# Using open source tools to investigate malware in the Android Operating System

Tareq Hanaysha, Dale Lindskog, Ron Ruhl
Department of Information Systems Security Management
Concordia University College of Alberta
7128 Ada Boulevard, Edmonton, AB T5B 4E4, Canada
hanaysha@live.com,{dale.lindskog, ron.ruhl}@concordia.ab.ca

***Abstract* — The Android operating system is increasingly exposed to a growing list of dangerous malware attacks, these attacks cost users and businesses. There is considerable research into Android malware detection, malware behavior, interaction and permissions. However, there is much less research effort around digital forensic on the Android platform, which has been the victim of the malware. Very recent enhancements to the open source Volatility framework make it a useful memory image forensic tool to investigate Android malware. Android memory forensics is immature, it is also complicated to conduct compared to other popular operating systems. In this paper, we explore and document the processes of building an open source Android forensics investigation environment, planting samples of Android malware, acquiring Android memory images of these samples and the forensic investigation of them. This research will facilitate future memory forensics investigation of Android systems and Android malware analysis.**

***Index Terms* — Android, Malware, Forensics, Memory Image, and Volatility Framework.**

## I. INTRODUCTION

Android operated devices are one of the most competitive technology devices in the market, with the fastest growing market share within the mobile industry [1]. Technology experts predict that Android will dominate the mobile market in the coming decade. Additionally, recent research shows a huge year over year increase in the number of Android specific malware attacks [1,3]. It is relatively straightforward to investigate such attacks when they occur on mature operating system platforms such as Windows and Linux. However, due to the immaturity of Android memory image forensics, it is relatively problematic and time consuming to conduct such investigations on Android systems. In this research, we take advantage of recent advances in Android memory forensics, utilizing the open source digital forensic suite, Volatility. we explore a sample of these malware attacks, this powerful investigation framework written in Python, recently capable of reading memory images from different Android kernel versions, and capable of performing a wide range of memory analysis and digital evidence extraction.

Volatility analyzes memory images, which must be extracted from the physical memory of the Android device, the only freely available utility capable of extracting these images is Linux Memory Extractor (LiME) [12]. This loadable kernel module can acquire the full Android memory address range, either over the network or via an SdCard [10]. LiME, along with various new Android specific Volatility plugins and a custom built profile for Volatility, are used in our research, in order to analyze running malware through the exploration of hidden processes, process structure, malicious Android package activities, process caches, suspicious network connections, and other suspicious executed code.

Commercial tools like forensic toolkit and Encase [19,17] can be used for Android content recovery and forensic investigation, and these tools are fairly easy to deploy, but are expensive solutions for small to medium business. This research illustrates Android memory forensics using open source, freely available tools. We use the recently available, first stable version of Volatility, rather than older beta releases, as this will provide more timely and accurate evidence and analysis, because of its very recently developed Android specific plugins which allow the investigator to explore Android Dalvik instances, process structure and memory caches.

In the following sections, we discuss current work in the Android memory forensics field, then describe building an Android memory forensics investigation environment, we also discuss the challenges involved in acquiring an Android memory image and finally we describe an experimental forensics investigation of a number of Android malware samples.

## II. Android memory forensics: an overview

There is wide consensus, at least in general outline, about the procedures involved in a forensic investigation [5,17,21]. NIST guidelines on cellphone forensics (SP 800-101 and 800-86), for example, conform to this consensus, and have defined the digital forensics procedure as consisting of four major steps: acquire, preserve, analyze and present. This specification is very similar to any typical forensic investigation process description, and applies regardless of the medium being investigated, including memory. Android memory forensics is immature, however, and each step in this process is more complicated than with more mature operating systems. Consider first acquisition: for the Linux operating system, Ivor Kollar`s dev/fmem memory acquisition tool has long been available, and this tool is capable of full Linux memory capture; other mature memory acquisition tools for Linux include Memdump by IBM, crash utility by Red Hat and many more [22,23,24]. The case is similar to the Windows operating systems, with many mature memory acquisition tools, such as the popular Encase WinEn and MoonSols toolkit, Once a memory image of these systems is acquired and preserved, many forensics investigation tools, including Volatility Crash dump analyzer, Raw Image Analyzer and FORENSIC TOOLKIT, come with mature Windows and Linux investigation tools. These tools are capable of reading different memory image formats, coming from different operating system versions, decoding them into digital artifacts, ready for evidence analysis and presentation. In brief, this process seems to be straightforward and well understood on popular operating systems platforms [7,10,17,19,25].

As noted, the situation is currently much more complicated when it comes to Android memory. While non-volatile flash NAND memory for an Android system can be easily acquired using Android Debug Bridge, the evidence extracted from this process will be helpful only for recovery of file system artifacts, e.g. videos or images, whereas it is in Android memory that we may find such potentially crucial artifacts as executed code, processes and their structure, active network connections, and so on.

Android memory acquisition is the first and an essential step in an Android memory forensics investigation, but Android memory is complicated to acquire, by comparison.

Like other operating systems, Android is constantly releasing new software version upgrades, and has diverse hardware support. This makes it difficult for forensics researchers to have an automated system for Android memory acquisition, since the latest and most mature solution, LiME, is developed for Linux (Android uses a modified Linux kernel), but is a loadable kernel module requiring compilation to work on Android, a task further complicated by the fact that Android runs on the advanced RISC machines architecture [17,19]. LiME is capable of acquiring a full memory image from an Android device, but must be compiled for the advanced RISC machines architecture and then compiled for an Android specific kernel image, and finally loaded as a kernel module into an Android device [8,10]. These steps are preliminary to an actual Android memory acquisition, and can be very challenging to perform, especially for those who are not experts in both Android and the Linux operating system [17], and in systems administration generally.

Volatility forensics investigation framework, an open source tool written in Python, utilizes plugins to analyze the structure of, and produce output about, memory. Volatility is capable of analyzing most operating systems, and recently advertises support for investigating Android phones. The main challenge of investigating Android phones was their memory address space support, since Android memory structure is different than other operating systems, and translating it to an understandable output was a challenge that developers have just recently achieved, with the help of LiME.

This is not to say that demonstrations of these tools are not documented. Joseph Syvle, e.g., introduces LiME and briefly studies an infected application on Android, including a high-level investigation of Android application permissions. Holger Macht [9] utilized LiME and Volatility in his research, and created Volatility privacy analysis plugin scripts. This documentation, however, does not focus on the complexities and details of the full forensic investigation process itself, including its preliminaries, and moreover was written at a time when Volatility advanced RISC machines support was premature.

## III. Android memory forensics investigation environment and acquisition

A forensics investigation is typically divided into four basic steps: acquisition, preservation, analysis and presentation. An obvious prerequisite to this process is that the investigator has a usable forensic investigation environment, so that the process can in fact be seen as consisting of five steps: build, acquire, preserve, analyze and present. The build step focuses on having those tools and techniques in place that are necessary for performing subsequent steps of the forensic investigation process, while acquisition consists in acquiring a forensically sound copy of the media under investigation. Because the first two steps of this process (build and acquire) are, for Android memory investigations, particularly complicated, and in fact interrelated, we dedicate this section to a discussion of their complexities and potential pitfalls, but we depict in outline the whole process in Figure 1.

The first two steps of this five step process are for us interrelated because, unlike in the case of more mature operating systems, we lack an automated Android memory acquisition process. Android operating system version updates are consistent, fast and diverse; moreover, Android prohibits automated installation of loadable kernel modules. These two facts about Android make an acquisition considerably more complicated, and make it essential that the proper tools and techniques are in place, or at least well understood, prior to the time sensitive acquisition step of a forensic investigation involving Android memory. The aspects of the build step that we cover here are: (1) ensuring that a usable LiME kernel module is available to load on to the Android device for memory acquisition; and (2) that Volatility, a memory forensics suite, is installed and prepared for an Android specific memory investigation.

Online guides and instructions like Volatility Wiki, the Android Development website, and others [4,5,6,8] have written excellent guides about building an Android memory investigation environment. They detail the cross-compilation process of an emulated Android kernel image, and the compilation of the LiME kernel module for that Android emulated image. However, there are complexities and pitfalls not emphasized in these online guides. For example, Android GNU compiler collection versions can conflict with Macintosh and Linux 12.6 64 bit operating systems, and lack of careful attention to these and similar details will make it impossible to prepare the LiME loadable kernel module source code for compilation.

In setting up the environment for an Android memory acquisition and analysis, we do not need a full Android development repository (10 GB in size), and using virtualization technology will allow snapshots or cloned backups of the investigation progress, which is especially important when the processes are immature and therefore buggy, as in the case of Android. The acquisition step in particular of Android memory is, however, challenging. When compiling an emulated Android goldfish kernel image, it is essential to set various environment variables in order to direct the kernel cross-compiler to the right Android tool chain, needed to support the host operating system; these variables can be defined within the host operating system bash profile path or at the terminal, and either approach will avoid compiler errors, once our kernel variables are defined, Android kernel image settings must be based on the current Goldfish kernel code file, in order to successfully load the LiME kernel module. Kernel module loading must to be enabled on the Android device containing the memory to be acquired, and this must be done during kernel configuration, and using the configuration file provided with the kernel code.

Furthermore, the Android kernel version and the LiME kernel module version must match, and this can be verified by examining the LiME kernel module headers and the compiled Android kernel version, a failure to match kernel and kernel module will produce an init_module failure error, because of Android's module versioning check policy, designed in part to protect users from security threats.

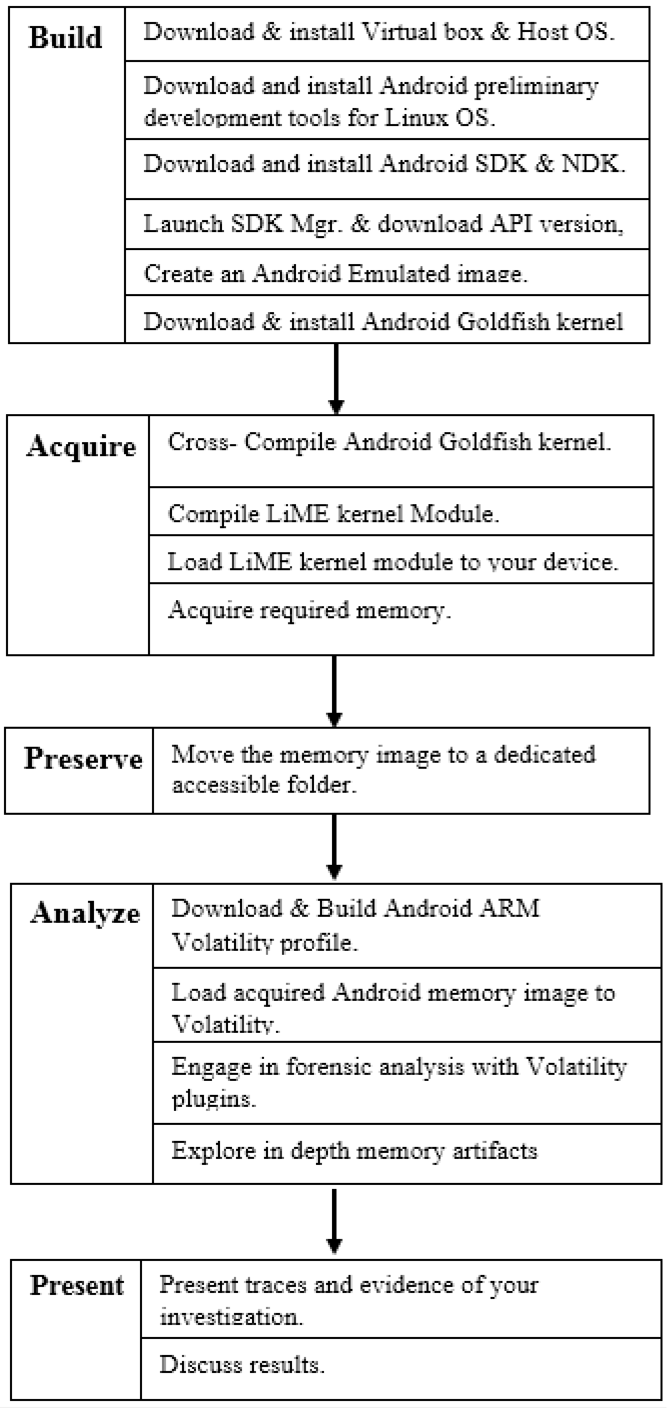
Build
Download & install Virtual box & Host OS.
Download and install Android preliminary development tools for Linux OS.
Download and install Android SDK & NDK.
Launch SDK Mgr. & download API version,
Create an Android Emulated image.
Download & install Android Goldfish kernel
Acquire
Cross- Compile Android Goldfish kernel.
Compile LiME kernel Module.
Load LiME kernel module to your device.
Acquire required memory.
Preserve
Move the memory image to a dedicated accessible folder.
Analyze
Download & Build Android ARM Volatility profile.
Load acquired Android memory image to Volatility.
Engage in forensic analysis with Volatility plugins.
Explore in depth memory artifacts
Present
Present traces and evidence of your investigation.
Discuss results.

Figure 1: Android Forensics Investigation workflow

## IV. EXPERIMENTAL INVESTIGATION

Like most modern operating systems, Android divides memory into user and kernel space. The Android kernel is based on the Linux kernel; with Android, user space contains the Dalvik virtual machine (VM), and application software (distributed in APK format) runs on top of the VM. Each Android package contains the process instructions, set in its class.dex file, the message digest of the signatures, and the resource file of permissions (named Manifest.xml) [25].

Android malware can be packaged as APKs [18,19], whether they be Worms, Trojans, Viruses, Rootkits or Botnets. We acquired a diverse sample of these Android packages, and then classified them according to their attack type, symptoms, purpose and impact on the user or system (Table 1).

| Title | Attack type | Symptoms | Purpose | Impact |
|---|---|---|---|---|
| O Bada Trojan | Backdoor | Escalated Privileges | Malicious Apps | System availability |
| ZitMo (Zeus) | Update | Confirmation of functions | mTAN | Money |
| SMS Trojan | Malicious | Escalated Privileges | Malicious Apps | System availability |
| Angry Birds | Malicious | Hidden transactions | Premium Charges | Money |

Table 1: Android malware attack classifications.

This experiment utilizes Volatility version 2.3.1, with a custom built Android advanced RISC machines profile, made specifically for our goldfish 2.6.29-ge3d684d kernel image; this image was cross-compiled using the 4.7 Android software development kit GNU compiler collection, and then equipped with a compiled LiME 1.1 module to the above kernel image. This set of tools was utilized to investigate the sample malware depicted in Table 1.

We first ran the Android kernel image emulator, acquired a clean memory image, preserved it to a dedicated folder, and then performed our memory investigation on it. Second we uploaded our first Android ZitMo infected application to the emulated Android kernel image, acquired another image after running the infected application, and then wiped the Android kernel image. Third, we uploaded the Angry Birds infected application, and again uploaded the LiME module, ran the Angry Birds application, followed the instructions from the application prompt, acquired an image, preserved the image and wiped. Fourth, we uploaded the Obada Trojan application, ran the emulator and, since this application is a backdoor, we couldn't find any actual Android application in our emulator to run, so we wiped the emulator kernel. Finally the SMSsend trojan was uploaded, along with LiME, and the malware was executed according to its instructions, and again the memory image was acquired and preserved. For each sample malware, data analysis was compared with the clean image, looking for:

- Hidden processes associated with malware infected applications.
- Changes to PIDs & PPIDs.
- Attempts to initiate Internet connections.
- Applications initiating system administration requests and escalated privilege access.
- Applications launching third party Android application stores.
- Noticeable changes to processing time, e.g. slow response.

In achieving this goal, first we ran psxview plugin, looking for hidden processes, by gathering the processes structure view from different places like the kernel proc list, the kernel process hash table and the kmem_cache, if a process is one of these places, but not in the other, then this leads to further investigation of that process, this is a suspicious behavior, where the pstree plugin was used to identify processes relationship and to see the hierarchical view of these processes, for instance looking at ZItmo, processes name com.Android.mms with PID 692 and UID 10017, falls under system_server PID 95, this process is a child of the parent Zygote PID 42, following this investigation procedure, we are able to identify processes, escalated privileges and malicious processes structure. For further evidence gathering and digging into a certain suspicious processes structure, we used pslist | grep zygote, looking into app processes on the disk, Zygote controls the Android Dalivk machine apps, it contains the shared libraries like the libdvm.so, these libraries control the Android process classes, piping Zygote to a file will grab all the contents of the Android Dalvik machine, when we did that using proc_maps –p 42 and save to zygote-maps file, then looked at the content of each sample and with the help og the pstree plugin, we figured all Android processes fall under the process system_server which is a sub processes of the Zygote, piping the system_server process map to a file using proc_maps –p 95 to a file called system_server-maps, at this point, we preserved data of all Zitmo system processes, For instance com.Android.mms process from pstree list, showed more details under system_server process, an inode 505 with reading flag, now we dumped a memory map of this application, and looked at each instance of this application to understand the structure, and compared it to the com.Android.mms code under the system_server process, we were able to see the code with the memory app but not in the system_server proc.

Then to look at this from a different angle, we moved into networking investigation, looking for networking activities, started with running arp plugin for every sample to see my arp table, then utilizing the ifconfig plugin, trying to figure my network interfaces and ip addresses, where we found the route_cache pluging showing caches of the best route to take when an application is trying to initiate a connection to an outside server, with that been said, any malicious address means evidence of malicious application activities and leads to further investigation, we took all the ip addresses and looked them up, using website that can identify blacklisted ip addresses.

There were a number of notable results derived from examining our memory images samples. Again, In the case of infection with Zitmo malware app, the plugin pslist showed a created PID of 559 and inode 505 within the system_server, notably after loading Zitmo app, the system to initiate a process named com.Android.mms, PID 692 under UID 10017, in an attempt to request the user to confirm premium message charges initiation. Whereas, in the Angry Birds application analysis, once this application was launched, a new process with PID 925 was also triggered and this application prompts the user to click on a link, in order to unlock the Angry Birds game to its full version, this second application was named ds.rio.unlocker in the process list within memory, under UID 10047, an argument that is leading to another processes of triggering the Internet browser in an attempt to access the Internet, we found this creating a memory bug, also to exhaust the system resources and make my Android very slow, after that, running the route_cache plugin, we were able to confirm attempts to connect to both 204.191.12.35 and 204.191.12.24 as well.

Installing and running the Smssend application directs the user to the Goolge Play store, and interesting enough, within the memory image acquired, PID 919 shows as an install.app process with UID 10047, this process seems to have multiple inodes within the Zygote (inodes 725, 529 and 2457) which seem like an attempt to escalate privileges; also, an android.browser process ID 955 with UID 10035 was visible in the memory image, with a series of child processes: a process named compiler with PID 961, one named background handler with PID 972, thread-80 with PID 942, and logcat with PID 947; these processes seemed to slow the operating system, and freeze the Google Play store connection.

## V. CONCLUSION AND FUTURE WORK

In this paper, we described Android memory imaging investigation processes and procedures, mainly building an Android environment for memory image investigation and its acquisition. We acquired samples of Android malware memory images, then examined this sample utilizing digital forensics science, this is found to be different than performing such an investigation on popular operating systems platforms like Linux, Windows and Macintosh. The acquisition of Android memory is yet to be automated, with LiME kernel module being the one and only open source solution available. We explore the process of loading this module to an Android system, utilizing it and acquiring an image. We detailed the challenges of cross-compiling LiME kernel module, the module versioning compatibility issues with Android kernel and the difficulty in loading kernel module to an Android kernel. This research also used Volatility framework to investigate the acquired images, with plugins that support LiME memory image address space, we were able to identify suspicious and hidden processes and in addition to that, we noted and observed escalated system privileges and internet connection requests, these networking requests to connect to malicious destination servers. With this being said, we found some of Volatility plugins still to mature and the Android memory acquisition processes to be automated, this field of Android system continues to evolve with the speed of Android version upgrades and diversity.

## APPENDIX

### Code Appendix

1. LiME makefile – sample file of LiME makefile modified.

```
obj-m := LiME.o

LiME-objs := tcp.o disk.o main.o

KDIR_GOLD := /home/hanaysha/Android-src/

KVER := $(shell uname -r)

PWD := $(shell pwd)

CCPATH := /home/hanaysha/Android-ndk/toolchains/Advanced RISC Machines-Linux-Androideabi-4.7/prebuilt/Linux-x86_64/bin/

default:

        # cross-compile for Android emulator

        $(MAKE) ARCH=Advanced RISC Machines CROSS_COMPILE=$(CCPATH)/Advanced RISC Machines-Linux-Androideabi- -C $(KDIR_GOLD) EXTRA_CFLAGS=-fno-pic M=$(PWD) modules

        $(CCPATH)/Advanced RISC Machines-Linux-Androideabi-strip --strip-unneeded LiME.ko

        mv LiME.ko LiME-goldfish.ko

        $(MAKE) tidy

tidy:

        rm -f *.o *.mod.c Module.symvers Module.markers modules.order \.*.o.cmd \.*.ko.cmd \.*.o.d

        rm -rf \.tmp_versions

clean:

        $(MAKE) tidy

        rm -f *.ko
```

2. Module dwarf compilation makefile [8]

```
obj-m += module.o

KDIR := /home/hanaysha/Android-src/

CCPATH := /home/hanaysha/Android-ndk/toolchains/Advanced RISC Machines-Linux-Androideabi-4.7/prebuilt/Linux-x86_64/bin/

DWARFDUMP := /home/hanaysha/dwarf/dwarfdump/dwarfdump

-include version.mk

all: dwarf

dwarf: module.c

	$(MAKE) ARCH=Advanced RISC Machines CROSS_COMPILE=$(CCPATH)/Advanced RISC Machines-Linux-Androideabi- -C $(KDIR) CONFIG_DEBUG_INFO=y M=$(PWD) modules
	$(DWARFDUMP) -di module.ko > module.dwarf
```

## REFERENCES

[1] Scott Wilson. Praveen Tanguturi. (Aug 2011). The Deloitte Open Mobile Survey 2012: The growth era accelerates. [Online] available: https://www.deloitte.com/assets/Dcom-Norway/Local%20Assets/Documents/Publikasjoner%202012/deloitte_openmobile2012.pdf

[2] Alcatel. Lucent. (July 2013). Kindsight Security Labs Malware Report – Q2 2013. [Online] Available: http://www.kindsight.net/sites/default/files/Kindsight-Q2-2013-Malware-Report.pdf

[3] TREND MICRO. TrendLabs. (2Q 2013). Security Roundup: Mobile Threats Go Full Throttle, Device Flaws Lead to Risky Trail.[Online] Available: http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/rpt-2q-2013-trendlabs-security-roundup.pdf

[4] Android Developers. (Aug 25, 2013). The Android Open Source Project. [Online] Available: http://source.Android.com/index.html

[5] Simson L. Garfinkel. (July 12, 2011). Navy's Research University: Android Forensics Simplified. [Online] Available: http://simson.net/ref/2011/2011-07-12%20Android%20Forensics.pdf

[6] Stephan Chenette. (Oct 23, 2013). IOActive labs: Building Custom Android Malware for Penetration Testing. [Online] Available: http://www.slideshare.net/schenette/2013-toorcon-san-diego-building-custom-Android-Malware-for-penetration-testing

[7] Volatile Systems (2013). (Nov 7, 2013). Volatility Framework. [Online] Available: http://code.google.com/p/Volatility

[8] Michael Hale. (Feb 25, 2013). Android Memory Forensics: Instructions on how access and use the Android support. [Online] Available: https://code.google.com/p/Volatility/wiki/AndroidMemoryForensics

[9] Ismael Valenzuela. (April 23, 2012). Acquiring volatile memory from Android based devices with LiME Forensics, Part I. [Online] Available:

http://blog.opensecurityresearch.com/2012/04/acquiring-volatile-memory-from-Android.html

[10] Lodovico Marziale. Joe Sylve. Andrew Case and Golden G. Richard. Acquisition and analysis of volatile memory from Android devices. [Online] Available: http://digitalforensicssolutions.com/papers/Android-memory-analysis-DI.pdf - "

[11] Mark Loiseau. (July 24, 2012). How to compile the Android Goldfish kernel. [Online] Available: http://blog.markloiseau.com/2012/07/how-to-compile-the-Android-goldfish-emulator-kernel

[12] Joe Sylve. (March 2013) LiME-forensics: LiME "Linux Memory Extractor". [Online] Available: https://code.google.com/p/LiME-forensics/

[13] Monnappa. (July 13, 2013). Advanced Malware Analysis Training Session 1-7 Malware Memory Forensics. [Online] Available: http://securityxploded.com/security-presentations.php

[14] Andrew Hoog. (June 15, 2011). Android Forensics: Investigation, Analysis, and mobile security for google Android. Syngress, E-Book-ISBN-13: 978-1-59749-652-0

[15] Vivek G Gite. (Aug 17, 2001). Linux Shell Scripting Tutorial Ver. 1.0. [Online] available:http://cse.yeditepe.edu.tr/~kserdaroglu/spring2013/cse331/labnotes/WEEK%201%20-%20UNIX%20BASICS/shellnotes.pdf

[16] Simon Leppert. (May 2012). Android memory dump a tech report: a student research paper. Friedrich-Alexander-University Erlangen-Nuremberg. [Online] available:

https://www1.informatik.uni-erlangen.de/filepool/thesis/Android_memory_forensics.pdf

[17] Michael Spreitzenbarth. (March 28, 2013). Dissecting the Droid: Forensic Analysis of Android and its malicious Applications. Friedrich-Alexander-University Erlangen-Nuremberg. [Online] Available: http://opus4.kobv.de/opus4-fau/frontdoor/index/index/docId/3286

[18] Holger Macht. (Jan 2013). Live Memory Forensics on Android with Volatility. Friedrich-Alexander-University Erlangen-Nuremberg. [Online] Available: https://www1.informatik.unierlangen.de/filepool/publications/Live_Memory_Forensics_on_Android_with_Volatility.pdf

[19] Yajin Zhou. Xuxian Jiang. (May 2012). Dissecting Android Malware: Characterization and Evolution: North Carolina state university. [Online] Available: http://www.csc.ncsu.edu/faculty/jiang/pubs/OAKLAND12.pdf

[20] Chung-Huang Yang. Yen-Ting Lai. (Jan 1,2012). Design and Implementation of Forensics Systems for Android Devices based on Cloud Computing: Nationa Kaohsiung Normal University. [Online] Available:

http://www.naturalspublishing.com/files/published/3r315k3w2rxp64.pdf

[21] Kollar, Ivor. (April 8,2010). Forensic RAM dump image analyser: Charles university in Prague. Mater Thesis. [Online] Available: http://hysteria.sk/~niekt0/fmem/doc/foriana.pdf

[22] Memdump. IBM. [Online] Avaialbe: http://publib.boulder.ibm.com/infocenter/tivihelp/v24r1/index.jsp?topic=%2Fcom.ibm.itcamfad.doc_7.1%2FABD001%2Fmsve2%2FIDSource%2Fhelps%2Fitcam_71_msve_help%2FDownloading_ISA.MDDforJ.html

[23] Anderson, David. (2003,2008). White paper: Red Hat Crash Utility. Redhat Software Inc. [Online] Available: http://people.redhat.com/anderson/crash_whitepaper/

[24] Haruyama, Takahiro. ( July 04, 2013). Windows Memory Forensics Analysis using Encase. [Online] Available: http://www.slideshare.net/takahiroharuyama5/takahiro-haruyama-ceic20110515

[25] Juanru Li. Dawu Gu. yuhao Lua. (2012). Android Malware Forensics: Reconstruction of Malicious Events: Dept of Computer Science and Engineering, Shanghai Jiao Tong University. [Online] Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6258204