**摘 要**

本文主要论述了一种基于C/S架构的教务管理系统的设计与实现。

该系统采用Java技术、以MySQL为后台数据库，以MyEclipse为开发工具，设计主要实现了教师，学生，管理员对系统的需求，能满足教师，学生和管理员同时登录系统。通过Java与MySQL的良好结合，实现了管理员管理班级、课程、教师、学生及课程编排等功能，也实现了学生，教师登录查询及成绩管理等功能，提高了系统的安全性和稳定性。系统运用JDBC连接数据库，对数据库表均实现了增、删、改、查功能。设计中大量运用面向对象的编程思想，整个系统按照MVC的开发思想，多处用到直接操作对象的方法，大大提高了系统的运行效率。不过，系统存在一定缺陷，比如系统结构太单一、功能过于简单等，还有待继续完善。

本论文主要按照软件工程的基本原理，阐述了对系统的可行性分析和需求分析，叙述了系统的总体设计和详细设计，详细论述了系统的实现过程，描述了系统进行的各项测试并顺利通过。

关键字：Java；JDBC；MySQL；C/S模式

**ABSTRACT**

This paper discusses a novel of C/S structure of the Educational Management System Design and Implementation.

The system uses Java technology to MySQL as the backend database to MyEcslipe for the development tools, mainly designed for teachers, students and administrators. The teachers, students and administrators can login the system at a time. Through a good combination of Java and MySQL to achieve the functions of the administrator, such as managing classes, courses, teachers, students and curriculum designing and so on. The system also realizes the querying function and performance management for the students and teachers. The combination also improves security and stability of the system. By JDBC, the database of the system are implemented by functions of delete, change, search. Extensive use of programming ideas including object-oriented, MVC and Methods to manipulate objects directly, greatly improve the efficiency of the system. However, the system has some defects too. For example, System Structure is too onefold, function too simple, etc, still remains to perfect.

According to the basic principles of software engineering, this thesis expounds the feasibility analysis of the system and demand analysis, describes the overall system design and detailed design, discusses implementation of the system in detail, and the test of the system which has passed smoothly.

**Keywords:** Java; JDBC; MySQL; C/S mode

目录

[第一章 绪论 - 1 -](#_Toc450746148)

[1.1 选题背景 - 1 -](#_Toc450746149)

[1.1.1 手机市场分析 - 1 -](#_Toc450746150)

[1.1.2 Android数据安全 - 1 -](#_Toc450746151)

[1.1.3 Android应用盗版情况 - 2 -](#_Toc450746152)

[1.2 选题意义 - 2 -](#_Toc450746153)

[1.3 论文组织结构 - 2 -](#_Toc450746154)

[第二章 第二章 相关基础知识 - 3 -](#_Toc450746155)

[2.1加密算法的选择 - 3 -](#_Toc450746156)

[2.2 Dalvik虚拟机基础知识 - 3 -](#_Toc450746157)

[第三章 第三章 文件加密与APP加固需求分析 - 12 -](#_Toc450746158)

[1.4 2.1 - 12 -](#_Toc450746159)

[1.5 2.2 App加固 - 12 -](#_Toc450746160)

[第四章 - 13 -](#_Toc450746161)

[第五章 第四章 文件加密系统系统设计 - 13 -](#_Toc450746162)

[1.6 4.1 密钥存储方案 - 13 -](#_Toc450746163)

[第六章 第五章 APP加固方案设计 - 13 -](#_Toc450746164)

[第七章 第六章 文件加密及APP加固详细设计 - 13 -](#_Toc450746165)

[第八章 第七章 测试 - 13 -](#_Toc450746166)

[第九章 第八章 总结与展望 - 13 -](#_Toc450746167)

[第十章 致谢 - 13 -](#_Toc450746168)

[第十一章 参考文献 - 13 -](#_Toc450746169)

# 绪论

## 选题背景

### 手机市场分析

Android自2012年2.2Froyo系统发布以来，一直占据着中国智能手机操作系统市场的大半壁江山，始终是唯一能与其相提并论的只有Apple公司的IOS系统，而根据国外市场分析公司Kantar的数据显示，2015年Q1和2016年Q1 Android系统市场占有率稳步提升3.4%，达到了76.4%。

从未来发展方向上看，Android系统由于其开放性，有很广阔的发展空间，比如车载系统、智能家居系统等等。

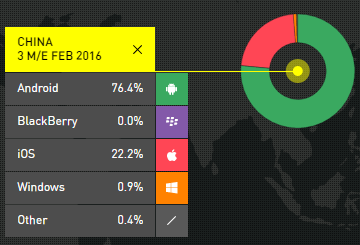
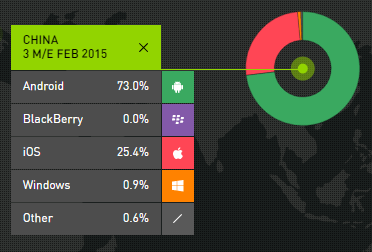


图1.1 2015及2016智能手机系统市场占有率

### Android用户数据安全

Android的安全问题一直饱受诟病，随着Android设备应用越来越广泛，安全问题也更应该引起重视。安全设计其实是贯穿了Android系统架构的各个层面的，覆盖系统内核、虚拟机、应用程序框架层以及应用层各个环节，力求在开放的同时，也恰当保护用户的数据、应用程序和设备的安全。Android安全模型主要提供以下几种安全机制：

1. 进程沙箱隔离机制
2. 应用程序签名机制
3. 权限声明机制
4. 访问控制机制
5. 进程通信机制
6. 内存管理机制

而对于用户数据的的安全性，Android主要是由访问控制机制来保护的，这很明显是远远不够的，针对Meizu MX3进行测试，Meizu MX3自带的文件浏览器中包含有锁定区，访问需要密码，但其具体位置在/data/media/0/.@meizu\_protbox目录下，利用具有root权限的文件浏览器直接访问这个目录，就可以绕过密码查看到锁定区中的所有文件，可以看到，这里只有访问控制，而文件本身是明文存储在文件系统中的，由于root权限很可能由恶意软件通过系统漏洞取得，所以这种做法很不安全。

### Android应用盗版情况

盗版软件一直是影响移动互联网健康发展的毒瘤，不但对用户隐私、资费等造成威胁，更严重影响了正版APP的品牌形象和推广效果，侵害了正版APP开发者的权益。而由于国内Android应用市场众多，各应用市场的审核机制和管理规范良莠不齐，致使很多用户无意中下载了被植入恶意程序的盗版应用。用户一旦安装这类应用，轻则会被垃圾信息和广告骚扰，重则会导致隐私信息泄露、恶意扣费、流量损失等危害。

据360公司2015年Android手机应用盗版情况调研报告显示，2015年Android手机应用盗版情况非常猖獗，平均每款正版APP对应92.7个盗版。其中，在软件类盗版APP中，系统工具类APP的平均盗版数量最多，平均每款正版APP对应324.2个盗版。

游戏类盗版APP中，模拟辅助类APP的平均盗版数量最多，平均每款正版游戏对应盗版228.6个。

软件类盗版APP中，便捷生活类盗版APP感染的设备数量最多，为237,878个；系统安全类盗版APP感染的设备数量为138,037；影音视听类盗版APP感染的设备数量为116,856个。

游戏类盗版APP中，经营策略类盗版APP感染的设备数量最多，为146,103个；休闲益智类盗版APP感染的设备数量为137,003；跑酷竞速类盗版APP感染的设备数量为83,650个。

## 选题意义

Android文件加密系统将在Android自有的访问控制基础之上，将需要加密存储的文件通过加密算法生成密文存储在硬盘中，进一步加强用户数据的安全性。并且通过APP加固手段保护源码，提高破解本程序的技术门槛，并且可以沿用到其他APP中，保护正版APP不被反编译。

## 论文组织结构

全文由七部分组成：

第一章阐述当前在Android设备占领绝大部分市场的情况下，系统对用户数据安全保障的情况以及Android应用加固保护源码的必要性。

第二章介绍了Android开发、APP运行的一些基础知识以及开发Android APP加固程序的一些关键技术。

第三章进行了对开发Android文件加密系统及APP加固程序的可行性分析，并且对该程序进行详尽的需求分析。

第四章根据第三章的需求分析，对系统从功能和架构上进行设计，为第五章的详细设计指明道路。

第五章对系统功能的实现从各个模块的代码编写角度进行详细的介绍。

第六章对系统进行全方位的测试，保证系统的可靠性及健壮性。

第七章总结了系统开发过程中遇到的问题，并且对系统的下一步发展方向进行了展望。

# 相关基础知识

本章涉及到系统中主要用到的一些关键技术的介绍，包括文件加密过程中加密算法的原理

## 2.1 开发环境

IDE: Android Studio。本系统使用Google公司提供的官方IDE Android Studio进行开发，Android Studio是打造高品质，高性能Android应用程序的最快方法。作为谷歌官方的IDE，Android Studio包含建立一个APP的所有工具，包括代码编辑器，代码分析工具，内存分析工具，模拟器等等。

打包工具：Gradle。Gradle是一个基于Apache Ant和Apache Maven概念的项目自动化建构工具。它使用一种基于Groovy的特定领域语言来声明项目设置，而不是传统的XML。当前其支持的语言限于Java、Groovy和Scala。

开发语言：Java。面向对象的Java语言一直有着不可比拟的优势，跟动态语言比起来，Java的执行效率以及安全性都无可比拟，并且结构规范，方便调试。而跟其他的静态语言相比，Java丰富的类库为它带来了跟多的优势。

测试设备：运行在Windows10系统上的Nexus 6 虚拟机，API level 22，以及Meizu MX3真机。

## 2.2 加密相关

### 2.2.1 加密算法的选择

加密算法根据密钥类型不同一般分为两类：对称加密算法和非对称加密算法。首先要决定使用对称还是非对称加密算法。

在加密算法的选择中，主要有以下区别：

1. 非对称加密算法的运行速度比对称加密算法要慢很多，当需要加密大量数据时，建议采用对称加密算法。
2. 对称加密算法不能实现签名，因此签名只能非对称算法。
3. 由于对称加密算法的密钥管理是一个复杂的过程，密钥的管理直接决定着他的安全性，因此当数据量很小时，可以考虑采用非对称加密算法。
4. 非对称加密算法基于未解决的数学难题，在破解上几乎不可能。对于对称加密算法密码算法，到了AES虽说从理论来说是不可能破解的，但从计算机的发展角度来看。公钥更具有优越性。
5. 选定了加密算法后，那采用多少位的密钥呢？一般来说，密钥越长，运行的速度就越慢，应该根据的我们实际需要的安全级别来选择，一般来说，RSA建议采用1024位的数字，ECC建议采用160位，AES采用128为即可。

由于我们的文件加密系统是在本地加密并不需要在网络中通信，而且可能会加密视频、音频等比较大的文件，而在加密数据时非对称加密的效率没有对称加密高，所以本系统需要采用对称加密算法。为了保障密钥的安全，再使用非对称加密算法加密密钥。

目前常用的对称加密算法主要有三种：

1. DES（Data Encryption Standard）:

数据加密标准，速度较快，适用于加密大量数据的场合。

1. 3DES (Triple DES):

是基于DES，对一块数据用三个不同的密钥进行三次加密，强度更高。

1. AES（Advanced Encryption Standard）：

高级加密标准，新一代的加密算法标准，速度快，安全级别高。

由于DES是1997年公布实施的，现在来说已经有些落后，3DES和AES的安全性都比其高很多，所以要在2DES和AES中选择一个，两者对比如下：

表2.1 AES与3DES加密算法的比较

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 算法名称 | 算法类型 | 密钥长度 | 速度 | 破解时间 | 资源消耗 |
| AES | 对称Block密码 | 128、192、256位 | 高 | 1490000亿年 | 低 |
| 3DES | 对称feistel密码 | 112或168位 | 低 | 46亿年 | 中 |

通过两者的对比，根据文件加密系统的实际情况，选择AES加密算法，在密钥长度的选择中，虽然通常长度越大的密钥破解难度越大，但是加解密效率也越低，而且随着计算机的发展，密钥长度对破解难度的影响并不是很大，所以采用128位密钥即可。

### 2.2.2 AES原理

AES加密过程是在一个4×4的[字节](https://zh.wikipedia.org/wiki/%E5%AD%97%E8%8A%82" \o "字节)矩阵上运作，这个矩阵又称为“体（state）”，其初值就是一个明文区块（矩阵中一个元素大小就是明文区块中的一个Byte）。（Rijndael加密法因支持更大的区块，其矩阵行数可视情况增加）加密时，各轮AES加密循环（除最后一轮外）均包含4个步骤：

1. AddRoundKey—矩阵中的每一个字节都与该次[回合密钥](https://zh.wikipedia.org/w/index.php?title=%E5%9B%9E%E5%90%88%E9%87%91%E9%91%B0&action=edit&redlink=1" \o "回合密钥（页面不存在）)（round key）做[XOR运算](https://zh.wikipedia.org/wiki/XOR" \o "XOR)；每个子密钥由密钥生成方案产生。
2. SubBytes—通过一个非线性的替换函数，用[查找表](https://zh.wikipedia.org/wiki/%E6%9F%A5%E6%89%BE%E8%A1%A8" \o "查找表)的方式把每个字节替换成对应的字节。
3. ShiftRows—将矩阵中的每个横列进行循环式移位。
4. MixColumns—为了充分混合矩阵中各个直行的操作。这个步骤使用线性转换来混合每内联的四个字节。最后一个加密循环中省略MixColumns步骤，而以另一个AddRoundKey替换。

表2.2 AES密钥长度与轮数的关系

|  |  |  |  |
| --- | --- | --- | --- |
| 密钥长度 | 128 | 192 | 256 |
| 圈数 | 10 | 12 | 14 |

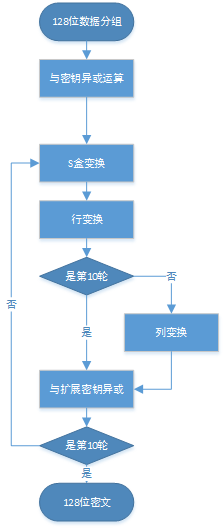


图 2.1 AES算法流程图

各个步骤的实现方式：

1. S盒变换（SubBytes）

这个步骤

1. 行变换 （ShiftRows）
2. 列变换 （MixColumns）
3. 扩展密钥（AddRoundKey）

## 2.3 Android Dalvik虚拟机

Dalvik虚拟机是Android设备平台的核心组成部分之一，可以说就是Google将Java虚拟机为了适合移动设备内存小、CPU速度慢而进行的优化。其指令集基于寄存器架构，执行其特有的文件格式——dex字节码来完成对象生命周期管理、堆栈管理、线程管理、安全异常管理、垃圾回收等重要功能。它的核心内容是实现库（libdvm.so），大体由C语言实现。依赖于Linux内核的一部分功能——线程机制、内存管理机制，能高效使用内存，并在低速CPU上表现出的高性能。每一个Android应用在底层都会对应一个独立的Dalvik虚拟机实例，其代码在虚拟机的解释下得以执行。

要实现APP加固程序，必须要对Dalvik虚拟机有一定的了解，包括其可执行文件dex文件的结构，以及Android APP动态加载的原理。

### 2.3.1 dex、odex文件结构

# class文件结构

ClassFile{

//唯一取值：0xCAFEBABE

u4 magic;

//class文件的版本号，和Java编译额器有关

u2 minor\_version;

u2 major\_version;

//常量池，长度为字符串数量加1，constant\_pool[0]留给JVM用

u2 constant\_pool\_count;

cp\_info constant\_pool[constant\_pool\_count - 1];

//class的类型和名字，类型有三种 0x0001:ACC\_PUBLIC

//0x0010:ACC\_FINAL 0x0200:ACC\_INTERFACE

u2 access\_flag;

u2 this\_class;

u2 super\_class;

//类interface数量，变量数量，方法数量(无论static还是非static)，

//属性数量，名字都作为字符串存在常良池中

u2 interfaces\_count;

u2 interfaces[interfaces\_count - 1];

u2 fields\_count;

field\_info fields[fields\_count - 1];

u2 methods\_count;

method\_info methods[methods\_count - 1];

u2 attributes\_count;

attribute\_info attributes[attributes\_count - 1];

}

```

# dex文件

class文件显然有很多可以优化的地方，比如每一个class文件都有一个常量池，如果有重复字符串就造成了资源浪费，所以Dalvik的dex文件对其进行了优化

![dex文件结构](http://upload-images.jianshu.io/upload\_images/1448134-d858b2d9c09032ee.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240)

先看看dex文件中的数据结构

> 下面很多代码定义在Android源码的 [DexFile.h](http://osxr.org:8080/android/source/dalvik/libdex/DexFile.h#0245) 中

|类型|含义|

|-----|-----|

|u1 | 等同于uint8\_t,一个字节的无符号数 |

|u2 | 等同于uint16\_t,两个字节的无符号数 |

|u4 | 等同于uint32\_t,四个字节的无符号数 |

|u8 | 等同于uint64\_t,八字节的无符号数 |

|sleb128 | 有符号LEB128，可变长度1~5字节 |

|uleb128 | 无符号LEB128,可变长度1~5字节 |

|uleb128p1 | 无符号LEB128值加1，可变长度1~5字节|

\* sleb128是dex文件中特有的数据类型，每个字节7个有效位，最高位取值1表示要用到第二个字节，以此类推但最长五个字节，如果读取到

第五个字节最高位仍为1，表示该dex文件无效，Dalvik虚拟机在验证dex时会失败返回

\* dex文件里采用了变长方式表示字符串长度。一个字符串的长度可能是一个字节（小于256）或者4个字节（1G大小以上）。字符串的长度大多数都是小于 256个字节，因此需要使用一种编码，既可以表示一个字节的长度，也可以表示4个字节的长度，并且1个字节的长度占绝大多数。能满足这种表示的编码方式有 很多，但dex文件里采用的是uleb128方式。leb128编码是一种变长编码，每个字节采用７位来表达原来的数据，最高位用来表示是否有后继字节。

查看dex方法

1. class转为dex文件，工具是sdk build\_tools下的dx命令。dx --dex --debug --verbose-dump--output=test.dex com/test/TestMain.class

2. 查看dex文件，利用build-tools 下的dexdump 命令查看，dexdump -d -l plain test.dex

## dex文件整体结构

整体结构比较简单，由七个结构体组成：

\* dex header 指定了dex文件的一些属性，并记录其他六个部分在dex文件中的物理偏移

\* string\_ids

\* type\_ids

\* proto\_ids

\* field\_ids

\* method\_ids

\* class\_def

\* data

\* link\_data

dexHeader结构体的组成

```c

struct DexHeader {

000 u1 magic[8]; //dex版本标识

u4 checksum; //adler32检验

u1 signature[KSHA1DIGESTLEN]; //SHA-1哈希值 长度为20，定义在DexFile.h中

020 u4 fileSize; //整个文件大小

u4 headerSize; //DexHeader结构大小 70 00 00 00

u4 endianTag; //字节序标记 预设78 56 34 12 即0x12345678，表示小端little-Endian字节序

u4 linkSize; //链接段大小

030 u4 linkoff; //连接段偏移

u4 mapoff; //DexMapList的文件偏移，这里mapoff等于dataOff

u4 stringIdsSize; //DexStringId的个数

u4 stringIDsOff; //DexStringId的文件偏移

040 u4 typeIdsSize; //DexTypeID的个数

u4 typeIdsOff; //DexTypeId的文件偏移

u4 protoIdsSize; //DexProtoId的个数

u4 protoIdsOff; //DexProtoId的文件偏移

050 u4 fieldIdsSize; //DexFieldId的个数

u4 fieldIdsOff; //DexFieldId的文件偏移

u4 methodIdsSize; //DexMethodId的个数

u4 methodIdsOff; //DexMethonId的文件偏移

060 u4 classDefsSize; //DexClassDef的个数

u4 classDefsOff; //DexClassDef的文件偏移

u4 dataSize; //数据段的大小

u4 dataOff; //数据段的文件偏移

}

```

> tips:

由上面结构体也可以看出来，Android 65K方法数问题的根本原因并不在于Dex文件方法索引长度限制

## dex文件结构分析

> tips:

这里 书中（Android软件安全与逆向分析）有一点不明白，说Dalvik虚拟机解析dex文件的内容，最终将其映射成DexMapList数据结构

，是说Dex文件生成过程中有Dalvik虚拟机的参与吗。

我分析了一个简单的Android程序，使用十六进制编辑器C32Asm，打开apk解压出的dex文件

![dex文件](http://upload-images.jianshu.io/upload\_images/1448134-162fdde8a7ece6da.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240)

上图就是完整DexHeader的数据，在注释里写得很清楚了，观察发现，mapOff值为0x00059178,这里要注意小端字节序，找到

![DexMapList](http://upload-images.jianshu.io/upload\_images/1448134-0e79187ed36283d2.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240)

红色框画出来的就是每个元素头部，其中第一个0x12，代表有16个DexMapItem结构

DexMapItem结构：

```c

struct DexMapItem{

u2 type; //类型，枚举常量

u2 unused; //未使用，用于字节对其

u4 size; //指定类型的个数

u4 offset; //指定类型数据的文件偏移

}

//type 的枚举类型

/\* map item type codes \*/

enum {

kDexTypeHeaderItem = 0x0000,

kDexTypeStringIdItem = 0x0001,

kDexTypeTypeIdItem = 0x0002,

kDexTypeProtoIdItem = 0x0003,

kDexTypeFieldIdItem = 0x0004,

kDexTypeMethodIdItem = 0x0005,

kDexTypeClassDefItem = 0x0006,

kDexTypeMapList = 0x1000,

kDexTypeTypeList = 0x1001,

kDexTypeAnnotationSetRefList = 0x1002,

kDexTypeAnnotationSetItem = 0x1003,

kDexTypeClassDataItem = 0x2000,

kDexTypeCodeItem = 0x2001,

kDexTypeStringDataItem = 0x2002,

kDexTypeDebugInfoItem = 0x2003,

kDexTypeAnnotationItem = 0x2004,

kDexTypeEncodedArrayItem = 0x2005,

kDexTypeAnnotationsDirectoryItem = 0x2006,

};

```

举个香甜的栗子，找到DexMapList中的StringIdItem，个数：0x39EE,偏移：0x0070，去找0x0070中的第一个

![0x0070](http://upload-images.jianshu.io/upload\_images/1448134-c4fa0873dbeb2fc8.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240)

看一看DexStringId的结构体：

``` c

struct DexStringId{

u4 stringDataOff; // 字符串数据偏移

}

```

偏移量是0x0012c120,找到它

![0x0012c120](http://upload-images.jianshu.io/upload\_images/1448134-ddd7c91f4b180d5a.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240)

已经找到字符串了

再找一个复杂些的，找到DexMapList中的MethodIdItem，个数0x000038DD,偏移0x00026E98,找到它

![0x00026E98](http://upload-images.jianshu.io/upload\_images/1448134-6bd08b4ef488361c.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240)

看一看DexMethodId的结构体：

```c

/\*

\* Direct-mapped "method\_id\_item".

\*/

struct DexMethodId {

u2 classIdx; /\* index into typeIds list for defining class \*/

u2 protoIdx; /\* index into protoIds for method prototype \*/

u4 nameIdx; /\* index into stringIds for method name \*/

};

```

# odex文件

odex文件有两种存在方式：

1. 从Apk文件中提取出来，与Apk文件存放在同一目录下且文件后缀为odex的文件，这种多是Android ROM的系统程序；

2. dalvik-cache缓存文件，这类odex文件仍然以dex作为后缀，存放在cache/dalvik-cache目录下，保存形式为"apk路径@apk名@classes.dex";

由于Android程序的Apk文件为Zip压缩包格式，Dalvik虚拟机每次加载他们时需要从Apk中读取classes.dex文件，这样会耗费很多CPU时间，而采用odex

方式优化的dex文件已经包含了加载dex必须的依赖库文件列表，Dalvik虚拟机只需检测并加载所需的依赖库即可执行相应的dex文件，这大大缩短了读取dex文件

所需的时间。

## odex文件整体结构

\* odex文件头

\* dex文件

\* 依赖库

\* 辅助数据

odex文件的写入和读取并没有像dex文件那样定义了全系列的数据结构，Dalvik虚拟机将dex文件映射到内存中后是DexFile格式，结构如下：

```c

/\*

\* Structure representing a DEX file.

\*

\* Code should regard DexFile as opaque, using the API calls provided here

\* to access specific structures.

\*/

struct DexFile {

/\* directly-mapped "opt" header \*/

const DexOptHeader\* pOptHeader;

/\* pointers to directly-mapped structs and arrays in base DEX \*/

const DexHeader\* pHeader;

const DexStringId\* pStringIds;

const DexTypeId\* pTypeIds;

const DexFieldId\* pFieldIds;

const DexMethodId\* pMethodIds;

const DexProtoId\* pProtoIds;

const DexClassDef\* pClassDefs;

const DexLink\* pLinkData;

/\*

\* These are mapped out of the "auxillary" section, and may not be

\* included in the file.

\*/

const DexClassLookup\* pClassLookup;

const void\* pRegisterMapPool; // RegisterMapClassPool

/\* points to start of DEX file data \*/

const u1\* baseAddr;

/\* track memory overhead for auxillary structures \*/

int overhead;

/\* additional app-specific data structures associated with the DEX \*/

//void\* auxData;

};

```

最前面的DexOptHeader就是odex的头，DexLink一下的部分是"auxillary section",即辅助数据段，记录了文件被优化后添加的一些信息。不过DexFile

机构描述的是加载金内存的数据结构，还有一些数据是不会加载进内存的。丰生强老师将odex文件结构定义整理如下：

```c

struct ODEXFile {

DexOptHeader header; //odex文件头

DexFile DexFile; //dex文件

Dependences deps; //依赖库列表

ChunkDexClassLookup lookup; //类查询结构

ChunkRegisterMapPool mapPool; //映射池

ChunkEnd end; //结束标识

}

```

## odex文件结构分析

ODEXFile的文件头DexOptHeader在DexFile.h文件中定义如下:

```c

struct DexOptHeader{

u1 magic[8]; //odex版本标识 ，目前固定值 64 65 79 0A 30 33 36 00

u4 dexOffset; //dex文件头偏移 ，目前0x28 = 40，等于odex文件头大小

u4 dexLength; //dex文件总长度

u4 depsOffset; //odex依赖库列表偏移

u4 depsLength; //依赖库列表总长度

u4 optOffset; //辅助数据偏移

u4 optLength; //辅助数据总长度

u4 flags; //标志，Dalvik虚拟机加载odex时的优化与验证选项

u4 checksum; //依赖库与辅助数据的校验和

}

```

# 文件加密及APP加固需求分析

## 2.1

## 2.2 App加固

Android平台以其免费和开源的特性使其占据了移动领域大半壁江山，越来越的人投身到移动应用开发中，但本文恐怕要先给Android应用开发者泼盆冷水，据360发布的《2014年中国手机安全状况报告》显示，2014全年，360互联网安全中心累计监测到Android用户感染恶意程序3.19亿人次，平均每天恶意程序感染量达到了87.5万人次。同时，Android应用被破解和盗版等事件也层出不穷。

很明显，Android平台已经成为恶意程序和破解者攻击的众矢之的，于是越来越多的Android开发者开始意识到应用安全的重要性。

# 文件加密及APP加固系统设计

## 4.1 密钥存储方案

# 文件加密及APP加固详细设计

# 测试

# 总结与展望

# 致谢

# 参考文献

[1]