Live Memory Forensics on Android with Volatility

Diploma Thesis

submitted:	January 2013
by:	Holger Macht
student ID number:	21300176

Department of Computer Science
Friedrich-Alexander University Erlangen-Nuremberg
D – 91058 Erlangen
Internet: `http://www1.informatik.uni-erlangen.de`

Live Memory Forensics on Android with Volatility

Diploma Thesis in Computer Science

by

Holger Macht

born on 18 August 1982 in Hof a.d. Saale, Germany

at

Department of Computer Science
Chair of Computer Science 1 (IT-Security)
Friedrich-Alexander University Erlangen-Nuremberg

Advisors: Dipl.-Wirtsch.-Inf. Michael Spreitzenbarth
Dipl.-Wirtsch.-Inf. Stefan Vömel

Abstract

More and more people rely on smartphones to manage their personal data. For many, it has become a constant companion for a variety of tasks, such as making calls, surfing the web, or using location-based services. Common usage always leaves traces in the main memory which could turn out to become digital evidence that can be valuable for criminal investigations. Recovering such data artifacts out of volatile memory from mobile devices is known as *live memory forensics*.

Until now, there is no solution for performing *live memory forensics* on the Android platform by a comprehensive bottom-up approach. The approach presented in this thesis acquires the main memory from target devices to conduct further analysis. To gain knowledge about the layout of data in physical memory, the three central aspects of the Android platform are analyzed: The Linux kernel, the *Dalvik Virtual Machine* and a chosen set of applications. To create a thorough software solution, the work extends *Volatility*, an advanced memory forensics framework. The result is a set of plugins to read data such as user names, passwords, chat messages, and email. The thesis also identifies a guideline for additional application analysis and the corresponding plugin creation process. The overall outcome of this thesis a software stack that fits into the toolkit of every digital forensic investigator.

Contents

List of Figures

List of Tables

List of Listings

1. Introduction and Motivation

Smartphones are on the rise (ITProPortal, 2012). Nearly every second adult in the US owns at least one device. For many people, a smartphone has become a constant companion not just because of its traditional phone capabilities, but rather because of its *smart* features. Those include managing schedules with a calendar application, surfing the web, or using location-based services. By all means, many smartphones are used for organizing personal data. It is not a surprise that among the people who are using smartphones are also those people pursuing criminal activities. For investigators, data stored on smartphones is likely to contain evidence crucial for resolving a criminal case.

This evidence can either be stored in persistent memory or as live data in the system's main memory. The latter is typically lost when a device runs out of battery power or is shut off, making it harder to recover. Hence, a forensic investigator needs the abilities and proper means to recover such data from a mobile smartphone device. This field of forensic investigation is also known as *live memory forensics*.

As of now, the majority of smartphones are powered by the Android operating system. It reached a market share of 75% in the third quarter of 2012 (International Data Corporation, 2012). Until now, no research project has performed *live memory forensics* on the Android platform by a comprehensive bottom-up approach. This gap is filled by this thesis. The thesis is presenting an approach which creates a thorough software stack for forensic analysis of the Android operating system.

We start in Chapter 2 with providing the background about the three major areas this thesis is built upon. This includes a classification of the term *live memory forensics*, an overview about the Android platform, and an introduction of the popular memory forensics framework Volatility. After considering related research projects and an alternative *live memory forensic* approach, we define the goals of this thesis and the further approach of the project.

The approach is split into two main tasks: **Memory Acquisition** (Chapter 3), and **Memory Analysis** (Chapters 4, 5, and 6).

In Chapter 3, we depict the memory acquisition process. This is done by defining the prerequisites, including the preparation of the target smartphones and the setup of a development environment. This leads to the discussion of a concrete acquisition method and concludes with the overall acquisition process.

After memory acquisition, we start looking at memory analysis in Chapter 4. For this purpose, we give an introduction into the Volatility framework and how to make it work together with the acquired memory images. As Android is a Linux-based operating system, we discuss the existing support for Linux kernels by the framework. This enables us to read operating system data structures.

Chapter 5 focuses on the central software component in Android, the *Dalvik Virtual Machine*. We look at its internals and what data it actually contains. For this purpose, we discuss chosen aspects of the Java programming language and its implementation within Android. The major part of the chapter introduces *Dalvik Virtual Machine* support for Volatility by showing the creation process of corresponding plugins. With these plugins, forensic investigators can read virtual machine specific data and are empowered to gain knowledge about application internals. The latter is the foundation for the following chapter.

We complete the software stack for Android memory forensics in Chapter 6 by showing how to analyze specific applications. For this purpose, we present plugins to read user data such as account names, passwords, and chat messages. This also illustrates how the formerly created software components can assist a forensic investigator in creating additional application plugins.

The source code written for this thesis has been submitted upstream to the Volatility project. This is presented in Chapter 7. We also discuss possible future work and remaining challenges. The chapter concludes by comparing the initial goals to their actual outcome.

2. Background and Related Work

The first chapter provides an introduction to the major areas this research project is about. The first area is *forensics*, for which we will outline the commonalities and differences of the terms *digtial forensics*, *mobile device forensics* and *live forensics*. The second topic is *Android*, an operating system for mobile devices. We will briefly describe its origins, its structure and components, and the relevance for this project. The third area is *Volatility*, an advanced memory forensics framework and the building block the work performed in this thesis is based on.

After classification of the major areas, related research projects and papers are put into context. In the course of that, an alternative method doing forensic investigation of an Android device will be outlined briefly.

At the end of this chapter, the goals of this thesis are defined. This leads us to a description of the further strategy and development process.

2.1. Digital and Mobile Device Forensics

Forensic science is a broad subject. Even Wikipedia lists more than 30 subdivisions (Wikipedia, 2012f). Examples are forensic DNA analysis, forensic linguistics or forensic psychology. In general, Houck and Siegel describe forensic sciences as follows:

> "Forensic science describes the science of associating people, places and things involved in criminal activities; these specific disciplines assist in investigating and adjudicating criminal and civil cases." (Houck & Siegel, 2009, p. 4)

Amongst the different branches is the discipline of digital forensics. This specific field is about forensic investigations for digital devices, or the data found within. From a high-level perspective, digital forensic sciences can be divided into five separate tasks: Data recovery, data analysis, extraction of evidence and the preservation and presentation of that evidence (Carrier, 2003). This thesis includes three of these individual tasks: Recovery of data from a mobile device, in this case a smartphone, analysis of the same, and extraction of evidence. However, the latter is only touched by guessing what could be of interest from a legal perspective and does not go into detail.

Because data is also recovered from smartphones, the specific forensic branch targeted in this thesis is also knows as *mobile device forensics*. It evolved out of the traditional computer forensic science, because small embedded devices have other constrains and might require different tools for forensic data recovery (Punja & Mislan, 2008). However, due to the fact that storage, memory, and processor power of smartphones and traditional computers are converging, more and more similarities evolve. Thus, a differentiation becomes more and more difficult.

One major goal for digital forensic investigators is to recover data stored on devices such as hard disks, or flash storage. Those technologies have a common characteristic, namely being persistent storage devices. That means that the data stored on them is not lost when the power is removed from the system. The data is stored in a non-volatile fashion, for instance on platters.

On the contrary, this thesis deals with live memory acquisition and analysis. Instead of storage devices like hard disks or flash memory, volatile data which can be found in a system's main memory (RAM) is of major interest. Forensic investigations targeting this kind of data is also known as *live forensics* (Adelstein, 2006), or *RAM forensics* (Urrea, 2006).

In this context, *live* tries to express that the focus lies on a system's current state. A so called snapshot at a specific point in time. Adelstein (2006, p. 64) compares this to a photograph of a specific scene of a crime. To get such a system's state, instead of an image of a persistent storage device, a copy of the main memory has to be acquired. This need to be performed while the system is live, fully functional and operational. If that is not the case, possibly because the main power has been removed, the volatile data is lost. The main memory contains the whole state of an operating system, including running and historical processes, open network connections, management data, or personal data. Having memory images available for further processing, higher level information, usually represented by

C structures (in-kernel information) or Java objects (Android applications) can be analyzed. Because of the fact that the image of volatile memory is usually stored on non-volatile storage by a forensic investigator, there is no immediate risk of loosing the acquired evidence. Therefore, forensic investigations can be postponed until a controlled environment is available.

2.2. The Android Platform

Android originates from the equally named *Android, Inc.* (Wikipedia, 2012a). The company founded Android and was later purchased by Google in 2005. In 2007, it was presented to the public by the *Open Handset Alliance*, consisting of companies like HTC, Samsung, Qualcomm, Texas Instruments, and last but not least, Google. In October 2008, the first publicly available phone running the Android platform was released. Since then, various hardware companies, including some of those just named, created and sold countless smart phones and tablet devices based on Android.

Android is a software stack consisting of a Linux kernel, a middleware layer, a virtual machine called *Dalvik Virtual Maschine* which is able to run applications written in Java, and some core applications like an internet browser or a messaging application. Third party applications which make use of the available application framework can be created, too.

At the bottom of the Android software stack, Android is powered by a Linux kernel. All higher layers rely on the kernel's core services such as security, memory management, process management, network stack, and driver model (Google Inc., 2012). Although it is based on the mainline kernel from `kernel.org`[1], it is extended by a set of patches. The changes made to the standard Linux kernel include random bug fixes, kernel infrastructure improvements, new hardware support, and standalone kernel enhancements for higher layer elements such as applications (Brähler, 2010). Many vendors selling Android powered devices modify the kernel even further.

However, none of the kernel changes are of immediate relevance for both memory acquisition and memory analysis performed in this project. There is just one

[1]`http://www.kernel.org`

feature added to the kernel and considered indirectly related to this project. It is called the *Low Memory Killer*, as opposed to the *Out of Memory Killer* in standard Linux kernels, and added on top of it, not interfering with its functionality (Brähler, 2010). As soon as a system runs out of memory, the *Out of Memory Killer* sacrifices and kills one or more processes to free up some memory. In contrast to that, the *Low Memory Killer* kills processes belonging to an application before the system observes negative effects. It does that based on a priority which is attached to an application and its current state. The different states together with their priority are discussed in Section 6.1.3.

The central software component of Android is the *Dalvik Virtual Machine*[2], abbreviated as DalvikVM in the following. On Android, every application runs in its own DalvikVM. It is a similar concept as the Java Virtual Machine, with a few, but significant differences. For example, the byte code created from the source files are compiled into *.dex*-files instead of *.class*-files. Those are more optimized for the target devices like smart phones or tablet computers. Those *.dex*-files are created by a tool called *dx* which compiles and optimizes multiple Java *.class*-files into a single file. Together with a configuration file (`AndroidManifest.xml`), and non-source-code files like images and layout descriptions, the *.dex*-file is packaged into a Android Package file (abbrev. APK) (Shabtai et al., 2009). Those are the Android equivalents to *.jar*-files and can be installed on the target device. Basically, an *.apk*-file is a ZIP-compatible file representing a single application.

On top of the stack are applications. They are written in Java and provide the actual user functionality. Examples include applications for text messaging, internet browsers, calendars, or games.

For the purpose of development, debugging, testing, and system profiling, the *Android Software Development Kit*[3] (abbrev. *Android SDK*) is provided. Besides the API libraries to build Java applications, it includes developer tools such as the *Android Debug Bridge*, also known as *adb*, or the *Dalvik Debug Monitor Server*, abbreviated as *DDMS*. We will make use of these tools at a later point in this thesis.

Android's source code is released under an open source license via the *Android*

[2]`http://code.google.com/p/dalvik/`
[3]`http://developer.android.com/sdk/index.html`

Open Source Project (AOSP)[4]. This includes the kernel source, and any other higher level components. Some phone vendors even provide their own modified kernel source code for download. Being able to read the source code of the Android and DalvikVM implementation is essential for this research project, because it enables us gain deep knowledge about how data structures will be laid out in memory.

2.3. The Volatility Framework

After memory acquisition (cf. Chapter 3) has been successfully performed and thus, a file representing the physical memory of a system is available, we intend to extract data artifacts out of it. Without further ado, we would only be able to extract ASCII strings laid out in a contiguous fashion. However, this approach is limited, because we intend to parse visual elements and try to extract whole data objects from the Android system. Furthermore, we intend to create a solution that can be followed up upon. It should be generic, and thus, can be used by other developers, researchers and forensic investigators. For that purpose, will make heavy use of a popular forensic investigation framework called Volatility[5]. In fact, our work will be based on its infrastructure.

Volatility is a "Volatile memory artifact extraction utility framework" (Volatilesystems, 2012). It is completely open source, released under the GNU General Public License[6] and written in Python. This makes it possible to base the work performed in this research project on its source code and to publish the results. At the time of writing, Volatility contains official support for Microsoft Windows, Linux and Mac OS. Starting from version 2.3, it will also contain support for the ARM architecture, and thus Android. In this project, we will use a preliminary, but already functional, ARM support. Given a memory image, Volatility can extract running processes, open network sockets, memory maps for each process, or kernel modules. Additional information about the functionality of each module can be found on the project homepage[1]. Volatility has a public API and comes with an extendable plugin system which makes it easy to write new code, sup-

[4]`http://source.android.com/`
[5]`https://www.volatilesystems.com/default/volatility`
[6]`http://de.wikipedia.org/wiki/GNU_General_Public_License`

port more operating systems, and add support for extracting additional artifacts. This makes it the perfect choice for basing our work upon it.

However, another framework for supporting Linux memory forensics has been considered shortly. It is called *Volatilitux*[7] and was supposed to provide Linux memory forensics at a time Volatility lacked support for it. However, as of now, it has only limited capabilities such as enumeration of running processes. Furthermore, the last code change has been made in late 2011[8], so no active development community exists and, thus, it is unlikely that more functionality will be added soon. Because of the fact that no real plugin system is available, it makes it rather unsuitable for further extensions which are required for this research project.

2.4. Related Work

The research areas corresponding to the topic handled in this thesis can be separated into three different fields, with increasing specialization: *Live Forensics*, *Linux Live Forensics*, and *Android Live Memory Forensics*.

Live Forensics have been discussed in multiple papers. Hay et al. (2009) looks at the topic from a higher level, drawing a concrete distinction between static and live analysis. He also outlines the different possibilities for live analysis, also considering, but not solely, memory analysis. However, he does not provide a operating system specific solution. The same applies to Adelstein (2006) and his paper “Diagnosing Your System without Killing it First”.

Linux Live Forensics has been a research topic for several years. Yen et al. (2009) and Urrea (2006) focus on that area. The latter describe the underlying concepts of a concrete Linux distribution by outlining kernel structures relevant for memory management which can be used to retrieve corresponding evidence. He uses a tool called `dd` to read the physical memory from a file called `/proc/mem`. However, this way of physical memory retrieval is considered flawed by Sylve et al. (2012), because it alters the evidence in a too intrusive way.

Instead, Sylve et al. (2012) developed their own solution capturing memory from Linux-based, so also Android-based, systems. They also illustrate how to acquire

[7]`http://code.google.com/p/volatilitux/`
[8]`http://code.google.com/p/volatilitux/source/browse/`

some basic kernel data with the help of Volatility. Their paper "Acquisition and analysis of volatile memory from android devices" targets the field of *Android Live Memory Forensics* and serves as a base for this thesis. In the same area, Thing et al. (2010) describe a method of analyzing memory images in regard to communication. They developed a tool called `memgrab` to capture the memory regions belonging to a specific process. The memory regions can then be searched for known patterns corresponding to chat messages. However, they do not intend to solve the problem for acquiring the whole physical memory.

Leppert (2012) showed a way for Android live analysis with just looking at the heap of specific, running applications. Due to its actuality, his approach is outlined next.

2.5. Alternative Approach: Heap Dump Analysis

The approach performed by Leppert (2012) for Android application forensics by just investigating the application's heap can be split into two basic tasks:

1. Acquisition of the heap dump. This can be done with a tool called *DDMS* which is provided by the Android SDK. The resulting file has a special format called a *heap profile* (file extension: `.hprof`).
2. Analysis of the heap dump with a memory analyzer such as Eclipse MAT[9].
3. Post-processing of the data provided by the memory analyzer.

The result of step 2 typically is a large list of strings originating from all instantiated `java.lang.String` classed found in the application's heap. This list can be post-processed to find patterns which are likely to be data of forensic interest, such as account names and passwords.

While this is a valid approach, it contains some flaws we try to circumvent in this thesis.

First, acquiring a heap dump is only possible for applications prepared for debugging. When developing Android applications, there is a flag called `android:debuggable=` in the application's configuration file named `AndroidManifest.xml`.

[9]`http://www.eclipse.org/mat/`

When set to *true*, it causes the application to open a debug port whenever the application is started on the target device. This port can be used by *DDMS* to acquire a heap dump from an application running on a device which is physically connected to a computer system. While the corresponding debug option is typically set to *true* during times of development, it is supposed to be disabled when an application is released to the public. If set to *false*, *DDMS* has no means of acquiring the heap dump. Although there is a way to modify the value of the debug option after the application has been installed, it is not applicable to real-world scenarios. This would include transferring the corresponding *APK*-file to a PC, unpacking it, modifying the `AndroidManifest.xml`, repacking and resigning the application. However, this will cause the application to get restarted at least once, invalidating the heap it was formerly using and which was intended to be investigated.

Second, even if the heap dump can be acquired, the possibilities for further analysis are limited. Although it is possible to read plain strings, there is no way to acquire more sophisticated artifacts such as whole class objects or even binary data.

Third, the solution proposed in this thesis composes a general way for investigating an Android system together with certain applications. Once the corresponding plugins have been written, they do not depend on having a specific memory dump available. There is one plugin for a specific application and task which can be used independently from the available memory dump.

Last but not least, the solution created in the context of this research project empowers a forensic investigator to analyze arbitrary applications, whether he has access to the application's source code, or even the application's *APK*, at all.

However, none of the mentioned research projects provide a solution for live memory forensic analysis on Android in a general way. This will be solved by taking the whole software stack into account. The solution proposed in this thesis is based on, and consequently continues the work done by Sylve et al. (2012).

2.6. Test Setup and Development Environment

For performing the investigations and tasks outlined in this thesis, we made use of two smartphones. A Samsung Galaxy S2 (I9100) and a Huawei Honor (U8860). Both are running Android version 4.0.3, also known as *Ice Cream Sandwich*. Therefore, all findings and created software will be based on this version.

In case of Volatility, a branch called *2.3-devel* will be used. This branch already includes the ARM support targeted for the final 2.3 release. Thus, the created plugins should work with anything equal or greater than version 2.3.

In Chapter 3, we introduce a kernel module called *LiME*. The module version this thesis is based on is version 1.1.

Each application which is analyzed in Chapter 6 will have a specific version. Those will be defined on the spot.

2.7. Thesis Goals and Development Process

The goal of this thesis is to provide a way of analyzing the complete software stack of an Android-based device, from bottom, to top. At the bottom, we aim to depict a concrete method how to acquire the physical memory, while at the top we create solutions to investigate common Android applications. The work done in this project lays the foundation, provides infrastructure support and shows some exemplary plugins to continue upon. The following steps have been identified to be mandatory for a thorough investigation of the whole Android software stack:

1. Memory Acquisition from Android devices (Chapter 3)
2. Memory Analysis of the operating system's kernel (Chapter 4)
3. Memory Analysis of the *Dalvik Virtual Machine* (Chapter 5)
4. Memory Analysis of specific Android Applications (Chapter 6)

For Steps (1) and (2), we depict and make use of already available solutions. However, up to date, no solutions exist for Steps (3) and (4) which requires own evaluation of the concepts and underlying data structures. We investigate single

Dalvik Virtual Machine instances to be able to create different Volatility plugins to read their data in an application independent way. The gained information is then used to perform investigation of three popular Android applications to illustrate its usage. The outcome are plugins for each of those applications which can be used to perform real-world forensic investigations. In the end, we should have created a stack of plugins which makes it possible to perform forensic investigations for arbitrary applications with little effort. All source code is supposed to be released under an open source license and should be submitted to the corresponding upstream project for knowledge sharing and inclusion.

Figure 2.1 illustrates the document structure and development process used in this thesis.

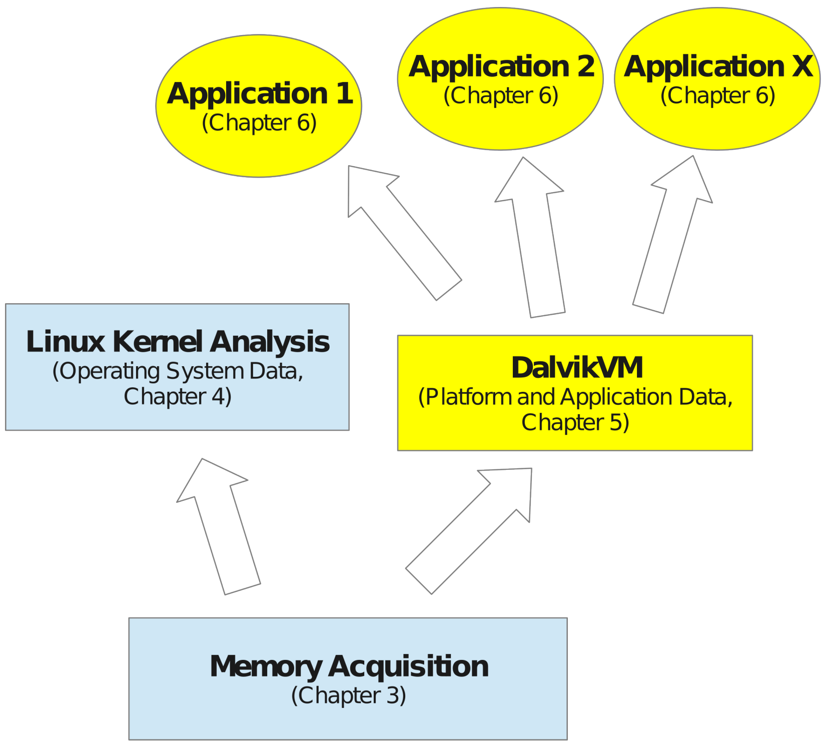


Figure 2.1.: Outline of the Forensic Investigation of the Android Software Stack

As a first step, the memory acquisition builds the foundation for all further analysis. Together with the preparation of the targeted mobile device and required software, Chapter 3 will outline the process of capturing memory images. In Chapter 4, we will evaluate different plugins for Linux kernel memory analysis which are already available, but crucial for every forensic investigation and for the forthcoming of this project. Their purpose is to read data from the Android kernel, which is a Linux kernel after all. We will look at their usage, functionality and underlying concepts. Chapter 5 depicts the internals of the data structures contained in the *Dalvik Virtual Machine* and we will create the necessary Volatility plugins to extract the corresponding artifacts. Together with extracting platform data common to all DalvikVM instances, we also gain application specific knowledge. This lays the foundation and provides the means for being able to analyze specific Android applications together with creating their task-specific plugins in Chapter 6.

3. Memory Acquisition from Android Devices

This chapter consists of three main sections. The first section will discuss available memory acquisition methods together with their obstacles. In the second section, we describe the required steps for properly preparing a smartphone for the tasks performed in the upcoming Chapters 4, 5, and 6. This includes physical access, unlocking the boot loader, *rooting*, and setting up a proper development environment. The third section depicts the concrete acquisition method with introducing a kernel module named LiME, how to make it available for the target device, and outlines the overall memory acquisition process.

3.1. Acquisition Methods

In order to analyze memory images, a copy of the RAM from a target device is required. There are two basic ways to acquire such data, by means of physical access, or remotely. However, no research making the latter possible on Android is currently known, so direct device access is the method of choice.

Sylve et al. (2012) evaluated local acquisition methods in their paper named "Acquisition and analysis of volatile memory from android devices". A kernel module called `fmem` which creates a character device to read the physical memory from was considered unsuitable, because it makes use of kernel functions which are not available on the ARM platform. Furthermore, the tool called `dd` which is used to read the memory from the character device is flawed on some Android devices and thus, cannot be used as a general solution. Another module which has been considered, but faces the same problems with `dd`, is called `crash`. Both methods use a solution divided into a kernel space (the module) and a user space (`dd`) part. Due to the resulting context switches, Sylve et al. (2012) found out that only 80% of the original memory could be recovered, rendering both solution unsuitable. The outcome of their own research about a platform-independent acquisition method a is a newly developed kernel module called LiME. It will be loaded into the kernel running on the target device and a detailed description can be found in Section 3.3.1.

However, one common problem all modules have to face are the security mechanisms of the kernel running on the target device. The Linux kernel uses a security mechanism called *module verification* (Sylve et al., 2012). It is intended to prevent the kernel from accepting incompatible or possibly malicious code to be inserted into the operating system. For instance, checksum information is stored for each function or structure definition corresponding to the kernel the module is compiled against. When the kernel tries to load a module, it tries to match this information, and if it does not succeed, it prevents the module from being loaded. There have been efforts to bypass these checks in the past, however, none of the solutions is perfect (Sylve et al., 2012). After all, no research making it possible to load a module in a kernel-agnostic way is currently known.

Instead of looking for a generic solution, an alternative approach is to create a pool of precompiled modules. Every module in the pool is compiled against a specific kernel, basically there is one module for each device and Android version. This is feasible for every device for which the corresponding vendor releases the kernel source code together with its build configuration. When trying to acquire the memory from a specific device, the corresponding module could be transferred to the device, loaded into the kernel, and used for dumping the main memory.

For the ease of this project, an own kernel needs to be installed and used on the target device. The LiME module will then be compiled against this specific kernel and can be loaded into the target operating system.

3.2. Prerequisites

3.2.1. Phone Preparation

To be able to load a module into the kernel, we need to bypass some security mechanisms. In the case of Android, one of them is unlocking the boot loader to be able to install a custom kernel on the target device. Another one is gaining root privileges, which is also referred to as *rooting* or *jailbreaking* (Wikipedia, 2012b). In general, due to the fact the Android runs a Linux kernel, basically every possibility of gaining administrator privileges for a Linux system also applies to Android. For instance, this can be accomplished by exploiting common security weaknesses in poorly written software, like it is done in a method called “Rage against the Cage” (Krahmer, 2010). What method works depends heavily on the device and the Android version it is powered by. The same applies to unlocking the boot loader, where every vendor might use a different technique to secure its boot process. Because of this, this thesis assumes that an unlocked, rooted device is already available, thus a concrete method will not be shown.

After rooting has been accomplished, the preferred method of transferring arbitrary binaries to the phone is using `adb`. It is a tool provided by the Android SDK and can also be used for debugging, diagnosis, and development. Once the phone has been physically connected (USB) to the development system, a custom kernel and the acquisition module (discussed in Section 3.3.1) can be transferred to the rooted phone and loaded into the operating system. The concrete method for exchanging the running kernel on a target device differs from vendor to vendor, so no common guideline can be provided at this point.

The development system needs to be capable of running `adb`. In order to use it, another thing needs to be assured: Android phones usually contain an option called *USB debugging*. It needs to be activated on the device. Starting from version 4 of Android, the corresponding switch can be found in *Settings->Developer Options* in the settings menu of the device.

3.2.2. Development Environment and Toolchain

Another prerequisite is the ability of the development system to provide a development environment and to produce software which is running on the target system. For this purpose, the Android SDK needs to be downloaded, installed, and configured correctly. The concrete steps include unpacking a compressed software archive to a common place and adjusting so-called *PATH-variables* for the binaries to be accessible from any location within a shell. From this point onwards, it is assumed that this has been accomplished, and a working copy of the development environment including compilers, an emulator, and the `adb` tool is available.

The precompiled tools provided by the standard Android SDK are supposed to run on a system powered by the x86 architecture. However, on most target devices this is not the case. Instead, as of now, ARM is quite common on devices running Android. This is the reason why all tools, modules, and binaries that need to run on the target device, need to be compiled on a host system for a specific target system first. This is also knows as *cross compilation* (Wikipedia, 2012d) and a *cross compiler* able to produce binaries for the target system is provided by the Android SDK. It is called `gcc`, the GNU Compiler Collection[1]. Before a kernel module can be compiled with the available toolchain, two environment variables need to be set on a common GNU/Linux system:

```
$ export ARCH=arm
$ export CROSS_COMPILE=$CCOMPILER
```

In this case, the variable $CCOMPILER contains the full path to the gcc binary. After the prerequisites have been set, the next step is to look at the actual memory acquisition, also known as memory imaging.

3.3. Memory Imaging

When trying to capture volatile memory from a mobile device, forensic soundness needs to be assured. A forensically sound process basically means that the method of gathering digital evidence alters the evidence as little as possible. The outcome should not be significantly different to the state before the memory acquisition.

Several methods of gathering volatile memory from Linux systems have been evaluated by Sylve et al. (2012). They come to the conclusion that none of the existing solution are feasible for the Android platform. The result, also in regard to forensic soundness, was the development of a kernel module called LiME, the *Linux Memory Extractor*[2], formerly known as DMD. The underlying principles, the usage, and the LiME image format will be briefly discussed in the next sections.

[1]http://gcc.gnu.org/
[2]http://code.google.com/p/lime-forensics/

3.3.1. LiME Kernel Module

The memory acquisition module used in this thesis is called LiME. It can be loaded into Linux kernels such as those running on Android devices to dump the physical memory either to a local file or over the network. LiME is the first module to allow full memory captures from Android devices (Sylve et al., 2012). To get more forensically sound results than with what was already available for common Linux systems, the authors payed special attention to minimizing interaction between kernel and user space during acquisition.

In order to acquire the physical memory from the operating system, the LiME module makes use of a kernel structure called `iomem_resource` (cf. Listing 3.1) to get the physical memory address ranges. Each `iomem_resource` has a field named `start` which marks the start of the physical memory and a field named `end`, which marks the end, respectively. Furthermore, the specific I/O memory resources which represent the physical memory regions are tagged by a field `name` which value has to be "System RAM". This is the case on at least X86 and ARM architectures. Memory images can either be written to a SD card attached to the target device or can be dumped via TCP to a host computer (Sylve et al., 2012). The latter is the method of choice in this thesis (cf. Section 3.4).

Listing 3.1: struct iomem_resource

```
struct resource iomem_resource = {
        .name    =          ,
        .start   = 0,
        .end     = -1,
        .flags   = IORESOURCE_MEM,
};
```

3.3.2. LiME Image Format

The LiME module version 1.1 offers three different image formats a memory image can be captured in (Joe Sylve, 2012). The format used is determined by a parameter passed at the command line when the module is loaded. The *raw* image format just concatenates all system RAM ranges and writes them either to disk or over TCP. The second method is called *padded* and includes all non system address ranges in the output. However, those ranges do not contain their original content, it is replaced with 0s. This causes the output to become a lot larger than actually needed.

The third format called *lime* is discussed in more detail because it is the method of choice in this research project. The *lime* format has been especially developed to be used in conjunction with Volatility. It is supposed to allow easier analysis, because a special address space to deal with this format has been added to Volatility.

Every memory dump based on the *lime* format has a fixed-size header, containing specific address space information for each memory range. This eliminates the need of having additional padding just to fill up unmapped memory regions. The LiME header specification version 1 can be seen in Listing 3.2 (Joe Sylve, 2012).

Listing 3.2: LiME Image Format Header

```
typedef struct {
 unsigned int magic; // Always 0x4C694D45 (LiME)
 unsigned int version; // Header version number
 unsigned long long s_addr; // Starting address of physical RAM
 unsigned long long e_addr; // Ending address of physical RAM
 unsigned char reserved[8]; // Currently all zeros
} __attribute__ ((__packed__)) lime_mem_range_header;
```

3.3.3. LiME and Kernel Cross Compilation

To make use of the LiME module, it needs to be cross-compiled to run on the target device. Furthermore, a compatible kernel is needed which needs to get installed on the target, too.

In order to do so, the source code for a kernel including the appropriate drivers is needed. Fortunately, some vendors provide such a kernel. For instance, for the primary device used in this project, a Samsung Galaxy S2 running Android version 4.0.3, a compressed archive containing the Linux kernel can be downloaded from the Samsung homepage[3]. After unpacking the archive named `GT-I9100_ICS_Opensource_Update7.zip` to a directory, the commands need to be executed to compile the kernel:

[3]`http://opensource.samsung.com`

```
$ export ARCH=arm
$ export CROSS_COMPILE=<path to Android SDK toolchain compiler>
$ make u1_defconfig
$ make
```

The first two commands will export environment variables used by the build system to define the target architecture and used compiler. The first *make-command* configures the build system while the seconds starts the actual build process. A lot of information about this topic can be found on the web. One example for possible troubleshooting are the XDA developer forums[4].

After successful kernel compilation, the kernel image can be found at `arch/arm /boot/zImage`. To be able to use the kernel on the target device, a so called boot image (`boot.img`) needs to be created which includes the created kernel and an initial RAM filesystem used for booting. The easiest way to create it is by fetching the existing boot image from the target device and just exchanging the kernel. For flashing the `zImage` to the device, a flash tool is required. Popular ones are *Odin*[5] (MS Windows) and *Heimdall*[6] (Linux). After the new kernel has been installed on the target device, the next step is to compile the LiME module.

To do so, the source code can be downloaded from the homepage[7] as a compressed archive. After unpacking the tar ball and changing to the `src/` directory, a simple `make` command should result in a file name called `lime.ko`. This is the actual kernel module and should be copied to the device to an arbitrary location. `adb` is likely to be best suited for this task. For the further course of this thesis, it is assumed that the module can be found on a SD card attached to the device at `/sdcard/lime.ko`. Now that all prerequisites have been met, we are able to capture the actual memory images.

3.4. Acquisition Process

Now that the LiME module is available on the target device and can be loaded into a matching kernel, the next step is the memory image acquisition. In order

[4]http://forum.xda-developers.com/
[5]http://forum.xda-developers.com/showthread.php?t=1347899
[6]http://forum.xda-developers.com/showthread.php?t=755265
[7]http://code.google.com/p/lime-forensics/downloads/list

to dump the physical memory via TCP, the smartphone needs to be connected to the host computer via USB. Afterwards, a TCP tunnel is created via port forwarding on both the host and the target. The kernel module dumps the physical memory over this tunnel to the host system. Assuming the module can be found at `/sdcard/lime.ko`, the following sequence of commands need to be executed to dump the physical memory over TCP to a file on the host:

Host:

```
$ adb forward tcp:4444 tcp:4444 # port number 4444
$ adb shell # creating a shell on the target device
$ su # creating a root shell
```

Target:

```
$ insmod /sdcard/lime.ko
```

Host:

```
$ nc localhost 4444 > ram.lime
```

In the last step, `nc` (netcat) connects to localhost on port 4444 and writes the received dump to the file `ram.lime` in the current directory. This is the actual RAM dump and can be used for further analysis.

3.5. Summary and Outlook

In this chapter, we have seen that the available memory acquisition methods for mobile devices are not perfect. This continues to be an area for further research.

We depicted a feasible solution to perform physical memory acquisition from an Android-based mobile device. The resulting memory images will be post-processed in the remaining parts of this thesis. These parts focus on the actual memory analysis and start with *Linux Kernel Analysis*, which can be considered *Android Kernel Analysis* at the same time.

4. Linux Kernel Analysis

Chapter 3 outlined how to acquire RAM dumps from the mobile devices, while in this chapter we use the acquired memory dumps to show how to do Linux kernel analysis. At first, we still disregard Android-specific analysis and concentrate solely on Linux memory forensics. After all, all Linux plugins provided by Volatility are actual Android plugins, too, because they only parse kernel data structures. After depicting how to create Volatility profiles, we show what Linux support is already provided by the framework. The profiles are required in order to use the acquired RAM dumps together with the analysis framework. Afterwards, we introduce some available Linux plugins, their usage and underlying concepts. This will assist in understanding the way Volatility parses memory artifacts. Also, some of them will be used in a later chapter when doing DalvikVM and application analysis.

4.1. Prerequisites

The first prerequisite is an actual memory dump. For the purpose of this chapter, it can be captured from any system, no matter if Android or from a standard Linux distribution. To make the image work together with the Volatility framework, we have to create a so-called *profile*. Afterwards, a short section will briefly introduce the Volatility usage.

4.1.1. Creating the Volatility Profile

Before image analysis can happen, a profile needs to be created which can passed to Volatility on the command line. A *Volatility profile* is a set of *vtype definitions* and optional symbol addresses. The concept of *vtype definitions*, also known as *vtypes*, will be depicted in Section 5.2.1.

After downloading and extracting the source code from the Volatility homepage, some of the default included profiled can be seen in Listing 4.1.

Listing 4.1: Volatility Profiles

```
$ python ./vol.py --info

--------
LinuxDebian2632x86  - A Profile for Linux Debian2632 x86
VistaSP0x64         - A Profile for Windows Vista SP0 x64
VistaSP0x86         - A Profile for Windows Vista SP0 x86
VistaSP2x86         - A Profile for Windows Vista SP2 x86
Win2003SP0x86       - A Profile for Windows 2003 SP0 x86
Win2003SP1x64       - A Profile for Windows 2003 SP1 x64
Win2008R2SP0x64     - A Profile for Windows 2008 R2 SP0 x64
Win2008R2SP1x64     - A Profile for Windows 2008 R2 SP1 x64
Win2008SP1x64       - A Profile for Windows 2008 SP1 x64
Win7SP0x86          - A Profile for Windows 7 SP0 x86
Win7SP1x64          - A Profile for Windows 7 SP1 x64
WinXPSP2x64         - A Profile for Windows XP SP2 x64
```

There exist several profiles for Microsoft operating systems, while for Linux there is only one. There are two main reasons for that: First, Volatility development has started from a Microsoft Windows point of view. So because Linux and Android support is quite new, there are less corresponding profiles available by default. Second, there are a lot more different flavours of different kernels available in the Linux and Android market. For Windows, there is basically always just one most recent version available. Being it Windows 8, Windows 7 or Windows 2000. And even for those versions, there are basically only up to four or five service packs which makes the overall number rather manageable. At least when compared to Linux.

For Linux, there are always a number of competing distributions such as Fedora, openSUSE or Ubuntu. Every release they publish ships with a different kernel. The situation is even worse when looking at Android. Every single hardware vendor such as HTC, Samsung or LG, usually has multiple devices for sale. Each of those running a different kernel, which multiplies quite fast. The problem with that is that each kernel needs its own Volatility profile. This basically leads to the need for creating an own profile for every smartphone used for a forensic investigation.

The Volatility wiki[1] describes a profile as follows:

> "A Linux Profile is essentially a zip file with information on the kernel's data structures and debug symbols. This is what Volatility uses to locate critical information and how to parse it once found.[...]" (Volatilesystems, 2012)

A profile is a compressed archive containing two files: A `System.map` and `module .dwarf`.

`System.map`

The `System.map` file contains the name and addresses of static data structures in the Linux kernel. Depending on the kernel's build configuration, it is typically created at the end of the compile process (cf. Section 3.3.3). For this purpose, a tool called `nm` is executed, taking the compressed kernel image called *vmlinuz* as a parameter. Basically on all major Linux distributions, the `System.map` file can be found in the `/boot` directory alongside with the actual kernel. In the case of Android, were a special kernel is compiled, it can be found in the kernel source tree after successful kernel compilation.

`module.dwarf`

The `module.dwarf` file emerges by compiling a module against the target kernel and afterwards extracting the *DWARF* debugging information out of it. *DWARF* is a standardized debugging format used by some source level debuggers to establish a logical connection between a output binary and the actual source code (Eager & Consulting, 2007). The *DWARF* debugging information is generated by the compiler and included in the output. In case of reading RAM dumps, it can also be exploited to provide valuable information about structure and method layouts and is used by Volatility for that purpose.

In order to create the `module.dwarf`, a utility called `dwarfdump`[2] is required. The Volatility source tree contains a directory `tools/linux/` and running `make` in that directory should compile the module and produce the desired file.

[1] `https://code.google.com/p/volatility/wiki/LinuxMemoryForensics`
[2] `http://reality.sgiweb.org/davea/dwarf.html`

Creating the Profile

Creating the actual profile is then just one simple step:

```
$ zip <profile_name>.zip <path_to_module.dwarf> \
    <path_to_System.map>
```

The resulting *ZIP*-file needs to be copied to the Volatility source tree to `volatility/plugins/overlays/linux/`. If everything is successful, the profile should show up in the profiles section of the Volatility help output (cf. Listing 4.1). At all further invocations of Volatility, it has to be passed with the *–profile* switch as a parameter.

4.1.2. Volatility Usage

Volatility uses the notion of *address spaces* for supporting different kind of memory dumps. For instance, it can read plain files, images acquired via Firewire, ARM images, and last but not least, the LiME format which has been discussed in Section 3.3.2. It was explicitly added to support the interaction with the newly developed LiME module, introduced in the same section. Volatility will probe the file format in a stacked fashion one after another and will figure out the correct address space by itself. This does not require any specification at the command line.

When looking at individual plugins in Section 4, it is important to be aware of the basic concepts of how Volatility treats individual data objects. It uses an object factory to deal with structure definitions. To quote the Volatility wiki:

> "Objects are the base element in volatility, and any time that data is needed from an address space it will usually be accessed through an object.
>
> [...]Objects are designed to behave as one would expect their python equivalents to behave (so a value that should be an int, should behave and have the same functions applicable to it, as a normal integer).[...]" (Volatilesystems, 2012)

The source code Listing 4.2 shows a typical way of data accessed in Volatility.

The parameters of the `obj.Object()` function have the following meaning:

Listing 4.2: Volatility Object Usage

```
1 #!/usr/bin/python
2
3 task = obj.Object(theType = "task_struct", vm = self.addr_space,
    offset = task_addr)
4 print task.comm
```

- `theType`: The type of the object and thus the actual layout of the data structure referenced by it.
- `vm`: The address space in which the object can be found. In the beginning, all addresses are virtual and need to be translated to physical offsets inside the RAM dump. The given address space usually is the address space of the process the object is used in.
- `offset`: The offset within the object's address space.

In this example, we are referencing an object of type `task_struct` within the kernel's address space at offset `task_addr`. The third line in Listing 4.2 accesses the structure member named `comm`.

To access the member fields, Volatility is aware of the individual offsets. Those can either be given by so-called "vtype definitions" (cf. Section 5.2.1) or figured out automatically by parsing the involved header files (cf. `struct task_struct` in Listing 4.3). For instance, the member field `state` can be found at offset 0, the void pointer `stack` at offset 4 and `usage` at offset 8. These are explanatory values for an arbitrary compiler and a 32-bit architecture.

The first prerequisite when running a Volatility plugin is to specify a proper profile. It is given to Volatility at the command line via the *–profile* switch. Additionally, the actual RAM dump is required and given as a parameter to *-f*. At the end of the command line, the plugin name followed by plugin specific, but optional parameters are given. A typical command line call might look as seen in Listing 4.4.

Listing 4.3: struct task_struct

```
struct task_struct {
  volatile long state;
  void *stack;
  atomic_t usage;
  unsigned int flags;
  unsigned int ptrace;

  /* [...] */
  char comm[TASK_COMM_LEN]; /* executable name excluding path */
  /* [...] */
}
```

Listing 4.4: Typical Volatility Usage

```
$ ./vol.py --profile=ProfileName
           -f /path/to/memory/dump.lime
           plugin_name -p option_parameter
```

4.1.3. Volatility Plugin Structure

When creating Volatility plugins, and for the better understanding of existing ones, we need to be aware of the general plugin structure. The plugin framework in Volatility has been designed with a modular approach. The plugins should be usable as standalone components called from the command line. However, it should also be possible that they are used by each other. Every plugin is able to call another one. More important, the results from one plugin can be used for further processing.

Listing 4.5 illustrates the basic four elements each plugin consists of.

Listing 4.5: Volatility Plugin Structure

```
class plugin_name():
    def __init__():
        [...]
    def calculate(self):
        [...]
    def render_text(self, outfd, data):
        [...]
```

The first is the plugin's name. It equals the class name and defines the name the plugin will have in the help output and when called on the command line. Then there are three methods. `__init__()` is the constructor of the class object and typically used for calling the super class constructor or for registering plugin specific command line options.

To be able to use the plugins from both source code and the command line, each one is separated into a `calculate()` and a `render_text()` method. The first does the actual calculations, reading the data from the dump, parsing it, preparing it for output. This is the method which is called from other plugins when no direct output and just the results are required. The `render_text()` function is called by the underlying framework whenever there is the request to print the data on the screen, typically when called from the command line.

All the Volatility plugins reside in the directory `volatility/plugins/linux/` within the source tree. New ones need to be placed in this directory, too, in order for the framework to find them.

4.2. Volatility Plugins for Linux

After showing the existing Linux support in Volatility, we will look at a chosen set of Linux plugins, together with their underlying data structures, implementation and output.

4.2.1. Existing Linux Support in Volatility

Although Android support in Volatility is quite new, Linux support is not. Because of that, a number of corresponding Linux plugins are already available.

They are listed in Listing 4.6 which shows an excerpt from the Volatility help output.

Listing 4.6: Volatility Linux Plugins

```
$ python ./vol.py --info

Plugins
-------
linux_arp            - Print the ARP table
linux_cpuinfo        - Prints info about each active processor
linux_dmesg          - Gather dmesg buffer
linux_dump_map       - No docs
linux_ifconfig       - Gathers active interfaces
linux_iomem          - Provides output similar to /proc/iomem
linux_lsmod          - Gather loaded kernel modules
linux_lsof           - Lists open files
linux_memmap         - Dumps the memory map for linux tasks.
linux_mount          - Gather mounted fs/devices
linux_netstat        - Lists open sockets
linux_pidhashtable   - Enumerates processes [...]
linux_proc_maps      - gathers process maps for linux
linux_psaux          - gathers processes [...]
linux_pslist         - Gather active tasks [...]
linux_psxview        - Find hidden processes [...]
linux_route_cache    - Lists routing table
```

In the following, we will have a deeper look at the implementation and underlying concepts of a subset of these plugins. On the one hand, this will illustrate how Volatility makes use of kernel structures to read the forensic data. On the other hand, the still to be created Android plugins will make use of some of them.

4.2.2. Plugin: linux_pslist

The linux_pslist plugin enumerates all the running processes in the system, similar to the Unix command `ps`:

Listing 4.7: Example linux_pslist Output

```
$ ./vol.py [...] linux_pslist

Volatile Systems Volatility Framework 2.1_rc3
Offset     Name              Pid Uid   Start Time
0xe6880000 init                1   0   Wed, 26 Sep 2012 17:01:21
```

```
0xe6880520 kthreadd          2    0   Wed, 26 Sep 2012 17:01:21
[...]
0xd9e4e3e0 .MtpApplication 4755  1000 Wed, 26 Sep 2012 17:03:25
0xd13723e0 id.defcontainer 4809 10091 Wed, 26 Sep 2012 17:03:49
0xd5337860 android.musicfx 4838 10140 Wed, 26 Sep 2012 17:03:55
```

For this purpose, Volatility makes use of the `init_task` structure defined within `init/init_task.c` in the Linux kernel source tree. It is a `task_struct` structure declared in `include/linux/sched.h`. The member field `tasks`, which is the head of a linked list, can be used to iterate over all running processes. In Python code, this looks as seen in Listing 4.8.

Listing 4.8: Shortened linux_pslist Implementation

```
1  init_task_addr = self.smap["init_task"]
2  init_task = obj.Object("task_struct", vm = self.addr_space,
       offset = init_task_addr)
3  for task in init_task.tasks:
4          yield task
```

In Line 1, the offset of the `init_task` structure within the memory dump is stored. It can be read from the `System.map` file provided by the profile (cf. Section 4.1.1). This is used in Line 2 to create an `Object` via the Volatility object factory. The `Object` represents a single task structure. Finally, the code walks the linked list containing all tasks and passes it to a output method via a generator function. The *linux_psaux* plugin not shown here uses a similar approach, just displays different information. The next plugin looked at is *linux_ifconfig*.

4.2.3. Plugin: linux_ifconfig

linux_ifconfig simulates the Linux `ifconfig` command. It lists the available network interfaces, together with their names, IP and MAC addresses (cf. Listing 4.9).

The corresponding source code (cf. Listing 4.10) first checks for the offset of `net_namespace_list` in the `System.map`, which is a kernel structure holding all network namespaces. Afterwards, it walks those network namespaces (Line 4) and iterates over all devices (Line 6), looking for those configured for the internet

Listing 4.9: Example linux_ifconfig Output

```
$ ./vol.py [...] linux_ifconfig

Volatile Systems Volatility Framework 2.1_rc3
lo        127.0.0.1        00:00:00:00:00:00
sit0      0.0.0.0          00:00:00:00:00:00
ip6tnl0   0.0.0.0          00:00:00:00:00:00
rmnet0    10.167.81.201    d6:b6:f6:00:e4:02
rmnet1    0.0.0.0          4a:23:45:80:05:05
rmnet2    0.0.0.0          00:00:00:00:00:00
```

protocol (cf. `in_device`), Wehrle et al. (2005)). In Line 7, the result is handed over to the output function.

Listing 4.10: Shortened linux_ifconfig Implementation

```
1 nslist_addr = self.smap["net_namespace_list"]
2 nethead = obj.Object("list_head", offset = nslist_addr, vm = self.addr_space)
3
4 for net in nethead.list_of_type("net", "list"):
5   for net_dev in net.dev_base_head.list_of_type("net_device", "dev_list"):
6   in_dev = obj.Object("in_device", offset = net_dev.ip_ptr, vm = self.addr_space)
7     yield net_dev, in_dev
```

4.2.4. Plugin: linux_route_cache

Together with knowledge about available and configured network interfaces (cf. Section 4.2.3), a forensically more interesting plugin is the *linux_route_cache* plugin. As the name might suspect, it reads and prints the route cache. This cache stores recently used routing entries in a fast hash lookup table. Thus, the goal is to extract this hash table out of an acquired RAM dump. When successful, this information could be used to make assumptions about possible network traffic happening in the past. The output Listing 4.11 shows three columns. The first is the interface name, the second the source, followed by the destination IP address.

Three symbols read from the `System.map` are of interest for the plugin implementation:

Listing 4.11: Example linux_route_cache Output

```
$ ./vol.py [...] linux_route_cache

Volatile Systems Volatility Framework 2.1_rc3
rmnet0 173.194.35.0    10.167.81.1
lo     127.0.0.1       127.0.0.1
rmnet0 195.71.11.67    10.167.81.1
rmnet0 212.18.3.18     10.167.81.1
rmnet0 212.18.3.18     10.167.81.1
rmnet0 63.116.58.131   10.167.81.1
rmnet0 173.194.35.9    10.167.81.1
```

- `rt_hash_mask`: Number of buckets in the hash. Due to the nature of the data structure hash, not every bucket needs to have a valid entry. This variable is initialized in `net/ipv4/route.c` in the kernel source tree.
- `rt_hash_table`: A pointer to the actual hash table. Also initialized in `net/ipv4/route.c`.
- `rt_hash_bucket`: Represents one single hash entry. A bucket contains a pointer to a structure named `rtable` (cf. `include/net/route.h`) containing all the routing information needed for the plugin.

The implementation in Listing 4.12 first creates the corresponding objects for the symbols found in `System.map` (Line 1 to 3). Starting from Line 5, the code checks all bucket slots and bails out if an entry is not valid. The name (Lines 11 to 14), the destination (Line 15), and the gateway (Line 16) are read and handed over to the output function in Line 18.

Listing 4.12: Shortened linux_route_cache Implementation

```
mask = obj.Object("unsigned int",offset = self.smap["rt_hash_mask"], vm = self.addr_space)
rt_pointer = obj.Object("Pointer", offset = self.smap["rt_hash_table"], vm = self.addr_space)
rt_hash_table = obj.Object(theType = "Array", offset = rt_pointer, vm = self.addr_space, targetType = "rt_hash_bucket", count = mask)

for i in range(mask):
  rth = rt_hash_table[i].chain
  if not rth:
    continue
  dst = rth.dst

  if dst.dev:
    name = dst.dev.name
  else:
    name = "*"
  dest = rth.rt_dst
  gw = rth.rt_gateway

  yield (name, dest, gw)
```

4.2.5. Plugin: linux_proc_maps

`linux_proc_maps` is the first plugin which will be directly required for implementing the Android and DalvikVM plugins in Section 5.1. When having direct access to a system running a shell, `/proc/$PID/maps` can be read for each individual process to acquire its memory mappings. Among others, the file lists the virtual memory addresses and access flags of the heap, stack, and dynamically linked libraries mapped into the process address space. The `linux_proc_maps` plugin reads the same information from a memory dump and tries to simulate the exact same output. An example is shown in Listing 4.13. Called without a parameter, the plugin lists the process maps of all running processes by making use of the *linux_pslist* plugin seen in Section 4.2.2. Given a parameter `-p`, the output can be limited to a specific PID.

The implementation (cf. Listing 4.2.5) iterates over all mappings found in either all or one task in Lines 1 to 4. It passes the results over to the output function. In order to do so, it uses the `walk_internal_list()` helper method. In source code

Listing 4.13: Example linux_proc_maps Output

```
$ ./vol.py [...] linux_proc_maps -p PID

Volatile Systems Volatility Framework 2.1_rc3
0x8000- 0xa000         r-x      0 259: 1   302 5398 /system/bin/
    app_process
0xa000- 0xb000         rw-   8192 259: 1   302 5398 /system/bin/
    app_process
[...]
0x40c30000-0x40c31000 rw- 303104 259: 1   902 5398 /system/lib/
    libdbus.so
0x40c31000-0x41513000 rw-      0   0: 4 4153 5398 /dev/ashmem/
    dalvik-heap
[...]
0x408f9000-0x409aa000 r-x      0 259: 1   915 5398 /system/lib/
    libdvm.so
0x409aa000-0x409b2000 rw- 724992 259: 1   915 5398 /system/lib/
    libdvm.so
```

Listing 4.14: Shortened linux_proc_maps Implementation

```
 1 for task in tasks:
 2   if task.mm:
 3     for vma in linux_common.walk_internal_list("vm_area_struct",
           "vm_next", task.mm.mmap):
 4       yield task, vma
 5 [...]
 6 def walk_internal_list(struct_name, list_member, list_start,
       addr_space = None):
 7   if not addr_space:
 8     addr_space = list_start.obj_vm
 9
10   while list_start:
11     list_struct = obj.Object(struct_name, vm = addr_space, offset
           = list_start.v())
12     yield list_struct
13     list_start = getattr(list_struct, list_member)
```

Line 11, a `vm_area_struct` object is created before reading the next item `vm_next` embedded in that structure. `vm_area_struct` is a kernel structure located in `include/linux/mm_types.h`.

4.3. Summary and Outlook

In this chapter, we showed how to use memory images with the Volatility framework by creating the corresponding profiles, one for each system and kernel version. We then introduced some commonly used Linux plugins. Some of them will help to analyze applications in Chapter 6, others will be directly used by the new plugins created in the same chapter.

Everything shown up to here is common to all systems running a Linux kernel. The further course of this thesis will focus on Android. We will have a look into the internals of the DalvikVM and will create plugins specifically designed to support forensic investigations of DalvikVM instances.

5. Android DalvikVM Analysis

This chapter can be split into two central aspects:

Section 5.1 will outline some concepts of the DalvikVM, what data is stored in each instance, and how Java classes are implemented. For this purpose, a short supplement will depict the concept of *Java reflection*. We will also look at the way how to recover instance objects from physical memory with reverse engineering their implementation.

The second part is Section 5.2, which will concentrate on the Volatility framework. We show the basic structure of plugins, depict how Volatility is aware of certain objects and their layout in memory, and illustrate how to use that information to parse Java objects. After describing the purpose of some helper functions, we will finally create plugins to analyze DalvikVM instances corresponding to concrete applications.

5.1. DalvikVM Analysis

The memory images acquired in a previous chapter represent the state of the system it was in at the time of acquisition. They contain the whole application's state and all its data, including the one from the virtual machine it is running in. This section will convey the underlying concepts and how to parse the internal data structures contained in the DalvikVM implementation.

5.1.1. Sharing Classes Amongst DalvikVM Instances

Each application runs in its own DalvikVM. However, certain data, such as often used classes and static class information are shared amongst all processes. The first process which is started and runs in a DalvikVM is called *zygote*. It is started at boot time and preloads and preinitializes classes which are later shared with other processes. Those classes are called *system classes* and referred to as such in the further course of this thesis. The *zygote* process is also responsible for forking further DalvikVM processes. It listens on a Unix socket for requests on starting applications and forks, using itself as a kind of template. Whenever a forked process reads data from a system class, the class information remains in the *zygote's* shared memory space. However, if a system class is accessed in a read-write fashion, the class information is copied over into the process own address space. This copy-on-write behaviour is different from the traditional Java Virtual Machine concept, where each process gets a full copy of all classes it requires (Bornstein, 2006). This knowledge is useful for understanding the forensic techniques used at a later time in this thesis.

5.1.2. Data Contained in the Dalvik Virtual Machine

The first question to answer is about what should be extracted out of an existing memory image. More specifically, the question is about what data is actually contained in it. Because every process on Android runs in a DalvikVM (cf. Section 2.2), this DalvikVM is a logical target as an entry point for a forensic investigation. The source code of Dalvik is publicly available, either as a standalone project[1] or as part of the Android Open Source Project. The latter is a more suitable choice, because for a specific Android version, the corresponding source code can be downloaded and investigated.

`DvmGlobals`

Case (2011) showed that a suitable entry point for extracting information out of a DalvikVM would be the object `DvmGlobals`. It is a structure available to every single DalvikVM instance and contains global data shared and used by application processes. An excerpt of its structure declaration can be seen in listing 5.1. The file it is declared in is `dalvik/vm/Globals.h` in the Android source tree[2].

`DvmGlobals` contains a lot of meta information for a specific DalvikVM instance. One that is required for this research project is a field called `loadedClasses`. It

[1] `http://code.google.com/p/dalvik/`
[2] `http://developer.android.com`

Listing 5.1: struct DvmGlobals

```
struct DvmGlobals {
  char*        bootClassPathStr;
  char*        classPathStr;

  size_t       heapStartingSize;
  size_t       heapMaximumSize;
  size_t       heapGrowthLimit;
  size_t       stackSize;

  /* [...] */

  /*
  * Loaded classes, hashed by class name.  Each entry is a
  * ClassObject*, allocated in GC space.
  */
  HashTable*   loadedClasses;

  /* [...] */

  /* field offsets - String */
  int          offJavaLangString_value;
  int          offJavaLangString_count;
  int          offJavaLangString_offset;
  int          offJavaLangString_hashCode;

  /* [...] */
}
```

is a pointer to a hash table containing all loaded classes known to this instance (cf. *system classes*). These classes contain meta information about their layout, size, and members which can later be used to access specific class instance data.

To summarize, a single DalvikVM instance belonging to one specific process contains the following information of interest:

1. A list of all loaded system classes.
2. Specific information about a single class, such as member fields, static variables, and method names.

Extracting this information together with creating a set of Volatility plugins for this purpose is the goal in the following parts of this chapter.

5.1.3. Parsing the Dalvik Virtual Machine

The Dalvik Virtual Machine is written in C++. As a consequence and for the further analysis, it is important to know that it uses simple C structures as its basic method to store internal data. Those simple structures have the big advantage of being laid out in memory in a nearly one to one relationship. The only thing to keep attention to is the structure alignment (Wikipedia, 2012e). Fortunately, if Volatility is aware of the correct structure definitions, it already provides proper means for accessing the encapsulated data objects (cf. Section 2.3).

The first challenge is to locate a concrete `DvmGlobals` instance in an existing memory dump taken from an Android device (cf. Chapter 3). For all we know, each process runs in its own DalvikVM, so each process has a corresponding `DvmGlobals` object mapped somewhere into its memory region. The DalvikVM is implemented as a shared library called *libdvm*, and it is dynamically loaded into every application process. Still, this information is quite vague, so to isolate the exact position, we look at the way it is defined in `/dalvik/vm/Init.cpp` within the source tree of Android (Case, 2011):

```
struct DvmGlobals gDvm;
```

Because `gDvm` is defined as an uninitialized variable, it will get placed into the BSS section (Wikipedia, 2012c) by the compiler, which in turn is inside the data section of where *libdvm* is mapped. Together with the output of the *linux_proc_maps* plugin, this information will be used in Section 5.2.4 for implementing the corresponding Volatility plugin. Once the `DvmGlobal` instance has been found in memory, it should be possible to access the member field called `loadedClasses` (cf. Listing 5.1). `loadedClasses` is a pointer to a hash table containing all the preinitialized system classes known to the process it belongs to. The hash table declaration can be seen in Listing 5.2.

The first noteworthy field is the `tableSize`. It is a simple integer and later used as delimiter to iterate over the whole table looking for valid entries until `numEntries` could be found. The next is `pEntries`, a pointer of type `HashEntry` (cf. Listing 5.3) pointing to an array allocated on the process heap. This array contains the actual entries, the loaded classes (`ClassObjects`). Each single entry

Listing 5.2: struct HashTable

```
struct HashTable {
    int         tableSize;       /* must be power of 2 */
    int         numEntries;      /* current #of "live" entries */
    int         numDeadEntries;  /* current #of tombstone entries */
    HashEntry*  pEntries;        /* array on heap */
    HashFreeFunc freeFunc;
    pthread_mutex_t lock;
};
```

Listing 5.3: struct HashEntry

```
struct HashEntry {
    u4 hashValue;
    void* data;
};
```

holds a `hashValue` and an actual data pointer which is able to point to arbitrary data (declared as `void`).

5.1.4. Supplement: Java Reflection

Before continuing into the details of how the DalvikVM implementation deals with Java objects, there needs to be some explanation of how Java implements its reflection[3] capabilities. Reflection provides the means of accessing meta data about an actual object, such as method names, instance fields, or static fields which can be queried at runtime. In Java, all objects but primitive types are implemented as references. Each reference type is derived from the `java.lang.Object`, which, for instance, provides the basic method `getClass()` used to query the actual class type at runtime (Oracle Corporation, 2012). In Java, every object belongs to a certain class object. This class object is also referred to as a system class throughout this document (cf. Section 2.2). A `ClassObject` inside the DalvikVM implementation is used to implement arbitrary class objects, and thus, its reflection features. Those are implemented in `java.lang.Object`, every object is derived from. Each object in the DalvikVM has a reference to the `ClassObject` it belongs to. It can also be considered as defining its type.

[3] http://en.wikipedia.org/wiki/Reflection_(computer_programming)

However, to store the actual meta data such as members and type information, the underlying virtual machine has to implement this in a certain way. How this is done for the DalvikVM is the subject of the following section.

5.1.5. The ClassObject

A `ClassObject` represents a single class and is declared in `dalvik/vm/oo/Object.h` in the Android source code tree. It contains information such as method and member names, the size of the object in memory, and its super class, just to name a few. It is used by the DalvikVM to implement arbitrary class objects used at the higher Java level and implements its reflection capabilities (cf. Section 5.1.4). It is the central point of interest when it comes to gather information about class objects and their concrete instances in memory at a later point in time. Listing 5.4 shows an excerpt of the data structure containing the most interesting items for forensic investigation performed in this project.

Listing 5.4: struct ClassObject

```
struct ClassObject {
  Object          obj;
  const char*     descriptor;
  DvmDex*         pDvmDex;
  size_t          objectSize;
  ClassObject*    super;
  int             interfaceCount;
  ClassObject**   interfaces;
  int             directMethodCount;
  Method*         directMethods;
  int             virtualMethodCount;
  Method*         virtualMethods;
  int             ifieldCount;
  int             ifieldRefCount;
  InstField*      ifields;
  const char*     sourceFile;
  int             sfieldCount;
  StaticField     sfields[];
};
```

The Class Descriptor (`const char* descriptor`)

The first important field in a `ClassObject` is the field named `descriptor`. It is a constant string containing the field descriptor of the referenced class (Lindholm et al., 2012).

The first character of the descriptor string defines the type of object. A “L” stands for a object reference, a “[“ for an array. Multiple “[“ characters specify the array dimension. Other characters specify primitive types. Table 5.1 lists all the possibilities.

Type	Character	Interpretation
B	byte	signed byte
C	char	Unicode character code point, encoded with UTF-16
D	double	double-precision floating-point value
F	float	single-precision floating-point value
I	int	integer
J	long	long integer
Lclass;	reference	an instance of a class named *class*
S	short	signed short
Z	boolean	true or false
[	reference	one array dimension

Table 5.1.: Java Type Specifiers

(Lindholm et al., 2012)

For instance, these are examples of possible descriptors:

(1) Ljava/lang/String;
(2) [Ljava/lang/Object;
(3) I
(4) Z

(1) depicts a reference to a `java.lang.String` object. (2) is an one dimensional array of references to `java.lang.Object` objects. (3) and (4) show primitive types, an integer and a boolean.

Every member of a class has a specific type, so the information contained in the descriptor will become valuable when there is the need to find and parse a specific object or primitive in physical memory.

Fields

There are two kinds of fields a class can have, instance, and static fields. Static fields are read and initialized from the underlying *Dex*-file and shared amongst all class instances using them. This way, they have only to be initialized and made available once. The `ClassObject's` member field `sfieldCount` contains the number of those static fields. Due to the fact that those can be discovered by other means than live memory forensics, they will not receive any further attention in this thesis.

Listing 5.5: struct Field

```
struct Field {
  ClassObject* clazz;
  const char* name;
  const char* signature;
  u4 accessFlags;
};
```

Listing 5.6: struct InstField

```
struct InstField {
  Field field;
  int byteOffset;


};
```

On the other hand, instance fields are bound to every single object instance. Their number can be read from `ifieldCount`, while `ifields` points to an array of `InstField` objects which are shown in Figure 5.6. Those objects have a pointer `clazz` as their first member, specifying the class the instance field was declared in. The member `name` is the variable or reference name, and the `signature` defines the type of the variable (cf. Table 5.1). `accessFlags` provides the access specifiers used for the higher level Java language such as *private*, *protected* or *public*.

The variable `ifieldRefCount` in `ClassObject` contains the number of object references, so it is always smaller or equal than `ifieldCount`. In the DalvikVM implementation at hand, the instance fields are sorted in a way that references always come first, followed by the primitive types, if there are any.

Methods

Next to fields, there are both direct and virtual methods. Both array pointers (`directMethods` and `virtualMethods`) point to `Method` structures seen in Listing

5.7. `directMethodCount` and `virtualMethodCount` depict the number of methods, respectively.

Listing 5.7: struct Method

```
struct Method {
  ClassObject*    clazz;
  const char*     name;
  const char*     shorty;
};
```

Inside the `Method` structure, there are three fields of interest. Again, the first `clazz` pointer points to a `ClassObject` the method is declared in. The `name` is just the name of the method while `shorty` specifies the signature in a short form similar to the descriptors seen in Table 5.1. The first character specifies the return value followed by the function arguments. For instance, a `shorty` string of "ZLI" specifies a method returning a boolean and taking an object reference and an integer as their two arguments.

Other Fields of Interest

There are more fields of interest in a `ClassObject`, though. `pDvmDex` points to the underlying *Dex*-file (cf. Section 2.2). `objectSize` holds the byte size of the whole object in memory. The `super` pointer holds a reference to the super class or NULL if there is none. The `super` pointer is needed when enumerating instance fields, because each `ClassObject` only holds the ones it defines directly. To also get those of the parent classes, the `super` classes need to be traversed.

`sourceFile` contains a string naming the underlying source file the corresponding class is implemented in. `interfaceCount` contains the number of interfaces this specific class implements directly, while `interfaces` holds the corresponding array. Most of the described fields will be used when creating the Volatility plugins in Section 5.2.

5.1.6. DalvikVM Implementation of Java Objects

The DalvikVM is the virtual machine running the Android Java applications (Lindholm et al., 2012). For this to accomplish, it needs to map the higher level

Java classes into data objects written in C++. The DalvikVM operates on two kinds of types, *primitive types* like integers or characters, and *reference types* representing Java objects. Looking at the DalvikVM implementation, each Java object is either a data, string, or an array object. The corresponding declarations can be seen in Figure 5.8, 5.9, 5.10 and 5.11. A structure `Object` always has a pointer `clazz` as its first field, representing the corresponding system class the object belongs to. The following unsigned integer `lock` is used for synchronization purposes. `StringObject` and `DataObject` are basically the same, except that the former has additional access methods that are not of further relevance in the scope of this thesis. The contained field `instanceData` holds a pointer (a reference), to the actual instance class object or primitive type implemented.

For instance, a Java `java.lang.Array` object will be mapped to a `DataObject`, and the `instanceData` pointer will in turn point to an `ArrayObject` (Yan & Yin, 2012).

The structure layouts are quite important because they can be exploited to draw conclusions about how the objects are laid out in memory. After all, they are simple C/C++ structures. For instance, when looking at the `ArrayObject`, the `clazz` pointer would be at offset 0 from the beginning of the structure, `length` at offset 8 and contents at offset 12. This information is later used when creating the structure definitions for Volatility in Section 5.2.1.

Listing 5.8: struct Object

```
struct Object {
  ClassObject* clazz;
  u4 lock;
};
```

Listing 5.9: struct DataObject

```
struct DataObject {
  Object obj;
  u4 instanceData[1];
};
```

Listing 5.10: struct StringObject

```
struct StringObject {
  Object obj;
  u4 instanceData[1];

};
```

Listing 5.11: struct ArrayObject

```
struct ArrayObject {
  Object obj;
  u4 length;
  u8 contents[1];
};
```

Although each Java object will be mapped to a `Data-`/`String-` or `ArrayObject`, there is even a smaller entity for storing data, a `JValue`.

The JValue

The `JValue` structure is the smallest unit for storing data in a DalvikVM. It is defined in the Android source tree in `dalvik/vm/Common.h` and can be seen in Listing 5.12.

Listing 5.12: struct JValue

```
union JValue {
    u1      z;
    s1      b;
    u2      c;
    s2      s;
    s4      i;
    s8      j;
    float   f;
    double  d;
    Object* l;
};
```

A `JValue` implements the JValue object pattern (Riehle, 2006) and combines both primitive types and object references into a single data object, namely a C union. It is the smallest object type a DalvikVM is aware of when implementing the higher level Java objects. Every value which is supposed to be read from physical memory is represented by a `JValue`.

5.2. Volatility Plugins for the Dalvik Virtual Machine

After outlining both the basics of the Volatility framework together with the concepts and implementation details of the DalvikVM, the next step is to transform that knowledge into source code. The following prerequisites discussed earlier should be already met:

1. Availability of one or more memory dumps from a phone running Android (cf. Section 3).

2. A working Volatility tree with support for Android (cf. Section 2.3).
3. A working Volatility profile consisting of a `System.map` and a `module.dwarf` debug file (cf. Section 4.1.1). Both files need to match the kernel which was running at the time the memory dump was acquired.

In the following sections, we will describe the development process of specific Volatility plugins for analyzing the Dalvik Virtual Machine. For that purpose, we outline the general layout of Volatility plugins to fit the underlying framework, show how to parse Java structures in memory, and introduce some of the created helper functions.

All plugins that are created use a common file name prefix *dalvik_**. For instance, plugins might be called *dalvik_find_gdvm_offset* or *dalvik_class_information*, depending on their functionality. In this chapter, we only show excerpts of relevant source code. The full source code can be found on the accompanying CD.

5.2.1. VType Definitions

vtypes are a concept in Volatility which describe the structures and their layouts used in an operating system. This includes their sizes, types, members, and their corresponding offsets. For instance, in Listing 5.13, the *vtype* definition of the `HashTable` structure is depicted that was previously described in Listing 5.2.

Listing 5.13: Volatility VType Definiton

```
1     'HashTable' : [ 0x18, {
2         'tableSize' : [ 0x0, ['int']],
3         'numEntries' : [ 0x4, ['int']],
4         'numDeadEntries' : [ 0x8, ['int']],
5         'pEntries' : [ 0xc, ['pointer', ['HashEntry']]],
6         'freeFunc' : [ 0x10, ['address']],
7         'lock' : [ 0x14, ['int']],
8         }],
```

The first line defines the name of the structure and its overall byte size, 0x18 (hexadecimal) or 24 (decimal) in this case. This is the whole space the structure will occupy in the physical memory image. Within Lines 2 to 7, the member fields are defined. The first element in each of those lines is the name, followed by a colon separating it from the type size and definition. For instance, `tableSize` is at offset 0x0, `numEntries` at offset 0x4, 4 bytes further. Line 5 contains a member named `pEntries`, specifying a pointer to a `HashEntry` structure which can be found at offset `0xc`.

The most convenient method for creating these definitions would be by some automatic means. One example for that are the `System.map` and the *DWARF* module in the Volatility profile (cf. Section 4.1.1). It contains the basic data structure definitions for the Linux operating system and is used in the plugins shown in Section 4. No manual *vtype* creation is required there. However, all the structures contained in the DalvikVM need proper *vtype* definitions, too. These have been created manually, and the full list can be found in appendix A.2. It lists the file called `dalvik_vtypes.py` which needs to be placed in the directory `volatility/plugins/overlays/linux/` inside the Volatility source code tree. In addition to providing the plain *vtype* structure definitions, it also contains some helper functions which can be attached to single objects. For instance, the `HashTable` structure is extended by a method called `get_entries()` which is a generator function parsing and returning the single entries contained in the `pEntries` pointer. How those functions and the vtype definitions are used will be shown next.

Usage within Volatility

As soon as the *vtype* definitions are created, they can be used within the Volatility source code. For illustration purposes, an example reading and parsing a `HashTable` object is provided in the Python code example in Listing 5.14.

Listing 5.14: Usage of VType Definitions in Volatility

```
1  gDvm = obj.Object('DvmGlobals', offset = 0x7123)
2  print gDvm.loadedClasses.numEntries
3  for entry in gDvm.loadedClasses.dereference().get_entries():
4      clazz = obj.Object('ClassObject', offset = entry)
```

It uses several objects and combines some of the previously discussed concepts. As shown in Line 1, a `DvmGlobals` object found at an arbitrary offset is created and stored in a variable named `gDvm`. The second line accesses the `loadedClasses` (a `HashTable`) member field in `gDvm` and prints the `numEntries` integer member field. The third line iterates over the list of entries found in the `HashTable`. To do so, it dereferences the `loadedClasses` pointer within `gDvm` and then uses the `get_entries()` helper function to "generate" the individual entries. Due to the fact that the above hash table contains `ClassObject` entries (cf. Section 5.1.3), in the forth line, a Volatility object representing these class objects is created. Each object can be found at offset `entry`, which is the loop element.

5.2.2. Parsing Java Structures in Memory

Every Java object, i.e. a `java.lang.String` or a `java.util.ArrayList`, needs a separate implementation inside the DalvikVM. This leads to different representations in physical memory. When trying to extract data from memory images, these objects need to be reverse engineered in order to get their data. This will be frequently required when creating the applications specific plugins in Chapter 6.

For this purpose, some helper functions have been created and are provided by a file called `dalvik.py` residing in `volatility/plugins/linux/`. One of the most frequently used functions will be `parseJavaLangString(stringObj)` and thus, we briefly describe its implementation to illustrate the underlying concepts. All the helper functions together with their full source code listing can be found on the accompanying CD.

`parseJavaLangString(stringObj)`

A Dalvik object representing a Java `java.lang.String` object has the following object members:

Field Name	Signature
`value`	`[C`
`hashCode`	`I`
`offset`	`I`
`count`	`I`

Table 5.2.: Member Fields of the DalvikVM's String Implementation

Although the Java Language Specification (Gosling et al., 2005) does not explicitly define the Java string implementation, it is typically implemented with the concept of a head and a body (Kawachiya et al., 2008). A graphical representation can be seen in Figure 5.1. The head is the DalvikVM `StringObject` while the body is the character array named `value` seen in the above table. The `value` is the unicode representation of the implemented string, and thus, has a length of two bytes for each character (The Unicode Consortium, 2011). In the DalvikVM, it is implemented as an `ArrayObject`. The `offset` specifies the location of the first character in the character array while the `length` defines the string length.

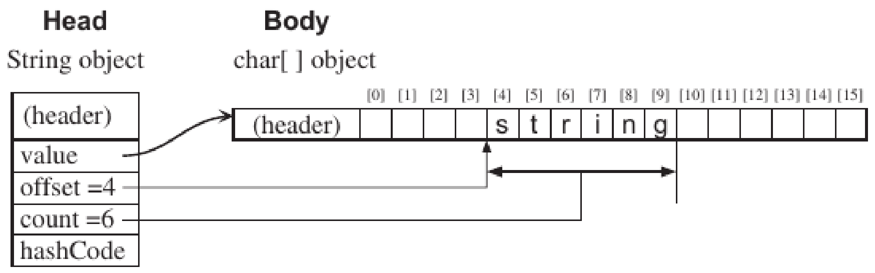


Figure 5.1.: Typical String Implementation in Java-based Virtual Machines

(Kawachiya et al., 2008)

The depicted string layout needs to be parsed from inside Python code which can be seen in Listing 5.15. From Line 5 to 12, the member fields `count`, `offset` and `value` are parsed. The `value` array is created in Line 14 while in Line 15, all parsed members are put together to create the final string.

Listing 5.15: Parsing a Dalvik String

```python
def parseStringObject(so):
  as = so.obj_vm
  c = so.clazz

  count_offset = so.byte_offset + c.getIFieldByName('count').
     byteOffset
  count = obj.Object('int', offset = count_offset, vm = as)

  offset_offset = so.byte_offset + c.getIFieldByName('offset').
     byteOffset
  offset = obj.Object('int', offset = offset_offset, vm = as)

  value_offset = so.byte_offset + c.getIFieldByName('value').
     byteOffset
  value = obj.Object('address', offset = value_offset, vm = as)

  arr = value.dereference_as('ArrayObject')
  string = obj.Object('String', offset = (arr.contents1.
     obj_offset+offset*2), vm = as, length = count*2)

  return string
```

5.2.3. Miscellaneous Helper Functions

There are a couple of other helper functions provided in the file `dalvik.py`. The ones considered most important are briefly described in the following table.

Function name	Description
parseJavaUtilArrayList(x)	Given an ArrayObject, it parses the referenced java.util.ArrayList and returns the entries with a generator function.
parseJavaUtilList(x)	Given an address pointing to a java.util.List, it extracts the underlying ArrayObject and hands the result over to parseJavaUtilArrayList().
parseArrayObject(x)	Checks if the given object is an ArrayObject and returns its elements as addresses.
getString(x)	Given an arbitrary object, it assumes that it is pointer to a string, creates a Volatility string object and returns it.
isDvmGlobals(x)	Given an arbitrary object, it checks if it is likely to be a DvmGlobals object and returns either true or false.
register_option_*(x)	When those functions are called by a plugin, they register a new command line parameter with the Volatility framework.
get_data_section_*(x)	Returns the address space start and end mappings for the data section of an arbitrary mapped element such as a dynamic library (cf. libdvm.so).
printChildren(x)	Takes a graphical layout object, printing its mChildren members. Can help parsing Android applications.

Table 5.3.: Volatility Plugin Helper Functions

5.2.4. Plugin: dalvik_find_gdvm_offset

The first plugin discussed is called *dalvik_find_gdvm_offset*. Like the name might suggest, its purpose is to locate the offset of the DvmGlobals objects within the data section of where libdvm.so is mapped into the process address spaces. As stated in Section 5.1.3, this is the base for any further DalvikVM analysis and serves as a base for the other plugins. The relevant code snippet is shown in Listing 5.16 and discussed in the following.

After initializing the basic variables, the helper function get_data_section_libdvm() (cf. Section 5.2.2) is used to iterate over all memory mappings of the given process (Line 6). It solely returns the memory

Listing 5.16: Plugin dalvik_find_gdvm_offset

```
 1 class dalvik_find_gdvm_offset(linux_common.AbstractLinuxCommand):
 2  def calculate(self):
 3    offset = 0x0
 4    mytask = None
 5 
 6    for task, vma in dalvik.get_data_section_libdvm(self._config):
 7      if not self._config.PID:
 8        if task.comm+"" != "zygote":
 9          continue
10      mytask = task
11      break
12 
13    proc_as = mytask.get_process_address_space()
14 
15    gDvm = None
16    offset = vma.vm_start
17    while offset < vma.vm_end:
18      offset += 1
19      gDvm = obj.Object('DvmGlobals', vm = proc_as, offset =
          offset)
20      if dalvik.isDvmGlobals(gDvm):
21        yield (offset - vma.vm_start)
```

mappings of the data section of `libdvm.so`. If no specific process PID has been specified on the command line, the first process we can be sure of running in a DalvikVM will be used. It is called *zygote* (Line 8, cf. Section 2.2). Starting from that position, the plugin scans and checks what is likely to be a `DvmGlobals` object (Lines 17 to 22). If found, it passes the found offset to the output function in Line 22. The output might look like in Listing 5.17.

Listing 5.17: Example dalvik_find_gdvm_offset Output

```
  $ ./vol.py [...] dalvik_find_gdvm_offset

DvmGlobals offset
-----------------
0x7c78
```

The found offset (0x7c78) is the offset from the start of the data section of the corresponding process and can now be passed to the other plugins.

5.2.5. Plugin: dalvik_vms

The purpose of the *dalvik_vms* plugin is to find all DalvikVM instances and to print required information contained in it. We assume that the `DvmGlobals` offset has been given on the command line. If that would not be the case, the *dalvik_find_gdvm_offset* plugin (cf. Section 5.2.4) could be used internally.

Listing 5.18: Plugin dalvik_vms

```
1  class dalvik_vms(linux_common.AbstractLinuxCommand):
2   def calculate(self):
3     offset = 0x0
4
5     gDvmOffset = int(self._config.GDVM_OFFSET, 16)
6
7     for task, vma in dalvik.get_data_section_libdvm(self._config):
8       gDvm = obj.Object('DvmGlobals', offset = vma.vm_start +
           gDvmOffset, vm = task.get_process_address_space())
9
10      # sanity check: Is this a valid DvmGlobals object?
11      if not dalvik.isDvmGlobals(gDvm):
12        continue
13      yield task, gDvm
```

The corresponding plugin code in Listing 5.18 saves the `gDvm` offset given on the command line in Line 5. It then walks the process mapping of `libdvm.so`, checking if a `DvmGlobals` object can be instantiated (Line 8 ff). If successful, it passes the task and the object to the output function which prints the structure members of `gDvm`. Those can be easily accessed due to the *vtypes* (cf. Section 5.2.1) being already available. An exemplary output can be seen in Listing 5.19.

It lists information about three DalvikVMs, together with their process IDs, and names they belong to. It also contains information about the number of preloaded classes. The next plugin will be used to list these, together with more detailed information.

5.2.6. Plugin: dalvik_loaded_classes

The *dalvik_loaded_classes* plugin lists all preloaded classes from a specific DalvikVM instance together with additional information. The most important bit is the

Listing 5.19: Example dalvik_vms Output

```
$ ./vol.py [...] dalvik_vms -o HEX

PID   name             heapStartingSize heapMaximumSize
----- ---------------- ---------------- ---------------
 2508 zygote                    5242880       134217728
 2612 system_server             5242880       134217728
 2717 ndroid.systemui           5242880       134217728

 stackSize  tableSize  numDeadEntries   numEntries
 ---------- ---------- ---------------- ---------------
      16384       4096                0            2507
      16384       8192                0            4123
      16384       8192                0            2787
```

class offset, which can later be used to list specific class information with the *dalvik_class_information* plugin discussed in Section 5.2.7.

Listing 5.20: Plugin dalvik_loaded_classes

```
1 class dalvik_loaded_classes(linux_common.AbstractLinuxCommand):
2   proc_maps = linux_proc_maps.linux_proc_maps(self._config).
        calculate()
3   dalvikVMs = dalvik_vms.dalvik_vms(self._config).calculate()
4
5   for task, gDvm in dalvikVMs:
6     for entry in gDvm.loadedClasses.dereference().get_entries():
7       clazz = obj.Object('ClassObject', offset = entry, vm = gDvm
            .loadedClasses.obj_vm)
8       yield task, clazz
```

In the first line of Listing 5.20, the plugin code uses the *linux_proc_maps* plugin (cf. Section 4.2.5) to get the correct process mappings for an arbitrary PID which has been specified on the command line. In the second line, the *dalvik_vms* plugin discussed before (cf. Section 5.2.5) is utilized to get a list of DalvikVM instances corresponding to the given process ID. The code then walks those tasks and DalvikVMs (Line 3), to get the concrete list of loaded classes (Line 6). For each of those classes, it constructs a `ClassObject` and passes it to the output function. The result might look like seen in Listing 5.21.

In addition to the Java descriptor (cf. Section 5.1.5) and source file the class was implemented in, the important information needed for further analysis is the

Listing 5.21: Example dalvik_loaded_classes Output

```
$ ./vol.py [...] dalvik_vloaded_classes -o HEX -p PID
PID   Offset     Descriptor                        sourceFile
---- ---------- --------------------------------- ----------------
4614 0x40c378b8 Ljava/lang/Long;                  Long.java
4614 0x40deb6d0 Ljava/io/Writer;                  Writer.java
4614 0x414e2f60 Lde/homac/Mirrored/ArticlesList; ArticlesList.jav
```

offset. It is the virtual address of the system class within its process address space and is required to list specific information about that single class. This is accomplished with the following plugin.

5.2.7. Plugin: dalvik_class_information

dalvik_class_information lists concrete information about a specific system class, such as the number of instance fields, the object size in memory, or method names. It is required to parse instance objects because it contains the byte offsets of each instance field and thus, the location in the physical memory image. If the plugin is supplied with a derived class object, it can also list instance fields of arbitrary super classes.

A shortened example output can be seen in Listing 5.22. It is the output for a `Ljava/lang/Long;` class at virtual address 0x40c378b8. The address has been taken from the output of the plugin discussed in the previous Section 5.2.6. It has one instance field named `value`, whose value can be found at offset 8 from the beginning of an instance object from the same kind of system class. Besides an `init()` method, it has methods (direct and virtual) we suspect from a class representing a long integer, such as `toString()`, `compare()` or `equals()`.

To get this information, the plugin code in Listing 5.23 first reads the class offset given on the command line (Line 4). In the following lines, the address space for a PID given on the command line is stored into the variable `proc_as`. Starting form Line 12, it is used to instantiate a `ClassObject` which is then passed to the output function in Line 13. The output function is not shown at this point, because it just accesses and prints the structure members of the `ClassObject`.

Listing 5.22: Example dalvik_class_information Output

```
$ ./vol.py [...] dalvik_class_information -o HEX -p PID \
                                          -c 0x40c378b8

objectSize directMethodCount virtualMethodCount
---------- ----------------- ------------------
        16                 0                 11

ifieldCount ifieldRefCount sfieldCount
----------- -------------- -----------
          1              0           6

------- Instance fields ------
name      signature  accessFlags byteOffset
-------- ----------- ----------- ----------
value            J           18          8

------- Direct Methods ------
name                                                shorty
---------------------------------------- --------------------
<init>                                   VJ
bitCount                                 IJ
compare                                  IJJ
toString                                 LJ
[...]

------- Virtual Methods ------
name                                                shorty
---------------------------------------- --------------------
equals                                   ZL
hashCode                                 I
intValue                                 I
[...]
```

5.2.8. Plugin: dalvik_find_class_instance

Until now, the plugins have just unveiled generic DalvikVM data and information about system classes. What is still missing is the handling of real forensic live data. Live data are objects applications read and write while running and not just static class information. So what is really needed is the location (the address) of instance objects inside the available memory dump. Together with the static information from the corresponding system class, live data can be extracted.

For that purpose, a plugin called *dalvik_find_class_instance* has been developed. Its purpose is to scan a memory region where it is likely to find a certain class

Listing 5.23: Plugin dalvik_class_information

```
class dalvik_class_information(linux_common.AbstractLinuxComman):
  def calculate(self):

    classOffset = int(self._config.CLASS_OFFSET, 16)

    proc_as = None
    tasks = linux_pslist.linux_pslist(self._config).calculate()
    for task in tasks:
      if task.pid == int(self._config.PID):
        proc_as = task.get_process_address_space()

    clazz = obj.Object('ClassObject', offset = classOffset, vm = proc_as)
    yield clazz
```

instance. Due to the fact that new class objects are typically instantiated on the heap, looking for a an instance object inside the DalvikVM heap is just a logical conclusion. The DalvikVM heap is mapped into each process address space. The plugin code seen in Listing 5.24 locates the start and end addresses of the corresponding data section from Line 8 to 12 by using the helper function *dalvik_get_data_section_dalvik_heap()*. Afterwards, it starts scanning at the beginning of the data section, trying to instantiate an `Object` (cf. Section 5.1.6). This `Object` would contain a reference to the desired `ClassObject`. In turn, this `ClassObject's clazz` pointer would point to the actual system class, the one given on the command line. So if the check in Line 18 succeeds, it is assumed that the correct address has been found. It is then handed over to the output function. This is done until the end of the Dalvik heap data section has been reached. During the evaluation of various memory images and class objects, this method of instance object retrieval was discovered as working reliably.

The first column in the output (cf. Listing 5.25) contains the system class for which a corresponding instance object should be found. The second column lists the class instances we are trying to locate. It also shows multiple rows containing different pointers for the class instance. For three reasons, it is not enough to stop looking after only one pointer has been found:

1. Not all objects in memory are still valid. The reference to the object might still be intact, so that the `clazz` pointer check succeeds, however, other

Listing 5.24: Plugin dalvik_find_class_instance

```
class dalvik_find_class_instance(linux_common.
    AbstractLinuxCommand):
  def calculate(self):
    classOffset = int(self._config.CLASS_OFFSET, 16)

    start = 0
    end = 0
    proc_as = None
    for task, vma in dalvik.get_data_section_dalvik_heap(self.
        _config):
      start = vma.vm_start
      end = vma.vm_end
      proc_as = task.get_process_address_space()
      break

    offset = start
    while offset < end:
      refObj = obj.Object('Object', offset = offset, vm = proc_as
          )

      if refObj.clazz.clazz == classOffset:
        sysClass = refObj.clazz.clazz
        yield sysClass, refObj.clazz
```

areas of the object's memory might have been overwritten already. If no Java code holds a reference to an object, the garbage collector is free to do whatever it thinks is best, including reassignment of the corresponding memory regions.

2. There might be multiple addresses (references) pointing to the same data object.
3. There might be even a huge coincidence, although possible, that the `clazz` pointer check succeeds although the corresponding memory area never contained an object of the class looking for.

So to be sure that a specific address really contains the desired instance object, the contained data needs to be always looked at and verified manually.

Listing 5.25: Example dalvik_find_class_instance Output

```
$ ./vol.py dalvik_class_information -p PID -c HEX

SystemClass              InstanceClass
------------------------ -------------
0x414e2f60                  0x414e3658
0x414e2f60                  0x4156bec8
[...]
```

5.3. Summary and Outlook

In this chapter, we analyzed the DalvikVM implementation and its underlying concepts. This lays the foundation for creating Volatility plugins to extract data artifacts belonging to specific applications. With the help of these plugins, we will be able to read class information and to gain knowledge about their corresponding instance objects. This is mandatory for the next chapter, where we will perform concrete application analysis together with the creation of the corresponding Volatility plugins.

6. Android Application Analysis

The general concepts of Android and the DalvikVM have been depicted in the previous chapters. This chapter will shift the view on a chosen set of Android applications and will outline the process of concrete forensic investigations. For this purpose, we show the general steps and entry points an forensic investigator has available to perform an application analysis. This requires us to have a brief look at the design of Android applications, especially in regard to the graphical user interface and its underlying concepts like widgets and layouts. To make use of that knowledge, we will show a way how to extract the widget information out of *APK*-files. Furthermore, we illustrate the Android application memory management to predict which application information can be extracted from a memory image. With the help of Volatility, three widespread Android applications will be investigated. This includes a twofold approach, application analysis and plugin creation.

6.1. Forensic Investigations of Android Applications

Chapter 5 showed what information and artifacts can be extracted from Dalvik Virtual Machine instances. This is general information common to all DalvikVMs. This made it possible to write universal Volatility plugins (cf. Section 5.2) which can be used for every single process or system class. However, the goal of this chapter is to extract higher level data from specific application processes such as graphical elements.

The very first requirement is the location of the desired data structures in the physical memory image. This requires deep knowledge about the data structures, their interaction, and their locations. Ideally, we have to analyze an open source application which source code is publicly available. For instance, this is done when analyzing the K9-Mail application in Section 6.2.1. In this case, reading and understanding the source code might directly lead to the class names containing the data we like to extract. It becomes more difficult when analyzing closed-source application.

In an abstract way, the forensic process could be described as follows:

1. **Get the process ID** of the target application (cf. plugin *linux_pslist* in Section 4.2.2).
2. **Find the name of the class** which contains the desired data, either by
   - **looking at the source code**, or
   - **guessing the name** by looking at all class names of the DalvikVM instance belonging to the process. This can be done with the help of the *dalvik_loaded_classes* plugin shown in Section 5.2.6. When in doubt, having a look at the class members with the *dalvik_class_information* plugin (cf. Section 5.2.7) might give valuable hints, too. Furthermore, most applications, and the ones considered in this thesis, have a graphical user interface. Everything that is visually presented to the user needs to have corresponding structures containing the data. The Android developer documentation states that element names in XML quite often correspond to class names in Java code (Google Inc., 2012). This relationship can be exploited to draw conclusions about the data organization within an application. For this purpose, a brief overview about Android layouts will be given in Section 6.1.1.
3. **Get the virtual offset of the chosen system class** by looking at the *dalvik_loaded_classes* plugin output (cf. Section 5.21).
4. **Locate a concrete class instance** of chosen system class with the *dalvik_find_class_instance* (cf. Section 5.2.8) plugin.
5. **Traverse the class instance's data structures** to acquire the desired information. This step is best performed with the help of an additional application specific Volatility plugin. This is the topic of this chapter.
6. **Print the desired memory artifacts.**

Steps 2 and 5 might require to traverse the graphical layout structure of the application. So a short overview about Android screen and layout building is given next.

6.1.1. Android Layouts

Android activities embed graphical elements such as text views, buttons or scrollable lists. In many cases, these graphical elements contain data which is of forensic interest, so understanding their structure is of importance for the application analysis.

The graphical elements are usually organized in different kinds of layouts, aligning their elements in a certain way, i.e. horizontally or vertically. Creating those layouts can be a twofold process: "Android allows you to create a screen layout in either Java code, or by declaring the layout in an XML file" (Morris, 2011). For all applications looked at in this thesis, the latter applies. The XML files are contained in the *APKs* (cf. Section 2.2) and describe the whole layout structure of the referred application. Section 6.1.2 describes the process of gathering those *APKs* from the target device and extracting their layout information.

First, we will have a look at the components Android has available for visual data alignment. There are four types of group layouts (Google Inc., 2012):

- `LinearLayout`: A layout organizing its elements either horizontally, or vertically.
- `RelativeLayout`: A layout organizing its elements in relative position to each other.
- `ListView`: Provides a scrollable list view of items.
- `GridView`: Provides a two-dimensional, scrollable grid view of items.

The layouts can be nested, creating a hierarchical structure. An example for this can be seen in Listing 6.1.

It describes a `LinearLayout`, containing a text view, a button and a `RealtiveLayout`, all aligned vertically. Each element has a unique attribute `id`, defining an identifier for layout referencing and access in Java code. The `RelativeLayout` starting in Line 6 is a nested layout containing an edit box and a button. The position of the button inside the `RelativeLayout` (Lines 8 and 9) is defined by the attribute `android:layout_below`.

From a Java perspective, all the layout element classes are derived from a common class `android.view.ViewGroup`. It defines two inherited members, `mParent`

Listing 6.1: Example Layout Definition (XML)

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/[...]"
              android:orientation="vertical">
    <TextView android:id="@+id/text" />
    <Button android:id="@+id/button1" />
    <RelativeLayout android:orientation="vertical">
        <EditText android:id="@+id/name" />
        <Button android:id="@+id/button2"
                android:layout_below="@id/name" />
    </RelativeLayout>
</LinearLayout>
```

and `mChildren`. The former is a single layout entity representing the parent, while the latter is an array possibly containing multiple child elements. For instance, in Listing 6.1, the outer `LinearLayout` has three children while the inner `RelativeLayout` has only one parent and two children. Those member fields will be frequently used when parsing layout structures to access forensic data contained within.

6.1.2. From APK to XML

To make use of the layout definitions, they need to be available. As stated above, they are contained within the *APK*-files of the corresponding applications. In Android 4.0.3, system applications are located in the directory `/system/app`, while non-system applications can be found in `/data/app`. They can be transferred to a computer system with `adb` (cf. Section 3.2.1).

However, during application compilation, the XML layout files are transformed into a binary format called *axml* (Haseman, 2008). After extraction of the *APK*-file, tools like *apktool*[1] or *axml2xml*[2] are available to transform the XML files back into a human-readable format.

[1]`http://code.google.com/p/android-apktool/`
[2]`http://code.google.com/p/android-random/downloads/detail?name=axml2xml.pl`

6.1.3. Android Application Memory Management

The goal in Chapter 6 is to extract data from a memory dump which belongs to a running, or even formerly running application. Whether the data can be extracted and is valid heavily depends on the state of the corresponding memory areas. Memory belonging to data visually presented to a user usually is unaltered and valid. However, restricting forensic investigations to this kind of data would mean that an available memory dump can only be used for specific artifacts. The typical usage of a mobile device usually differs, though. The user opens multiple applications and views, switches back and forth between them, and moves them to the background or foreground. That basically means that data written into certain application memory areas is not necessarily lost when the corresponding applications or views are hidden. For data extraction, it is enough that the corresponding memory areas remain unaltered, whether the visual presentation was available at the time the dump was captured is only of secondary relevance. For that reason, the basics of Android data visualization together with the underlying memory management has to be taken into account when evaluating the possibly available artifacts in a memory dump.

The fundamental component of a graphical user interface belonging to an Android application is an *Activities.* "An Activity is an application component that provides a screen with which users can interact in order to do something, [...]" (Google Inc., 2012). Quite often, an activity corresponds to a single view and capability. Typically multiple activities belong to a single application and can be started and stopped independently from each other. There are other application components such as *Services*, *Content Providers*, and *Broadcast Receivers.* While the basic concepts still apply, those do not having a graphical representation and are of minor relevance to this research project.

The life cycle of an Android activity or another application component can be separated into three states (Google Inc., 2012). The states are an important factor when the system, and thus the *Low Memory Killer* (cf. Section 2.2) has to decide whether or not a process, and thus the application, needs to be killed in times of memory shortage. The states are:

- *Resumed:* A state where the application is visible, active and can receive user input. In this state, the application's underlying process is never killed due to memory shortage.

- *Paused:* The corresponding application is still active, but not directly visible. This is the state an application transfers to, when another application is started overlapping the old one. Applications in this state are only killed by the *Low Memory Killer* in extremely low memory situations.
- *Stopped:* The application is in the background, not visible to the user and not active. However, "the Activity object is retained in memory, it maintains all state and member information, but is not attached to the window manager" (Google Inc., 2012). In case of memory shortage, the memory belonging to the activity will be freed.

Memory areas belonging to an application are basically only freed, and thus, assigned to another application and possibly overwritten, when they are in state *Stopped*. But even then, and quite unlikely when analyzing a system with a decent amount of main memory, there is a huge possibility that even data from already killed applications is completely valid and still available in the memory dump. We discover this to be true when extracting contact information in Section 6.2.3. Memory areas of applications in state *Resumed* are always unaltered, while it is very likely for those in state *Paused*.

Summarizing, memory dumps certainly contain the data belonging to activities which were visible at the time of acquisition. But it is still very likely to contain valid data for other activities belonging to the same application and still possible for those running in background or even already killed, to a certain extend, at least.

6.2. Applications

In the further course of this chapter, we perform exemplary forensic investigations, together with depicting the corresponding Volatility plugins, for data contained in three Android applications. Those are *K-9 Mail*[3], a powerful mail reader, *WhatsApp*[4], a text messaging application, and the standard contacts application shipped with Android.

[3] http://code.google.com/p/k9mail/
[4] http://www.whatsapp.com/

All plugins that are created use the common file name prefix *dalvik_app_**. Furthermore, if there are more than one plugin for an application, an additional specifier defines its concrete task. For instance, application plugins might be called *dalvik_app_k9mail_accounts* or *dalvik_app_k9mail_mail*. The first is the application plugin for the *K-9 Mail* application reading its account data, while the purpose of the latter is to read just a single mail. To illustrate the plugin writing, we only show relevant source code snippets. The full source code can be found on the accompanying CD.

6.2.1. K-9 Mail

The first application we look at is K-9 Mail. K-9 Mail is a popular open source email client able to handle multiple IMAP mail accounts, has a folder view, a mail view, and of course, email composing capabilities (K-9 Mail Project, 2012). Each of those features have a corresponding activity view displaying the respective data. The application version the analysis is based on is 4.200. We will extract mail accounts together with their credentials, a list of received mails and show how to read their content.

Parsing Mail Accounts

(*dalvik_app_k9mail_accounts*)

The activity shown in Figure 6.1 shows a typical view for the K-9 Mail application configured with multiple mail accounts. At first glance, two elements are of forensic interest: The account's name and corresponding email address. However, we will see that there is more data attached to an account, such as default folder names, the storage method, the mail access method (IMAP, POP, etc.), and even the user name and password. Extracting this information is the goal.

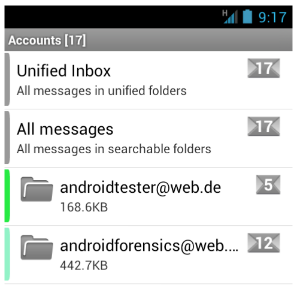


Figure 6.1.: K-9 Mail Email Accounts

The first step (cf. Section 6.1) is to find the process ID of K-9 Mail with the *linux_pslist* plugin. After locating the DalvikVM's global structure offset with *dalvik_find_gdvm_offset*, it can be supplied to the *dalvik_loaded_classes* plugin. Amongst all loaded system classes, there is one with a descriptor `Lcom/fsck/k9/activity/Accounts;`, implemented in a file called `Accounts.java`. According to this information, it is likely that it contains references to the individual accounts. Looking at its members with the *dalvik_class_information plugin* strengthens this suspicion: It has a member field called `accounts` which is an array of `Lcom/fsck/k9/BaseAccount;` objects. In turn, each `Lcom/fsck/k9/BaseAccount;` object contains members like `mInboxFolderName` or `mDescription`, specifying the default folder name for the inbox and and an account description. It also has a member named `mStoreUri` which contains the credentials and access method for the corresponding account. This includes the user name and password which are stored as plain text.

The whole hierarchy for accessing the desired information can be seen in Listing 6.2. The level of indentation describes the relationship between class objects and their member fields. The elements in brackets describe the type of member field in the same line. Bold elements are the elements we finally extract.

Listing 6.2: Access Hierarchy for K-9 Mail Account Information

```
Lcom/fsck/k9/activity/Accounts;
-> [Lcom/fsck/k9/BaseAccount;
   -> Lcom/fsck/k9/BaseAccount;
      -> mInboxFolderName (Ljava/lang/String;)
      -> mDescription (Ljava/lang/String;)
      -> [...]
   -> Lcom/fsck/k9/BaseAccount;
      -> mInboxFolderName (Ljava/lang/String;)
      -> [...]
```

The next step is to find an instance object with the *dalvik_find_class_instance* plugin. This object's offset can then be passed to the newly created *dalvik_app_k9mail_accounts* plugin. Parts of its source code can be seen in Listing 6.3. In Line 4, a `ClassObject` is created from the given instance object offset. The helper function `getIFieldAsArray()` is used to iterate over all `BaseAccount` objects starting from Line 6. The following lines just access the member fields, passing their names together with their contents to the output function.

An exemplary output can be seen in Listing 6.4. It also shows the `mStoreUri`

Listing 6.3: dalvik_app_k9mail_accounts Plugin

```
class dalvik_app_k9mail_accounts(linux_common.[...]):
 def calculate(self):

   c = obj.Object('ClassObject', offset = classOffset, vm = a)

   for ref in c.getIFieldAsArray("accounts"):
     print "-------- Account: -------"

     clazz = obj.Object('ClassObject', offset = ref, vm = a)

     yield "mInboxFolderName", clazz.getIFieldAsString(
         "mInboxFolderName")
     yield "mStoreUri", clazz.getIFieldAsString("mStoreUri")
     yield "mAccountNumber", clazz.getIFieldAsBool(
         "mAccountNumber")
```

Listing 6.4: dalvik_app_k9mail_accounts Plugin Output

```
$ ./vol.py [...] dalvik_app_k9mail_accounts -p PID -c HEX

member field      value
----------------- ---------------------------------------------
-------- Account:
mInboxFolderName  INBOX
mStoreUri   imap+ssl://androidforensics:androidtester@imap.web.de
mDescription      androidforensics@web.de
mAccountNumber                                                0

-------- Account:
mInboxFolderName  INBOX
mStoreUri      imap+ssl://androidtester:androidtester@imap.web.de
mDescription      androidtester@web.de
mAccountNumber                                                1
```

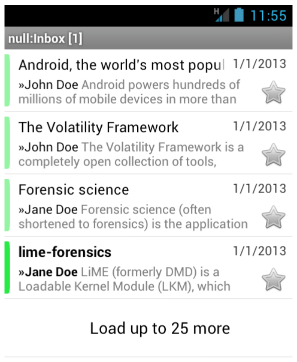
member field which contains the account type (IMAP), the user name (androidforensics) and the password (androidtester). Even the IMAP server address can be extracted (imap.web.de).

Listing Emails

dalvik_app_k9mail_listmails

Another activity frequently used in K-9 Mail is a view to list all mails inside a folder. Together with the sender, subject, and time, a short preview of the message text is shown (cf. Figure 6.2).

The steps for a forensic application investigation outlined along with the previous plugin and depicted in Section 6.1 also apply for this plugin. However, from now on, they are considered self-evident and will not be described again.


11:55
null:Inbox [1]

Android, the world's most popul 1/1/2013
»John Doe Android powers hundreds of millions of mobile devices in more than

The Volatility Framework 1/1/2013
»John Doe The Volatility Framework is a completely open collection of tools,

Forensic science 1/1/2013
»Jane Doe Forensic science (often shortened to forensics) is the application

lime-forensics 1/1/2013
»Jane Doe LiME (formerly DMD) is a Loadable Kernel Module (LKM), which

Load up to 25 more

Figure 6.2.: K9-Mail Email List

Looking at the *dalvik_loaded_classes* plugin unveils that a class called `MessageList` corresponds to the shown activity and marks the base element for further investigation. Analyzing the source code and layout XML file leads to an access hierarchy shown in Listing 6.5.

Listing 6.5: Access Hierarchy for K-9 Mail List View

```
Lcom/fsck/k9/activity/MessageList;
-> mMessageListFragment (Lcom/fsck/k9/fragment/
    MessageListFragment;)
   -> mListView (Landroid/widget/ListView;)
      -> mChildren[x] (Landroid/widget/RelativeLayout;)
         -> mChildren[1] (Landroid.widget.TextView;)
            -> mText (Ljava/lang/String; mail subject)
         -> mChildren[4] (Landroid.widget.TextView;)
            -> mText (Ljava/lang/String; mail time)
         -> mChildren[2] (Landroid.text.SpannableString;)
            -> mText (Landroid.text.SpannableString;)
               ->  mText (Ljava/lang/String; mail preview)
```

The `MessageList` has a member field of type `MessageListFragment` containing a `ListView` with multiple graphical layouts. Each of them is embedding another information like the mail subject or preview.

Data access inside the plugin code (cf. Listing 6.6) is straightforward. It shows how to traverse the classes and iterates over all shown mail fragments, storing

Listing 6.6: dalvik_app_k9mail_listmails Plugin

```
class dalvik_app_k9mail_listmails(linux_common.[...]):
  def calculate(self):
    c = obj.Object('ClassObject', offset = classOffset, vm = as)

    c = c.getIFieldAsClassObject("mMessageListFragment")
    c = c.getIFieldAsClassObject("mListView")
    c = c.getIFieldAsClassObject("mChildren")

   for i in dalvik.parseArrayObject(c.obj_offset, as):
     layout = obj.Object('ClassObject', offset = i, vm = as)

     arrObj = layout.getIFieldAsArray("mChildren")

     tv = obj.Object('ClassObject', offset = arrObj[1], vm = as)
     subject = dalvik.getStringFromTextView(tv)
```

the mail subject in an equally named variable. Output of that plugin might look like seen in Listing 6.7.

Listing 6.7: dalvik_app_k9mail_listmails Plugin Output

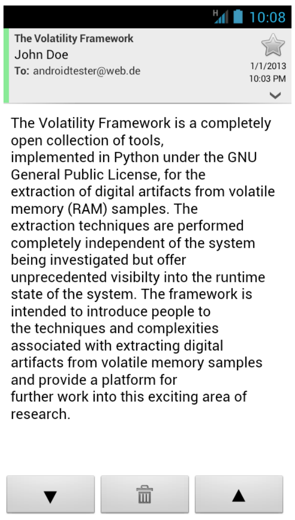
```
$ ./vol.py [...] dalvik_app_k9mail_listmails -p PID -c HEX

time      subject                        preview
------    ----------------------------   -------
1/1/2013 Android, the world's most...   Android powers hundred...
1/1/2013 The Volatility Framework       The Volatility Framework...
1/1/2013 Forensic Science               Forensic science (often...
1/1/2013 lime-forensics                 LiME (formerly DMD) is a...
```

The list view of mails could be helpful when deciding whether a specific mail could be of forensic interest. If so, the following plugin can be used to reads its whole content.

Reading Mail

(dalvik_app_k9mail_mail)

Reading single mails might also be valuable to a forensic investigation. Of course, K-9 Mail provides a way for doing so. Clicking on a mail fragment in the folder view opens the mail activity. Besides sender and recipient(s), the mail body is shown (cf. Figure 6.3). The corresponding Android activity is called `MessageView`. It has members named `mMessage`, `mSubject`, `mFrom` and `mTo`, containing the corresponding data artifacts of interest. Listing 6.8 shows the thorough access hierarchy.



The Volatility Framework
John Doe
To: androidtester@web.de
1/1/2013
10:03 PM

The Volatility Framework is a completely open collection of tools, implemented in Python under the GNU General Public License, for the extraction of digital artifacts from volatile memory (RAM) samples. The extraction techniques are performed completely independent of the system being investigated but offer unprecedented visibilty into the runtime state of the system. The framework is intended to introduce people to the techniques and complexities associated with extracting digital artifacts from volatile memory samples and provide a platform for further work into this exciting area of research.

Figure 6.3.: K-9 Mail Mail Activity

Listing 6.8: Access Hierarchy for K-9 Mail Mail View

```
Lcom/fsck/k9/activity/MessageView;
-> mMessage (Lcom/fsck/k9/mail/store/LocalStore$LocalMessage;)
   -> mBody (Lcom/fsck/k9/mail/store/LocalStore$LocalTextBody)
      -> mBody (Ljava/lang/String; mail body)
-> mSubject (Ljava/lang/String; mail subject)
-> mFrom ([Lcom/fsck/k9/mail/Address;)
   -> Lcom/fsck/k9/mail/Address;
      -> mAddress (Ljava/lang/String; sender address)
      -> mPersonal (Ljava/lang/String; sender name)
   -> [...]
-> mTo ([Lcom/fsck/k9/mail/Address;)
   -> Lcom/fsck/k9/mail/Address;
      -> mAddress (Ljava/lang/String; recipient address)
   -> [...]
```

The Python source code has been written accordingly and a shortened example is shown in Listing 6.9. In Lines 7 and 8, the mail body text is extracted while in Lines 10 to 15, the sender names are read.

A typical output might look as seen in Listing 6.10.

Listing 6.9: dalvik_app_k9mail_mail Plugin

```
class dalvik_app_k9mail_mail(linux_common.AbstractLinuxCommand):
  def calculate(self):
    c = obj.Object('ClassObject', offset = classOffset, vm = as)
    c = c.getIFieldAsClassObject("mMessageViewFragment")
    c = c.getIFieldAsClassObject("mMessage")

    mBody = c.getIFieldAsClassObject("mBody")
    body = mBody.getIFieldAsString("mBody")

    from_addr = [ ]
    from_name = [ ]
    for i in c.getIFieldAsArray("mFrom"):
      clazz = obj.Object('ClassObject', offset = i, vm = as)
      from_addr.append(clazz.getIFieldAsString("mAddress"))
      from_name.append(clazz.getIFieldAsString("mPersonal"))
```

Listing 6.10: dalvik_app_k9mail_mail Plugin Output

```
$ ./vol.py [...] dalvik_app_k9mail_mail -p PID -c HEX

subject              from_name    from_addr
-------------        ------------ ---------------
The Volat...ramework John Doe     androidfo...s@web.de

 to_addr                 body
 ----------------------- --------------
 androidtester@web.de    The Volatility Framework is...research.
```

6.2.2. WhatsApp

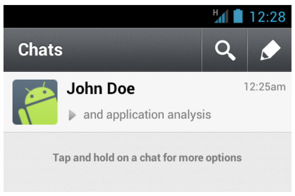
WhatsApp is a real time messaging application available for various operating systems like Apple iOS, Android and Symbian (WhatsApp Inc., 2012b). In addition to plain text messages, the application is also able to share media files like images or videos. It is based on the standard contact list to find communication counterparts. In August 2013, the company announced that they where able to count over 10 million messages a day (WhatsApp Inc., 2012a), making it one of the most popular in the mobile IT industry. The *WhatsApp* version used in this thesis is 2.8.9108.

In the following, two Android activities for *WhatsApp* are considered.

Parsing Conversations

(*dalvik_app_whatsapp_conversations*)

The first forensic analysis is done for the conversations activity. It lists the most recent or active communications. An example view is provided in Figure 6.4.



John Doe 12:25am
and application analysis
Tap and hold on a chat for more options

Figure 6.4.: WhatsApp Conversations

Each conversation has the sender's name, an icon, and shows the last text which was either sent or received. Listing 6.12 shows the corresponding access hierarchy. Starting from a `Lcom/whatsapp/Conversations;` class object, several levels of layouts need to be traversed in order to extract the desired data. A `ListView` object contains several rows organized in a `LinearLayout`, each representing a single conversation. The object described as `Lcom/whatsapp/TextEmojiLabel;` is an application specific class able to display conversation text together with special symbols like smileys. The text part can be extracted by accessing the member field `mText`, which is a Java string.

Listing 6.11: dalvik_app_whatsapp_conversations Plugin Output

```
$ ./vol.py [...] dalvik_app_whatsapp_conversations -p PID -c HEX

Name                                  Time
------------------------------------- --------
John Doe                              12:25am
```

Listing 6.13 shows the corresponding plugin code. It has to reverse engineer the layout elements in order to reach the underlying artifacts.

The output of the plugin might look as seen in Listing 6.11.

Listing 6.12: Access Hierarchy for WhatsApp Conversations

```
Lcom/whatsapp/Conversations;
-> mList (Landroid/widget/ListView;)
   -> mChildren[0] (Landroid/widget/RelativeLayout;)
      -> mChildren[1] (Landroid/widget/LinearLayout;)
         -> mChildren[0] (Landroid/widget/LinearLayout;)
            -> mChildren[0] (Lcom/whatsapp/TextEmojiLabel;)
               -> mText (Ljava/lang/String; name of conversation
                  counterpart)
         -> mChildren[0] (Landroid/widget/LinearLayout;)
            -> mChildren[1] (Landroid/widget/TextView;)
              -> mText (Ljava/lang/String; time string)
         -> mChildren[1] (Landroid/widget/LinearLayout;)
            -> mChildren[1] (Lcom/whatsapp/TextEmojiLabel;)
               -> mChildren[2] (Lcom/whatsapp/TextEmojiLabel;)
                  -> mText (Ljava/lang/String; preview string)
   -> mChildren[x] (Landroid/widget/RelativeLayout;)
   -> [...]
```

Listing 6.13: dalvik_app_whatsapp_conversations Plugin

```python
class dalvik_app_WhatsApp_conversations(linux_common.[...]):
  def calculate(self):
    c = obj.Object('ClassObject', offset = classOffset, vm = as)
    mList = c.getIFieldAsClassObject("mList")

    arrObj = mList.getIFieldAsArray("mChildren")
    l = obj.Object('ClassObject', offset = arrObj[0], vm = as)
    arrObj = l.getIFieldAsArray("mChildren")
    l = obj.Object('ClassObject', offset = arrObj[1], vm = as)
    arrObj = l.getIFieldAsArray("mChildren")
    l = obj.Object('ClassObject', offset = arrObj[0], vm = as)

    arrObj = l.getIFieldAsArray("mChildren")
    tv = obj.Object('ClassObject', offset = arrObj[0], vm = as)
    name = tv.getIFieldAsString("mText")

    tv = obj.Object('ClassObject', offset = arrObj[1], vm = as)
    time = tv.getIFieldAsString("mText")

    yield name, time
```

Reading a Conversation

(*dalvik_app_whatsapp_conversation*)

While the previous plugin extracted data from multiple conversations, the one handled in this section concentrates on the single conversations view like one is shown in Figure 6.5.

Listing 6.14: Access Hierarchy for WhatsApp Conversation

```
Lcom/whatsapp/Conversation;
-> Kb (Landroid/widget/TextView;)
  -> mText (Ljava/lang/String; Communication counterpart)
-> mList (Landroid/widget/ListView;)
   -> mChildren[x] (Lcom/whatsapp/qf; conversation rows)
      -> mChildren[1] (Landroid/widget/LinearLayout;)
         # Either mChildren[1] _or_ mChildren[2] is valid
         -> mChildren[1] (Landroid/widget/LinearLayout;)
            -> mChildren[0] (Landroid/widget/RelativeLayout;)
               -> mChildren[1] (Lcom/whatsapp/TextEmojiLabel;)
                  -> mText (Ljava/lang/String; Received message)
         -> mChildren[2] (Landroid/widget/RelativeLayout;)
            -> mChildren[1] (Lcom/whatsapp/TextEmojiLabel;)
               -> mText (Ljava/lang/String; Sent message)
```

How to access the data elements can be seen in Listing 6.14. Starting from a `Lcom/whatsapp/Conversation;` class, we extract the communication counterpart in the member field `Kb`. A `ListView` is the starting point of traversing the layout structure. Finally, the received and sent messages ca be read.

The corresponding plugin code to extract the data from within Volatility is shown in Listing 6.15. At Lines 5 and 6, the code extracts the conversation counterpart's name which is stored in the variable `buddy`. In the following lines, the layout elements are parsed to extract the sent messages.

The result might look like in Listing 6.16.

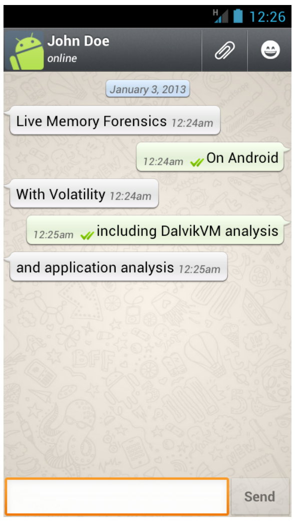

12:26
John Doe
online
January 3, 2013
Live Memory Forensics 12:24am
12:24am On Android
With Volatility 12:24am
12:25am including DalvikVM analysis
and application analysis 12:25am

Figure 6.5.: WhatsApp Conversation

Listing 6.15: dalvik_app_whatsapp_conversation Plugin

```
class dalvik_app_WhatsApp_conversation(linux_common.[...]):
  def calculate(self):
    c = obj.Object('ClassObject', offset = classOffset, vm = as)

    tv = c.getIFieldAsClassObject("Kb")
    buddy = tv.getIFieldAsString("mText")

    mList = c.getIFieldAsClassObject("mList")
    for row_ref in mList.getIFieldAsArray("mChildren"):
      l = obj.Object('ClassObject', offset = arrObj[2], vm = as)

      arrObj = l.getIFieldAsArray("mChildren")
      tv = obj.Object('ClassObject', offset = arrObj[1], vm = as)

      sent = tv.getIFieldAsString('mText')
```

Listing 6.16: dalvik_app_whatsapp_conversation Plugin Output

```
$ ./vol.py [...] dalvik_app_whatsapp_conversation -p PID -c HEX

Conversation with John Doe:
sent                           received
------------------------------ ---------------------------------
                               Live Memory Forensics
On Android
                               With Volatility
including DalvikVM analysis
                               and application analysis
```

6.2.3. Contacts

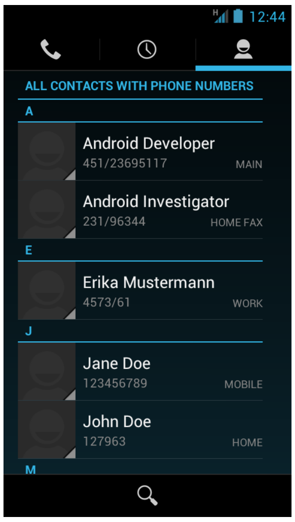
(*dalvik_app_contacts*)

Most mobile device, and cell phones in particular, have a way for managing contacts. In case of smartphones, it is used to store the contact's phone numbers, addresses, and additional meta information. This information could be of broad interest for a forensic investigation.

On every smart phone, the dialer application is one of the most used functionalities. The dialer view in Android *Ice Cream Sandwich* consists of three basic parts: The dialer itself, a view showing the latest calls, and a contacts view.

These parts correspond to the top icons which can be seen in Listing 6.6.

Whenever initiating or answering a call, the dialer application is involved. To speed up GUI responsiveness, opening one of the functionalities causes the application to preload the other views with their appropriate data items. This makes it quite likely that the contacts list from the dialer application is available in the phone's memory. Both the dialer and the standalone contacts application share the same data, although it is less likely that the latter has been accessed previously to capturing the memory image. So in the following plugin, we concentrate on trying to extract the contacts from the dialer application view.



12:44

ALL CONTACTS WITH PHONE NUMBERS

A

Android Developer
451/23695117 MAIN

Android Investigator
231/96344 HOME FAX

E

Erika Mustermann
4573/61 WORK

J

Jane Doe
123456789 MOBILE

John Doe
127963 HOME

M

Figure 6.6.: Android Dialer

The approach taken in the previous sections has always been to find a class object representing a list of items, locating an instance object of the same, and passing that to the corresponding plugin on the command line. The plugin was responsible for traversing the list and extracting single items out of it.

Now we make use of a different approach: During investigation of the preloaded classes, it was not possible to identify a class corresponding to one list of contacts. However, we were able to locate the class named `Lcom/android/contacts/list/ContactListItemView;`, representing a single contact. The use of *dalvik_find_class_instance* unveiled multiple instance objects. Each each of them can now be passed to the newly created plugin, retrieving the name, number and type of the number of the contact. With this method, all of the available contacts could be restored. On a related note, we were even able to recover contacts which were previously deleted but still available in the application's memory space.

Listing 6.17 shows how to access the data artifacts from a `ContactListItemView` object, while Listing 6.18 shows how the data access works inside the plugin.

In Line 5 of the source code, the text view is parsed. In Line 6, we extract the string from the `SpannedString;`, and in Line 7 we save the contact's name in the equally named variable. The data access for the phone number and the type of

Listing 6.17: Access Hierarchy for Contacts

```
Lcom/android/contacts/list/ContactListItemView;
-> mNameTextView (Landroid/widget/TextView;)
   -> mText (Landroid/text/SpannedString;)
      -> mText (Ljava/lang/String; Contact Name)
-> mDataView (Landroid/widget/TextView;)
   -> mText (Landroid/text/SpannedString;)
      -> mText (Ljava/lang/String; Number)
-> mLabelView (Landroid/widget/TextView;)
   -> mText (Landroid/text/SpannedString;)
      -> mText (Ljava/lang/String; Number Type)
```

Listing 6.18: dalvik_app_contacts Plugin

```
class dalvik_app_contacts(linux_common.AbstractLinuxCommand):
  def calculate(self):
    c = obj.Object('ClassObject', offset = classOffset, vm = as)

    tv = c.getIFieldAsClassObject("mNameTextView")
    ss = tv.getIFieldAsClassObject("mText")
    name = ss.getIFieldAsString("mText")
```

the phone number is analogous.

Together with the usage of *dalvik_find_class_instance*, the output can be seen in Listing 6.19.

Listing 6.19: dalvik_app_contacts Plugin Output

```
$ ./vol.py [...] dalvik_find_class_instance -p PID -c HEX

SystemClass                    InstanceClass
------------------------------ ------------------------------
0x4177a820                     0x417be9e0
0x4177a820                     0x418fc748
0x4177a820                     0x41862278
[...]

$ ./vol.py [...] dalvik_app_contacts -p PID -c 0x417be9e0
Name                 Number               Number Type
-------------------- -------------------- -----------
Jane Doe             123456789            Mobile

$ ./vol.py [...] dalvik_app_contacts -p PID -c 0x418fc748
Name                 Number               Number Type
-------------------- -------------------- -----------
Max Mustermann       5896/342             Home Fax
```

6.3. Summary and Outlook

In this chapter, we have finally shown how to perform real-world application analysis. We depicted the general process for a forensic investigation and practically applied the knowledge gathered from the previous chapters. The illustration of access hierarchies together with the actual source code can provide a valuable guideline for additional application plugins.

In the last chapter, we will submit the created source code to the Volatility project, look at possible subsequent research possibilities, and will conclude the thesis.

7. Future Work and Conclusion

The last chapter will describe when and where the created source code was submitted to the Volatility project. After looking at possible future work, we will summarize our insights, check if we have reached the thesis goals and will draw a conclusion.

7.1. Source Code Submission

In the course of this thesis, we have made heavy use of the Volatility project. Without the provided framework, the performed tasks for Android memory analysis would have been much more challenging. This was made possible because the project uses an open development model.

To conform to the open source license used in Volatility, all the plugins are released under *GNU General Public License*. Furthermore, a first iteration of infrastructure support and plugins was sent to the Volatility development mailing list on 16th October 2012[1]. It received positive feedback.

Alongside with the plugins, a documentation file called `README.dalvik` was included. It can be seen in appendix A.2. It can also assist to get a fast overview about the plugins and how they interact.

A cleaned up set of plugins, including bug fixes, further enhancement and support for specific applications was submitted to the same list on 6th January 2013[2]. The same version of plugins can be found on the accompanying CD.

[1] http://lists.volatilesystems.com/pipermail/vol-dev/2012-October/000187.html
[2] http://lists.volatilesystems.com/pipermail/vol-dev/2013-January/000198.html

7.2. Future Work

This research project advanced the field of Android live memory forensics by outlining a thorough overview and by providing a software infrastructure to perform real-world forensic investigations. However, starting from the outcome of this thesis, of course there is still some work that could be followed up on.

To illustrate the dependencies and interaction between the different plugins, we made heavy use of command line options. However, all the plugins have the proper means of calling each other to interchange data. So an investigation making use of multiple plugins, calling each other one after another, could be simplified by creating one final plugin aggregating the different tasks.

Also, the plugins parsing the DalvikVM are dependent on one specific Android and DalvikVM version. There is no guarantee that they will work with future implementations. The Dalvik plugins and related files could be extended to deal with different versions. Alternatively, different plugins targeting different versions could be provided.

The same applies to the application plugins depicted in Chapter 6. They are just meant to illustrate the development process. On top of the existing infrastructure, more plugins can be created, targeting various applications and specific application versions. A huge set of application plugins could make their way into the toolkit of any forensic investigator.

However, this thesis does not solve the problem of creating a kernel-agnostic module that can be used on arbitrary devices. We still need modules fitting the target kernels. With the vast variety of devices in the market, all running different kernels and Android versions, this continues to be a challenge for further research.

7.3. Conclusion

This thesis provides a thorough overview about the memory acquisition and memory analysis possibilities of the Android platform. It presents a software stack to enable forensic investigators to perform application analysis with as little efforts as possible. Another goal is the submission of the created plugins into to the Volatility project.

As a solution for memory acquisition already exists, it was possible to shift the main focus to memory analysis. During the project planning, it became obvious that a stacked approach would be best: The analysis of the Android kernel, of the middleware layer (DalvikVM), and the graphical applications. While the kernel analysis was intended to query memory mappings and to identify processes belonging to specific DalvikVM instances, the information from the latter should assist in the creation of the application plugins. The approach turned out to be feasible. As soon as the infrastructure support in Volatility and the first application investigations were finished, application analysis turned out to be a straightforward process. Hence, this made it possible to develop a guideline for the analysis of further applications.

The Volatility framework has been of indispensable assistance. Without it, a lot of analysis work would have been much more challenging, if not impossible. Actually, the open concept and source code availability of most involved projects laid the foundation for drawing conclusions about the layout of data objects in physical memory. This is the reason why the created software stack was submitted to the Volatility project, to share the work and the gained knowledge with other researchers or forensic investigators.

Together with the work performed in this project, Volatility is now able to provide a thorough set of plugins for memory analysis of the Android platform. It can assist real-world forensic investigations and the set of plugins suits well into the toolkit of every digital forensic investigator.

Bibliography

Adelstein, F. (2006). Live forensics: diagnosing your system without killing it first. *Commun. ACM*, 49(2), 63–66.

Bornstein, D. (2006). Dalvik VM Internals. In *Google I/O conference 2008, presentation video and slides.*

Brähler, S. (2010). Analysis of the Android Architecture. Student Research Paper, System Architecture Group, University of Karlsruhe, Germany.

Carrier, B. D. (2003). Defining Digital Forensic Examination and Analysis Tools Using Abstraction Layers. *IJDE*, 1(4).

Case, A. (2011). Memory Analysis of the Dalvik (Android) Virtual Machine. In *SOURCE Seattle 2011, presentation video and slides.*

Eager, M. J. & Consulting, E. (2007). Introduction to the DWARF Debugging Format. *Group.*

Google Inc. (2012). Android Developer Documentation. `http://developer.android.com`.

Gosling, J., Joy, B., Steele, G., & Bracha, G. (2005). *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)).* Addison-Wesley Professional.

Haseman, C. (2008). *Android Essentials.* Apresspod Series. Apress.

Hay, B., Nance, K., & Bishop, M. (2009). Live Analysis: Progress and Challenges. *Security & Privacy, IEEE*, 7(2), 30–37.

Houck, M. & Siegel, J. (2009). *Fundamentals of Forensic Science.* Elsevier Science.

International Data Corporation (2012). Worldwide Smartphone OS Market Share, 2012Q3. `https://www.idc.com/getdoc.jsp?containerId=prUS23771812`.

ITProPortal (2012). Smartphone Ownership On The Rise, Claims US Study. `http://www.itproportal.com/2012/03/02/25ca8c92-645e-11e1-a090-fefdb24f8a11/`.

Joe Sylve (2012). LiME - Linux Memory Extractor, Instructions v1.1. `http://lime-forensics.googlecode.com/files/LiME_Documentation_1.1.pdf`.

K-9 Mail Project (2012). K-9 Mail Project Homepage. `http://code.google.com/p/k9mail/`.

Kawachiya, K., Ogata, K., & Onodera, T. (2008). Analysis and Reduction of Memory Inefficiencies in Java Strings. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08 (pp. 385–402). New York, NY, USA: ACM.

Krahmer, S. (2010). "Rage Against the Cage". `http://c-skills.blogspot.de/2010/08/droid2.html`.

Leppert, S. (2012). Android Memory Dump Analysis. Student Research Paper, Chair of Computer Science 1 (IT-Security), Friedrich-Alexander-University Erlangen-Nuremberg, Germany.

Lindholm, T., Yellin, F., Bracha, G., & Buckley, A. (2012). *The Java^{TM} Virtual Machine Specification*. Oracle America, Inc., Java SE 7 Edition edition.

Morris, J. (2011). *Android User Interface Development: Beginner's Guide*. Packt Publishing, Limited.

Oracle Corporation (2012). The Java Tutorials. `http://docs.oracle.com/javase/tutorial/`.

Punja, S. G. & Mislan, R. P. (2008). Mobile Device Analysis. *Small Scale Digital Device Forensics Journal*, 2(1).

Riehle, D. (2006). Value object. In *Proceedings of the 2006 conference on Pattern languages of programs*, PLoP '06 (pp. 30:1–30:6). New York, NY, USA: ACM.

Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., & Dolev, S. (2009). Google Android: A State-of-the-Art Review of Security Mechanisms. *CoRR*, abs/0912.5101.

Sylve, J., Case, A., Marziale, L., & Richard, G. G. (2012). Acquisition and analysis of volatile memory from android devices. *Digital Investigation*, 8(3-4), 175 – 184.

The Unicode Consortium (2011). *The Unicode Standard.* Technical Report Version 6.0.0, Unicode Consortium, Mountain View, CA.

Thing, V. L., Ng, K.-Y., & Chang, E.-C. (2010). Live memory forensics of mobile phones. *Digital Investigation*, 7, Supplement(0), S74 – S82. The Proceedings of the Tenth Annual DFRWS Conference¡/ce:title¿.

Urrea, J. M. (2006). An Analysis of Linux RAM Forensics. Master's thesis, Naval Postgraduate School, Monterey California.

Volatilesystems (2012). Volatility Project Homepage. https://code.google.com/p/volatility/.

Wehrle, K., Pählke, F., Ritter, H., Müller, D., & Bechler, M. (2005). *The Linux Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel.* Prentice Hall.

WhatsApp Inc. (2012a). Twitter Message. https://twitter.com/WhatsApp/status/238680463139565568.

WhatsApp Inc. (2012b). WhatsApp Project Homepage. http://www.whatsapp.com.

Wikipedia (2012a). Android (operating system) — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Android_(operating_system)&oldid=530060896.

Wikipedia (2012b). Android rooting — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Android_rooting&oldid=528094233.

Wikipedia (2012c). .bss — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=.bss&oldid=525501314.

Wikipedia (2012d). Cross compiler — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Cross_compiler&oldid=518389473.

Wikipedia (2012e). Data structure alignment — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Data_structure_alignment&oldid=525379887.

Wikipedia (2012f). Forensic science — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Forensic_science&oldid=520203985.

Yan, L. K. & Yin, H. (2012). DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12 (pp. 29–29). Berkeley, CA, USA: USENIX Association.

Yen, P.-H., Yang, C.-H., & Ahn, T.-N. (2009). Design and Implementation of a Live-analysis Digital Forensic System. In *Proceedings of the 2009 International Conference on Hybrid Information Technology*, ICHIT '09 (pp. 239–243). New York, NY, USA: ACM.

A. Appendix

A.1. Volatility vtype Definitions for the DalvikVM

Listing A.1: dalvik_vtypes.py

```
# Volatility
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or (at
# your option) any later version.
#
# This program is distributed in the hope that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
# General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

"""
@author:       Holger Macht
@license:      GNU General Public License 2.0 or later
@contact:      holger@homac.de
"""

import volatility.obj as obj
import volatility.plugins.linux.dalvik as dalvik

dalvik_vtypes = {
    #dalvik/Common.h
    'JValue' : [ 0x4, {
        'bool' : [ 0x0, ['bool']],
        'int' : [ 0x0, ['int']],
        'Object' : [ 0x0, ['address']],
        }],
    #dalvik/oo/Object.h
    'Field' : [ 0x10, {
        'clazz' : [ 0x0, ['pointer', ['ClassObject']]],
        'name' : [ 0x4, ['pointer', ['char']]],
        'signature' : [ 0x8, ['pointer', ['char']]],
        'accessFlags' : [ 0xc, ['unsigned int']],
        }],
    'InstField' : [ 0x14, {
        'clazz' : [ 0x0, ['pointer', ['ClassObject']]],
        'name' : [ 0x4, ['pointer', ['char']]],
        'signature' : [ 0x8, ['pointer', ['char']]],
        'accessFlags' : [ 0xc, ['unsigned int']],
        'byteOffset' : [ 0x10, ['int']],
        }],
    'StaticField' : [ 0x18, {
        'clazz' : [ 0x0, ['pointer', ['ClassObject']]],
        'name' : [ 0x4, ['pointer', ['char']]],
        'signature' : [ 0x8, ['pointer', ['char']]],
        'accessFlags' : [ 0xc, ['unsigned int']],
        # can take up to 8 bytes
        'value' : [ 0x10, ['JValue']],
        }],
    'Object' : [ 0x8, {
        'clazz' : [ 0x0, ['pointer', ['ClassObject']]],
        'lock' : [ 0x4, ['int']],
        }],
    'DataObject' : [ 0xc, {
        'clazz' : [ 0x0, ['pointer', ['ClassObject']]],
        'lock' : [ 0x4, ['int']],
        'instanceData' : [ 0x8, ['address']],
        }],
    'StringObject' : [ 0xc, {
        'clazz' : [ 0x0, ['pointer', ['ClassObject']]],
        'lock' : [ 0x4, ['int']],
        'instanceData' : [ 0x8, ['unsigned int']],
        }],
    'ArrayObject' : [ 0x14, {
        'clazz' : [ 0x0, ['pointer', ['ClassObject']]],
        'lock' : [ 0x4, ['int']],
        'length' : [ 0x8, ['unsigned int']],
        #including padding of 4 bytes
        'contents0' : [ 0xc, ['address']],
        'contents1' : [ 0x10, ['address']],
        }],
    'Method' : [ 0x38, {
        'clazz' : [ 0x0, ['pointer', ['ClassObject']]],
        'name' : [ 0x10, ['pointer', ['char']]],
        'shorty' : [ 0x1c, ['pointer', ['char']]],
        'inProfile' : [ 0x34, ['bool']],
        }],
    #libdex/DexFile.h
    'DexOptHeader' : [ 0x4, {
        'magic' : [ 0x0, ['unsigned long long']],
        'dexOffset' : [ 0x8, ['unsigned int']],
        }],
    'DexHeader' : [ 0x4, {
        'fileSize' : [ 0x10, ['unsigned int']],
        }],
    'DexFile' : [ 0x4, {
        'pOptHeader' : [ 0x0, ['pointer', ['DexOptHeader']]],
        'pHeader' : [ 0x4, ['pointer', ['DexHeader']]],
    }],
#dalvik/DvmDex.h
'DvmDex' : [ 0x4, {
    'pDexFile' : [ 0x0, ['pointer', ['DexFile']]],
    }],
#dalvik/oo/Object.h
'ClassObject' : [ 0xa4, {
    'clazz' : [ 0x0, ['pointer', ['ClassObject']]],
    'lock' : [ 0x4, ['int']],
    'instanceData0': [ 0x8, ['address']],
    'instanceData1': [ 0xc, ['address']],
    'instanceData2': [ 0x10, ['address']],
    'instanceData3': [ 0x14, ['address']],
    'descriptor' : [ 0x18, ['pointer', ['char']]],
    'descriptorAlloc' : [ 0x1c, ['pointer', ['char']]],
    'accessFlags' : [ 0x20, ['unsigned int']],
    'serialNumber' : [ 0x24, ['unsigned int']],
    'pDvmDex' : [ 0x28, ['pointer', ['DvmDex']]],
    'status' : [ 0x2c, ['int']],
    'verifyErrorClass' : [ 0x30, ['address']],
    'initThreadId' : [ 0x34, ['unsigned int']],
    'objectSize' : [ 0x38, ['unsigned int']],
    'elementClass' : [ 0x3c, ['pointer', ['ClassObject']]],
    'arrayDim' : [ 0x40, ['int']],
    'primitiveType' : [ 0x44, ['int']],
    'super' : [ 0x48, ['pointer', ['ClassObject']]],
    'classLoader' : [ 0x4c, ['pointer', ['Object']]],
    'initiatingLoaderList' : [ 0x50, ['int']],
    'interfaceCount' : [ 0x58, ['int']],
    'interfaces' : [ 0x5c, ['pointer', ['pointer', ['ClassObject']]]],
    'directMethodCount' : [ 0x60, ['int']],
    'directMethods' : [ 0x64, ['address']],
    'virtualMethodCount' : [ 0x68, ['int']],
    'virtualMethods' : [ 0x6c, ['pointer', ['Method']]],
    'vtableCount' : [ 0x70, ['int']],
    'vtable' : [ 0x74, ['pointer', ['pointer', ['Method']]]],
    'iftableCount' : [ 0x78, ['int']],
    'iftable' : [ 0x7c, ['address']],
    'ifviPoolCount' : [ 0x80, ['int']],
    'ifviPool' : [ 0x84, ['int']],
    'ifieldCount' : [ 0x88, ['int']],
    'ifieldRefCount' : [ 0x8c, ['int']],
    'ifields' : [ 0x90, ['pointer', ['InstField']]],
    'refOffsets' : [ 0x94, ['int']],
    'sourceFile' : [ 0x98, ['pointer', ['char']]],
    'sfieldCount' : [ 0x9c, ['int']],
    'sfields' : [ 0xa0, ['pointer', ['StaticField']]],
    }],
#dalvik/Hash.h
'HashEntry' : [ 0x8, {
    'hashValue' : [ 0x0, ['unsigned int']],
    'data' : [ 0x4, ['pointer', ['void']]],
    }],
'HashTable' : [ 0x18, {
    'tableSize' : [ 0x0, ['int']],
    'numEntries' : [ 0x4, ['int']],
    'numDeadEntries' : [ 0x8, ['int']],
    'pEntries' : [ 0xc, ['pointer', ['HashEntry']]],
    'freeFunc' : [ 0x10, ['address']],
    'lock' : [ 0x14, ['int']],
    }],
#dalvik/Globals.h
'DvmGlobals' : [ 0x1c, {
    'bootClassPathStr' : [ 0x0, ['pointer', ['char']]],
    'classPathStr' : [ 0x4, ['pointer', ['char']]],
    'heapStartingSize' : [ 0x8, ['unsigned int']],
    'heapMaximumSize' : [ 0xc, ['unsigned int']],
    'heapGrowthLimit' : [ 0x10, ['unsigned int']],
    'stackSize' : [ 0x14, ['unsigned int']],
    'verboseGc' : [ 0x18, ['bool']],
    'verboseJni' : [ 0x19, ['bool']],
    'verboseClass' : [ 0x1a, ['bool']],
    'verboseShutdown' : [ 0x1b, ['bool']],
    'jdwpAllowed' : [ 0x1c, ['bool']],
    'jdwpConfigured' : [ 0x1d, ['bool']],
    'jdwpTransport' : [ 0x20, ['int']],
    'jdwpServer' : [ 0x24, ['bool']],
    'jdwpHost' : [ 0x28, ['pointer', ['char']]],
    'jdwpPort' : [ 0x2c, ['int']],
    'jdwpSupend' : [ 0x30, ['bool']],
    'profilerClockSource' : [ 0x34, ['int']],
    'lockProfThreshold' : [ 0x38, ['unsigned int']],
    'vfprintfHook' : [ 0x3c, ['address']],
    'exitHook' : [ 0x40, ['address']],
    'abortHook' : [ 0x44, ['address']],
    'isSensitiveThreadHook' : [ 0x48, ['address']],
    'jniGrefLimit' : [ 0x4c, ['int']],
    'jniTrace' : [ 0x50, ['pointer', ['char']]],
    'reduceSignals' : [ 0x54, ['bool']],
    'noQuitHandler' : [ 0x55, ['bool']],
    'verifyDexChecksum' : [ 0x56, ['bool']],
    'stackTraceFile' : [ 0x58, ['pointer', ['char']]],
    'logStdio' : [ 0x5c, ['bool']],
    'dexOptMode' : [ 0x60, ['int']],
    'classVerifyMode' : [ 0x64, ['int']],
    'generateRegisterMaps' : [ 0x68, ['bool']],
    'registerMapMode' : [ 0x6c, ['int']],
    'monitorVerification' : [ 0x70, ['bool']],
    'dexOptForSmp' : [ 0x71, ['bool']],
    'preciseGc' : [ 0x72, ['bool']],
    'preVerify' : [ 0x73, ['bool']],
    'postVerify' : [ 0x41, ['bool']],
    'concurrentMarkSweep' : [ 0x75, ['bool']],
    'verifyCardTable' : [ 0x76, ['bool']],
    'disableExplicitGc' : [ 0x77, ['bool']],
    'assertionCtrlCount' : [ 0x78, ['int']],
    'assertionCtrl' : [ 0x7c, ['address']],
    'executionMode' : [ 0x80, ['int']],
    'initializing' : [ 0x84, ['bool']],
    'optimizing' : [ 0x85, ['bool']],
    'properties' : [ 0x88, ['address']],
    'bootClassPath' : [ 0x8c, ['address']],
    'bootClassPathOptExtra' : [ 0x90, ['address']],
    'optimizingBootstrapClass' : [ 0x94, ['bool']],
    'loadedClasses' : [ 0x98, ['pointer', ['HashTable']]],
    'classSerialNumber' : [ 0x9c, ['int']],
    'initiatingLoaderList' : [ 0xa0, ['address']],
    'internLock' : [ 0xa4, ['int']],
    'internedStrings' : [ 0xa8, ['address']],
    'literalStrings' : [ 0xac, ['address']],
    'classJavaLangClass' : [ 0xb0, ['address']],
    'offJavaLangRefReference_referent' : [ 0x1c8, ['int']],
    'offJavaLangString_value': [ 0x214, ['int']],
    'offJavaLangString_count': [ 0x218, ['int']],
    'offJavaLangString_offset': [ 0x21c, ['int']],
    'offJavaLangString_hashCode': [ 0x220, ['int']],
    'gcHeap' : [ 0x290, ['address']],
    #TODO: missing objects, but this should be enough for now
    }],
}

class HashTable(obj.CType):
"""A class extending the Dalvik hash table type"""
def get_entries(self):
    offset = 0x0
    count = 0
    while offset < self.tableSize * 0x8:
        hashEntry = obj.Object('HashEntry', offset = self.pEntries + offset, vm = self.
            obj_vm)
        # 0xcbcacccd is HASH_TOMBSTONE for dead entries
        if hashEntry.hashValue == 0 or hashEntry.data == 0xcbcacccd:
            offset += 0x8
            count += 1
            continue

        yield hashEntry.data

        # each HashTable entry is 8 bytes (hash* + data*) on the heap
        offset += 0x8
        count += 1

class ClassObject(obj.CType):
"""A class extending the Dalvik ClassObject type"""

def getIFields(self):
    clazz = self
    # is this an instance object? If so, get the actual system class
    while dalvik.getString(clazz.clazz.descriptor)+"" != "Ljava/lang/Class;":
        clazz = clazz.clazz
    while clazz:
        i = 0
        while i < clazz.ifieldCount:
            yield obj.Object('InstField', offset = clazz.ifields+i*0x14, vm = clazz.obj_vm)
            i+=1
        clazz = clazz.super

def getIField(self, nr):
    count = 0
    for field in self.getIFields():
        if count == nr:
            return field
        count += 1

def getIFieldByName(self, name):
    for field in self.getIFields():
        if dalvik.getString(field.name)+"" == name:
            return field
    return None

def getIFieldAsString(self, name):
    ifield = self.getIFieldByName(name)
    jvalue = obj.Object('JValue', offset = self.obj_offset + ifield.byteOffset, vm = self.
        obj_vm)

    if jvalue.Object == 0:
        return "NULL"

    return dalvik.parseJavaLangString(jvalue.Object, self.obj_vm)

def getIFieldAsBool(self, name):
    ifield = self.getIFieldByName(name)
    jvalue = obj.Object('JValue', offset = self.obj_offset + ifield.byteOffset, vm = self.
        obj_vm)

    return jvalue.bool

def getIFieldAsInt(self, name):
    ifield = self.getIFieldByName(name)
    jvalue = obj.Object('JValue', offset = self.obj_offset + ifield.byteOffset, vm = self.
        obj_vm)

    return jvalue.int

def getIFieldAsClassObject(self, name):
    ifield = self.getIFieldByName(name)
    jvalue = obj.Object('JValue', offset = self.obj_offset + ifield.byteOffset, vm = self.
        obj_vm)

    return obj.Object('ClassObject', offset = jvalue.Object, vm = self.obj_vm)

def getIFieldAsObject(self, name):
    ifield = self.getIFieldByName(name)
    jvalue = obj.Object('JValue', offset = self.obj_offset + ifield.byteOffset, vm = self.
        obj_vm)

    return obj.Object('Object', offset = jvalue.Object, vm = self.obj_vm)
def getIFieldAsArray(self, name):
    ifield = self.getIFieldByName(name)
    jvalue = obj.Object('JValue', offset = self.obj_offset + ifield.byteOffset, vm = self.
        obj_vm)

    array = [ ]

    for o in dalvik.parseArray(jvalue.Object, self.obj_vm):
        # not all array fields have a value
        if o == 0:
            continue
        array.append(o)

    return array

def getDirectMethods(self):
    i = 0
    while i < self.directMethodCount:
        method = obj.Object('Method', offset = self.directMethods+i*0x38, vm = self.obj_vm)
        yield method
        i+=1

def getVirtualMethods(self):
    i = 0
    while i < self.virtualMethodCount:
        method = obj.Object('Method', offset = self.virtualMethods+i*0x38, vm = self.obj_vm)
        yield method
        i+=1

class DalvikObjectClasses(obj.ProfileModification):
conditions = {'os': lambda x: x == 'linux'}
before = ['LinuxObjectClasses']

def modification(self, profile):
    profile.vtypes.update(dalvik_vtypes)
    profile.object_classes.update({'HashTable': HashTable,
                                   'ClassObject': ClassObject})
```

A.2. Plugin Documentation README.dalvik

Listing A.2: README.dalvik

```
Dalvik Support for Volatility
=============================

The following plugins are provided:

 - dalvik_find_gdvm_offset
 - dalvik_vms
 - dalvik_loaded_classes
 - dalvik_class_information
 - dalvik_find_class_instance
 - dalvik_app_k9mail_accounts
 - dalvik_app_k9mail_listmails
 - dalvik_app_k9mail_mail
 - dalvik_app_whatsapp_conversations
 - dalvik_app_whatsapp_conversation
 - dalvik_app_contacts
 - dalvik_app_mirrored

All plugins are actually linux plugins, so they need a valid profile and
lime [1] memory dump.

The plugins have been successfully tested on two Android devices running
Ice Cream Sandwich (ICS): Huawei Honor (U8860) and Samsung Galaxy S2
(I9100).

The Volatility 2.3-devel branch, or any later version, is needed.
Especially revision r2659 has been verified to work properly with these
plugins.


Detailed plugin description:
============================

dalvik_find_gdvm_offset
----------------------

The global struct DvmGlobals (gDvm) [2] is the foundation for all
provided plugins. To locate it in an actual memory dump, we need to know
where the data section (in which gDvm is mapped) of libdvm is mapped
within a specific process. This information can be taken from the
proc_maps plugin. For example (for zygote):

0x408f9000-0x409aa000 r-x          0 259: 1          915            2508 /system/lib/libdvm.so
0x409aa000-0x409b2000 rw-     724992 259: 1          915            2508 /system/lib/libdvm.so

So the data section starts at 0x409aa000. Within this range, gDvm can be
found. The dalvik_find_gdvm_offset scans this address space and tries to
locate gDvm and finally prints its offset. This offset can be given to
all further plugins via the '-o' switch in order to prevent rescanning,
which saves quite some time.

Optional argument: -p PID, --pid=PID
 Specify the PID of one process you know of to run in a DalvikVM. For
 instance, zygote. Speeds up offset calculation.


dalvik_vms
----------

Lists all Dalvik Virtual Maschines found in the memory dump and some
additional information such as heapStartingSize, number of loaded
classes, etc.. Limit to specific VMs with the '-p PID' switch.
```

```
Optional argument: -o GDVM_OFFSET (in hex)
 Specify the gDvm offset to speed up calculations. See the
 dalvik_find_gdvm_offset plugin for more information

Optional argument: -p PID, --pid=PID
 Limit to specific VMs which correspond to the given PID.


dalvik_loaded_classes
---------------------

List all loadedClasses from a specific DalvikVM instance together with
some information. Most important is the 'Offset' column, which can be
used for listing specific class information with the
dalvik_class_information plugin.

Optional argument: -o GDVM_OFFSET (in hex)
 Specify the gDvm offset to speed up calculations. See the
 dalvik_find_gdvm_offset plugin for more information

Optional argument: -p PID, --pid=PID
 Limit to specific VM which correspond to the given PID.


dalvik_class_information
------------------------

List concrete information about a specific system class, such as number
of instance fields or method names.

Mandatory argument: -c CLASS_OFFSET, --class_offset=CLASS_OFFSET
 Offset of a class object within its process address space. Usually
 taken from the dalvik_loaded_classes plugin.

Mandatory argument: -p PID, --pid=PID
 This needs to match the process in which the class object of interest
 is defined. Specifically, this is the PID printed on the same row as
 the CLASS_OFFSET argument from the dalvik_loaded_classes plugin.

Optional argument: -o GDVM_OFFSET (in hex)
 Specify the gDvm offset to speed up calculations. See the
 dalvik_find_gdvm_offset plugin for more information


dalvik_app_*
------------

Concrete instance objects (in contrast to preloaded system classes) are
allocated in the dalvik-heap of each process. So in order to analyze
specific applications together with there instance data, we need a
concrete instance object pointer. This pointer can be aquired manually,
for instance via hprof heap dumps (cf. Eclipse MAT) or via methods of
scanning. For the latter, the dalvik_find_class_instance (see below) is
provided. It takes a pointer to a system class (got via the
dalvik_loaded_classes plugin) and scans te dalvik heap for possibly
matching instance objects. The aquired pointer can then be passed to the
corresponding app plugins. Please note: The dalvik_find_class_instance
plugin might require quite some time (>5m) to find an appropriate
pointer.

Example plugin for reading app information: dalvik_app_mirrored

 Given an instance object ('-c'), it lists the current active article
 titles shown by the application called 'Mirrored', a news reader. Of
 course, this requires an appropriate memory dump.

Mandatory argument: -c CLASS_OFFSET, --class_offset=CLASS_OFFSET
 Offset of a concrete class instance object. The
 dalvik_find_class_instance plugin can help to find one.

Mandatory argument: -p PID, --pid=PID
 This needs to match the process in which the class object of interest
 is defined.


dalvik_find_class_instance
--------------------------

Takes a process ID and a system class offset and tries to locate
instance objects of the system class within the processes address space.

Mandatory argument: -c CLASS_OFFSET, --class_offset=CLASS_OFFSET
 Offset of a class object within its process address space. Usually
 taken from the dalvik_loaded_classes plugin.

Mandatory argument: -p PID, --pid=PID
 This needs to match the process in which the class object of interest
 is defined. Specifically, this is the PID printed on the same row as
 the CLASS_OFFSET argument from the dalvik_loaded_classes plugin.


Helper modules:
===============

dalvik.py
---------

Helper functions for parsing DalvikVM objects such as java/lang/String
or array lists.


dalvik_vtypes.py (volatility/plugins/overlays/linux/)
-----------------------------------------------------

Data structure definitions and extending helper functions.


Explanatory Volatility session
```

```
==============================

 [...] = --profile=Linux<insert your profile here>x86 -f <insert lime memory dump here>

 $ ./vol.py [...] dalvik_find_gdvm_offset
 DvmGlobals offset
 -----------------
 0x7c58

 $ ./vol.py [...] linux_pslist | grep Mirrored
 0xe0684960 .homac.Mirrored      1547            10066           Tue, 04 Sep 2012 18:24:44 +0000

 $ ./vol.py [...] dalvik_loaded_classes -o 0x7c58 -p 1547 | grep 'ArticlesList;'
 PID   Offset     Descriptor                        sourceFile
 ----- ---------- --------------------------------- -----------------
  1547 0x415059d0 Lde/homac/Mirrored/ArticlesList;  ArticlesList.java

 $ ./vol.py [...] dalvik_find_class_instance -p 1547 -c 0x415059d0
 SystemClass                                        Instance
 -------------------------------------------------- -----------------
 0x415059d0                                         0x415060c8
 [...]

 $ ./vol.py [...] dalvik_app_mirrored -p 1547 -c 0x415060c8

 Nr  Title
 --- --------------------------------------------------
   1 Paralympics-Teilnehmerin Wyludda: Zweite Karriere nach Olympia-Gold
   2 Antarktis: Tourismus nicht Schuld an Pinguin-Schwund
   3 Installation in Rio: Guck mal, wer da traeumt
   [...]


[1] http://code.google.com/p/lime-forensics/
[2] cf. dalvik/vm/Globals.h in ICS's source tree
```

Eidesstattliche Erklärung

Hiermit versichere ich, dass diese Abschlussarbeit von mir persönlich verfasst ist und dass ich keinerlei fremde Hilfe in Anspruch genommen habe. Ebenso versichere ich, dass diese Arbeit oder Teile daraus weder von mir selbst noch von anderen als Leistungsnachweise andernorts eingereicht wurden. Wörtliche oder sinngemäße Übernahmen aus anderen Schriften und Veröffentlichungen in gedruckter oder elektronischer Form sind gekennzeichnet. Sämtliche Sekundärliteratur und sonstige Quellen sind nachgewiesen und in der Bibliographie aufgeführt. Das Gleiche gilt für graphische Darstellungen und Bilder sowie für alle Internet-Quellen.

Ich bin ferner damit einverstanden, dass meine Arbeit zum Zwecke eines Plagiatsabgleichs in elektronischer Form anonymisiert versendet und gespeichert werden kann. Mir ist bekannt, dass von der Korrektur der Arbeit abgesehen werden kann, wenn die Erklärung nicht erteilt wird.

Nürnberg, den 7. Januar 2013

Holger Macht