doi 10.3969/j.issn.1671-1122.2013.01.008

一种 Android 恶意程序检测工具的实现

张文[1]，严寒冰[2]，文伟平[1]

（1. 北京大学 软件与微电子学院信息安全系，北京 102600；
2. 国家计算机网络应急技术处理协调中心，北京 100029）

摘 要：目前，Android 上恶意程序的识别主要通过静态检测，但普遍识别率不高。文章基于静态检测原理，使用了一种基于行为的检测方法，以变量跟踪以及函数等价匹配的方式来判断一个 Android 安装包中是否存在恶意行为，从而增大了静态检测的准确率。在文章中，以短信吸费程序为样本，实现了这种基于行为分析的恶意程序检测工具。并在测试中证明了它的有效性。

关键词：Android；恶意程序检测；行为分析

中图分类号：TP393.08 文献标识码：A 文章编号：1671-1122（2013）01-0027-06

Implementation of a Malware Detect Tool on Android

ZHANG Wen[1], YAN Han-bing[2],WEN Wei-ping[1]

（1. *Department of Information Security, SSM, Peking University, Beijing* 102600*, China;*
2. *National Computer network Emergency Response technical Team / Coordination Center, Beijing* 100029*, China*）

Abstract: At present, malware is mainly detected through static-based approach, but the recognition rate is low. Based on the mechanism of static detection, this paper gave a behavior-based approach which analysis the behavior of sensitive function in the application. In this way, using the traces of variables and equivalent function matching to determine whether an Android package have malicious behavior. The behavior-based approach improves the accuracy of the static detection. This paper implement a detect tool aim at SMS Trojan and is proved its effectiveness in testing.

Key words: Android; malware detection; behavior analysis

0 引言

目前常见的移动平台恶意程序为短信扣费程序，近年来，Android 平台下已连续发生多起因吸费软件泛滥引发的安全事件。如 2011 年 2 月出现的“安卓吸费王”恶意扣费软件至今已连续植入到超过 100 款应用软件中，诱骗下载强行扣费。2011 年 7 月，“安卓蠕虫群”恶意软件批量来袭，更对手机用户构成了严重的资费损失。短信扣费程序通常使用本文所提到的技术植入在热门应用中。一旦用户安装，便向某些服务提供商发送业务请求短信，这些业务的开通，通常不需要用户回执确认。一旦开通，便产生巨额的业务费，不法分子便可以从中牟利。

恶意代码检测通常分为静态检测和动态检测，静态检测主要是对应用的文件进行特征码分析，通过匹配特征库，来识别恶意程序。本文主要是研究和实现恶意代码行为特征提取和检测方法的实现。为了这个目的，第一步就要对恶意代码进行分析，然后研究恶意代码行为特征及其与系统数据和系统控制之间的关系，因此，本文从恶意代码特征描述和恶意代码检测这两个方面来展开工作。

1 检测原理

恶意短息吸费程序是在用户使用软件的时候，后台调用系统提供的短信接口，例如 SmsManager 类中的 sendTextMessage 程序向某一地址发送某些内容的短信，发送的短信通常是某些服务的开通代码，并且没有确认开通的回馈消息，一旦开通这个服务，移动运营商就会从用户的话费中扣取一定的费用支付给服务提供商。通常在吸费程序发送短信的过程中，为了不被用户发现，通常会隐藏掉发送短信的记录，即不会将短信保存在发件箱中。通过这个特性，我们总结出一个规律，通常隐藏自己行为痕迹的程序，很可能为一个恶意程序。

以扣费程序常用的 sendTextMessage 函数为例，该函数的第四个参数为用来保存这条消息的 Intent，如果这个参数为 null，那

收稿时间：2012-12-15

基金项目：国家自然科学基金资助项目 [61170282]

作者简介：张文（1986-），男，广西，硕士研究生，主要研究方向：网络安全；严寒冰（1975-），男，江西，高级工程师，主要研究方向：图像处理、信息安全等；文伟平（1976-），男，湖南，副教授，主要研究方向：网络攻击与防范、恶意代码研究、信息系统逆向工程和可信计算技术等。

么发送的消息就不会在系统中保留。所以说，我们可以通过分析该函数中参数来确定一个程序的安全与否。无论是使用组合程序绕过权限机制，还是使用逆向工程来潜入，发送短信不保留记录是恶意程序的固定的“行为”，所以这可以看作是一种基于行为的静态检测方法。这种检测方法，不仅要检测函数的调用信息，还要考虑函数的某些参数。从而衍生出跟踪方法中的变量以及函数依赖问题。下面将详细分析变量的跟踪和函数的依赖问题。

就使用 sendTextMessage 检测恶意吸费程序而言，重点侦测的是其第四个参数是否为 null。假设 sendTextMessage 存在于方法 F 中，依照 smali 语法，如果在文件中直接使用变量 null，会在文件开头声明它。我们可以以一个集合 V = {} 来表示这个方法中受 null 影响的变量。若首先出现一个赋 null 值语句 ：const/4 v2，#int 0，那么 V = {v2}，之后，每出现一次赋值语句，都在 V 中匹配用来赋值的变量，如果用来赋值的变量存在于 V 或是为 null，那么将被赋值的变量加入 V 中。

接下来说函数依赖问题。因为 sendTextMessage 总是要在某个类中的某个方法中被使用，那么 sendTextMessage 以及其他函数之间可能会存在依赖关系。我们设一个函数可表示为 F = {Vin，Vout，CList}，其中，Vin 为传入参数 ；Vout 为传出参数，包括返回值和影响的外部变量，和 Vin 存在交集 ；CList 为这个函数的调用列表，被调用的函数都存在这个列表中。若函数 F 调用 sendTextMessage，且 Vin 可导出 sendTextMessage 的第四个参数 vp 为 null，那么，我们可以说 F 与 sendTextMessage 等价，其等价因子为 Vin 中使 vp 为 null 的集合 VPin ；从而检测 sendTextMessage 的 vp 是否为 null 等同于检测 F 中 VPin 是否是 Vin 的一个子集。所以，在检测过程中，不仅要检测 sendTextMessage 函数，还要检测受 sendTextMessage 影响的其他函数。

2 各模块的实现

我们知道，Android 安装包其实是一个 zip 文件，它所有的类都被编译到了 classes.dex 文件中。要获得这些类的行为，首先得将 classes.dex 从 apk 文件中解压出来，然后进行反编译，之后按照上面所说的方法跟踪变量和找出相互等价的函数，才能找出程序是否有恶意行为，所以，这个检测工具主要分为 4 个模块 ：解压模块、反编译模块、变量跟踪模块和函数等价匹配模块，如图 1 所示。

图 1 检测工具模块

本检测工具检测的具体过程为 :

1）用户输入要检测的 apk 文件。

2）判断输入文件是否为 zip 压缩包，如果不是，输入错误，退出。

3）从 apk 文件中把 classes.dex 解压出来。

4）反编译 classes.dex。

5）在反编译过程中，首先查看是否有定义的需检测函数，如果没有，结束。如果有那么建立函数等价关系表，然后对每个函数进行检测。

图 2 为检测过程流程图。

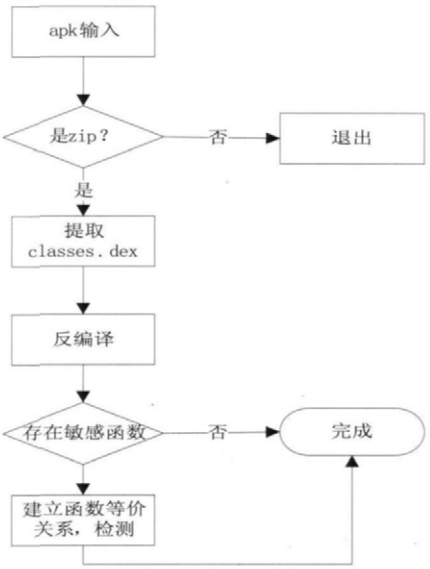

图 2 检测流程图

2.1 解压模块的实现

本模块主要分析 zip 文件格式以及如何使用程序将我们需要的 classes.dex 文件从 apk 包里解压出来，当然，我们并不真正解压该 apk 包，只是将 classes.dex 的数据量提取，然后解压到一个临时文件内，然后将这个临时文件提供给其他模块分析。下面就来具体介绍本模块使用到的数据结构和算法。

Zip 主要使用的压缩算法为 Deflate 压缩算法，该算法是 LZ77 的一个变种算法，实现起来是比较复杂的，当然，不用我们自己实现，目前网上有这个算法的源码。目前 Linux 系统中默认自带 zlib.h 头文件，该头文件是 zlib 库的接口。zlib 是提供数据压缩用的函式库，由 Jean-loup Gailly 与 Mark Adler 所开发，初版 0.9 版在 1995 年 5 月 1 日发表。zlib 使用 DEFLATE 算法，最初是为 libpng 函式库所写的，后来普遍为许多软件所使用。此函式库为自由软件，使用 zlib 授权。截至 2007 年 3 月，zlib 是包含在 Coverity 的美国国土安全部赞助者选择继续审查的开源项目。zlib 的实现，可以在其官方网站获得。

在 zip 文件格式的相关文档上，zip 文件的结构图如图 3 所示。

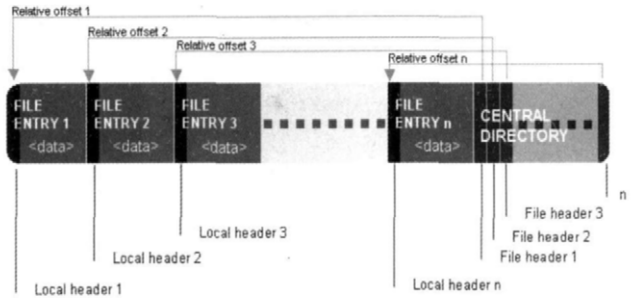

图 3 zip文件结构图

和平常的zip解压方法不同的是，我们对于该zip解压模块的功能并不需要太全，如不需要查看里面的文件，不需要添加；需要的只是把classes.dex解压出来就可以了。因此，本模块的具体过程为：

1）在文件的尾端找到目录区结束标志块的签名0x06054b50。

2）通过目录区结束块中的信息找到目录区起始位置和目录数目。

3）依次遍历目录区，找到名为“classes.dex”的目录。

4）根据找到的目录获得压缩数据的起始位置，压缩方法、压缩大小及解压大小。

5）调用zlib库进行解压。

2.2 反编译模块的实现

本模块主要反编译apk包中的classes.dex文件，该文件由解压模块从Android安装包中解压出来。在Android编译环境中，通过编译工具的编译，会把java文件编译成dex文件，其实这主要是因为java虚拟机和Dalvik虚拟机的不同，Dalvik虚拟机的栈是基于寄存器的，所以要对字节码进行一些优化，以保证程序能在嵌入式系统上流畅的运行。还有，dex文件共用一些类名称，常量字符串，这样，它的体积较小，效率较高，这也是嵌入式上的一种优化。

反编译模块的最终目的是在dex文件中找到各函数字节码定义的位置，然后根据官方对字节码的定义找出其含义从而分析这些方法的行为。分析行为的时候，主要参照上面提到的检测原理。因为目前关于dex文件格式分析的资料较少，在本节中对dex文件中各种的数据结构做了相对较详细的抽象。dex文件反编译的基本过程为：

1）读取文件标识符，判断是否为dex格式文件。

2）通过MethodID列表，获取这个dex文件中所用到的方法，检测是否存在我们定义的要检测的函数，如果没有，那么表示这个应用没有要检测的恶意行为。

3）通过类定义列表，依次遍历dex文件中所有的类，并反编译类函数。

4）在反编译的同时，建立函数等价关系，并通过参数来判定应用是否存在恶意行为。

一个dex文件的简略结构如图 4所示。dex文件头是一个dex文件的总纲，因为在进行恶意程序检测过程中，我们关心的是类中函数的名字和这个函数的字节码，所以图中只给出了取得函数名和该函数的Dalvik字节码的方式。在下面的小节中，将详细介绍图中的模块和代表的意义。

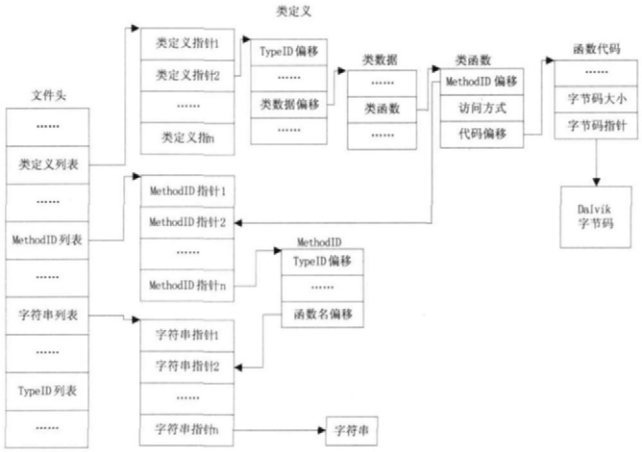

图 4 dex文件简要结构图

2.2.1 文件头的抽象

像其他文件一样，dex文件也有一个文件头，它包含着这个文件的总体信息，就像一本书的目录，根据相关资料，对dex文件头可以做如下抽象：

```
typedef struct DexHeader {
    char         magic[8];          /* .dex file signature*/
    unsigned int  checksum;          /* adler32 checksum */
    unsigned int  signature[20];     /* SHA-1 hash */
    unsigned int  fileSize;          /* length of entire file */
    unsigned int  headerSize;        /* offset to start of next section */
    unsigned int  endianTag;         /* endianness tag */
    unsigned int  linkSize;          /* size of link section, often 0 */
    unsigned int  linkOff;           /* offset of link section, from file start */
    unsigned int  mapOff;            /* offset of map item, from file start */
    unsigned int  stringIdsSize;     /* count of strings */
    unsigned int  stringIdsOff;      /* offset of string list */
    unsigned int  typeIdsSize;       /* count of type identifiers */
    unsigned int  typeIdsOff;        /* offset of type identifier list */
    unsigned int  protoIdsSize;      /* count of prototype identifiers */
    unsigned int  protoIdsOff;       /* offset of prototype identifier list */
    unsigned int  fieldIdsSize;      /* count of field identifiers */
    unsigned int  fieldIdsOff;       /* offset of field identifier list */
    unsigned int  methodIdsSize;     /* count of method */
    unsigned int  methodIdsOff;      /* offset of method list */
    unsigned int  classDefsSize;     /* count of class definitions */
    unsigned int  classDefsOff;      /* offset of class definition list */
    unsigned int  dataSize;          /* size of data section */
    unsigned int  dataOff;           /* offset of data section, from file start */
} DexHeader;
```

这个dex文件头主要申明了编译信息所在的位置，主要是起始位置和大小，包括连接段、map数据、字符串列表、类型列表、原型列表、字段列表、函数列表、类定义列表以及数据段列表的信息。其中比较重要的是类定义列表和函数列表。这两个列表中记载着dex文件中所有的类和方法，反编译时，通过类来索引方法。我们注意到，在这个头中就指出了本dex

文件调用的所有的函数，因此，在某种情况下，我们可以先扫描所有的函数，如果里面没有我们所需要的函数，那么便可以结束扫描。这样便可以大大地节约检测时间。

Dex 文件头的 magic 字段是用来判断是否是 dex 文件的标识符，通常是“ dex\n ”，但有时候也会是“ dey\n ”，这代表着这个 dex 文件是经过 Dalvik 虚拟机优化的。标识符占 magic 前四个字节，magic 的后四个字节为版本号，通常为“ 035\0 ”。其他各字段的含义如代码注释中所描述。

2.2.2 类定义的抽象

Java 是一种纯面向对象的语言，所有的变量和方法都封装在类中，所以，我们要先解析 dex 文件中的类。要找到 .dex 中定义的类，可以通过 DexHeader 结构体中的 ClassDef0ff 和 ClassDefSize 来确定 .dex 中类定义列表。可以如下抽象一个类定义列表中的项 ：

```
typedef struct DexClassDef {
    unsinged int  classIdx;            /* index into typeIds for this class */
    unsinged int  accessFlags;         /* access flag, such as "public " */
    unsinged int  superclassIdx;       /* index into typeIds for superclass */
    unsinged int  interfacesOff;       /* file offset to DexTypeList */
    unsinged int  sourceFileIdx;       /* index into stringIds for file name */
    unsinged int  annotationsOff;      /* file offset to annotations section */
    unsinged int  classDataOff;        /* file offset to classData section */
    unsinged int  staticValuesOff;     /* file offset to DexEncodedArray */
} DexClassDef;
```

类定义列表中的每一项代表着的 dex 文件中所定义的类，特别的，它在类定义列表中的排列有一定的特性，如不能有名字相同的类，父类和接口必须出现在子类的前面，主要是方便在运行的时候不要往返读取这个列表。

在这个结构体中，classIdx 指向的是 dex 文件头中的 typeID 列表中的一项，由 typeID 可以找到字符串列表指针中的一项，然后可以找到代表这个类的字符串，展示出来一般是包名、类名，该类中的子类（如果是临时定义的类，会以数字来表示），例如 “: Lcom/zw/firstPro/FirstProActivity$1 ”，因为在检测的时候并不太关心一个类的名字，这里忽略它。

2.2.3 类数据抽象

依据上节所说，要反编译一个类，classData0ff 是 classDef 中最重要的字段，下面就来 classData0ff 所代表的东西，classData0ff 可以抽象成如下数据结构 ：

```
typedef struct DexClassData {
    unsigned int     staticFieldsSize;     /* the number of static field */
    unsigned int     instanceFieldsSize;   /* the number of instance field */
    unsigned int     directMethodsSize;    /* the number of direct method */
    unsigned int     virtualMethodsSize;   /* the number of virtual method */
    DexField *       staticFields;         /* the defined static fields */
    DexField *       instanceFields;       /* the defined static fields */
    DexMethod *      directMethods;        /* the defined static fields */
    DexMethod *      virtualMethods;       /* the defined static fields */
} DexClassData;
```

这个结构体代表着一个类中的所有数据 ：变量和方法。其中变量有普通变量和静态变量，方法有普通方法和虚方法。在这个结构体中，维护着各种变量和函数的列表，通过这些列表，就可以依次遍历这个类中的变量和函数。

其中，DexField 是一个如下定义的结构体 ：

```
typedef struct DexField {
    unsigned int  fieldIdx;        /* index to a field_id_item */
    unsigned int  accessFlags;     /* access flag, such as "public " */
} DexField;
```

这个结构体代表着这个类中定义的变量的一个列表。其中，fieldIdx 指向的是 DexHeader 中所描述的 fieldIds 列表中的一项，其实就是类型。同一种形式的变量组成一个列表，在这列表里（如 staticFields），fieldIdx 的值都必须由大到小排列。

而 DexMethod 则是一个如下定义的数据结构 ：

```
typedef struct DexMethod {
    unsigned int  methodIdx;       /* index to a method_id_item */
    unsigned int  accessFlags;     /* access flag, such as "public " */
    unsigned int  code0ff;         /* file offset to a code_item */
} DexMethod;
```

这个结构体代表的是类中包含的函数。其中，methodIdex 指向的是 DexHeader 中所描述的 methodIds 列表中的一项，可以看做是一个 methodID 的偏移，表现出来就是 methodIds 列表的一个下标。对于 methodIds 列表中的某一项，可以做出如下数据抽象 ：

```
typedef struct DexMethodId {
    unsigned short  classIdx;  /* index into typeIds list for defining class */
    unsigned short  protoIdx;  /* index into protoIds for method prototype */
    unsigned int    nameIdx;   /* index into stringIds for method name */
} DexMethodId;
```

ClassIdx 指向定义该方法的类名，和类定义中的 classIdx 是一个意思。nameIdx 最后指向一个字符串，该字符串就是这个方法的名字。通过这个名字，就可以判断在文件中是否有我们需要检测的函数。

2.2.4 所有函数的扫描

因为从 dex 文件头可以直接找到 methodIds 列表，这意味着不用遍历所有类就能找到 dex 文件中所有的函数名，如上文所述，在这里便可以开始检测，如果没有检测到已定义的相关方法，那么就不需要继续遍历类了。检测的时机应该在刚读出文件头结构的时候。但在文件头结构中，各偏移都是针对 dex 文件开头。由于 dex 文件普遍都不大（不大于 1M），所以可以把整个 dex 文件都映射到内存中来，避免频繁读取文件。所以，可以重新抽象出一个数据结构 ：

这个结构体可以看作是 dex 文件的总体描述。将文件映射到内存时，将文件在内存中的起始地址保存到 baseAddr 中，然后通过 DexHeader 中的偏移量和 baseAddr 计算出各个结构体列表的首地址，结合这些首地址和 DexHeader 中的列表大小，就可以操作各个结构体列表了。读入文件，初始化这个结构体之后，便可以对所有函数进行扫描了。本来按文档中的定义，StringIds 应该也是一个结构体，但考虑到该结构体中只有一个偏移量，这里索性将其抽象成一个只有偏移量的数组。以检测 sendTextMessage 函数为例，扫描代码如下 ：

```
int methodSize = (int)pDexFile->pHeader->methodIdsSize;
for (i = 0; i < methodSize; i++) {
const DexMethodId * pMethodId = &pDexFile->pMethodIds[i];
const DexStringId * pStringId = &pDexFile->pStringIds[pMethodId->nameIdx];
const unsigned char  * ptr = pDexFile->baseAddr + *pStringId;
// Skip the uleb128 length.
while (*(ptr++) > 0x7f) /* empty */ ;
if (memcmp(ptr, "sendTextMessage", 16) == 0){
break;
}
}
if (i == methodSize){
printf("the apk is safe!\n");
return;
}
```

这样找出来的函数名并不包括类名，所以会有重复的方法名，在它的作用并不在于判断找到的 sendTextMessage 是我们定义的那个能发送短信的函数，只是在没有这个函数的情况下退出程序。毕竟，有 sendTextMessage 不一定能发送短信，没有就一定不能发送短信。如果找到有 sendTextMessage 函数，那么就可以通过上面的 dex 文件结构图，通过类遍历方法，然后依次反编译，建立函数等价关系，然后进行检测。

2.2.5 字节码反编译

在前文中，我们找到了需要编译的 Dalvik 字节码，本节主要简述反编译它们的方法。在这里的反编译中，并不是所有的方法都需要反编译，例如，类中会有一个 <init> 函数，这个函数是 Android 编译环境编译程序加进去的，开发者不能更改，在遍历类中方法时便可以略过。

虽然在 DexCode 结构体中定义的字节码是以无符号短整型为单位，但是 Dalvik 指令集却是少于 256 个，并且是使用一字节来表示。这是因为，除了单字节的指令之外，还有三种比较特殊的状态 :packed-switch-payload, sparse-switch-payload, fill-array-data-payload Format。这三种状态需要两字节来表示，格式为 nop+(0x01，0x02，0x03)。如果按字节来读的话，nop 太多了，每次都要判断后一位，消耗会比较大。所以这里统一读 2 字节，先判断是否为这三种状态，然后再处理。这三种状态对本文的作用并不大，忽略之。

对于每种操作指令，其操作数是固定的，即它们的指令长度固定。可以用一个长度为 256 的数组来存储指令长度，然后以指令代码值为下标来索引，以增加获取指令长度的速度。总结文档中提供的指令规律，可以如下抽象一个指令 ：

```
typedef struct Instruction {
    unsigned int        vA;
    unsigned int        vB;
    unsigned long long  vB_wide;
    unsigned int        vC;
    unsigned int        arg[5];
    OpCode          opCode;
} Instruction;
```

其中，OpCode 为一个枚举型变量，保存着指令代码值，上面的是指令可能的参数，要把指令抽象成一个数据结构的主要原因是 ：指令的参数几乎都不是整位出现的，每次读取参数时都得进行位操作，抽象成一个结构体可以减少这种消耗。在本文中，反编译并不需要关注所有指令，只需要三类 ：变量定义相关，赋值相关和函数调用相关，const 系列，move 系列和 invoke 系列。遇到这三类指令，才填充这个结构体，然后根据他们的含义，就可以进行变量跟踪和方法等价相关操作。

2.3 变量跟踪模块

变量跟踪模块主要作用是记录函数中的变量的值，从而判断调用某个函数时是否传入了特定的参数，如对于 sendTextMessage 主要跟踪为 null 的变量。需要注意的是，变量跟踪和函数等价匹配并分开的，它们都是在对函数进行反编译时完成，即遇到变量跟踪相关指令则进行变量操作，遇到函数调用则进行方法等价的创建。

变量跟踪主要关注变量定义指令和赋值指令，即 const 系列和 move 系列指令，但并不是所有 const 和 move 指令都需要，如 move 系列指令只要考虑操作数为 2 的一些指令。为了方便使用，定义如下函数 ：

```
int getInstructionType(OpCode op);
```

此函数主要返回指令 op 的类型，在本节中，指令类型只分为定义、赋值、调用和其他。实现主要是使用 switch-cast 将我们认为有用的指令作一个分类，因为实现比较简单，这里不再写出实现过程。

变量跟踪可以使用一个集合来存储所有具有同一值的变量，因为用于存储指令操作数的 dalvik 寄存器都是顺序出现。按照规则，一个函数使用的变量和它本身的参数个数和就是这个函数使用的寄存器数，可以在 DexCode 结构体中读取一个函数所使用的寄存器数。因为在建立函数等价关系时，得使用函数的参数，所以实现变量跟踪得有两个集合 ：函数参数变量和函数定义变量。函数定义的变量由 const 指令给出，主要跟踪其 null 变量 ；函数参数变量为定义的寄存器后几个，用来建立函数等价关系，例如有如下形式 ：

```
name           : 'onReceive'
type           : '(Landroid/content/Context;Landroid/content/Intent;)V'
registers      : 10
```

那么函数 onReceive 的参数为 v8 和 v9。变量跟踪使用一个数组即可，数组由固定的值表示固定的含义 ：0 表示 null，1 表示第一个参数，2 表示第二个参数，依次类推且 -1 表示其他值。数组长度为寄存器个数。那么跟踪变量使用的数据结构可以定义为 ：

```
typedef struct ValueTracker {
    unsigned int        num;
    char *              values;
} ValueTracker ;
```

设定之后，遇到 const 系列和 move 系列指令时，按操作数给 values 赋值即可。在本检测工具中，如果检测到使用了

android.telephony.SmsManager类中的sendTextMessage函数，并且第四个参数为null，那么提示用户这个应用不安全。

2.4 函数等价匹配模块

函数等价匹配则是记录了函数的依赖关系，从目的上来说，这是变量跟踪的进一步加强。函数等价匹配主要防止的是当间接调用需要检测的函数时，变量跟踪不准确。以sendTextMessage为例，当其被一个函数封装，并且sendTextMessage的参数都是由封装函数外部传入，那么在封装函数内进行变量跟踪就会无效，此时应该判断封装函数的参数。函数调用主要关心的是invoke系列指令。

本检测工具中，建立函数等价关系使用的结构如图 5所示。

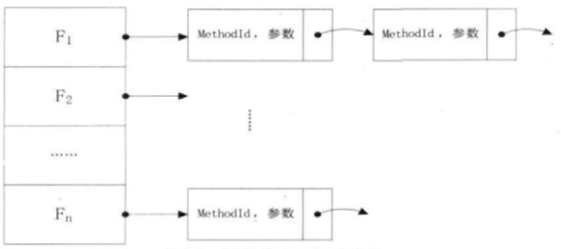

图 5 函数等价关系结构

图 5是dex文件中所有函数的被调用关系，F1到Fn为出现过的函数，使用MehodID列表的下标methodIdx顺序排列，因为methodIdx是从0开始，所以可以使用数组实现，每个数组元素为一个指向链表的指针。每个函数指向的链表为直接调用过该函数的函数。链表中每个节点包含的信息为调用函数的methodIdx以及调用时使用的参数（ValueTracker结构体）。节点如下抽象：

```
typedef struct CallerNode {
    unsigned int        methodIdx;
ValueTracker *     node;
CallerNode *     next ;
} ValueTracker ;
```

在读入dex文件头时，便可以建立函数列表，之后在反编译的过程中，遇到invoke系列指令时，若调用函数的参数和被调用函数的参数有关，则创建一个节点。

3 测试

目前，国家有关部门所公布的短信恶意吸费程序都是由恶意开发者向一些流行应用中加入吸费代码改编而成。本节选取的测试样本为42个从应用市场appchina上随机下载的Android应用。测试主机为ThinkPad R400 A52，操作系统为Windows7 32位旗舰版。

经本工具测试，发现8款有恶意短信行为的应用，其中两款需要建立函数等价关系才能确认。其应用大小及测试时间如表1和表2所示：

表 1 无需建立函数等价关系数据

大小(B)	993570	3853439	2441935	16949	3223424	3220453
时间(s)	0.075000	0.094000	0.046000	0.099000	0.181000	0.177000

表 2 需要建立等价关系数据

大小(B)	1842362	7165873
时间(s)	0.191000	0.742000

值得注意的是，此8款程序安装到测试手机上时，360安全卫士和QQ手机管家均未发出警告。使用这两款软件对apk安装包进行扫描，也未发出警告。其余34款应用的大小和检测时间如图6所示。

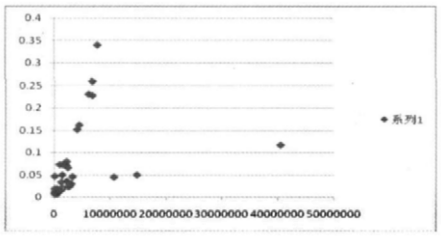

图 6 未发现恶意行为检测数据

除去大于10M的三款游戏应用，我们发现检测时间和文件大小基本上成正比。

对比数据，可以推测出如果检测同样大小的三种应用，可以发现，无恶意的应用检测时间要远小于恶意程序的检测时间，需要建立函数等价关系的恶意程序的检测时间要大于普通恶意程序检测时间。这说明在实现过程中的优化是有效果的。

4 结论和未来工作

本文首先针对当下Android检测软件的不足，使用基于行为的静态检测方法实现了一个针对恶意短信吸费程序的检测工具，然后通过测试证明其可行。

未来的工作包括以下内容：1）对于Android恶意程序，本文只注重于静态检测，缺乏动态检测的相关研究，如果能在Android系统上进行实时动态检测，这将是一个很大的进步。2）本文只着重于短信吸费程序的检测，如何对其他恶意程序实现检测还有待进一步的研究。（责编 杨晨）

参考文献:
[1] 冯登国，赵险峰 . 信息安全技术概论 [M]. 北京：机械工业出版社，2009.
[2] 王蕊，冯登国，杨轶，苏璞睿 . 基于语义的恶意代码行为特征提取及检测方法 [J]. 软件学报，2012，(02)：378-393.
[3] 宋杰，党李成，郭振朝，赵萌 . Android OS手机平台的安全机制分析和应用研究 [J]. 计算机技术与发展，2009，(06)：152-155.
[4] 秦英 . 基于行为的跨站脚本攻击检测技术研究与实现 [D]. 西安：西安电子科技大学，2010.
[5] Samuel T King, Peter M Chen. Subvert Implementing Malware With Virtual Machines[J]. University of Michigan, 2006.
[6] Schwartz, Mathew. Reverse-Engineering, 2001.
[7] lib. zlib source code [EB/OL]. http://www.zlib.net/.
[8] Wiki Pedia. Zip file format [EB/OL]. http://en.wikipedia.org/wiki/ZIP_(file_format).
[9] Google. Dalvik Executable Format [EB/OL]. http://source.android.com/tech/dalvik/dex-format.html.
[10] Google. Bytecode for the Dalvik VM [EB/OL]. http://source.android.com/tech/dalvik/dalvik-bytecode.html.