软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2017,28(9):2264−2280 [doi: 10.13328/j.cnki.jos.005181]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种基于 Huffman 和 LZW 编码的移动应用混淆方法*

李承泽, 於剑波, 张 淼, 徐国爱, 孔浩浩

(北京邮电大学 网络空间安全学院,北京 100876)

通讯作者: 李承泽, E-mail: xyq547133@163.com

摘 要: 二进制混淆技术在规避恶意软件分析、防止利用逆向工程篡改中扮演着重要的角色.一些广泛使用的混淆技术关注于基于语法的检测,基于语义的分析技术在很多年前也已经被提出以防止逃避检测.近年来,一些考虑到统计特征和基于语义的二进制混淆技术开始被提出,这些方法开始关注混淆的隐蔽性,但总体来说效率较低或无法同时考虑到安全性的要求.提出一种针对 Android 移动应用的、基于 Huffman 编码和 LZW 编码的二进制混淆技术,同时将强度、开销和隐蔽性等考虑在内,具备规避基于统计特性和语义特征检测的能力.该技术构造混淆所需的指令编码表,一方面利用编码表对原始指令序列进行置乱,提高混淆技术的隐蔽性;另一方面,将核心编码表从代码执行数据段分离,通过白盒 AES 加密的方式在提高混淆技术本身安全性的同时隐藏密钥及密钥查找算法.研发出该技术工具原型 ObfusDroid.最后,从安全强度、开销、平台适应性和隐蔽性这几个方面,对该技术进行评估和阐述.

关键词: 二进制;混淆;隐蔽性;Huffman;LZW;白盒 AES 加密
中图法分类号: TP311

中文引用格式: 李承泽,於剑波,张淼,徐国爱,孔浩浩.一种基于 Huffman 和 LZW 编码的移动应用混淆方法.软件学报,2017,28(9):2264−2280. http://www.jos.org.cn/1000-9825/5181.htm

英文引用格式: Li CZ, Yu JB, Zhang M, Xu GA, Kong HH. Obfuscation tool for mobile apps based on huffman and LZW encoding. Ruan Jian Xue Bao/Journal of Software, 2017,28(9):2264−2280 (in Chinese). http://www.jos.org.cn/1000-9825/5181.htm

Obfuscation Tool for Mobile Apps Based on Huffman and LZW Encoding

LI Cheng-Ze, YU Jian-Bo, ZHANG Miao, XU Guo-Ai, KONG Hao-Hao

(School of Cyber Space Security, Beijing University of Posts and Telecommunications, Beijing 100876, China)

Abstract: Binary obfuscation plays an essential role in evading malware analysis and tampering with reverse engineering. Some widely used code obfuscation techniques focus on evading syntax based detection, however semantic analysis techniques have been developed to thwart their evasion attempts. Recently some binary obfuscation techniques with potential of evading both statistical and semantic detections have been proposed, taking concealment into account but lacking efficiency or security strength. This study proposes a binary obfuscation technique for mobile apps based on LZW and Huffman encoding to offer the potential of evading both statistical and semantic detections while taking intensity and concealment into account. This technique constructs the required instruction encoding tables. On one hand, it scrambles the sequence of original instructions with encoding tables to improve the intensity and concealment. On the other hand, it reinforces intensity by separating the encoding tables encrypted by white-box AES from code segment, concealing the key and lookup algorithm, in order to evading attacks on keys. A prototype tool for this technique, called ObfusDroid, is put forward, and an evaluation on ObfusDroid is given from aspects of intensity, cost, compatibility and concealment to demonstrate its capability of evading statistical analysis.

Key words: binary; obfuscation; concealment; Huffman; LZW; White-box AES cryptography

* 基金项目: 国家高技术研究发展计划(863)(2015AA017202)
Foundation item: National High Technology Research and Development Program of China (863) (2015AA017202)
收稿时间: 2016-07-10; 修改时间: 2016-09-04; 采用时间: 2016-11-10; jos 在线出版时间: 2017-02-20
CNKI 网络优先出版: 2017-02-20 14:09:05, http://www.cnki.net/kcms/detail/11.2560.TP.20170220.1409.023.html

过去几十年,在软件混淆(obfuscation)领域,很多学者已经做过很多研究工作.针对传统 PC 软件,许多可行的方法被提出,也有很多成熟的混淆工具诞生.混淆技术在规避恶意软件分析、防止利用逆向工程篡改中发挥着非常重要的作用,尤其是针对传统 PC 软件领域中的可执行 PE 文件.在过去 5 年,随着移动互联网的迅猛发展,移动应用软件(本文简称应用)的数量持续高速增长,与此同时,移动应用的安全问题也急剧增加,尤其是 Android 平台应用.

Android 应用使用 Java 语言开发,运行在 dalvik 或者 ART 虚拟机环境中.在开发过程中,源代码被编译得到 dalvik 字节码,直接运行在 dalvik 环境或经过优化后运行在 ART 环境中.Android 应用可以运行在手机、平板电脑、智能手表或者其他移动终端中,这些设备所处的环境非常复杂,攻击者可以通过将应用安装在由他们定制的终端中实现对应用的破解、篡改,移动应用的安全形势非常严峻.据我们的统计分析数据,相当一部分应用软件被黑客攻击,应用面临着破解、去除广告、嵌入恶意代码等安全威胁.

早期的混淆研究主要集中在混淆算法本身上,Collberg[1,2]最早将混淆技术分为布局混淆、数据混淆、控制流混淆和预防性混淆.进入移动互联网时代后,为了应对上述形势,许多针对移动应用的保护技术已经被提出,移动应用的混淆方法已经不仅仅局限于 Collberg 提出的体系中,许多其他技术也逐渐结合起来,例如基于源代码的混淆技术、基于二进制的加壳技术、加密技术等.

对于软件混淆的评估指标,国内外都已经有了一定的理论研究成果,目前,研究主要集中在混淆强度、时间空间开销等方面.但软件混淆强度和开销并不是混淆技术评估的唯一指标,混淆算法是否容易被辨识出来,也是混淆算法评估的一个重要指标.通常来说,攻击者喜欢迎接挑战,换言之,攻击者喜欢攻击被保护过的应用软件.如果忽视了保护的隐蔽性,保护措施越多,应用软件越容易成为攻击者的目标.攻击者可以通过统计特性,很容易地找出一款应用软件是否被保护、使用哪种方式进行了保护.一旦攻击者确认了保护的方式,通过逆向、动态调试、APIHook 等技术,可以很轻易地在自己特定的终端中实现对应用软件的破解.

为了应对这种情况,本文介绍了一种基于 Huffman 编码和 LZW 编码的混淆技术,将应用保护的强度、开销和隐蔽性等同时考虑在内.使用这种混淆技术,攻击者很难从统计特性和语义特征层面区分出应用是否被保护,以及用什么方式进行了保护.在混淆前,该技术方法会对代码片段中的所有指令按混淆算法生成的次序重新组合.然后,我们将该技术方法实现,研发出一款关注安全保护隐蔽性的混淆原型工具.最后,我们对这种技术进行评估、对比分析.本文前言和第 1 节介绍相关背景和相关工作.第 2 节、第 3 节讲述本文提出的算法、架构和实现技术.第 4 节用实验对混淆算法进行评估和分析.

1 相关工作

Collberg 最早提出了 4 项混淆算法评估的指标:强度(intensity)[1]、弹性(resilience)[1]、开销(cost)[1,3]和隐蔽性(stealth,concealment)[3].其中,强度表示算法对程序增加的复杂度,一般来说,混淆强度越大,混淆后的应用程序越难以理解;弹性主要反映的是混淆算法前后,应用软件对程序运行的平台适应性;开销指的是混淆过程中由代码转换带来的时间和空间的额外开销;隐蔽性[3]是 Collberg 最后提出的一个指标,代表混淆方法本身是否容易暴露出导致混淆方法被辨别出来因素.

目前已经存在的移动应用混淆技术主要关注的是信息隐藏、通过加密或编码的方式保护应用中的元数据和应用数据、置乱应用程序的控制流,目的是为了增加逆向工程和理解代码的难度.

Wang 等人[4]提出了平展控制流的混淆算法,在某种程度上能够抵抗静态反汇编分析.后来,Balachandran 等人[5]使用不透明谓词断言的方法改进了平展数据流的混淆算法.但无论是原本的方法还是改进后的方法,由于在应用软件运行之前所有混淆代码都可以静态加载,难以抵制动态分析和动态调试.目前,在移动应用平台中已经出现了多种动态测试框架,例如 iOS 平台的 Cydia、Android 平台的 Cydia Substrate 和 Xposed 等框架.在这些框架的基础上进行二次开发,可以实现在终端上的行为监测.一旦攻击者确认了采用平展控制流的混淆算法,攻击者将混淆后的应用软件运行在动态测试终端中,能够通过 API 和参数分析获取到原始应用软件的运行数据.

陈等人[6]提出了一种改进的基于同态加密的移动应用混淆技术.同态加密[7,8]最早由 Sander 等人提出,是基

于数学难题的计算复杂性理论的密码学技术.与传统的数据加密不同,同态加密允许在没有解密算法和解密密钥的条件下对加密数据进行运算,解密后的结果和明文状态下直接计算的结果相同.陈等人改进了 Sander 提出的同态加密算法,解决了原本同态加密存在的问题,但是仍然有很多问题没有办法解决,同态加密只限于整数环的加密,被加密的函数只局限于实数域的初等函数,对关系运算加密的计算和通信开销过大.在移动应用混淆中,由于存在上述提到的不足,基于同态加密的移动应用混淆技术难以覆盖到应用中的全部函数,并且混淆计算的开销对于移动平台应用来说代价难以接受.

Rong 等人[9]提出了一种基于重加密的安全混淆方法,主要为了解决云平台中隐私数据的可用性和隐私性问题.在基于重加密的安全混淆方法中,需要用到重加密函数来隐藏所有的隐私数据.作者在文献[9]中重点阐述了对几种重加密函数的研究,并在此基础上实现了对隐私数据的安全混淆.基于这种方法,可以对移动应用中的敏感数据和代码进行安全混淆,但是基于重加密的安全混淆方法依然主要依靠加密的方式进行保护,在混淆之后,被安全混淆方法保护的数据和代码,随机性和信息熵发生了很大的变化.徐国爱等人[10]提出:通过统计特性就可以辨别出是否被保护,甚至可以通过随机性和信息熵判断出使用了哪种加密方式.

作为 Collberg 等人[1]提出的混淆评估模型中的重要部分,不仅强度和隐蔽性需要考虑在内,开销也是一个重要的评估指标.通常来说,代码混淆过程会在原代码片段的基础上加入一定的花指令或冗余代码等,在混淆过程中,一般都会带来空间和时间上的开销.何炎祥等人[11]提出了一种自适配的代码混淆方法,他们关注的是降低混淆带来的开销研究.这种方法不会对整个源代码进行混淆,而是使用一种流敏感方法,选择高度相关的代码段部分进行保护.Wu 等人[13]考虑到混淆隐蔽性,提出了一种基于 Huffman 编码的二进制混淆方法,以应对基于统计特性和语义分析的检测.这种方法会对待保护的二进制程序片段进行自动化切分,对每一个部分计算出编码所需的 Hufman 编码树.在混淆过程中需要用到模仿函数(mimic fuction),模仿函数会根据指令的频率分布和熵值,对原始的指令序列进行重新排列.Huffman 编码树是一种自下向上的树,因此对 Huffman 树中的节点,只会出现两种情况:该节点没有子节点,或者该节点有两个子节点.模仿函数重新排列过程实际上与 Huffman 编码解压缩过程相似,因此在混淆过程中,每一个字节的指令会以原指令序列为元指令进行扩展,因此导致了过大的空间开销.

为了解决上述的问题和不足,本文提出了一种基于 Huffman 编码和 LZW 编码的移动应用混淆技术.考虑到 Huffman 编码在混淆中带来的过多的冗余开销,而 LZW 编码对原始数据长度并不会导致过多的冗余,因此在我们的方法中,我们使用 LZW 编码对指令片段中的指令序列进行重新排列,而使用 Huffman 编码对每一种类型的指令参数进行重新排列.在混淆前后,冗余的代码指令全部基于原有指令生成并重新排列,因此,混淆前后,指令的随机性和信息熵并不会发生很大的变化,从某种意义上保证了混淆方法的隐蔽性,也降低了混淆方法带来的空间开销.从性能上来说,LZW 编码也是一种非常高效的编码算法,保证了混淆的时间开销.此外,在混淆过程中,一段代码片段会根据不同的指令生成一个 LZW 编码表,会根据相同指令的不同参数生成多个 Huffman 编码表,上述两类编码表可以通过动态方式加载,编码表独立于整个代码片段存在,可以通过多种不同的加密方式对编码表进行保护,混淆方法的安全强度主要取决于对编码表的加密强度.考虑到经典对称密码和公钥密码分别涉及到密钥和证书的保存及使用等问题,本文采用白盒 AES 算法,在对编码表进行保护的同时实现了密钥的隐藏.

本文的贡献在于以下几点:

(1) 利用 LZW 编码对主体指令进行混淆,极大地降低混淆在时间上和空间上的开销;

(2) 结合 LZW 和 Huffman 编码方式,保证了混淆方法的隐蔽性;

(3) 混淆所需编码表,独立于整个代码段,整个混淆方法的安全强度取决于对编码表的加密强度.使用白盒 AES 算法对编码表进行保护,在保证安全性的同时,合理隐藏密钥及加解密方法,抵抗密钥攻击.

2 算法设计

在整个混淆设计流程中,包含两个重要流程模块:混淆模块和运行模块.

混淆模块包含 4 个组件,分别是:反编译器(disassembler)、预处理器(pre-processor)、混淆器(obfuscator)和重

编译器(re-assembler).其中,

- 反编译器将应用程序中的可执行程序(DEX,ODEX 等)逆向,将指令解析到 Dalvik 指令层面;
- 预处理器对反编译得到的中间代码进行分析,得到混淆器所需要的重要信息,并且计算指令和参数等统计数据.之后,混淆器根据统计数据对整个代码进行分片,对分片中的指令进行摘要;
- 然后,混淆器为每一个片段生成一个 LZW 编码表,为片段中每一类指令生成一个 Huffman 编码表,由这些编码表构成指令摘要表(instruction digest table,简称 IDT).之后,混淆器会根据生成的指令摘要表,使用混淆算法将指令进行置乱;
- 重编译器将置乱后的指令重新打包编译.打包编译过程中,使用加密算法对指令摘要表进行保护,考虑到加密算法和密钥的安全性,使用白盒 AES 算法对指令摘要表进行保护.与此同时,将用来解析加载的壳嵌入到重新编译得到的可执行程序中.

整个过程如图 1 所示.

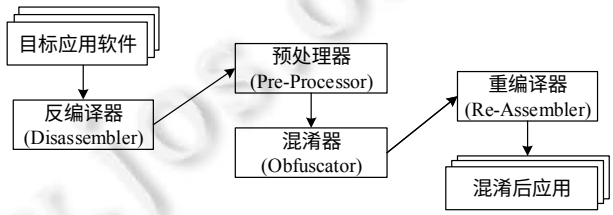


Fig.1 Design process of the obfuscation module

图 1 混淆模块设计流程

运行模块被嵌入在混淆后应用的加载壳中.运行模块包含 3 个组件,分别是预处理器(pre-processor)、解码器(decoder)和加载器(instruction-loader).其中,预处理器对混淆后的应用进行初步解析,从中提取并解密得到指令摘要表;根据指令摘要表,解码器计算出指令序列的真实顺序,并通过寻址计算找到每一条指令的实际位置;运行模块并不对混淆后的可执行程序进行恢复处理,加载器会按照真实的指令顺序找到下一条要执行语句的地址,直接将指令加载到内存中,执行相应的代码片段.整个过程如图 2 所示.

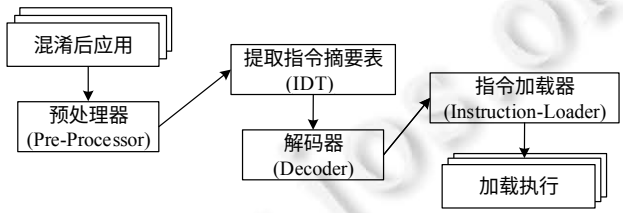


Fig.2 Design process of the execution module

图 2 运行模块设计流程

2.1 预处理:指令结构

LZW 和 Huffman 编码是两种常用的编码算法,广泛应用在数据压缩领域.Huffman 编码是理论上的空间最优编码算法,但是 Huffman 编码的执行效率非常差.经实际测试,1GB 数据使用 Huffman 进行完全压缩需要花费大约半个小时,时间代价让人难以接受.相比而言,LZW 的执行效率则高的多.更重要的是,本文的研究目的是借助于编码算法实现混淆的目的,以提高程序本身的安全性,并非研究压缩算法.

在混淆模块的预处理阶段,指令首先根据函数作用范围(function scope),将整个可执行代码切分成若干个指令(instruction)片段.每一条指令都可以被转换成两个字段:指令前缀(prefix)字段和参数(parameter)字段,前缀字段和参数字段都是二进制操作码.首先,预处理过程中,预处理器会对指令进行摘要,得到每一条指令的指令

前缀.例如,操作码 0x01 表示将第 2 个参数的值赋值到第 1 个参数中,而操作码“0x07”则表示将第 2 个参数的引用赋给第 1 个参数.尽管这两个指令操作存在很大的区别,但他们都具有赋值的含义,因此,我们为这两个操作码设置共同的指令前缀 MOV.因此,许多不同的指令会被摘要成相同的前缀.具备相同前缀的不同指令,表现在参数不同.我们用如图 3 所示的结构来表示指令.

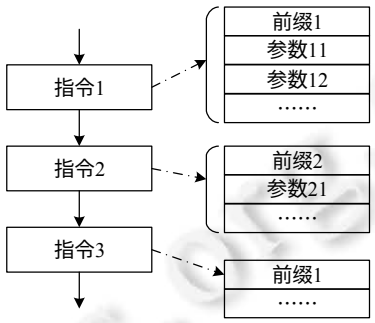


Fig.3 Structure of instructions

图 3 指令数据结构

如图 3 所示,每一条指令可以分成前缀字段和参数字段,指令 1 和指令 3 具有相同的指令前缀,但它们的指令并不相同,说明指令参数结构不同.

2.2 预处理:指令摘要表

LZW 算法的核心目的是尽可能减少重复指令的出现次数,在计算的过程中会建立并维护一个 LZW 编码表.在算法的初始阶段,构造一个空的编码表.编码表中用来记录已知的原子指令或指令序列,所有被添加到编码表中的原子指令或指令序列都被认为是已知序列.首先,将代码片段中的原子指令添加到编码表中,然后依次将最短未知序列添加到编码表中,用该序列在编码表中的序号代替该序列.LZW 编码表生成算法如下所示.

算法 **1**. LZW 指令编码表生成算法.

1. 初始化 LZW 编码表;
2. 对指令片段中的指令进行摘要;
3. 遍历片段中所有原子指令前缀,如果该前缀不在 LZW 编码表中,添加该前缀到 LZW 编码表中;
4. 从起点重新遍历指令前缀,记为 $atom_i$,记录指令前缀序列,如果序列满足以下条件:

$$\begin{cases}\langle atom_1,atom_2,...,atom_m\rangle \in Table_{LZW}\\ \langle atom_1,atom_2,...,atom_m,atom_{m+1}\rangle \notin Table_{LZW}\end{cases},$$

则称$\langle atom_1,atom_2,\ldots,atom_m\rangle$是最长已知序列,$\langle atom_1,atom_2,\ldots,atom_m,atom_{m+1}\rangle$是最短未知序列.将序列$\langle atom_1,atom_2,\ldots,atom_m,atom_{m+1}\rangle$添加到 LZW 编码表中;

5. 计算并返回输出.如果序列$\langle atom_1,atom_2,\ldots,atom_m\rangle$在 LZW 编码表中序号为 $Count_m$,则此时输出值为 $Count_m$;
6. 以$\langle atom_{m+1}\rangle$为序列的起始节点,重复步骤 4~步骤 6 操作,直到整个序列完全遍历;
7. 得到输出序列.

如图 4 所示为一段指令序列的 LZW 编码表生成流程.在图 4 中,首先初始化 LZW 编码表,对该片段中的所有指令进行摘要,用“指令 A”表示指令前缀为 A 的一类指令;指令前缀序列为$\langle A,B,B,A,B,A,B,A,C\rangle$,然后将序列中的所有的原子指令前缀$\{A,B,C\}$加入到 LZW 编码表中,分别标记序号为 1,2,3;从头开始遍历指令并记录指令序列,$\langle A\rangle$序列存在于编码表中,而$\langle A,B\rangle$不在编码表中,因此将序列$\langle A,B\rangle$添加到编码表中,依次编码,即为 4,输出为$\langle A\rangle$序列的序号,即 1;然后从指令$\{B\}$(第 1 个指令 B)起,重复上述计算过程,依次得到 LZW 编码表中的输出.

Fig.4 Generating process of LZW encoding table

图 4 LZW 编码表生成流程

Huffman 算法的核心目的是通过构造编码树,实现对指令的快速寻址.由于 Huffman 编码是空间最优的编码算法,代价是时间开销过大,而且编码树越大,时间开销越难以接受.因此在本文的混淆算法中,如何合理控制 Huffman 编码树的规模也是一个重要问题.为了解决这个问题,在本文中,Huffman 编码仅仅针对某一指令片段中前缀相同指令的参数进行编码.首先,对同前缀指令的参数类型和频数进行统计,将不同参数类型指令作为节点,按频数从低到高排列;将频数最低的两个节点合并,作为一个新节点,新节点的频数为两节点频数之和,即两节点成为新节点的叶子节点;以此方法重复排序、合并,最终构造成一个完整的编码表.在某一个指令片段中,会生成若干个 Huffman 编码表.Huffman 编码表生成算法见算法 2.

算法 **2**. Huffman 指令编码表生成算法.

1. 对同前缀指定的参数类型和频数进行统计;
2. 将不同参数类型指令作为节点,按照频数从低到高排列,节点记为$\{A,B,C,D,E,\ldots,N\}$,节点对应频数记为$\{a,b,c,d,e,\ldots,n\}$;
3. 获取频数最低的两个节点,即$\{A,B\}$,以他们为叶子节点构造成树,记节点为$\{A+B\}$,频数为 $a+b$;
4. 将节点$\{C,D,E,\ldots,N,A+B\}$重新排序,重复步骤 2~步骤 4 操作,直到所有节点都被添加到同一棵树中;
5. 此二叉树即为 Huffman 编码表.

如图 5 所示为一段指令序列的 Huffman 编码表生成流程.

Fig.5 Generating process of Huffman encoding table

图 5 Huffman 编码表生成流程

如图 5 所示为指令⟨1,2,3⟩的 Huffman 编码表生成流程.针对同一代码片段中同一指令前缀的指令参数,生成一个 Huffman 编码表,由图 5 可见,共生成了两个 Huffman 编码表.指令 1 有两种参数类型,频数分别为 i 和 j(图

中 $i=j=1$),因此得到右上角的 Huffman 编码树,其中,参数 1 类型对应左子树,寻址记为 0;参数 3 类型对应右子树,寻址记为 1.

在预处理阶段模型中,待混淆的二进制可执行应用程序,被切分成若干片段.在每一个指令片段中会有大量的指令,每一个指令都包含指令前缀.对于每一个指令片段,存在一张编码整个片段中指令前缀的 LZW 编码表.对于每一种前缀,都会有一张 Huffman 编码表来记录所有含有该前缀指令的参数情况.我们把 LZW 编码表和 Huffman 编码表的集合成为指令摘要表(instruction digest table,简称 IDT).指令摘要表中的每一条记录,在混淆和运行过程中,都会被转换成指令模板(instruction template,简称 IT).混淆器和加载器都是通过指令模板对待保护和待执行的指令进行解析.

2.3 混淆——核心阶段之一

在编码阶段,LZW 和 Huffman 是最核心的编码算法.与压缩过程中用到这两种算法不同,我们并不使用它们来减少空间利用率,而是利用它们实现指令顺序的置乱.本文提到的混淆、加载的过程与压缩、解压缩的过程完全相反,在混淆过程中,我们先根据得到的指令摘要表进行解压缩,在加载过程中进行压缩寻址.

混淆过程中,LZW 编码和 Huffman 编码结合使用的算法实现见算法实现 1.

算法实现 **1**. 混淆.

输入:

Bin:输入指令序列;

LZWTable:预处理阶段得到的 LZW 指令摘要表;

HuffmanTable:预处理阶段得到的 Huffman 指令摘要表;

LZWOutputSequence:LZW 编码过程输出序列;

计算:

```
Initialize LZWOutputSequence;
if Bin not empty then
    while (item=Bin.getCurrentItem()) not null do
        tmpOutput=Lookup(LZWTable,item);
        Append tmpOutput to LZWOutputSequence;
    end while
end if
if LZWOutputSequencenot empty then
    Update ResultSequence with LZWOutputSequence {
        TmpResultSequence=LZWOutputSequence-1;
    };
    while (item=TmpResultSequence.getCurrentItem()) not null do
        parameters=Lookup(HuffmanTable,item);
        updateParameters(item,parameters);
    end while
end if
ResultSequence=ReAssemble(TmpResultSequence);
```

输出:

ResultSequence.

在预处理阶段,对待混淆应用根据函数作用域范围进行切分,得到很多指令序列片段.对于每一个片段,计算出一个 LZW 编码表和 Huffman 编码表.指令序列片段、LZW 编码表和 Huffman 编码表为混淆算法的输入.

对于整个指令序列,我们查询 LZW 编码表,得到指令的编码输出,将输出结果记录到 *LZWOutputSequence*

中,直到整个指令序列完全遍历.之后,我们对 *LZWOutputSequence* 进行逆序操作,得到序列 *TmpResultSequence*.混淆过程中,LZW 编码流程如图 6 所示,*TmpResultSequence* 中,指令序列顺序与最终混淆后指令序列顺序基本一致.对于 *TmpResultSequence* 中的每一个指令,都存在一个 Huffman 编码表对其参数进行混淆,参数混淆时基于指令的机器码直接用 Huffman 算法进行寻址展开处理,最终将一条指令扩展成多条指令.Huffman 编码流程如图 7 所示.

Fig.6 LZW encoding process during obfuscation

图 6 混淆阶段 LZW 编码流程

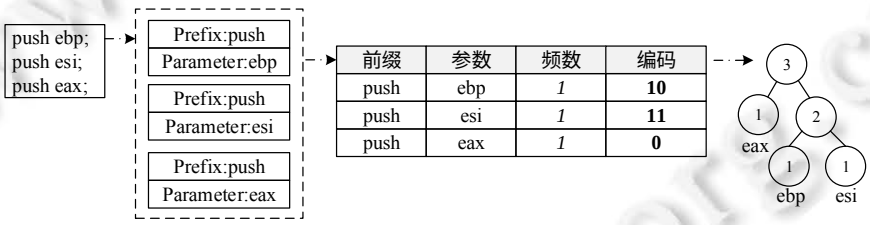


Fig.7 Huffman encoding process during obfuscation

图 7 混淆阶段 Huffman 编码流程

指令 *A* 表示指令前缀为 *A* 的一类指令.在图 6 中,根据预处理得到的 LZW 编码表,原指令序列的编码输出为⟨1,2,2,4,7,3⟩,经逆序得到混淆输出为⟨3,7,4,2,2,1⟩.根据编码表指令前缀序列⟨*A*,*B*,*A*,*B*,*A*,*B*,*A*,*C*⟩先经切分转变为序列⟨⟨*A*⟩,⟨*B*⟩,⟨*B*⟩,⟨*A*,*B*⟩,⟨*A*,*B*,*A*⟩,⟨*C*⟩⟩,进而逆序为⟨⟨*C*⟩,⟨*A*,*B*,*A*⟩,⟨*A*,*B*⟩,⟨*B*⟩,⟨*B*⟩,⟨*A*⟩⟩.为了保证混淆加载的安全性,LZW 编码表在保存过程中,并不会把 LZW 编码的输出进行保存.

对图 7 所示指令片段中同一指令前缀的指令参数进行 Huffman 编码,得到右边所示 Huffman 编码表.图中指令 *push ebp* 对应机器码为 0x01010000,根据机器码进行直接寻址,该指令即被扩展成多条指令(e.g.,*push eax*, *push ebp*,*push ebp*,...).

对于攻击者来说,如果无法完整获取到 LZW 编码表,则无法得到指令加载的正确顺序;如果无法完整获取到 Huffman 编码表,则无法确认多条含参数指令中哪些指令是冗余指令.指令摘要表体积很小,可以通过各种公钥体系算法进行加解密处理,保证指令摘要表的安全性.

2.4 解码——核心阶段之一

混淆后,应用安装、运行在 Android 智能终端中,需要经过解码阶段,指令序列才会被正确地加载到内存中执行.解码器和指令加载器是解码过程中需要用到的两个组件.

在解码阶段,首先,预处理器会从整个应用程序中找到指令摘要表;解码器根据指令摘要表计算出能够用来解析混淆后应用的指令模板,使用 LZW 算法和 Huffman 算法计算出正确的指令序列,寻址得到正确的指令参数;指令加载器将正确指令地址对应的指令加载到内存中,指令执行.

算法实现 **2**. 解码.

输入:

Bin:输入指令序列;

LZWTable:预处理阶段得到的 LZW 指令摘要表;

HuffmanTable:预处理阶段得到的 Huffman 指令摘要表;

计算:

```
Initialize OutputSequence;
if Bin not empty and LZWTable not empty and HuffmanTable not empty then
    LoadTableIntoMem(LZWTable,HuffmanTable);
    TmpInputSequence=InstructionSplitByLZW(Bin,LZWTable);
    while (itemList=TmpInputSequence.getCurrentItem()) not null do
        tmpOutput=InstructionRestoreByHuffman(HuffmanTable,itemList);
        Append tmpOutput into TmpResultSequence;
    end while
    while (item=TmpResultSequence.getCurrentItem()) not null do
        ReAddress=RelocateInstruction(LZWTable,item);
        Append ReAddress into ResultSequence;
    end while
end if
```

输出:

ResultSequence.

首先,加载器将 LZW 编码表和 Huffman 编码表加载到内存中,将混淆后的指令序列作为输入.当指令摘要表和输入指令序列非空时,解码器对输入序列进行解码,记录解码得到的寻址地址,经地址序列记录并输入到加载器中.Huffman 解码算法见算法 3.

算法 **3**. Huffman 解码算法.

1. 解码器按照指令前缀,提取出每一个相邻的、指令前缀相同的指令序列片段;
2. 根据前缀找到对应的 Huffman 编码表;
3. 用 Huffman 编码表中的编码表示指令;
4. 拼接指令编码,恢复原始指令的机器码;
5. 将各个片段恢复得到的机器码保存,得到新的指令序列.

如图 8 为一段指令序列的 Huffman 编码表生成流程.

如图 8 所示,左侧指令序列中的 6 个“push”指令解码所需 Huffman 编码表为同一张表.将每条指令对应的编码值获取到,并将编码值进行拼接,得到混淆前的指令.

通过 Huffman 解码计算得到新的指令序列,根据 LZW 编码表和 LZW 的寻址方式(如下所示)计算出全部指令的实际寻址地址.

定义(最大匹配指令序列). 设 $EncodingTable_{LZW}$ 为 LZW 编码表,$\langle instr_1,instr_2,\ldots,instr_i,\ldots,instr_n\rangle$为任意一段指令序列,$\{instr_i\}$为任意指令,如果指令序列满足$\langle instr_1,instr_2,\ldots,instr_n\rangle\in EncodingTable_{LZW}$,且$\langle instr_1,instr_2,\ldots,instr_n,instr_{n+1}\rangle\notin EncodingTable_{LZW}$,则称$\langle instr_1,instr_2,\ldots,instr_n\rangle$为 $EncodingTable_{LZW}$ 上的一个最大匹配指令序列.

定义(最大有效匹配指令序列). 设 $EncodingTable_{LZW}$ 为 LZW 编码表,序列 $\langle instr_1',instr_2',...,instr_{n-1}',instr_n'\rangle$ 和

$\langle instr_1,instr_2,\ldots,instr_{n-1},instr_n\rangle$为 $EncodingTable_{LZW}$ 上的前后相邻的两个最大匹配指令序列,则称满足$\langle instr_1,instr_2,\ldots,instr_k,instr_1'\rangle \in EncodingTable_{LZW}(1\le k\le n)$的最长序列$\langle instr_1,instr_2,\ldots,instr_k\rangle$为最大有效匹配指令序列.

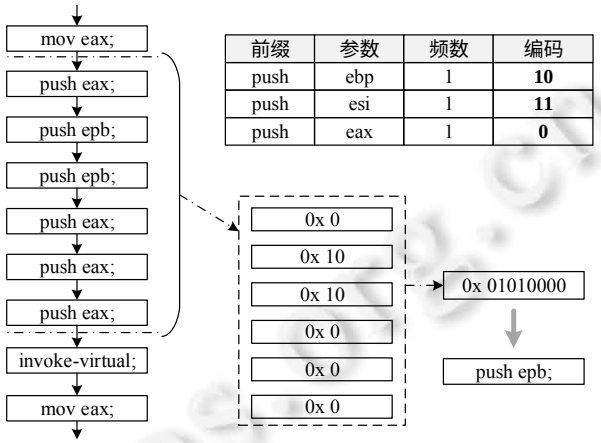


Fig.8 Process of Huffman decoding algorithm for some sequence of instructions

图 8 某个指令序列片段的 Huffman 解码算法流程

LZW 解码算法见算法 4.

算法 **4**. LZW 解码算法.

1. 从待解码指令序列头部起,选取第 1 个最大匹配指令序列,输出该序列在 LZW 编码表中的编号(ID);
2. 依次选取之后所有的最大有效匹配指令序列,依次输出这些序列在 LZW 编码表中的 ID;
3. 将得到的所有 ID 逆序排列;
4. 根据逆序后的 ID,从 LZW 编码表中,找到对应的指令.

如图 9 为一段指令序列的 Huffman 编码表生成流程.

Fig.9 Process of LZW decoding algorithm for some sequence of instructions

图 9 某个指令序列片段的 LZW 解码算法流程

如图 9 所示,左侧为混淆后的指令序列,其中,用下标区分先后出现的同前缀指令,如A_1,A_2等.根据 LZW 编码表,首先获取第 1 个最大匹配指令序列,为$\langle C\rangle$,输出为 3;从指令$\{C\}$起,依次选择之后的最大有效匹配指令序列,依次为$\langle A_1,B_1,A_2\rangle$,$\langle A_3,B_2\rangle$,$\langle B_3\rangle$,$\langle B_4\rangle$,$\langle A_4\rangle$,对应输出依次为$\langle 7,4,2,2,1\rangle$;最后,逆序后得到$\langle 1,2,2,4,7,3\rangle$.在解码器完成重新寻址后,指令加载器会按照解码得到的序列进行指令加载,即可以图 6 中左边的原始指令序列执行.

2.5 指令摘要表保护

在混淆过程中重编译阶段,指令摘要表独立于整个可执行程序保存,重编译器使用加密算法对指令摘要表

进行保护;在应用程序运行过程中预处理阶段,预处理器从可执行程序中提取出加密后的指令摘要表,通过解密得到解码阶段用到的指令摘要表.考虑到加密算法和密钥的安全性,本方案中使用白盒 AES 算法对指令摘要表进行保护.

AES 算法中,密钥需扩展成轮子密钥,扩展算法是确定且可逆的,并且轮子密钥直接出现在计算内存空间中,直接参与基本操作运算,难以抵抗白盒攻击.白盒 AES 算法由 Chow 等人[12]提出,Chow 白盒 AES 算法实现中通过如下 3 个阶段实现对密钥的隐藏:(1) 密钥信息嵌入到有限域计算中;(2) 构造表查找系统,将密码算法中的运算转换成表查找操作;(3) 利用随机化、非线性化操作和编码等手段,隐藏查找表.见算法实现 3.

- 标准 AES 算法(步骤(a)):Chow 白盒 AES 加密算法将 *AddRoundKey* 和下轮 *SubBytes* 组合为一个步骤 T,把 *ShiftRows* 移到 T 之前.在 *AddRoundKey* 结合到 9 轮循环时,用于异或的轮密钥为上轮轮密钥,需预先 *ShiftRows* 移位;
- 中间算法流程(步骤(b)):Chow 白盒算法中每轮 *AddRoundKey* 和下轮 *SubBytes* 都可以和输入矩阵 X_r 组成查找表 T^r.密钥确定的前提下,此过程可由查表方式替代,即为 *T-Box*,以此实现密钥和轮密钥的隐藏.*T-Box* 与 *MixColumns* 步骤结合,扩展成 *TY-Box*;
- 最终,Chow 白盒 AES 流程见步骤(c).

算法实现 **3**. 指令摘要表保护流程.

(a) 标准 AES 流程

state←*plaintext*
AddRoundKey(*state*,k_0)
FOR (r=1,…,9)
  SubBytes(*state*)
  ShiftRows(*state*)
  MixColumns(*state*)
  AddRoundKey(*state*,k_r)
SubBytes(*state*)
ShiftRows(*state*)
AddRoundKey(*state*,k_{10})
ciphertext←*state*

(b) Chow 白盒 AES 中间流程

state←*plaintext*
FOR (r=1,…,9)
  ShiftRows(*state*)
  AddRoundKey(*state*,k_{r-1})
  SubBytes(*state*)
  MixColumns(*state*)
ShiftRows(*state*)
AddRoundKey(*state*,k_9)
SubBytes(*state*)
AddRoundKey(*state*,k_{10})
ciphertext←*state*

(c) Chow 白盒 AES 流程

state←*plaintext*
FOR (r=1,…,9)

ShiftRows(*state*)

TY-Box(*state*)

ShiftRows(*state*)

T-Box(*state*,10)

ciphertext←*state*

为抵御已知明文密码导出密钥攻击,加入可逆矩阵 *MB*,增加查找表个数,实现 *TY-Box* 内部置乱.在生成查找表时,每个 *TY-Box* 查找表都需要事先乘以 *MB* 矩阵,在后续的运算中,通过乘以 MB^{-1} 来抵消运算结果.使用内部置乱,查找表和查找方法仍然暴露在白盒环境中,攻击者在不计算密钥的前提下可以直接查表解密数据,因此使用外部编码对查找表进行保护.外部编码首先对明文异或随机数列进行编码,该随机序列与第 1 轮和最末一轮查找表结合,编码后明文经第 1 轮查找将编码抵消,而在最末一轮查表时又添加了编码,因此,10 轮结束后得到的是编码后的密文,经解码得到真正的密文.

使用 Chow 白盒 AES 算法,AES 每一轮变换拆分成若干模块,对每个模块进行置乱编码的同时将密钥信息嵌入到模块操作中,实现将 AES 多轮运算转换成查表操作的目的;此外,在表查找系统中加入随机化和非线性化,利用随机双射对查找表进行编码,隐藏查找表内容,最终实现密钥和查找表的隐藏.

3 实验设计

基于上述混淆算法,我们研发了一套针对 Android Arm 架构的原型系统,系统部署环境是 Ubuntu 14.04 LTS.在本节,通过实验对本文的混淆算法进行评估.

首先,实验选择的应用全部来源于百度移动应用商店,这是国内目前使用人数最多的移动应用商店.在本实验中,通过爬虫技术,从百度应用商店中最热门的应用中随机选取了 200 个应用软件作为我们的实验对象.将 200 个测试软件输入到原型系统中,输出为混淆后的应用软件.

然后,使用混淆模块对输入应用软件进行混淆.反编译器基于 Apktool 工具二次开发,对 AndroidManifest.xml 文件进行解码,将 classes.dex 文件解析成 Dalvik 指令.预处理器将 Dalvik 指令按照函数作用范围切分,得到若干片段,对指令片段进行摘要,得到指令摘要表.混淆器根据得到的指令摘要表,使用第 2.3 节中的算法对指令序列进行混淆.提取出来的指令摘要表,经过加密处理后,独立于 Dex 可执行文件,在本系统中,加密后的指令摘要表放置于应用的 Assets 文件夹下.此外,加密后的指令摘要表也可以放置于 Dex 可执行程序的头部和数据部之间.重编译模块对混淆后的指令和运行时用到的指令加载器进行重编译打包.

最后,混淆后的应用软件安装在 Android 终端中,统计待测 200 个应用软件的运行情况.加密算法选择上,文本选择白盒 AES 加密算法,并与标准 AES 算法进行对比分析.

4 评估与分析

本文从安全强度、开销、平台适应性和隐蔽性这 4 个角度对本文混淆算法进行阐述和评估.

4.1 强度(**intensity**)

本文提出的混淆方法的安全强度取决于对指令摘要表的加密强度,指令摘要表独立于混淆后的可执行程序.尽管本文提出的混淆方法适用于移动平台,但指令摘要表本身体积很小,即使采用高强度的加密算法,也不会对整个系统的运行效率产生较大的影响.考虑到密钥或证书隐藏的问题,本文使用白盒 AES 算法对指令摘要表进行加密.

Billet 等人[14]提出了一种 BGE 攻击方法,选择特定查找表,合并成一个可以用输入输出表示的函数,使用代数的方法去掉其中的非线性部分,提取出隐藏在 *T-Box* 中的密钥.Michiels 等人[15]改进成通用攻击方法,实现类似算法的白盒攻击.尽管如此,在移动应用环境下,结合应用防反编译、防调试、校验等手段,从侧面降低分析环境的透明度,提升攻击者分析密钥的难度,可以在一定程度上有效地保护移动应用代码.

4.2 开销(cost)

混淆过程用到 LZW 和 Huffman 两种编码作为核心算法,一方面,虽然 LZW 不增加指令规模,但 Huffman 对指令参数进行混淆时会增加若干冗余指令,会占用额外空间,而且保存白盒 AES 加密后的指令摘要表和白盒密码表需要占用一定的空间;另一方面,保护后的指令摘要表需要通过动态方式进行白盒 AES 解密、指令加载器通过解码寻址后对指令进行加载,因此会有额外的空间开销.

本文对测试的 200 个应用混淆前后进行空间开销统计,如图 10 所示.

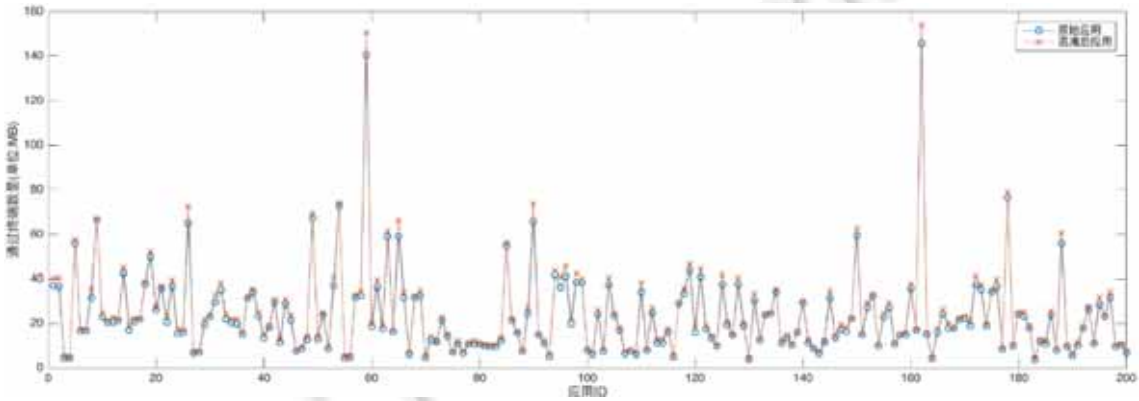


Fig.10 Comparison of space cost before and after obfuscation

图 10 混淆前后空间开销对比

从图 10 可以看出:本文提出的混淆方法前后,对于移动应用的空间开销较小,经过计算,空间开销不超过 8%.从第 2.4 节的图 8 可以看出:图 8 左侧混淆后的 6 条指令中包含 5 条冗余指令,存在很大空间资源的浪费.Android 中的 Dalvik 指令通常由 8 位机器码表示,当指令片段过短时,Huffman 编码表中记录的编码值很短,在混淆时带来极大的空间浪费,因此在混淆前做代码切片时,尽可能保证指令数量合理,即,Huffman 中编码表中的平均编码长度尽可能接近 8 位.本文中,混淆前预处理器是根据函数作用范围进行切分的,在本文后续工作中,提出了一种基于平均编码长度动态切分的改进算法.

本文对测试的 200 个应用混淆前后进行时间开销统计,如图 11 所示.

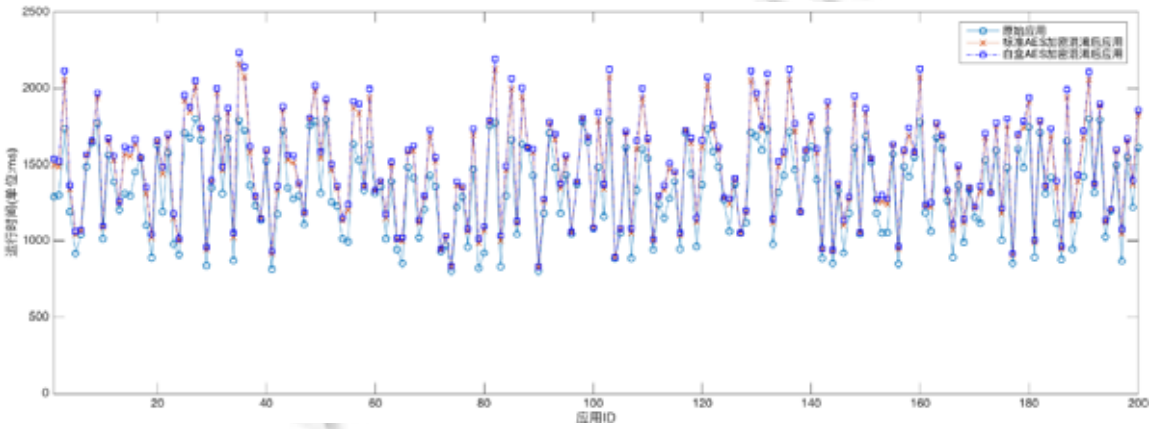


Fig.11 Comparison of timecost before and after obfuscation

图 11 混淆前后时间开销对比

本文将混淆前、标准 AES 加密编码表混淆和白盒 AES 加密编码表混淆的软件批量自动安装运行在测试终端中,对应用的启动时间进行统计.测试终端型号为 Google Nexus 5,Android 系统版本号为 6.0.1.从图 11 可见:本文提出的混淆方法虽然在空间开销上较低,但是在应用的启动加载时间上的开销较大.经统计,启动时间平均

开销约在 20%~25%之间,影响较大.此外,从图 11 可以看出:对于本文提出的方法,使用白盒 AES 加密编码表相比标准 AES 加密编码表,时间开销相差不大.由于应用在启动时会将指令进行全部加载,因此,本混淆方法只影响到应用启动时间,而对应用启动后的运行效率不会产生较大影响.

4.3 平台适应性(**compatibility**)

基于本文提出的混淆方法,我们研发出了 Android 平台的混淆原型工具.为了验证本方法混淆后的应用能够在 Android 智能终端中正常运行,我们进行了 Android 平台适应性评估.

测试样本是第 3 节中随机选取的 200 个样本,输入到原型工具后,得到 200 个混淆后的应用.使用百度移动云测试中心(MTC)为 200 个混淆后的应用进行适配性测试.在 MTC 中,上传 200 个原始应用和 200 个混淆应用,分别为每个应用选取最常用机型和系统中的随机 50 中组合,进行适配性测试.实验结果如图 12 所示.

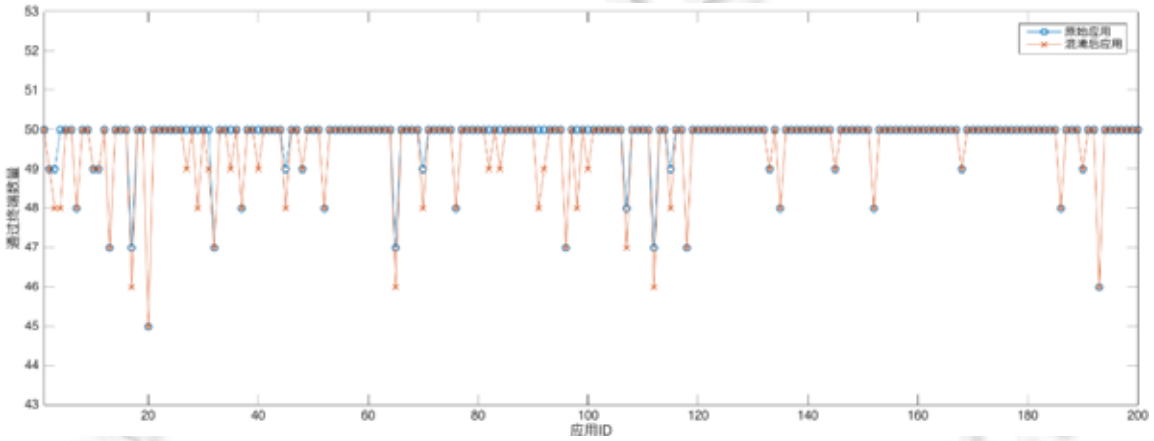


Fig.12 Compatibility of original apps and obfuscated apps

图 12 混淆前后应用的平台适应性

从图 12 可知:在待测的 200 个应用软件中,有 21 个应用软件在混淆后,适配终端数量下降.对无法适配的应用和终端情况进行统计,得到表 1.

Table 1 Device description of incompatibility

表 **1** 不适配终端系统型号描述

名称	不适配系统的描述	名称	不适配系统的描述
5	MIUI, Android 2.1;	72	MIUI, Android 2.1;
6	MIUI, Android 2.1, Android 2.2;	84	MIUI, Android 2.3.4;
19	MIUI, Android 2.1;	86	HUAWEI, Android 2.3.5;
29	MIUI, Android 2.1;	93	HUAWEI, Android 2.3.4;MIUI, Android 2.3.5;
31	MIUI, Android 2.1, Android 2.3.5;	94	HTC, Android 2.1;
33	SAMSUNG, Android 2.1;	100	HTC, Android 2.3.4;HUAWEI, Android 2.2;
37	MIUI, Android 2.3.4;	102	MIUI, Android 2.3.5;
42	HTC, Android 2.3.5;	109	SAMSUNG, Android 2.3.5;
47	HUAWEI, Android 2.3.5;	114	HTC, Android 2.2;
67	MIUI, Android 2.3.4;	117	MIUI, Android 2.3.5;

由表 1 所示:混淆后应用只有在特定型号终端的早期版本(Android 2.x)上出现了无法适配的问题,部分特定厂商的早期版本智能终端,使用的是厂商深度定制的 Android 系统,相比 GoogleAOSP 的标准系统来说,厂商对部分 API 进行了优化或禁用,导致指令加载器无法正常的运行.MTC 中测试终端系统版本覆盖 Android 2.x 至 Android 6.0 的系统,对于主流系统来说,通过原型工具混淆后的应用能够在主流终端中正常运行.

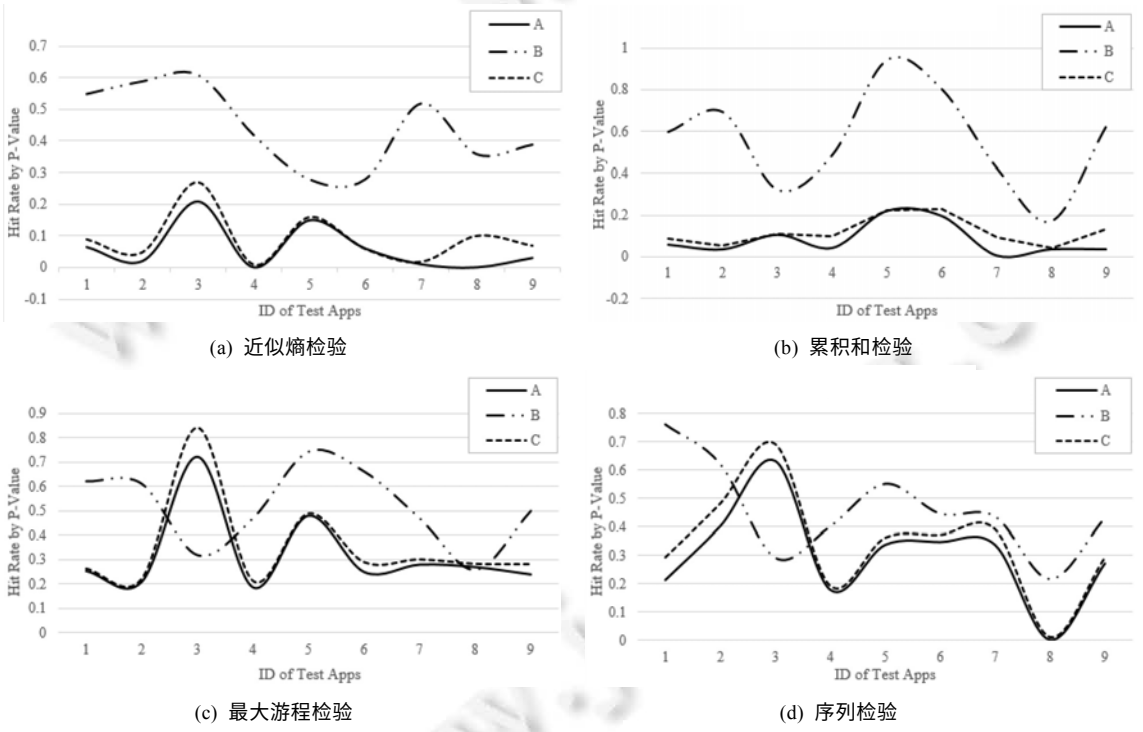
4.4 隐蔽性(**concealment**)

隐蔽性评估通常用做密码学中对加密算法好坏的评估.很多学者提出了基于统计特性来进行隐蔽性分析

的方法.Rukhin 等人在 2001 年[16]提出并在 2010 年[17]重新阐述了这种测试方法,证明了基于统计的方法与密码学的关系,此方法一般可以用来评估加密算法是否有效.美国国家标准技术研究所(National Institute of Standards and Technology,简称 NIST)一般用到一些常用的方法来对加密算法进行分析,例如频数检验(frequency test)、重叠模板匹配检验(overlapping template matching test)、非重叠模板匹配检验(non-overlapping template matching test)、线性复杂度检验(linear complexity test)、近似熵检验(approximate entropy test)、最大游程检验(longest run test)等.徐等人[10]提出一种基于随机性分析的数据加密功能的检验方法,用随机性分析方法判断数据块是否经过加密处理.如果混淆前后应用的统计特性发生了较大变化,说明混淆方法的隐蔽性较差.

在本节中,使用统计特性对隐蔽性进行分析.从第 3 节得到的 200 个应用中随机抽取 10 款应用,对混淆前和混淆后的应用统计特性进行分析.用 *A* 表示混淆前应用样本集合,用 *B* 表示使用梆梆加固(目前国内移动互联网加固行业知名服务公司)保护后的样本集合,用 *C* 表示用本文原型工具混淆后的样本集合.对上述 3 个集合,分别使用近似熵检验、累积和检验、最大游程检验和序列检验这 4 种方法对样本进行检测分析.

在本次检验中,我们使用 NIST 提供的公开工具,设定比特流长度为 102 400,近似熵检验中块长度为 10,序列检验中块长度为 16.其他主要参数为 NIST 公开工具默认参数.经过测试,统计得到如图 13 所示.



(a) 近似熵检验 (b) 累积和检验

(c) 最大游程检验 (d) 序列检验

Fig.13 Assessment by four statistical test

图 13 4 种统计检验的评估结果

从图 13 可以看出:使用梆梆进行应用加固后,无论是哪种统计检验,得到的 *P-Value* 值相比原始应用都出现较大变化,尤其是图 13(a)和图 13(b).梆梆加固采用移动应用软件的加壳技术,原始应用的可执行程序使用了分片加密和块加密的方式进行保护,因此,原本指令结构不再可读,程序随机性发生了很大变化.相比而言,使用本文提出方法混淆后,应用本身的随机性虽然有一定的变化,但总体而言变化并不大.本文所使用的混淆方法并没有对主要指令进行加密处理,而是通过编码算法将指令顺序进行更换,或以原有指令进行冗余扩展,因此,随机性并没有发生很大的变化,混淆方法本身具有一定的隐蔽性.从图 13 中可以看出:在图 13(c)和图 13(d)中,仍然存在一些不一致的点,经分析,NIST 工具把移动应用当做一个二进制文件,把数据进行固定切分,然后进行统计分

析.而这种切分难以保证按照可执行程序的语法结构进行切分,存在一定的不合理性.

5 总结与展望

本文提出了一种基于 Huffman 和 LZW 编码的针对移动应用的二进制混淆技术,这种混淆技术将混淆的开销和隐蔽性考虑在内,通过动态加载方式为提高安全保护强度提供新的解决思路.

从第 4 节评估分析可以看到,目前,本文混淆技术还需要一些优化和改进.

- 首先,开销仍然是一个需要优化的部分,虽然空间开销较低,但是目前以函数作用范围切分代码段的做法稍显武断,容易造成空间利用的不合理.在后续研究过程中,我们将会把 Huffman 编码的平均编码长度因素考虑在内,提出一种动态混淆技术;
- 另外,在隐蔽性评估中可以发现:目前业内通用的 NIST 测试标准,在移动应用随机性评估中,由于不考虑语法和语义,因此存在评估不合理的情况.将来可以提出一种基于语义的随机性检验方法;
- 此外,考虑到移动应用的数量和增速,对所有应用都进行安全保护,在计算资源有限的情况下,很显然是难以实现的,因此,作为移动应用的重要组成部分,组件重要性评估也是我们之后的一个重点研究方向之一,我们通过对移动应用内在关联的研究,研究应用组件重要性评估算法,对应用中的重要节点,进行有针对性的安全保护.

References:

[1] Collberg C, Thomborson C, Low D. A taxonomy of obfuscating transformations. Vol.148. New Zealand: Department of Computer Science, The University of Auckland, 1997. 1−32.

[2] Collberg C, Nagra J. Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Upper Saddle River: Addision Wesley Professional, 2009. 163−243.

[3] Collberg C, Thomborson C, Low D. Manufacturing cheap, resilient, and stealthy opaque constructs. In: Proc. of the 25th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. ACM Press, 1998. 184−196.

[4] Wang C. A security architecture for survivability mechanisms [Ph.D. Thesis]. Virginia: University of Virginia, 2001.

[5] Balachandran V, Emmanuel S. Software protection with obfuscation and encryption. In: Proc. of the Int'l Conf. on Information Security Practice and Experience. Springer-Verlag, 2013. 309−320.

[6] Cheng L. Research on techniques security mobile code based on homomorphic encryption [Ph.D. Thesis]. Guangzhou: South China University of Technology, 2009 (in Chinese with English abstract).

[7] Sander T, Tschudin CF. Protecting mobile agents against malicious hosts. LNCS on Mobile Agents and Security, 1998,1419:44−60.

[8] Sander T, Tschudin CF. Towards mobile cryptography. In: Proc. of the 1998 IEEE Symp. on Security and Privacy. IEEE, 1998. 215−224.

[9] Cheng C, Zhang F. Obfuscationfor multi-user encryptionandits application in cloud computing. Concurrency and Computation: Practice and Experience, 2015,27(8):2170−2190.

[10] Xu GA, Zhang M, Ma JL, Guo CQ, Guo YH. A test method based on the random analysis of data encryption [Patent Documentation]. H04L1/00I, 101888282A, 2010 (in Chinese).

[11] He YX, Chen Y, Wu W, Chen N, Xu C, Liu JB, Su W. A program flow-sensitive self-modifying code obfuscation method. Computer Engineering and Science, 2012,34(1):79−85 (in Chinese with English abstract).

[12] Chow S, Eisen P, Johnson H, van Oorschot PC, Oorschot V. White-Box cryptography and an AES implementation. LNCS, 2003,2595: 250−270.

[13] Wu Z, Gianvecchio S, Xie M, Wang H. Mimimorphism: A new approach to binary code obfuscation. In: Proc. of the 17th ACM Conf. on Computer and Communications Security. ACM Press, 2010. 536−546.

[14] Billet O, Gilbert H, Ech-Chatbi C. Cryptanalysis of a white box AES implementation. In: Proc. of the Selected Areas in Cryptography. Berlin, Heidelberg: Springer-Verlag, 2005. 227−240.

[15] Michiels W, Gorissen P, Hollmann HDL. Cryptanalysis of a generic class of white-box implementations. In: Proc. of the Selected Areas in Cryptography. Berlin, Heidelberg: Springer-Verlag, 2009. 414−428.

[16] Rukhin A, Soto J, Nechvatal J, Smid M, Barker E, Leigh S, Levenson M, Vangel M, Banks D, Heckert A, Dray J, Vo S. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Booz-Allen and Hamilton Inc Mclean Va, 2001.

[17] Rukhin A, Soto J, Nechvatal J, Smid M, Barker E, Leigh S, Levenson M, Vangel M, Banks D, Heckert A, Dray J, Vo S. Statistical test suite for random and pseudorandom number generators for cryptographic applications. NIST special publication, Vol.800(22): NIST, Technology Administration, U.S. Department of Commerce, 2010.

附中文参考文献:

[6] 陈良.基于同态加密的移动代码安全技术研究[博士学位论文].广州:华南理工大学,2009.

[10] 徐国爱,张淼,马健丽,郭承青,郭燕慧.一种基于随机性分析的数据加密功能的检验方法[专利文献].H04L1/00I,101888282A,2010.

[11] 何炎祥,陈勇,吴伟,陈念,徐超,刘健博,苏雯.基于程序流敏感的自修改代码混淆方法.计算机工程与科学,2012,34(1):79−85.

李承泽(1989－),男,山东威海人,博士,主要研究领域为移动应用软件逆向,安全分析和安全加固,网络攻防对抗,系统安全测评.

徐国爱(1972－),男,博士,教授,博士生导师,主要研究领域为密码学,网络安全,信息安全,应用软件安全.

於剑波(1993－),男,硕士,主要研究领域为移动应用逆向,安全分析,安全加固.

孔浩浩(1992－),女,硕士,主要研究领域为移动应用逆向,安全分析.

张淼(1980－),男,博士,副教授,主要研究领域为移动应用安全分析和加固,网络攻防对抗,信息安全舆情分析.