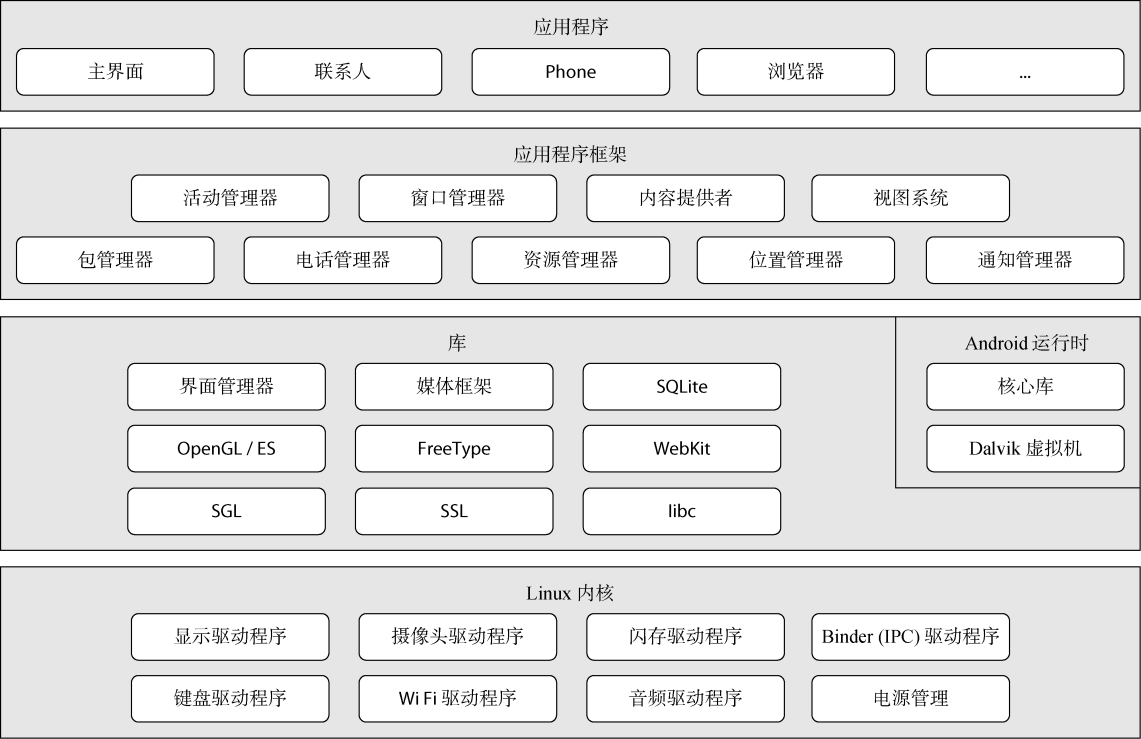
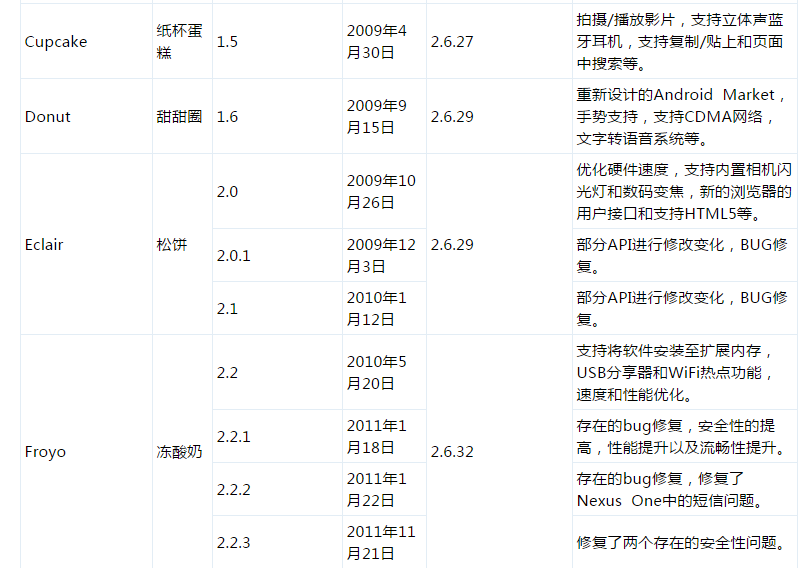
基于Android内存镜像的恶意软件取证研究

1. 绪论
   1. 研究背景与意义
   2. 国内外研究现状
      1. 恶意软件检测研究
      2. 内存取证研究
   3. 论文主要工作
   4. 论文的结构安排
2. 相关知识介绍
   1. Android综述
      1. Android体系结构



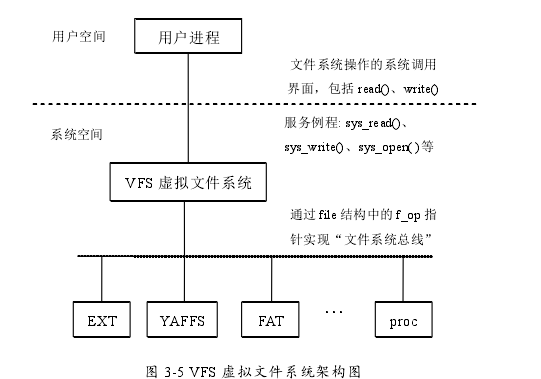
* + 1. Android应用程序概述
  1. Android系统面临的安全威胁
     1. APK重打包技术
     2. 动态加载技术
     3. 运行时自修改技术
  2. 内存取证相关研究
     1. 内存镜像获取
     2. Linux介绍



* + - 1. Proc虚拟文件系统

虚拟文件系统VFS是Linux内核的基本组成部分。从功能实现角度来说，内核只做了两件事情：执行进程和给进程提供一个与外界交互的窗口。虚拟文件系统即是提供一个交互的窗口。

对于虚拟文件系统来说，它一方面要保证快速访问实际文件系统中的数据，一方面还要保证文件和数据能够正确保存。Linux通过告诉缓存协调这两个需求，在高速缓存中不仅缓存数据还管理这操作系统和块设备之间的接口。VFS的设计思想是使得不同的文件系统的实现对用户保持透明，从而提供一个统一的、抽象的文件系统接口，使程序使用同一组系统调用，如read()、write()、open()等操作。

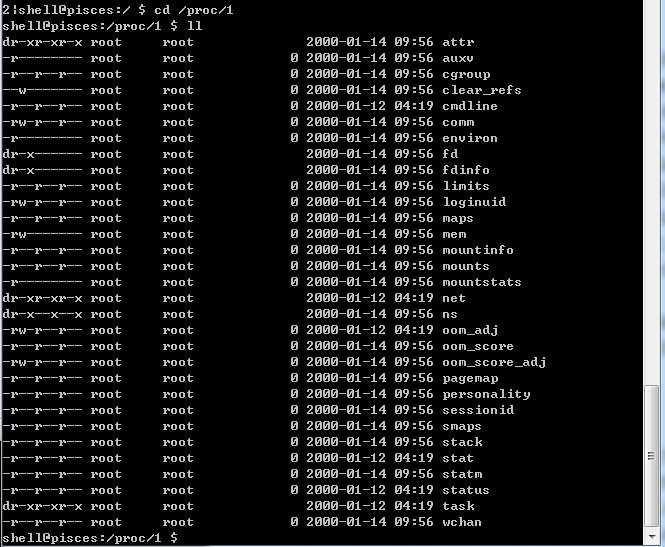


proc文件系统就是一个虚拟文件系统，通过文件系统的接口实现，用于输出系统的运行状态，只在内存当中，而不占用外存空间。它以文件系统的形式为应用进程和操作系统之间的通信提供了一个界面，用户和应用程序可以通过/proc得到系统的信息，设置可以修改某些系统配置。由于系统的信息(如进程等)是动态改变的，所以用户或应用程序读取proc文件的时候，proc文件系统是动态的从内核系统中读取所需要的信息的。

/proc目录下的文件都是特殊的文件，目录下的主要内容包括以下几个部分：

* 以进程PID号做为文件名的子目录，其中包含了进程的运行环境和运行状态，包括环境变量、命令行、CPU占用时间、内存映射表、已打开文件的inode等信息；
* 系统中各种资源的管理信息，如/proc/swaps、/proc/slabinfo等；
* 系统中各种设备的相关信息，位于/proc/devices目录下；
* 文件系统的相关信息，如/proc/mounts、/proc/filesystem等；
* 动态可加载模块的相关信息，如/proc/modules、/proc/kallsyms等；
* 系统版本号以及其他各种统计与状态信息。

由于/proc以文件系统的接口实现，因此可以像访问普通文件一样访问它。例如通过cd /proc/1可以进入1号进程(init)的相应目录。下图就是截图，其中有几个比较重要的符号连接：maps目录下包含该进程的所有的依赖库信息，显示进程的虚拟地址空间是如何使用的；task目录包含该进程的所有线程；cmdline存储进程的完整名字；



* + - 1. KLM动态可加载模块机制

Android操作系统的Linux内核属于单内核体系结构，它将进程管理、内存管理、中断处理、文件系统等编译为一个整体，以实现内核子模块中间的紧密衔接。这种内核体系结构提供良好的运行速度和稳定性，但是也导致系统的可扩展性比较差。为使内核保持较小的体积并能够方便的进行功能扩展，Linux提供了可加载内核模块机制。

可加载内核内核模块是已完成编译并连接可执行文件的程序，能够实现内核的部分功能，在系统需要的时候将其动态的加载到内中并成为内核中的一个组成部分，而不必对内核重新编译或重启系统。在不需要的时候，可以将其从内核中卸载。这种机制在保证了内核可扩展性和灵活性的同时，保证了内核镜像拥有比较小的体积。

由于模块在插入后做为Linux内核的一部分来运行的，因此可以在模块中使用一些由内核导出的资源，如sys\_call\_table，这样就可以根据该地址直接修改系统调用的入口，从而实现改变系统调用。因此LKM可以当作后门来使用，一旦攻击者能够将恶意内核模块加载进系统能够内核，就可以肆意的对系统进行各种各种，如隐藏文件、进程、网络连接等。而这些攻击处在系统的内核层面，所以能够逃过一般的应用级安全工具的检测。此外，LKM也可以当作取证工具来使用，通过加载进入内核中，可以访问到内核中的一些信息，比如获取内存镜像等。

1. Android内存镜像分析
   1. Linux进程和内存管理
      1. Linux进程管理
         1. 进程基本概念

进程是Unixu操作系统最基本的抽象之一，是一个程序在一个数据集上的一次运行，除了可执行程序代码外还包含其他资源，包括打开的文件、挂起的信号、内核内部数据等，是系统资源分配的基本单位。

如果只使用进程来进程调度的话，每次CPU切换的时需要保存的数据太大，所以引入了线程的概念。线程是进程中的活动对象，每个线程拥有独立的程序计数器、进程栈和寄存器，是CPU调度的基本单位。而Linux中对进程和线程的区分并不明显，都是使用进程描述符task\_struct来描述的，只不过线程需要共享进程的资源，在task\_struct中的表现就是共享父进程的资源(比如地址空间、打开的文件等)。

* + - 1. 进程分类

作为通用操作系统，Linux调度器将进程分为了如下所示的三类。

交互式进程：此类进程有大量的人机交互，因此进程不断的处于睡眠状态，等待用户的输入。典型的应用比如桌面程序，Android可见进程都是交互式进程。这种进程对系统的响应时间要求比较高，否则会让用户感觉系统反应迟钝。

批处理进程：此类进程不需要人机交互，在后台运行，一般需要占用大量的资源，但是能够忍受响应速度的延迟。典型的应用比如编译器、网络下载等，Android系统中的服务进程和后台进程即是批处理进程。

实时进程：实际进程对调度延迟的要求最高，这些进程往往执行非常重要的操作，要求立即响应并执行。Android系统中的前台进程都是实时进程，需要对用户发出的指令立即响应。

* + - 1. 进程调度

进程程序在一个数据集上的一次运行，其执行需要CPU的调度。Linux的进程调度是基于分时技术的，允许多个进程并发意味着CPU的时间被分为多个时间片，给每个可运行的进程分配一片。

task\_struct的第一个字段表示了进程的状态，具体含义如下：

volatile long state; /\* -1 unrunnable, 0 runnable, >0 stopped \*/

当state值为0的时候，表示进程处于可执行状态，可以参与CPU的调度。进程调度是操作系统的核心功能，CPU需要从所有的就绪进程中选择一个优先级最高的进程来执行。

Linux2.4及之前是采用的O(n)调度器。在task\_struct中有两个指针struct task\_struct \*next\_task,\*prev\_task把所有的可运行状态(state=0)的进程连接成为一个循环双链表，CPU调度的时候每次需要遍历整个双链表来找到优先级最高的进程，为其分配CPU的执行权。采用静态优先级和动态优先级相结合的方法来确定进程的最终优先级，实时进程具有高于所有普通进程的优先级，普通进程可以设置基本优先级，然后根据执行时间的划分来确定动态优先级。O(n)调度器的性能是与就绪队列长度相关，因而进程数目受到限制，并且由于所有的CPU都共享同一个就绪队列，为了防止因为竞争产生的错误需要额外的加锁，效率更为底下。

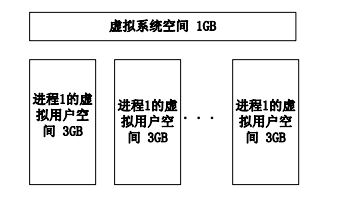
为了改进O(n)调度器，在Linux2.6.0中实现了O(1)调度器，调度时间恒定，和就绪队列的任务数没有关系。O(1)借助非常复杂的公式在常量的时间内获取到优先级最高的进程，但是有些程序会让该调度器的性能显著下降，导致交互式进程反映缓慢。此外，O(1)调度器确定优先级的公式太多，使用了大量难以维护和阅读的复杂代码，虽然性能得到了解决但是由于代码过于复杂，后续的维护非常困难。

因此在Linux2.6.36中采用了完全公平调度器CFS(Completely Fair Scheduler),不在使用各种复杂的公式来对进程进行划分，而是对所有的进程平等对待，所有任务的执行时间都按照同一规则来决定。在全局上依旧将进程划分为实时进程和普通进程，只改进了对普通进程的调度策略，不再使用优先级队列而是采用了红黑树的实现，红黑树的每个节点的键值是定义的虚拟运行时间也是有优先级来决定的。

由2.3.2讨论可知，Android2.2以后的发行版本都是采用的2.6.3+的Linux内核版本，所调度器采用的均是CFS调度器。

* + 1. 内核虚拟地址转换

进程地址空间由进程可寻址的虚拟内存组成，进程虚拟内存空间包括内核态空间和用户态空间，其中内核态空间是所有进程共享页表页目录。Linux 简化了分段机制，使得虚拟地址与线性地址总是一致，因此， Linux 的虚拟地址空间也为0～4G 字节。Linux 内核将这4G 字节的空间分为两部分：将最高 的1G 字节（从虚拟地址0xC0000000 到0xFFFFFFFF），供内核使用，称为“内核空间”；而将较低的3G 字节（从虚拟地址0x00000000 到0xBFFFFFFF），供各个进程使用，称为“用户空 间”。因为每个进程可以通过系统调用进入内核，因此，Linux 内核由系统内的所有进程共享。System.map内核符号表枚举的内核变量和函数的地址是虚拟地址，而内存镜像中的地址都是物理地址，在分析内存地址的时候需要把虚拟地址映射成物理地址。



内核维持着一组自己使用的页表，驻留在内核全局也目录中。该目录存放在swapper\_pg\_dir变量中，大小为PAGE\_SIZE(在32位环境中为4K，64位环境中为8k,android大多数都是32位环境)。

在ARM Linux系统中有两种页表结构：一种是3G以上一级页表，用于内核态寻址，是静态的；一种是ARM MMU的两级页表结构，用与虚拟地址转换为物理地址，是动态的。Linux操作系统从3G开始，把虚拟地址沿着物理内存开始，一直映射到物理内存的边界，这部分页表只起到虚拟地址到物理地址的转换作用。3G以下的地址采用二级页表模型，不过为了使得ARM MMU能够在Linux的虚拟内存框架下工作，做了一些设计：在Linux虚拟内存模型下，需要记录页面的dirty、可读、可写等属性，但是ARM L2的页表项并不能跟Linux的要求一一对应，而且L2页表项又被硬件全部占满，所以对于每个L2页表Linux准备了两份，一份给硬件MMU使用一份给Linux本身使用。

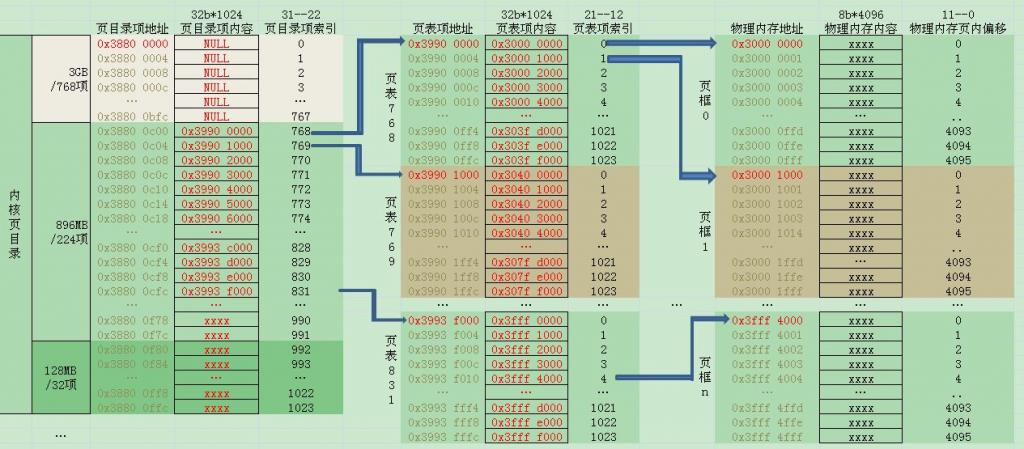
Linux在初始化时，调用pagetable\_init()(位于/arch/arm/mm/init.c)根据物理内存为内核初始化页目录项和页表项。由于内核页表是在内核态中使用，因此只映射0xC0000000 到0xFFFFFFFF的地址空间，即是主内核目录前768项均为NULL。同时内核全局目录的标志位设置及为0x1e3,表明页面的dirty、可读、可写等属性，含义如下：

/\**0x1e3*==Present, Accessed, Dirty, Read/Write, Page Size, Global\*/

对于内核空间而言，给定一个虚拟地址x，其物理地址是从x中减去PAGE\_OFFEST。Linux提供了一个宏\_pa()(位于/include/linux/)来实现这个转换过程，其定义如下：

#define \_\_pa(x)   ((unsigned long) (x)-PAGE\_OFFSET)

实际上在物理内存小于4G以及没有其他映射方式的时候，主内核全局页目录从768至991是从物理内存0处开始映射的。因为如果从页目录768到1023都采取直接映射，最多也只能同时覆盖1GB物理内存，所以内核干脆将最后的32项不参与直接映射儿时可以多次重复映射，每次映射都是暂时的，这样前面的224个页表直接映射(可以同时覆盖896M物理内存)，后面的32项页表通过重复映射，能覆盖的地址就不止128M。一个计算机系统的物理内存大小都是固定的，所以前面224个内核页表（对应页表目录项768-991）的内容可以确定。如果物理内存小于896M，则可以只使用768~991的页表就可以全部覆盖，页表项的内容在整个系统运行期间固定不管；如果物理内存为2G，那么Linux在启动的时候先为前面的896M物理内存建立内核页表，剩下的1G+128M的物理内存不得不在使用某个页框的时候用992到1023项页目录的临时映射。



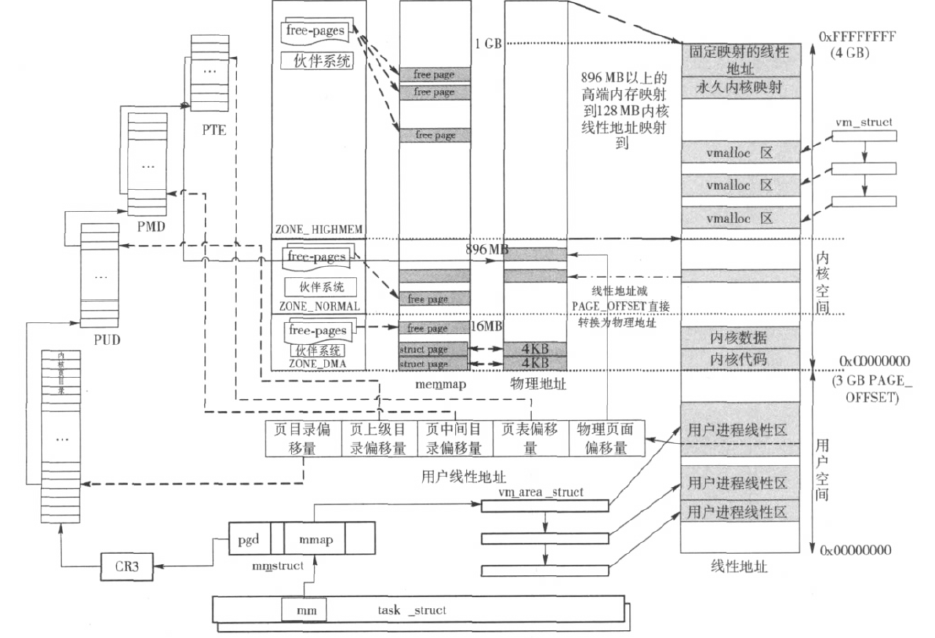
但是如果出现固定映射、永久内核映射以及非连续内存映射的时候，这种线性规则被打破。因此，尽管Linux提供宏操作\_pa()来实现内核虚拟地址向物理地址转换的方法，在实际应用中还是要根据主内核也全局目录来检查转换的合理性，这种检查方式如下：

\*(pgd + i) == (i – 768)\*4M + 0x1e3

如果第i(991>= i >=768)项页目录不满足该式，那么就不能通过\_pa()进行地址地址转换，而要用模拟硬件的MMU进行地址转换。System.map中的内核符号地址是GCC在编译连接的时候得到的，内核符号地址是根据Linux内核映像启动时加载到物理内存中的位置和虚拟地址之间的转换得到的。

* + 1. Linux内核结构介绍

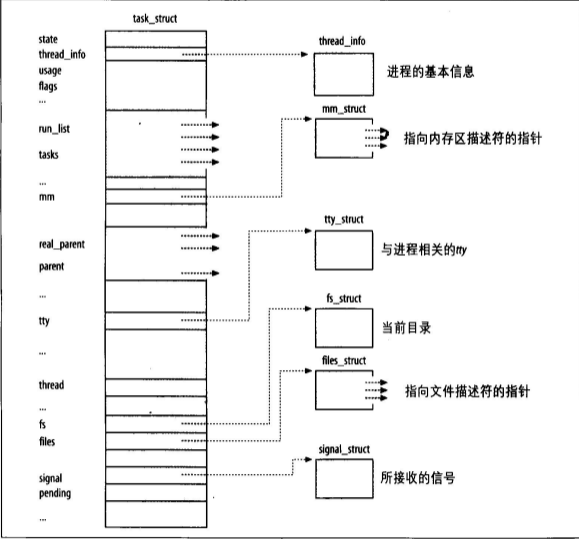
计算机中易失性数据大部分都在内存中，进行内存镜像分析的时候需要先将内存的结构整理清楚。下图展示了Linux操作系统所采用的虚拟内存管理模式中的重要数据结构、结构之间的交互，物理内存、虚拟内存之间的关系等。本文使用的Linux\_kernel源码使用的是[android-goldfish-3.4](https://android.googlesource.com/kernel/goldfish/+/android-goldfish-3.4)，goldfish是一种虚拟的ARM处理器在Android的仿真环境中使用。



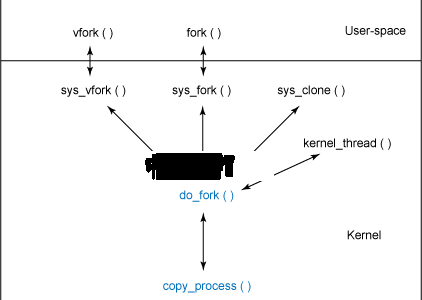
下面对一些本文需要用到的数据结构进行分别的介绍。

* + - 1. 进程描述符task\_struct

对于所有多任务操作系统，进程都是一个非常重要的概念。并且所有的进程都需要一个进程描述符，存储进程的相关信息。Linux也不例外，这个结构就是task\_struct(位于/include/linux./sched.h)，进程只有通过这个数据结构，才能成为内核调度的一个基本单位。



Linux中除了第0个进程以外，其他所有进程都是由一个已有进程在用户态中完成创建的，通过clone()系统调用实现fork()。fork()、vfork()、\_clone()库函数都是根据各自需要的参数标志去调用clone(),最后指向do\_fork()函数（位于/kernel/fock.c）中，系统会为子进程创建一个task\_struct结构，并且从父进程中继承一些数据，然后把新进程插入进程树，等待进行进程管理、调度等。



task\_struct的结构非常大，在32位机器上大约有1.7K，其包含的数据能够完成的描述一个正在执行的程序：打开的文件、进程的地址空间、挂起的信息、进程的状态等。

Linux内核将task\_struct的指针放在thread\_info结构体中，而这个结构体又和内核栈一块被放在8KB的内核空间thread\_union中，具体内容如下所示：

struct thread\_info{

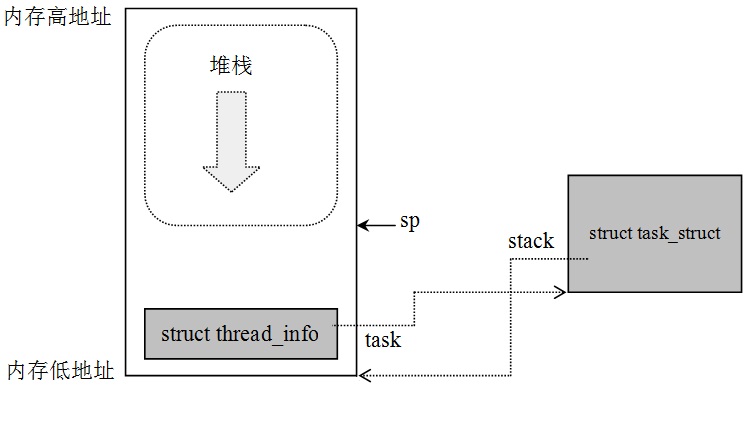
struct task\_struct \*task;

struct exec\_domain \*exec\_domain;

.....

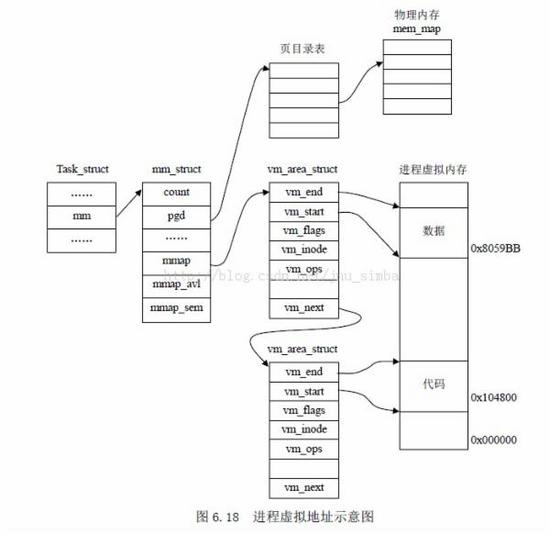
};

可以看出，一个进程描述符task\_struct的首地址和内核堆栈的首地址是一样的。通过查找内核堆栈的首地址就可以得到进程描述符的首地址了。



* + - 1. 内存描述符mm\_struct

在进程的 task\_struct 结构中包含一个指向 mm\_struct 结构的指针， mm\_strcut 用来描述一个进程的虚拟地址空间，称为内存描述符 。该结构还包含有指向 vm\_area\_struct 结构 的几个指针，每个 vm\_area\_struct 代表进程的一个虚拟地址区间。 vm\_area\_struct 结构含有指向vm\_operations\_struct 结构的一个指针， vm\_operations\_struct 描述了在这个区间的操作 。 vm\_operations 结构中包含的是函数指 针；其中，open、close 分别用于虚拟区间的打 开、关闭，而nopage 用于当虚存页面不在物理内存而引起的“缺页异常”时所应该调用的函 数， 当 Linux 处理这一缺页异常时（请页机制），就可以为新的虚拟内存区分配实际的物理内存 。



* + - 1. 就绪进程队列runqueue

Linux在开机启动中，如果开启CONFIG\_SMP,则会为每个CPU核创建了一个进程就绪队列，代码如下：

DEFINE\_PER\_CPU (struct rq, runqueues);

可以看到，其中使用了一个结构体struct rq，就是进程的就绪队列runqueue(位于/kernel/sched/sched.h)，其重要成员如下：

struct rq {

....

/\* runqueue lock: \*/

raw\_spinlock\_t lock; //自旋锁，对rq进行操作的时候仍然需要锁定

unsigned long nr\_running; /本CPU上的就绪进程数，是实时和普通进程的总和

struct cfs\_rq cfs; //普通进程的就绪队列，采用cfs调度算法

struct rt\_rq rt; //实时进程的就绪队列，采用静态优先级的调度算法

....

}

根据 分析可知，Linux内核中，进程被分为普通进程和实时进程，因而runqueue中使用不同数据结构来组织：使用struct cfs\_rq cfs来组织普通的进程队列，采用CFS调度策略，使用了红黑树的数据结构；使用struct rt\_rq rt实时进程，主要使用了struct rt\_prio\_array active来保存实时进程,使用了双链表的数据结构。

struct cfs\_rq(位于/kernel/sched/sched.h)中和进程组织的主要成员如下：

struct cfs\_rq {

...

struct rb\_root tasks\_timeline; //红黑树的根节点

struct rb\_node \*rb\_leftmost; //红黑树最左边的子节点

struct sched\_entity \*curr, \*next, \*last, \*skip; //当前运行的调度实体、下一个实体等

struct rq \*rq; //所属的runqueue

...

}

CFS调度器中不是直接使用task\_struct来调度普通进程，而是使用struct sched\_entity，每个进程描述符里面都有一个该结构体的指针。struct sched\_entity(位于/include/linux /sched.h)的主要成员如下：

struct sched\_entity {

struct rb\_node run\_node; //红黑树节点

struct list\_head group\_node; //所属的

...

u64 exec\_start; //进程开始时间

u64 sum\_exec\_runtime; //总共执行时间

...

}

该结构体有两个作用：一个是包含进程调度需要用到的信息，比如进程运行时间、睡眠时间、调度参数等；一个是用来组织进程，普通进程是使用rb\_node的红黑树节点来实现。rb\_node(位于)的结构如下：

struct rb\_node

{

unsigned long rb\_parent\_color;

#define RB\_RED 0

#define RB\_BLACK 1

struct rb\_node \*rb\_right;

struct rb\_node \*rb\_left;

}

因此，可以通过cfs\_rq中的tasks\_timeline当作根节点，然后通过rb\_node来遍历整个runqueue中的就绪队列。

* + - 1. Linux内核链表list\_head

从上面的各种内核数据结构中都可以看到，Linux内核中大量使用了循环双链表list\_head，其定义(位于/include/linux/types.h)如下:

struct list\_head {

struct list\_head \*next, \*prev;

};

可以看出，在此结构中只是定义了两个相同类型结构的指针：prev和next指针，和一般的双链表相比，其并没有定义数据域。struct list\_head的出现场景很多，例如在进程描述符task\_struct中定义的list\_head就包括如下所示：

struct task\_struct {

……

struct list\_head tasks; //全部进程描述符

struct list\_head local\_pages; //本地页面链表

struct list\_head thread\_group; //线程组链表

……

};

可见task\_struct被包含到了多个链表当中，其对应的数据类型就应该是本身。而在mm\_struct结构中也有list\_head的出现，如下所示：

struct mm\_struct {

……

struct list\_head mmlist; /\*所有活动(active)mm的链表 \*/

……

};

其数据结构就应该是mm\_struct。如果按照标准链表的方式定义双链表，则需要定义多个不同的双链表,每种的数据类型均不一样。

没有在链表中定义数据域正是Linux在设计和实现时巧妙、灵活的应用。如果在双链表结构中也定义数据域的话，那么其数据类型只能有一个，当内核中需要多个双链表且其数据类型不一样的时候就需要为每一种数据类型就定义一个双链表，这样就使得内核不够灵活、不易扩展。为此Linux内核中使用这种颠覆性的设计：不在链表中包含数据，而是将链表的节点包含到数据结构里面。这样只需要在需要连接成为链表的数据结构中包含struct list\_head成员，就可以将这个数据结构的不同成员连接起来，而不需要考虑每个结构的类型。

没有在链表中定义数据域，那么遍历这个链表的时候，如何从list\_head变量获取到其所有在的数据结构变量就很困难。而在Linux中是通过一个宏定义list\_entry(位于/include/linux/list.h)来实现的，具体代码如下：

/\*\*

\* list\_entry - get the struct for this entry

\* @ptr: the &struct list\_head pointer.

\* @type: the type of the struct this is embedded in.

\* @member: the name of the list\_struct within the struct.

\*/

#define list\_entry(ptr, type, member) container\_of(ptr, type, member)

可以看见list\_entry对应的是宏container\_of(位于/include/linux/kernel.h)，具体代码如下：

#define container\_of(ptr, type, member) ({

const typeof( ((type \*)0)->member ) \*\_\_mptr = (ptr);

(type \*)( (char \*)\_\_mptr - offsetof(type,member) );

})

container\_of宏的作用是根据一个结构体变量中的一个域成员变量的指针来获取指向这个结构体变量的指针。typeof是GNU C对标准C的扩展，它的作用是根据变量获取变量的类型；(char \*)\_\_mptr转换为字节型指针。(char \*)\_\_mptr - offsetof(type,member) )用来求出结构体起始地址（为char \*型指针），然后(type \*)( (char \*)\_\_mptr - offsetof(type,member) )在(type \*)作用下进行将字节型的结构体起始指针转换为type \*型的结构体起始指针。通过这个方法，list\_entry就可以利用list\_head的指针来获取其所在的结构体变量的指针。

* 1. 隐藏进程检测方法

恶意软件在执行其他恶意操作的时候，可能会结合各种隐藏手段来逃避杀检测，其中最常用的就是隐藏进程和文件的方法，防止通过ps等命令来发现可疑的进程。正常的软件并会不有这些隐藏操作，所有如果发现了这些操作就可以认为其就是一个恶意软件。

* + 1. 常用隐藏进程方法

类似与ps、top等的命令能够列出除了0进程以外包括init进程的其他所有进程。因此，简单的隐藏进程的方法就是对它们进行修改来实现。具体做法就是劫持ps等关键系统调用sys\_getdents(),过滤掉需要隐藏的进程信息，不让其显示。这种做法只是隐藏了调用命令的显示结果，在/proc文件目录下面还是能够找到隐藏的进程信息的。

在Linux中直接查询进程信息的系统调用是不存在的，类似与top和ps等查询进程信息的命令都是通过proc文件系统来实现的，而proc文件系统是通过虚拟文件系统接口来实现的，因此通过劫持文件系统操作集(inode\_operation和file\_operations)中的有关函数，使得进程信息无法通过文件系统展示给用户。这个方法需要判断输出的信息是否与进程相关，如果无关则不应该隐藏。

proc文件系统除根节点/proc是静态设置的，其他都是动态生成的，当用户需要查看某个具体的进程目录时，才会动态的生成该目录的目录项以及索引节点。因此，可以劫持生成目录时遍历的函数proc\_pid\_lookup()，遇到需要隐藏的进程则直接跳过就可以。

以上方法都是通过KLM劫持具体的系统调用，来完成对进程的隐藏的。正常软件是不需要隐藏自身的，所以在取证的时候，如果发现有进程有隐藏的行为，则可以认定该进程所对应的软件就很有可能是恶意软件，取证的时候需要特别注意。KLM需要按照特定的结构来实现，普通的APK应用程序并不能够按照KLM的方式来加载到内核里面去。

* + 1. 隐藏进程检测方法

通过3.2.1的讨论得知，隐藏进程的方法都是通过劫持系统调用来实现的，这些并没有修改具体的内核对象，3.1.1中讨论的Linux管理进程的链表都是存在的，并没有被修改。所以通过对比进程链表就可以发现被隐藏的进程。此外，无论隐藏进程的技术如果发展都需遵循一个事实：一个进程需要被执行，则必须通过CPU调度，则必须加入到runqueue里面。因此，本文基于内存镜像的隐藏进程检测方法如下：

1)进程出现在展示队列中，而没有出现在ps命令结果ps\_tasks中，即

2)进程控制块P出现在进程直接互联的统计链表all\_tasks(以init\_task为循环双链表的头部)中，但是并没有出现在展示进程队列中,即；

3)进程控制块P出现在runqueue\_tasks链表中，而没有出现在统计链表all\_tasks中，即。

其中ps\_tasks是使用ps命令获取的所有进程列表，list\_tasks是通过ls /proc目录然后抓取数字目录而获取的进程列表，all\_tasks是3.1.1中讨论的struct list\_head tasks的循环双链表，runqueue\_tasks是所有CPU调度时的运行队列总和。通过对比以上几种tasks中的进程链表，如果满足其中任意一条则认为是该进程有很大可能是隐藏进程，该进程所对应的软件或模块为恶意软件的可能性很大，需要取证的时候进行重点关注。此外，由3.1.1的讨论可知，Linux中进程和线程都是使用task\_struct的结构体来表示，区别只是是否共享父进程的资源，而ps\_tasks只会包含进程、all\_tasks会包含进程和线程，所以list\_task会有两个不同的展示，一个是只包含进程的，一个是包含了线程的，的获取方式是/proc/${pid}/task目录下的所有数字目录。

* + 1. tasks获取方法
       1. ps\_tasks获取

ps\_tasks是使用ps命令来获取的所有进程列表，并不需要内存镜像，通过adb工具连接需要取证的Android系统，然后使用 ps > /sdcard/data/ ps\_tasks.txt 即可生成一个ps\_tasks文件，最后通过adb pull /sdcard/data/ps\_tasks.txt ~/tasks即可获取到ps\_tasks文件，最后提取出PID那一列得到最终的进程列表。

* + - 1. list\_tasks获取

list\_tasks是通过遍历/proc目录下的数字文件夹来获取到的进程列表，和ps\_tasks类似，也不需要使用到内存镜像。通过adb工具连接到待取证的Android系统，然后使用命令ll /proc/ > /sdcard/data/list\_tasks1，最后同样通过 adb pull传输到取证工作机上，然后提取出以数字为名字的目录的名字当作最终的进程列表。然后使用一个python脚本去获取每个目pid目录下的task子目录的名字得到list\_tasks2。

* + - 1. all\_tasks获取

按照task\_struct结构的分析可知，Linux系统中的所有进程是组织在一个以init\_task为头的循环双链表里面，所以只要能够获取到init\_task的task\_struct的地址，然后通过struct list\_head tasks就可以遍历获所有进程task\_struct的链表，然后在每个task\_struct中提取出对应的pid\_t pid就可以到的最后的all\_tasks进程列表。

init\_task是进程0的进程控制块。在linux启动过程中，进入分页后的内核代码段通过lss stack\_start %esp指令来为进程0建立内核态堆栈，在/arch/arm/kernel/ head.S中定义了内核态堆栈的结构，如下所示：

ENTRY(stack\_start)

       .long init\_thread\_union+THREAD\_SIZE

        .long \_\_BOOT\_DS

可以看到内核态堆栈由init\_thread\_union表示，其在/include/linux/sched.h中被定义为一个全局变量：

extern union thread\_union init\_thread\_union;

union thread\_union {

       struct thread\_info thread\_info;

       unsigned long stack[THREAD\_SIZE/sizeof(long)];

};

由3.1.1.1中的讨论可知，task\_struct的首地址和thread\_info的首地址重合，而init\_thread\_union是做为一个内核全局变量存在的，可以从System.map里面导出从而得到init\_task的虚拟地址，代码如下：

cat System.map | grep init\_task\_union

按照3.1.2中介绍的方法，使用宏\_pa()转换为物理地址，然后从内存镜像中获取到对应地址的数据，可以获取到init\_task的完整数据。使用宏\_pa()的时候需要主要内核态地址空间是否大于1G,如果是的话则需要使用公式xxxx来进行校验，后续部分需要使用模拟MMU来转化物理地址。当获取到init\_task后就可以根据其内部的变量struct list\_head task来遍历获取所有的进程描述符，获取方法按照2.5.1.4中介绍的宏list\_entry来实现。

* + - 1. runqueue\_tasks获取

内核启动的时候，start\_kernel()调用了sched.init()函数来对每个可用CPU上的runqueueu进行初始化，部分执行代码如下:

void \_\_init sched\_init(void){

....

for\_each\_possible\_cpu(i) {

struct rq \*rq;

rq = cpu\_rq(i); //#define cpu\_rq(cpu) (&per\_cpu(runqueues, (cpu)))

raw\_spin\_lock\_init(&rq->lock);

rq->nr\_running = 0;

rq->calc\_load\_active = 0;

rq->calc\_load\_update = jiffies + LOAD\_FREQ;

init\_cfs\_rq(&rq->cfs);

init\_rt\_rq(&rq->rt, rq);

....

}

....

}

可知使用宏cpu\_rq(位于/kernel/sched/sched.h)能够获取指定的CPU的就绪队列，而宏cpu\_rq是直接使用(&per\_cpu(runqueues, (cpu)))进行替换，得知所有的继续队列是保存在一个数组runqueues中的，并且使用宏per\_cpu来遍历。

其中runqueues的初始化在3.1.3.3的讨论中可知，其是通过DEFINE\_PER\_CPU来初始化的，展开该宏得到如下代码：

\_attribute\_((\_section\_(".data.percpu"))) \_typeof\_(struct rq)per\_cpu\_\_runqueues;

说明其是在.data.percpu这个数据段中定义了一个变量per\_cpu\_runqueues,其类型是struct rq,就是进程就绪队列的结构，而这个变量也是相对段首变量\_\_per\_cpu\_start的偏移量。

此外，在系统启动过程中，start\_kernel()会调用setup\_per\_cpu\_areas来完成一些初始化工作，代码如下：

static void \_\_init setup\_per\_cpu\_areas(void){

...

 for (i = 0; i < NR\_CPUS; i++, ptr += size) {

   \_\_per\_cpu\_offset[i] = ptr - \_\_per\_cpu\_start;

   memcpy(ptr, \_\_per\_cpu\_start, \_\_per\_cpu\_end - \_\_per\_cpu\_start);

  }

}

在该函数中，内核为每个CPU分配了专有的数据区并将.data.percpu中的数据拷贝到其中。由于数据从\_per\_cpu\_start处转移到了私有的数据区中，因此获取变量的值就不能直接使用原来的值，还需要做加上一个额外的偏移量，从上面的代码可以看出，这个额外的偏移量就是全局变量\_per\_cpu\_offset[i],其中i是对应CPU的编号。

综上，获取CPUi的就绪进程队列的首地址的计算公式为：

\_per\_cpu\_start+per\_cpu\_runqueues+\_per\_cpu\_offset[i]

而\_per\_cpu\_start和\_per\_cpu\_offset均是全局变量，可以从System.map中导出，方法如下：

cat System.map |grep \_per\_cpu\_start

cat System.map |grep \_per\_cpu\_offset

获取到所有cpu的就绪队列首地址过后，按照3.1.3.3的讨论可以通过遍历以tasks\_timeline为根节点的红黑树，进而获取到所有调度对象sched\_entity的链表，最后通过3.1.3.4中介绍的宏container\_of来获取到对应的task\_strcut。

* 1. 可疑进程检测方法

并不是所有的恶意软件都会实现隐藏进程的功能。因此，在分析隐藏进程之后还需要分析其他可能的恶意软件。本文是基于Andorid平台，因此恶意软件的主要来源就是APK应用程序。为了减少分析范围，所以还需要检测出可疑的进程然后重点分析他们，完成对可疑软件是否为恶意软件的分类。

* + 1. 恶意应用的分类

根据《[移动互联网恶意代码描述规范](http://www.cnblogs.com/xiaojinma/archive/2012/12/22/2829573.html)》，在用户不知情或者未授权的情况下，在移动终端中安装、运行以达到不当目的或违反国家相关法律法规行为的可以执行文件、代码模块或代码片段可以认为是移动互联网恶意代码。Android平台大多数都是移动平台，所以也符和这个定义。按照其规定的认定标准，Android平台的恶意应用的一般行为包括以下几个部分：

1. 恶意扣费(payment):这类恶意软件在用户不知情或未授权的情况下，通过欺骗用户、隐蔽执行等方式订购各类收费业务或使用移动端支付，进而导致用户蒙受经济损失；
2. 隐私窃取(privacy):这类恶意软件在未经用户授权的情况下，读取用户存储在设备上的个人隐私数据，例如短信、通讯录、通话记录、地理位置等；
3. 远程控制(remote):这类恶意软件在没有收到远程控制指令的时候，并没有什么恶意的行为，但是当接收到远程指令过后，可以根据指令执行其他的恶意行为，其具有远程控制属性；
4. 恶意传播(spread):这类恶意软件通过自动化复制、感染、投递等方式，将恶意软件自身、自身的衍生物或其他恶意代码进行扩散，该类恶意软件通过包含广告代码，通过大量扩散骗取用户点击进而获取其他收入；
5. 资费消耗(expense):这个行为和恶意扣费并不一样，这类恶意软件并不会订购服务商提供的收费业务，而是频繁的进行常规的电话呼叫、短信发送得操作，恶意消耗话费；
6. 系统破坏(system):这类恶意软件会通过各种手段来影响系统的正常使用，造成数据丢失，同时还可能对移动通信网络、互联网络或其他服务的正常使用造成破坏和干扰；
   * 1. 恶意应用的典型行为

从3.3.1的恶意代码分类可以看出，恶意应用在进程恶意操作的时候需要使用到移动终端的电话、短信、网络等模块，大多数情况下都是在隐蔽或者欺骗用户的情况下执行。而Android应用中，这些权限的获取都是通过Android体系结构中的应用层框架(Application Framework)来提供的,需要调用特定的类来获取系统的底层支持，一般是以动态链接.so文件来提供。

综合2.3的介绍，Android平台的恶意软件入侵方式一般为：攻击者通过对正常软件的重新打包，把恶意代码置入其中；然后通过加壳或自修改等方式来隐藏恶意代码，放置杀毒软件的发现；或者通过远程下载的方式，在重新封装的代码中并没有直接置入恶意代码，而是仅置入了一段可以访问远程服务器的代码，这样就可以避免移动设备上的杀毒软件，成功将恶意代码通过网络下载到本地上执行。

* + 1. 可疑进程检测方法

3.3.2中介绍的行为要么需要JNI实现对共享库中本地方法的直接调用，要么就是需要网络通信的支持。如果应用既没有使用网络，也没有其他的隐藏方式，则其为恶意软件可能行就很小。通过分析进程是否调用了网络，是否有链接了自定义或系统提供的敏感的.so文件，可以找出进程是否可能为恶意进程，然后在使用第四章的介绍的检测方法进行检测可以最终确定一个应用是否是恶意应用。

此外，有很多论文介绍了通过分析APK的权限，也可以用来检测一个应用是否可疑。

* + - 1. 网络端口检测

Socket是用户API与网络协议栈之间的一个中间接口层，用户通过调用Socket API将报文传递给协议栈或者从协议栈中读取报文。strcut socket是一个中间层的数据结构，详细内容如下：

struct socket {

socket\_state  state; /未分配、未连接、正连接、已连接、断连接，只对tcp有效

short   type; // socket类型，如SOCK\_DGRAM=1，SOCK\_STREAM=2，SOCKET\_RAW=3

struct sock  \*sk; // 网络层sock的表示

....

};

其中struct sock \*sk是重点，整个TCP/IP协议栈都是围绕这个结构展开，也是状态转换的接入点。在/include/net/tcp.h中定义了tcp\_hashinfo的全局变量，其类型为inet\_hashinfo，保存了当前系统中各种状态的tcp\_sock连接(包括established,listen和bind)，其主要代码如下：

struct inet\_hashinfo {

....

struct inet\_ehash\_bucket \*ehash; //established状态的

struct inet\_bind\_hashbucket \*bhash; /bind状态的

unsigned int            ehash\_size              //established的长度

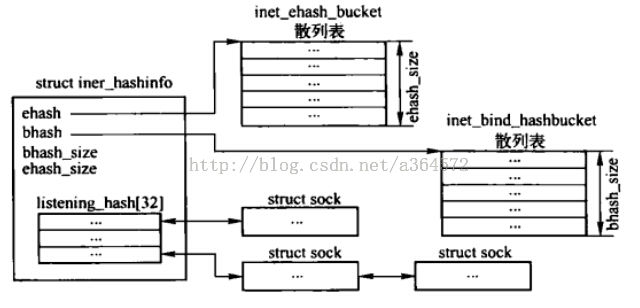
unsigned int            bhash\_size;             //bind的长度

struct inet\_listen\_hashbucket listening\_hash[INET\_LHTABLE\_SIZE]；  //listen状态的

....

};

当内核创建一个sock并绑定端口后，该套接字会被加到bhash链表中；接下来调用listen之后，该sock进入listen状态，放入listening\_hash中；最后经过三次握手建立连接成功过后会被加入到ehash链表中。这三个结构的实现几乎一样，都是指向一个数组，具体如图【】所示。



通过遍历这三个hash链表，使用3.1.3.4介绍的宏container\_of可以获取到所有的sock，进而获取所有的socket地址。

Linux系统在启动过程中会调用sokcet\_init()实现sock\_fs\_type的注册，文件系统存在过后才可以创建socket。用户自定义一个socket的时候，实际上是在sockfs文件系统中创建一个文件节点inode，并建立struct inode和struct socket结构。为了关联这两种结构，Linux内核中实现了socket\_alloc结构，将socket和vfs\_inode封装在一起，struct socket\_alloc(位于/include/net/sock.h)定义如下:

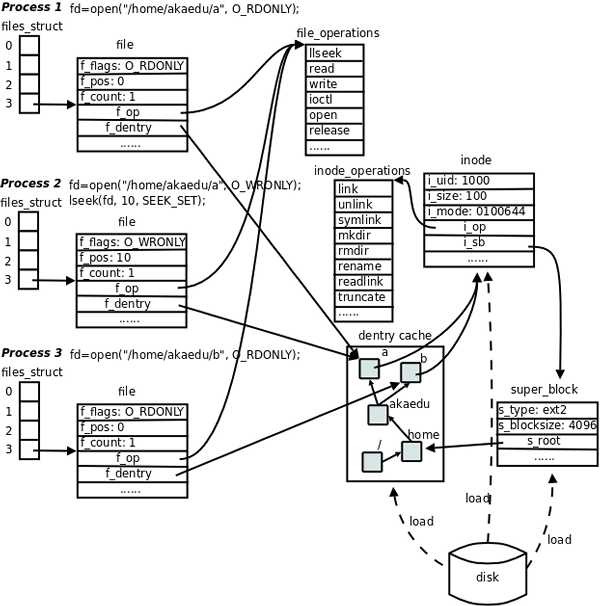
struct socket\_alloc {

struct socket socket;

struct inode vfs\_inode;

};

从上面可以看出，Socket在Linux中的最终表现是一个文件inode(直接是一个文件，而不是一个指针)，而在进程描述符task\_struct中含有一个struct files\_struct \*files指针指向了所有的文件描述符。从图[]可以看出虚拟文件系统中的模块调用情况，通过files指针该进程所有打开的文件指针file，每个file结构体都有一个指向dentry结构体的指针，每个dentry结构体都有一个指针指向inode结构体。



综上，通过从System.map中导出tcp\_hashinfo的地址，最终获取到所有的socket地址，然后偏移整个socket结构体大小的地址，获取到对应的inode地址。按照3.2.3.3介绍的方法，获取到所有的task\_struct结构，进而获取到所有文件的inode。通过对比两个inode，可以确定每个socket所对应的进程信息。

* + - 1. 链接库检测

按照3.1.3.2的介绍可知，可以从进程描述符task\_struct中直接获取到内存描述符mm\_struct，而内存描述符又包含了内存区域描述符vm\_area\_struct(VMA)。VMA结构体描述了指定地址空间内连续区间上的一个独立内存范围，内核将每个内存区域当作一个单独的内存对象管理，相应的操作也是一样的。VMA的重点内容如下：

struct vm\_area\_struct {

struct mm\_struct \* vm\_mm; //相关的mm\_struct对象

unsigned long vm\_start; //区间的首地址

unsigned long vm\_end; //区间的未地址

struct vm\_area\_struct \*vm\_next, \*vm\_prev; //所有的VMA组成链表的链表指针

pgprot\_t vm\_page\_prot; //访问控制权限

unsigned long vm\_flags; //VMA标志

...

}；

其中VMA标志表明了该内存段的页面行为和信息，VM\_READ、VM\_WRITE、VM\_EXEC标志了内存区域中页面的读、写和执行权限，而敏感的行为一般都是需要可执行的，所以重点关注VM\_EXEC权限的VMA块。

按照Android创建进程的COW规则，所有的应用进程都是zygote的子进程。新进程复制Zygote的可共享虚拟地址空间的页表和页目录，而不可共享区域由Linux自身的COW机制在写时复制。

因此通过对比Zygote进程和其他普通进程的所有标识为VM\_EXEC的内存段，可以判断出该进程是否有自己的链接库，如果有的话，则可能是一个恶意应用。

* + - 1. APK权限检测

Android应用所申请的权限信息可以通过解压安装包然后分析AndroidMainfest.xml轻松获取到，而权限信息在一定程度上反映了自身特征，所以对应用程序的权限分析可以初步判断该应用的恶意倾向。

本文采用文献【】所介绍的方法，通过信息熵聚类算法把相关性高的权限聚合在一起形成一个权限簇，通过卡方检验计算计算不同权限簇在分类中权重影响因子，然后使用改进的朴素贝叶斯分类算来来实现对Android应用是否为恶意软件进行初步分类。

在取证过程中，有可能得不到全部正在运行中的程序所对应的安装包APK。然而，通过分析源码frameworks/base/services/java/com/android/server/PackageManagerService.java可知，在应用程序安装过程中，其会调用parsePackage ()方法来解析APK中包含的AndroidMainfest.xml文件，进而获取到其对应的权限、属性、Llinux用户ID名称等信息，写入到文件/data/system/packages.xml中。通过对packages.xml文件的分析，然后对比前面获取的all\_tasks，获取到所有运行进程对应的应用权限信息，分析其是否为可疑软件。



* 1. 本章小结

1. Android恶意应用分类检测
   1. 引言

按照3.3介绍的方法，检测出可疑的进程后，需要进一步检测进程所对应的应用程序是否为恶意软件。目前恶意代码检测技术有很多，如基于特征码的检测、基于行为的检测、沙箱技术等。基于特征码检测的方法通过匹配已有的特征规律来鉴别恶意软件，对未知的恶意代码检测率比较低。因此基于行为的检测又是恶意软件检测的主要方法，这种方法通过分析软件的行为，来判断待测软件是否具有恶意性。根据检测时机的不同，基于软件行为检测又可以分为动态行为监测和静态行为检测。

动态行为检测又被称为外部行为检测，通过把待测软件运行在沙箱、虚拟机甚至实体机里面，检测软件行为，包括系统调用、状态对比、行为跟踪等，进而判断该软件是否为恶意代码。动态检测方法的效率比较高，结果也比较准确，能够检测到未知恶意软件以及通过混淆、加密等技术变种而来的恶意软件。但是动态检测也有自己的不足，比如实施起来比较复杂，需要特定的运行环境才能监控到软件行为。

静态行为检测是指不需要运行待检测软件，而是直接分析软件的安装包所包含的文件、软件的执行代码、依赖的外部资源等，从中提取出程序的运行流程、系统资源调用情况、软件的函数API序列等静态行为，并于已知恶意软件进行对比，从而判断出待测软件是否是恶意软件。静态行为检测实施起来比较简单方便，但是对于一些使用了特殊方法来逃避静态检测的恶意软件，则是无能为力。比如2.2中介绍的Android系统中恶意软件常用的反静态检测方法，比如加壳等方法使Dex文件无法反编译为smali文件，从而无法提取API调用序列得到静态行为。

本文是对恶意软件的取证研究，主要是对待检测Android系统里面的已经运行的软件进行去分析。通过获取可疑进程的内存镜像从中提取出dex文件，则可以成功反编译然后提取API调用序列。同时引入概率神经网络，先使用已知分类的进行学习，最后用来检测从内存中提取出的dex文件是否为恶意代码，从而完成对恶意软件的取证工作。

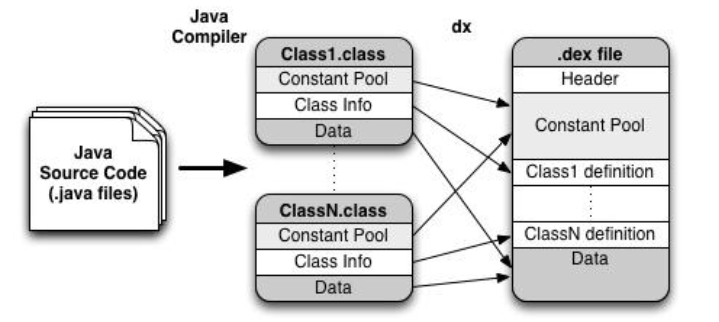
* 1. APK逆向分析

从上节的介绍可知，静态行为检测和动态行为检测在恶意软件取证过程中都有不足。因此，结合取证的特性——待检测的恶意应用处于运行过程中，通过逆向分析获取已经脱壳、自修改过等正常的dex文件，可以为后续的静态行为检测提供比较正确的数据。

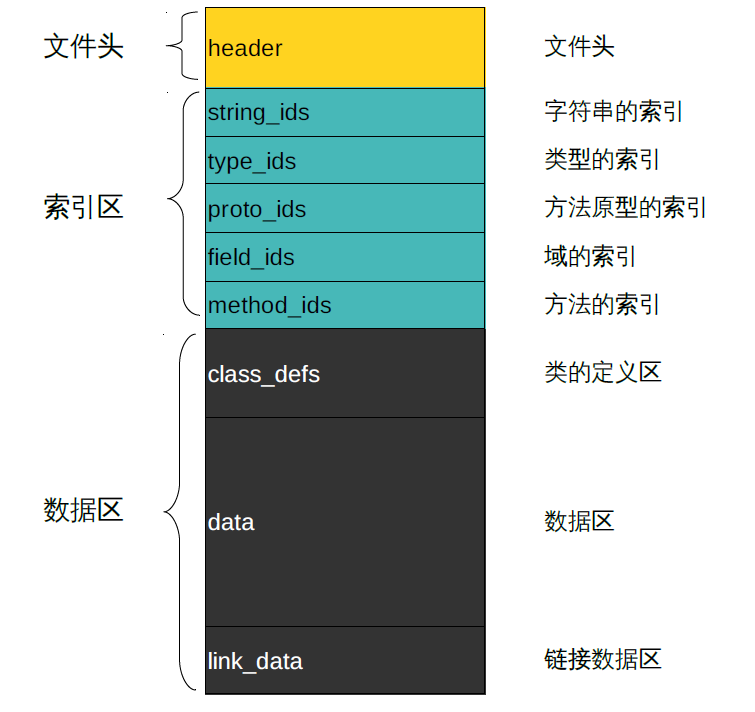
本节涉及到部分Android源码，使用的AOSP源码是android-4.4\_r1.2，build版本是KRT16S，可以按照<https://mirrors.tuna.tsinghua.edu.cn/help/AOSP/>中介绍的方法来下载。

* + 1. DEX文件结构

classes.dex文件是Android系统运行于Dalvik虚拟机上的可执行文件，也是Android应用程序的核心所在。



从Java源文件生成DEX文件的基本映射关系如图【】所示。Java源文件通过Java编译器生成class文件，然后通过dx工具转换为classes.dex文件。Dex文件和class文件一个最大的不同为Dex文件从整体上看是个索引结构，类型、方法名、字段名等信息都存储在一个常量池中——而不是像class文件是每个非匿名内部类都对应一个常量池——从而最大可能减少了存储空间。一个Dex文件的基本结构如图【】所示，相关的结构申明定义在DexFile.h中，在Android源码中的路径为/dalvik/libdex/DexFile.h，下面对重要的结构分别介绍。



header:Dex文件头，包含magic字段、adler32校验值、SHA-1哈希值、string\_ids的个数以及File\_size，其结构固定占用0x70个字节。magic是dex文件的开头，值为”dex/n035/0”,二进制表示为64 65 78 0A 30 33 35 00，通过它可以粗略的校验一个文件是否为dex文件。File\_size表示Dex文件的总长度，通过它和magic可以获取到整个Dex文件。header的主要字段如表【】所示。

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名称 | 偏移值 | 长度 | 描述 |
| magic | 0x0 | 8 | 魔术字段，格式如”dex/n/036/0” |
| checksum | 0x8 | 4 | 校验码 |
| signature | 0xC | 20 | SHA-1签名 |
| file\_size | 0x20 | 4 | Dex文件总长度 |
| string\_ids\_size | 0x38 | 4 | 字符串列表中字符串个数 |
| string\_ids\_off | 0x3C | 4 | 字符串列表基址 |
| class\_defs\_size | 0x60 | 4 | 类定义表中类的个数 |
| class\_defs\_off | 0x64 | 4 | 类定义表中列表基址 |

string\_ids:存储了Dex文件中字符串资源的索引信息，使用了一个结构体DexStringId,详细内容如下所示：

struct DexStringId{

u4 stringDataOff

};

该结构体只有一个成员变量，记录了目标字符串在Dex文件中的实际偏移量。当虚拟机需要读取该字段的时候，只需将magic的地址加上stringDataOff所指的偏移量，即可得到该字符串在内存中的实际物理地址。

type\_ids:存储资源的索引信息，采用和string\_ids一样的设计方法，保存的是目标类型在字符串索引中的序列号。

proto\_ids:存储了原型资源的索引信息，申明了方法原型，包括方法名、返回类型、参数列表。

field\_ids:保存了字段资源的索引信息，包括所属类、字段类型、字段名等。

method\_ids:保存了Dex文件中所有类方法数据的索引信息，包括所属类、方法原型、方法名。

class\_defs：类资源定义，使用了DexClassDef对资源进行规格化，具体数据结构如下：

struct DexClassDef {

u4 classIdx; // 表明所属类的类型，指向类型索引表

u4 accessFlags; //访问标志，如public private static等

u4 superclassIdx; //指向父类信息，Java对象有且只有一个父类

u4 interfacesOff; //接口信息，指向一个DexTypeList数据结构

u4 sourceFileIdx; //源文件所在名字，指向stringIdx索引

u4 annotationsOff; //指向注解的目录结构

u4 classDataOff; //指向DexClassPath结构，表示类的数据部分

u4 staticValuesOff; //表示类中的静态数据

};

calss\_defs数据结构涵盖了一个类所需要的全部资源，classDataOff是一个地址偏移量，指向了一个DexClassData的数据结构，包含了一个类的静态字段、实例字段、类方法、对象方法等，定义在/dalvik/libdex/DexClass.h中。而方法字段又指向一个DexMethod的数据结构，表明了方法的信息，包括方法名、访问权限以及DexCode的偏移量。DexCode是用来记录Dex文件在目标类中的方法字节码，定义如下：

struct DexCode {

u2 registersSize; //需要的寄存器个数

u2 insSize; //数据参数个数

u2 outsSize; //外部方法使用的寄存器数

u2 triesSize; //tries个数,try/catch个数

u4 debugInfoOff; //调试信息地址

u4 insnsSize; //方法指令个数

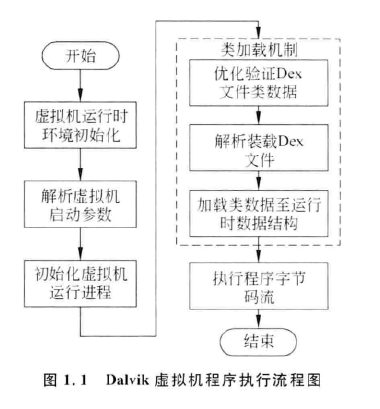
u2 insns[insnsSize];//真实指令数组

};

Dex文件通过DexCode数据结构来管理类方法的全部执行信息，包括了寄存器信息、调试信息以及最重要的指令信息，insns存储了所有的Dalvik指令。

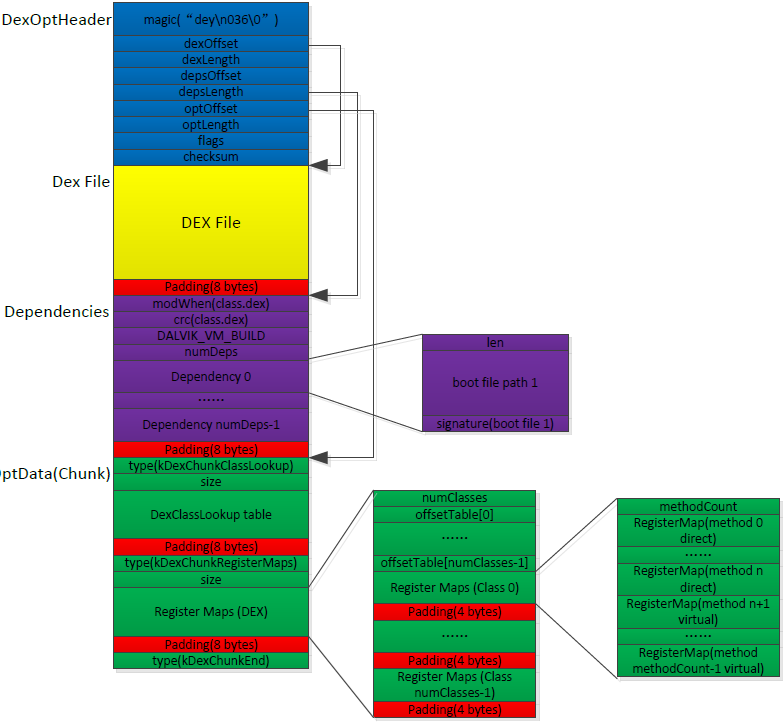
* + 1. Dalvik类加载机制

对于Java虚拟机而言，将类class加载到虚拟机运行时环境的过程叫做类加载。类加载机制的主要功能是将应用程序中的操作码以及程序数据提取并加载到虚拟机内部，以保证程序的正确执行。具体到Dalvik虚拟机就是将Dex文件中的指令与数据提取出来，并加载到Dalvik虚拟机与一个ClassObject数据结构进行关联，实现在虚拟机中执行字节码。类加载具体流程如图【】所示：



* + - 1. Dex文件验证与优化

当虚拟机获取程序中的classes.dex文件后，首先将这个Dex文件进行初步的验证工作，主要包括：验证Magic，校验SHA-a签名，计算Dex校验和等几个方面，当校验完成后，则开始进行优化。Dex文件优化的目的是根据当前平台特性对程序中的字节码进行部分替换，并为Dex文件增加辅助信息以提高Dalvik运行效率，最后输出一个优化文件Odex，其结构如图[]所示。通过magic可以识别odex文件的类型，通过dexOffset可以得到dex的文件的首地址(dexOffset的值就是DexOptHeader的长度，固定为28)，通过dexOffset+dexLength可以得到dex文件的结束地址。



为了更大程序的保证原dex文件的数据的安全性以及优化机制的独立性，优化机制并没有直接改写原Dex文件，而是从新创建了一个Odex文件并以严格的机制要求优化信息写入该文件，主要包括依赖库关系、寄存器映射关系以及类的索引关系，这些管理可以提高类加载机制的执行效率。同时在优化过程中还根据平台特性对原Dex文件进行部分字节码替换（如对字段的替换，把查找改为直接引用），最后将修改过后的Dex文件写入Odex文件中。

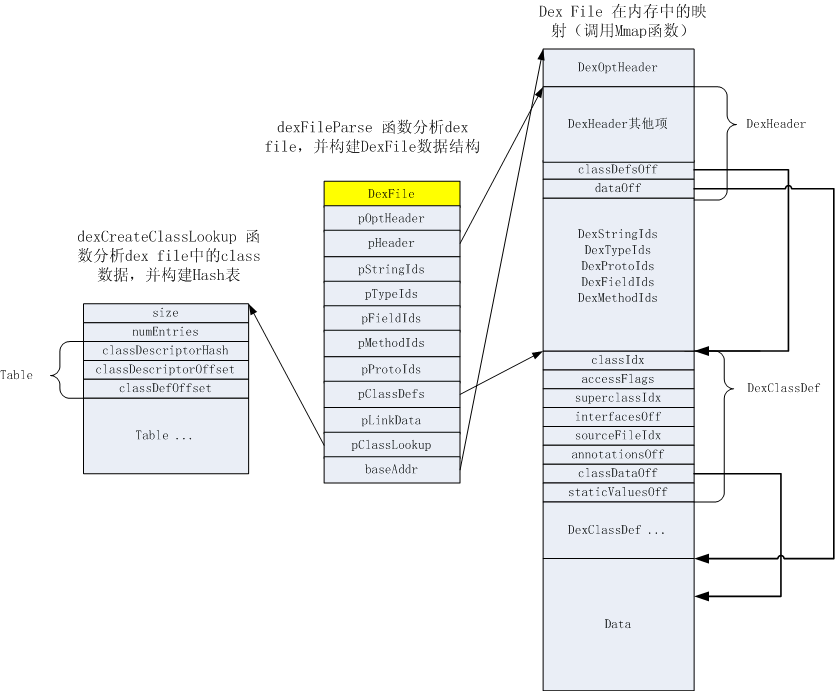
依赖库优化是指Dalvik虚拟机在程序执行前期通过优化机制将Dex的本地函数库整合到Odex文件中，可以在一定程度上提高程序的执行效率。类索引信息的建立是优化机制的重要工作之一，通过为Dex文件建立一个哈希表DexClassLookup，其中的每一项为table结构体对象，记录了一个类的描述符哈希值、类描述符在Dex文件中的偏移以及类定义区的偏移地址，使得类加载机制可以很快的定位类资源地址并完成类加载。

Android设备在应用程序第一次启动的时候往往耗费较长的时间，在这期间虚拟机对目标Dex文件行了验证与优化并为之生成了Odex文件。当用户再次启动的时候，Odex文件会代替原有的Dex文件被虚拟机引用执行，不在需要进行优化，因而启动时间就要短很多。

* + - 1. Dex文件解析

Dex文件与由多个Class文件整合而成，这是的Dalvik在加载一个目标类之前需要对Dex文件进行一系列的处理。Dex文件解析的主要目的就是对Dex文件进行读取分析，并建立一个DexFile结构体实例来描述该Dex文件，使得类加载函数可以通过该结构对目标类进行全部的数据进行索引并提取，完成类的实际加载工作。

在应用程序启动的时候，Dalvik虚拟机通过mmap()函数把Odex文件直接应到到内存中，使得普通的内存读取操作也可以访问dexfile中的内容。Odex文件在解析过程中比较快的原因就是可以把类索引信息、依赖库信息、寄存器关系信息等提前与DexFile进行关联。完成优化数据的处理之后，Dalvik调用dexFileParse()函数对写入Odex文件的Dex数据进行解析，将DexFile数据结构中各个成员变量与Dex文件的各个数据部分进行关联，使得虚拟机能够更加高效的对Dex文件中的数据进行查找并获取，关联效果如图[]所示。



* + 1. GDB调试获取Dex

从Dalvik虚拟机的类加载机制可以看出，APK应用的入口都是通过dex文件的。因为不管如果对APK本身加密、加壳、混淆等，最后在运行的时候，一定要把dex文件分离出来。

DexFile中各个引用都是直接指向dexfile在映射区的位置，并没有自己复制一份数据过去。说明Odex在内存中是连续存在的，Dex文件在内存中也应该是连续存在的。如果能够获取到这个Dalvik虚拟机对应的进程coreDump，那么通过4.2.1介绍的Dex文件结构，找到Magic魔数和File\_size字段，就可以整个的获取到脱壳的Dex文件，最后使用静态行为分析方法就可以判断出该应用程序是否为恶意软件。

* + - 1. GDB介绍

GDB是一个由GUN开源组织发布的基于Unix/Linux操作系统下的程序调试工具。GDB由三个部分组成：1)用户接口(user interface)，除支持传统的CLI接口还支持mi接口；2)符号处理层(symbol handing)，需要使用到BFD/Opcodes库，分别用来反汇编ELF/Core文件，变量/函数/类型的显示否是由本层来完成的;3)目标系统控制层，使用ptrace系统调用来实现对其他进程的控制。

ptrace系统调用提供了一个方法，该方法使一个程序(追踪者)可以观察和控制另外一个程序的执行，并检查和修改被追踪者的内存及寄存器，主要用于断点调试和追踪系统调用。gdb就是基于ptrace这个系统调用来实现调试代码的，通过这个调用在被调试程序和gdb之间建立追踪关系，然后所有发给被调试的信息都会被gdb截获，gdb根据截获的信息，查看被调试的程序的相应内存地址，并控制被调试的程序运行状态。

gdb有两种方式可以建立追踪关系：1)利用fork+execve执行被测试的程序，子进程在执行execve之前调用ptrace(PTRACE\_TRACEME)，建立与父进程(debugger)之间的跟踪关系。2)debugger可以调用ptrace(PTRACE\_ATTACH,pid,...),建立与进程号为pid的进程之间的跟踪关系，使自己变成被调试程序的父进程(使用ps可以看到)。用attach建立的跟踪关系，可以使用ptrace(PTRACE\_DETACH,pid,...)来解除关系。

* + - 1. core dump获取

gdb一般用来调试C/C++程序，因此可以用来调试Dalvik虚拟机。通过使用gcore，可以获取到进程的coredump，里面包含了进程当前的执行内容，包含程序运行时内存、寄存器状态、堆栈指针以及内存管理信息等。通过前面的介绍可知，Dex文件在内存中是连续存在的，因而可以从core dump中获取到。

根据3.3.3介绍的方法获取到可疑的进程列表，得到所有可疑进程的进程ID，并且对比list\_tasks把所有的线程排除掉，最后只剩下进程。因为所有的Android进程都是通过zygote进程通过fork()复制出来的，只有对应的子进程才会有Dex文件的内存映射。

从4.2.3.1的介绍可知，gdb调试必须是通过父进程调试子进程，所以连接方法只能通过PTRACE\_ATTACH来实现。gcore可以为每个指定的进程创建核心映像，获取到coredump文件。将生成的coredump复制回取证工作机，进行Dex的提取工作。

* + - 1. Dex文件获取

获取到coredump文件后，需要从里面提取出Dex文件。从4.2.2的分析中可知，coredump中实际存在的文件是odex文件，是dalvik虚拟机对dex文件进行优化后的文件。结合dex文件格式和odex文件格式的介绍可知，这两个文件里面有几个固定字段，可以用来判定odex文件的位置，固定字段如表【】所示。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 字段 | 位置 | 长度 | 值 | 十六进制表示 | 相对偏移 |
| magic | DexOptHeader | 8 | dey/n036/0 | 64 65 78 0A 30 33 35 00 | 0x00 |
| dexOffset | DexOptHeader | 4 | 0x28 | 28 00 00 00 | 0x08 |
| magic | DexHeader | 8 | dex/n035/0 | 64 65 79 0A 30 33 36 00 | 0x28 |
| header\_size | DexHeader | 4 | 0x70 | 70 00 00 00 | 0x4C |
| dexLength | DexOptHeader | 4 | -- | 和file\_size相等 | 0x0C |
| file\_size | DexHeader | 4 | -- | 和dexLenght相等 | 0x48 |

综上，通过暴力搜索coredump文件，获取odex文件的的方法如图【】所示，通过转换odex文件即可得到dex文件。



* 1. 敏感API提取

API(应用程序结构，Appliction Programming Interface)是一些预先定义的函数，目的是提供应用程序与开发人员基于某软件或硬件得以访问一组例程的能力，而又无需访问源码。Android API位于Android体系结构的第三层，是一套系统调用的接口，为上层应用提供了丰富的框架支持。这些应用利用Android API完成各种软件行为，包括发送短信、访问网络、订阅收费服务等敏感行为，较好的反映了应用的正式行为，可以软件静态行为来进行恶意倾向分析。

* + 1. Smali语法介绍

Dalvik字节码(使用Smali语言描述)是Dalvik虚拟机设计的一种指令集，从Java字节码演化过来的，是一种跨平台的中间代码，可以运行在支持Dalvik虚拟机的各种平台上。Dex文件最后的CodeData中的insns字段即是存放了该Dex文件的全部执行指令。和Java虚拟机不同的是，Dalvik虚拟机是基于寄存器的，可以使用的虚拟寄存器达65536个，每个寄存器的大小是32位，两个相邻的寄存器可以表示64位的数据。在实际操作中，Java源码需要先生成Java字节码(class文件)，在经过dx工具转换为符合Smali语法的Dalvik字节码。

Dalvik虚拟机基于寄存器设计，Java字节码转换为Dalvik字节码时，调用栈已经确定，其中明确指出了使用的寄存器个数以及额外的其他数据。寄存的表示方法有两种：v命名法，采用以”v”开头的方式表示函数中用到的局部变量与参数，以v0开始依次递增，前面表示本地寄存器，后面的表示传入参数寄存器；p命名法，对函数的局部变量命名还是以”v”开头，对传入的参数以”p”开头，p命名法更容易判断寄存器到底是局部变量寄存器还是参数寄存器。如指令”invoke-virtual {v1, p1}, Ljava/util/zip/GZIPOutputStream;->write([B)V”表示该指令需要一个局部变量寄存器、一个传入参数寄存器。非static函数中p0表示”this”,static函数中p0表示第一个参数。

Smali语言中，操作码是1B，操作数是以16为一个单元，除了Java对象和数据是引用类型，其他都是原始类型。基本类型被一个简单的字符描述，类型描述符如表[]所示。

|  |  |
| --- | --- |
| 语法 | 含义 |
| V | void,只能用于返回值类型 |
| Z | boolean，布尔值 |
| B | byte |
| S | short |
| C | char |
| I | int |
| J | long(64位) |
| F | float |
| D | double(64位) |
| L | Java对象 |
| [ | 数组类型 |

出基本类型外，对象、数组、字段、方法等都是通过直接引用字符串常量池来实现的，具体表示如下：

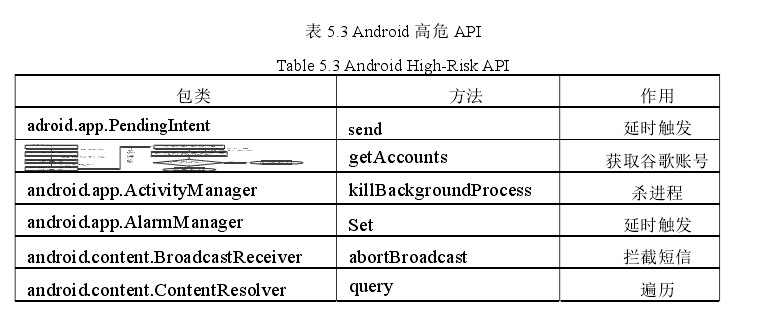
1. 对象以”Lpackage/name/ObjectName;”的形式表示，前面的L表示这是一个Java对象，pacakge/name是该对象所在的包名，ObjcetName表示对象名,如”Ljava/lang/String;”相当于java.lang.String。
2. 数组的表示是通过”[“来实现，多维数据就是多个”[“,如”[I”表示int[]、”[[I”表示int[][];对象数组的表示是通过”[”和”L”结合实现，如”[Ljava/lang/String;”表示String对象数组。
3. 字段的表示是通过类型结合类型来实现,由对象名、字段名、字段类型组成，其中字段名和字段类型用”:”隔开。如”Lct/ax$c;->a:Ljava/util/concurrent/locks/Lock;”表示所属ct.a类、类型为java.uti.concurrent.locks.Lock、名字的a的一个字段。
4. 方法的表示比字段的表示更加复杂，因为Java可以支持静态多态，Dalvik虚拟机通过方法名、类型参数、返回值来详细描述一个方法，具体为” LObject;->MethodName(BII;)V;”。如”Ljava/util/Map;->put(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;”表示Map对象中的put方法，转换为对应的Java源码形式的格式为 Object java.util.Map.put(Object,Object)。
   * 1. 逆向提取敏感API

Dex文件是class文件(jar包)通过dx工具编译成为dex文件的，dx工具调用Smali汇编器将Java字节码转换为符合Smali语法的Dalvik字节码。字节码是用二进制表示的，没有办法直接查看，所以需要使用Baksmali反汇编器来反编译Dex文件，生成Smali格式的反汇编代码.smali文件，每个Java类都对应一个.smali文件。

在.smali文件中，以4.3.1介绍的Dalvik指令的方式显示，可以直接查找需要的代码。提取API即是查找方法调用，因此只需要查找方法调用的指令即可。在Dalvik指令中，方法调用的基础指令是invoke，根据方法类型的不同，共有5条方法调用指令，如表[]所示：

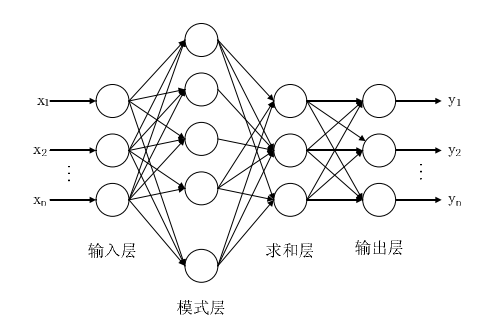
|  |  |
| --- | --- |
| 指令 | 描述 |
| invoke-virtual | 调用实例的虚方法(最常用形式，动态绑定) |
| invoke-direct | 调用实例的直接方法(final/private) |
| invoke-static | 调用实例的静态方法(static) |
| invoke-super | 调用父类方法(super) |
| invoke-interface | 调用接口方法 |

Java代码可以通过混淆来来保护代码，但是有几个地方并不能混淆：使用反射、依赖系统接口、依赖配置文件。API都是系统接口，所以并不能被混淆。因此通过扫描invoke指令后面的方法来获取到我们需要的API。通过3.2.2的介绍可知，涉及到网络、通信、本地信息、系统安全、地理位置等行为的API，可以认定为敏感API。按照文献【】给出统计结果，Android API中敏感API共计41个，如表【】所示。



* 1. Android恶意应用分类检测
     1. 概率神经网络分类器介绍

获取到敏感API调用序列过后，需要进行分类检测，而分类检测需要分类器，本文选取概率神经网络来作为恶意应用检测的分类器。概率神经网络不仅具有神经网络固有的样本数据适应性强、学习能力好的优点，同时相比BP神经网络等，又具有训练迅速、参数调节少、分类能力强、网络结构清晰等有点。在面对输入层参数数目更新时，概率神经网络只需要相应调解对应的模式层单元，不用重建整个网络，可以很好的应对AndroidAPI升级后敏感API对应的特征向量改变的情况。



如图[]所示，概率神经网络基本结构包含四层：输入层，负责接受输入特征向量，神经元个数与特征维数相等；模式层计算输入特征向量各个样本之间的匹配关系，神经元个数等于传入训练样本的总和；第三层是求和层，对模式层计算出的概率进行求和，神经元个数等于需要判断的结果个数，在恶意软件检测中神经元个数应该为2;输出层是由简单的阈值判别器构成，在正常软件和恶意软件的估计概率密度中选择一个较大后验概率的神经元作为真个神经网络的输出。

* + 1. 分类器学习过程

单个分类器的分类结果可能并不准确，因而需要多个分类器集成，最后采用投票的方式来确定分类结果。在分类器集成方面，Boostring和Bagging方法是两种主要的方法。

Bagging算法主要是通过多次重复抽样并训练，形成多个假设，然后将多个假设组成最终的假设，用于具体的分类任务。这种方法可以并行计算，速度比较快，但是每次抽样都是采用的均匀抽样，相互之间没有关系，只是对分类器进行简单的组合，没有发挥出分类器组合的威力。

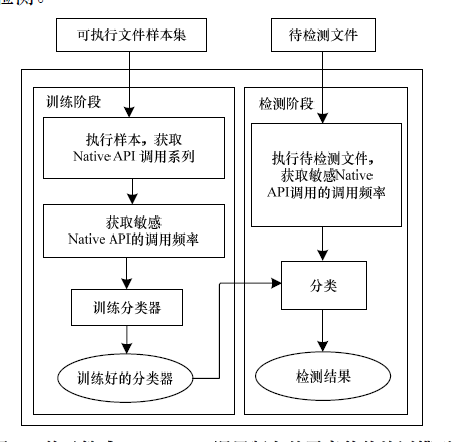
Boostring算法主要思想是在每轮训练中，样本的提取是根据上一轮的错误率来实现的。每次训练过后，下一轮对训练失败的样例赋予较大的权重，加强对错误样例的学习。最终的预测函数对分类问题采用有权重的投票方式，因而具有较高的分类精度。

为了提交Android恶意应用检测泛化能力和分类准确率，同时考虑到算法的运行效率问题，本文利用概率神经网络作为分类器，并采用Bagging学习方式来集成多个分类器，最后采用有权重的投票方式训练出最后的分类器来实现恶意应用的检测。

为了训练分类器，需要有大量的已知状态的样本，包括正常应用和恶意应用。本文首先从一个共享病毒库VirusShare收集了5000个Android应用当作恶意应用样本，并从Google Play下载了5000个正常软件当作正常样本。需要注意的是，有很多正常的应用也嵌入了广告插件、支付插件、辅助插件等调用了敏感API的应用，因而需要排除这些插件对训练的的影响，提起敏感API的时候需要把这些插件排除。训练样本的Dex文件获取方法按照2.1.2介绍的直接解压APK文件即可得到。

* + 1. 基于Android API的恶意应用分类

综合敏感API提取过程，整个基于概率神经网络集成的Android恶意软件分类过程为：1)获取到所有训练样本的Dex文件；2)获取到待测样本的Dex文件；3)提取敏感API得到输入向量，注意排除正常插件的影响；4)使用概率神经网络和Bagging算法多次训练样本，得最后的分类器；5)使用分类器对待测样本进行分类；6)得到最后的分类结果，确定待测样本是否为恶意软件。



* 1. 本章小结

1. 实验与结果分析
   1. 实验环境搭建
   2. 内存镜像获取
   3. 隐藏和可疑进程检测
   4. 恶意应用分类
2. 总结与展望