android平台下的SO文件进行加壳，同时其本身也提供脱壳功能，使用upx –d命令即可对UPX加壳文件自动脱壳。

基于静态分析的脱壳技术，主要是利用IDA、readelf、objdump等工具静态分析加壳文件的程序流程，并编写相应的脱壳算法对加壳文件完成静态解密，从而脱壳。

基于动态分析的脱壳技术，是使用IDA等工具附加运行中的加壳程序，对运行中的加壳程序进行跟踪与分析，并记录相关寄存器的值，最终找到原程序的入口（OEP），从而完成脱壳。

基于工具的脱壳技术只能针对性的脱壳，无法处理未知壳和变种壳。而基于静态分析的脱壳技术和基于动态分析的脱壳技术对脱壳者有很高的水平要求，涉及到计算机软件的各个方面的知识[[[13]](#endnote-13)]，其脱壳的成功率很大程度上依赖于脱壳者的能力，并且其具有成本高、效率低的缺点。

虽然壳检测技术与脱壳技术在Window平台已经取得了较好的研究成果，但是在Android平台下，对于SO文件的壳检测技术与脱壳技术研究的进展还十分缓慢。到目前为之，无论是在网上还是在各类期刊杂志都找不到一套完整的技术理论。因此，目前迫切需要寻找一种在Android平台下切实可行的SO文件壳检测与脱壳方法。

## 1.3 研究内容

本文致力于研究基于Android平台的SO壳检测与脱壳技术中需要解决的技术问题，核心研究内容包括以下5个方面：

1）系统地研究了Android平台与JNI机制，详细分析了Android平台上的SO文件格式。

2）探讨了流行的SO文件加壳技术，并根据加壳文件与未加壳文件在SO文件格式上的不同这一特点，采用了一种基于SO文件格式的静态特征提取方法。

3）通过将机器学习相关理论应用到壳检测领域，采用了一种基于朴素贝叶斯分类器的壳检测方法。

4）探讨了Android平台上的SO文件的加载机制和壳的加载机制，基于上述研究，采用了一种基于soinfo内存转存的脱壳方法。

5）对本文实现的壳检测与脱壳系统进行实验，并对结果进行分析。

## 1.4 论文主要工作及组织结构

本文全篇分为六个章节：

第1章 绪论。主要介绍了本文的研究背景，壳检测技术与脱壳技术在国内外的研究现状，以及本文的主要工作和组织结构。

第2章 Android平台与ELF文件格式。首先介绍了Android的系统架构和JNI机制；最后分析了本文的研究对象SO文件的文件格式，即ELF文件格式。

第3章 基于朴素贝叶斯分类器的壳检测技术。首先介绍了壳的概念和现有的加壳技术，深入分析了加壳文件与未加壳文件在SO文件格式上的差异，从而得到了初选特征；接着利用线性变换对初选特征进行降维得到最终的特征集，最后在以上的研究基础上，利用朴素贝叶斯分类方法实现壳检测。

第4章 基于soinfo内存转存的脱壳技术。首先介绍了SO文件和壳的加载机制；然后在前文的研究基础上找到了一个脱壳的时机，并利用自脱壳技术完成初步脱壳；最后通过头部修复、段地址修复、重定位表修复、重建节头和清除壳入口等一系列手段重构SO文件，最终实现完全脱壳。

第5章 实验结果与分析。通过实验样本对本文提出的基于Android平台的SO文件壳检测与脱壳系统进行实验，并对实验结果进行相应的分析。

第6章 总结与展望。首先对全文进行系统性的总结，并指出本文的主要贡献。最后提出进一步工作的方向。

# 第2章 Android平台与SO文件格式

Android平台与传统PC平台在很多方面都有着巨大的差异，比如系统架构、主流可执行文件等，因此研究Android平台的相关知识，将有助于研究Android平台上的SO文件壳检测与脱壳技术。本章首先对Android平台相关知识进行了系统的介绍，最后深入分析了SO文件格式。

## 2.1 Android系统概要

### 2.1.1 Android系统简介

Android系统是一种应用于智能移动设备（例如：智能手机、平板电脑等）的移动操作系统，最早由Google公司独立开发和管理，后由Google公司组建的开放手机联盟（Open Handset Alliance，简称OHA）共同开发管理[[[14]](#endnote-14)]。

自从2008年9月，Google公司正式发布第一代Android操作系统，即Android1.0系统以来，Android系统以惊人的速度不断更新换代，几乎每一年都会发布数个更新版本，表2-1展示了正式发布的系统版本号以及对应的发布时间。

表2-1 Android历史版本号与发布

|  |  |  |  |
| --- | --- | --- | --- |
| **版本号** | **发布时间** | **版本号** | **发布时间** |
| 1.0 | 2008年9月 | 4.0-4.0.4 | 2011年10月 |
| 1.1 | 2009年2月 | 4.1-4.1.2 | 2012年6月 |
| 1.5 | 2009年4月 | 4.2-4.2.2 | 2012年10月 |
| 1.6 | 2009年9月 | 4.3-4.3.1 | 2013年7月 |
| 2.0-2.1 | 2010年1月 | 4.4-4.4.4 | 2013年9月 |
| 2.2 | 2010年5月 | 5.0-5.0.2 | 2014年6月 |
| 2.3-2.3.7 | 2010年12月 | 5.1-5.1.1 | 2015年3月 |
| 3.0-3.2.2 | 2011年2月 | 6.0-6.0.1 | 2015年5月 |

从表2-1中可以看到，截至2015年5月，Android系统的最新版本号为6.0，代号为Marshmallow，中文译为“棉花糖”，随着不断更新的Android系统版本，Android系统的市场占有率也一路高歌猛进。据Strategy Analytics于2014年公布的统计报告显示，Android设备的出货量已经超过10亿部，远远超过其最大竞争对手苹果设备1.93亿部的出货量[[[15]](#endnote-15)]。同时据Gartner发布的预测，在2015年采用Android系统的设备占有率将达到60%左右[[[16]](#endnote-16)]，这意味着Android将在出货量和市场份额全面超过苹果。而在国内，Android手机已然处于霸主地位，数据显示，2014年上半年，在中国智能手机市场总销量中，采用Android系统的手机占比超过8成，这意味着每销售10部智能手机，就有8部Android智能手机。与其他移动操作系统相比，Android系统具有五大优势，分别为：

1. 开放性

与ios、window phone、symbian等移动操作系统不同，Android系统具有完全的开源性，任何人都可以快捷方便的从互联网获取Android系统所有发布版本的源代码，并且移动端厂商和移动软件开发者不需要担心任何专利权障碍，可以快速加入到Android联盟中来。基于上述原因，Android平台在极短的时间里积累了大量的人气，厂商、开发者和消费者共同的热情，持续推动着Android系统的进步。

1. 不受运营商的束缚

在智能移动操作系统出现以前，通常是运营商控制着手机的网络和功能，用户无法自主安装手机应用。随着ios、symbian等智能移动操作系统的出现，手机的网络功能大大提升，运营商的控制权得到了一定的削弱。随着Android系统发布以及2G、3G和4G移动网络的出现，用户彻底摆脱了运营商的限制。

1. 丰富的硬件

Android不仅在软件方面有着极大的优势，同时在硬件方面也有着更多的选择，硬件厂商可以通过定制Android源代码，推出更具有特色性的功能或者产品，给用户提供了更多的选择。

1. 便利的开发环境

Android为应用程序的开发提供了完整的开发平台和全面的开发工具类库，任何个人或公司都可以通过这个完备的开发环境，创造出极具个性的应用程序。

1. 无缝结合的Google应用

Android系统是由Google公司组建的开放手机联盟共同开发管理的，因此采用Android系统的手机中包含了Google提供的各种功能和服务，例如Google地图、Google邮件和Google搜索等。这些优质的应用为用户提供了极大的方便。

### 2.1.2 Android系统架构

和其他操作系统相同，Android系统也采用了层次化架构，从顶层到底层共分为4层，分别是应用程序层、应用程序框架层、系统运行库层和Linux内核层[[[17]](#endnote-17)]。具体架构如图2.1所示。

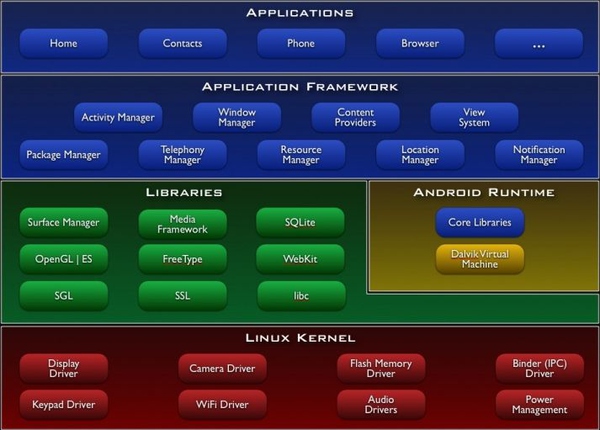


图2.1 Android平台系统架构图

1. 应用程序层

Android中使用的所有应用程序都位于这一层，比如系统自带的短信、联系人等系统程序，或者第三方开发的所有应用程序。

1. 应用程序框架层

构建Android应用时用到的SDK类库和一些未公开接口的类库都位于这一层，这些类库为开发人员提供了快捷的开发环境。

1. 系统运行库层

这一层包括了一个由C/C++代码编写的核心库，该核心库的作用是为应用程序框架层的各个组件提供支持。

1. Linux内核层

Android系统的底层为Linux内核，Linux内核主要负责管理硬件以及为上层提供核心系统服务，其具体功能主要有：内存管理，进程管理，网络协议栈和驱动管理。

## 2.2 Android应用的组成结构

Android应用程序的安装包为一个后缀名为.apk的文件，一般称为apk文件。apk文件其实是一个压缩文件包，格式为zip，其内部包含二进制代码、资源文件、清单文件等。通常一个apk文件解压后包含以下几个文件：

1）AndroidManifest.xml：Android应用的全局配置文件，它向Android系统介绍了本应用的很多配置信息，例如应用的名字、版本、申请的权限信息和启动入口界面等。对于一个apk文件，该文件是必须存在的。

2）classes.dex文件：二进制代码文件，是在程序编译时由Java源码编译后生成的Dalvik字节码文件。

3）META-INF目录：存放了apk中所有文件的签名信息，用来防止apk文件被恶意篡改。

4）res目录：该目录存放了各种资源文件。

5）lib目录：该目录存放了本文的研究对象SO文件，SO文件是用C或者C++语言编写并通过NDK工具编译而成的。Android应用通常通过JNI机制调用SO文件中的代码。

6）assets目录：和res目录类似，该目录也存放了各种资源文件，不同的是，assets目录下可以添加任意深度的子目录，而res目录不支持。

7）resources.arsc：编译后的二进制资源文件。

## 2.3 JNI机制与NDK

JNI，全称为Java Native Interface，中文译为Java本地调用。其作用是为运行在Java虚拟机（VM）内部的Java代码和运行在底层的C/C++代码提供交互支持[[[18]](#endnote-18)]。

对应Android平台，JNI的一般用途为：

1）对效率要求较高的场景，某些情况下，程序逻辑较为复杂或需要大量的运算，此时就可以使用JNI，这是因为C/C++比Java效率更高。

2）平台移植的需求，如果在其他平台（C/C++）上已经有成熟的代码库，对这些代码稍加改动就可以移植到Android平台，再利用JNI调用对应代码即可，这样可以节省大量的重复工作时间。

3）安全性，由于Java语言本身的特性，导致攻击者很容易破解由Java语言编写而成的Dex文件，而SO文件具有更好的安全性，使用JNI大大提高了破解的难度。

NDK，全称为Native Development Kit，它是一系列工具的集合，通过这些工具，开发者可以快速开发C/C++代码并编译为SO文件，并能自动将SO文件放入apk文件的lib目录中，极大的简化了开发者的工作[[[19]](#endnote-19)]。NDK的几个重要功能如下：

1）可以将C/C++源码编译成适用于Android平台的二进制文件（主要为SO文件）。

2）能够自动将编译后的SO文件打包到Android应用程序安装包中。

3）提供NDK开发相关的文档、实例及规范等。

终上所述，通过NDK工具，可以使用C/C++语言来编写代码并编译成SO文件，以及将SO文件打包进apk文件中；通过JNI机制，可以在应用程序的Java代码中调用SO文件中的相关函数。

## 2.4 ELF文件格式概述

ELF，全称为Executable and Linkable Format，中文译为可执行连接文件格式，是Linux系统的常用可执行文件格式[[[20]](#endnote-20)]。本文研究的对象SO文件即为ELF文件中的一种类型，因此研究ELF文件格式是非常必要的。为了更加直观的理解SO文件格式，本文使用NDK安装目录中提供的libhello-jni.so作为演示例子，并且使用GNU binutils包中的readelf工具解析。

### 2.4.1 ELF文件类型

ELF文件可以被标记为下列5种类型：

1）ET\_NONE：这是一个未知类型。它表示文件类型未知，或尚未被定义。

2）ET\_REL：这是一个可重定位文件。可重定位文件包含代码节和数据节，通常是作为链接器的输入，输出为可执行文件、共享库文件或其他可重定位文件。典型的可重定位文件是以.o结尾的文件。

3）ET\_EXEC：这是一个可执行文件。又称为程序，包含一个程序执行的入口点。典型的可执行文件是以.out结尾的文件。

4）ET\_DYN：这是一个共享库文件，通常是由进程动态加载进内存中。典型的共享库文件是以.so结尾的文件。

5）ET\_CORE：这是一个核心文件。它是在进程崩溃或当进程收到SIGSEGV信号（段错误信号）时，生成的一个包含进程的内存映像的文件。GDB调试工具可以解析核心文件或调试核心文件，以确定是什么原因导致程序崩溃。

其中，本文的研究对象SO文件属于类型为ET\_DYN的ELF文件。

### 2.4.2 文件格式

ELF文件的文件格式包括两种并行视图，分别描述了两个过程的不同需求，如图2.2所示。



图2.2 目标文件格式

在编译和链接阶段，ELF文件被视为节头描述的全部节的集合，即为链接视图；在加载阶段，ELF文件被视为程序头描述的全部段的集合，即为执行视图。

### 2.4.3 数据表示形式

在ELF的文件结构定义里面会使用多种数据类型，表2-2列出了后续将要出现的数据类型的定义。

表2-2 ELF文件中的数据表示形式

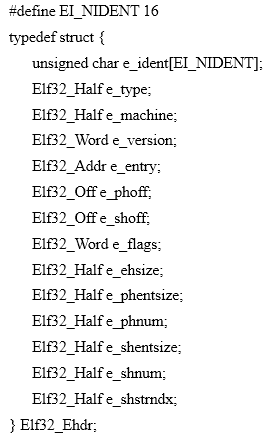
|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **大小** | **对齐** | **目的** |
| Elf32\_Addr | 4 | 4 | 无符号程序地址 |
| Elf32\_Half | 2 | 2 | 无符号中整数 |
| Elf32\_Off | 4 | 4 | 无符号文件偏移 |
| Elf32\_Sword | 4 | 4 | 带符号整数 |
| Elf32\_Word | 4 | 4 | 无符号整数 |
| Unsigned char | 1 | 1 | 无符号小整数 |

从表2-2中可以看到，目标文件中的所有数据格式都必须符合对齐规则，并且所有数据格式的大小和对齐均相同，如Elf32\_Addr的大小为4字节，对齐也为4字节。这样使得数据结构可以包含显示填充。

## 2.5 ELF头

ELF头位于ELF文件的起始处，描述了文件其他部分的结构。包含的信息有：系统相关信息，文件类型，段相关信息，节相关信息。

ELF头是由Elf32\_Ehdr类型的结构数组组成，该类型的结构定义如下：



使用readelf –h命令读出整个ELF头的内容，如图2.3所示：



图2.3 ELF头部

这里只讨论系统相关信息和文件类型，而段相关信息和节相关信息会在本文后续介绍。

Magic（e\_ident）一共包含16个字节，其中前4个字节为魔术字（Magic Number），其值按顺序次为0x7f、0x45、0x4c、0x46，也就是”\127ELF”，表示这是一个ELF文件，所有ELF文件的魔术字都是相同的。Class（e\_ident的第5个字节）表示该文件适用于32位架构。Data（e\_ident的第6个字节）表示文件的数据存储方式，该文件为小端模式。Version（e\_version）表示文件的版本，目前只有EV\_CURRENT（1）有效。Type（e\_type）表示文件的类型，该文件是共享库文件，也就是SO文件。Machine（e\_machine）表示文件支持的体系结构，该文件适用于ARM体系。Entry point address（e\_entry）表示程序入口点，对于共享目标文件来说，不需要程序入口点，因此e\_entry是未使用的，默认为0。Flags（e\_flags）表示与文件相关的，特定与处理器的标志。

ELF头部是ELF文件的基本部分，作为整个ELF文件的索引地图，系统在加载ELF文件到内存中时，会首先检测并读取ELF头部，从而加载文件剩余部分。然而由于文件的复杂性，实际加载过程中，部分头部成员是不被使用的，也就是可以被篡改的。例如，目前只对e\_ident的前6个字节进行校验，壳程序可以自定义e\_ident的后10个字节；SO文件作为共享库文件，本身不需要程序入口点，故壳程序可修改e\_entry。

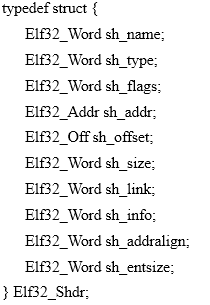
## 2.6 节

节是ELF文件中用以装载内容数据的最小容器，不同名称的节其装载的内容的性质也是不一样的。图2.3中，Start of section headers（e\_shoff）表示节头与文件第一个字节之间的偏移（以字节为单位）。Size of section headers（e\_shentsize）表示节头的表项所占用的字节数。Number of section headers（e\_shnum）表示节头的表项数目。

### 2.6.1 节头

节头包含文件各节的基本信息，节头是用于程序链接阶段的，在链接阶段，链接器认为ELF文件是由各种节组成的，不同的节保存着不同的内容，节头中的每个表项都对应了一个节，因此链接器可以通过解析节头得到每个节的数据。

节头是由Elf32\_Shdr类型的结构数组组成，该类型的结构定义如下：



使用readelf –S命令读出整个节头的内容，如图2.4所示：

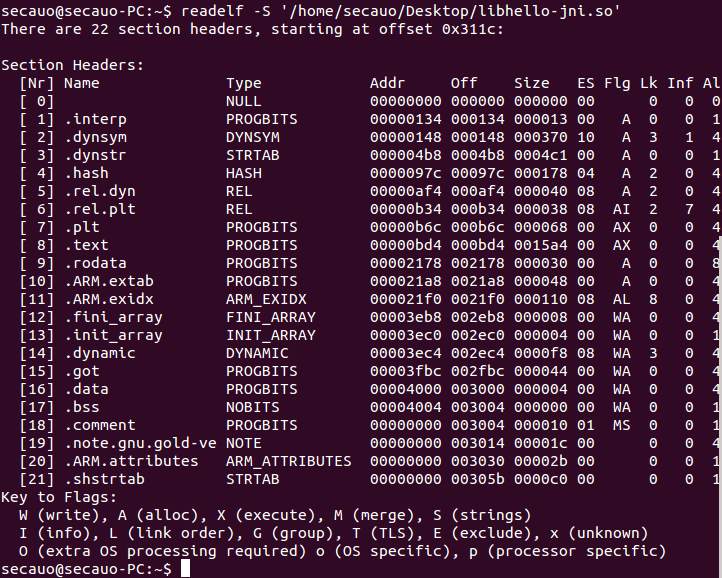


图2.4 节头

从图2.4可知，libhello-jni.so中包含了22个节，与ELF头的e\_shnum一致。Name（sh\_name）为节名称，sh\_name为节名称在字符串表的索引。Type（sh\_type）表示了节的类型，节的类型决定了节的内容，表2-3列出了常见的节类型。Addr（sh\_addr）表示了节的首字节映射到内存中的虚拟地址。Off（sh\_offset）表示了节的首字节在文件中的偏移。Size（sh\_size）表示了节在文件中所占用的字节数。ES（sh\_entsize）只对某些类型的节有用，表示了这些节中所包含的固定大小的项目的个数。Lk（sh\_link）和Inf（sh\_info）分别表示了节头索引和节的附加信息。Al（sh\_addralign）描述了节在文件中是如何对齐的。

表2-3 常见的节类型

|  |  |  |
| --- | --- | --- |
| **名称** | **取值** | **含义** |
| SHT\_NULL | 0 | 未使用，没有定义成员值。 |
| SHT\_PROGBITS | 1 | 标识由程序定义的信息，这些信息的格式和含义完全由程序确定。 |
| SHT\_SYMTAB | 2 | 标识符号表。 |
| SHT\_STRTAB | 3 | 标识字符串表。 |
| SHT\_RELA | 4 | 标识包含显式加数的重定位项。 |
| SHT\_HASH | 5 | 标识符号散列表。 |
| SHT\_DYNAMIC | 6 | 标识动态链接的信息。 |
| SHT\_NOTE | 7 | 标识以某种方法标记文件的信息。 |
| SHT\_REL | 9 | 标识不包含显式加数的重定位项。 |
| SHT\_SHLIB | 10 | 该节类型保留但暂未指明语义。 |
| SHT\_DYNSYM | 11 | 与符号表不同，该表只包含一组尽可能少的动态链接符号。 |
| SHT\_INIT\_ARRAY | 14 | 标识包含指针数组的节，这些指针指向初始化函数。 |
| SHT\_FINI\_ARRAY | 15 | 标识包含指针数组的节，这些指针指向终止函数。 |

注意图2.4中的Flg（sh\_flag），表示了节的相关标志。其中W（write）表示节里存储的数据是可写的，X（execute）表示节里存储的是机器指令。A（alloc）表示节会被加载器加载到内存中。而与此相反，那些没有A（alloc）标志的节只是用于SO文件的链接阶段或者用于其他逆向工具的解析，而不会被加载到内存中。当加载SO文件时，加载器会加载那些具有A（alloc）标志的节，而其他节会被继续留在SO文件内。

另一方面，从执行视图来看，加载器认为SO文件是由各个段构成的，而节头是可以忽略的。因此，ELF头中的节相关成员都是可以抹去的，例如e\_shoff，e\_shentsize，e\_shnum。

值得注意的是，SO文件是默认包含所有信息的，但是可以使用strip工具将不具有A（alloc）标志的节从SO文件中删除掉，或者抹去ELF头中的节信息。处理后的SO文件依然能正常运行，只不过调试器和反汇编程序可利用的信息较少，readelf、objdump、ida等工具也无法分析处理后的SO文件，这是因为这些工具的逆向分析工作都依赖于节头。因此，大部分壳程序都会利用这种手段来对抗逆向工程。

### 2.6.2 特殊节

表2-4列出了libhello-jni.so文件中的特殊节的名称、类型和内容解释。

表2-4 典型SO文件的节

|  |  |  |
| --- | --- | --- |
| **类型** | **名称** | **内容** |
| SHT\_PROGBITS | .interp | 程序的解释程序的路径。 |
| SHT\_DYNSYM | .dynsym | 动态符号表。 |
| SHT\_STRTAB | .dynstr | 动态链接时需要的字符串。 |
| SHT\_ | .hash | 标号的散列表。 |
| SHT\_REL | .rel.dyn | 重定位相关信息 |
| SHT\_REL | .rel.plt | 重定位相关信息 |
| SHT\_PROGBITS | .plt | 过程连接表。 |
| SHT\_PROGBITS | .text | 机器指令，常值数据和常值串。 |
| SHT\_PROGBITS | .ARM.extab | 重建函数现场相关的信息。 |
| ARM\_EXIDX | .ARM.exidx | 重建函数现场相关的信息。 |
| SHT\_PROGBITS | .rodata | 只读数据，可以读取但不能修改。 |
| FINI\_ARRAY | .fini\_array | 终止代码队列。 |
| INIT\_ARRAY | .init\_array | 初始化代码队列 |
| SHT\_DYNAMIC | .dynamic | 与动态链接相关的信息。 |
| SHT\_PROGBITS | .got | 全局偏移量表。 |
| SHT\_PROGBITS | .data | 初始化数据。 |
| SHT\_NOBITS | .bss | 未初始化数据。 |
| SHT\_PROGBITS | .comment | 版本控制信息。 |
| SHT\_NOTE | .note.gnu.gold-version | gold连接器的版本信息。 |
| ARM\_ATTRIBUTES | .ARM.attributes | ARM处理器相关的信息。 |
| SHT\_STRTAB | .shstrtab | 节名称。 |

表2-4列出的节名称均有前缀点（.），这意味着这些节名都是系统保留的，这些节都具有特定的内容，并且用于SO文件的加载与运行。本文对与研究内容密切相关的字符串表节、符号表节、重定位节、动态节和初始化节进行了详细研究。

### 2.6.3 字符串表节

顾名思义，字符串表节的内容为一系列字符串，这些字符串是用来描述符号和节名的。

使用readelf –x命令读出字符串表节的内容，如图2.5所示。



图2.5 .dynstr字符串表节

图2.5命令中的3是.dynstr节在节头中的索引值，从图2.4中可以找到。图2.5中的输出结果可读性不高，表2-5为上述字符串表节的示例。

表2-5 .dynstr字符串表节示例

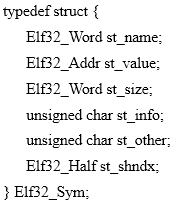
|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 索引 | +0 | +1 | +2 | +3 | +4 | +5 | +6 | +7 | +8 | +9 |
| 0 | \0 | \_ | \_ | c | x | a | \_ | f | i | n |
| 10 | a | l | i | z | e | \0 | \_ | \_ | c | x |
| 20 | a | \_ | a | t | e | x | i | t | \0 |  |

如表2-5所示，第一个字节为空字符，最后一个字节也为空字符，从而保证每个字符串均已空字符结尾。

字符串表节在程序运行的时候是不必要的，但是其对分析人员来说是非常必要的，因为自然形式的字符串更加容易理解。因此，一些较强的壳程序会抹去字符串表节的内容，这样在使用IDA等反汇编工具的时候，将看不到字符串形式的函数名字，这使得逆向分析更加困难。

### 2.6.4 符号表节

程序源代码中的常量、变量和函数在编译之后都会变成相应的符号，这些符号就存放在符号表节中，它们通常用于定位和重定位。符号表索引是指符号在符号表节中的下标，例如0表示符号表节中的第一个符号，该符号是未定义的。符号表节是由Elf32\_Sym类型的结构数组组成，该类型的结构定义如下：



使用readelf –s命令读出整个符号表节的内容，如图2.6所示。

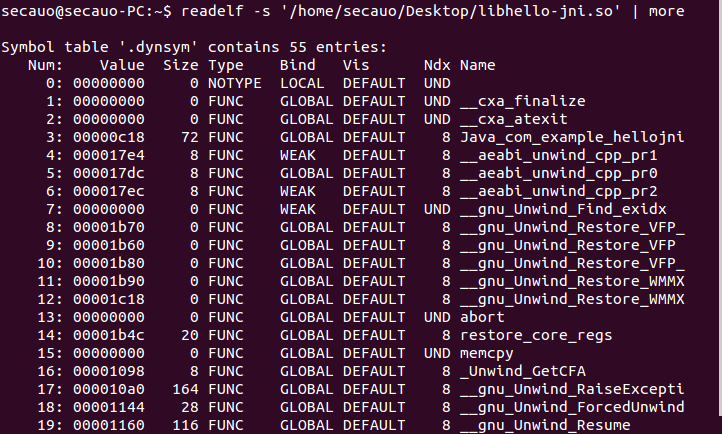


图2.6 .dynsym符号表节

Value（st\_value）表示关联符号的值，可能是绝对值或地址，对于SO文件来说，表示的是运行时的内存地址。Size（st\_size）表示符号的大小。Type表示符号的类型，Bind表示符号的绑定属性，两者均由st\_info定义。Vis（st\_other）在目前版本暂未使用，无含义。Ndx（st\_shndx）表示相关节头索引，如果为Ndx为UND则表示该符号在其他模块定义，以后需要重定位。Name（st\_name）表示符号名在字符串表节的索引。

图2.6中的Type，表示符号的类型，通过如下的宏转换得到相应的值：

#define ELF\_ST\_TYPE(info) ((uint32\_t)(info) & 0xf)

在SO文件中，常见的符号类型有4种，符号类型的名称，值，含义如表2-6所示：

表2-6 常见的符号类型

|  |  |  |
| --- | --- | --- |
| **名称** | **取值** | **含义** |
| STT\_NOTYPE | 0 | 无类型，没有具体含义 |
| STT\_OBJECT | 1 | 表示符号类型为数据对象。 |
| STT\_FUNC | 2 | 表示符号类型为函数。 |
| STT\_SECTION | 3 | 表示该符号和节相关，用于重定位。 |

图2.5中的Bind，表示符号的绑定属性，通过如下的宏转换得到相应的值：

#define ELF\_ST\_BIND(info) ((uint32\_t)(info) >> 4)

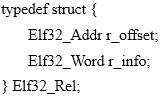
在SO文件中，常见的符号的绑定属性有3种，其名称，值，含义如表2-7所示。

表2-7 常见的符号绑定属性

|  |  |  |
| --- | --- | --- |
| **名称** | **取值** | **含义** |
| STB\_LOCAL | 0 | 局部符号。 |
| STB\_GLOBAL | 1 | 全局符号。 |
| STB\_WEAK | 2 | 弱符号，类似于全局符号，但是其优先级要低一些。 |

### 2.6.5 重定位节

SO文件在加载进内存后，还需要通过重定位节修正其符号的内存地址。例如，在编译阶段，编译器并不知道一个外部函数的地址，此时会使用0占位，并且在此处绑定一个符号。当加载SO文件时，系统会利用重定位节中的相关信息，填入正确的地址[[[21]](#endnote-21)]。在x86和ARM架构中，重定位节是由Elf32\_Rel类型的结构数组组成，该类型的结构定义如下：



使用readelf –r命令读出整个重定位节的内容，如图2.7所示。

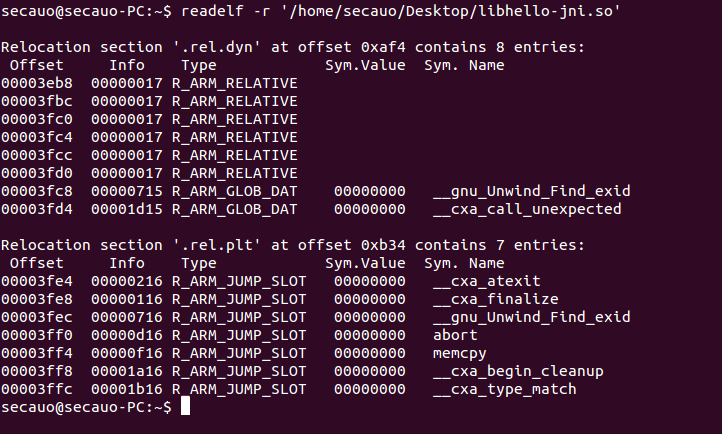


图2.7 重定位节

从图2.7可知，libhello-jni.so包含两个可重定位节，分别为.rel.dyn节和.rel.plt节。其中.rel.dyn节重定位的地方在.got表内，主要是针对外部数据变量符号，例如全局数据。定位过程：获得符号对应value后，根据rel.dyn节中对应的offset，修改.got表对应位置的value；而rel.plt节重定位的地方在.got.plt表内，主要是针对外部函数符号，一般是函数首次被调用时候重定位。定位过程：获得符号对应value后，根据rel.plt节中对应的offset，修改. got.plt表对应的value。

Offset（r\_offset）表示应用重定位行为的地址，对于SO文件来说，就是该地址单元的运行时地址。Info（r\_info）表示具有受重定位影响因素的符号表索引和重定位应用的类型。Info（r\_info）包含了重定位符号在本地符号表中的索引和重定位类型

图2.7中的Type表示重定位类型，通过如下的宏转换得到相应的值：

#define ELF32\_R\_TYPE(info) ((info) & 0xff)

重定位类型意味着如何去修改受影响的地址单元，也就是按照何种方式去计算需要最后放在受影响单元里面的值[[[22]](#endnote-22)]。对于适用于ARM架构的SO文件，常见的重定位类型有5种，重定位类型的名称，取值，含义如表2-8所示。

表2-8 常见的重定位类型

|  |  |  |
| --- | --- | --- |
| **名称** | **取值** | **含义** |
| R\_ARM\_JUMP\_SLOT | 22 | 该属性的重定位地址一般位于.rel.plt节，重定位后其offset指向调用函数的地址。 |
| R\_ARM\_GLOB\_DAT | 21 | 该属性的重定位地址一般位于.got节，重定位后其offset指向调用函数的地址。 |
| R\_ARM\_ABS32 | 2 | 该属性的重定位地址一般位于.rel.dyn节，其对象通常为通过全局函数指针的方式调用的外部函数。 |

图2.7中的Sym.Value和Sym.Name分别表示重定位符号的值和名称，通过如下的宏转换得到相应的值：

#define ELF64\_R\_SYM(info) ((info) >> 32)

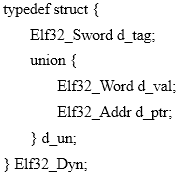
重定位的大概流程如下：

1）遍历某个重定位节（例如图2.7中的.rel.dyn节或.rel.plt节），遍历每一个重定位项目，执行以下处理。

1. 求出需要进行重定位的地址。
2. 求出重定位的目标地址。
3. 求出重定位类型，并根据不同的类型，执行重定位操作。

### 2.6.6 动态节

动态节包含了进程动态链接过程中需要的信息，表2-4中类型为SHT\_DYNAMIC的.dynamic节即为动态节。动态节中包含各种数据，驻留在该节头的结构包含了其他动态链接信息的地址。动态节是由Elf32\_Dyn类型的结构数组组成，该结构类型定义如下：



使用readelf –d命令读出整个动态节，如图2.8所示。

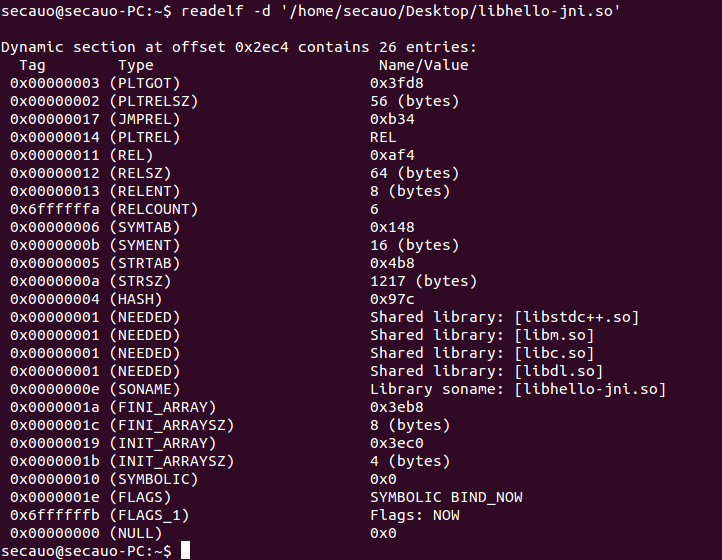


图2.8 动态节

从图2.8可知，libhello-jni.so的动态节包含了24个入口。Tag（d\_tag）表示动态节的类型，控制着d\_un的解释。Type即为动态节的类型。Name/Value由d\_tag和d\_un共同控制，其中d\_val表示了具有不同解释的整形变量，d\_ptr表示了一个虚拟地址，通常用于重定位。

### 2.6.7 初始化节

在SO文件完成装载和链接过程以后，Android会自动执行SO文件的初始化代码队列。初始化代码位于.init节和.init\_array节中，并且所有的初始化函数都会在可执行文件得到控制之前发生。

.init\_array节中的数据可以通过IDA来进行查看，如图2.9所示。

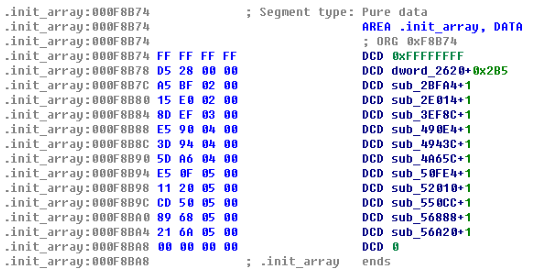


图2.9 .init\_array节

由图2.9可知，.init\_array包括几个指针：0xffffffff、0x00000000以及普通指针。有些数组的值为0或者-1的，需要忽略。

由于.init\_array中指针指向的代码会首先运行，基于这一特性，大多数壳程序会将壳代码放入.init\_array中，从而首先获得程序的控制权。

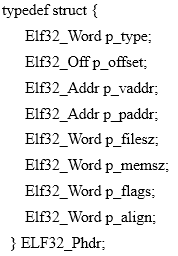
## 2.7 段

一个单独的段通常会包含几个不同的节，例如一个可加载、可读以及可执行的段通常包括动态字符串节.dynstr、动态符号节.dynsym、代码节.text等。图2.2中，Start of program headers（e\_phoff）表示程序头与文件第一个字节之间的偏移（以字节为单位）。Size of program headers（e\_phentsize）表示程序头的表项所占用的字节数。Number of program headers（e\_phnum）表示程序头的表项数目。

### 2.7.1 程序头

程序头告诉系统如何创建一个进程镜像，它是从加载执行的视角来看待ELF文件的，从它的角度来看，ELF文件被划分为不同的段，所有信息都是以段的形式存放，每个段在程序头中都有对应的表项来描述。

程序头是由Elf32\_Phdr类型的结构数组组成，该结构类型定义如下：



使用readelf –l命令读出整个程序头，如图2.10所示。

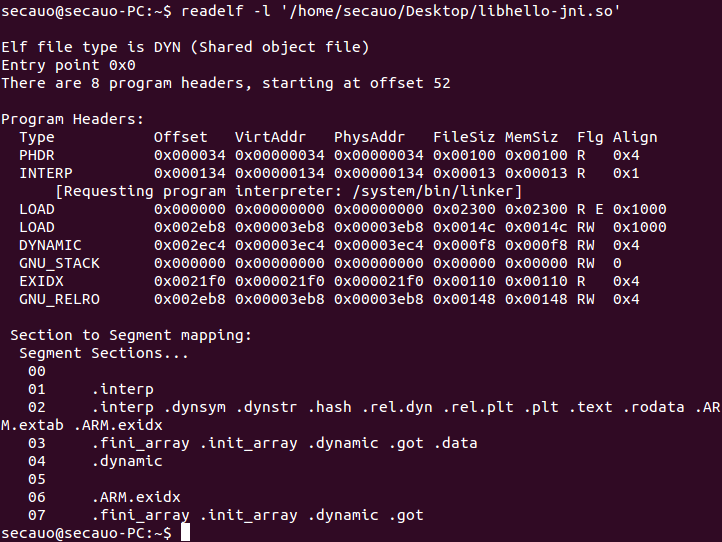


图2.10 程序头

从图2.10可知，libhello-jni.so包含8个段。Type（p\_type）表示段的类型，或者如何解释此数组元素的信息，表2-9列出了常见的段类型。Offset（p\_offset）表示段的首字节在文件中的偏移（以字节为单位）。VirtAddr（p\_vaddr）表示段的首字节映射到内存中的虚拟地址。PhysAddr（p\_paddr）表示段的物理地址，Android系统并未使用该成员。FileSiz（p\_filesz）表示段在文件映像中所占用的字节数。MemSiz（p\_memsz）表示段在内存映像中所占用的字节数。Flg（p\_flags）表示段的属性，也就是R（可读），W（可写）和X（可执行）三种属性。例如图中类型为DYNAMIC的段是可读、可写的。Align（p\_align）描述段在文件和内存中是如何对齐的。

表2-9 常见的段类型

|  |  |  |
| --- | --- | --- |
| **名称** | **取值** | **含义** |
| PT\_NULL | 0 | 未使用，没有定义成员值。 |
| PT\_LOAD | 1 | ELF文件装载时，需要加载的段。 |
| PT\_DYNAMIC | 2 | 保存了由动态连接器（即 类型为PT\_INTERP的段中指定的解释器）使用的信息。 |
| PT\_INTERP | 3 | 指定程序从可行性文件映射到内存之后，必须调用的解释器。 |
| PT\_NOTE | 4 | 指定辅助信息的位置和大小。 |
| PT\_PHDR | 6 | 保存程序头部。 |

### 2.7.2 可加载段

可执行文件或共享目标文件有且至少包含一个类型为PT\_LOAD的段。这种类型的段被描述为可装入段，这意味着该段是将要被加载或映射到内存。例如，对于SO文件来说，其通常包含以下两个类型为PT\_LOAD的段：

1. 包含机器指令的代码段。
2. 包含全局变量和动态链接信息的数据段。

上述两个段将会被映射进内存，映射初始位置由p\_vaddr成员确定，并且由p\_align成员确定对齐值。加载大小为p\_memsz。

### 2.7.3 动态段

类型为PT\_DYNAMIC的段又称为动态段，该段专用于动态链接并包含了链接器所需要的信息，其程序头元素指向.dynamic节。关于动态段的详细内容请参考2.6.6节。

## 2.8 本章小结

本章系统介绍了Android平台与JNI机制，深入分析了SO文件格式，即ELF文件格式，并且详细介绍了与本文相关的特殊节区，分别为字符串表节、符号表节、重定位节、动态节和初始化节。为后续壳检测与脱壳技术的研究提供了技术支撑。

# 第3章 基于朴素贝叶斯分类器的壳检测技术

本章采用了一种基于朴素贝叶斯分类器的壳检测方法，该方法对已知壳和未知壳均有较高的检测率。

## 3.1 加壳文件

### 3.1.1 壳的概念

自然界中，壳可以用来保护植物的种子，也可以用来保护动物的身体。同理，在计算机领域中，壳可以用来保护软件以防止其被逆向分析。这里的壳指的是一段先于程序运行且专门用来保护软件的代码。由于这段代码和自然界的壳的目的是一样的，因此这段代码又被称为“壳”[[[23]](#endnote-23)]。

加壳是改变原程序的执行逻辑以及对其数据进行加密保护的过程[[[24]](#endnote-24)]。加壳的目的一般有两个，其一是缩小原程序的文件体积，以减小其占用的本地空间和节约网络带宽。其二是防止原程序被逆向分析。

### 3.1.2 流行的加壳技术

本文通过对现有壳的分析总结，将流行的加壳技术分为三类，分别为基于JNI机制的加壳技术[[[25]](#endnote-25)]，基于特定节和特定函数的加壳技术和基于自定义头部的加壳技术。

基于JNI机制的加壳技术，该方案基于对Android应用现有的调用SO文件的JNI运行机制的研究与分析，对原SO文件进行整体加密，加壳SO文件在运行阶段，对原SO文件进行解密加载，并对全部JNI函数进行重新绑定。此方法对于JNI机制依赖性强，保护性较弱。通过拦截获取壳运行过程中解密在本地的原SO文件即可完成脱壳。

基于特定节和特定函数的加壳技术，是指将SO文件的特定节或特定函数进行加密，在SO文件加载时完成解密。该方案又分为基于特定节的加解密和基于特定函数的加解密，二者都是通过解析SO文件结构，对指定代码区域进行加密，并提前在源码中插入解密代码，区别在于前者加解密的对象是节，后者加解密的对象为函数块。该方案加壳强度较高，缺点是只适用于有源码的情况。

基于自定义头部的加壳技术，SO文件在装载时，加载器不会使用文件头的e\_shoff、e\_shentsize、e\_shnum和e\_shstrndx这4个字段，也就是说这4个字段是可以被修改的，被修改之后，使用readelf和IDA等工具会解析失败，但该SO文件仍可以正常运行。该方案正是基于此原理，其实现简单，可以有效对抗静态分析。

其中，基于JNI机制的加壳技术由于保护性较弱，实际应用较少。而后两种加壳技术往往会结合在一起运用，使用后两种方案结合的加壳SO文件能够有效阻止程序被静态分析和动态调试。

值得注意的是，后两种加壳方案都对SO文件结构进行了处理，这使得加壳SO文件与未加壳SO文件在SO文件结构上出现了差异。

### 3.1.3 未加壳文件与加壳文件的差异

未加壳文件是指C/C++源码通过NDK工具编译生成的SO文件，加壳文件是指各个加壳工具对未加壳文件进行加壳后生成的SO文件。虽然二者从程序运行上看几乎是完全一致的，具有相同的实际功能。然而本文通过对大量未加壳样本与加壳样本进行对比，发现未加壳文件与加壳文件的SO文件格式存在很大的不同，下面本文将会举例说明。

从前文的研究可知，Android在加载SO文件时，SO文件的ELF头部分成员并未使用，而大多数加壳文件会对这些成员进行修改，具体表现有：未加壳文件的Entry point address为默认值0，而某些加壳文件会修改Entry point address以用来存储壳相关信息。其次，加壳文件可能修改了Flags的值；最后，加壳文件还会抹去或者篡改头部的节信息。如图3.1和图3.2所示。

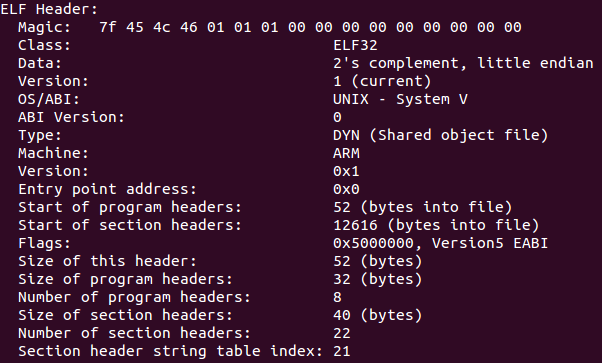


图3.1 未加壳SO文件的头部内容

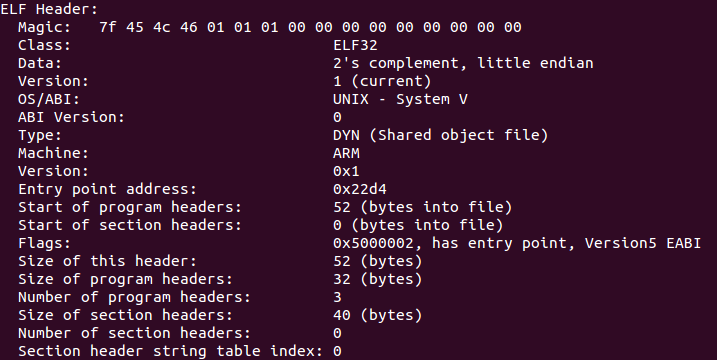


图3.2 某加壳SO文件的头部内容

某些加壳文件还会修改段信息，具体表现有：首先，未加壳文件通常有8个段，而加壳文件只有3个段；其次某些加壳文件包含一个属性为可读、可写、可执行的段，而未加壳文件没有；最后，加壳文件由于抹去了头部的节信息，因此也缺少section to segment mapping部分的内容。如图3.3和图3.4所示。

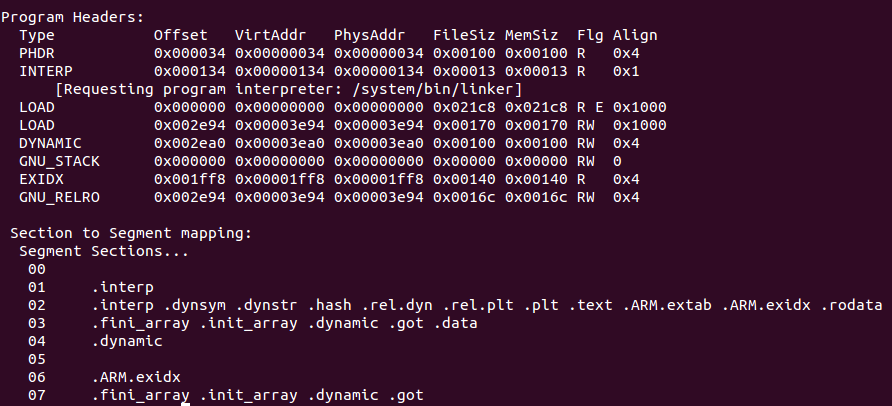


图3.3 未加壳SO文件的段内容

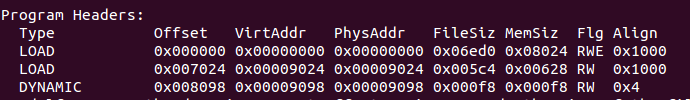


图3.4 某加壳SO文件的段内容

壳代码通常位于init\_array节中，因此加壳文件的ini\_array数组的长度往往更长。如图3.5和图3.6所示。



图3.5 未加壳SO文件的init\_array



图3.6 加壳SO文件的init\_array

## 3.2 壳检测技术的框架设计

### 3.2.1 设计思想

1）检查的具体需求

所谓的需求是指，当给定一个待检测样本时，在判定该样本是否加壳的基础上，是否还需要返回表示壳的种类，如返回壳的编号或者壳名称。这取决于研究的目的，本文最终的研究目的是对一个加壳样本进行脱壳，而壳检查技术只是达到这一目的的手段。另一方面，本文研究的是一种通用性的脱壳方法，该方法无需得知壳的具体类型，只需要得知样本是否加壳，即可完成后续的脱壳工作。

因此，本文壳检查技术的需求是判定一个待检查样本是否加壳，而不用知道壳的具体类型，返回真表示加壳，返回假表示未加壳。

2）如何判断是否加壳

对于一个经验丰富的分析人员来说，判断一个SO文件是否加壳判断方法多种多样，而且通常而言非常容易。但是如果将这些方法一一列举出来后，对于程序来模拟人工分析也存在下面2个问题：首先，大部分方法实现起来比较复杂，例如一个加壳SO文件通常包含较多的加密解密代码，这对于具有一定技术水平的分析人员而言，是较为容易发现的，而如果用程序来实现，则较为困难。其次，有些方法误判率非常高，例如SO文件中包含了一些特征字符串，特征字符串对于分析人员有着极大的启发作用，能有效帮助判断SO文件是否加壳，而如果简单的认为包含了该特征字符串即为加壳SO文件，又会导致极高的误判率。

本文借鉴恶意代码检测的相关方法，将机器学习的相关理论应用到壳检测领域中，采用了一种基于朴素贝叶斯分类器的壳检测方法。

3）特征集的问题

分类器的训练与分类的效果很大程度上取决与SO文件的特征集的好坏。

特征集的选择一般要经过2个步骤，分别为：

（1）特征抽取，分为静态特征抽取和动态特征抽取两种方案[[[26]](#endnote-26)]，静态提取是指在不加载SO文件的前提下利用相应的解析工具获取SO文件的文件结构或代码静态特征。动态提取是指通过加载SO文件，记录其在运行时的相关行为特征，一般有两种技术用于动态提取，其一为虚拟化技术，其二为动态跟踪分析技术。

（2）特征变换，一般情况下，通过特征抽取获得的特征中存在很多的冗余特征或无用特征，如果直接将抽取的特征用于后续的分类工作会影响分类器的分类效果。因此，特征变换是十分必要的。特征变换的作用是通过一种数学方法将输入特征中冗余或无用的特征剔除掉，最终得到维度更低的特征。常用的特征变换方法有：主成份分析法，线性判别分析法等。

本文的特征抽取方案为基于文件结构特征的静态特征抽取方案，特征变换采用了LDA算法。

4）分类算法的选择

在获得经过降维的特征集后，接着就是利用特征集对分类器进行训练，经过训练后的分类器就可以用于壳检测工作了。

常用的分类算法有SVM分类算法、决策树分类算法、朴素贝叶斯分类算法、人工神经网络分类算法和KNN算法（K-Nearest Neighbour）等。表3-1为上述5种分类算法的优缺点[[[27]](#endnote-27)]。

表3-1 常见分类算法的优缺点

|  |  |  |
| --- | --- | --- |
| **分类算法** | **优点** | **缺点** |
| SVM分类算法 | 可用于样本数量较少的情况 | 对缺失数据敏感，对非线性问题没有通用解决方案。 |
| 决策树分类算法 | 分类过程十分直观，易于理解，不需要对特征进行降维变换。 | 难以处理缺失数据的情况，忽略了数据集中属性之间的相关性。 |
| 朴素贝叶斯分类算法 | 算法简单，对任何样本都有比较稳定的分类效果。 | 需要样本的属性值独立于样本的分类属性，分类决策存在错误率。 |
| 人工神经网络分类算法 | 具有较高得到分类准确率，针对复杂的非线性关系也能有很好的分类效果。 | 需要大量的参数，学习时间过长，甚至有可能失败。 |
| KNN算法 | 算法简单，适用于样本容量比较大的情况。 | 输出的可解释性不强，计算量较大。 |

本文综合考虑上述5种分类算法的优缺点，最终决定使用朴素贝叶斯分类算法。理由如下：人工神经网络分类算法的学习时间过长，并且存在可能失败的风险；KNN算法输出的可解释性不强，而本文需要得到可解释性强的分类结果。SVM分类算法和决策树分类算法均需要完备的数据集，对缺失数据较为敏感，而由于时间关系，本文提供的样本数量较少，极有可能出现缺少数据的情况。

### 3.2.2 总体框架

基于朴素贝叶斯分类器的壳检测技术主要包括4大模块，分别为文件格式校验模块，特征模块，训练模块，壳检测模块。总体框架如图3.7所示。

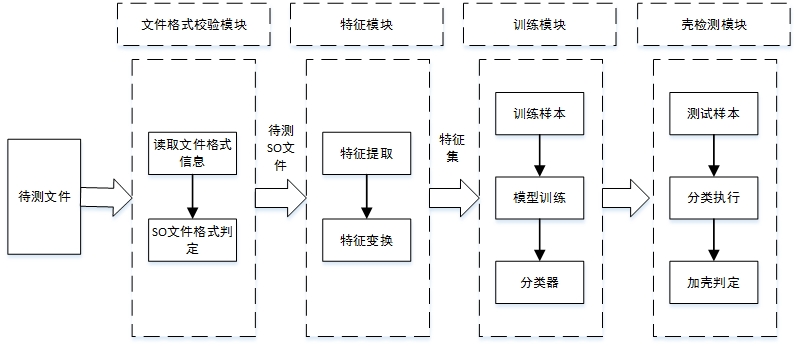


图3.7 壳检测总体框架

其中，文件格式校验模块的主要功能是检查待检测样本是否为有效的SO文件，目的是剔除掉无效的输入，避免产生后续无必要的工作量；特征模块的目的是抽取出SO文件的所有特征，接着基于线性变换的原理，对抽取出的特征进行变换，最终得到一组最具代表性的特征集；训练模块的功能是训练分类器，其目的是通过对大量样本的分类学习过程，得到一个较为准确的指标以作为分类依据；壳检查模块的目的是利用经过训练后的分类器对待检测样本进行分类以判定其是否加壳，该模块是壳检测框架的核心。

## 3.3 SO文件格式检测

本文壳检测方法的研究对象为32位的Android系统下的SO文件，故在进行壳检测之前，应该先判断待测文件是否为有效的SO文件。

SO文件格式检测方法比较简单，主要是头部某些数据进行校验，具体步骤如下：首先查看文件头部的四个魔术（Magic Number）是否有效，即是否为‘.ELF’；其次查看文件头部的第5个字节是否为ELFCLASS32，即判断是否是32位平台；查看文件头部的第6个字节是否为ELFDATA2LSB，即判断文件是否采用的小端格式；查看文件头部的e\_version成员的值是否为EV\_CURRENT，即判断文件的版本号是否有效；最后查看文件头部的e\_type成员的值是否为ET\_DYN，即判断文件类型是否为共享库；如果上述所有值全部匹配，则该文件是一个有效的SO文件。在SO文件格式中，ELFCLASS32、ELFDATA2LSB、EV\_CURRENT、ET\_DYN的值分别为1、1、1、3。

SO文件格式检测的伪代码如下所示，其中header的类型为ELF32\_Ehdr结构体。

读取待检测文件头部，并将其赋给header。

if (header.e\_ident[0] == 0x7f && header.e\_ident[1] == 0x45 && header.e\_ident[2] == 0x4c && header.e\_ident[3] == 0x46)

if (header.e\_ident[4] == ELFCLASS32)

if (header.e\_ident[5] == ELFDATA2LSB)

if (header.e\_version == EV\_CURRENT)

if (header.e\_type == ET\_DYN)

该文件是有一个有效的SO文件

return true;

return false;

## 3.4 特征集

### 3.4.1 特征抽取

本文根据对壳的先验知识，通过前文对SO文件格式的研究，解析SO文件结构信息，初步提取了表3-2所示的初选特征。

表3-2 SO文件可用分类特征

|  |  |  |
| --- | --- | --- |
| **特征描述** | **数据类型** | **数量** |
| ELF头 | integer | 12 |
| PT\_LOAD1头 | integer | 8 |
| PT\_LOAD2头 | integer | 8 |
| .dynstr节头 | integer | 11 |
| .dynsym节头 | integer | 6 |
| .shstrtab节头 | integer | 11 |
| .init\_array节头 | integer | 11 |

对表3-2中提取的初选特征简单描述如下：

1）ELF头，选取了除e\_ident的前6个字节、e\_type、e\_version以外的所有属性，这是因为上述几个成员对于每个ELF文件来说都是相同的。

2）PT\_LOAD1头，第1个可加载段的程序头，选取了该头部的所有8个属性。

3）PT\_LOAD2头，第2个可加载段的程序头，选取了该头部的所有8个属性。

4）.dynstr节头，该节的内容包含了动态链接时所需的全部字符串，选取了该部分的所有内容。

5）.dynsym节头，该节的内容包含了动态链接时所需的符号表信息，选取了该部分的所有内容。

6）.shstrtab节头，该节的内容包含了所有节的名称字符串，选取了该部分的所有内容。

7）.init\_array节头，该节用于存放初始化函数地址，选取了该部分的所有内容。

### 3.4.2 特征变换

初选特征中包含了大量的属性，其中大部分属性对加壳SO文件检测是无用的，因此需要进行特征变换来剔除这些无用属性。特征变换的具体作用是降低初选特征的维度，以减小计算复杂度，从而提高后续的分类准确度。

本文引入线性判别分析（Linear Discriminant Analysis，LDA）算法[[[28]](#endnote-28)]以达到消除冗余特征的目的，该算法的基本思想是，将高维的样本数据投射到最优鉴别向量空间，以实现分类信息的抽取和特征空间的降维，投影后保证样本数据在新的子空间的类间距离最大和类内距离最小，即样本数据在该空间中有最好的可分离性[[[29]](#endnote-29)]。

假设训练样本集是，其中为维向量，代表每个训练样本包含了个特征，为训练样本集的个数。类样本是，相对应的是类的样本是，其中类代表未加壳样本类，类代表加壳样本类。按照LDA的思想，需要寻找到一个最佳的投影方向，使得样本按照公式（3-1）中的线性变换投影到低维度中的。

（3-1）

首先本文寻找每类样本的均值，按照（3-2）和（3-3）中的公式分别计算出类的均值和类的均值。

（3-2）

（3-3）

然后按照（3-4）和（3-5）中的公式分别计算出类的散列矩阵和类的散列矩阵。

（3-4）

（3-5）

本文定义，如果是可逆的，那么按照公式（3-6）即可求出最佳方向。

（3-6）

因此，本文只需要求出训练样本的均值和方差就可以求出最佳的方向，求出后，挑出最大的个特征值对应的特征向量作为变换矩阵，即完成了特征变换。

本文使用向量空间模型来格式化数据，每个SO文件按照文件格式被定义为一个64维向量。提取所有的特征向量建立了一个矩阵，使用LDA算法，得到10个主要特征，最后深入分析这10个特征，根据实际意义格式化特征向量，得到如下SO文件特征集：

SO\_Info = {D, S, N, E, F}

具体含义如下：

D代表头部部分成员的值是否为默认值，检测SO文件头部的e\_ident、e\_entry、e\_flags三个成员的值是否为默认值，其中e\_ident成员的后10个字节的默认值全为0，e\_entry的默认值为0，e\_flags的默认值为0x5000000。如果是则D的值为1；否则为0。

S代表头部是否包含有效的节信息，检测SO文件头部的e\_shoff、e\_shentsize、e\_shnum、e\_shstrndx四个成员的值是否有效，其中e\_shoff成员不应大于文件大小，e\_shentsize成员为40，e\_shnum成员大于0且不超过50，e\_shstrndx成员为e\_shnum减1。如果是则S的值为1；否则为0。

N代表是否存在奇怪的节名称，检测所有节数据结构的名称字段是否存在非标准名字、名字不能打印或为空的情况，如果存在则N的值为1；否则为0。

E代表具有可执行属性的段的个数是否超过1个，检测所有段中具有可执行属性的个数，如果个数超过1则E的值为1；否则为0。

F代表具有几个初始化函数，包括三个类型：init函数；init\_array数组中的函数个数；JNI\_OnLoad。三种类型均无，则F的值为0；每有1个初始化函数，F的值加1。

## 3.5 朴素贝叶斯分类方法

### 3.5.1 贝叶斯定理

贝叶斯定理（Bayes theorem）[[[30]](#endnote-30)]是一种概率论定理，它提供了一种在已知某事件的条件概率的情况下，计算出两个事件交换后的概率的方法。

给出定理之前要引入一些记号，假定有两个相关事件和，表示在事件已经发生的情况下，事件发生的概率。表示在事件已经发生的情况下，事件发生的概率。表示事件发生的概率。表示事件发生的概率。其中，、和是容易得到的，而可以通过贝叶斯定理计算得到，贝叶斯定理如下所示：

（3-7）

### 3.5.2 朴素贝叶斯分类

朴素贝叶斯（Naive Bayes）[[[31]](#endnote-31)]分类是一种基于贝叶斯定理的分类算法，它广泛应用于机器学习中的分类学习。它的基本思想为：首先假定待分类样本的各个特征属性之间是相互独立的，其次计算出各个类别下各个特征属性的先验概率，接着利用贝叶斯定理求解待分类样本属于各个类型的概率，最后哪个概率最大，就认为此待分类样本属于哪个类别。

对于加壳SO文件的分类而言，假设为一个SO文件样本，而每个为的一个特征属性，共有n个特征属性，表示所有SO文件样本，其中表示未加壳SO文件样本，表示加壳SO文件样本，对于一个待分类SO文件样本，计算：

（3-8）

其中，表示训练样本集中类别为的概率，表示特征为且类别为的概率，表示特征为的概率，表示样本的类别为的概率。对与给定的训练样本集来说，公式（3-8）中的分母的值是固定的，因此只需要计算分子的值，并比较值的大小，若大于则待分类SO文件样本为未加壳SO文件，否则为加壳SO文件。

朴素贝叶斯分类器的算法逻辑简单，易于实现，对于不同的数据有着同样的分类效果。

### 3.5.3 训练与分类的过程

朴素贝叶斯分类器的工作流程主要分为训练阶段和分类阶段，两个阶段主要内容如下：

1）训练阶段

训练阶段，也就是构造分类器的过程。首先收集SO文件的训练样本集，通过对训练样本的解析，得到一个基于SO文件格式的初选特征。其次利用LDA线性变换对初选特征进行降维，对降维后的特征进行格式化处理，以得到一个适用于朴素贝叶斯分类器的特征集，表3-3给出了部分训练数据实例。接着将特征集作为朴素贝叶斯分类器的样本输入，训练成功后，就可以得到贝叶斯公式中先验概率和条件概率，即计算训练样本中未加壳样本和加壳样本的出现频率，也就是公式（3-8）中的和，计算各个特征属性分别对未加壳样本和加壳样本的条件概率估计，也就是式1中的，并记录结果。最后就得到一个可用于SO文件壳检测的朴素贝叶斯分类器。

表3-3 部分训练数据

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **编号** | **D** | **S** | **N** | **E** | **F** | **C** |
| 0 | 1 | 1 | 1 | 1 | 1 | packed |
| 1 | 1 | 1 | 0 | 0 | 1 | packed |
| 2 | 1 | 1 | 1 | 0 | 1 | packed |
| 3 | 1 | 1 | 0 | 1 | 1 | packed |
| 4 | 0 | 0 | 0 | 0 | 1 | packed |
| 5 | 1 | 1 | 1 | 1 | 2 | packed |
| 6 | 1 | 0 | 1 | 1 | 1 | packed |

训练阶段是整个朴素贝叶斯分类中最为重要的一个阶段，特征集的好坏和训练样本的质量决定了训练效果，而训练质量决定了整个分类器最终的分类效果

2）分类阶段

分类阶段，这个阶段的任务是使用分类器对待分类样本进行分类，其输入是完成训练阶段的朴素贝叶斯分类器和选定的SO文件的测试样本，输出是测试样本是否加壳。分别计算测试样本属于未加壳文件和加壳文件的后验概率，也就是公式（3-8）的和，比较这两个后验概率的大小即可判定测试样本是否加壳。

分类阶段是机械系阶段，无需人工干预，由程序完成。

两个阶段的总体流程如图3.8所示。

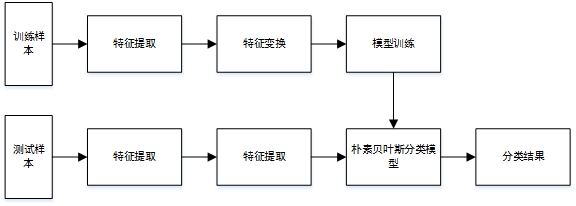


图3.8 朴素贝叶斯分类器的工作流程

## 3.6 本章小结

本章首先研究了现有的加壳技术，发现未加壳文件与加壳文件在SO文件格式上存在较大差异。在这个研究基础上，将机器学习理论应用到壳检测领域中，采用了一种基于朴素贝叶斯分类器的壳检测方法。该方法对人工抽取出的特征进行变换得到一个新的特征集，接着利用该特征集训练分类器，最终使用训练完成的分类器即可判定SO文件是否加壳。本文在第5章对该壳检测系统进行了实验测试。

# 第4章 基于soinfo内存转存的脱壳技术

在对恶意软件进行查杀时，仅仅识别出软件是否加壳是不够的，为了更好的检测软件的恶意行为，本文还需要对加壳软件进行脱壳处理。本章借鉴PE文件脱壳的已有经验，依据Android平台的特性，采用了一种基于soinfo内存转存的脱壳方法，该方法对已知壳和未知壳的脱壳都有较高的成功率。

## 4.1 加载机制

### 4.1.1 SO文件的加载机制

介绍Android下SO文件的加载机制有助于研究壳的加载机制以及对后续脱壳技术的研究提供理论支撑。本文通过分析Android4.4的相关源码，了解了Android是如何将SO文件加载到内存，如何进行链接，如何完成初始化工作。整个加载过程分为4个部分，分别为准备，装载，链接和初始化。

1）准备

在进行真正的加载工作之前，Android会首先做一些准备工作，例如参数合法性校验，权限检测等。本文以Android加载SO文件的Java层接口函数System.loadLibrary(String libName)为入口，逐步往下分析，得到准备阶段的函数调用关系如图4.1所示。



图4.1 准备阶段的函数调用关系

准备阶段的主要工作如下，首先扫描APK内部的nativeLibrary目录（apk的lib目录，/vendor/lib，/system/lib），获得并返回SO文件的绝对路径，如果获取失败则抛出异常；接着检查是否已经加载了该SO文件，如果加载过则不需要重复加载；最后调用dlopen函数进入装载阶段。

2）装载

当Android完成准备工作后，就进入了装载阶段，装载阶段主要完成对SO文件中需要加载的段的内存映射工作。本文以装载阶段的起始函数dlopen()为入口，逐步往下分析，得到装载阶段的函数调用关系如图4.2所示。



图4.2 装载阶段的函数调用关系

装载阶段的主要工作如下，首先遍历soinfo链表，检查是否已经加载了该SO文件，如果加载过则不需要重复加载，否则继续；其次打开SO文件，读取ELF头，并验证该SO文件是否为有效的SO文件，如果验证失败则返回，否则继续。接着读取程序头，并调用mmap()将其映射到指定的页中，根据程序头的信息，计算出加载该SO文件的起始地址和所需总的内存空间大小，并通过mmap()创建足够大的匿名内存空间，以便能够容纳所有可加载的段。然后将所有PT\_LOAD属性的段加载至合适的地址空间，其中代码段与数据段的相对位置与文件的运行域一致；然后校验程序头是否在内存中；最后为该SO文件在共享库链表中分配一个soinfo节点，并根据前面获得的信息，初始化其数据结构。

3）链接

SO文件属于共享库文件，需要参与动态链接。Android在完成SO文件的装载过后，就会进入链接阶段。本文以链接阶段的起始函数为soinfo\_link\_image ()为入口，逐步往下分析，得到链接阶段的函数调用关系如图4.3所示。



图4.3 链接阶段的函数调用关系

链接阶段的主要工作如下，首先从程序头中获取.dynamic节信息，并对soinfo结构体的dynamic进行赋值；其次遍历dynamic数组，根据每个元素的标记信息进行相应的处理；接着依次获取dynamic数组中定义的每一个外部SO文件的soinfo节点；最后遍历重定位表，根据每个元素的重定位类型，对相对应的符号进行重定位，重定位是整个链接阶段的关键部分，并且与后续的SO脱壳密切相关。

4）初始化

Android在完成SO文件的装载和链接后，就会执行初始化构造函数，即进入初始化阶段。本文以初始化阶段的起始函数为CallConstructors()为入口，逐步往下分析，得到初始化阶段的函数调用关系如图4.4所示。



图4.4 初始化阶段的函数调用关系

初始化阶段的主要工作如下，首先依次执行外部SO文件的初始化构造函数；其次先执行.init节的代码再执行.init\_array节的代码；最后获取JNI\_OnLoad函数的地址，如果存在，则调用JNI\_OnLoad。可以发现这些初始化构造函数都是在SO文件的真正逻辑开始之前执行的，这也解释了为什么壳程序通常都把壳代码放在这里。

### 4.1.2 壳的加载机制

几乎所有的壳都会首先获得程序控制权，其大多采用修改程序入口地址为壳入口地址的技术手段[[[32]](#endnote-32)]。对于SO文件来说，并没有传统意义上的类似于main函数的程序入口，而是存在一个初始化函数队列，系统加载SO文件时，在完成装载和动态链接的工作后，便会执行该初始化函数队列，即调用位于.init节和.init\_array节中的代码以及JNI\_OnLoad函数。壳通过把壳代码放入初始化函数队列中，使得在SO文件加载时优先获得程序控制权。壳的加载过程如图4.5所示。

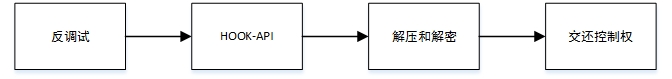


图4.5 壳的加载过程

反调试：大多数壳程序为了对抗脱壳会采取一系列的技术手段，例如：检测模拟器，阻止gdb、IDA等调试工具的附加调试，监控自身内存防止被第三方程序修改等。

HOOK-API：壳会修改原程序的一些重要API函数的地址，例如：mmap，open、mprotect等。这样壳在交还控制权后，仍然能够间接获得程序控制权。

解压和解密：壳程序不能影响原程序的正常执行，即要保证原程序各个函数的应有功能，所以壳程序一般都会对原文件的各个区块进行相应的解压和解密工作。

交还控制权：此时原程序可以正常工作了。

## 4.2 脱壳技术的框架设计

### 4.2.1 设计思想

1）通用性的问题

通用性指的是无需考虑壳的种类，对已知壳和未知壳都有同样的脱壳效率。单一性指的是先识别出特定类型的壳，然后使用对应的脱壳算法进行脱壳[[[33]](#endnote-33)]。现有的各种加壳方法具有着不同的加密算法和原理，如果要针对每种壳都提出对应的脱壳方法，需要经验丰富的分析人员耗费大量时间进行逆向工作，并且各个壳的收集工作也极为困难。为了避免上述问题，本文研究的是一种无需判定壳类型，对已知壳、变形壳、多重壳和未知壳均有较高脱壳效率的通用性脱壳方法。

2）安全性的问题

加壳文件也许是具有恶意行为的程序，而对于脱壳，往往需要运行或加载加壳文件，因此脱壳过程可能会对宿主环境造成危害。在脱壳系统中，一方面，可以通过设计一个可控的执行环境来保证宿主的安全，传统的可控执行环境有三种，分别为基于虚拟机的受控环境[[[34]](#endnote-34)]，基于调试器的受控环境和基于系统调用监控的受控环境。上述三种受控环境均可避免恶意行为对系统造成破坏，但是其实现复杂性较高，对效率也有所影响。另一方面，可以使用专用的测试机，一旦受到任何恶意行为攻击都可以回滚到初始状态。考虑到降低脱壳系统复杂性这一点，本文采用测试机来解决安全性的问题。

3）让壳程序自己脱壳

加壳文件虽然在程序流程上与未加壳文件有所差异，但是它们都具有同样的实际功能，也就是说壳程序在执行过程中会释放原代码和数据到内存中来[[[35]](#endnote-35)]。通过前文对SO文件的加载机制和壳的加载机制的研究，知道现有的壳程序都有一个共同的特征，即为原文件在本地完成加密，加壳文件运行时，壳程序首先获得控制权并在内存中完成解密，壳程序执行完毕后退出，控制权交还给原文件[[[36]](#endnote-36)]。也就是说当壳程序执行完毕后，此时在内存中就可以得到脱壳后的文件了。

终上所述，加壳文件在运行时，壳程序会自动在内存中完成脱壳，此时只需找到壳程序执行完毕的点，即可完成脱壳。

4）脱壳后的修复工作

通常情况下，脱壳后的程序还需要进行修复处理，原因是多方面的。其一，大多数壳程序会篡改SO文件头的节相关字段，这样并不会影响程序的正常运行，却会导致IDA无法解析；其二，脱壳后的数据是与内存加载后是一致的，而内存中的数据已经对原SO文件进行了部分修改，这导致脱壳后的程序是无法被动态调试的，更无法直接运行。因此，还需要对脱壳后的SO文件进行修复，以达到可以静态分析和动态调试的目的。

### 4.2.2 总体框架

基于soinfo内存转存的脱壳技术主要包括3大模块，分别为OEP模块，内存dump模块，重构模块。总体框架如图4.6所示。

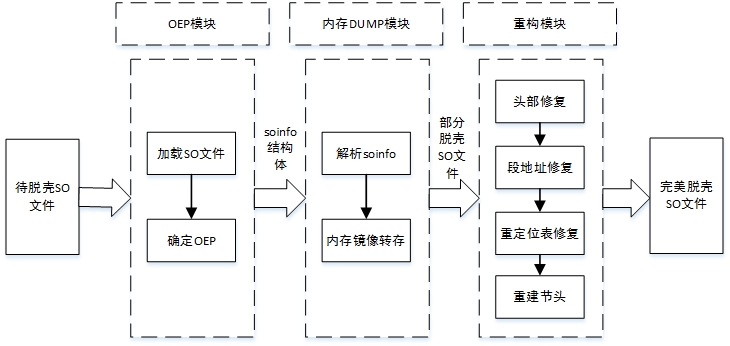


图4.6 脱壳总体框架

其中OEP模块的目的是寻找原程序的入口点，也就是确定脱壳的时机，一个好的脱壳时机应该是壳程序已经完成所有解压和解密工作；内存dump模块的目的是根据soinfo结构体得到待脱壳文件在内存中的起始地址和长度大小，最终将解密后的镜像文件从内存中拷贝到本地文件中来；重构模块的目的是对脱壳后的镜像完成修复工作，主要包括头部修复技术，段表地址修复，重定位表修复，重建节和清除壳入口。通过上述修复工作，使得脱壳后的文件可以被静态分析和动态调试，该模块是脱壳技术的核心模块。

## 4.3 寻找OEP

OEP（Original Entry Point），即程序的真正入口点[[[37]](#endnote-37)]。根据前文的研究，可以知道壳程序在完成功能后会将控制权交给原程序，也就是会执行到OEP。壳代码通常位于SO文件的初始化构造函数队列中，也就是位于.init节、.init\_array节中的代码以及JNI\_OnLoad函数。

由于本文研究的是一种通用性脱壳方法，无法确定壳代码的具体位置，因此本文选择的OEP应该位于所有初始化函数执行完毕后。Android下SO文件的初始化构造函数执行顺序如图4.7所示。



图4.7 初始化构造函数执行顺序

从图4.7可知，JNI\_OnLoad函数是最后被执行的，也就是说当JNI\_OnLoad函数执行完毕时，就是来到了OEP。

## 4.4 soinfo结构体

每一个SO文件在装载成功后，其在内存中都会有一个与之对应的结构体，该结构体存放了SO文件的相关信息，以用于SO文件的后续功能。该结构体由soinfo类型的结构数组组成，该类型的部分成员的类型，名称，含义，如表4-1所示。

表4-1 soinfo结构体的部分关键成员

|  |  |  |
| --- | --- | --- |
| **类型** | **名称** | **含义** |
| char | name | SO文件的名称字符串。 |
| const Elf32\_Phdr \* | phdr | 指向程序头的虚拟地址。 |
| size\_t phnum | phnum | 程序头的表项数目。 |
| Elf32\_Addr | entry | SO文件的入口地址。 |
| Elf32\_Addr | base | SO文件第一个字节在内存中的虚拟地址。 |
| unsigned | size | SO文件在内存中的大小（以字节为单文）。 |
| Elf32\_Dyn \* | dynamic | 指向.dynamic节。 |
| unsigned | flags | 表示与文件相关的，特定与处理器的标志。 |
| const char \* | strtab | 指向.strtab节。 |
| Elf32\_Sym \* | symtab | 指向.symtab节。 |
| size\_t | nbucket | bucket数组的项数。 |
| size\_t | nchain | chain数组的项数。 |
| unsigned \* | bucket | 包含了符号表的索引。 |
| unsigned \* | chain | 包含了符号表的索引。 |
| unsigned \* | plt\_got | 指向\_\_global\_offset\_table\_\_。 |
| Elf32\_Rel \* | plt\_rel | 指向.rel.plt节。 |
| size\_t | plt\_rel\_count | .rel.plt节的表项数目。 |
| Elf32\_Rel \* | rel | 指向. rel.dyn节。 |
| size\_t | rel\_count | . rel.dyn节的表项数目。 |
| linker\_function\_t \* | init\_array | 指向.init\_array节。 |
| size\_t | init\_array\_count | .init\_array节的表项数目。 |
| linker\_function\_t \* | fnit\_array | 指向.fnit\_array节。 |
| size\_t | fnit\_array\_count | .fnit\_array节的表项数目。 |
| unsigned \* | ARM\_exidx | 指向.ARM.exidx节。 |
| size\_t | ARM\_exidx\_count | .ARM.exidx节的表项数目。 |

可以看到soinfo结构体中包含了大量的关于SO文件的信息，借助于解析soinfo结构体，可以大量减少文件解析的工作量，并且绕过加壳软件的各种加密算法，已到达通用脱壳的目的。

## 4.5 内存dump

dump，用中文翻译过来就是转存，也就是抓取内存镜像，就是把内存指定地址的映像文件读取出来，然后用文件的形式保存下来[[[38]](#endnote-38)]。

在Window平台下dump的实现主要依靠诸如OD、LordPE和PETools等工具来实现，这些工具基本都是利用Module32Next函数来获取将要被dump进程的信息[[[39]](#endnote-39)]。对于Android平台，目前并没有有效的dump工具，本文通过解析SO文件对应的soinfo结构体完成了程序化内存dump。其具体步骤如下：

1）调用dlopen函数加载待脱壳文件，从而得到对应的soinfo结构体，若dlopen函数返回成功（即返回值为1），则进入第2）步，否则失败退出。

2）调用dlsym函数获取JNI\_OnLoad函数地址，若SO文件中存在该函数，则执行该函数。

3）解析soinfo结构体信息，得到soinfo结构体中的成员base和size，也就是内存镜像的起始地址和总大小，将内存镜像保存到本地文件中。

流程图如图4.8所示。

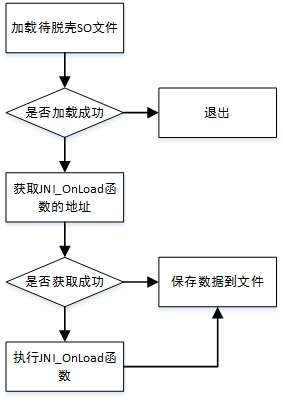


图4.8 dump模块工作流程

## 4.6 重构SO文件

在完成内存dump后，还需要对转存之后的内存镜像进行重构工作，包含5部分内容，分别为头部修复、段地址修复、重定位表修复、重建节头和清除壳入口。

### 4.6.1 头部修复

头部修复技术比较简单，本文将需要修复的头部成员分为3类，分别为固定成员、段相关成员以及节相关成员。

固定成员指的是对于每个SO文件来说都是固定不变的成员，这些成员描述了系统信息或者校验信息。其包含5个成员，分别为：e\_ident、e\_type、e\_version、e\_ehsize、e\_flags。其中e\_ident共有16个字节，前6个字节的值分别为：’7f’，’45’，’4c’，’46’，‘1’，‘1’，后续10个字节全部为0。e\_type的值为ET\_DYN。e\_version的值为EV\_CURENT。e\_ehsize的值为ELF头所占用的字节数，也就是Elf32\_Ehdr结构体的大小。e\_flags不用修复。

段相关成员指的是用于装载阶段的成员，这些成员描述了段的相关信息。其包含3个成员，分别为：e\_phoff、e\_phentsize、e\_phnum。其中e\_phoff和e\_phnum不用修复，e\_phenntsize的值为程序头的表项所占用的字节数，也就是Elf32\_phdr结构体的大小。

节相关成员指的是用于链接阶段的成员，这些成员描述了节的相关信息。其包含4个成员，分别为：e\_shoff、e\_shentsize、e\_shnum、e\_shstrndx。e\_shentsize的值为节头的表项所占用的字节数，也就是Elf32\_Shdr结构体的大小，e\_shoff、e\_ shnum和e\_shstrndx将在后续的重建节头中修复。

### 4.6.2 段地址修复

通过前文对SO文件格式的分析，可以知道在程序头中存在两个成员，分别为p\_offset和p­\_vaddr，前者指的是从文件头到该段第一个字节的偏移，后者指的是段的第一个字节将被映射到内存中的虚拟地址。

Android加载SO文件时，是将段加载到p\_vaddr虚拟地址的。由于内存对齐的原因，p\_vaddr和p\_offset的值有时是不同的。例如图4.9中，PT\_LOAD2段和DYNAMIC段的p\_vaddr比p\_offset大了0x1000，这是因为这两个段的内存对齐大小为0x1000。本文从内存中dump的数据是跟内存加载后一致的，而IDA解析SO文件是通过p\_offset确定段在文件中的偏移地址的，因此这里需要将PT\_LOAD段和PT\_DYNAMIC段的p\_offset改为和p\_vaddr一样的值，以使得IDA可以正常加载SO文件和解析导入函数。具体修复步骤如下：

1）解析ELF头部，通过e\_phoff成员得到程序头的地址，通过e\_ phnum成员得到程序头包含的表项个数。

2）遍历所有程序头，将p\_offset成员的值修改为p\_vaddr成员的值。

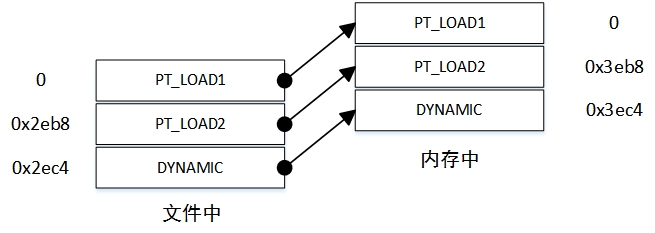


图4.9 内存映射示意图

### 4.6.3 重定位表修复

重定位发生在SO文件的链接阶段，因此通过dump得到的镜像是已经完成了重定位工作的，而重定位得到的value只是在当时的进程环境下是有效的，是一个实际物理地址，因此需要将被重定位的value重新修复为偏移地址，这样dump后的SO文件才能被正常加载起来。

为了方便理解，本文编写了一个测试SO文件，将测试SO文件加载后，利用dump模块得到一个dump文件。

下面是dump后SO文件中动态段里.rel节部分内容，如图4.10所示。



图4.10 动态段里.rel节的内容

从图中可知.rel节对应的数据位于距文件头部0x0c7c字节的位置，通过前文对重定位节的分析，可知该处的内存是由一个Elf32\_Rel结构体数组构成，如图4.11所示。



图4.11 .rel节的内容

其中Elf32\_Rel结构体中的r\_offset成员的值为0x2e94，而r\_offset成员指向的值会被重定位，下面是测试SO文件该处的值，如图4.12所示。



图4.12 测试SO文件的r\_offset指向的值

下面是DUMP文件该处的值，如图4.13所示。



图4.13 dump文件的r\_offset指向的值

其中该SO文件在内存中的基地址0x77645000，对比图4.12和图4.13，不难发现dump文件的r\_offset成员指向的值为测试SO文件的r\_offset成员指向的值加上SO的基地址。也就是说，重定位修复的时候只需要减去基地址就可以恢复为原始值。

重定位修复函数的定义如下：

reverseRelocate(Elf32\_Rel \*rel, size\_t count);

rel为待修复的重定位表首地址，count为表项个数。其中rel和count的值都可以从soinfo结构体中得到。.rel.dyn表修复：参数rel的值为soinfo结构体中的rel成员，参数count的值为soinfo结构体中的rel\_count成员；rel.plt表修复：参数rel的值为soinfo结构体中的plt\_rel成员，参数count的值为soinfo结构体中的plt\_rel\_count成员。

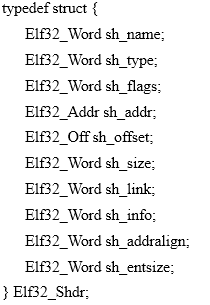
reverseRelocate函数的具体流程如下：

1. 遍历重定位表，根据重定项的r\_info成员获得重定位类型和重定位项对应的符号在符号表中的索引；
2. 利用SO文件中的 .hash 表，根据符号名快速地查找符号在哪个SO文件中定义；
3. 当找到了符号的定义，计算符号的地址sym\_addr；最后根据符号的重定位类型，结合sym\_addr逆向重定位。

### 4.6.4 重建节头

从SO文件加载机制可知，系统加载SO文件时并不需要节头，也就是说dump后的文件并不包含节头，然而静态分析常用的readelf、objdump、ida等工具对SO文件的分析都依赖与节头内容，因此重建SO文件的节头是十分必要的。

重建节头就是指重新为每一个节创建对应的节头，并填充数据。为了方便阐述，这里再次给出节头的结构体定义：



对于一个指定的特殊节，除了sh\_name、sh\_addr、sh\_offset、sh\_size以外，其他成员的值的都是已知的。表4-2给出了各个节中其他成员的值。

表4-2 各个节中的已知成员的值

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **节名** | **sh\_type** | **sh\_flags** | **sh\_link** | **sh\_info** | **sh\_addralign** | **sh\_entsize** |
| .dynsym | 11 | 2 | 2 | 1 | 4 | 16 |
| .dynstr | 3 | 2 | 0 | 0 | 1 | 0 |
| .hash | 5 | 2 | 1 | 0 | 4 | 4 |
| .rel.dyn | 9 | 2 | 1 | 0 | 4 | 8 |
| .rel.plt | 9 | 2 | 1 | 6 | 4 | 8 |
| .plt | 1 | 6 | 0 | 0 | 4 | 0 |
| .text | 1 | 6 | 0 | 0 | 32 | 0 |
| .ARM.extab | 1 | 2 | 0 | 0 | 4 | 0 |
| .ARM.exidx | 1879048193 | 130 | 7 | 0 | 4 | 8 |
| .rodata | 1 | 2 | 0 | 0 | 8 | 0 |
| .fini\_array | 15 | 3 | 0 | 0 | 4 | 0 |
| .init\_array | 14 | 3 | 0 | 0 | 4 | 0 |
| .dynamic | 6 | 3 | 2 | 0 | 4 | 8 |
| .got | 1 | 3 | 0 | 0 | 4 | 0 |
| .data | 1 | 3 | 0 | 0 | 8 | 0 |
| .bss | 8 | 3 | 0 | 0 | 8 | 0 |
| .shstrtab | 3 | 0 | 0 | 0 | 1 | 0 |

通过表4-2，本文就完成了节头中所有已知成员的重建，接下来需要重建sh\_name、sh\_addr、sh\_offset、sh\_size。首先重建sh\_addr、sh\_offset、sh\_size。根据表4-2所示，soinfo结构体中包含了大部分节的信息，因此通过解析soinfo结构信息就可以完成绝大部分节的重建工作，重建步骤如下（注：步骤中的所有成员均属于SO文件对应的soinfo结构体）：

1）通过base成员得到SO文件在内存中的基地址。

2）通过symtab、rel、plt\_rel、ARM\_exidx、fini\_array、init\_array、dynamic这7个成员分别得到.dynsym、.rel.dyn、.rel.plt、.ARM\_exidx、.fini\_array、. init\_array、. dynamic这7个节在内存中的虚拟地址，记这个虚拟地址为VA。则节的sh\_addr成员为VA，而由于内存对齐的原因，节的sh\_offset成员可能为VA，也可能为VA - 0x1000。

3）通过nchain、rel\_count、plt\_rel\_count、ARM\_exidx\_count、fini\_array\_count、init\_array\_count、dynamic\_count这7个成员分别得到.dynsym、.rel.dyn、.rel.plt、.ARM\_exidx、.fini\_array、. init\_array、. dynamic这7个节的表项数目，记这个表项数目为COUNT，表项大小为SIZE。则节的sh\_size成员的值为COUNT \* SIZE。

上述7个节的重建结果如表4-3所示。

表4-3 部分节的重建结果

|  |  |  |  |
| --- | --- | --- | --- |
| **节名** | **sh\_addr** | **sh\_offset** | **sh\_size** |
| .dynsym | symtab - base | symtab - base | nchain \* 16 |
| .rel.dyn | rel - base | rel - base | rel\_count \* sizof(ELF32\_Rel) |
| .rel.plt | plt\_rel - base | plt\_rel - base | plt\_rel\_count \* sizeof(Elf32\_Rel) |
| . ARM\_exidx | ARM\_exidx - base | ARM\_exidx - base | ARM\_exidx\_count \* 8 |
| .fini\_array | fini\_array - base | fini\_array - base - 0x1000 | fini\_array\_count \* 4 |
| .init\_array | init\_array - base | init\_array - base -0x1000 | init\_array\_count \* 4 |
| .dynamic | dynamic - base | dynamic - base - 0x1000 | dynamic\_coun \* sizeof(Elf32\_Dyn) |

接下来，如果.dynamic节修复成功，则通过解析.dynamic节完成对.dynstr节和.hash节的修复工作。具体修复步骤如下：

1）通过strtab成员得到.dynstr节在内存中的虚拟地址，用虚拟地址减去基地址的值即为.dynstr节的sh\_addr和sh\_offset，通过解析.dynamic节找到DT\_STRSZ，从而得到.dynstr节的sh\_size。

2）通过解析.dynamic节找到DT\_HASH，从而得到.hash节在内存中的虚拟地址，用虚拟地址减去基地址的值即为.hash节的sh\_addr和sh\_offset，nbucket成员的值加上nchain成员的值再加上2的总和再乘以4即为.hash节的sh\_size。

目前为止，还剩下.plt、.got、.text、.ARM.extab、.rodata、.data、.bss、.shstrtab共8个节的节头尚未修复。

其中，由于优化的原因，.plt节的开头总是固定代码，如图4.14所示。

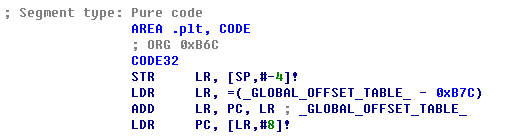


图4.14 .plt节的开头代码块

因此借鉴Window内核中搜索未导出符号的思路，可以通过搜索前16个字节来确定.plt节的sh\_addr和sh\_offset，这16个字节为：0xe52de004, 0xe59fe004, 0xe08fe00e, 0xe5bef008。.plt节由固定16字节加上4字节的\_\_global\_\_offset\_table变量和需要重定位的函数地址构成，函数地址又与.rel.plt节中的结构一一对应，因此，.rel.plt节的表项数目乘以12再加上20即为.plt节的sh\_size。

.got节存在两种结构，即为{.got,.got\_plt}和{. got\_plt,.got}。其中，.got节中的项一定出现在.rel节中，.got\_plt中的项于.rel\_plt中的项是一一对应关系。因此，.got节的重建步骤如下：

1）解析.dynamic节，读取DT\_PLTGOT，从而获取\_\_global\_\_offset\_table变量，记为plt\_got。

2）读取plt\_got - 4地址的值，在.rel节中进行搜索，如果匹配成功，则说明.got节的结构为{.got,.got\_plt}，转到第3）步；否则为{. got\_plt,.got}，转到第4）步。

3）继续向前搜索匹配，直到匹配失败，此处即为.got节的sh\_addr和sh\_offset。调整搜索位置到got\_plt，向后搜索，在.rel\_plt节中进行匹配，直到匹配失败，此处即为.got节的末尾，从而可计算得到.got节的sh\_size。

4）读取plt\_got - 4地址的值，在.rel\_plt节中进行搜索，直到匹配失败，此处即为.got节的sh\_addr和sh\_offset。调整搜索位置到got\_plt，向后搜索，在.rel节中进行匹配，直到匹配失败，此处即为.got节的末尾，从而可计算得到.got节的sh\_size。

.text、.ARM.extab、.rodata、.data这4个节，通过各个节之间的相对位置确定其sh\_addr、sh\_offset和sh\_size。

现在还剩下最后一个. shstrtab节尚未重建，从前文可知，. shstrtab节内容存放了所有节的名称字符串。也就是说所有节的sh\_name成员均是指向.shstrtab节中字符串的索引。因此，本文可以自行构造出. shstrtab节的内容，部分内容如表4-4所示。

表4-4 .shstrtab节部分内容示意图

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \0 | \46 | s | h | s | t | r | t | a | b |
| \0 | \46 | d | y | n | s | y | m | \0 | \46 |
| d | y | n | s | t | r | \0 | \46 | h | a |
| s | h | \0 | \46 | r | e | l | \46 | r | e |

最后，为了不影响原有SO文件的结构，将节头添加至SO文件的尾部，并修正ELF头部中的节相关成员，也就是e\_shoff、e\_ shnum和e\_shstrndx。至此本文完成了大部分的节头重建工作，重建后的SO文件已经可以静态分析和动态调试了。

SO文件的重构整体流程如图4.15所示。

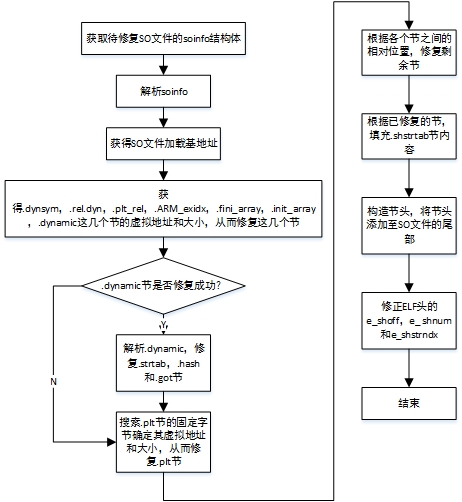


图4.15 SO文件的重构流程

### 4.6.5 清除壳入口

在完成上述脱壳工作后，SO文件已经完成了解密工作，但是此时的程序入口仍然位于.init和.init\_array节中的壳程序，也就是说此时如果加载该SO文件，壳程序依然会首先获得程序控制权，因此本文还需要清除壳入口。

Android调用.init节和.init\_array节代码的源代码如下：

CallFunction("DT\_INIT", init\_func);

CallArray("DT\_INIT\_ARRAY", init\_array, init\_array\_count, false);

其中，CallFunction函数的作用是调用.init节的初始化代码，参数init\_func即为调用地址，若init\_func的值为0或-1则不执行初始化代码；CallArray函数的作用是依次调用.init\_array节中的所有初始化代码，参数init\_array为初始化代码数组，init\_array\_count为初始化代码的个数，若参数init\_array\_count的值为0则不执行初始化代码。因此，本文需要通过修改SO文件的相关信息，使得init\_func和init\_array\_count的值为0。

具体实现步骤如下：

1）通过SO文件的soinfo结构体信息，得到.dynamic节信息。

2）遍历.dynamic节，依次找到d\_tag成员为DT\_INIT（.init节）和DT\_INIT\_ARRAY（.init\_array节）的表项，表项由一个Elf32\_Dyn类型的结构体数组构成。

3）将.init节表项中的d\_ptr成员修改为0。

4）将.init\_array节表项中的d\_val成员修改为0。

## 4.7 本章小结

本章主要采用了一种基于soinfo的SO文件脱壳方法，soinfo结构体中包含了大量的SO文件信息，通过解析SO文件对应的soinfo结构体，可以十分方便的对自脱壳后的内存镜像完成dump，同时，还可以根据soinfo结构体中的节相关信息对dump后的SO文件进行重建工作，最终得到一个可用于静态分析和动态调试的脱壳后SO文件。本文在第5章对该脱壳系统进行了实验测试。

# 第5章 实验结果与分析

## 5.1 壳检测实验

### 5.1.1 WEKA平台概述

WEKA（Waikato Environment for Knowledge Analysis），中文译为怀卡托智能分析环境，是新西兰怀卡托大学开发的一款免费的，非商业化基于JAVA环境下开源的机器学习与数据挖掘软件[[[40]](#endnote-40)]。

作为一个完备的工作平台，WEKA不止实现了大量的机器学习算法（包括分类、回归、聚类、关联规则），还集成了数据预处理，以及在交互式界面上的可视化功能。同时，WEKA也提供了基于Java的开发接口，以便使用者能将WEKA集成到自己的程序中，或是用自己开发和改进的算法扩展WEKA。

总而言之，WEKA具有强大的机器学习和数据挖掘功能，为开发者提供了一个非常有利的开发平台。本文使用了WEKA中内置的朴素贝叶斯分类器对待检测样本进行训练和分类。

### 5.1.2 实验环境与数据

壳检测实验是在一台基于x86系统的PC机上进行的，其处理器型号为E3-1231，主频为3.4GHZ，使用Intel集成显卡，8GB的物理内存以及1TB的硬盘空间，实验操作系统为Ubuntu 14.04。

实验数据为320个未加壳样本和130个加壳样本，其中未加壳样本来自于Android各个版本的系统SO文件以及从各个Android应用市场下载的APK文件中提取的SO文件，加壳样本来自于网络中收集的一些恶意软件中的加壳SO文件以及各个加壳工具对SO文件进行加壳后生成的加壳SO文件。

所有样本又分为训练样本集和测试样本集。训练样本集共有280个样本，其中包含200个未加壳样本，80个加壳样本，训练样本集用于分类器的训练。测试样本集共有170个样本，其中包含120个未加壳样本，50个加壳样本，测试样本集用于分类器的壳检测。

### 5.1.3 评价指标

为了更好的描述实验结构，记TP（True Positive）为加壳样本分类正确的数量，FP（False Positive）为未加壳样本被分类为加壳样本的数量，FN（False Negative）为加壳样本被分类为未加壳样本的数量，TN（True Negative）为未加壳样本分类正确的数量。由这4种类型可以得到一个分类结果矩阵，如表5-1所示。

表5-1 分类结果矩阵

|  |  |  |
| --- | --- | --- |
| **预测样本类别**  **实际样本类别** | **加壳** | **未加壳** |
| 加壳 | TP | FN |
| 未加壳 | FP | TN |

基于以上分类结果矩阵，可以引申出以下各测试指标：

正确率（Precision），设表示在所有被分类器划分为加壳样本中，确实是加壳样本的概率，计算公式如下：

（5-1）

召回率（Recall），设表示在所有确实是加壳样本中，被分类器划分为加壳样本的概率，计算公式如下：

（5-2）

误报率（fallout），设表示在所有确实是未加壳样本中，被分类器划分为加壳样本的概率，计算公式如下：

（5-3）

F1值（F1-Score），设为正确率和召回率的加权调和平均，是二者的综合评价指标，具有更好的代表性，计算公式如下：

（5-4）

### 5.1.4 实验结果与分析

根据3.4节给出的方法，将所有未加壳样本和加壳样本转化为特征集后，保存为ARFF（Attribute-Relation File Format）格式文件，并输入到WEKA软件中进行实验，实验算法为WEKA自带的bayes-NaiveBayes算法（亦即朴素贝叶斯算法），不修改算法默认参数。

设表示加壳样本数量，表示未加壳样本数量，壳检测实验的结果如表5-1所示。

表5-1 壳检测实验结果

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **实际样本类别**  **/** | **预测样本类别**  **/** | **正确率** | **召回率** | **误报率** | **F1值** |
| 50/120 | 103/67 | 0.824 | 0.84 | 0.12 | 0.075 |

从表5-1的结果可知，本文实现的壳检测系统取得了较好的分类效果，其正确率、召回率以及F1值三项指标都在80%以上，并且误报率也较低。

通过对测试样本分析发现，壳检测系统在处理一些第三方SO文件时会出现误报或漏报的情况，通过对这些SO文件进行深入研究，发现其中部分SO文件由于本身就需要完成一系列的初始化工作，因此包含了1个以上的初始化函数，导致被错误分类为加壳样本，也就是出现误报。而另一部分SO文件虽然只包含1个JNI\_OnLoad函数且不具有其他分类特征，但是仍然具有加密代码的行为，因此属于一种强度很小的加壳SO文件，由于壳强度较小导致被错误分类为未加壳样本，也就是出现漏报。现阶段由于水平有限，本文的壳检测系统还无法处理上述SO文件。后续考虑可以收集更全面的加壳样本，从而得到一个更具有代表性的特征集，以提高分类器的壳检测效果。

总的来说，对于本文给定的实验数据和特征集来说，朴素贝叶斯分类算法能够较为准确地检测SO文件是否加壳，证明了本文采用的基于朴素贝叶斯分类器的壳检测方法具有一定的可行性，实现了论文预期的目标。

## 5.2 脱壳实验

### 5.2.1 实验环境与数据

脱壳实验是在一台基于Android系统的手机上进行的，其手机型号为Nexus 5，实验操作系统为Android 4.4。

实验数据为130个加壳样本，来源为网络中收集的一些恶意软件中的加壳SO文件以及各个加壳工具对SO文件进行加壳后生成的加壳SO文件。

### 5.2.2 实验结果与分析

脱壳实验共有2个，分别为脱壳系统运行实验和脱壳系统效果实验。

1. 脱壳系统运行实验

实验对象是一个使用几维工具加壳后的SO文件，实验目的是测试脱壳系统是否能正常运行。

该加壳SO文件中定义了一个JNI函数，该函数的作用是返回一个字符串“ I am ele7enxxh.”，函数的源代码如下：

jstring Java\_com\_ele7enxxh \_Test\_Test\_getStr( JNIEnv\* env, jobject thiz ) {

return (\*env)->NewStringUTF(env, "I am ele7enxxh.");

}

使用IDA解析后，反汇编效果如图5.1所示。

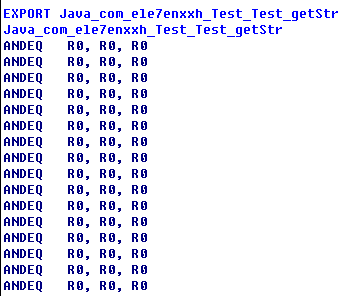


图5.1 加壳SO文件的反汇编效果

从图5.1中可以发现函数的实际代码被替换了，使用本文设计的脱壳系统对其进行脱壳，使用IDA对脱壳SO文件进行解析，反汇编效果如图5.2所示。

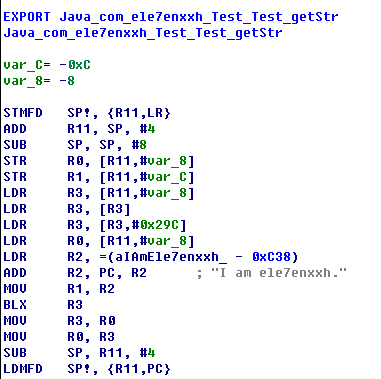


图5.2 脱壳SO文件的反汇编效果

从图5.2中可以发现已经函数的实际代码已经恢复了，接着用脱壳SO文件替换掉APK文件中的加壳SO文件，并且对APK文件重新签名，安装运行后，脱壳SO文件可以正常运行。

综述，本脱壳系统运行正常，即可以对该加壳SO文件完成脱壳，并且脱壳后的SO文件可以被静态分析和动态调试，这证明了本脱壳系统的设计原理是正确可行的。

2）脱壳系统效果实验

为了方便描述，本文的脱壳系统简称为SDR。在进行本系统实验的同时，使用udog和MGP作为实验数据的对比。实验结果如表5-2所示。

表5-2 脱壳实验结果

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **壳类型** | **能否脱壳** | | | **脱壳后能否运行** | | | **脱壳时间消耗/S** | | |
| **udog** | **MGP** | **SDR** | **udog** | **MGP** | **SDR** | **udog** | **MGP** | **SDR** |
| NAGA | √ | √ | √ | × | × | √ | 0.834 | 0.822 | 1.987 |
| 几维 | √ | √ | √ | × | × | √ | 1.08 | 1.56 | 2.139 |
| 梆梆 | × | × | × | × | × | × | 无 | 无 | 无 |
| 未知 | √ | √ | √ | × | √ | √ | 1.273 | 1.21 | 2.38 |

从表5-2可以看出，本文实现的脱壳系统可以对除了梆梆外的其它三种壳完成脱壳，并且脱壳后的SO文件可以正常运行。而udg和MGP两种脱壳工具虽然也可以完成脱壳，但是脱壳后的SO文件只能被静态分析，无法动态调试。从脱壳时间消耗来看，由于本系统具有更加完善的修复功能，因此较其它两种脱壳工具，消耗的时间会稍多，不过仍然控制在可接受范围内，平均只多耗时1秒。

通过对测试样本分析发现，本脱壳系统在处理梆梆工具加壳的SO文件时，不能完成脱壳。经过对该壳进行深入研究，发现梆梆壳的执行原理与其他壳不一致。与一般壳解密后则不再获得程序控制权不同，梆梆壳在运行过程中，会多次重新接管程序，采取了函数执行前解密，执行完毕后再次加密的保护手段，导致本文实现的脱壳系统的内存dump模块失败，最终造成脱壳失败。现阶段由于水平有限，本文的脱壳系统还无法处理这种类型的壳。后续考虑可以采用监控内存数据变化的方案确定内存dump的时机，从而进一步提高脱壳成功率。

总的来说，和udg和MGP两种脱壳工具相比，本文实现的脱壳系统能够对更多壳进行脱壳，并且脱壳后的SO文件可以正常运行。整个脱壳过程的时间消耗虽然更高，但仍然处于一个数量级之内。

## 5.3 本章小结

本章主要通过训练和测试样本对SO文件壳检测与脱壳技术进行实验。实验证明，本文采用的基于朴素贝叶斯分类器的壳检测方法和基于soinfo内存转存的脱壳方法具有较好的实际效果，达到了论文的预期目标。

# 第6章 总结与展望

越来越多的恶意软件将核心代码存储在SO文件，并且在软件发布前对SO文件进行加壳保护。查壳与脱壳是加壳恶意软件查杀过程中非常重要的一个环节，如果软件没有脱壳，则分析与破解无法进行下去。因此，研究基于Android平台的SO文件壳检测与脱壳技术是十分有必要的，本文主要包含两个研究点，一是完成了SO文件是否加壳的检测，二是在此基础上研究了自动化脱壳方法。

## 6.1 研究工作总结

本文主要做了以下工作：

首先，本文详细介绍了SO文件格式，让读者对壳检测与脱壳的研究对象有一个初步的了解，为后续的研究工作提供理论支撑。

其次，本文在对现有的加壳技术进行了研究之后，并且结合未加壳与加壳文件的差异性，采用了一种基于朴素贝叶斯分类器的壳检测方法。

接着，本文在对Android下SO文件的加载机制以及壳的加载机制进行了研究之后，采用了一种基于soinfo内存转存的脱壳方法。

最后，本文对壳检测系统和脱壳系统分别进行了实验测试和结果分析，结果表明壳检测系统和脱壳系统都具有良好的效果，基本完成了论文预期的目标。

本文的主要工作有以下3点：

1）研究了一种基于Android平台的SO文件壳检测技术

对一个文件进行脱壳，首先要做的就是判断该文件是否加壳。Window平台的PE文件壳检测技术发展成熟，PE壳检测方法多种多样。且都具有不错的检测率。然而，在Android平台下，查壳技术还十分匮乏，特别是对SO文件的壳检测技术还鲜有人研究，不论是各大安全论坛还是期刊杂志，都找不到相关研究资料。因此，寻找到一种可以有效检测Android平台的SO文件是否加壳的技术是有必要的。

在对现有的加壳技术进行了研究之后，并且结合未加壳与加壳文件的差异性，本文采用了一种基于朴素贝叶斯分类器的壳检测方法。该方法首先抽取出SO文件的初选特征，再利用LDA算法对初选特征进行线性变换，得到一个新的特征集，利用这个新特征集和训练样本对WEKA平台的朴素贝叶斯分类器进行训练，最后训练结束的分类器即可检验待测样本是否加壳。实验表明，该壳检测方法具有较高的准确率和较低的误报率，效果比较好。

2）研究了一种基于Android平台的加壳SO文件脱壳技术

和壳检测技术的的研究现状相同，目前Android平台下的加壳SO文件脱壳技术研究才刚刚起步。网络上可以找到一些分析人员利用各种逆向工具通过动态调试脱壳的技术文章，但是都没有形成一套成熟的脱壳理论。因此，寻找到一种通用自动化的的脱壳方法是有必要的。

在对Android下SO文件的加载机制以及壳的加载机制进行了研究之后，本文采用了一种基于soinfo内存转存的脱壳方法。该方法通过壳代码的存储位置获取目标壳程序执行完毕的时机点，也就是原程序的OEP，然后解析soinfo结构体，从而获得内存镜像的起始地址和长度大小，即可内存dump，完成初步脱壳。

3）研究了SO文件重构技术

一般情况下，由于各方面原因，初步脱壳后的SO文件还无法被IDA等逆向工具静态分析，更无法被动态调试。基于这一点，本文通过头部修复、段地址修复、重定位表修复、重建节头和清除壳入口实现了重构SO文件。该方法具有自动化程度高，不需要人工干预的优点。经重构后的SO文件可以被静态分析，也可以被动态调试，实现了完全脱壳。

## 6.2 研究展望

尽管本文对基于Android平台的SO文件壳检测与脱壳技术做了一定的研究，实验也表明其具有良好的效果，但是还存在一些不足，今后可从以下几个方面做进一步的研究：

1）壳检测方法的不足

本文所提出的壳检测方法主要存在两点不足。其一，只能检测SO文件是否加壳，但不能判定壳的具体类型，下一步工作可以结合特征码壳检测技术以确定壳类型。其二，本文由于时间有限，收集到的加壳样本数量和种类都是比较少的，这降低了特征集的准确性，使得最终的分类检测率的可靠性不强，下一步工作是从各个渠道收集更多的加壳样本，重新提取特征集，以提高分类器的分类效果，从而提高壳检测的能力。

2）内存dump时机点的问题

本文的脱壳思想基于壳的执行机制，就是壳首先获得程序控制权，然后在内存中自行脱壳，壳执行完毕后退出，控制权交还给原程序。目前而言，大多数壳几乎都是这种机制，但是为了避免被脱壳，壳程序也在不断发展，例如在程序正常运行时，壳可以重新接管控制权；虚拟机保护技术。因此，本文后续的研究工作是优化脱壳时机点的选择。

3）部分节头的重建问题

本文提出的重建节头的方案中，有部分节头是根据各个节之间的相对位置来重建的，例如.text、.ARM.extab、.rodata、.data这4个节。如果这4个节的相对位置发生了变化，就会导致重建失败。因此，本文后续的研究工作是寻找一种新的重建方法以避免上述缺点。

# 参考文献

# 作者在读期间科研成果简介

1. 韩子诺，刘嘉勇. 基于Android平台的SO加固技术研究[J]. 现代计算机:专业版, 2015(36).

# 致 谢

光阴荏苒，日月如梭，转眼间三年紧张而又充实的研究生生活即将画上句号。回想三年的研究生时光，无论是在生活方面还是学习方面我都受益颇多。在生活方面，我与实验室的同学们互相帮助、共同进步，彼此之间建立了良好的友谊；在学习方面，让我从一个对信息安全一窍不通的门外汉成为了一名合格的信息安全研究者。值此论文完稿之际，我要向所有在我读研期间给予我支持、帮助和鼓励的人表示我最诚挚的谢意。

首先，我要特别感谢我的指导老师刘嘉勇教授对我的悉心教导。从毕业论文的选题、构思、大纲撰写、方案的实现、初稿撰写到最终的定稿，都离不开刘老师对我的热情帮助和耐心指导，没有刘老师的指引，这篇论文是不可能完成的。刘老师丰富的专业知识、严谨的科研态度和对学术的钻研精神，都是值得我终生学习，并且受益终生的。

其次，感谢四川大学信息安全研究所的各位老师和同学们。感谢各位老师为我提供的丰富资源和项目平台，使我的专研能力得到了极大的提高，为我毕业后立足于社会提供了软件支撑。特别要感谢方勇教授、周安民教授、胡勇教授和刘亮老师对我生活以及学习上的关心和帮助。感谢信息安全专业的所有同学们，谢谢你们对我的包容和鼓励，特别要感谢刘露平师兄对我在专业能力上的帮助。

接着，感谢530寝室的所有室友们，谢谢三年来的朝夕相伴，感谢你们对我睡觉打鼾的容忍，在毕业分离之际，也衷心的祝愿你们事业有成，祝愿我们友谊长存。

最后，特别感谢我的父母，感谢你们二十多年的养育之恩，感谢你们对我的无私关爱，没有你们，就没有今天的我。正是有了你们一路上陪伴和支持，我才能安心学习，并顺利完成我的毕业论文。

# 声 明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了论文中特别加以标注和致谢的地方以外，论文中不包含其他人已经发表或撰写过的研究成果，也不含为获得四川大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

本学位论文成果是本人在四川大学读书期间，在导师指导下取得的，论文成果归四川大学所有，特此声明。

论文作者：

指导教师：

年 月 日

# 学位论文版权使用授权书

本学位论文的作者完全了解四川大学有关保留、使用学位论文的规定，有权利保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权四川大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩影或扫描等复印手段汇编学位论文。

（保密的学位论文在解密后适用本授权书）

学位论文作者签名： 导师签名：

签字日期： 年 月 日 签字日期： 年 月 日

学位论文作者毕业后去向：

工作单位： 电话：

通讯地址： 邮编：

1. [] 姜路, 李臻. 广播电台发展手机网业务的思考[J]. 广播与电视技术, 2010, 37(12):56-57. [↑](#endnote-ref-1)
2. [] 何昱晨, 石文昌. 基于Android移动客户端的互联网数据安全实证研究[J]. 中兴通讯技术, 2015(3):34-37. [↑](#endnote-ref-2)
3. [] 袁志坚,王春平,陈融等.Android平台安全威胁及其应对策略[J].计算机技术与发展,2 [↑](#endnote-ref-3)
4. [] 2014年Android恶意代码发展报告. http://blog.avlyun.com/. 2014. [↑](#endnote-ref-4)
5. [] Dex. <http://www.dex.com/>. 2016. [↑](#endnote-ref-5)
6. [] ELF Application Binary Interface. <https://docs.oracle.com>. 2016. [↑](#endnote-ref-6)
7. [] ZjDroid. <http://bbs.pediy.com/showthread.php?t=190494>. 2015. [↑](#endnote-ref-7)
8. [] Zhang Y, Luo X, Yin H. DexHunter: Toward Extracting Hidden Code from Packed Android Applications[M]// Computer Security -- ESORICS 2015. Springer International Publishing, 2015. [↑](#endnote-ref-8)
9. [] 黄剑军. 基于带权欧氏距离的壳检测与脱壳技术的研究[D]. 杭州电子科技大学, 2009. [↑](#endnote-ref-9)
10. [] 段钢. 加密与解密（第三版） [M]. 电子工业出版社, 2008. [↑](#endnote-ref-10)
11. [] Lyda R, Hamrock J. Using Entropy Analysis to Find Encrypted and Packed Malware[J]. IEEE Security & Privacy Magazine, 2007, 5(2):40-45. [↑](#endnote-ref-11)
12. [] Choi Y, Kim I, Oh J, et al. PE File Header Analysis-Based Packed PE File Detection Technique (PHAD)[C]// Computer Science and its Applications, 2008. CSA '08. International Symposium on. IEEE, 2008:28 - 31. [↑](#endnote-ref-12)
13. [] 黄步根. 计算机取证中系统分析技术研究[J]. 信息网络安全, 2011(3):39-42. [↑](#endnote-ref-13)
14. [] Alliance O H. Open Handset Alliance[M]. Betascript Publishing, 2010. [↑](#endnote-ref-14)
15. [] Android shipped 1 billion smartphones worldwide in 2014. <https://www.strategyanalytics.com/>. 2015. [↑](#endnote-ref-15)
16. [] 安卓设备基数已达二十亿. <http://pad.zol.com.cn/503/5032232.html>. 2015. [↑](#endnote-ref-16)
17. [] Wang C, Duan W, Ma J, et al. The research of Android System architecture and application programming[C]// Computer Science and Network Technology (ICCSNT), 2011 International Conference on. IEEE, 2010:785-790. [↑](#endnote-ref-17)
18. [] Kim Y J, Cho S J, Kim K J, et al. Benchmarking Java application using JNI and native C application on Android[C]// Control, Automation and Systems (ICCAS), 2012 12th International Conference on. 2012:284-288. [↑](#endnote-ref-18)
19. [] Chin R. The Android Native Development Kit (NDK)[M]. Apress, 2014. [↑](#endnote-ref-19)
20. [] 朱裕禄. Linux系统下的ELF文件分析[J]. 电脑知识与技术:学术交流, 2006(26):111-113. [↑](#endnote-ref-20)
21. [] 张和君, 张跃. Linux动态链接机制研究及应用[J]. 计算机工程, 2006, 32(22):64-66. [↑](#endnote-ref-21)
22. [] 何先波, 唐宁九, 吕方,等. ELF文件格式及应用[J]. 计算机应用研究, 2001, 18(11):144-145. [↑](#endnote-ref-22)
23. [] 刘晓冬. 软件加壳技术的研究与实现[D]. 沈阳工业大学, 2006. [↑](#endnote-ref-23)
24. [] 高艳军. 数据安全管理系统加壳技术研究与实现[D]. 国防科学技术大学, 2008. [↑](#endnote-ref-24)
25. [] 张译恬, 王纯. 基于安卓系统JNI机制的SO库加固方案设计[J]. 电信技术, 2014(10):90-93. [↑](#endnote-ref-25)
26. [] Javed A, Akhlaq M. On the Approach of Static Feature Extraction in Trojans to Combat against Zero-Day Threats[C]// International Conference on It Convergence and Security. IEEE, 2014:1-5. [↑](#endnote-ref-26)
27. [] 丁浩. 数据挖掘中常用分类算法的分析比较[J]. 菏泽学院学报, 2015(5):47-50. [↑](#endnote-ref-27)
28. [] Webb A R. Linear Discriminant Analysis[J]. Institute for Signal and Information Processing, Mississippi State University [Online]. Available: http://www.isip.msstate. edu/publications/reports/isip\_internal/1998/linear\_discrim\_analysis/lda\_theory.pdf, 2010, 3(6):237-280. [↑](#endnote-ref-28)
29. [] 莫增文. 基于时间序列分析技术的预测模型设计与应用[D]. 中国科学院大学(工程管理与信息技术学院), 2014. [↑](#endnote-ref-29)
30. [] Petroianu P D M G, Osswald M P M. Bayes-Theorem[M]. Springer Berlin Heidelberg, 2000. [↑](#endnote-ref-30)
31. [] Kohavi R, John G H. Wrappers for feature subset selection[J]. Artificial Intelligence, 1997, 97(1–2):273-324. [↑](#endnote-ref-31)
32. [] 李文. 基于壳技术的软件保护研究[D]. 电子科技大学, 2012. [↑](#endnote-ref-32)
33. [] 余三超. 基于虚拟机的通用自动化脱壳系统[D]. 电子科技大学, 2010. [↑](#endnote-ref-33)
34. [] Zhang L, Kong X. Embedded Trusted Computing Environment Build Based on QEMU Virtual Machine Architecture[C]// Seventh International Symposium on Computational Intelligence and Design. IEEE, 2015. [↑](#endnote-ref-34)
35. [] 李露, 刘秋菊, 徐汀荣. PE文件中脱壳技术的研究[J]. 计算机应用与软件, 2010, 27(9):279-282. [↑](#endnote-ref-35)
36. [] 赵中树. 基于Windows平台的脱壳技术研究与实现[D]. 电子科技大学, 2012. [↑](#endnote-ref-36)
37. [] Iwamoto K, Wasaki K. Detecting Original Entry Point based on Comparing Runtime Library Codes in Malware Unpacking[J]. Ieice Technical Report Internet Architecture, 2011, 111:57-62. [↑](#endnote-ref-37)
38. [] 钟足峰, 邬建平, 胡逢彬. 随机融合自修改加壳方法研究应用[C]. International Conference on Future Information Technology and Management Engineering. 2010. [↑](#endnote-ref-38)
39. [] 黄剑军. 基于带权欧氏距离的壳检测与脱壳技术的研究[D]. 杭州电子科技大学, 2009. [↑](#endnote-ref-39)
40. [] Remco R. Bouckaert, Eibe Frank, Mark A. Hall,等. WEKA—experiences with a Java Open-Source Project[J]. Journal of Machine Learning Research, 2010, 11(5):2533-2541. [↑](#endnote-ref-40)