東南大學

毕业设计(论文)报告

题　目基于 Android 平台的内存取证方法的研究与实现

高等理工实验班　院（系）信息工程　专业

学　　号 61312101

学生姓名 白 岚

指导教师 胡爱群

起止日期 2016 年 2 月 21 日 ~ 2016 年 6 月 5 日

设计地点 无线谷六号楼

目录

图片目录

摘要

Android 操作系统是目前最为市场份额最高的移动端操作系统，Android 平台应用的下载量与使用频率也在持续增高。随着 Android 的流行，与之相关的违法犯罪行为与各类纠纷不断增多，因此，对于 Android 平台的取证研究的需求也越发强烈。Android 内存的取证可以获得从非易失性存储器中所不易获得的重要信息，如用户密码、被删除的用户数据等，因此 Android 内存取证在取证学科中有着重要的意义。本文将对 Android 平台内存取证进行研究，采用了通用性高的进程内存取证的方法，以求高效、准确地对 Android 设备进行内存取证。

首先，本文分析了 Android 平台内存取证的研究背景和意义，对现存的 Android 内存获取与分析的方法进行了详细的讨论，并指出了它们存在的问题。

其次，本文研究了 Android 内存取证的主要原理与技术，包括 Android 系统的基本架构、Android 进程与内存、以及内存中信息的编码与重要信息匹配时所需的关键技术。

最后，本文着重研究了 Android 内存取证的实现。本文首先对 Android 系统的传统进程内存获取方法进行了优化，大大提高了 Android 系统内存获取的效率，使得内存镜像的读取时间降至传统方案的 15%以内、转存时间与所占空间降至了传统方案的 5%以内；随后对邮件、即时通信、支付三类应用的进程内存进行了分析，并在成功获取了在用户界面和非易失性存储器中无法获取的用户数据；最终设计了一款 Android 内存取证软件，克服了传统的基于命令行的取证工具学习周期长的缺点，降低了内存取证工作的难度。

关键词：Android 内存取证 进程 应用 用户数据

Abstract

Among the mobile operating systems, Android OS has the highest market share. In recent years, Android Apps are downloaded and used more and more frequently. As Android becomes popular, the number of infractions, crimes, and disputes related to Android is increasing. Therefore, researches on Android forensics are demanded eagerly. Some vital information which cannot be accessed in non-volatile memory, such as password and deleted user profiles can be achieved by Android volatile memory forensics. Thus Android volatile memory forensics is of great significance in Forensics Science. In this paper, we will research on Android volatile memory forensics. By forensics of virtual memory of processes, we dump and analyze the volatile memory of Android devices efficiently and accurately.

First, the paper presents the background and significance of Android volatile memory forensics, and discusses existing methods of Android volatile memory dump and analysis.

Second, the paper introduces related principles and techniques, including the architecture of Android OS，Android processes and memory, character encoding methods in RAM and regular expression, which is the crucial technique for matching information in RAM.

At last, the paper emphasizes on the realization of Android volatile memory forensics. The paper improves the traditional methods for memory dump of Android processes and significantly enhances the efficiency. The time for reading the memory of our scheme is decreased to less than 15% of that of traditional method, and the time for transfer the memory image and the size of the image is decreased to 5% of the traditional schemes. Then, the memory images of three kinds of Apps—email, instant message, and payment, are analyzed, through which we get user data which cannot be find by user interface or non-volatile memory. Finally, a softeware for Android memory forensics is designed, which makes users can dump and analysis memory of Android device more easily than using the command-line-based tools.

Key words: Android, Forensic, Process, App, User Data

第一章. 绪论

1.1. 引言

Android 是目前市场占有率最高的操作系统，它连续多年在市场份额上远远领先于其他系统，并有长期保持其地位的态势。随着更多的搭载 Android 系统的手机和平板电脑的发售，Android 设备的持有量仍在不断增加，同时，Android 应用仅在 Google Play Store 的下载量就两倍于占有率第二的 IOS 应用下载量。因此，Android 系统的广泛使用和其预计中的增长、以及 Android 应用的大量下载，使得对 Android 平台及应用的取证的研究变得尤为重要。

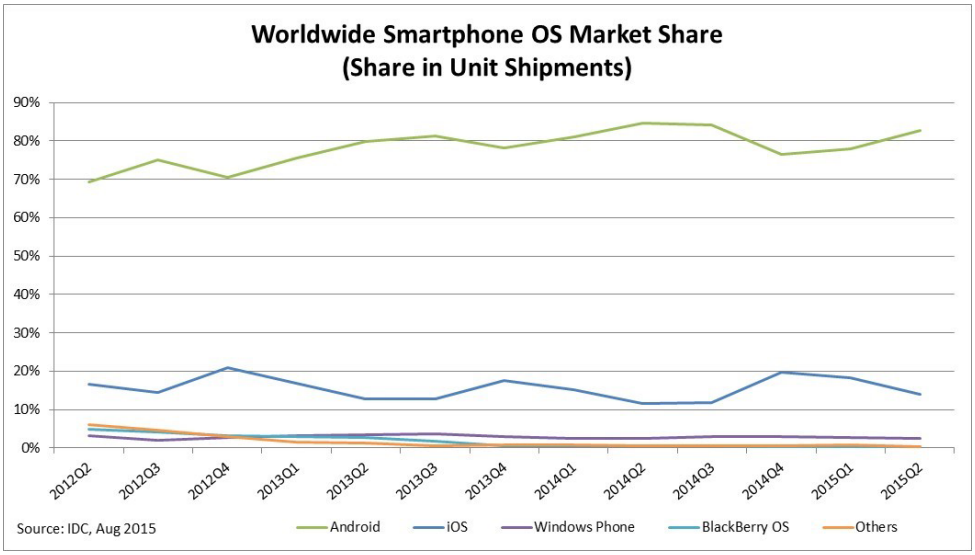

图 1.1.1 全球智能手机操作系统市场占有率

智能手机中所储存的信息，比如聊天记录、邮件、通信记录等等，可以被用作调查中的证据。通常来说，获取这些信息的手段主要是对手机中的非易失性存储空间（NAND flash）进行取证。但是，这一取证方法难以获得某些易失性数据，如用户名与密码、应用数据、已被删除的临时数据等等，因此，我们需要考虑对智能手机进行内存取证。

内存取证是计算机取证科学的重要分支，是指从物理内存和交换文件里查找、提取和分析易失性证据，是基于传统文件系统取证的补充。当计算机或智能手机系统处于活跃状态时,易失性内存中保存着关于系统运行时状态的重要信息,比如口令、打开文件、进程信息、网络连接、系统状态信息等，以及一些临时数据文件。进行易失性内存取证,就是通过对物理内存和页面交换文件进行提取、拷贝,并在另一台计算机或中对获取的数据进行分析,重构出原先系统中的信息。

因此，通过内存取证，我们极有可能获取一些用户在 Android 平台应用中留下的敏感信息，包括登录信息（用户名、密码等）、已经删除的聊天记录与邮件等，这些信息有时难以通过对非易失性存储空间的取证来获得，所以内存取证对于调查者来说有着极为重要的意义。

介于 Android 平台的广泛使用、Android 应用的大量下载，以及内存取证的有效性和重要性，本文我们将专注于 Android 平台的内存取证。

1.1. 本文研究背景

内存取证主要分为两步：内存镜像获取、对获取的物理内存镜像进行分析。本节也将从

这两方面介绍当前的研究背景。

1.1.1. Android 手机内存镜像获取研究现状

本节介绍几种当前或曾经较为流行的 Android 内存镜像获取方法，他们各有优劣。

1.1.1.1. /dev/mem 内存镜像获取接口

在被禁用之前，/dev/mem [1]是被广泛使用的内存镜像获取接口。在有 root 权限的前提下，这个接口允许人们直接通过 dd 等命令直接读写 RAM。

但是，/dev/mem 为经验不够丰富的调查者提出了一些挑战。首先，很多设备的内存映射并不是从物理内存的零偏移量开始连续映射，这就使调查者可能无意中访问到一些敏感区域，从而导致整个内存或计算机系统的崩溃。另一个问题就是通过/dev/mem 只能访问前 896MB 的物理内存空间，这对于现在的取证工作来说是远远不够的。

介于/dev/mem 的局限性，及其因安全性被禁用的背景，研究者需要开发新的内存获取手段。

1.1.1.2. Fmem 内存镜像获取工具

fmem[1]是在 Intel 架构的计算机上广为使用的内存取证工具，它通过创建/dev/mem 特征设备，使用户可以通过 dd 等命令来获取物理内存镜像。

关于 fmem 获取通常为 ARM 架构的 Android 设备内存镜像的效果，[2]中进行了测试。遗憾的是，因为基于 ARM 架构的 Android 系统缺少实现 fmem 中必要步骤的函数。同时，Android 系统中的转存命令 dd 对于地址的编码操作与 Linux 中有所不同，从而导致错误，这使 fmem 无法正确获取 Android 内存镜像。

1.1.1.3. LiME 物理内存镜像获取工具

LiME（Linux Memory Extractor）[3]是目前 Linux 与 Android 设备完整内存镜像获取较为流行的方法。LiME 的工作原理是：

1. 对内核的 iomem_resource 结构进行语法分析，得到系统内存的物理内存地址；
2. 对于内存的每个页面，进行物理内存地址到虚拟内存地址的转换；
3. 读取所有范围中的页面并把它们写入到 SD 卡的文件中或者通过 TCP 套接字传输。

当载入模块时，调查者提供一个路径名来存储内存镜像，或提供一个 TCP 接口来侦听。物理地址范围信息被内核模块直接处理、内存镜像直接从内核中读出，以减少与用户空间的交互，并且减少用户空间拷贝命令，比如 dd 命令[4]的使用。这省去了大量的系统调用 dd 和 cat 时需要的内核活动。这个模块同样尝试避免使用内核文件系统缓冲器和网络缓冲器，以最小化内存在获取过程中所受到的污染。

但是，LiME 的使用非常不便，首先需要绕过内核检查机制，其次必须获取内核源码，对其进行交叉编译后导入。尽管 Android 操作系统是开源的，但是手机厂商往往会对系统进行定制，并且不一定公开定制过的内核源码，使 LiME 在这些手机上不可用。这也促使我们去寻求一些更为通用的内存镜像获取方法。

另外导入内核源码的过程可能改变手机原先的状态，使得调查者无法获取需要的用户数据。

1.1.1.4. AndroidSDK 的堆内存查看工具 DDMS

DDMS[5]全称为 Dalvik Debug Monitor Service，是一款 AndroidSDK 中的进程 Dalvik 虚拟机的堆栈查看工具。它通过 ADB 获得进程 id，并连接到虚拟机调试器，分配计算机 8600 及之后的端口监控虚拟机，可以获取虚拟机 heap 的拷贝。DDMS 作为 Android 开发环境的一部分，并不轻便易用，更适合开发者调试，而不十分适用于取证。

1.1.2. Android 内存镜像分析研究现状

1.1.2.1. Volatility 物理内存镜像分析工具

Volatility[2]是一款对完整物理内存镜像分析的专业软件，能够进行 Windows、OS、Linux 等多种系统的内存分析。它可以将基于 ARM 设备中的虚拟地址转换为物理地址，一旦 ARM 地址空间转换成功执行，我们就能够使用 Volatility 机制下的 Linux 插件来对 LiME 获取的 Android 内存进行分析，从而获得内存映射、打开的文件、网络连接以及类似/proc/iomem 的信息。另外，Volatility 中的插件可以用于用户态与内核中的恶意程序的查找与分析。但是 Volatility 并不长于 Android 应用用户数据的恢复。

1.1.2.2. Eclipse 的内存分析器 MAT

MAT[6]全称 Memory Analyzer tool，是 Android 的重要开发工具 Eclipse 的内存分析器，是一种快速，功能丰富的 Java 堆分析工具。

DDMS 获取的堆栈信息经过转换：

./hprof-conv xxx-a.hprof xxx-b.hprof

可以被 MAT 识别。MAT 可以分析内存占用状态、内存单元所表示的数据类型等信息，并支持 SQL 语句搜索内存数据。通过 MAT 可以获得一些数据信息，但是 MAT 有与 DDMS 类似的缺陷——适用于开发者调试，需要一定的使用技巧，对普通取证者不十分友好。

1.1.3. Android 平台内存取证的问题与前景

由于Android平台内存取证的重要价值以及当前发展的局限性，一些研究者也在Android内存取证进行实验与探讨。

在[7]中，作者通过 DDMS 和 MAT 的结合分析了 Facebook、Twitter 和 Gmail 等应用的内存，已获得用户名和密码等信息。这种方法有一定的可行性，但是工具过于庞大，且主要用于开发者分析内存泄露，在内存取证方面缺少针对性，使用起来对于取证工作者来说略显繁琐。在[8]中，作者首先通过 LiME 获取了完整物理内存镜像，随后对应用“微信”的内存进行分析。尽管在论文中给出了分析内存镜像从而得到聊天记录的结果，但是由于缺少必要的细节，这一结果难以被重现而投入到实际的取证应用中。

因此，我们需要一款轻便易用的 Android 平台内存取证工具。首先，这款工具需要在取证方面具有针对性，专注于通过内存取证获取其他取证手段难以获取的用户数据。其次，这款工具需要对并无太多软件开发经验者友好。第三，这款工具要有一定的通用性，只要设备获取了 root 权限就可以使用。第四，这款工具要有高效率，让取证者能够快速完成取证。最后，要给出一些内存镜像分析的方法，使研究结果可重现，对其他的研究者有参考意义。

1.2. 本文的主要工作

本文将对 Android 系统进程内存的获取与分析进行研究，并尝试构建一个一键获取-分析进程内存的工具。由于条件的限制，本文中的主要实验将在目前市场占有率最高的 Android

版本——Android 4.4（kitkat）上进行，选用的测试手机型号为 HUAWEI P6-C00。

本文一共分五个章节，主要内容如下：

第一章主要分析了 Android 平台取证的重要性与研究现状，介绍了多种当前 Android 设备内存获取与分析的手段，并指出了 Android 平台取证的前景以及主要需解决的问题。本章在最后说明了本论文的主要内容和重点。

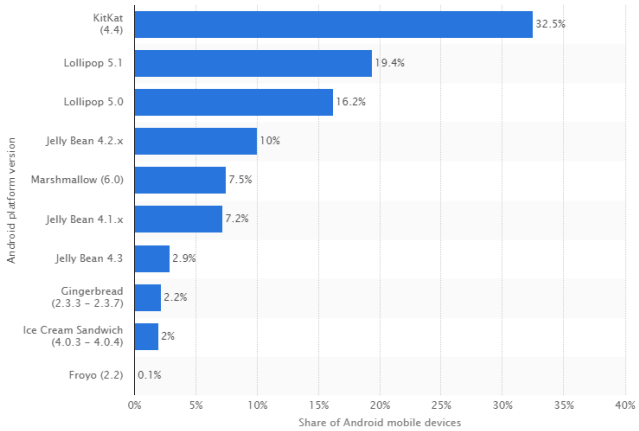

图 1.2.1 Android 各版本市场占有率

第二章研究了 Android 内存镜像的获取和分析的技术原理。首先，该章介绍了 Android 系统的基本架构以、Android 进程及其内存管理和本文中所用到的高通用性的 Android 内存获取方法的基本原理，随后详细介绍了编码与正则表达式等内存数据分析的必要背景知识

第三章是本文作者的 Android 内存取证实现方案。该章首先介绍了本文所采用的内存镜像获取方案，并将实现效果与采用相似原理的传统方案相比较，体现出本文方案效率的提高。随后，该章介绍了本文作者对 Android 内存镜像的分析。本文的主要目的在于通过内存镜像来获取 Android 平台应用的用户数据。由于不同的应用的用户数据在内存中以不同的结构保存，因此本章首先对一些典型的应用内存镜像中的数据进行分析，随后对其中部分开发了自动化的内存数据分析方法。该章还展示了开发的 Android 内存取证软件。该软件具备检测手机状态、一键获取内存镜像，一键提取有效数据等功能。轻便易用。

第四章对全文内容进行了总结，并给出了 Android 内存取证的进一步研究方向。

第二章.　Android 内存取证技术原理概述

2.1. 引言

本章主要讨论 Android 内存取证技术的主要原理，作为本文进一步深入研究 Android 平台内存取证的基础。本章将首先简要讨论 Android 系统架构，剖析其内存分配机制。随后将简要阐述/proc 伪文件系统和 PTRACE 系统函数，介绍本文获取内存镜像所依靠的用户态接口。最后讲述了字符编码的基本方式与正则表达式，为分析 Android 内存镜像提取信息提供了理论指导和有效工具。

2.2. Android 系统基本架构

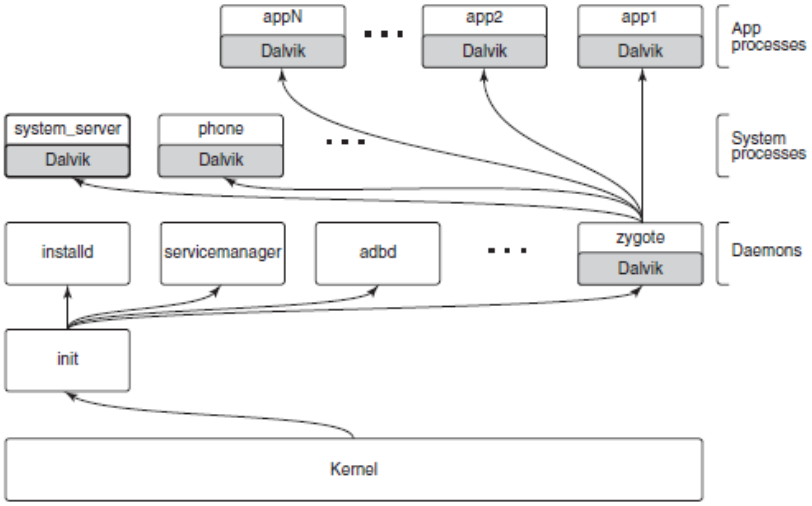

图 2.2.1 Android 系统基本架构

Android 是基于标准 Linux 内核的操作系统，但是在用户态的实现上，Android 又与传统的 Linux 发行版有许多差别[9]。与传统的 Linux 一样，Android 的第一个用户空间进程是 init。然而，Android init 后台进程的启动更关注底层，比如说文件系统管理与硬件接入，而不是上层，比如工作调度。与 Linux 进程相比，Android 进程有一个新加入的层 Dalvik 虚拟机（在 4.4 版本后 ART 成为一种新的可选的虚拟机），Dalvik VM 中运行的是 Android 系统中所有以 Java 实现的部分。图 2.2.1 展示了基本的 Android 进程结构。最先启动的是 init 进程，这一进程分裂出了一系列底层后台进程。其中之一是 zygote，这是所有高级语言 Java 所编写的进程的基础。

在 Android 中，每个应用都在他自己的进程中运行，每个进程都有自己的 Dalvik 虚拟机。当一个应用开始运行以后，它所对应进程的 Dalvik 虚拟机会从操作系统中获取部分内存。Dalvik 的堆（heap），会由 zygote 预先载入。对于 Java 语言，堆内存主要放置对象实例。每个 Dalvik VM 会使用堆来进行垃圾收集，其中包含 Java 对象存储的有效数据。在下一章中，我们将对 Android 进程及其内存管理机制进行详细的介绍。

2.3. Android 进程及进程的内存空间

由于本文中的取证工作将针对 Android 进程内存，本节将对 Android 内存进程及其内存空间进行介绍。

2.3.1. 进程内存空间与 RAM 之间的关系

进程的内存空间是虚拟内存，而程序的运行需要依赖于 RAM（物理内存）。如图 2.3.1 所示，程序运行时申请虚拟内存,操作系统将虚拟内存映射到 RAM。

RAM 作为进程运行的必要资源，对系统性能和稳定性有着决定性影响。另外，RAM 的一部分被操作系统留作他用，比如显存等等，这些都是由操作系统控制，另外，进程只能操作虚拟地址空间，无法直接操作 RAM。

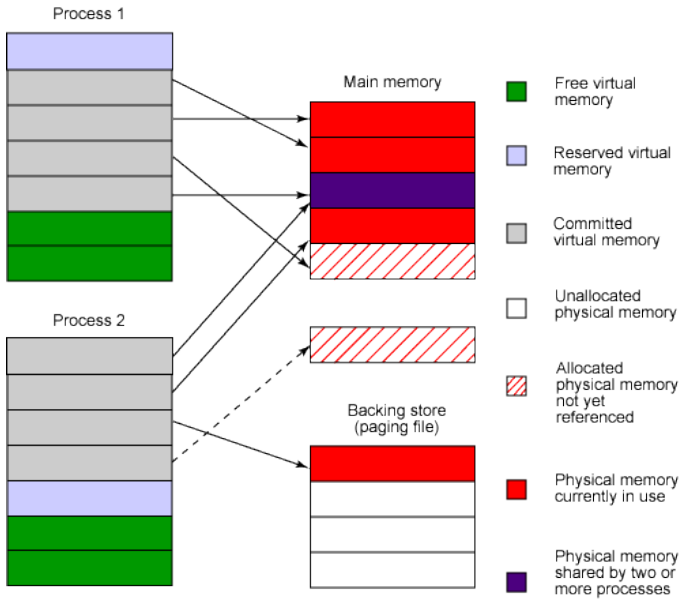

图 2.3.1 进程内存空间与 RAM 之间的关系

2.3.2. Android 中的进程

Android 进程按编写语言和是否包含 Dalvik 实例可以分为两大类：

1. Native 进程：这是采用 C/C++实现、不包含 Dalvik 实例的 Linux 进程，如/system/bin/ 目录下面的程序文件运行后都是以 Native 进程形式存在的。

2. Java 进程：进程入口主函数为 Java 函数、包含实例化的 Dalvik 虚拟机实例的 Linux 进程。Dalvik 虚拟机实例一个以系统函数 fork()创建的 Linux 进程为宿主进程，因此，Android 的 Java 进程实际上就是一个有 Dalvik 实例的 Linux 进程。与 Native 进程相比，Java 进程的内存分配与管理更加复杂。Android 中的应用程序基本上都是此类进程，如桌面、电话、联系人，以及我们常用的微信、QQ 等等。对于这些进程，如果内存不足，而又有其他为更紧急的进程需要更多内存，Android 系统可能会决定关闭一个进程。在该进程中运行的应用程序组件也随之被销毁。当这些组件需要再次工作时，会重新创建一个进程。在决定关闭哪个进程以腾出更多内存空间时，Android 系统会考虑正在运行的进程对用户的重要程度。例如，相比于一个拥有可见 activity 的进程，Android 系统更有可能去关闭一个 activity 已经在屏幕上不可见的进程。这意味着，是否终止一个进程，取决于运行在此进程中组件的重要程度。依此，这些进程可以被分为如下几类[10]：

   a) 前台进程（Foreground）

      用户当前操作必须的进程。满足下面任一条件时，进程认为是前台进程：其中运行着正与用户交互的 Activity;其中运行着绑定于正与用户交互的 Activity 的服务 Service;其中运行着前台服务 Service 服务，该服务以 startForeground()方式被调用；

其中运行着正在执行生命周期回调方法（onCreate()、onStart()或 onDestroy()）的服务 Service;其中运行着正在执行 onReceive()方法的 BroadcastReceiver。

一般情况下，在任何时刻，前台进程都不是多数，只有当内存不能够维持它们同时运行时才会被终止。通常，Android 在终止这些进程时已经到了内存分页状态(memory paging state)，终止某些前台进程是为了保证用户界面不会停止响应。

b) 可见进程(Visible)

可见进程是没有前台组件、但对用户在屏幕上所见内容有所影响的进程。可见进程的条件是：其中运行着不在前台的、但是被 onPause()方法调用的、用户仍然可见的 Activity。比如说前台 Activity 弹出了一个对话框，然而之前的 Activity 还被允许显示在后方；或者其中运行着绑定于前台 Activity 的服务 Service。可见进程也是很重要的进程，只有在无法维持所有前台进程同时运行的情况下才会被终止。

c) 服务进程(Service)

这类进程运行着 startService()方法所启动的服务，它不会升级为前台或可见进程。服务进程一般不直接被用户所见，但他们往往在执行一些用户关心的操作，例如后台播放 MP3 或者上传、下载程序。在系统无法维持所有可见和前台进程时会被终止。

d) 后台进程(Background)

后台进程是当前用户不可见的，即其 ActivityonStop()方法已被调用的进程。这些进程对用户体验影响较小，在任何时刻系统都有可能终止他们，来为前台进程、可见进程和服务进程提供内存资源。

e) 空进程(Empty)

空进程不包含任何的活动应用程序组件。其目的是被当作缓存（Cache），来减少下次在该进程中运行组件的启动时间。空进程是对进程进行一种资源的预留，但是系统需要在进程缓存和内缓存之间平衡资源，所以会经常终止空进程。

以上的各种进程，入口主函数都是由 Java 编写，因此，要了解这些进程的内存管理机制，就要了解 Java 的内存管理与垃圾回收机制。

2.3.3. Java 内存管理与垃圾回收机制

Android 中，每个 Java 进程的 Dalvik 虚拟机的内存管理与垃圾回收机制，与 JVM（Java Virtual Machine）类似。在 Java 中，通过 new 关键字来为对象分配内存空间，由垃圾收集机制来回收内存空间。

2.3.3.1. JVM 结构

图 2.3.2 是 JVM[11]结构图，Android Dalvik VM 的结构与之类似。它主要由四个部分组成：Class Loader 子系统、Execution Engine 执行引擎、Runtime Data Area 运行方法区以及 Native Interface 本地方法区，我们主要介绍 Runtime Data Area 运行方法区，也就是通常所说的 JVM 内存。由图可见，Runtime Data Area 运行方法区主要由 5 个部分组成：

Method Area: 存储被装载的 Class 的元信息，是线程共享的；

Heap：堆空间。一个 JVM 实例中只存在一个堆空间，存放对象信息，是线程共享的；

Java Stack：栈。JVM 直接对 Java 栈进行两种操作，以帧为单位的出栈和压栈，非线程

共享；

Program Counter register：程序计数器，非线程共享。

Native Method Area：本地方法栈，非线程共享。

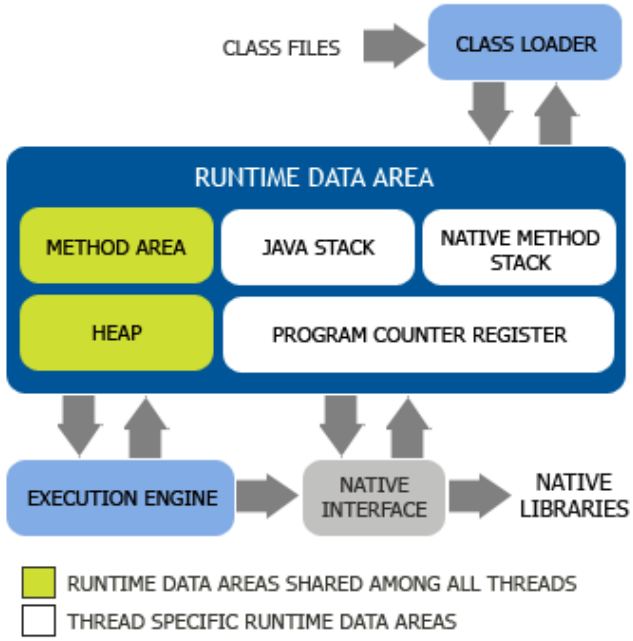

图 2.3.2 JVM 结构图

2.3.3.2. JVM 的垃圾回收

JVM 的垃圾回收把对象按生命周期分为年轻代（Young）、年老代（Tenured）、持久代（Perm），并对不同生命周期长度的对象调用不同的垃圾回收算法。

1. 年轻代(Young)

年轻代分为三个区，其中一个 Eden 区，两个 Survivor 区。Java 程序里生成的大部分新对象都在 Eden 区中，当 Eden 区被占满时，仍存活的对象被复制到其中一个 Survivor 区，当此 Survivor 区的空间被占满时，此区的存活对象又被复制到另外一个 Survivor 区，当这个 Survivor 区也满了的时候，从前一个 Survivor 区复制过来的此时依然存活的对象，被复制到年老代。

2. 年老代（Tenured）

年老代存放的是上文年轻代复制过来的对象，也就是在年轻代中依然存活的对象，且是在 Survivor 区满了以后复制来的。一般来说，年老代里，对象生命周期都比较长。

3. 持久代（Perm）

存放静态的类和方法，对垃圾回收没有显著的影响。

2.4. Android 进程内存空间的访问与读取

2.4.1. /proc 伪文件系统

/proc [12]是一个继承自 Linux 的伪文件系统，它实际并不占用存储空间。它被用作一个访问内核数据结构的接口。/proc 文件夹下的绝大多数部分是只读的，但是一些文件也用于

改变内核变量。Android /proc 文件夹下的大多数内容需要以 root 用户权限来访问。在本文中，我们主要关注/proc/<pid>/maps 与/proc/<pid> /mem 文件。

/proc/<pid>/maps 文件中包含了关于当前已经被映射的虚拟内存区域的相关信息。基于 Linux 2.0 之后版本的内核的操作系统中，该文件基本形式如下：

```
Address           perms offset  dev  inode      pathname
8cbd4000-8ccd1000 rw-p 00000000 00:00 0
8cd00000-8ce00000 rw-p 00000000 00:00 0
8ce00000-8cf00000 rw-s 00000000 00:04 25961  /dev/ashmem/MemoryHeapBase (deleted)
bea7d000-bea9e000 rw-p 00000000 00:00 0          [stack]
ffff0000-ffff1000 r-xp 00000000 00:00 0          [vectors]
```

其中第一列 Address 是进程所占的地址空间，perms 中各个字母所表示的意思分别为：

r = read

w = write

x = execute

s = shared

p = private (copy on write)

offset 为在进程地址里偏移量，dev 为映射文件的主次设备号，inode 是的映像文件的节点号。Pathname 是映射文件的路径名，其中包含如下几个字段的需要特别关注:

heap:堆，存储 Java 对象实例；

stack:栈，存储函数与基本数据类型；

(deleted):表示相应页面在用户空间已经移除，但是仍存在于内存中。

本项目中，我们通过/proc/<pid>/maps 获取所需求内容在虚拟内存中的位置，并通过 /proc/<pid> /mem 访问虚拟内存页面，从相应位置读取内存镜像，对内存镜像的不同部分进行分别存储。

这一方法对基于 Linux 内核的操作系统普遍有效，因此在 Android 设备上具有通用性，并且能将进程内存中较为重要的部分提取出来以供分析。

2.4.2. ptrace

ptrace[13]是 Linux 的一个用户态的调试接口，在 Android 等 Linux 内核系统中，我们可以通过它获取运行进程的全部内存页面，包括进程的 Dalvik VM 的堆内存页面。

ptrace 系统函数提供了一种让一个进程（跟踪者）观察和控制另一个进程（被跟踪者）的执行的方法，并且跟踪进程能检测和改变被跟踪进程的内存核寄存器。最初，它被用于断点调试和系统函数跟踪。

首先,被跟踪者需要附加于跟踪者，这一行为是对单个线程的操作，对于多线程的进程，每个线程会被单独地附于跟踪进程。因此，一个被跟踪者是一个线程，而不是一个（多线程的）进程。

ptrace 的通常形式为：

ptrace(PTRACE_foo, pid, ...)

其中，pid 是需要被跟踪的相应线程的 id。第一个参数 PTRACE_foo 决定了 ptrace 的运行模式。

PTRACE_foo 参数可取值有：

PTRACE_ME

PTRACE_PEEKTEXT

PTRACE_PEEKDATA

PTRACE_PEEKUSER

PTRACE_POKETEXT

PTRACE_POKEDATA

PTRACE_POKEUSER

PTRACE_GETREGS

PTRACE_GETFPREGS,

PTRACE_SETREGS

PTRACE_SETFPREGS

PTRACE_CONT

PTRACE_SYSCALL,

PTRACE_SINGLESTEP

PTRACE_DETACH

本文主要关注的是 PTRACE_ATTACH 和 PTRACE_DETACH。通过 PTRACE_ATTACH 和 PTRACE_DETACH,可以实现进程的附加和分离。

PTRACE_ATTACH 可以通过 pid 指定一个进程，让他被当前调用的进程跟踪。这个跟踪进程会收到一个 SIGSTOP 信号，但是不一定会随着这个信号马上就停下来。我们需要使用 waitpaid 来等待被跟踪进程停止。具体实现如下：

ptrace(PTRACE_ATTACH, target_pid, 0, 0);

waitpid(target_pid, NULL, 0);

PTACE_DETACH 使原先被停止的被跟踪进程重启。在 Linux 内核系统中，用这种方法可以使一个被跟踪进程与跟踪分离，无论这个跟踪的过程是如何被初始化的。具体实现方法如下：

ptrace(PTRACE_DETACH, target_pid, 0, 0);

2.4.3. 通过/proc 与 ptrace 访问 Android 进程内存空间

由 2.4.1 节可知，用户可以通过/proc/<pid>/mem 访问进程虚拟内存。然而，对/proc/<pid>/mem 的访问需要先暂停并跟踪相应的进程，这就需要使用 ptrace 系统函数。具体步骤为：

1. 使用 ptrace(PTRACE_ATTACH, target_pid, 0, 0)跟踪目标进程；
2. 通过 waitpid(target_pid, NULL, 0)等待目标进程停止；
3. 读取/proc/<pid>/mem 中的虚拟内存；
4. 完成读取后，通过 ptrace(PTRACE_DETACH, target_pid, 0, 0)结束跟踪，并重启被

跟踪的进程。

这一访问和读取内存的方法，只要可以访问/proc/<pid>/mem 并调用 ptrace 系统函数即可实现，因此，任何拥有 root 权限的用户均可以以此方法访问和读取进程的虚拟内存。这也是 Android 平台获取内存镜像的一个有通用性的方法。

2.5. 内存信息常用字符编码

在获得内存镜像之后，我们可以对其进行分析。获得的内存镜像是一个二进制文件，要从中提取信息，就要知道这些信息可能采取何种编码。本节将介绍几种常用的编码方法，以作为在内存分析时的参考。

2.5.1. ASCII

ASCII（American Standard Code for Information Interchange，美国标准信息交换代码）[14]是一套基于拉丁字母的计算机编码系统，是现今最通用的单字节编码系统，

ASCII 码使用 7 或 8 位二级制数来表示 128 或 256 种字符。标准的 ASCII 码使用 7 位二进制数来表示大小写字母、数字、标点、控制字符等等。下面简要介绍这 128 个字符的分配：

0~31、127 是控制字符和通信专用字符，其余是可显示字符。控制符如换行、回车、退格、删除等等。通信专用字符，如文件头、文件尾等等。这些字符没有固定的图形显示，但是在不同的应用程序中会对文本显示有不同的影响。

在可显示字符中，32 为空格，48-57 为 0-9，65-90 为英文大写字母，97-122 为英文小写字母，其余为标点与运算符号等等。

ASCII 作为美国标准，很难满足所有国家与语言的需要。对于其他英语国家，如货币符号等就无法显示；对于其他拉丁语语言，重音符号无法显示。而对于本文的研究来说，一些只包含英文字符的信息，如用户名、密码等等，可能以 ASCII 显示。但是，ASCII 对中文等语言的显示不适用，那么，内存镜像中的如聊天记录这样必须包含多种语言的信息，显然会采用其他编码方式。

2.5.2. Unicode 与 UTF-16

Unicode（Universal Multiple-Octet Coded Character Set）[15]是计算机科学领域的一项标准，包括字符集、编码方案等，它为每种语言中的每个字符定了唯一的二级制编码，以满足跨语言、跨平台文本处理的需求，并向下兼容 ASCII。

UTF-16[16]是 Unicode 的一种常见的实现方式，以 16 位无符号整数位单位。UTF-16 有两种不同的字节序，称为 BE（Big　Endian，大端）和 LE（Little Endian，小端）。这是由于不同的操作系统对于编码的理解不同所造成的。如果一个字符的 UTF-16 编码是 6c49，占存储空间 2 字节，UTF-16 BE 编码即为 49 所在字节在前，6c 所在字节在后（496c），而 UTF-16 LE 编码则为 6c49。

采取 LE、BE 何种编码方式需要参考 BOM（Byte Order Mark, 字节顺序标记），即在传输的字符流前，先传输 BOM 字符。这个字符的编码是 FEFF，这个编码与将它两个字节相反的编码 FFFE，在 UTF-16 中都是未定义的码位。UTF-16 BE 对应的 BOM 是 FEFF，UTF-16 LE 对应的 BOM 是 FFFE。在本文对内存镜像的分析中，UTF-16 是一种最为常见的信息编码方式。

2.6. 正则表达式

在内存文件分析中，为了获取某些信息，需要匹配符合某些规则的字符串。正则表达式[17]使用单个字符串来描述、匹配符合某个句法规则的字符串，可以用来检索、替换符合某个模式的文本，如全为英文字母、全为数字、包含某种符号且其他部分全为数字等等。许多语言中都内置了正则表达式引擎。正则表达式的具体规则较为琐碎，在[17]中有详细描述。在本文的研究中，我们可能会使用正则表达式来匹配一些结构上特别具有特征的信息，如邮箱地址等。

2.7. 本章小结

本章对本文中 Android 平台内存取证所需要的基本理论与工具进行了详细的阐述。首先剖析了 Android 系统架构与内存分配，以便在知道内存分配原理下科学地获取内存镜像；随后描述了 Android 内存获取时所需要的系统调用和用户态接口，为内存镜像获取提供了工具；最后介绍了常见的字符编码方式和正则表达式，为内存分析提供了思路。本章为下文讨论 Android 平台内存取证的方案与具体实现奠定了理论基础。

第三章. Android 内存取证的方案与实现

3.1. 引言

在上一章中，我们探讨了 Android 内存取证所依赖的基本原理与技术。在本章中，我们将对 Android 内存取证进行实现，并设计 Android 内存取证软件。

Android 内存取证的基本流程如图 3.1.1。本章将以此流程为基础，对本文所做的 Android 内存取证有关工作进行介绍。为了使我们的软件能够对绝大多数已获取 root 权限 Android 设备进行取证，本章将参照 2.4 节，通过/proc 伪文件系统和 ptrace 函数针对进程获取虚拟内存镜像。同时，本章将考虑内存取证时获取应用用户数据的需求，对传统的进程内存获取方法进行了改进。随后，本章将针对几款不同类型的应用，对他们的进程内存进行了分析。最终，本文作者设计了图形界面，为取证工作者提供了一款简单易用的内存取证工具。

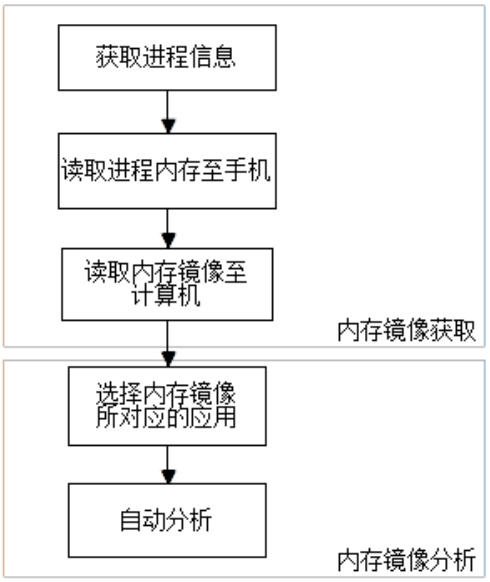

图 3.1.1 本文所设计的 Android 内存取证软件基本工作流程

3.2. Android 内存镜像获取

3.2.1. 内存镜像获取程序设计

对于本文的内存取证工作来说，内存镜像的获取是第一步。本节中，我们将通过编写 C 语言程序来获取内存镜像。内存镜像获取的基本思路如图 3.2.1 所示。具体可视为如下几步：

1. 输入需要获取的内存进程 pid、内存镜像保存位置等信息；
2. 逐行读取/proc/<pid>/maps 文件；
3. 根据本行的 pathname 属性，判断该内存区段是否需要被读取；
4. 通过/proc/<pid>/mem 读取进程虚拟内存（具体方法如 2.4.3 节），储存至目标文件。

开源网站 Github 上的一款获得较高收藏数的 Android 内存取证工具[18]中，作者以类似于图 3.2.1 的思路获取内存镜像。这款软件同样专注于用户数据信息的读取。在根据 pathname 判断内存区段是否需要读取时，设置条件为包含“heap”、“stack”、“deleted”即读取。其中

包含 heap 是与堆相关的信息，包含 stack 是栈相关的信息，包含 deleted 表示表示相应页面在用户空间已经移除，但是仍存在于内存中。这三个信息在目标为获取应用数据的内存取证上确实值得关注。尽管这款软件能够获得 Android 进程包含有价值的取证信息的虚拟内存镜像， 但是由于缺少必要的优化，获取的内存镜像所占空间过大（超过 700MB），且时间过长。

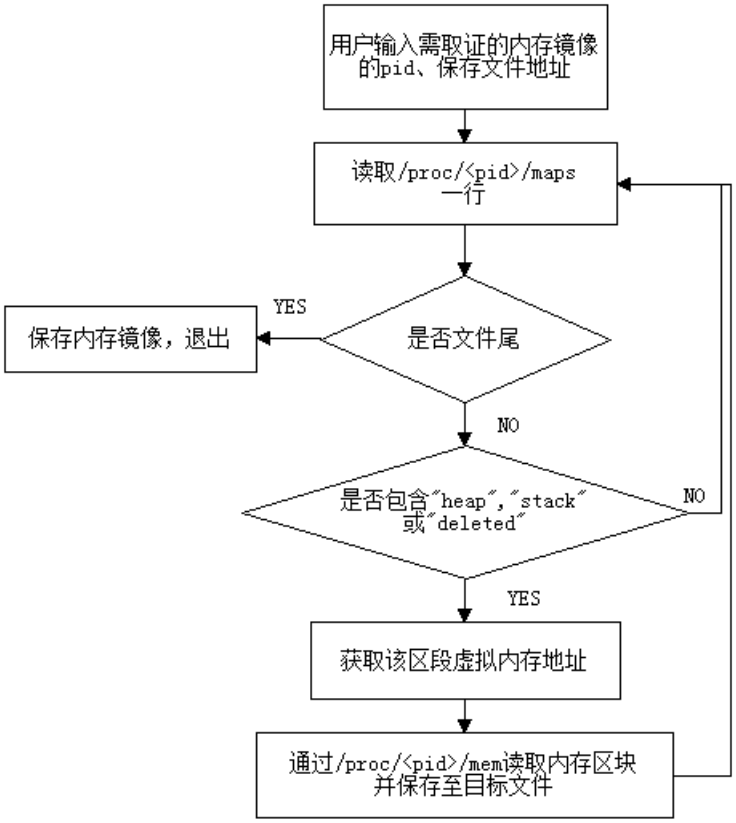

图 3.2.1 获取进程虚拟内存镜像的传统方案流程图

然而，在取证过程中，效率是值得关注的。高效地获取内存镜像是提高内存取证效率的关键一步。对于内存取证来说，可以缩短的时间主要是如下三种：

1. 内存镜像读取时间；
2. 内存镜像传输时间，即将获取的内存镜像拷贝到电脑的时间；
3. 内存镜像分析处理时间

第 1、2 两种效率的提高均与内存镜像的获取直接相关，而就第三种来说，在内存取证目标明确的情况下，如果能适当缩小获取内存镜像的所占空间，也能够大大缩短内存镜像分析处理的时间。因此，良好的内存镜像获取方法能够大大地提升整个内存取证的效率。

显然，[18]中的软件的内存镜像获取方式，对于内存取证的读取时间、传输时间、分析处理时间上都是极大的负担。

本节将对原始的进程内存镜像获取思路进行优化，以提高内存镜像获取效率，缩短取证过程所需的时间。

首先，读取内存镜像是为下一节的数据信息分析与提取服务的。我们要提取的目标数据是明确的、有限的，他们不会在所有的内存区域上出现，因此，我们可以考虑适当地缩小需

要读取的内存镜像的范围。

同时，我们可以通过 pathname 这一参数了解每个内存区段所映射的文件，从而了解其功能。首先，查看/proc/<pid>/maps 如图 3.2.2 所示。

```
6163e000-61e3e000 rw-p 00000000 00:04 3250       /dev/ashmem/dalvik-bitmap-1 (deleted)
61e3e000-61e9a000 r--p 00000000 b3:13 773683     /data/dalvik-cache/system@framework@conscrypt.jar@classes.dex
61e9a000-6269a000 rw-p 00000000 00:04 3251       /dev/ashmem/dalvik-bitmap-2 (deleted)
6269a000-626cf000 r--p 00000000 b3:13 773684     /data/dalvik-cache/system@framework@okhttp.jar@classes.dex
626cf000-626d6000 r--p 00000000 b3:13 773685     /data/dalvik-cache/system@framework@core-junit.jar@classes.dex
626d6000-626f3000 rw-p 00000000 00:04 3261       /dev/ashmem/dalvik-aux-structure (deleted)
626f3000-6271d000 rw-p 00000000 00:04 1730       /dev/ashmem/dalvik-aux-structure (deleted)
6271d000-6273f000 rw-p 00000000 00:04 3272       /dev/ashmem/dalvik-aux-structure (deleted)
6273f000-62740000 r--s 0000c000 b3:10 73759      /system/framework/mms-common.jar
62740000-6d1eb000 rw-p 00000000 00:04 3252       /dev/ashmem/dalvik-mark-stack (deleted)
6d1eb000-6d1ec000 rw-p 00000000 00:04 3259       /dev/ashmem/dalvik-aux-structure (deleted)
6d1ec000-6d27d000 r--p 00000000 b3:13 773690     /data/dalvik-cache/system@framework@hwframework.jar@classes.dex
6d27d000-6d29e000 r--p 00000000 b3:13 773693     /data/dalvik-cache/system@framework@mms-common.jar@classes.dex
6d29e000-6d69f000 rw-p 00000000 00:04 3253       /dev/ashmem/dalvik-card-table (deleted)
6d69f000-6d6a0000 ---p 00000000 00:04 3254       /dev/ashmem/dalvik-LinearAlloc (deleted)
6d6a0000-6d9fa000 rw-p 00001000 00:04 3254       /dev/ashmem/dalvik-LinearAlloc (deleted)
6d9fa000-6db34000 rw-p 0035b000 00:04 3254       /dev/ashmem/dalvik-LinearAlloc (deleted)
6db34000-6e69f000 ---p 00495000 00:04 3254       /dev/ashmem/dalvik-LinearAlloc (deleted)
6e69f000-6e9b8000 r--p 00000000 b3:13 773682     /data/dalvik-cache/system@framework@core.jar@classes.dex
6e9b8000-6ea72000 rw-p 00000000 00:04 1717       /dev/ashmem/dalvik-aux-structure (deleted)
6ea72000-6ea9d000 r--p 00000000 b3:13 773692     /data/dalvik-cache/system@framework@voip-common.jar@classes.dex
6ea9d000-6eaa0000 rw-p 00000000 00:04 3274       /dev/ashmem/dalvik-aux-structure (deleted)
6eaa0000-6eab8000 rw-p 00000000 00:04 4935       /dev/ashmem/dalvik-aux-structure (deleted)
```

图 3.2.2 查看/proc/<pid>/map 文件

可以看出，我们之前读取的内存来自较多不同 pathname 为了了解各个内存区段主要的功能，本文作者以 com. facebook. katana 进程为例，将读取的不同 pathname 的内存区块分别储存，文件名的形式为：进程所对应的应用+pathname。将内存镜像按 pathname 保存至不同文件的程序流程如图 3.2.3 所示：

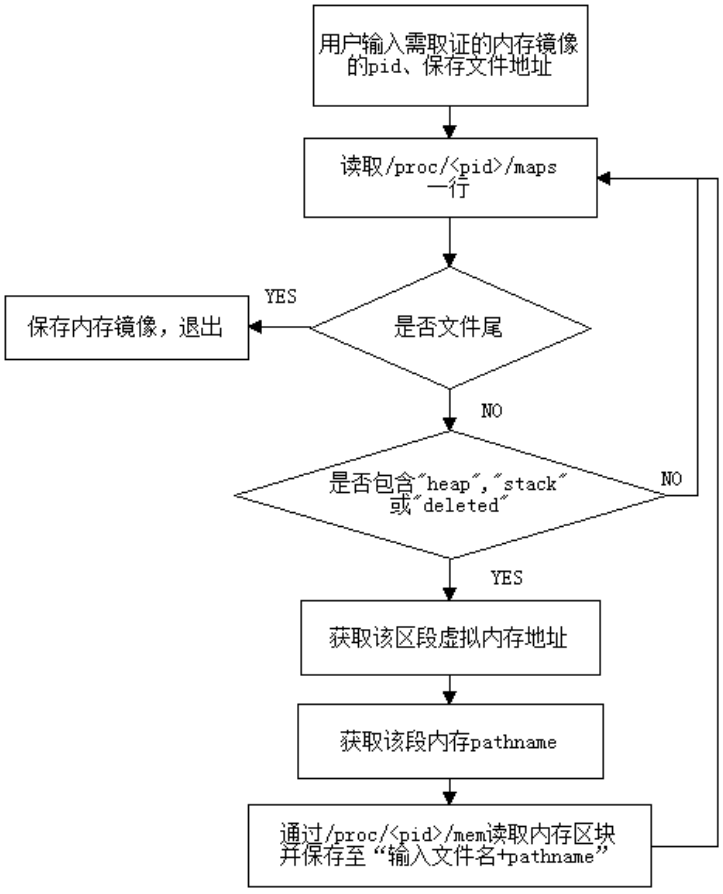

图 3.2.3 将内存镜像按 pathname 储存的基本流程

因为文件名中包含‘/’会让系统认为该文件名是一个路径，因此，实际存储内存镜像文件时，用‘_’代替文件名中的‘/’。获取结果如图 3.2.4 所示。其中，[heap]、[stack]是系统的 Native 堆、栈，包含“dalvik”字段的均与 Dalvik 的堆管理与垃圾收集相关，其中（下面对于文件名包含/dev/ashmem/*的文件，均略去/dev/ashmem）：

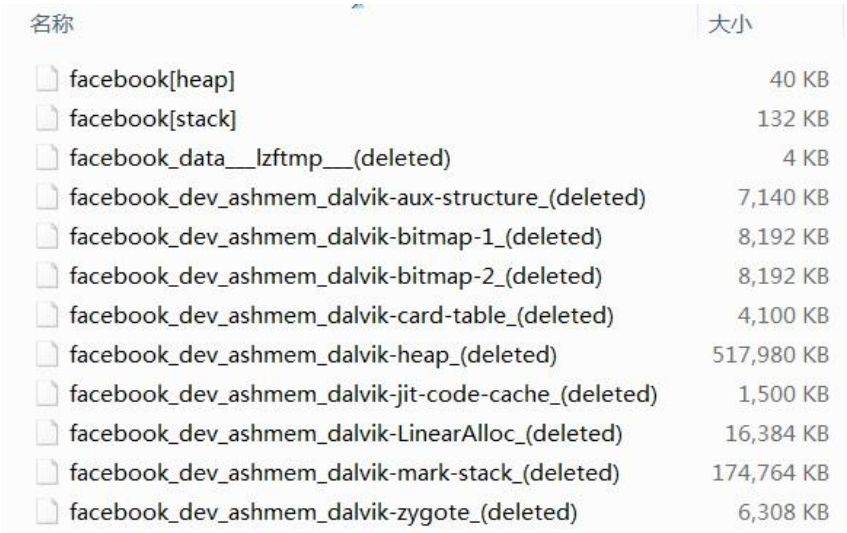

图 3.2.4 com. facebook. katana 内存镜像获取结果

dalvik-bitmap-1 与 dalvik-bitmap-2 是 Dalvik 堆管理的数据结构。在 Dalvik 中， 垃圾收集是一种标记与清除的过程，每一个以 8 字节为一组的内存单元都会在 bitmap 中以一个 bit 标记为使用或释放。两个 bitmap 分别标记在运行时和在 GC（garbage collection，垃圾收集）时被使用的堆内存 dalvik-mark-stack 被用于垃圾收集标记步骤。 这一步骤会对 bitmap 进行迭代，因此这是一种需要栈的广度优先搜索。

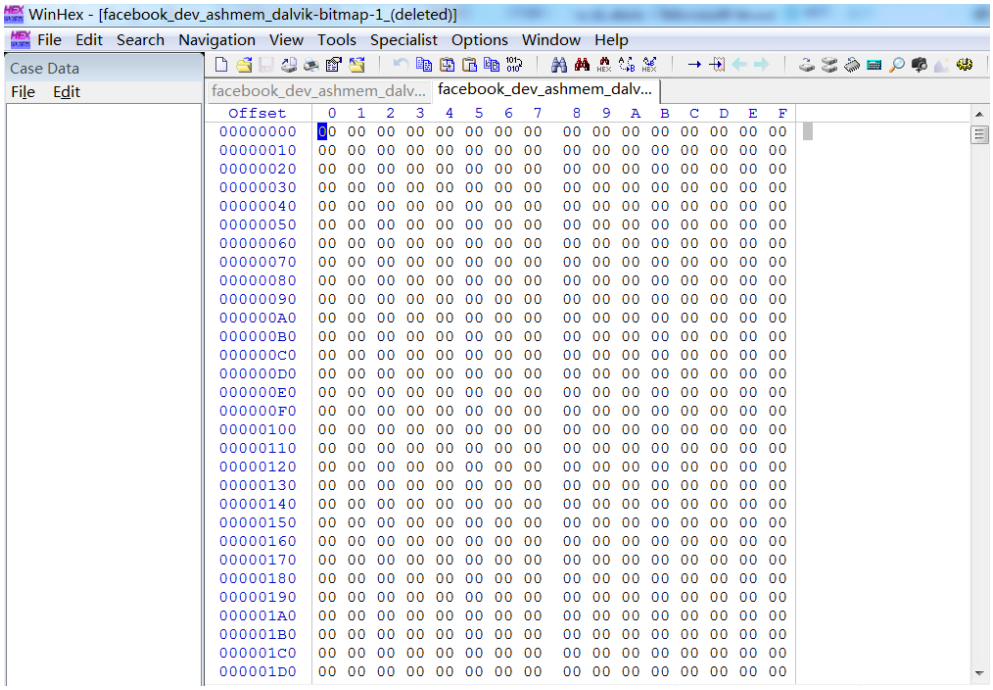

图 3.2.5 用 winhex 查看 dalvik-bitmap-1 二进制文件

dalvik-card-table 被用于同时进行垃圾收集。在标记步骤中，进程会做一些其他工作，也要使用到内存，这些 card-table 被用于记录脏内存页面。

dalvik-heap 为进程使用的堆内存。dalvik-zygote 是一个被所有进程共享的内存区域，主要是一些框架性的资源。dalvik-jit 是 JIT（Just in Time）内存，将字节码转换为机器码。dalvik-LinearAlloc 主要是函数、类定义、线程栈等数据。 dalvik-aux-structure 是一些辅助性的数据，比如一些常量引用。

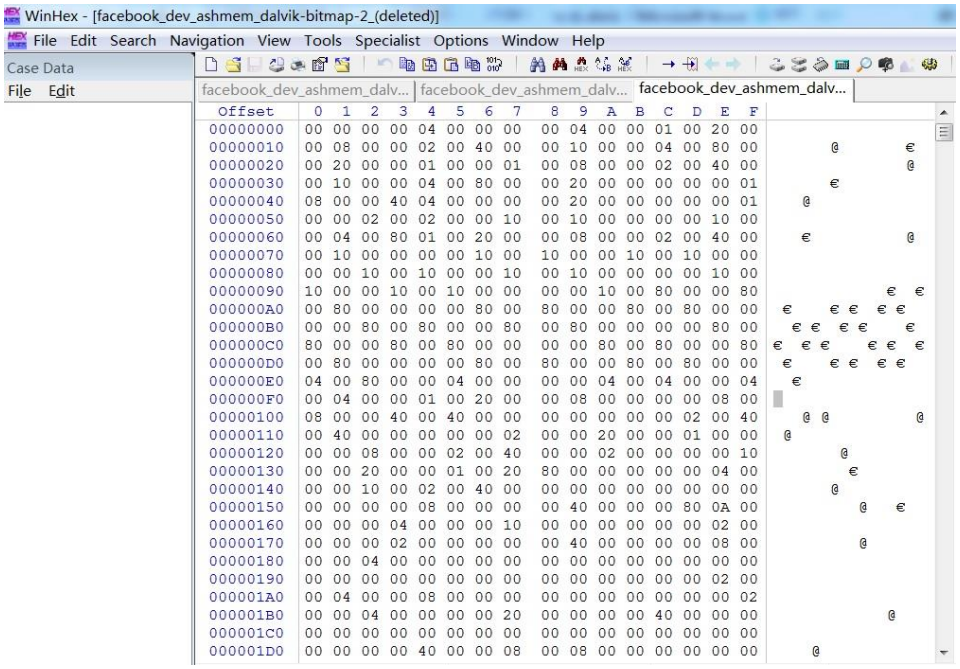

图 3.2.6 用 winhex 查看 dalvik-bitmap-2 二进制文件

因此我们可以知道，查看 dalvik-bitmap、dalvik-mark-stack、dalvik-card-table 可以了解基本的内存分配信息。而要查找内存数据，则需在 dalvik-heap、dalvik-jit、dalvik-LinearAlloc、dalvik-aux-structure 等实际包含数据的内存区域查找，其中以 dalvik-heap 最为重要。

图 3.2.5 是在软件 winhex[19]中二进制文件 dalvik-bitmap-1 的十六进制表示, 图 3.2.6 是二进制文件 dalvik-bitmap-2 的十六进制表示。由图可知，bitmap 中大量的 bit 都为 0，说明无论是在运行时间，还是在垃圾收集时，虚拟内存中都有大量字节未被使用。通过直接查看获取的 dalvik-heap 二进制文件，也可以印证这一点。

图 3.2.7 是在 winhex 中查看 dalvik-heap 的结果，其中有大量的空字符 0。由此我们可以知道，Dalvik 堆中会有大量连续的未被使用的内存，这些只含有空字符的内存对我们的取证工作来说是不需要的，同时，如果这些未被使用的内存被获取，会大大地延长将内存镜像传输至电脑的时间，以及分析处理内存镜像时查找有效内容的时间，因此，我们可以考虑在获取内存镜像时将大段连续的未被占用的内存空间过滤，以减少传输时间和分析时间。

对于过滤内存中的未被使用的连续的全零区域，有两种主要的方法：

第一是通过读取 dalvik-bitmap 文件，对内存中的每 8 个字节在运行时间或垃圾收集时是否被使用进行判断。如果被使用，则读取并存储在目标文件中，如果未被使用，则跳过。这个方法存在一些问题，其中最主要的是，如果堆内存的读取顺序与 dalvik-bitmap 中的顺序不能对应，或者是这两个文件之一读取的时候有错位或遗漏，就会造成判断的错位，并且，这个错位会一直传递到文件的最后。因此，一旦有读取错误的发生，导致的后果是非常严重的。因此，我们需要考虑另一种方法。

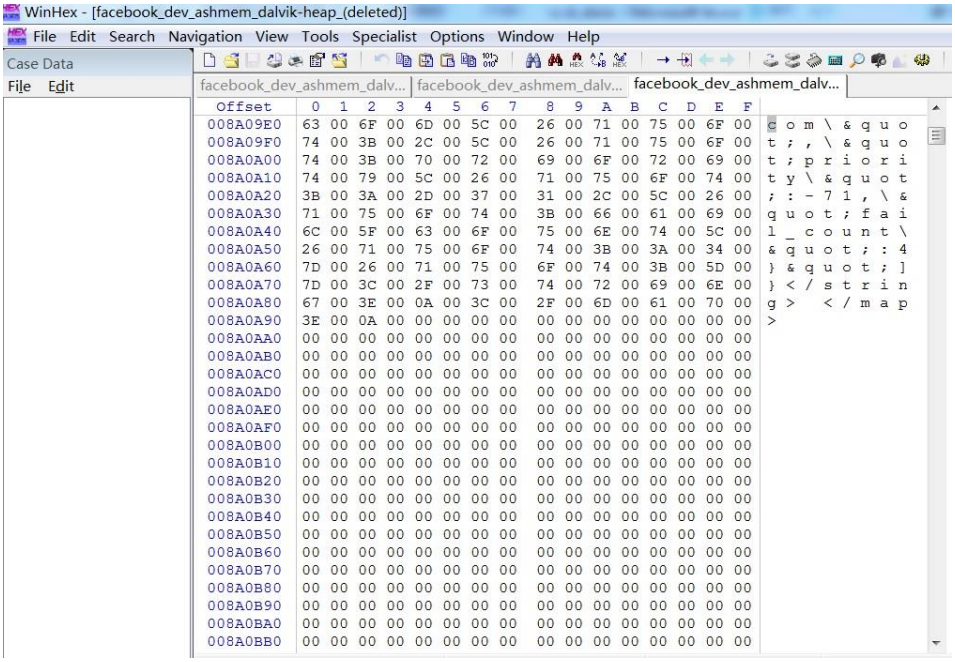

图 3.2.7 在 winhex 中查看 dalvik-heap 文件

由于通过观察 dalvik-heap 可知，一般未被使用或被释放的内存区域相对集中，因此在过滤这些内存块时，我们可以采用一个更加简单的方法：直接读取一定大小的内存块，判断是否为全零，如果不是，则读取并存储在目标文件中，否则跳过。

我们以此思想实现了获取内存镜像的程序 mem_heap.c。该程序主要有三个部分：

1. 主函数：int main(int argc, void **argv)，执行时需要有两个输入变量，依次为：

   整型变量 int target_pid:需获取内存镜像的目标进程的 pid；

   字符串变量 char* outdir：保存内存镜像的目标路径。

2. 内存获取函数：void pullPidMemory(char *outdir, int target_pid)，两个输入变量分别为主函数中的 outdir 和 target_pid.

3. 读取指定地址内存区域函数：void readMem(int pid, unsigned int start, unsigned int len, FILE *out_file)，四个输入变量分别是：

   整型变量 pid：需获取内存镜像的目标进程的 pid；

   整型变量 int start： 需获取的内存区域在虚拟内存中的起始地址；

   整型变量 int len： 需获取的内存区域的字节数；

   文件对象 outfile：存储内存镜像的目标文件。

mem_heap.c 的具体工作流程如图 3.2.8 所示。

3.2.2. 内存镜像获取程序的使用方法

需要单独用 mem_heap 程序获取内存镜像时，需要执行如下步骤：

1. 在 linux 系统中下载并安装编译工具 gcc-arm-linux-gnueabi：

   sudo apt-get install gcc-arm-linux-gnueabi

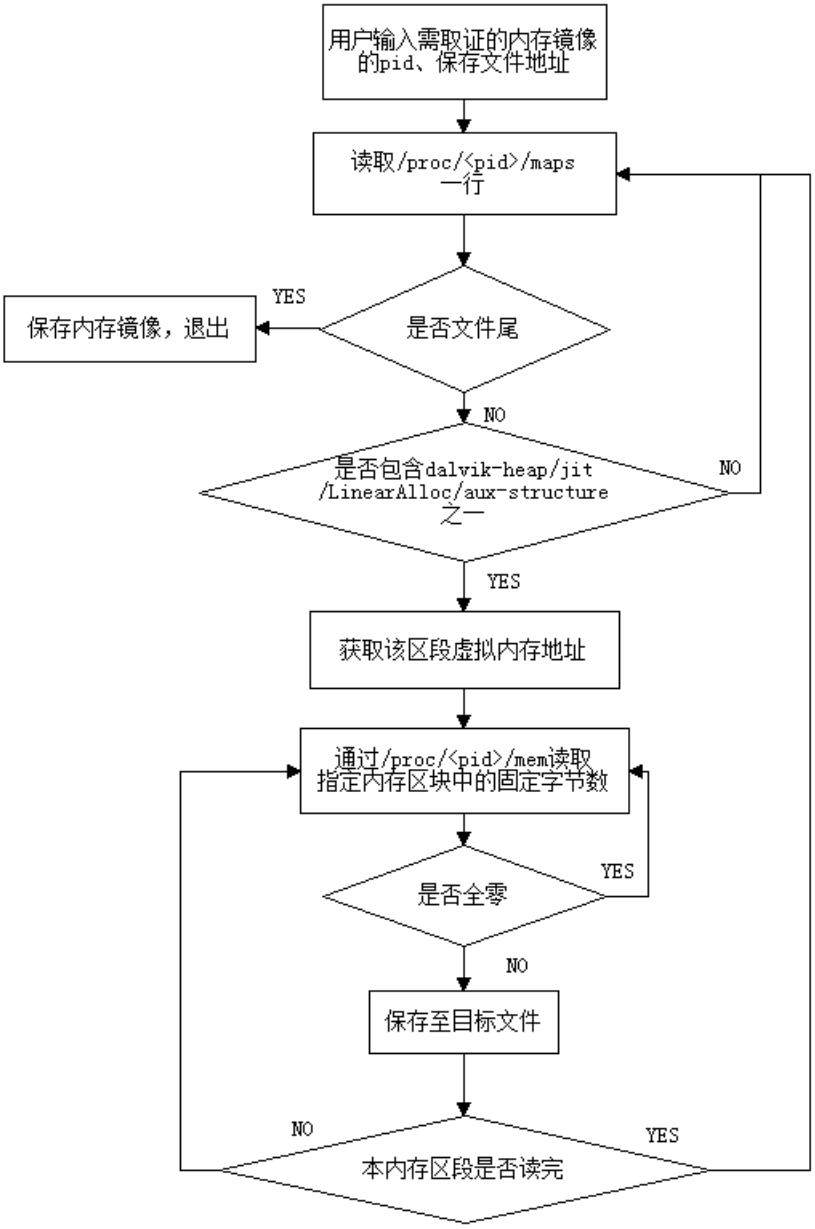

图 3.2.8 mem_heap 运行流程

2. 通过 gcc-arm-linux-gnueabi 编译工具将 c 文件编译为可执行文件：

   arm-linux-gnueabi-gcc mem_heap.c -static -o mem_heap

3. 将编译得到的可执行文件导入 Android 设备的/data 文件夹：

   adb push mem_heap /data/mem_heap

   如果权限不足（Permission Denied），则需通过 chmod 命令修改/data 文件夹权限。

4. 查看运行进程的 pid：

   adb shell ps

   adb shell su –c ’dumpsys meminfo ’（需要 root 权限）

5. 运行 mem_heap 可执行文件获取进程内存镜像至指定文件：

   adb shell su –c ‘./data/mem_heap <pid> <dir>’

6. 将获取的内存镜像导入电脑：

   adb pull <dir>

3.2.3. 内存镜像获取的效率比较

本节对[18]中的内存获取方法进行了改进，从而提高了效率。为了比较两种内存镜像获取方法的效率，我们以几种常见应用为例进行了测试，结果如表 3.2.1。

表 3.2.1 优化后的内存镜像获取方案与原方案效率对比

进程名	对比项	大小/MB	获取时间/s	传输时间/s
com.netease.mobimail	原内存镜像获取方法	745	74.30	187.02
	优化后的方法	9.51	9.68	2.59
优化方法获取内存所需时间/空间资源占原方法百分比		1.28%	13.0%	1.38%
com.facebook.katana	原内存镜像获取方法	737	131.79	185.21
	优化后的方法	22.96	12.65	6.68
优化方法获取内存所需时间/空间资源占原方法百分比		3.12%	9.6%	3.61%
com.tencent.mm	原内存镜像获取方法	728	107.67	151.7
	优化后的方法	21.47	13.57	6.84
优化方法获取内存所需时间/空间资源占原方法百分比		2.95%	12.6%	4.51%
com.immomo.mm	原内存镜像获取方法	731	120.14	162.39
	优化后的方法	20.51	11.73	6.73
优化方法获取内存所需时间/空间资源占原方法百分比		2.81%	9.8%	4.14%

从表 3.2.1 中可以看出，本节所采用的方法不仅减小了需获取的内存镜像大小，节省了空间资源，更可以显著地提升内存镜像的传输转存效率。因为大量未被使用的内存被过滤，让实际获取的内存镜像所占空间大大缩小，因此将内存镜像从移动端设备传输至计算机的时间也大大减少了，新方案的传输时间被控制在原先的 5%以内。

对于内存镜像的读取效率，本节中的方法对此虽然有所提高，但是和传输效率相比提高有所不足。这是因为在检测一段内存是否被使用时，必须将其从/proc/<pid>/mem 中读取至字符串中，这一步在优化和未优化的方法中都需要经历并消耗时间。本节所采用的方法虽然在一定程度上缩小内存读取范围，将读取内容主要限定在 dalvik-heap，dalvik-jit，dalvik-aux-structure 等几个部分中，但是由于这几部分本来就占用了较大的空间（如图 3.2.4，仅 dalvik-heap 就占据了获取的 700 余 MB 内存镜像超过 500MB 的空间），所以这一举措对缩短获取内存的时间的效果是有限的。另外，本方法中读取内存镜像时多次迭代可能在一定程度上降低代码效率。不过，因为大量的未被使用的内存没有被写入镜像文件，因此写入文件的时间被节省，内存镜像读取效率也有所提高。

最后，由于读取的内存镜像的大小大大缩小了，分析内存镜像时，查找特定的内容的时

间也会缩小，因此内存分析效率也会大大提高。

3.3. 内存镜像分析

由于在上一节，我们是按进程获取了内存镜像，在本节中，我们也将按不同的进程及其代表的应用对内存镜像进行分析。我们在邮件管理、即时通信、支付三类收到广泛关注、同时其记录与调查取证密切相关的应用中各选一款较为具有代表性的进行分析，以获取用户名、密码、邮件记录、交易记录、银行卡记录、wifi 连接记录等对取证十分重要的信息。下文中，我们将分别对网易邮箱（com.netease.mobimail）、陌陌（com.immomo.momo）、微信（com.tencent.mm）（支付部分）进程的内存镜像进行处理与数据提取。

3.3.1. 网易邮箱（com.netease.mobimail）内存镜像分析

邮件是现代通信技术中较为正式的一种，因此对于邮箱的取证非常重要的地位。当前，通过移动端应用处理邮件越发普遍，同时，一款邮件类应用往往可以让用户绑定多个邮箱，与用户的大量个人信息相关联。因此，在 Android 平台对邮件类应用取证是很有意义的。网易邮箱是国内一款用户较多的邮箱，它的 Android 应用同样支持用户绑定其他邮箱，因此，本文以其作为 Android 平台邮箱类应用的代表，对其进行分析。

3.3.1.1. 用户名获取

在从内存镜像中获取电子邮件的用户名之前，我们需要首先确认它在内存镜像中是否存在。邮箱类应用中的用户名一般是邮件地址，有相对较为明显的特点。一般的形式为：

用户标识符@域名

用户标识符一般包含的字符有英文大小写字母、阿拉伯数字和’_’、’.’、’-’等符号。域名中所包含的字符与之类似，并且，一般来说至少会出现一次’.’符号。考虑到这些字符均为英文中常见符号，因此，邮箱地址在内存镜像中有可能是以 ASCII 或 Unicode 编码方式存在的。在已知登录过的邮箱用户名（bailan0506@gamil.com、 bailan0506@163.com、bailan_seu@qq.com）的情况下，我们直接在获取的内存镜像中对其 ASCII 和 Unicode 形式进行搜索，确认其在内存镜像中确实存在，如图 3.3.1、图 3.3.2 所示。

```
00002870  88 8B 1B 42 9B B0 34 1F  00 00 00 00 12 00 00 00  ˆ‹ B›°4
00002880  2D 2D 72 75 3B 00 00 00  50 C9 B8 41 00 00 00 00  --ru;   PÉ¸A
00002890  12 00 00 00 00 00 00 00  62 00 61 00 69 00 6C 00          b a i l
000028A0  61 00 6E 00 30 00 35 00  30 00 36 00 40 00 31 00  a n 0 5 0 6 @ 1
000028B0  36 00 33 00 2E 00 63 00  6F 00 6D 00 AB 00 00 00  6 3 . c o m «
000028C0  E8 B1 B8 41 00 00 00 00  00 00 00 00 00 00 00 00  è±¸A
000028D0  00 00 00 00 00 00 00 00  BC 81 04 77 00 00 00 00          ¼  w
000028E0  10 00 03 00 E6 11 00 50  00 D0 4C 74 07 00 00 00      æ  P ÐLt
000028F0  00 00 00 00 01 00 00 00  0C 00 00 00 00 00 00 00
00002900  00 00 00 00 00 00 00 00  58 C0 BC 41 C8 53 1C 42          XÀ¼AÈS B
00002910  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00002920  01 00 00 00 48 8E B5 6D  09 00 00 00 88 8E B5 6D      HŽµm    ˆŽµm
00002930  15 00 00 00 88 90 B5 6D  00 00 00 00 00 00 00 00      ˆ µm
00002940  00 00 00 00 00 00 00 00  01 00 00 00 01 00 00 00
00002950  30 8E B5 6D 00 00 00 80  27 79 03 77 00 00 00 00  0Žµm   €'y w
00002960  E0 C0 B8 41 1B 00 00 00  70 BC 27 42 00 00 00 00  àÀ¸A    p¼'B
00002970  D0 67 34 42 F0 C7 1F 42  00 3C 20 42 23 00 00 00  Ðg4BðÇ B < B#
00002980  E0 C0 B8 41 00 00 00 00  A0 8C 1B 42 9B B0 34 1F  àÀ¸A     Œ B›°4
00002990  00 00 00 00 12 00 00 00  69 75 73 65 3B 00 00 00          iuse;
000029A0  50 C9 B8 41 00 00 00 00  12 00 00 00 00 00 00 00  PÉ¸A
000029B0  62 00 61 00 69 00 6C 00  61 00 6E 00 30 00 35 00  b a i l a n 0 5
000029C0  30 00 36 00 40 00 31 00  36 00 33 00 2E 00 63 00  0 6 @ 1 6 3 . c
```

图 3.3.1 内存镜像中 Unicode 编码的邮箱地址

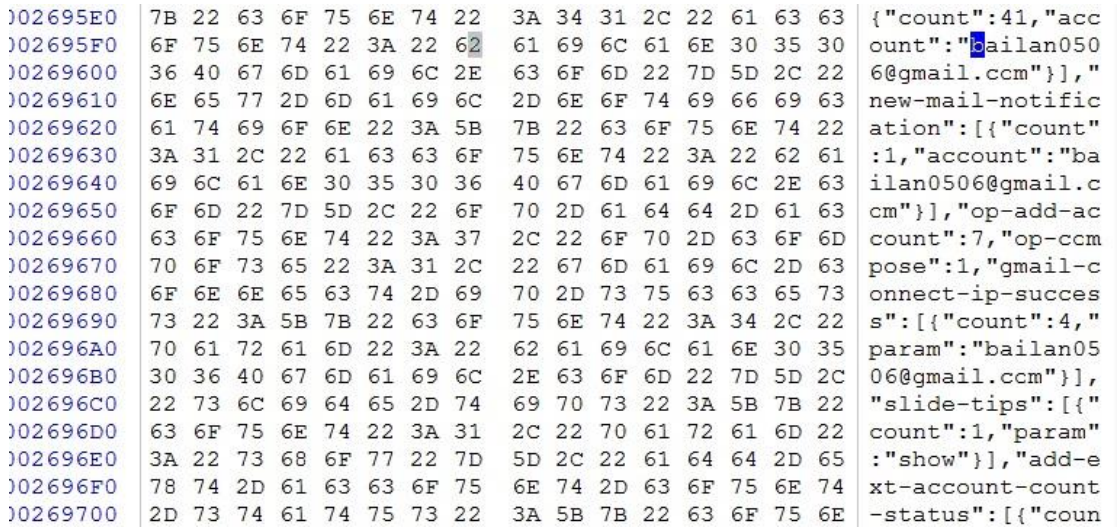

图 3.3.2 内存镜像中 ASCII 编码的邮箱地址-1

从图 3.3.2、图 3.3.2 中可以看出，用户的邮箱地址在内存镜像中以 Unicode、ASCII 两种编码均有出现。同时我们可以得出，进程内存镜像 Unicode 编码采用的是 UTF-16LE 编码方式。

尽管内存镜像中可以找到用户邮箱地址，但是这不代表着在我们未知用户邮箱的时候还可以准确地找到它们。为了提取用户邮箱地址这一有效信息，我们需要找到邮箱地址所内存区域的数据结构。

在 Unicode(UTF16-LE)编码的条件下，由图 3.3.1 可知邮箱地址之前会出现一串十六进制表示为 50C9B841 的字符串，这可以作为提取邮箱地址的标志。但是由于这一标志在内存镜像中出现过于频繁，且可能在下次获取内存的时候会改变，因此，在内存镜像中提取 Unicode 编码的邮件地址并不十分可行。

在 ASCII 编码条件下，由图 3.3.2 可以看出，邮箱地址所在的区域的数据有较为明显的数据结构，更方便提取。图 3.3.2 中，邮箱地址前有前缀“account”:，但是查看获得的内存镜像可得，邮箱地址前的提示信息并不唯一，如图 3.3.3 中，邮箱地址之前的提示信息为“param”:。

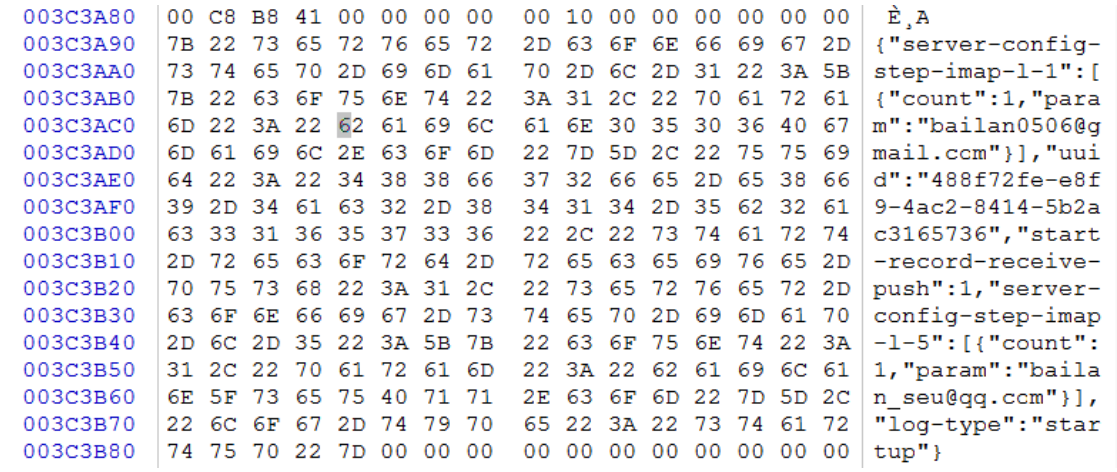

图 3.3.3 内存镜像中 ASCII 编码的邮箱地址-2

邮箱前提示信息的多样会为邮箱地址的提取造成一定的干扰，但是，内存镜像中 ASCII

编码的邮箱地址还有一个特征为前后存在有引号，因此，我们可以结合这一特征，和上文所述的邮箱地址格式，得到匹配邮箱地址的正则表达式为：

"\"[0-9a-zA-Z\.\-_]+@[0-9a-zA-Z\-_]+\.[0-9a-zA-Z\.\-_]+\""

即一个头尾为引号的、中间必依次出现一次'@'和一次'.', '@'之前、'@'与'.'之间、'.'之后均需要有一个或以上字符，这个字符必须是英文字母、数字、'.'、'_'或'-'。

以此正则表达式为基础，我们编写了一个 python 脚本，以二进制读取模式打开内存镜像文件，获取与正则表达式所匹配的字符串，并通过函数 strip()来去除头尾的引号。为了防止提取结果有重复，以 Dictionary 数据结构储存获取的结果，并以获取的邮件地址作为键，以查找到的次数作为值。主要代码为：

```
for search_line in read_file:
    a=re.finditer("\"[0-9a-zA-Z\.\-\_]+@[0-9a-zA-Z\-\_]+\.[0-9a-zA-Z\.\-\_]+\"",search_line
    )
    if a!=None:
        for i in a:
            mailaddress[i.group().lower().strip("\"")]=mailaddress.get(i.group().lower(),0)+1
```

最后，分析内存镜像得到：

以下可能是您曾经登陆的电子邮箱账号:

bailan0506@gmail.com

bailan_seu@qq.com

bailan0506@163.com

曾经登陆过的电子邮箱账号均被找到。

3.3.1.2. 用户登录密码获取

获取登录密码是破解一个账户的关键，对取证来说至关重要。在用户态下，一旦输入密码、完成登录，密码一般就不再可见，这时，通过内存取证获取密码就显示出其重要意义。

```
000F7040  90 91 BC 41 00 00 00 00  00 00 00 00 00 00 00 00   '¼A
000F7050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000F7060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000F7070  00 00 00 00 00 00 00 00  6F 00 72 00 33 00 00 00          o r 3
000F7080  50 C9 B8 41 00 00 00 00  09 00 00 00 00 00 00 00  PÉ¸A
000F7090  31 00 71 00 39 00 32 00  25 00 49 00 49 00 31 00  1 q 9 2 % I I 1
000F70A0  31 00 00 00 A0 9F CE 41  30 00 00 00 1B 00 00 00  1    ŸÎA0
000F70B0  40 A9 B9 41 00 00 00 00  01 00 00 00 E8 6F D9 42  @©¹A         èoÙB
000F70C0  01 00 00 00 23 00 00 00  60 AC BC 41 00 00 00 00      #    `¬¼A
000F70D0  60 6B 23 42 18 6B 23 42  01 00 00 00 00 00 00 00  `k#B k#B
000F70E0  20 00 00 00 13 00 00 00  30 63 D0 42 00 00 00 00           0cÐB
000F70F0  30 4D C1 42 23 00 00 00  E0 C0 B8 41 00 00 00 00  0MÁB#    àÀ¸A
000F7100  18 C0 2D 42 00 00 00 00  00 00 00 00 17 00 00 00   À-B
000F7110  00 00 00 00 4B 00 00 00  50 C9 B8 41 00 00 00 00      K    PÉ¸A
000F7120  17 00 00 00 00 00 00 00  49 00 47 00 4E 00 4F 00          I G N O
000F7130  52 00 45 00 5F 00 43 00  41 00 43 00 48 00 45 00  R E _ C A C H E
000F7140  5F 00 45 00 58 00 50 00  49 00 52 00 41 00 54 00  _ E X P I R A T
000F7150  49 00 4F 00 4E 00 00 00  00 00 00 00 AB 00 00 00  I O N        «
```

图 3.3.4 内存镜像中 Unicode 编码的密码

一些网站会对用户所设置的密码的所用字符进行限制，但是因为邮箱应用可以绑定多个

邮箱，规则不易统一。因此本节中认为密码可以使用 ASCII 编码中所有可见字符，长度在 6-20 位之间。

与获取用户名时相同的是，我们需要首先在已知密码的情况下验证其是否在内存镜像中存在。我们在内存镜像中对密码的 ASCII 与 Unicode(UTF-16LE)编码形式进行搜索，所得结果如图 3.3.4、3.3.5 所示。此处所使用的密码为 1q92%II11。

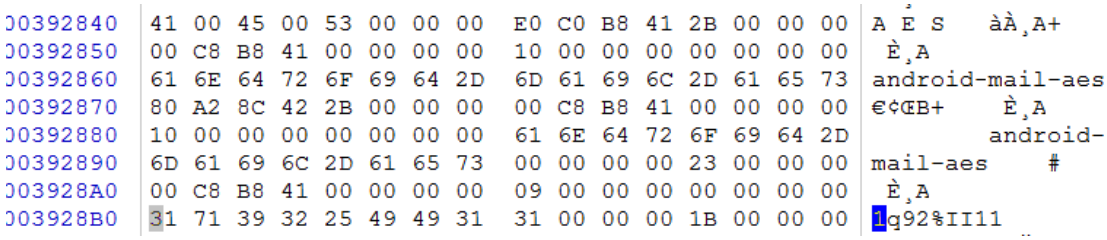

图 3.3.5 内存镜像中 ASCII 编码的密码

由图 3.3.4 可知，密码会以 Unicode 编码存在于内存镜像中，同时，与邮箱地址相同的是，用户登录密码之前也会出现一串十六进制表示为 50C9B841 的字符串。同样，由于这一标志在内存镜像中出现过于频繁，且可能在下次获取内存的时候会改变，再加上密码自身的字符格式特点并不鲜明，因此，用 Unicode 获取内存镜像中的密码的方法并不十分可行。但是，在 ASCII 编码条件下，可以看到密码之间有一段字符串为”android-mail-aes”，这应当是密码在内存中需要采用 AES 加密而造成的。”android-mail-aes”这一字符串在内存中出现次数并不多，并且，经验证，往往与用户登录密码同时出现，因此，我们可以通过查找”android-mail-aes”之后的符合登录密码形式的字符串来获取用户登录密码。匹配密码的正则表达式为：

“[!-~]{6,20}”

该正则表达式可匹配所有 6-20 位由可见字符构成的字符串。由于“android-mail-aes”会在密码之前连续多次出现，因此，我们需要在匹配得到的字符串中去除“android-mail-aes”。获取用户登录密码的 python 脚本的工作流程如图 3.3.6 所示，为了多次提取到同一密码的问题，我们同样用 Dictionary 数据结构来存储，核心代码为：

```
head="android-mail-aes"
password=dict()
for line in read_file:
    pos=line.find(head)
    if pos!=-1:
        search_line=line[pos:]
        a=re.finditer("[!-~]{6,20}",search_line)
        if a!=None:
            for i in a:
                if not ("android-mail-aes" in i.group()):
                    password[i.group()]=password.get(i.group(),0)+1
```

运行脚本，分析内存镜像得到结果：

以下是您可能曾经使用的密码:

t#Cpu#CH

MyB`NyB

1q92%II11

QyBPQyB

提取结果之中包含了所用的密码。由于密码的形式并不鲜明，因此可能存在误提取的情况。不过考虑到一般来说电子邮箱并不会对用户尝试密码的次数有非常严格的限制，因此，在实际取证过程中，我们可以通过直接尝试所有获取结果来找到真正的登录密码。

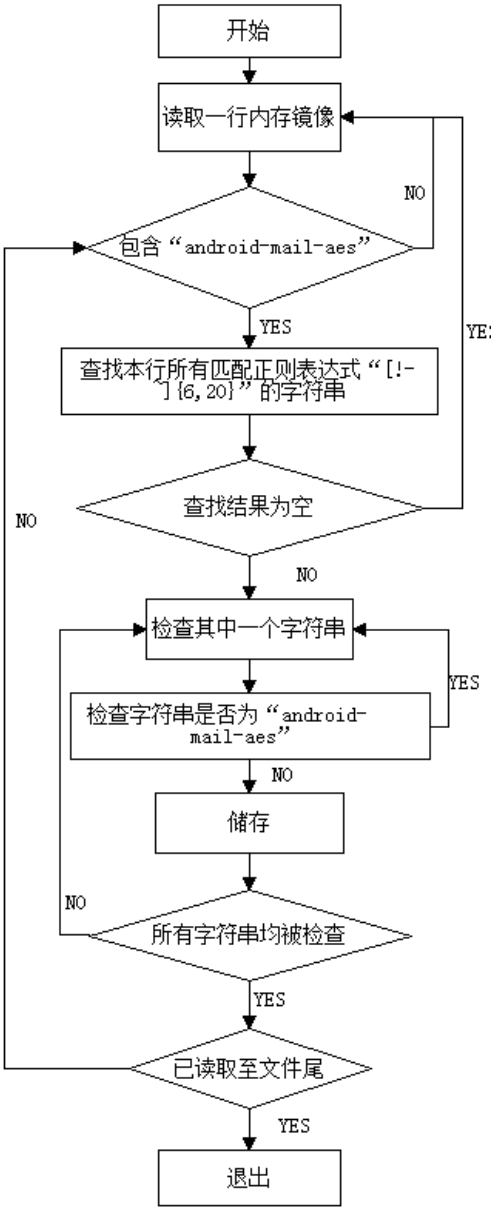

图 3.3.6 从内存镜像中获取密码的流程

3.3.1.3. 发件人地址和邮件内容获取

在内存中获取用户名和密码之后，我们可以登录到邮箱账户，查看该用户的邮件。但是，对于一些用户已经删除的邮件则无法查看到。但是，如果用户曾用 Android 手机接收或查看该邮件，我们就可以通过内存取证的方式，从内存中获取邮件内容。

在内存中，邮件的存在形式并不如我们实际上在用户态所见的那样有一定的格式和规律。Android 应用用户所接收过的每封邮件的发件人邮件地址和邮件内容在内存中分别处于不同的区域。如图 3.3.7 中，我们可以看见一个单独的电子邮件地址，但是附近并没有相关的内容。在图 3.3.8 中，有一封邮件的内容，但是我们并不能在附近找到其他相关的信息，如发件人、发件时间等等。因此我们只能对发件人和邮件内容分别提取。

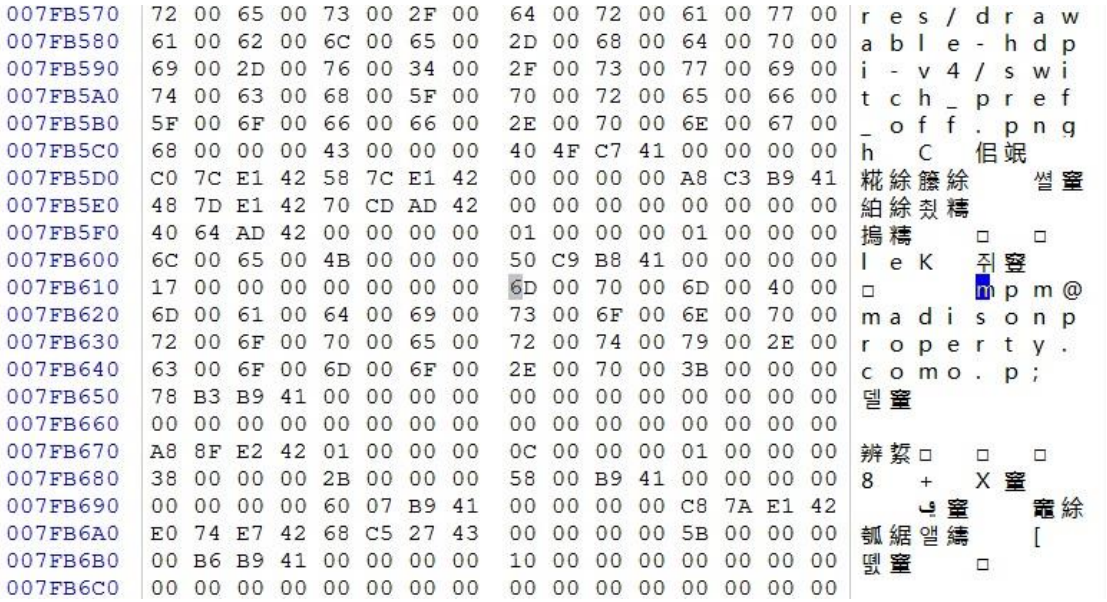

图 3.3.7 内存镜像中出现的一个发件人邮箱地址

图 3.3.8 内存镜像中的一段邮件内容

很显然，由于邮件内容可能包含除英文之外的多种字符，电子邮件的内容在内存中主要是以 Unicode(UTF-16LE)编码形式存在。而对于发件人邮件地址，我们对一些发件人邮箱地

址的 ASCII 和 Unicode 编码形式在内存中进行了搜索，结果是无法找到 ASCII 码形式的发件人邮箱，可以找到 Unicode 编码形式的发件人邮箱。对于这一结果，我们通过两个正则表达式：

"[0-9a-zA-Z\.\-_]+@[0-9a-zA-Z\-_]+\.[0-9a-zA-Z\.\-_]+"

u"[a-zA-Z\.\-_]+@ [a-zA-Z\-_]+\.[0-9a-zA-Z\.\-_]+"

对内存中的内容进行匹配来验证。前者匹配 ASCII 编码的邮件地址；后者在将读取的内存内容转换为 Unicode 形式后，匹配 Unicode 编码的邮件地址。验证结果为，由第一个正则表达式，我们未能找到除了用户邮箱之外的邮件地址，而用第二个正则表达式，我们找到了超过 60 个邮件地址。因此，内存中的发件人邮箱地址主要是 Unicode 编码，我们可以采用第二个正则表达式来匹配，获取发件人邮箱地址的部分结果如下：

mpm@madisonproperty.com

no-reply@duolingo.com

kiehls@mail.kiehls-usa.comK

account@mail.zhihu.com

iJGA@notifications.google.com

邮箱地址末位可能会出现一些多余的字符，可以很容易地被发现并被人工去除。

获取了发件人电子邮箱地址之后，我们需要获取邮件正文内容。由图 3.3.8 可知，内存中的邮件正文内容前后并没有十分显著的数据结构。但是，一般来说，邮件正文内容都会达到一定长度，并且由连续的可见字符某几个控制字符构成，具体到本文所在的场景，即为英文字母、汉字字符、中英文标点符号、空格等，而且通常来说，文本中至少出现一次空格，因此匹配邮件正文的正则表达式为：

u"[\u0020-\u007e\u4e00-\u9fa0\u3001-\u3005\uFF01-\uFF1F\u2013\u2014\u2026]+\s[\u0020-\u007e\s\u4e00-\u9fa0\u3001-\u3005\uFF01-\uFF1F\u2013\u2014\u2026]{40,}"

其中\u0020-\u007e 即为 ASCII 中的可见字符，被 Unicode 兼容。\u4e00-\u9fa0 为所有汉字字符，其余为常见中文标点。\s 为空格、换行符、分页符或制表符。并且，我们将可匹配为邮件正文的字符串限定为出现第一个空格后至少有 40 个字符。同时，为了防止将非邮件内容误判为邮件内容，我们对以正则表达式获取的内容进行进一步判断。首先去除开头的空格以后，开头字符必须是大写英文字母（满足英文句首大写的规则）或汉字，并且第一个单词不全为英文大写字母。

同时，由图 3.3.8 可知，内存中邮件内容会出现整体或是部分重复的情况。在如获取邮箱地址、密码一样，将邮件正文读取后，以 Dictionary 储存，只能做到使完全相同的邮件正文内容不重复存储，如果某次获取的内容是另一次的子集，则会被重复储存。因此，在完成内容获取之后，将所有获取的内容排序，从前向后遍历，如果后一项包含前一项，则删除前一项。在实验中，由于我们发现，获取内容末位有时会包含几个并不为邮件内容的字符，因此，将删除前一项的判定规则设定为在前一项不包含最后 4 个字符的情况下，后一项包含前一项。获取邮件内容的流程如图 3.3.9。最终，我们从进程内存镜像中获取了 18 份邮件正文。并且，在实验中，我们能够在内存中找到被删除的邮件内容，并将其提取。

如图 3.3.10 所示，本文作者收到一封来自 Dropbox 的邮件并完全删除（从收件箱、垃圾箱中移除）了这封邮件，随后，我们通过之前编写的 python 脚本在获取的进程内存镜像中尝试读取本邮件。最终，我们成功地在内存镜像之中找到了该邮件的部分内容：

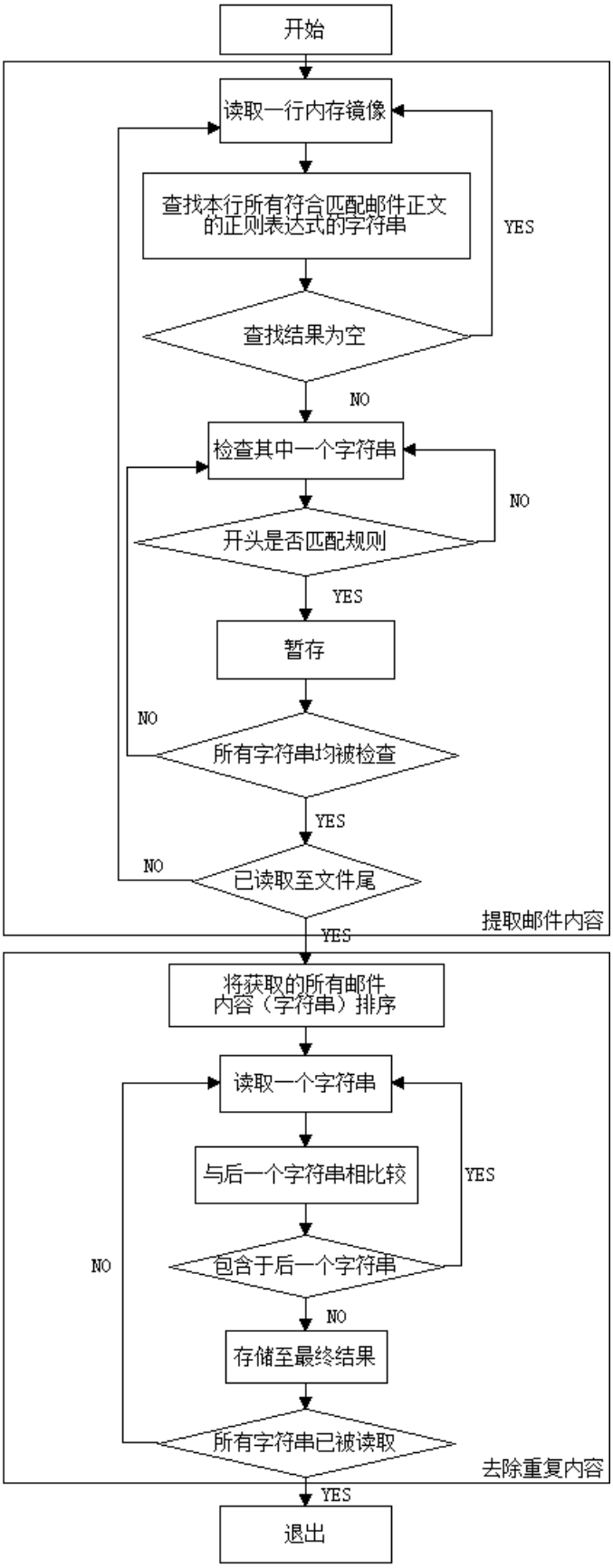

图 3.3.9 在内存镜像中提取邮件内容

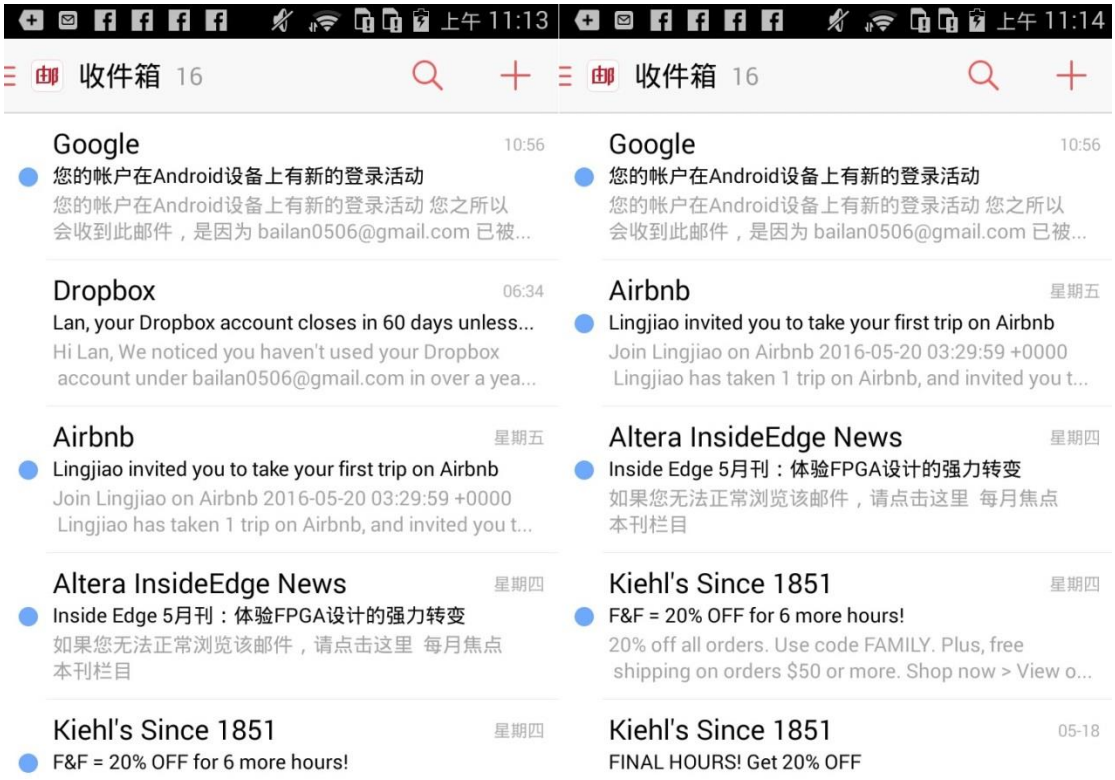

图 3.3.10 删除一封来自 Dropbox 的邮件

Hi Lan, We noticed you haven't used your Dropbox account under bailan0506@gmail.com in over a year. Want to keep your account? If so, sign into Dropbox before July 23, 2016. Sign in to keep your account Click the button or go directly to https://www.dropbox.com/login Recently used Dropbox but still received this email? If you recently used Dropbox, we

尽管邮件内容不能被完全重现，但是能够在邮件被完全删除的情况下通过内存取证恢复部分邮件内容，这依然是非常有意义的。

3.3.1.4. 在不同情境下对网易邮箱进程内存进行分析的结果比较

本节中，我们在不同情境下对网易邮箱的进程内存进行了获取，并统计了获取的用户邮箱域名、密码、发件人邮箱、邮件内容四种有效信息的次数。

表 3.3.2 不同情境下的网易邮箱内存取证

	获取用户邮箱域名次数	获取密码密码总次数	获取有效密码次数	获取包含有效密码结果次数	获取发件人邮箱次数	获取有效发件人邮箱次数	获取邮件次数	获取有效邮件内容次数
情境一	3	4	1	1	61	56	18	17
情境二	1	1	1	1	119	110	18	12
情境三	1	47	6	2	117	107	21	21

表 3。3..2 中，以下几个项目的含义为：有效密码指的是可以用来登录用户邮箱的密码；包含有效密码的结果可以看作有效密码+少量无效字符；在获取发件人邮箱时，可能会误获取一些并非用户邮箱地址的字符串，可以容易地被人工识别出并除去，剩余的即为有效发件人邮箱；同样，一些获取的邮件内容也可能是以无意义的字符构成的，可以被判定为非有效

邮件内容。

情境一是在登录 3 个不同的邮箱之后立即对网易邮箱应用对应的进程进行内存取证。这次取证获取了所有的用户邮箱域名，但是只获得了最后一次的输入的有效密码。因为是刚刚登录，能获取的发件人邮箱和邮件内容也数量少于其他两种情境。

情境二是在登录过后一段时间后，并同步了一些邮件后对网易邮箱应用对应的进程进行取证。这次取证只获取了一个用户邮箱的域名和其对应的密码。但是内存中获取的发件人邮箱地址数随着同步邮件增多而增多，但是内存中获取的有效邮件内容的数量并未因此而提升。可能的解释是因为邮件内容占用内存较多，部分相关内存被释放或覆写了。

情境三是在登出后对网易邮箱应用的对应进程进行取证。同样的，这次只获取了一个用户邮箱的域名，但是获取了多个有效的邮箱密码，不过在登出之后，有更多的字符串被误判为邮箱密码。由于在登出时在各个邮箱之间切换，时获取的有效邮件内容反多于前两个情境下的取证。在登出之后，从用户界面是无法知道登录邮箱、密码、发件人邮箱、邮件内容等信息，但是通过内存取证，这些信息被成功获取了，这也体现出了内存取证的重要价值。

3.3.2. 陌陌（com.immomo.momo）内存镜像分析

现代生活中，即时通信变得越发普及。陌陌作为一款面向陌生人交友的即时通信应用，可能为一些纠纷与犯罪行为提供重要的证据。因此，对 Android 平台的陌陌应用取证有着关键的价值。本节中，我们将以陌陌为即使通信软件的代表，对其内存镜像进行分析。

3.3.2.1. 用户档案信息获取

用户档案信息包含一个人各方面的资料，对于取证有着重要的价值。在已知陌陌 ID 的情况下，我们在内存镜像文件中对用户信息进行检索，发现陌陌用户信息在内存镜像中主要以 ASCII 编码形式存在，并且有明显的数据结构，例如：

*{"ok":true,"profile":{"momoid":"363664648","relationship":0,"name":"luna_bai","sex":"F","age":24,"regtime":"2016-05-03","bind_email":"NO","birthday":"1992-01-01","sign":"","sign_time":1462256810,"photos":["E107EB2D-4207-C144-7B23-760FE060208420160503"],"signex":{"time":0},"loc_lat":0,"loc_lng":0,"loc_timesec":1462256810,"distance":0,"constellation":"\u6469\u7faf\u5ea7","interest":"","website":"","aboutme":"","hangout":"","countrycode":"+86","phonenumber":"13*******07"…….*

"pversion":5,"relation":"follow","momoid":"321761789","show_momoid":"321761789","is_special_friend":0,"live_push":1,"ok":true,"timesec":1462256813}.....

我们可以从中获取的信息有用户的陌陌 ID、与本手机用户的关系、昵称、性别、注册时间、生日、国家、手机号、与本机用户的距离等等。在内存中，上述信息出现之前均有提示字符串，分别为 momoid、relation、name、sex、regtime、birthday、countrycode、phone。

包含用户信息的内存块组织结构较为多样，但有共同的特点是每出现一个新用户的档案资料，一定会出现字段”momoid”：用户的陌陌 ID。因此，本节中将字符串"\"momoid\""在一段内存中是否存在，作为判断该段内存中是否有用户信息的标识。同时，由于存储用户信息的内存区域包含在一对{}之中，因此，在找到包含字符串"\"momoid\""的内存区域后，可以用正则表达式：

'{[\x20-\x7e]+}'

来匹配包含用户资料信息的字符串。

由于用户信息中元素较多，因此定义一个用户类 user，以储存用户信息：

```
class user:
    momoid=""
    name=""
    sex=""
    relation=""
    regtime=""
    birthday=""
    countrycode=""
    phone=""
    distance=""
    def print_profile(self):
        …….
    def write_profile(self,file):
        ……
```

提取一个用户每一项信息的基本方法均为，找到对应的提示字符串，并在“：”之后用正则表达式匹配一个不含“{”、“}”、“，”的字符串，以得到需要的信息。该正则表达式为：

"[\x20-\x2b\x2d-\x7a\x7c\x7e\x7f]+"

同样的，为了防止同一用户信息在最终结果中重复出现，我们在提取内存信息的 python 脚本中，以 Dictionary 数据结构储存用户信息。其中以一个陌陌用户的唯一独特标识陌陌 ID 作为键，以用户信息对象作为值。

由于在内存中，本 Android 设备用户的信息与其他用户的信息存储格式较为相似，但是与本机用户的关系（user 类中的 relation 成员）一项会为空。因此，在对一个用户的资料获取完毕后，我们对其 relation 成员进行检查，如果这一项为空字符串，那么判断此用户为本机用户。

我们使用获取的陌陌内存镜像对编写的脚本进行测试，获得用户信息如结果如下：

用户资料：

陌陌 id：363664648

昵称：\u6d59\u6c5f\u676d\u5dde

性别：F

注册时间：2016-05-03

生日：1992-01-01

手机号：13*******07

这可能是本手机登陆的账号。

用户资料：

陌陌 id：321761789

昵称：\u5173\u6ce8\u7684\u8bdd\u9898

关系：follow

可见在内存镜像中，我们可以获得本机用户信息及其好友信息。同时，经实验测试，这些信息在登出后部分仍可以找到。

3.3.2.2. 用户聊天记录获取

聊天记录是调查者需要取得的关键证据，对取证来说非常重要。因此，我们需要对内存镜像中的聊天记录进行获取。与 3.3.1.2 节中的邮件正文内容信息相似，聊天记录在内存中以 Unicode(UTF-16LE)编码存在。

陌陌进程内存中的聊天记录主要以三种结构存在：

[1462256842694212, 1462256848317, , , 2, 0, 0, 01c74343, {"imageUploadedLength":0,"failcount":0,"usertitle":"","snapTimeSecond":0,"audiotime":0,"tailIcon":"","fileSize":0,"custombubble":"","isPlayed":0,"layout":0,"action":"","fileUploadProgrss":0,"snapcount":0,"originImgSize":0,"realdistance":-1,"sayhi":0,"imageHeight":0,"fileUploadSuccess":0,"newsource":"","tailAction":"","textv2":"","content":"你在干嘛呀","diatance":524,"bubbleStyle":0,"source":"","distanceTime":1462256788000,"tailTitle":"","isOriginImg":false,"fileName":"","notShowInSession":0,"imageWidth":0}, 0, 01c74343]

sendMessage---message : Message [msgId=01c74343, remoteId=321761789, content=你在干嘛呀, actions=, action=null, type=0,status=1, diatance=524.0, loadRes=false]

{"imageUploadedLength":0,"failcount":0,"snapTimeSecond":0,"audiotime":0,"tailIcon":"","fileSize":0,"custombubble":"","isPlayed":0,"layout":0,"action":"","fileUploadProgrss":0,"snapcount":0,"originImgSize":0,"realdistance":-1,"sayhi":0,"imageHeight":0,"fileUploadSuccess":0,"tailAction":"","textv2":"","content":"luna_bai 关注了你","diatance":530,"bubbleStyle":0,"distanceTime":1460338105000,"tailTitle":"","isOriginImg":false,"notShowInSession":0,"imageWidth":0}

经与 Android 手机 App 中的实际内容分析验证得：

对于第一种结构的聊天记录，前两个十进制数字分别为 16 位和 13 位的时间戳，可以用 time.localtime()函数转换为代表该消息的发送时间和被查看的时间，第一个’{’前的十六进制数字为消息的 ID。其余可获取的有效信息为"content":后的消息内容，以及"diatance":后的聊天双方距离，单位为米。对于这种形式的聊天记录，如果第一个时间戳为 0，则为本手机用户接受的消息。若两个时间戳均不为 0，那么则为本手机用户发出的消息。

对于第二种结构的聊天记录，只在该消息是本手机用户发出的情况下存在。其中，msgId=后的十六进制数字是消息 ID，content=后的内容为消息内容，diatance=后的内容为通信双方的距离。

对于最后一种聊天记录，能够获得的有效信息仅有消息内容和聊天双方距离。

对于三种结构的聊天记录，均包含关键词"content":或 content=，因此，本节中将字符串"content":或 content=在一段内存中是否存在，作为判断该段内存中是否有聊天记录的标识。接着，在这段内存中以如下正则表达式匹配所有中英文可见字符、空格、标点所组成的字符串以作为聊天记录相关信息的搜索范围：

u"[\u0020-\u007e\u4e00-\u9fa0\u3001-\u3005\uFF01-\uFF1F\u2013\u2014\u2026]+"

并且，符合如下正则表达式的为第一种结构的聊天记录：

u"\[[0-9][\u0020-\u007e\u4e00-\u9fa0\u3001-\u3005\uFF01-\uFF1F\u2013\u2014\u2026]+\]"

以 sendMessage 开头的为第二种结构的聊天记录、其余为第三种结构的聊天记录。

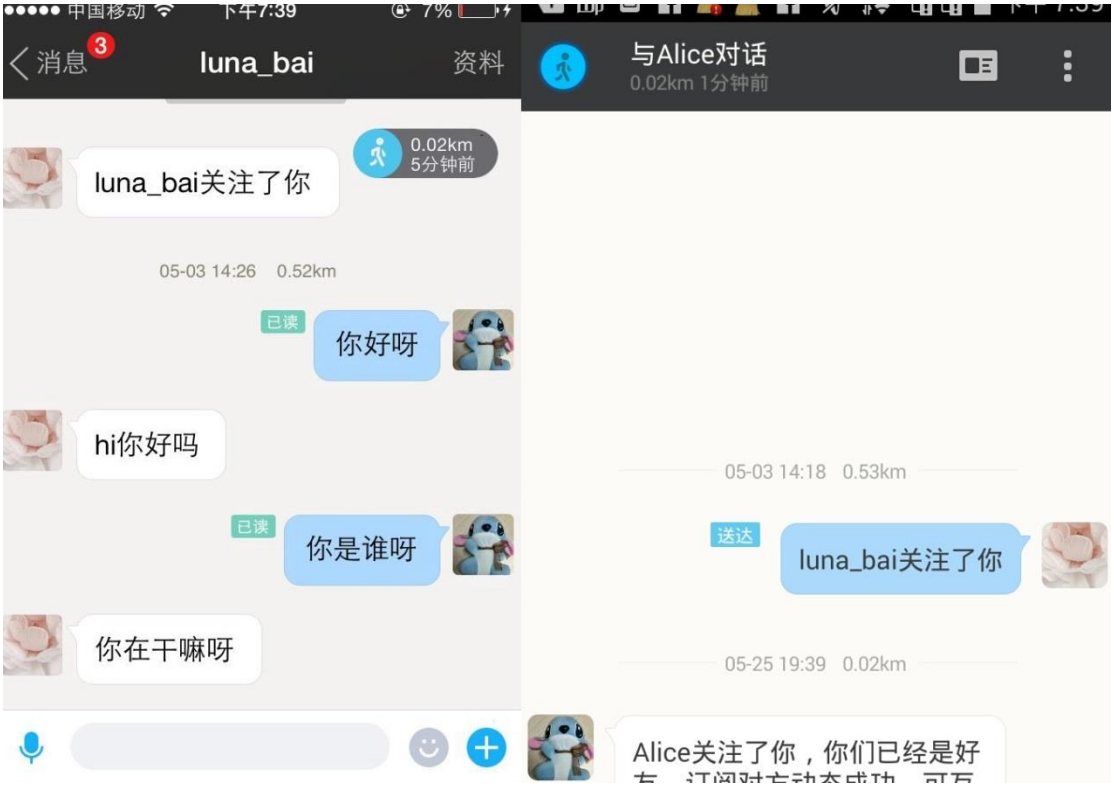

图 3.3.11 删除陌陌中的聊天记录

我们定义了如下对象 msg 对内存中的消息记录进行存储：

```
class msg:
    msg_id=""
    content=""
    sender=""
    send_time=""
    receive_time=""
    distance=""
    msg_form=0
    def print_profile(self):
        …….
    def write_profile(self,file):
        ……
```

成员变量依此表示：消息 id、消息内容、发送或接受、发送时间、接受/查看时间、聊天双方的距离。

与 3.3.2.1 节类似，我们以 Dictionary 数据结构存储获取的 msg 对象，以消息 ID 为键、

msg 对象为值。但是，第三类聊天记录中没有消息 id，为了防止出现这类聊天记录中的消息从未以其他两种形式出现而被漏掉的情况出现，我们将这类消息的消息 ID 设置为“unknown”+从 1 开始的序列号。

在测试中，我们用两台手机进行相互聊天，并在测试的手机上删除了聊天记录，如图 3.3.11 所示。

在保持登录状态下，读取内存镜像，可提取聊天记录：

消息 id：01c74343

内容：你在干嘛呀

发送 or 接收：发送

发送时间：2016-05-03 14:27:22

接收/查看时间：2016-05-03 14:27:28

聊天双方的距离：524m

内容：luna_bai 关注了你

聊天双方的距离：530m

内容：hi 你好吗

聊天双方的距离：524m

消息 id：0a2ca202

内容：你是谁呀

发送 or 接收：接收

接收/查看时间：2016-05-03 14:27:17

聊天双方的距离：524m

消息 id：000f8d21

内容：hi 你好吗

发送 or 接收：发送

发送时间：2016-05-03 14:27:00

接收/查看时间：2016-05-03 14:27:03

聊天双方的距离：524m

在登出状态下读取进程内存镜像，可获取聊天记录：

内容：你是谁呀

聊天双方的距离：524m

内容：hi 你好吗

聊天双方的距离：524m

内容：你在干嘛呀

聊天双方的距离：524m

可见，通过 Android 内存取证，我们可以在聊天记录被删除的情况下获取它们，甚至在用户已登出的情况下扔找到部分聊天记录。

3.3.2.3. 用户连接 wifi 记录信息

用户曾经连接的 wifi，尤其是 wifi 的 SSID、BSSID 等资料，与用户曾经的行踪等信息密切相关，对于取证也有重要的意义。Wifi 所在内存区域也由 Unicode（UTF-16LE）编码，基本形式为：

SSID: SEU-NLCT, BSSID: 00:19:70:fd:0b:c7, MAC: 00:66:4b:ee:05:6d, Supplicant state: COMPLETED, RSSI: -80, Link speed: 1,Net ID: 2, Metered hint: false

可见，其数据结构也非常明确。因此只需找到包含“SSID”的内存区域，提取：和，之间的信息即可。对于 wifi，我们主要提取其 SSID、BSSID、本机 MAC 地址信息。并定义一个 wifi 类：

```
class wifi:
    ssid=""
    bssid=""
    mac=""
    def print_profile(self):
        ……
    def write_profile(self,file):
        ……
```

并以 Dictionary 数据结构储存，以 SSID 为键，以 wifi 对象为值。

3.3.2.4. 在不同情境下对陌陌进程内存进行分析的结果比较

本节中，我们在不同情境下对陌陌的进程内存进行了获取，并统计了获取的用户信息、聊天记录两种有效信息的次数，如表 3.3.3 所示。

表 3.3.3 在不同情境下的陌陌应用进程内存取证

	获取用户名数	聊天记录总条数	获取不重复聊天记录次数	获取重复聊天记录次数
情境一	2	5	5	1
情境二	2	5	4	1
情境三	3	5	3	0

其中，情境一是在聊天之后直接获取进程内存后分析的结果，此时，可以分析得到所有聊天记录，并能获取用户与好友的较为详细的资料。情境二是在删除聊天记录之后获取进程内存而分析的结果。此时，与情境一相比，有一条聊天记录无法被获取。情境三是在删除聊天记录并登出后对进程内存进行获取并分析。此时我们仍可以获得用户信息和部分聊天记录，但是获取的结果不如在前两种情境下详细。比如用户信息只有陌陌 ID、聊天记录只有内容和聊天双方距离等等。

通过内存取证，我们获得了一些在用户界面或非易失性内存中无法查看的信息，比如被删除的聊天记录、在应用中登出后曾登录用户的痕迹等等，这体现了内存取证的重要价值。

3.3.3. 微信（com.tencent.mm）（支付部分）内存镜像分析

微信是目前中国最为普及的聊天社交支付一体的 APP，微信支付也被到了广泛的使用和关注。由于在 3.3.2 节中，我们已经对聊天社交类软件陌陌进行了分析，因此，在本节中，我们只对微信的支付部分进行取证分析。我们主要分析两个方面，一是支付账单与交易记录，二是绑定银行卡记录。

3.3.3.1. 用户绑定银行卡记录提取

微信中的银行卡记录包含用户的卡号、真实姓名等有效信息。对银行卡信息进行内存取证也有着重要的意义和价值。内存中银行卡信息存在的基本结构为：

*"bank_name":"中国银行","bank_phone":"95566","bank_type":"BOC_DEBIT", "bankacc_type":"1","bind_day_quota":"0","bind_once_quota":"10000000","bind_serial":"40630979740535978298","bind_tail":"8096","day_quota_1":"300000","day_quota_3":"1000000","expired_flag":"0","mobile":"138******07","once_quota_1":"300000","once_quota_3":"1000000","bank_card_tag":"1","extra_bind_flag":"NORMAL","bind_icharacter4":"3","support_micropay":1,"support_save":"1","support_fetch":"1","fetch_pre_arrive_time":"1460735999","draw_available":"2","draw_type":"1","export_start":"7","export_end":"18","bankacc_type_name":"储蓄卡","draw_status":"SR^Y|T0^Y|T1^Y","arrive_type":"0","maintain_starttime":"","maintain_endtime":"","fetch_pre_arrive_time_wording":"明天 24 点前到账"}],"virtual_card_array":[],"user_info":{"is_reg":"1","true_name":"白岚","bind_card_num":"1","icard_user_flag":"2","cre_name":"身份证","cre_type":"1","transfer_url":"https:\/\/wx.tenpay.com\/f2f?t=AQAAAPAEgDIn6OwQkNtlXq2Yz%2Bs%3D","touch_info":{"is_open_touch":"0","use_touch_pay":"0"},"lct_wording":"零钱理财*

因此，我们可以看出，内存中包含的银行卡信息有银行卡尾号、银行名、银行卡类型、持卡人真实姓名等信息。提取方法与 3.3.2.1 节中提取陌陌用户档案信息的方式类似。我们定义 card 类存储此信息：

```
class card:
    bank_name=""
    bank_type=""
    bind_tail=""
    mobile=""
    name=""
    def print_profile(self):
        ……
```

```
def write_profile(self,file):
    ……
```

并以 dictionary 数据结构存储。其中以银行卡尾号为键、card 对象为值。

3.3.3.2. 用户交易记录获取

微信交易记录与个人财产息息相关，在取证上有重要的意义。单笔交易记录和月度交易记录在内存中的基本结构为：

{"Transid":"4002962001201603234208448757","TotalFee":1288,"GoodsName":"大众点评-订单 1109081690","CreateTime":1458704550,"TradeStateName":"支付成功","ModifyTime": 1458704563,"FeeType":"CNY","StatusType":3,"PayType":0,"IapTotalFee":"","AppName":"","AppNickName":"","AppThumbUrl":"","StatusHead":"","StatusBody":"","ProductCount":0,"Products":[],"FeeColor":"#000000","StatusColor":"#888888","ActualPayFee":1288,"BillId":"a610f25620a1070028896277"}

{"year":2016,"month":3,"feetext":"消费：￥225.49 "}

因此，我们可以看出，内存中包含的单笔账单信息有订单编号、购买商品、应付金额、实付金额、创建时间、支付时间、订单状态等，月度消费信息包含时间与金额。提取方法与 3.3.2.1 节中提取陌陌用户档案信息的方式类似。我们定义 trans 类存储单笔消费账单、定以 month 类存储月度总消费：

```
class trans:
    id=""
    fee=""
    goods=""
    creat_time=""
    modified_time=""
    state=""
    type=""
    actual_pay=""
……
class month:
    year=""
    month=""
    money=""
……
```

对二者均以 Dictionary 类储存。分别以订单编号（id）、时间（year+month）为键、trans 或 class 对象为值。

3.4. 内存取证软件设计

为了方便调查取证人员通过我们编写的程序获取内存镜像，我们通过 C# windows 编程编写了一款内存取证软件。可以让使用者以简单的点击按钮的步骤获取内存镜像、并一键获

取分析结果。软件界面如图 3.4.1 所示。

软件功能主要分五个部分：查看连接手机状态、查看内存中的进程信息、内存镜像获取、内存镜像分析、查看结果与搜索。

3.4.1. 手机连接状态检查

本软件需要连接已经获取 root 权限的手机使用，因此，第一步我们需检测手机连接状态、手机基本信息和是否已获取 root 权限。我们通过 adb（Android Debug Bridge）来获取这类信息。而在 C#中使用 adb 命令，可使用 Process 类[20]。

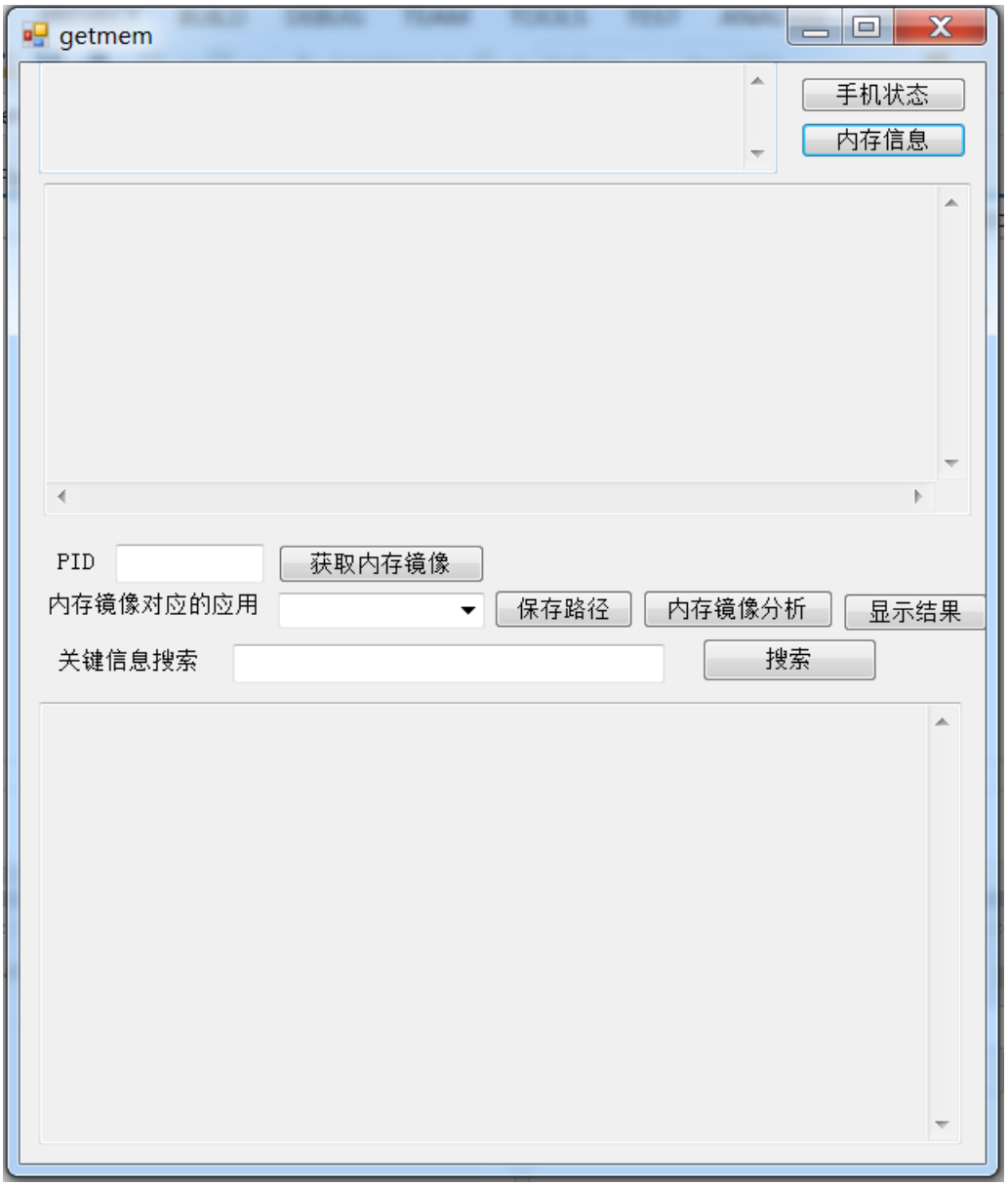

图 3.4.1 Android 内存取证软件界面

获取手机状态信息的基本命令为：

adb shell getprop + 需获取的内容

在本软件中，我们需要获取的手机信息与对应的命令为：

手机型号：*adb shell getprop ro.product.device*

手机制造商：*adb shell getprop ro.product.manufacturer*

序列号：*adb shell getprop ril.cdma.deviceid*

系统版本：*adb shell getprop ro.build.version.release*

Dalvik 堆大小：*adb shell getprop dalvik.vm.heapsize*

如果手机未连接，则这些命令不会返回标准输出，只会有错误输出，因此，可以以此判断手机是否连接。

判断手机是否已获取 root 权限，则需使用命令：

adb shell su –c

这一命令被用于不进入 adb shell 模式下直接以 root 权限执行-c 后的指令，在已获取 root 权限的情况下，若-c 后不加任何指令，则会得到标准输出：

option requires an argument – c …….

否则，则为未获取 root 权限。

图 3.4.2 在未连接手机、连接已 root 手机、连接未 root 手机后的单击“手机状态”按钮

在未连接手机、连接了已获取 root 权限的手机、连接了未获取 root 权限的手机后，单击“手机状态”按钮，软件运行结果如图 3.4.2 所示。

3.4.2. 内存中运行进程信息获取

由 3.2 节可知，本文中获取进程内存信息时需已知进程 pid。查看内存中进程及其 pid 的命令为：

adb shell su –c ‘dumpsys meminfo’

同时，我们根据 Android 进程的优先级层次分类及 *dumpsys meminfo* 的输出，将进程分成 Native、System、Persistent、Foreground、Visible、Perceptible、A Services、Home、BServices、Cached 类，并统计了每一类的进程数量。

用户可以在已连接手机的状态下单击内存信息按钮，获取内存中的进程及其 pid 等信息，如图 3.4.3 所示。

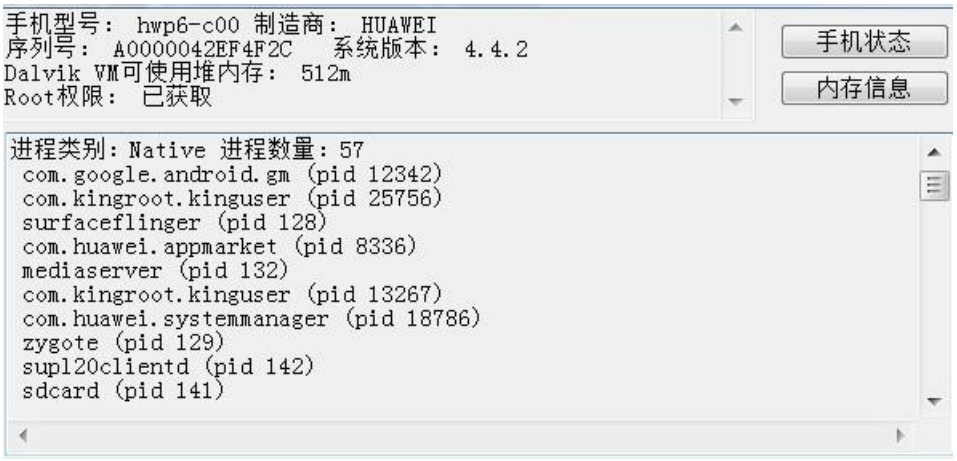

图 3.4.3 单击内存信息按钮后显示内存中的进程信息

3.4.3. 获取内存镜像

在运行软件之前，我们需要如 3.2.2 节所述，将 mem_heap 可执行文件导入 Android 手机/data 文件夹。在本软件中，我们同样通过 Process 类执行命令：

adb shell su -c 'chmod 777 /data'

adb shell su -c './data/mem_heap + id + sdcard/ + id + .mem

adb pull sdcard/" + id + ".mem " + Application.StartupPath

adb shell su -c 'rm sdcard/" + id + .mem'

其中 id 为进程 pid，Application.StartupPath 为软件运行路径。

用户可以在 PID 标签旁的文本框输入进程 pid，然后单击“获取内存镜像”按钮，内存镜像即会被保存至当前程序运行路径。

3.4.4. 内存镜像分析与关键信息搜索

用户在需要分析内存镜像时，必须先选择获取的镜像所对应的应用，随后单击“保存路径”按钮选择分析结果的保存路径，再单击“内存镜像分析”选择需要分析的内存镜像文件。随后，软件会根据用户选择的内存镜像对应的应用，运行对应的 python 脚本来分析内存镜像，并在分析完成后提示，如图 3.4.4 所示。

图 3.4.4 内存镜像分析

随后，用户可以单击“显示结果”按钮来显示内存镜像分析结果，如图 3.4.5 所示。

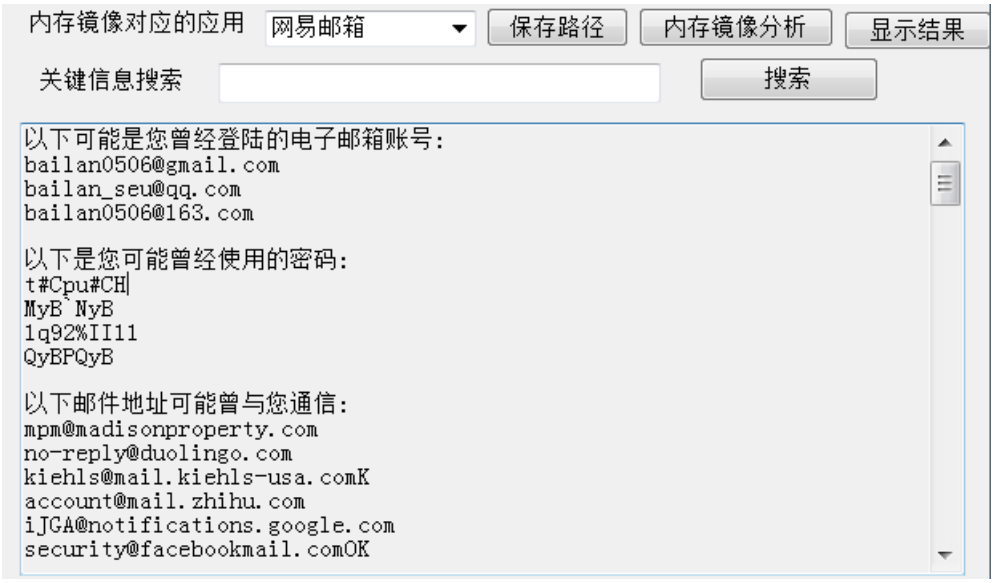

图 3.4.5 显示内存镜像分析结果

若用户发觉分析结果过多，可以在关键信息搜索的标识旁的文本框内输入需要的关键信息，并单击搜索按钮，即可检索得到需求的信息，如图 3.4.6 所示。

图 3.4.6 通过关键词在分析结果中搜索信息

3.5. 本章小结

本章分 Android 内存镜像获取、内存镜像分析、带有图像见面的取证软件设计这三部分对 Android 内存取证的软件实现进行了阐述。在内存镜像获取方面，我们对内存镜像的获取方法进行了优化，提高了内存镜像获取的效率。在内存镜像分析方面，我们通过编写脚本，在内存中获取了三款应用的用户数据，并且其中一些数据在某些场景下无法直接在手机上通过用户界面获取。最后，我们完成了一个易用的内存取证软件，用户可以通过此软件轻松获取内存镜像，并对其进行分析。可见，本章在 Android 内存取证研究上取得了一定的进展。

第四章. 全文总结与展望

4.1. 本文总结

本文对 Android 内存取证进行了研究，开发了 Android 内存取证软件，并且与前人相比，对取证的效率进行了一定的提高。现将本文的工作总结如下：

第一章为绪论。绪论对本文的研究背景和意义进行了详细深入的阐述，讲解了 Android 平台取证的发展现状，并在最后说明了本文的主要内容。

第二章为进一步研究 Android 平台内存取证提供了理论基础。该章介绍了 Android 系统与进程的基本概念，如系统的基本架构、进程的内存管理，为获取进程内存提供了理论依据。随后介绍的 Android 系统中的/proc 伪文件系统和 PTRACE 系统函数，以及 Android 设备通用内存获取方法的基本原理，为内存镜像获取提供了基本手段。最后介绍的常见字符编码与正则表达式，则是提取内存中的有效信息时的必备知识与工具。

第三章对 Android 平台内存取证进行了实现。首先，该章实现了对进程内存的获取，并且和前人的实现相比进行了改进，效率上实现了显著的提高。接着，该章对邮件类、即时通信类、支付类应用的代表网易邮箱、陌陌、微信进行了进程内存获取和分析。通过内存取证，我们获取了一些在用户界面中无法获得的信息。比如已经删除的邮件、聊天记录和已登出用户的邮件、聊天记录、用户名与密码等等，体现了内存取证的重要价值。最终，本章设计了一款方便易用的内存取证软件。该软件集内存获取、内存分析于一体，能让对计算机系统了解不深的调查取证人员轻松完成 Android 平台的内存取证，获取目标应用中的用户数据。

本章对本文所做的工作进行了总结。并对未来 Android 内存取证的进一步研究方向进行了展望。

总体来说，本文在第二章中选取了高通用性的内存镜像获取方案，并运用在第三章的实践当中，使本文所开发的取证工具适用于大多数已获取 root 权限的 Android 设备。在第三章中实现了高效提取内存镜像、一键分析内存镜像并取得非易失性存储器中较难获得的信息、并开发了带有图形界面的取证软件。因此，我们的取证研究成果具有通用性强、效率高、针对性强、简便易用的特点，对其他研究者的 Android 平台内存取证研究具有较高的参考价值。

4.2. 进一步的研究方向

本文中前三章的研究显示出 Android 内存取证的重要价值，然而，Android 内存取证目前来说并不是一个非常成熟的研究领域，仍然有许多问题值得进一步研究。

首先，在内存获取方面，有两点较为重要的可进一步研究的空间：

1. 目前，几乎所有的基于软件的 Android 内存镜像获取方式都需要获取 root 权限。这让取证者无法对未获取 root 权限的 Android 手机进行取证。
2. 本文所用的内存获取方法是对运行进程的内存进行获取，暂时不能对已经终止的进程内存或完整的物理内存进行获取。这给内存取证带来了一些局限性。

其次，在内存镜像分析上，可改进一步研究之处为：

1. 可对更多的应用进行更为详细的内存镜像分析，并提高其效率。
2. 可尝试提取一些目前出现在内存中，但是没有明显数据结构的信息。

3. 本文只尝试了对明文信息进行提取，可对内存中的密文信息提取进行研究。

致谢

首先向全程指导我毕业设计的胡爱群老师致敬。胡老师丰富的经验、严谨的治学态度、扎实的学术功底让我深受感触。他给予的指导帮我打开了学术研究的大门，让我了解到了如何在研究中发现有价值的问题，并且一步一步地钻研。这些指导，对我以后的学习与研究有着非常重要的意义！

感谢实验室的李涛老师、邢月秀学姐在我毕设遇到困难时给予我的帮助！

感谢实验室共同完成毕设宋昌、庄浩宇、徐昊、张俊芙等同学，大家互相的监督与鼓励是我前进的动力！

最后，感谢二十多年来父母对我的栽培！

參考文獻

[1] Ligh, Michael Hale, et al. The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory. John Wiley & Sons, 2014.

[2] Sylve, Joe, et al. "Acquisition and analysis of volatile memory from android devices." Digital Investigation 8.3 (2012): 175-184.

[3] Sylve, Joe. "Lime-linux memory extractor." Proceedings of the 7th ShmooCon conference. 2012.

[4] "Dd (Unix)." Wikipedia. Wikimedia Foundation. Web. 28 May 2016. https://en.wikipedia.org /wiki/Dd_(Unix)

[5] "Using DDMS." Android Developers. Web. 28 May 2016. http://developer.android.com /tools/debugging/ddms.html

[6] "Memory Analyzer (MAT)." Eclipse Memory Analyzer Open Source Project. Web. 28 May 2016. eclipse.org/mat

[7] Leppert, Simon. "Android memory dump analysis." Student Research Paper, Chair of Computer Science 1 (2012).

[8] Zhou, Fan, et al. "Dump and analysis of Android volatile memory on Wechat." Communications (ICC), 2015 IEEE International Conference on. IEEE, 2015.

[9] Tanenbaum, Andrew. "Modern operating systems." (2009).

[10] "Processes and Threads." Android Developers. Web. 28 May 2016. https://developer.android.com/guide/components/processes-and-threads.html#

[11] "Java Virtual Machine Technology." Java Virtual Machine Technology. Web. 29 May 2016. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/

[12] "Proc(5) - Linux Man Page." Proc(5): Process Info Pseudo-file System. Web. 29 May 2016. http://linux.die.net/man/5/proc

[13] "Ptrace(2) - Linux Man Page." Ptrace(2): Process Trace. Web. 29 May 2016. http://linux.die. net/ man/2/ptrace

[14] Gorn, Saul, Robert W. Bemer, and Julien Green. "American standard code for information interchange." Communications of the ACM 6.8 (1963): 422-426.

[15] de Normalisation, Organisation Internationale. "Universal Multiple-Octet Coded Character Set." (1992).

[16] Hoffman, Paul, and Francois Yergeau. UTF-16, an encoding of ISO 10646. RFC 2781, February, 2000.

[17] Thompson, Ken. "Programming techniques: Regular expression search algorithm." Communications of the ACM 11.6 (1968): 419-422.

[18] MobileForensicsResearch/mem." GitHub. Web. 29 May 2016. https://github.com/ MobileForensicsResearch/mem

[19] "WinHex: Computer Forensics & Data Recovery Software, Hex Editor & Disk EditorWindows XP/2003/Vista/2008/7/8/8.1/2012/10, 32 Bit/64 Bit*." WinHex: Hex Editor & Disk Editor, Computer Forensics & Data Recovery Software. Web. 29 May 2016. https://www.x-ways.net/winhex/

[20] "Process Class (System.Diagnostics) ".MSDN. Web. 29 May 2016. https://msdn.microsoft.

com/zh-cn/library /system.diagnostics.process.aspx

[21] 张瑜, et al. "内存取证研究与进展." Journal of Software 26.5 (2015).

[22] Bloks, Rudolf HJ. "The IEEE-1394 high speed serial bus." Philips Journal of Research 50.1 (1996): 209-216.

[23] "Memory Imaging ". ForensicsWiki. Web. 29 May 2016.http://forensicswiki.org/ wiki/Memory_Imaging

[24] MacIver, Douglas (2006-09-21). Penetration Testing Windows Vista BitLocker Drive Encryption (PDF). HITBSecConf2006, Malaysia: Microsoft. Retrieved 2008-09-23.

[25] Sierra, Kathy, and Bert Bates. Head first java. " O'Reilly Media, Inc.", 2005.

[26] Macht, Holger. "Live memory forensics on android with volatility." Friedrich-Alexander University Erlangen-Nuremberg (2013).

[27] Heriyanto, Andri P. "Procedures and tools for acquisition and analysis of volatile memory on android smartphones." (2013).

[28] 张辉极, and 薛艳英. "基于 Android 系统的取证技术分析." 信息网络安全 4 (2012): 58-60.

[29] Mahajan, Aditya, M. S. Dahiya, and H. P. Sanghvi. "Forensic analysis of instant messenger applications on android devices." arXiv preprint arXiv:1304.4915 (2013).

[30] "Memory Analysis for Android Applications ".Android Developers.Web. 29 May 2016. http://android-developers.blogspot.com/2011/03/memory-analysis-for-android.html

[31] 杜吉志, and 徐明昆. "Android 系统内存管理研究及优化." 软件 12(2012)

[32] 敬凯. "内在镜像数据图像证据分析." 学位论文.重庆邮电大学.(2012)

[33] 康磊. "内存镜像中文本数据提取方法研究." 学位论文.重庆邮电大学.(2013)