# 引言

## 研究背景

## 研究现状

## 研究内容

## 研究意义

## 论文组织内容

# 文件加密与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。

# 第三章 预备知识

## 3.1 加密算法的选择

## 3.2 Dalvik虚拟机基础知识

# class文件结构

```c

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];

}

```

class文件实操，先写一个简单的Java文件：

```java

package com.example;

public class MyClass {

public static void main(String[] args){

String s = "hello world";

System.out.println(s);

}

}

```

然后调用javac MyClass.java生成MyClass.class

调用javap -verbose MyClass.class查看

![class文件结构.jpg](http://upload-images.jianshu.io/upload\_images/1448134-209193ced54381b7.jpg?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240)

## 常量池

> tips：constant\_pool\_count值为常量池数组长度+1，就像上图中常量第一个元素

以#1开头，0默认是给VM用的

常量池的元素类型这样表示：

```c

cp\_info {

//特别注意，这是介绍的cp\_info 是相关元素类型的通用表达。

u1 tag; //tag 为1 个字节长。不论cp\_info 具体是哪种，第一个字节一定代表tag

u1 info[]; //其他信息，长度随tag 不同而不同

}

```

tag的取值：

\* tag=7 <==info 代表这个cp\_info 是CONSTANT\_Class\_info 结构体

\* tag=9 <==info 代表CONSTANT\_Fieldrefs\_info 结构体

\* tag=10 <==info 代表CONSTANT\_Methodrefs\_info 结构体

\* tag=8 <==info 代表CONSTANT\_String\_info 结构体

\* tag=1 <==info 代表CONSTANT\_Utf8\_info 结构体

看几个例子，字符串结构体：

```c

CONSTANT\_Utf8\_info {

u1 tag; //取值为1

u2 length; //下面就是存储UTF8 字符串的地方了

u1 bytes[length];

}

```

类信息结构体

```c

CONSTANT\_Class\_info {

u1 tag; //tag 取值为7，代表CONSTANT\_Class\_info

u2 name\_index; //name\_index 表示代表自己类名的字符串信息位于于常量池数组中哪一个，也就是索引

}

```

# 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

}

```

# 第四章 文件加密系统系统设计

## 4.1 密钥存储方案

# 第五章 APP加固方案设计

# 第六章 文件加密及APP加固详细设计

# 第七章 测试

# 第八章 总结与展望

# 致谢

# 参考文献