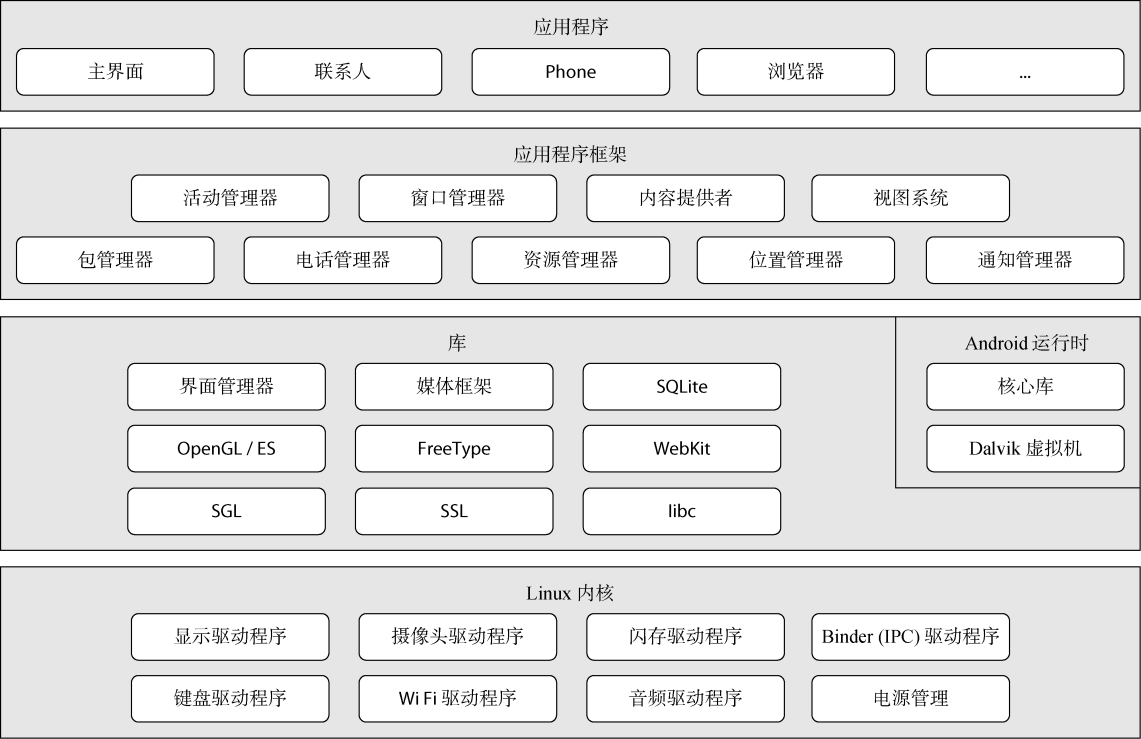
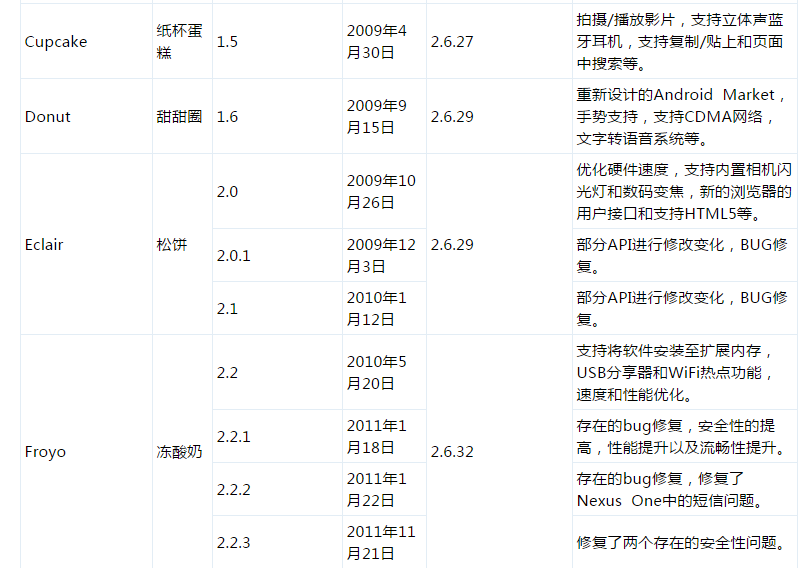
基于Android内存镜像的恶意软件取证研究

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



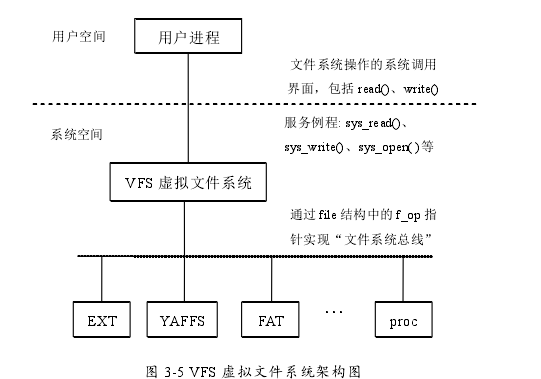
* + 1. Android应用程序概述
  1. Dalvik虚拟机和dex文件
     1. Dalvik虚拟机
     2. dex文件结构
  2. Android系统面临的安全威胁
     1. APK重打包技术
     2. 动态加载技术
     3. 运行时自修改技术
  3. 内存取证相关研究
     1. 内存镜像获取
     2. Linux介绍



* + - 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。

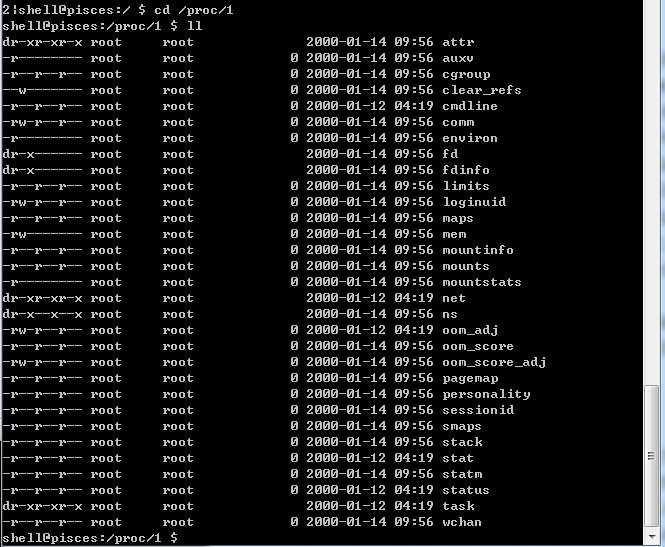


proc文件系统就是一个虚拟文件系统，通过文件系统的接口实现，用于输出系统的运行状态，只在内存当中，而不占用外存空间。它以文件系统的形式为应用进程和操作系统之间的通信提供了一个界面，用户和应用程序可以通过/proc得到系统的信息，设置可以修改某些系统配置。由于系统的信息(如进程等)是动态改变的，所以用户或应用程序读取proc文件的时候，proc文件系统是动态的从内核系统中读取所需要的信息的。

/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目录下包含了该进程的所有的依赖库信息，显示了进程的虚拟地址空间是如何使用的；task目录包含了该进程的所有线程；cmdline存储了进程的完整名字；



* + - 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，但是也导致了系统的可扩展性比较差。为了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。

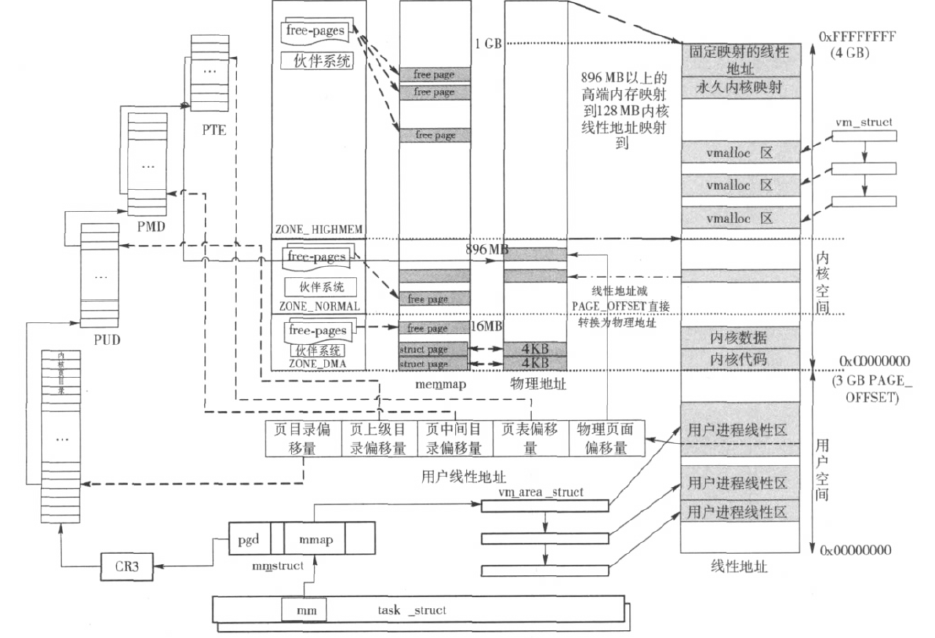
由于模块在插入后做为Linux内核的一部分来运行的，因此可以在模块中使用一些由内核导出的资源，如sys\_call\_table，这样就可以根据该地址直接修改系统调用的入口，从而实现改变系统调用。因此LKM可以当作后门来使用，一旦攻击者能够将恶意内核模块加载进系统能够内核，就可以肆意的对系统进行各种各种，如隐藏文件、进程、网络连接等。而这些攻击处在系统的内核层面，所以能够逃过一般的应用级安全工具的检测。此外，LKM也可以当作取证工具来使用，通过加载进入内核中，可以访问到内核中的一些信息，比如获取内存镜像等。

1. Android内存镜像分析
   1. Linux进程管理

进程管理是操作系统的核心功能。Android是采用的Linux内核，并对其进行了一些修改，但是其进程管理并没有进行修改，所以研究Android的进程管理需要研究Linux的进程管理方法。

* + 1. Linux内存结构剖析

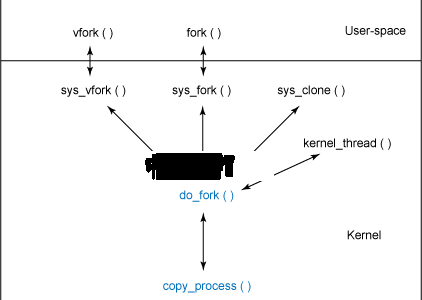
计算机中易失性数据大部分都在内存中，进行内存镜像分析的时候需要先将内存的结构整理清楚。下图展示了Linux操作系统所采用的虚拟内存管理模式中的重要数据结构、结构之间的交互，物理内存、虚拟内存之间的关系等。本文使用的Linux\_kernel源码使用的是android-goldfish-3.4,地址是https://android.googlesource.com/kernle/android-goldfish-3.4



* + - 1. 进程PCB结构task\_struct

所有的操作系统中，如果有进程管理则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，在源码中位于

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，具体内存如所示：

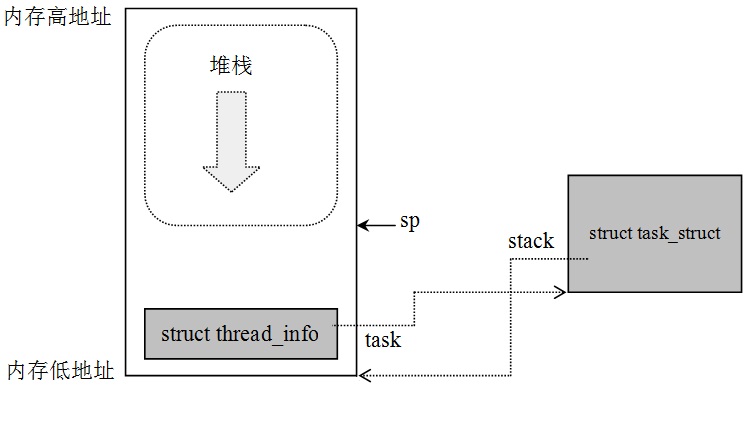
struct thread\_info{

struct task\_struct \*task;

struct exec\_domain \*exec\_domain;

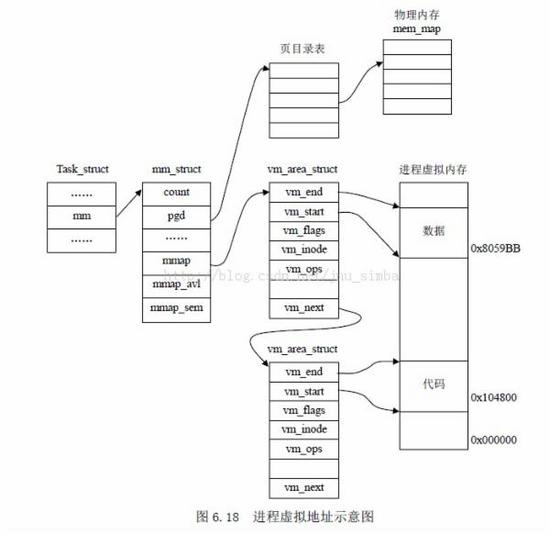
.....

};



* + - 1. 内存管理结构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 。进程的 mm\_struct 则包含装入 的可执行映像信息以及进程的页目录指针pgd。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 。



runqueue;

list\_head;

* + 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。

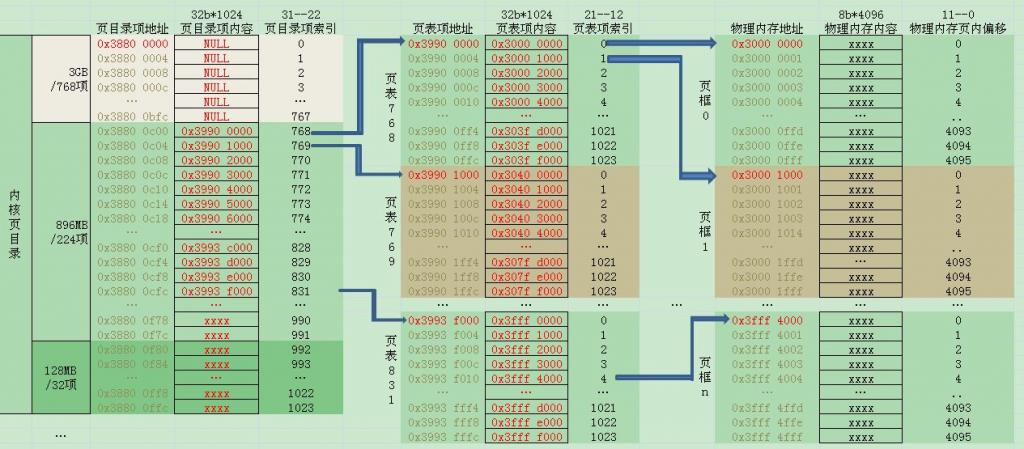
Linux在初始化时，调用pagetable\_init()(位于/arch/arm/mm/init.c)根据物理内存为内核初始化页目录项和页表项。由于内核页表是在内核态中使用，因此只映射0xC0000000 到0xFFFFFFFF的地址空间，即是主内核目录前768项均为NULL。同时内核全局目录的标志位设置及为0x1e3,表明页面的dirty、可读、可写等属性，含义如下：

/\**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了。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提供了宏操作\_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. 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文件系统除了根节点/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了。

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，如果发现了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list \_tasks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list \_tasks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ll /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可以认为是该进程是隐藏进程，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。

* + 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\_tasks，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。

* + - 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

* + - 1. runqueue\_tasks获取
  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，在分析了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，所有还需要检测出可疑的进程然后重点分析他们，并判断其所对应的软件是否是恶意软件。

* + 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

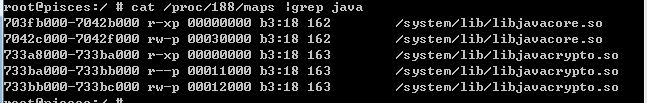
从3.3.1的恶意代码分类可以看出，恶意应用在进程恶意操作的时候需要使用到移动终端的电话、短信、网络等模块，大多数情况下都是在隐蔽或者欺骗用户的情况下执行。而Android应用中，这些权限的获取都是通过Android体系结构中的应用层框架(Application Framework)来提供的,需要调用特定的类来获取系统的底层支持，一般是以动态链接.so文件来提供。

同时恶意应用为了避开手机中的安全软件的检测特别是静态检测，会加入一些特殊的手段来完成应用程序的防检测机制，包括混淆、加壳、动态加载dex技术等，详情在2.3中有描述。这些行为都需要额外的代码来实现，一般也是通过动态链接库来完成。

* + 1. 可疑进程检测方法

在/proc/${pid}/maps里面可以看到指定进程所有的动态链接库，如果找到特定的动态链接库文件，包括特殊的系统动态连接库和自定义的动态连接库，那么这个进程就有可能是恶意软件，当作可以进程来并进行后续的检测查看其是否是真正的恶意软件。

/proc/${pic}/maps不仅提供了指定进程所加载的动态链接库，还指定了这些链接库对应在进程空间的虚拟地址、文件权限等。



1. 基于概率神经网络的恶意软件检测
   1. APK反逆向分析
   2. 高危API提取
   3. Android恶意软件分类
2. 恶意软件取证系统的实现
   1. 环境搭建
   2. 模块设计
      1. 内存镜像获取
      2. 隐藏进程检测
      3. 恶意软件检测
   3. 系统测试
      1. 隐藏进程测试及分析
      2. 恶意软件检测及分析
3. 总结与展望