



Lehrstuhl für Informatik 1  
Friedrich-Alexander-Universität  
Erlangen-Nürnberg



## **MASTER'S THESIS**

# **PRACTICAL INFEASIBILITY OF ANDROID SMARTPHONE LIVE FORENSICS**

**APPLICABILITY CONSTRAINTS OF LIME AND VOLATILITY**

Philipp Wächter, Dipl.-Betriebsw. (DH)

München, April 30, 2015

Examiner: Prof. Dr.-Ing. Felix Freiling  
Advisor: Michael Gruhn, M. Sc.



This thesis is dedicated to my wife and my children. Over the last three years they admirably bore my passion to graduate though being in full-time employment.



## **Eidesstattliche Erklärung / Statutory Declaration**

---

Hiermit versichere ich eidesstattlich, dass die vorliegende Arbeit von mir selbständig, ohne Hilfe Dritter und ausschließlich unter Verwendung der angegebenen Quellen angefertigt wurde. Alle Stellen, die wörtlich oder sinngemäß aus den Quellen entnommen sind, habe ich als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt. Alle eingereichten Versionen der Arbeit sind identisch.

I hereby declare formally that I have developed and written the enclosed thesis entirely by myself and have not used sources or means without declaration in the text. Any thoughts or quotations which were inferred from the sources are marked as such. This thesis was not submitted in the same or a substantially similar version to any other authority to achieve an academic grading. All submitted versions are identical.

---

München, April 30, 2015

---

Philipp Wächter



## **Zusammenfassung**

RAM-Forensik wird immer mehr als wertvolle Erweiterung zur bisherigen Forensik persistenter Speicher erkannt. Nach ersten Erfolgen mit Windows-Systemen, können mittlerweile auch Linux- und MacOS-Systeme gut analysiert werden. Bewährte Tools sind LiME zur Speicher-Akquirierung bei Linux und Android sowie Volatility zur Analyse. Android basiert auf Linux und es gibt bereits Forschungsergebnisse zur entsprechenden Hauptspeicher-Untersuchung. Diese Arbeiten konzentrieren sich überwiegend auf ein virtualisiertes Android.

In dieser Thesis klären wir die praktische Anwendbarkeit in der Strafverfolgung. Dafür haben wir untersucht, inwiefern es praktikabel ist, den Hauptspeicher eines konfiszierten Smartphones mit LiME auszulesen und mit Volatility auszuwerten. Der Vorgang wurde im Allgemeinen aus bewährten Techniken im Zusammenhang mit Windows und Linux hergeleitet. Anschließend wurde er auf Android-Geräte übertragen. Zusätzlich haben wir eine Ansatz erarbeitet, um den zur Verfügung stehenden Zeitrahmen für eine Speicher-Akquise zu ermitteln.

Wir kommen zu dem Ergebnis, dass nur in seltenen Fällen zusätzlich zur regelmäßig angewendeten Tot-Analyse auch eine Live-Analyse angewendet werden kann. Androids Sicherheits-Mechanismen können den erfolgreichen Einsatz von LiME im Rahmen einer Strafverfolgung vereiteln. Die Überwindung dieser Sperren riskiert nicht nur den analysierten Hauptspeicher sondern auch persistenten Speicher. Zudem dürften die in diesem Zusammenhang anzuwendenden Methoden selten gerichtsfest sein.





## **Abstract**

There is an increasing awareness of RAM forensics as a useful enhancement to the forensics of persistent memory. After initial achievements for Windows systems, Linux and Mac OS X can now also be well analyzed. Reliable tools include LiME for Linux and Android memory acquisition and Volatility for analysis. There are already research papers on the corresponding RAM analysis for Linux based Android devices. These papers are mainly directed at virtualized Android devices.

This thesis examines their practical feasibility for law enforcement. We examined to what extent it is feasible to acquire volatile memory from a seized smartphone leveraging LiME and to interpret it with Volatility. The related processes were generally derived from proven techniques developed for Windows and Linux. Afterwards the process was adapted to work with Android devices. Additionally, we developed an approach to evaluate a time frame for the memory acquisition.

We arrive at the conclusion that only in rare cases can a live analysis successfully add value to the regularly performed dead analysis. Android's security features can defeat successful utilization of LiME in the context of criminal proceedings. Overcoming these obstacles endangers not only the targeted RAM but also the persistent memory of the device. Furthermore, the related methods are unlikely to stand up in court.



# Contents

<b>List of Figures</b>	<b>x</b>
<b>List of Tables</b>	<b>xi</b>
<b>List of Listings</b>	<b>xiii</b>
<b>List of Acronyms</b>	<b>xv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Task . . . . .	2
1.3 Related Work . . . . .	2
1.4 Results . . . . .	3
1.5 Outline . . . . .	4
1.6 Acknowledgments . . . . .	5
<b>2 Background</b>	<b>7</b>
2.1 Smartphones . . . . .	7
2.1.1 Smartphone activities . . . . .	7
2.1.2 Smartphone OSs . . . . .	8
2.2 Memory forensics . . . . .	10
2.2.1 Memory acquisition with LiME . . . . .	12
2.2.2 Memory analysis with Volatility . . . . .	15
2.3 Android security features . . . . .	17
2.3.1 Android partitions and boot process . . . . .	17
2.3.2 USB Debugging and ADB . . . . .	20
2.3.3 Rooting a booted Android . . . . .	20
2.3.4 A second look at related work . . . . .	21
<b>3 Virtual machines – Windows, Linux, Android</b>	<b>23</b>
3.1 VirtualBox Windows Guest . . . . .	24
3.2 VirtualBox GNU/Linux Guests . . . . .	25
3.2.1 Debian GNU/Linux . . . . .	26
3.2.2 Fedora GNU/Linux . . . . .	27
3.2.3 Manjaro GNU/Linux . . . . .	30
3.2.4 OpenSUSE GNU/Linux . . . . .	33
3.2.5 Ubuntu GNU/Linux . . . . .	34
3.2.6 Cross compilation . . . . .	34

3.2.7	Volatility timekeeper bug . . . . .	35
3.2.8	Recapitulation . . . . .	37
3.3	Android Emulator . . . . .	39
3.3.1	Compile Goldfish kernel . . . . .	39
3.3.2	Compile LiME and Volatility for Goldfish kernel . . . . .	42
3.3.3	Recapitulation . . . . .	45
3.4	Conclusion . . . . .	46
<b>4</b>	<b>Real machines – Android smartphones</b>	<b>49</b>
4.1	Google Nexus S: Loss of volatile memory . . . . .	51
4.1.1	Unlock boot loader . . . . .	51
4.1.2	Flash factory image . . . . .	52
4.1.3	Flash custom recovery and root device . . . . .	52
4.1.4	Compile LiME and Volatility modules . . . . .	53
4.1.5	RAM acquisition and analysis . . . . .	55
4.2	HTC Magic: Root without loss of volatile memory . . . . .	59
4.3	Sony Xperia Mini Pro – Root and memory dump . . . . .	62
4.3.1	Rooting with Eroot . . . . .	62
4.3.2	Preparing memory acquisition . . . . .	64
4.3.3	Memory acquisition and analysis . . . . .	67
4.4	Sony Xperia Z3: Too new to be targeted . . . . .	68
4.5	Identical twins . . . . .	68
4.5.1	Samsung Galaxy S III / S III LTE . . . . .	69
4.5.2	Samsung Galaxy S III mini(s) . . . . .	70
4.6	Conclusion . . . . .	72
4.6.1	10 steps to Android smartphone RAM analysis . . . . .	72
4.6.2	Virtualized vs. real Android . . . . .	73
<b>5</b>	<b>Memory endurance – Evidence erosion</b>	<b>75</b>
5.1	Memory endurance test with Android Virtual Device (AVD) . . . . .	75
5.2	Memory endurance test with Nexus S . . . . .	78
5.2.1	Preperation . . . . .	79
5.2.2	Results . . . . .	80
5.3	Conclusion . . . . .	81
<b>6</b>	<b>Conclusion and Future Work</b>	<b>85</b>
<b>7</b>	<b>Appendices</b>	<b>87</b>
7.1	Sony Xperia Mini Pro . . . . .	87
7.2	Memory endurance test scripts . . . . .	90
7.3	AVD: Bash script for multiple memory dumps over time . . . . .	90
7.4	Nexus S: Bash script for multiple memory dumps over time . . . . .	93
7.5	Bash script for merging process lists of multiple memory dumps . . . . .	95
7.6	Memory endurance test results . . . . .	96
	<b>References</b>	<b>101</b>

# List of Figures

1.1	Consoles with corresponding OS logos . . . . .	5
2.1	IDC: Worldwide Smartphone OS Market Share, Q4 2014 . . . . .	9
2.2	IDC: Worldwide Smartphone Vendor Market Share, Q4 2014 . . . . .	10
2.3	LiME: Memory segment header definition . . . . .	13
2.4	LiME: Memory segment header in dump . . . . .	13
2.5	Goldfish memory dump: Volatility plugin “limeinfo” . . . . .	13
2.6	Nexus S: /proc/iomem . . . . .	14
2.7	Nexus S: LiME dump contains just two out of three memory ranges . . .	14
2.8	Twitter: Specific Volatility profiles for every variant of the Linux kernel .	16
2.9	Android boot process . . . . .	18
3.1	Windows: Moonsols DumpIt output . . . . .	24
3.2	Windows: Volatility output . . . . .	25
3.3	Debian: Compile LiME . . . . .	27
3.4	Debian: Create Volatility profile . . . . .	27
3.5	Debian: Initiate memory acquisition . . . . .	27
3.6	Debian: List processes with Volatility . . . . .	28
3.7	Fedora: Install dependencies . . . . .	29
3.8	Fedora: Compile LiME and create Volatility profile . . . . .	29
3.9	Fedora: Initiate memory acquisition . . . . .	29
3.10	Fedora: List processes with Volatility fails . . . . .	30
3.11	Manjaro: Install dependencies . . . . .	31
3.12	Manjaro: Compile LiME and create Volatility profile . . . . .	31
3.13	Manjaro: Initiate memory acquisition . . . . .	31
3.14	Manjaro: List processes with Volatility . . . . .	32
3.15	OpenSUSE: Install dependencies . . . . .	33
3.16	OpenSUSE: Compile LiME and create Volatility profile . . . . .	33
3.17	OpenSUSE: Initiate memory acquisition . . . . .	34
3.18	OpenSUSE: Open port 4444 in SUSE Firewall . . . . .	34
3.19	Cross compile LiME and Volatility for Fedora 21 . . . . .	36
3.20	Volatility fix in tools/linux/module.c . . . . .	37
3.21	Create Android Virtual Device (AVD) . . . . .	40
3.22	AVD: Compile Goldfish kernel (1/3) . . . . .	40
3.23	AVD: Compile Goldfish kernel (2/3); make menuconfig . . . . .	41
3.24	AVD: Compile Goldfish kernel (3/3) . . . . .	42
3.25	AVD: Compile LiME and create Volatility profile . . . . .	44

3.26	AVD: LiME TCP transfer . . . . .	44
3.27	AVD: Start Volatility . . . . .	45
4.1	Google Nexus S . . . . .	50
4.2	Nexus S: build.prop . . . . .	51
4.3	Nexus S: Unlock boot loader . . . . .	52
4.4	Nexus S: Flash factory image . . . . .	52
4.5	Nexus S: Unlock boot loader; install TWRP2 and SuperSU . . . . .	53
4.6	Nexus S: Screens while unlocking and sideloading SuperSU . . . . .	54
4.7	Nexus S: /proc/version . . . . .	55
4.8	Nexus S: Downloads for kernel compilation . . . . .	56
4.9	Nexus S: Compile LiME and create Volatility profile . . . . .	57
4.10	Nexus S: LiME TCP transfer . . . . .	58
4.11	Nexus S: List processes with Volatility . . . . .	58
4.12	HTC Magic . . . . .	59
4.13	HTC Magic: build.prop and /proc/version . . . . .	60
4.14	HTC Magic: Gain temporary root permissions . . . . .	61
4.15	Sony Xperia Mini Pro: build.prop and /proc/version . . . . .	63
4.16	Sony Xperia Mini Pro: Eroot screens . . . . .	64
4.17	Sony Xperia Mini Pro: root shell . . . . .	65
4.18	Sony Xperia Mini Pro: Superuser installed by Eroot . . . . .	65
4.19	Sony Xperia Mini Pro: Kernel compilation . . . . .	66
4.20	Sony Xperia Mini Pro: List processes with Volatility fails . . . . .	66
4.21	Sony Xperia Mini Pro: Check memory dump segments . . . . .	67
4.22	Samsung S III and S III LTE: front and back . . . . .	69
4.23	Samsung S III and S III LTE: labels under battery . . . . .	70
4.24	Samsung S III mini (GT-I8190 and GT-I8200N): front and back . . . . .	71
4.25	Samsung S III mini (GT-I8190 and GT-I8200N): labels under battery . . . . .	71
5.1	Change of Goldfish memory over 24 hours (view 1) . . . . .	76
5.2	Change of Goldfish memory over 24 hours (view 2) . . . . .	76
5.3	Change of Goldfish memory over 24 hours (processes) . . . . .	77
5.4	Nexus S: Install “adb Insecure” . . . . .	79
5.5	Change of Nexus S memory over 24 hours (view 1) . . . . .	80
5.6	Change of Nexus S memory over 24 hours (view 2) . . . . .	80
5.7	Change of Nexus S memory over 24 hours (upper 75 processes) . . . . .	82
5.8	Change of Nexus S memory over 24 hours (lower 32 processes) . . . . .	83
7.1	Sony Xperia Mini Pro: Compile LiME and create Volatility profile . . . . .	88
7.2	Sony Xperia Mini Pro: LiME TCP transfer . . . . .	88

# List of Tables

2.1	Examples of current smartphone Operation Systems (OSs) . . . . .	9
2.2	Examples of past smartphone OSs . . . . .	9
2.3	Volatility 2.4 Windows profiles . . . . .	16
3.1	Differences noticed during GNU/Linux experiments . . . . .	38
3.2	Comparison of the Virtual Machine (VM) experiments . . . . .	46
4.1	Tested Android devices . . . . .	50
4.2	Nexus S: Figuring out which kernel to build . . . . .	55
4.3	HTC Magic: Available kernel sources . . . . .	60
4.4	HTC Dream: Available kernel sources . . . . .	60
4.5	Samsung Galaxy S III models . . . . .	69
4.6	Samsung Galaxy S III mini models . . . . .	70
4.7	Comparison of the VM and real phone experiments . . . . .	73
5.1	Nexus S: Occurrence of processes . . . . .	81
7.1	Memory endurance test / AVD / 4 hour pass / reference is first dump . .	96
7.2	Memory endurance test / AVD / 1 hour pass / reference is first dump . .	96
7.3	Memory endurance test / AVD / 15 minutes pass / reference is first dump	97
7.4	Memory endurance test / AVD / 4 hour pass / reference is previous dump	98
7.5	Memory endurance test / AVD / 1 hour pass / reference is previous dump	98
7.6	Memory endurance test / AVD / 15 minutes pass / reference is previous dump . . . . .	99





# List of Listings

3.1	AVD/Makefile.LiME.cross . . . . .	43
3.2	AVD/Makefile.Volatility.cross . . . . .	43
4.1	NexusS/Makefile.Volatility.cross . . . . .	56
4.2	NexusS/Makefile.LiME.cross . . . . .	57
7.1	SonyXperiaMiniPro/Makefile.Volatility.cross . . . . .	87
7.2	SonyXperiaMiniPro/Makefile.LiME.cross . . . . .	89
7.3	multidump-goldfish.sh . . . . .	90
7.4	multidump-nexuss.sh . . . . .	93
7.5	multidump-goldfish-pslist.sh . . . . .	95



# List of Acronyms

<b>ADB</b>	Android Debug Bridge
<b>AFoD</b>	“A Fistful of Dongles”
<b>ASIC</b>	Application-specific integrated circuit
<b>AOSP</b>	Android Open Source Project
<b>APT</b>	Advanced Packaging Tool
<b>ARM</b>	Advanced RISC Machines
<b>AV</b>	Anti Virus
<b>AVD</b>	Android Virtual Device
<b>bash</b>	Bourne-again shell
<b>cpio</b>	copy in and out
<b>CPU</b>	Central Processing Unit
<b>DDMS</b>	Dalvik Debug Monitor Server
<b>DKMS</b>	Dynamic Kernel Module Support
<b>DLL</b>	Dynamic-link library
<b>DMA</b>	Direct Memory Access
<b>DMD</b>	Droid Memory Dumper
<b>DVM</b>	Dalvik Virtual Machine
<b>FROST</b>	Forensic Recovery Of Scrambled Telephones
<b>gcc</b>	The GNU Compiler Collection
<b>GNU</b>	GNU’s Not Unix
<b>GUI</b>	graphical user interface
<b>HIPS</b>	Host-based Intrusion Prevention System
<b>I/O</b>	input/output
<b>ISO</b>	International Organization for Standardization
<b>ISO image</b>	disk image using ISO 9660 file system
<b>LAN</b>	local area network

<b>LiME</b>	Linux Memory Extractor
<b>LKM</b>	loadable kernel module
<b>LNK</b>	Microsoft Windows filename extension for shortcuts to local files
<b>LTE</b>	Long Term Evolution
<b>MAC</b>	media access control
<b>MSC</b>	Mass storage mode
<b>NFC</b>	Near Field Communication
<b>PC</b>	personal computer
<b>PCI</b>	Peripheral Component Interconnect
<b>PCIe</b>	PCI Express
<b>PIN</b>	personal identification number
<b>OEM</b>	Original Equipment Manufacturer
<b>OS</b>	Operation System
<b>OSS</b>	Open-Source Software
<b>OTA</b>	over the air
<b>QEMU</b>	Quick Emulator
<b>RAM</b>	Random Access Memory
<b>RISC</b>	Reduced instruction set computing
<b>ROM</b>	Read-Only Memory
<b>RPM</b>	RPM Package Manager (originally Red Hat Package Manager)
<b>SDK</b>	Software Development Kit
<b>SoC</b>	System on a Chip
<b>TCP</b>	Transmission Control Protocol
<b>TWRP</b>	Team Win Recovery Project
<b>UAC</b>	User Account Control
<b>UI</b>	user interface
<b>URI</b>	Uniform Resource Identifier
<b>URL</b>	Uniform Resource Locator
<b>USB</b>	Universal Serial Bus
<b>VM</b>	Virtual Machine
<b>WLAN</b>	wireless LAN
<b>zip</b>	Archive file format; onomatopoeia for “move at high speed”

# INTRODUCTION

---

*“Forensic practitioners are well acquainted with push-button forensics software. They are an essential tool to keep on top of high case loads – plug in the device and it pulls out the data. Gaining access to that data is a constant challenge against sophisticated protection being built into modern smartphones. Combined with the diversity of firmware and hardware on the Android platform it is not uncommon to require some manual methods and advanced tools to get the data you need.”*

Thomas Cannon, Director of Research and Development, NowSecure

## 1.1 Motivation

Mobile phone forensics is daily routine for law enforcement. There are several plug-and-play solutions on the market for the criminal prosecutor. Promotion promises—to phrase it a bit exaggerated—you just connect the phone, press a button and all the data stored on the device is getting extracted and prepared report-ready. But, this is at best valid for non-volatile data.

For some very good reasons (cmp. section 2.2) live memory forensics in general is raising in popularity. This is even though this kind of investigation presumes advanced technical skills. With this trend come an increasing number of scientific papers about smartphone

live forensics (cmp. section 1.3), particularly regarding Android. It is sound to assume that an Android smartphone's volatile memory can be examined in principle. Also it is indisputable that such an examination can deliver quite valuable findings. But: Is Android live forensics always feasible or even advisable?

## 1.2 Task

We will discuss if Android live memory forensics can be solved by practitioners who have to keep on top of high case loads. What is needed to acquire the volatile memory? How can the memory dump be analyzed? What puts obstacles into the examiner's path? Are there reasons which make Android live memory forensics impossible? Memory acquisition is done using the Linux Memory Extractor (LiME) module (introduced in section 2.2.1). The Volatility framework is used for the memory dumps' analysis (introduced in section 2.2.2).

## 1.3 Related Work

Due to the novelty of Android live memory forensics there is a manageable amount of related academic work.

Thing et al. [71] developed both their own memory acquisition tool *memgrab* and a memory dump analyzer *MDA* for the Android platform. The former connects to an Android phone and retrieves a copy of the volatile memory, while the latter decodes and extracts information from the retrieved data. The used hardware is specified as “*an Android mobile phone, the Google development set*”.

In 2011 Sylve Sylve et al. [64], Sylve [65] developed *Droid Memory Dumper (DMD)*, “*the first methodology and toolset for acquiring full physical memory images from Android devices*” which today is known as Linux Memory Extractor (LiME).

Ali-Gombe [2] described in 2012 how to extract messages sent and received by the Motorola Motoblur application. Though LiME was known to the author, *memfetch* [79] was used to only extract the application's memory instead of the full device's memory.

Müller and Spreitzenbarth [47] proved 2012 the cold-boot-attack applicable to Android smartphones. A cold-boot-attack sidelines any lock screen (PIN, password, etc.) by booting into a minimal custom OS while at the same time preserving most of the former RAM. They were able to acquire the full device's volatile memory of a Galaxy Nexus independent of the boot loader being locked or unlocked. The downside of their method is the loss of the non-volatile memory in case of a locked boot loader.

In 2013 Macht [44] presented extensions to the Volatility framework with focus on the Dalvik Virtual Machine (DVM) and specific popular Android applications.

Apostolopoulos et al. [10] discovered “*authentication credentials in volatile memory of Android mobile devices*”. They worked with the Dalvik Debug Monitor Server (DDMS) on

emulators and phones without mentioning if or how the latter were prepared. Later in [78] they described using LiME.

Stirparo et al. [61] presented 2013 their work on extracting data like online banking credentials out of Android memory. They worked completely on Android Virtual Devices (AVDs).

The currently most important publication about memory forensics is “*The Art of Memory Forensics*” [42] from 2014. This book covers the OSs Windows, Linux and MacOS. Though the string “Android” appears only four times in this over 800 pages work, quite a lot of the Linux part is transferable.

In 2014 Sun et al. [63] presented a new approach to memory acquisition. They refer to the “TrustZone” [3, 11] which ARM already introduced 2004 with the ARMv7 architecture. The ARMv7 architecture can be found in many mobile devices, e.g. in Apple (A4, A5, A6), nVidia (Tegra), Qualcomm (Snapdragon), Samsung (Exynos) and Texas Instruments (OMAP) processors. The TrustZone is a system level isolation solution, which divides the mobile platform into two isolated execution environments, called normal domain (e.g. with the Android OS) and secure domain. Sun et al. developed “*a TrustZone-based reliable memory acquisition mechanism called TrustDump, which is capable of obtaining the RAM memory and CPU registers*” of the OS in the normal domain. For that purpose a memory acquisition module called TrustDumper is installed in the secure domain.

We will have a further look at these sources in section 2.3.4.

## 1.4 Results

Our experiments and observations made us arrive at the conclusion that in most cases Android smartphone live memory forensics is not feasible for law enforcement purposes. There are too many factors that make memory acquisition and/or analysis uncertain or even impossible and could yet endanger the non-volatile memory. We identified these disturbing factors:

- |   |                          |  |
|---|--------------------------|--|
| 1 | Identify model:          | A smartphone model can not always be identified solely by visual inspection. (cmp. section 4.5)  |
| 2 | Identify OS:             | Without unlocked screen or enabled USB debugging there is no way to identify the OS version. This information is indispensable for attacks like FROST. (cmp. section 2.3.3)                                      |
| 3 | Root exploit:            | The availability of an appropriate “reboot-less” root exploit at exactly the right time is anything but reliable. (cmp. section 2.3.3)   |
| 4 | Lock screen:             | USB debugging is needed in some attack scenarios. With enabled lock screen there is no way to enable it without deleting the volatile memory contents. (cmp. section 2.3.2)                                      |
| 5 | Availability of sources: | Theoretically the kernel sources should be available. But this is not always the case. (cmp. section 4.2)  |
| 6 | Kernel configuration:    | If the OS version is known and official sources as well as a kernel configuration file are available, there is still no guarantee, that the actual kernel was compiled the official way. (cmp. section 3.3.1)    |
| 7 | Evidence erosion:        | The fact that the RAM is permanently changing, results in a steadily increasing loss of evidence over time. This prohibits long lasting investigations about how to overcome security barriers. (cmp. chapter 5) |

Even if the conditions were appropriate, there is no one-click solution. Instead time pressure calls for a highly specialized forensics expert who can do the job with all tools at hand and without the need to gather more information.

It is important to repeat that the scope of this work is the practical feasibility of Android live memory forensics in criminal investigations. Our negative conclusion focuses on exactly this scope. It does not apply for live memory forensics of

- desktop and server systems or rather their OSs (in particular Windows, Linux, MacOS) in the field of criminal investigations, or
- groomed Android systems in the field of malware analysis or otherwise academic research.

For these use cases we rate live memory forensics as invaluable.

## 1.5 Outline

Following this introduction, in chapter 2 we go into some key terms of this thesis. We expose our perception of *smartphones* and why we should deal with *memory forensics* in general. The basic thesis’ tools *LiME* and the *Volatility* framework are depicted as well as *Android security features* that give the forensics expert a hard time.



In chapter 3 we describe memory acquisition and analysis on the basis of VMs. We start with Windows 7, discuss the differences of five GNU/Linux distributions and close with an Android Virtual Device.

Based on the experiences with the virtual machines we address real devices in chapter 4. We begin with a groomed scenario and then elaborate on several obstacles we faced while experimenting with miscellaneous hardware.

The time consuming work so far led to the question if there is a time limit after that the target's memory has changed that much that acquisition and analysis became meaningless. In chapter 5 we try to approach an answer with memory endurance tests.

Our conclusions and proposals are summarized in chapter 6.

This work includes a lot of code examples. We used predominantly a GNU/Linux bash.<sup>1</sup> From GNU/Linux bash sometimes there will be a switch to an Android shell and there are also examples with Windows command line. In order to visualize the current environment the logo of the corresponding OS is shown in each case (cmp. fig. 1.1).



**Figure 1.1:** Console representations are marked with corresponding OS logos in order to clarify the current OS context.

## 1.6 Acknowledgments

We have to thank Joe Sylve, Andrew Case, Holger Macht, and many other for patiently answering our newcomer questions.

We are proud that the Volatility project allowed us to contribute our timekeeper patch (cmp. section 3.2.7).

---

<sup>1</sup>For reference—if the reader might not be familiar with a GNU/Linux command or its arguments—we recommend [explainshell.com](http://explainshell.com) [39] for first orientation.



## BACKGROUND

---

In this chapter we enlarge upon the basic terms employed in this work. We illustrate the focus on *Android smartphones* and explain the rising significance of *memory forensics* in general. In addition, the essential software tools *LiME* and *Volatility* are introduced. Finally we offer a résumé of those *Android security features* we address mainly in chapter 4.

### 2.1 Smartphones

Our investigative targets are Android smartphones. *Smartphones* on the one hand because of the interwovenness of these little devices with the life of so many people. *Android* on the other hand because of its mobile OS market share.

#### 2.1.1 Smartphone activities

Smartphones are an integral part of many people's lives. A representative survey by Bitkom Research [53] says that in February 2015 63% of the Germans age 14 and above used smartphones. 58% of them used the alarm function to start into the day which 83% managed with assistance of the smartphone's calendar. 71% communicated via short messages and 70% used social networks.

Pew Research Center [52] say that 7% of U.S. American adults “do not have broadband access at home, and also have relatively few options for getting online other than their cell phone”. For younger adults at ages 18–29 this rises to 15%.

Google’s *Our Mobile Planet* website allows a little insight in their treasure of data. For 2013<sup>1</sup> they show a similar picture for many countries. (See [29] for an example chart we created based on Google’s data.)

A distressing but striking indicator for the pervasiveness of mobile phones in the lives of many people is the role they play in car crashes. The AAA Foundation for Traffic Safety researched police-reported motor vehicle crashes from 963,000 drivers aged 16-19 in 2013 [23]. The driver was engaged in cell phone use in 12% of all crashes. Filmmaker and documentarian Werner Herzog in 2013 directed a short film on the dangers of texting and driving. “From One Second to the Next” [36] looks at how four lives were impacted by texting-related accidents. Matt Ritel, a reporter for The New York Times, won a Pulitzer Prize in 2010 for his series of articles about distracted driving. In 2014 he wrote “A Deadly Wandering” [54] about one crash from Werner Herzog’s film with two people dead.

Smartphones even disclose information about their users without the need of forensic investigation. Android devices for instance constantly search for known wireless networks using their original MAC address.<sup>2</sup> This allows for the transparent generation of movement profiles and is already in use by marketing companies.

Last but not least there is an upcoming trend to control *smart* homes with *smart* phones. Hence access to such a smartphone can imply virtual and practical admittance to the owner’s home.

### 2.1.2 Smartphone OSs

There are many OSs in the smartphone cosmos. Google’s Android and Apple’s iOS are dominant. Others try to gain their market share (cmp. table 2.1 and fig. 2.1). Even mobile phones with a discontinued OS (cmp. table 2.2) continue to be used. The author for example still uses phones and tablet computers with 2012 cancelled WebOS. Last but not least the building of a smartphone is no longer the sole domain of big companies as do-it-yourself projects now prove [35, 60].

In the mobile device market Android plays the prominent role. In 2014 over one billion Android devices were sold according to Gartner [27] and IDC [37].

Android is prevalent as operating system for smart phones, tablets, wearables, car infotainment systems etc. There is a wide range of devices for low prices. The OS is unrivaled

---

<sup>1</sup>the most current data available at Google’s *Our Mobile Planet* as of this writing

<sup>2</sup>Apple’s iOS since version 8 uses random MAC addresses for WLAN scans. For rooted Android devices additional software like Chainfire’s *Pry-Fi* [14] is available.

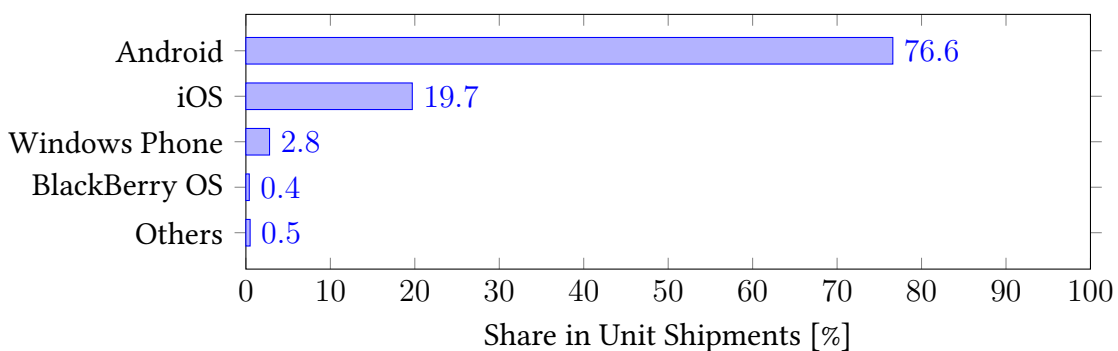
<sup>3</sup>Samsung’s 2013 smart camera NX300M was the first consumer product based on Tizen. But the first smartphone with Tizen, the Samsung Z1, was released in January 2015.

**Table 2.1:** Examples of current smartphone OSs

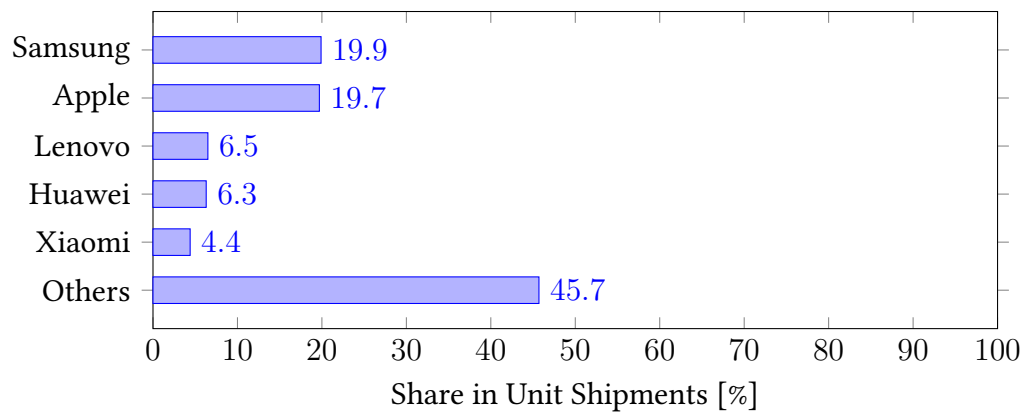
OS	by	since	examples for devices
BlackBerry OS	BlackBerry	1999	BlackBerry devices
iOS	Apple	2007	iPhone, iPod Touch, iPad, iPad Mini, Apple TV
Android	Google	2008	first device was HTC Dream; countless followed
Windows Phone	Microsoft	2010	Nokia Lumia devices
Firefox OS	Mozilla	2013	Alcatel OneTouch Fire E, Geeksphone Revolution, ZTE Open C
Sailfish OS	Jolla	2013	Jolla Phone
Ubuntu Touch	Canonical	2014	BQ Aquaris E4.5 Ubuntu Edition
Tizen	mainly Intel and Samsung	2015 <sup>3</sup>	Samsung Z1 (India)

**Table 2.2:** Examples of past smartphone OSs

OS	by	from	to
Palm OS	Palm	1996	2004
Symbian	Nokia	2002	2011
Windows Mobile	Microsoft	2003	2010
WebOS	Palm / HP	2009	2012
Bada OS	Samsung	2010	2013

**Figure 2.1:** IDC: Worldwide Smartphone OS Market Share, Q4 2014 [37]

in letting manufacturers, developers and consumers put their own ideas into practice while offering an uniform platform for all of them. It is usually not much work to gain root permissions.<sup>4</sup> For quite a lot of devices the pre-installed Android version can be replaced by a customized ROM. This makes Android both the most chaotic and the most dynamic system compared to competitors like Apple’s iOS or Microsoft’s Windows Phone.



**Figure 2.2:** IDC: Worldwide Smartphone Vendor Market Share, Q4 2014 [38]

Biggest vendors in the smartphone market are Samsung and Apple (see fig. 2.2). In the Android market however, Samsung dominated in 2014.

## 2.2 Memory forensics

In an interview for Eric Huber’s blog “A Fistful of Dongles” (AFoD) [33]<sup>5</sup> Andrew Case—a core developer of Volatility—summarizes his view of Memory Forensics:

*“Memory forensics is the examination of physical memory (RAM) to support digital forensics, incident response, and malware analysis. It is [sic] has the advantage over other types of forensics, such as network and disk, in that much of the system state relevant to investigations only appears in memory. This can include artifacts such as running processes, active network connections, and loaded kernel drivers. There are also artifacts related to the use of applications (chat, email, browsers, command shells, etc.) that only appear in memory and are lost when the system is powered down.” [33]*

An example for the latter is given by Lenny Zeltser who interacted with a scammer posing as a help desk technician [80]. This person accessed websites leveraging the Microsoft

---

<sup>4</sup>But it is problematic to gain root privileges without losing the current RAM content. This is discussed repeatedly in this thesis.

<sup>5</sup>As the blog is no longer active the citation refers to an archived version on [web.archive.org](http://web.archive.org).

Windows HTML Help application `%windir%\hh.exe` instead of a common web browser. Zeltser afterwards analyzed the incident. Only memory forensic methods were able to disclose the URLs.

*“Furthermore, attackers are well aware that many investigators still do not perform memory forensics and that most AV/HIPS systems don’t thoroughly look in memory (if at all). This has led to development of malware, exploits, and attack toolkits that operate solely in memory. Obviously these will be completely missed if memory is not examined. Memory forensics is also being heavily pushed due to its resilience to malware that can easily fool live tools on the system, but have a much harder time hiding within all of RAM.” [33]*

Schulz [56] describes how an Android application can dynamically load encrypted code during runtime and have it decrypted during execution only. In this case the analyst’s only chance to access meaningful code is memory analysis.

*“Besides the aforementioned items, memory forensics is also becoming heavily used due to its ability to support efficient triage at scale and the short time in which analysis can begin once indicators have been found. Traditional triage required reading potentially hundreds of MBs of data across disk looking for indicators in event logs, the registry, program files, LNK files, etc. This could become too time consuming with even a handful of machines, much less hundreds or thousands across an enterprise. On other hand, memory-based indicators, such as the names of processes, DLLs, services, and kernel drivers, can be checked by only querying a few MBs of memory. Tools, such as F-Response, makes [sic] this fairly trivial to accomplish across huge enterprise environments and also allow for full acquisition of memory if indicators are found on a particular system.*

*The last reason I will discuss related to the explosive growth of the use of memory forensics is the ability to recover encryption keys and plaintext versions of encrypted files. Whenever software encryption is used, the keying material must be stored in volatile memory in order to support decryption and encryption operations. Through recovery of the encryption key and/or password, the entire store (disk, container, etc.) can be opened. This has been successfully used many times against products such as TrueCrypt, Apple’s Keychain, and other popularly used encryption products. Furthermore, as files and data from those stores are read into memory they are decrypted so that the requesting application (Word, Adobe, Notepad) can allow for viewing and editing by the end user. Through recovery of these file caches, the decrypted versions of files can be directly reconstructed from memory.” [33]*

During the following chapter 3 we briefly deal with Windows and Linux before we draw the connection to Android memory forensics.

### 2.2.1 Memory acquisition with LiME

As of this writing Joe Sylve's *LiME* [1, 64, 65] is the most accurate Open-Source Software (OSS) memory acquisition tool available for systems with a Linux kernel. That is, amongst others, because it is implemented as a loadable kernel module (LKM), which is a convenient way to extend the Linux kernel without the need to completely recompile it. Using this technique means that no context switches between userland and the kernel are required to transfer data. [42, pg. 580 et seqq.]

A drawback however is that the examiner usually has to compile the external kernel module using header files<sup>6</sup>. That is because kernel modules have to be compatible with the kernel they are loaded into. Therefore Linux memory acquisition with LiME is somehow less convenient than the comparable task at a Windows system. We depict both in the following chapter 3.

Forensic principles forbid compiling any software on a computer which is subject to the investigation. Compilation would significantly change the system by installing persistent and temporary files not to mention the modifications to the system's RAM. Thus, a second system with the same kernel version and configuration should be set up. Alternatively one can cross-compile the module. As discussed later in connection with Android (see chapter 4) there are situations where the more complex requirements of cross-compiling can't be avoided.

On Linux systems the file `/proc/iomem` shows the current map of the system's memory for each physical device (see fig. 2.6 for an exemplary excerpt). LiME basically copies those RAM segments listed in `/proc/iomem` which are named "System RAM". The user can choose from three different file formats to store memory dumps:

1. raw: All "System RAM" ranges simply get concatenated.
2. padded: Starting from physical address 0 all non-"System RAM" ranges get padded with zeros.
3. lime: Like "raw" but the memory segments are prepended with a 32 bytes header containing address space information.

The latter is our format of choice. This is because Volatility contains an address space definition for the "lime" format. Furthermore this format is useful for debugging. The LiME memory segment header definition is shown in fig. 2.3 and a commented hex dump in fig. 2.4.

If the LiME memory dump was successful and if there is a working Volatility profile available (cmp. section 2.2.2), then the Volatility plugin `limeinfo` prints the enclosed memory segments. An example taken from the experiment in section 3.3 is shown in fig. 2.5.

---

<sup>6</sup>On GNU/Linux systems header files often are located in a version specific directory under `/lib/modules`. Android devices usually come without their sources.



```
typedef struct {
    unsigned int magic;
    unsigned int version;
    unsigned long long s_addr;
    unsigned long long e_addr;
    unsigned char reserved[8];
} __attribute__((packed)) lime_mem_range_header;
```

Figure 2.3: LiME: Memory segment header definition

```
$ xxd -c 8 -l 32 goldfish.lime
00000000: 454d 694c 0100 0000  EMiL....
00000008: 0000 0000 0000 0000  ....
00000010: ffff ff1f 0000 0000  ....
00000018: 0000 0000 0000 0000  ....
```

Memory segment header (little-endian):

- magic `LiME` & version `1`
- start address `0x 0000 0000`
- end address `0x 1fff ffff` ⇒ 512MiB
- the last 16 bytes are reserved

Figure 2.4: LiME: Memory segment header in dump

```
$ export VOLATILITY_LOCATION=file:///tmp/dump/goldfish.AVD00900.047.lime
$ export VOLATILITY_PROFILE=LinuxAndroid_Goldfish_3_4_67-01413-g9ac497fARM
$ python vol.py limeinfo
Volatility Foundation Volatility Framework 2.4
Memory Start Memory End Size
-----
0x0000000000 0x1fffffffff 0x200000000
```

Figure 2.5: Goldfish memory dump: Volatility plugin “limeinfo”

When we did our first experiments with acquiring Android memory dumps they all failed. We used the option to write the dump to the targets file system. The lime format's memory segment headers helped during debugging as they showed that the first from three expected segments was always missing. This was not the case though when selecting the option to transfer the dump over TCP. Together with Joe Sylve we ascertained that LiME's default behavior to attempt to use Direct I/O caused the error [74]. Joe Sylve hereupon changed the default to not attempting Direct I/O.

```
shell@android:/ $ cat /proc/iomem | grep -i ram
30000000-323ffffff : System RAM
35000000-35ffffff : onedram
40000000-4b7aefff : System RAM
50000000-57ffffff : System RAM
57f00000-57ffffff : ram_console
```

**Figure 2.6:** Nexus S: `/proc/iomem` lists three `System RAM` memory ranges for LiME to acquire

We now briefly depict the debug process. Based on `/proc/iomem` (cmp. fig. 2.6) we expected to find three segments in the LiME dump:

1. 30000000-323ffffff
2. 40000000-4b7aefff
3. 50000000-57ffffff

```
$ # >>> start of dump = memory segment header of second expected segment
$ xxd -l 0x20 ~/android/dump/NexusS_4.0.4.dump
00000000: 454d 694c 0100 0000 0000 0040 0000 0000 EMiL.....@....
00000100: ffe7 7a4b 0000 0000 0000 0000 0000 0000 ..zK.....
$ # >>> next header belongs to third expected segment
$ xxd -s $((0x20 + 0x4b7aefff - 0x40000000 + 1)) -l 0x20
~/android/dump/NexusS_4.0.4.dump
b7af020: 454d 694c 0100 0000 0000 0050 0000 0000 EMiL.....P....
b7af030: ffff ef57 0000 0000 0000 0000 0000 0000 ...W.....
$ # >>> total size of these two segments incl. headers ...
$ echo $(( 0x20 + 0xb7aefff + 1 + 0x20 + 0x7ffffff + 1 ))
325775424
$ # >>> ... equals total size of dump file
$ stat -c %s ~/android/dump/NexusS_4.0.4.dump
325775424
```

**Figure 2.7:** Nexus S: LiME dump contains just two out of three memory ranges

Each memory segment's header has a fixed length of 32 bytes (`0x20`). Within each header the corresponding segment's start and end addresses are given. With this knowledge we

can walk through the dump from one segment to another (cmp. fig. 2.7). By doing so we only spot the second and third segment. The first is clearly missing.

## 2.2.2 Memory analysis with Volatility

Andrew Case provides a résumé of Volatility’s history and philosophy in his interview for Eric Huber’s blog AFoD [33]:

*“The Volatility Project was started in the mid-2000s by Aaron Walters and Nick Petroni. Volatility emerged from two earlier projects by Nick and Aaron, Volatools and The FATkit. These were some of the first public projects to integrate memory forensics into the digital investigation process. Volatility was created as the open source version of these research efforts and was initially worked on by Aaron and Brendan Dolan-Gavitt. Since then, Volatility has been contributed to by a number of people, and has become one of the most popular and widely used tools within the digital forensics, incident response, and malware analysis communities.*

*Volatility was designed to allow researchers to easily integrate their work into a standard framework and to feed off each other’s progress. All analysis is done through plugins and the core of the framework was designed to support a wide variety of capture formats and hardware architectures. As of the 2.4 release (summer 2014), Volatility has support for analyzing memory captures from 32 and 64-bit Windows XP through 8, including the server versions, Linux 2.6.11 (circa 2005) to 3.16, all Android versions, and Mac [OS X] Leopard through Mavericks.”*

One aspect of Volatility’s modularity is the *profiles* concept. “Profiles in Volatility allow for plugins to be written generically while the backend code handles all the changes between different versions of an operating system (e.g. Windows XP, Vista, 7, and 8).” [33] For each combination of a specific OS version and hardware architecture (x86, x64, ARM) a separate profile is needed. Any profile includes [42, pg. 55]:

- A collection of the VTypes, overlays, and object classes
- Metadata such as the operating system’s name, the kernel version, and build number
- Indexes and names of system calls
- Global variables that can be found at hard-coded addresses in some operating systems
- Low-level types for native languages (usually C), including the sizes for integers, long integers, and so on
- System map: Addresses of critical global variables and functions (Linux and OS X only)

By default Volatility (as of this writing) comes with a set of 31 Windows profiles (cmp. table 2.3). Additionally Windows 10 beta support is available from the project’s GitHub repository [73].

**Table 2.3:** Volatility 2.4 Windows profiles

OS	Architecture	Service pack			
		/	SP1	SP2	SP3
Windows XP	x86			●	●
	x64		●	●	
Windows Vista	x86	●	●	●	
	x64	●	●	●	
Windows 7	x86	●	●		
	x64	●	●		
Windows 8	x86	●	●		
	x64	●			
Windows 8.1	x64		●		
Windows 2003	x86	●	●	●	
	x64		●	●	
Windows 2008	x86		●	●	
	x64		●	●	
Windows 2008 R2	x64	●	●		
Windows Server 2012	x64	●			
Windows Server 2012 R2	x64	●			

**Figure 2.8:** Twitter conversation about the need for specific Volatility profiles for every variant of the Linux kernel.

By Brian Keefer (@chort0) and Andrew Case (@attrc).



According to Ligh et al. [42, pg. 583] Volatility supports Linux kernel versions from 2.6.11 to 3.14 which adds up to at least 40 base kernels and 500 sub-versions plus countless user compiled kernels. Such a huge amount of variation makes it infeasible to provide all prospective Linux profiles. Thus each time an examiner analyzes a new Linux system they concurrently have to create a new individual profile. A tiny exception to this rule is a small—but intended to grow—collection of profiles for standard GNU/Linux installations provided by the Volatility Foundation [72].

In general, though onerous, a Linux profile can be generated without great difficulty. We run through some examples in section 3.2. What is unproblematic for Linux becomes full

of stumbling blocks for Android. We discuss these obstacles in chapter 4.

Creating a Volatility Linux profile means generating a set of VTypes (structure definitions) and a `System.map` file for a particular kernel version and packing those together into one .zip file.

The VTypes can be extracted from the compiled Linux kernel file `vmlinux` if available. Else Volatility’s kernel module `tools/linux/module.c` has to be compiled against the kernel to be analyzed. `module.c` declares members of all the types needed. Its compilation adds the type definitions into the module’s debugging information. However `dwarfdump`—a tool that parses the debugging information from ELF files, such as the Linux kernel and kernel modules—extracts the structure definitions the way Volatility needs them.

To locate static data structures, the `System.map` file is needed because it contains the name and address of every static data structure used in the kernel. The kernel build process creates the file by using `nm` on the compiled `vmlinux` file. If a GNU/Linux distribution comes with a `System.map`—and many do<sup>7</sup>—it should be found in the `/boot` folder. Alternatively one can again extract it with `nm` out of the `vmlinux` file if available. (This is demonstrated in section 3.2.3.) Otherwise the target kernel’s compilation has to be reproduced which produces, amongst other files, the kernel `vmlinux` and the `System.map`. The latter is the standard procedure for Android (cmp. chapter 4).

## 2.3 Android security features

Android was designed for mobile devices which in general are more exposed than stationary systems. This results in some characteristic security features separate from Linux. A complete description of these differences would fill books. Inside this section we we will just enter into some aspects which are referred to in later sections.

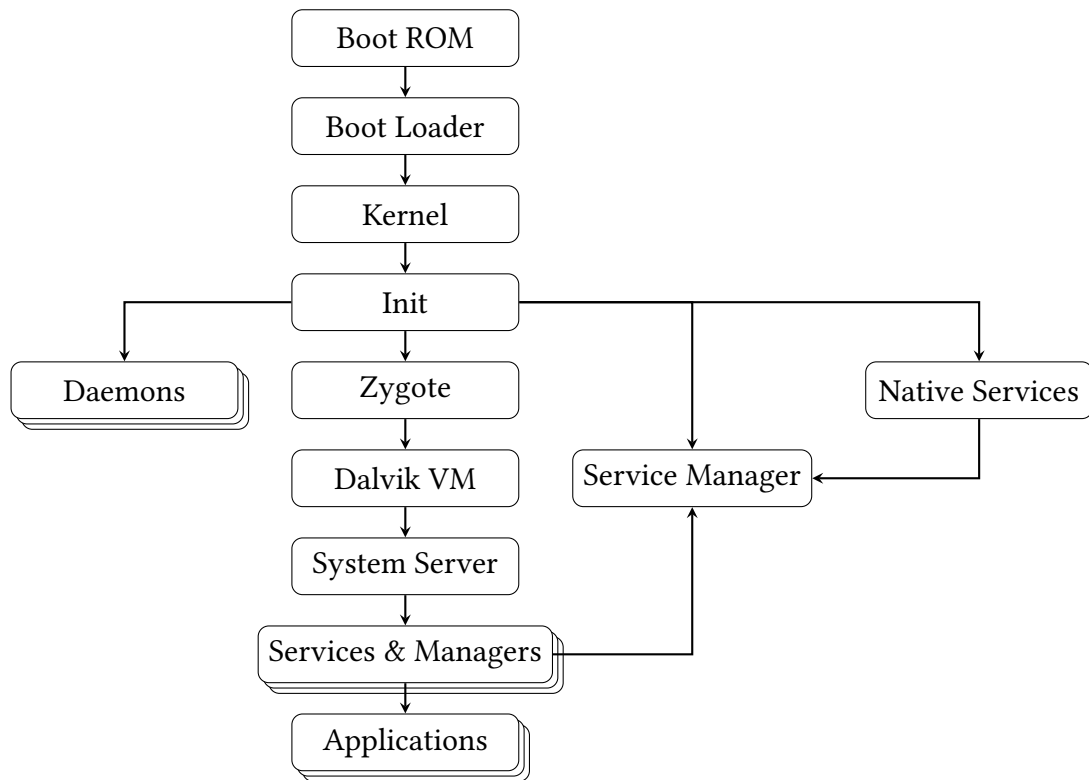
### 2.3.1 Android partitions and boot process

Linux is well supported by LiME and Volatility. But even though Android is based on a Linux kernel, its modifications and additions add up to major differences. Knowledge about the Android partitions is presupposed in order to understand most rooting techniques. Especially the *recovery* partition is mentioned in sections 2.3.3 and 4.1.3. To grasp the idea of the partitions is also a prerequisite for understanding the boot process which in turn is helpful in understanding what to expect in Android RAM.

The following is mainly based on Drake et al. [19], Elenkov [21] and additional details and illustrations from blog posts [40, 45, 51]. For better context recognition, terms from the summary fig. 2.9 appear in bold and italic typeface.

---

<sup>7</sup>see the Manjaro Linux experiment at section 3.2.3 for a counterexample



**Figure 2.9:** Android boot process

An Android device’s persistent storage memory is subdivided into logical storage units or divisions which are called partitions. Although the partition layout varies between vendors and platforms, and two different devices typically do not have the same partitions or the same layout, we can say that the most common are the *boot*, *system*, *data*, *recovery*, and *cache* partitions. [19]

When the device gets switched on, the **Boot ROM** is executed. This small piece of code is hardwired in the CPU Application-specific integrated circuit (ASIC). It determines the boot partition and loads the first stage of the *Boot Loader* from the boot media into the internal RAM. Once this is completed the *Boot ROM* gives over execution to the *Boot Loader*.

The **Boot Loader** is tailor-made for the device’s board and processor architecture. Device manufacturers might either build on popular boot loaders like RedBoot [20], U-Boot [17], Qi [49], or they develop own proprietary boot loaders. In particular the latter allow Original Equipment Manufacturers (OEMs) for implementing locks and restrictions.

*Boot loaders* execute in stages which we won’t further differentiate here. During these stages external RAM is detected and set up in order to load and execute further *boot loader* code. This main *Boot Loader* is the first major program that will run and may contain code to set up file systems, additional memory, network support, the modem CPU, low level memory protections, security options, etc. It also provides support for loading recovery images or putting the phone into a “fastboot mode” (also named “download

mode”). These allow for the writing (usually called flashing) of raw partition images to the device’s persistent storage, as well as booting transient system images (without flashing them to the device). (cmp. sections 2.3.3 and 4.1.3)

Finally, the *Boot Loader* will place the *Linux kernel* and the root file system RAM disk (*initrd*) from the boot media into the RAM as well as boot parameters for the kernel to read when it starts up. Now a jump to the *kernel* is performed which then assumes responsibility for the system.

The **Linux kernel**—the heart of the Android—is responsible for the process creation, inter process communication, device drivers, file system management etc. It first sets up everything that is needed for the system to run by initializing memory, I/O areas, interrupt controllers, setting up memory protections, caches and CPU scheduler, and mounting the root file system. Once the memory management units and caches have been initialized the system will be able to use virtual memory and launch user space processes.

The initial user space process executed on booting is the ***init*** executable located in the root folder. It is the “grandmother” [e.g. 40, 51] or “father” [19] of all other user-space processes because every other process in the system will be launched from this process or one of its descendants. The *init* process sets up the system, mounts directories like `/sys`, `/dev`, `/proc`, creates *daemons* and launches *native service* processes as described by the start up script `init.rc` (e.g. `rild` for telephony, `mtpd` for VPN access, and the Android Debug Bridge daemon (`adbd`)).

The very first service started by *init* is the ***service manager***. It manages all further services running in the system. Every service created registers itself with this process and gets a handler. Other processes will use this information for references.

One of the first *init* processes created on boot is ***Zygote***<sup>8</sup>. It initializes the *Dalvik VM* and forks to create multiple instances to support each android process. It facilitates using shared code across the VM instances resulting in a low memory footprint and minimal startup time which is ideal for embedded systems. Finally *Zygote* starts the *System Server*.

The ***System Server*** is the first Java component to run in the system. It will start all the Android Framework ***services*** and ***managers*** such as the battery service, Wi-Fi service, package manager or window manager, to name but a few.

Once the services are started and running Android broadcasts an `ACTION_BOOT_COMPLETED` event to all applications that have registered to receive this broadcast intent in their manifest. When this is complete, the system is considered fully booted.

The Android package manager parses each `.apk` file in `/system[/vendor]/app` and validates its `AndroidManifest.xml`. The application that is configured as the “Home” in its manifest—which is typically the launcher application—is launched resulting in the appearance of the user interface (UI).

---

<sup>8</sup>“A zygote (from Greek ζυγωτός *zygōtós* “joined” or “yoked”, from ζυγούν *zygoun* “to join” or “to yoke”), is the initial cell formed when two gamete cells are joined by means of sexual reproduction. [...] In single-celled organisms, the zygote divides to produce offspring. [...]” [77]

### 2.3.2 USB Debugging and ADB

In case of a running Android device being used as evidence, any form of the device's communication should be blocked. However, the examiner, when tending to acquire a RAM dump with LiME, needs a communication channel to the target device: first to place the LiME LKM and second to transfer the memory data to his forensic workstation. The predestined lane for this task is leveraging the Android Debug Bridge (ADB) over an USB cable.

Stock Android comes with this option disabled for security reasons. In order to enable it one must manually go to the system settings, switch on the “Developer options” and within these authorize “USB debugging”. For both actions a separate security question has to be confirmed. As this is direct interaction with the target's GUI a lock screen (PIN code, password, etc.) could get in the way very efficiently.

It is sometimes possible to activate ADB without interacting with the GUI. In 2013 Ossmann and Osborn [50] presented their research on particular Android devices featuring multiplexed wired interfaces. They demonstrated an attack against a function that permits unauthorized access to the device and to user data. Though the vulnerability was proven in general their degree of success varied significantly according to the installed software and the differences presented by different hardware platforms.

In section 3.3.2 we leverage this kind of communication to send commands with root permission to an Android Virtual Device. The same command fails when we send it to a real smartphone (e.g. in section 4.1.4). The reason is stated in the error message: “adb cannot run as root in production builds”. In this case we circumvent the restriction by manually opening a shell on the smartphone and entering the command here. In section 5.2 we have to execute commands by a shell script. Hence we replace the ADB daemon temporarily by *adb Insecure* from XDA developer Chainfire [13].

### 2.3.3 Rooting a booted Android

As stated in section 2.2.1 LiME is a kernel module which has to be copied onto the target device and activated with the `insmod` command. The latter requires root permissions.

*Root* is a special user on Unix-like systems sometimes referred to as the *superuser* because it has the broadest permissions to access the OS and its resources. Programs that are supposed to only be run by root belong to both the user and group root. A regular user can temporarily gain root permissions by using the superuser command `su`, if he knows the root password and if configuration allows root to logon at all. Alternatively a single command can be run as root by preceding the `sudo` command if the user is member of the sudo group.

As a matter of security Android by default denies root permissions to the human user. In contrary to Linux the human user has no user account by himself. In fact, every single Android application operates under its own user ID and group. There is nothing like `su`



or `sudo` in principle. Thus rooting means basically—albeit oversimplified—to copy a `su` binary to `/system/xbin`. There are almost always solutions to root an Android device even if the manufacturer tries to circumvent this. Normally it takes a few days or weeks after the release of a new device or after an OS update until the first rooting instructions can be found in the Internet. Typically, a big constraint regarding this work’s focus is that the device has to be rebooted which erases the volatile memory we want to acquire. Even more destructive is unlocking the boot loader which is often an essential part of the rooting recipe and by definition deletes even the otherwise non-volatile user data. We demonstrate this kind of rooting step by step in section 4.1. Lossless variants are used in sections 4.2 and 4.3.1.

Müller and Spreitzenbarth [47] developed a method to acquire volatile memory from low temperature cooled Samsung Nexus devices. They called their method Forensic Recovery Of Scrambled Telephones (FROST). It is based on flashing a custom recovery image and thus assumes an unlocked boot loader. Otherwise the boot loader has to become unlocked during the process and all persistent data is lost. Nevertheless in this case the RAM still can be dumped.

Drake et al. [19, chap. 3] describe in general how to theoretically root a booted Android device. Initial temporary root access ought to be gained by getting an ADB root shell through an unpatched Android security flaw. Permanent root access can then be achieved by placing a `su` binary with `setuid` root permissions onto the system partition. The system partition therefore has to be remounted in read/write mode beforehand and again remounted in read-only mode afterwards. The main challenge however at this juncture is to have an available security flaw and a corresponding exploit at hand.

### 2.3.4 A second look at related work

It is interesting to check the papers mentioned in section 1.3 for statements about used devices and how security mechanisms were handled. In general we can classify three groups of Android devices:

- Android Virtual Devices (AVDs): No obstacles for the examiner.
- Google Nexus devices: These devices are meant for developers. They have security mechanisms that allow for normal, everyday use. But the owner is free to deactivate them.
- all others: Most non-Nexus devices are protected by additional mechanisms against modification.

Thing et al. [71] just mention: *“The mobile phone used in our investigation was an Android mobile phone, the Google development set”*. No further modifications are discussed.

Sylve [65] tested LiME (or rather DMD at this time) on four different phones (two HTC EVO 4G, Droid Eris, Droid 2) and of course the Android Emulator. He mentions *“[...] an investigator should only use rooting techniques that have been verified to work reliably*

*on a particular device and furthermore, verified not to have undesirable consequences, such as introduction of malicious code. The chosen rooting technique should also not require the device to be reset, which will likely wipe volatile memory.*” But the paper’s focus was not on “rooting toolkit quality management”.

Ali-Gombe [2] gets root access without rebooting on two Motorola devices with a rooting tool called “Androot” [34]. These phones were from 2009 (Motorola Droid, also known as Milestone) and 2010 (Motorola Flipout). The paper is from 2012 which means the phones were two years old when examined.

Macht [44] writes: *“What method works depends heavily on the device and the Android version it is powered by. [...] Because of this, this thesis assumes that an unlocked, rooted device is already available”.*

Apostolopoulos et al. [10] work with DDMS on emulators and phones without mentioning how they were prepared. Later in [78] they described using LiME but mentioned some limitations: *“1. It requires rooted devices [...] 2. [...] The source code of kernel is not always available [...] 3. It requires the config.gz file [...]”.*

So we already see a trend: For research purposes it is easiest to work with a virtualized Android device. If real hardware is needed a Google Nexus places the least obstacles in the researcher’s path. Other hardware is applicable if there is a rooting solution available. Most times the hardware is prepared for experiments by unlocking the boot loader, rooting or disabling the lock screen. However, in contrast to research papers a law enforcement forensics specialist is not free to chose a smartphone model. They have to analyze whatever arrives in their lab.

# 3

## VIRTUAL MACHINES – WINDOWS, LINUX, ANDROID

---

Forensic memory analysis is a young discipline. The work on Volatility started in the mid-2000s. It was originally developed for use with Windows operation systems. Around 2010 Andrew Case contributed his work on Linux memory analysis to the Volatility project. Today Android stands out because of the discrepancy between its ubiquity and the comparatively small amount of research for this operating system in the field of memory analysis.

In this chapter we demonstrate the ease of getting started with memory analysis on Windows. For a Linux target the same process requires additional steps and with different GNU/Linux Systems we additionally have to pay attention to divergent characteristics. Finally Android—though based on a Linux kernel—has to be handled quite differently in comparison with GNU/Linux.

For the sake of convenience this chapter's experiments are performed with Virtual Machines. Windows and GNU/Linux are emulated with VirtualBox, Android is represented as an AVD. Publications predominantly mention AVDs in their reviews. This allows for efficient research whereas working with real devices is more complex as we discuss in chapter 4.

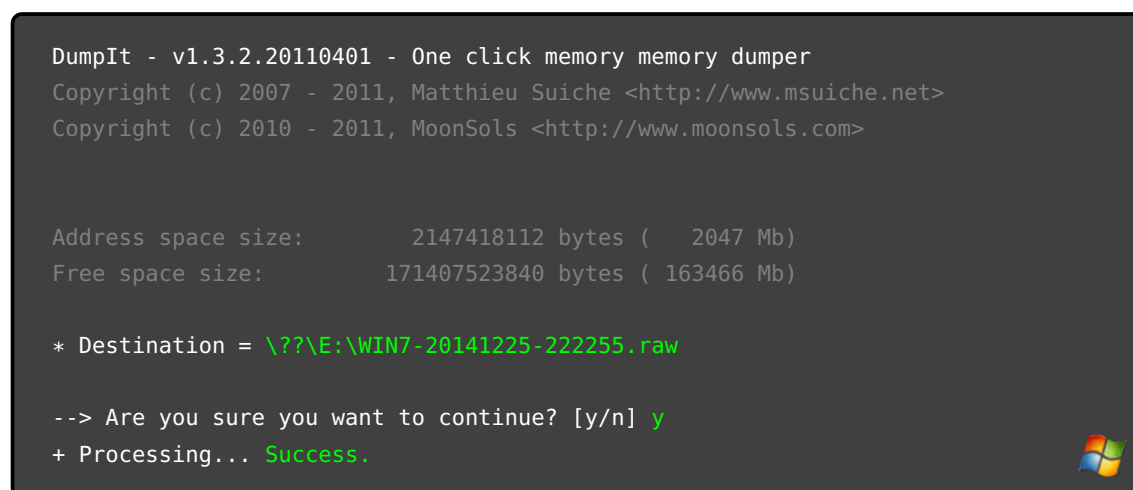
## 3.1 VirtualBox Windows Guest

A simple demonstration shall show how effortless memory analysis can be when starting out with a Windows target.

There are numerous approaches to acquiring a Windows PC's memory. For RAM up to 4 GB *Inception* [43] is a noteworthy tool for hardware-based acquisition. It exploits PCI-based Direct Memory Access (DMA). That means it can attack over any PCI/PCIe interfaces such as FireWire or Thunderbolt, etc. and has full read/write access to the lower 4 GB of RAM on the victim's computer. In addition, Volatility itself supports acquisition and interrogation of memory over FireWire [42, pg. 79].

The Volatility project lists 10 software-based acquisition tools [42, pg. 79 et seqq.]. Most of them are commercial. We demonstrate how straightforward the process is using the freely available *MoonSols DumpIt* [62] on a Windows 7 computer.

DumpIt is a single executable which can be saved to a USB flash memory drive. Once inserted into the target computer's USB port, the *DumpIt* executable can be run. After confirming the Windows 7 User Account Control (UAC) security question, the program writes a memory dump to the flash drive. The console output is shown in fig. 3.1.



```
DumpIt - v1.3.2.20110401 - One click memory memory dumper
Copyright (c) 2007 - 2011, Matthieu Suiche <http://www.msuiche.net>
Copyright (c) 2010 - 2011, MoonSols <http://www.moonsols.com>

Address space size:      2147418112 bytes (   2047 Mb)
Free space size:         171407523840 bytes ( 163466 Mb)

* Destination = \\??\\E:\\WIN7-20141225-222255.raw

--> Are you sure you want to continue? [y/n] y
+ Processing... Success.
```

Figure 3.1: Windows: Moonsols DumpIt output

Volatility can analyze the data straightaway as shown in Figure fig. 3.2. Just by using this dump Volatility can determine the Windows version and propose the correct profile. The command `python vol.py -info` lists about 30 profiles for the different major versions of Windows.

Though this demonstration was quite simple, it is important to note that several requirements had to be met:

- physical access to the computer
- access to the UI

```

$ python vol.py -f ~/tmp/WIN7-20141225-222255.raw imageinfo
Volatility Foundation Volatility Framework 2.4
Determining profile based on KDBG search...

Suggested Profile(s) : Win7SP0x86, Win7SP1x86
AS Layer1 : IA32PagedMemoryPae (Kernel AS)
AS Layer2 : FileAddressSpace (/tmp/WIN7-20141225-222255.raw)
PAE type : PAE
DTB : 0x185000L
KDBG : 0x8296ec30
Number of Processors : 1
Image Type (Service Pack) : 1
KPCR for CPU 0 : 0x8296fc00
KUSER_SHARED_DATA : 0xffdf0000
Image date and time : 2014-12-25 22:23:03 UTC+0000
Image local date and time : 2014-12-25 23:23:03 +0100
$ python vol.py -f ~/tmp/WIN7-20141225-222255.raw --profile Win7SP1x86 pslist
Volatility Foundation Volatility Framework 2.4
Offset(V) Name PID PPID Thds Hnds Sess Wow64 Start Exit
0x84a3c630 System 4 0 88 553 -1 0 2014-12-25 20:12:19 UTC+0000
0x85ede380 smss.exe 288 4 2 29 -1 0 2014-12-25 20:12:19 UTC+0000
0x865227a0 csrss.exe 368 360 9 432 0 0 2014-12-25 20:12:20 UTC+0000
<snip>

```

Figure 3.2: Windows: Volatility output

- administrator privileges

The examiner requires at minimum physical access to the computer. That is the USB port if a USB flash drive is used as well as an input device—usually a keyboard and/or a mouse—in order to answer the UAC question and to start the program. The latter requires an unlocked screen and a running user account with sufficient privileges.

In case the screen was locked by a password and/or privileges had to be escalated the beforementioned tool Inception will be useful. For privilege escalation on many Windows systems the “NTLM reflection attack through WebDAV” can be leveraged. Google’s security research team disclosed this vulnerability and a proof-of-concept exploit in March 2015 [30]. It is reported that Microsoft won’t fix this issue.

## 3.2 VirtualBox GNU/Linux Guests

Compared to Windows the effort increases with a GNU/Linux target. As Android is based on a Linux kernel we now examine how to start the memory analysis of a GNU/Linux system.

We demonstrate the utilization of LiME and adjacent analysis with Volatility targeting five VirtualBox VMs which are each running a different GNU/Linux distribution in their

particular current release as of this writing. These are:

1. Debian 7.7 stable (64 bit)
2. Fedora 21 Workstation (64 bit)
3. Manjaro Linux Xfce 0.8.11 (64 bit)
4. openSUSE 13.2 (64 bit)
5. Ubuntu Desktop 14.04.1 LTS (64 bit)

Two major differences to the Windows example demonstrated before (cf. section 3.1) will now become apparent. We used LiME for acquisition. There were almost no ready-to-use Volatility Linux profiles available.

The following demonstrations show that the process of making an applicable LiME module as well as creating a Volatility profile might differ significantly between varying GNU/Linux distributions. Not only the VirtualBox guest additions which at minimum are needed to exchange terminal text and files between host and guest conveniently are handled in different ways. But also commands, software, and files available come in various flavors.

LiME is able to write the acquired memory dump to either a local drive or over the network. For these experiments the local drive option is depicted. With Android in section 3.3 we will use the network option.

The Linux VM experiments have been processed on a Windows workstation. The Windows standalone version of Volatility is used to proof the interoperability of memory dump and Volatility profile.

### 3.2.1 Debian GNU/Linux

The first system looked at is a *Debian 7.7 stable (64 bit)*. After installation from the ISO image [68] the commands `apt-get update` and `apt-get upgrade` install all updates. A VM snapshot  $S_1$  preserves this state for later RAM acquisition.

Debian differs from the subsequently described GNU/Linux distributions by providing full VirtualBox guest additions by default. Additionally Debian and Ubuntu are the only tested systems to come with both a compiler and the Linux header files installed during the standard installation process. With the command `apt-get install dwarfdump` the only software which has to become installed subsequently is added. It will later be needed for creating a Volatility profile.

The LiME source code from 504ENSICS Labs [1] can become compiled as illustrated in fig. 3.3. The Volatility profile creation displayed in fig. 3.4 shows a Debian characteristic: The standard installation did not come with the `zip` command. Instead we leverage the `7z` command available in default Debian.

As we have now both the compiled LiME LKM and the Volatility profile, the VM state is stored to snapshot  $S_2$ . Previous state  $S_1$  which is to be examined is restored. As before

```
$ unzip LiME-master.zip
<snip>
$ cd LiME-master/src
$ make
<snip>
mv lime.ko lime-3.2.0-4-amd64.ko
```

**Figure 3.3:** Debian: Compile LiME

```
$ unzip volatility-master.zip
<snip>
$ cd volatility-master/tools/linux/
$ make
<snip>
dwarfdump -di module.ko > module.dwarf
<snip>
$ 7z a -tzip Debian_7.7.0_3.2.0-4-amd64.zip module.dwarf /boot/System.map-$(uname -r)
<snip>
```

**Figure 3.4:** Debian: Create Volatility profile

```
$ sudo insmod lime-3.2.0-4-amd64.ko "path=/home/foo/debian.lime format=lime"
```

**Figure 3.5:** Debian: Initiate memory acquisition

with DumpIt in the Windows example (cmp. section 3.1) the LiME module has to be copied onto the target system. Acquisition is started with the `insmod` command (cmp. fig. 3.5).

With the memory dump and the tailored profile Volatility can be called to analyze the dump; for instance list the processes as shown in fig. 3.6

### 3.2.2 Fedora GNU/Linux

The second experiment is done on a *Fedora 21 Workstation (64 bit)* system [69]. The software is ensured to be up to date by the command `yum update` and stored to snapshot  $S_1$ .

In order to be able to compile the LiME LKM and create the Volatility profile, the kernel headers, a compiler, and `dwarfdump` have to be installed with `yum`. The VirtualBox guest additions can be installed from the VirtualBox ISO image. To compile successfully the guest additions require Dynamic Kernel Module Support (DKMS). The whole process

```

D:\vol>$volatility-2.4.standalone.exe --plugins=. -f debian.lime --profile LinuxDebian-7_7_0-3_2_0-4-amd64x64 linux-pslist
Volatility Foundation Volatility Framework 2.4

```

Offset	Name	Pid	Uid	Gid	DTB	Start Time
0xfffff88003e1c0740	init	1	0	0	0x000000003da85000	2015-01-14 10:12:57 UTC+0000
0xfffff88003e1c0040	kthreadd	2	0	0	-----	2015-01-14 10:12:57 UTC+0000
0xfffff88003e1c5780	ksoftirqd/0	3	0	0	-----	2015-01-14 10:12:57 UTC+0000
0xfffff88003e1c5080	kmworker/0:0	4	0	0	-----	2015-01-14 10:12:57 UTC+0000
0xfffff88003e1c97c0	kmworker/u:0	5	0	0	-----	2015-01-14 10:12:57 UTC+0000
0xfffff88003e1c90c0	migration/0	6	0	0	-----	2015-01-14 10:12:57 UTC+0000
0xfffff88003e1ce800	watchdog/0	7	0	0	-----	2015-01-14 10:12:57 UTC+0000
0xfffff88003e1ce100	cpuset	8	0	0	-----	2015-01-14 10:12:57 UTC+0000
0xfffff88003e1d0840	khelper	9	0	0	-----	2015-01-14 10:12:57 UTC+0000
0xfffff88003e1d0140	kdevtmpfs	10	0	0	-----	2015-01-14 10:12:57 UTC+0000
0xfffff88003e1e1880	netns	11	0	0	-----	2015-01-14 10:12:57 UTC+0000
0xfffff88003e1e1180	sync_supers	12	0	0	-----	2015-01-14 10:12:57 UTC+0000
<snip>						
0xfffff88003c9c1780	tracker-miner-f	3288	1000	1000	0x0000000003d0b6000	2015-01-14 10:13:39 UTC+0000
0xfffff88003c9c1080	tracker-store	3289	1000	1000	0x0000000003d0c1000	2015-01-14 10:13:40 UTC+0000
0xfffff88003cd4e7c0	nm-applet	3290	1000	1000	0x0000000003d0f8000	2015-01-14 10:13:40 UTC+0000
0xfffff88003c9e21c0	dconf-service	3295	1000	1000	0x0000000003d139000	2015-01-14 10:13:41 UTC+0000
0xfffff88003ce54840	gnome-shell-cal	3318	1000	1000	0x0000000003ce8d000	2015-01-14 10:13:46 UTC+0000
0xfffff88003ca158c0	mission-control	3323	1000	1000	0x0000000003cf0e000	2015-01-14 10:13:46 UTC+0000
0xfffff88003cf06780	goa-daemon	3327	1000	1000	0x0000000003cfbd000	2015-01-14 10:13:46 UTC+0000
0xfffff88003bc35880	gnome-terminal	3360	1000	1000	0x0000000003c024000	2015-01-14 10:13:59 UTC+0000
0xfffff8800373b0780	gnome-ptty-helpe	3366	1000	1000	0x0000000003c753000	2015-01-14 10:14:00 UTC+0000
0xfffff88003cf54040	bash	3367	1000	1000	0x0000000003c05e000	2015-01-14 10:14:00 UTC+0000
0xfffff88003da9a740	sudo	3418	0	1000	0x0000000003caab000	2015-01-14 10:16:27 UTC+0000
0xfffff8800373b0080	insmod	3419	0	0	0x0000000003ca8b000	2015-01-14 10:16:30 UTC+0000



Figure 3.6: Debian: List processes with Volatility



```
$ sudo yum install gcc kernel-devel dkms libdwarf-tools
<snip>
$ sudo /run/media/foo/VBOXADDITIONS_4.3.10_93012/VBoxLinuxAdditions.run
<snip>
```



**Figure 3.7:** Fedora: Install dependencies

```
$ cd ~
$ unzip LiME-master.zip
<snip>
$ cd LiME-master/src
$ make
<snip>
mv lime.ko lime-3.17.8-300.fc21.x86_64.ko

$ cd ~
$ unzip volatility-master.zip
<snip>
$ cd volatility-master/tools/linux/
$ make
<snip>
dwarfdump -di module.ko > module.dwarf
<snip>
$ sudo zip -j Fedora_21_3.17.8-300.fc21.x86_64.zip module.dwarf
/boot/System.map-'uname -r'
<snip>
```



**Figure 3.8:** Fedora: Compile LiME and create Volatility profile

```
$ sudo insmod lime-3.17.8-300.fc21.x86_64.ko "path=/home/foo/fedora.lime
format=lime"
```



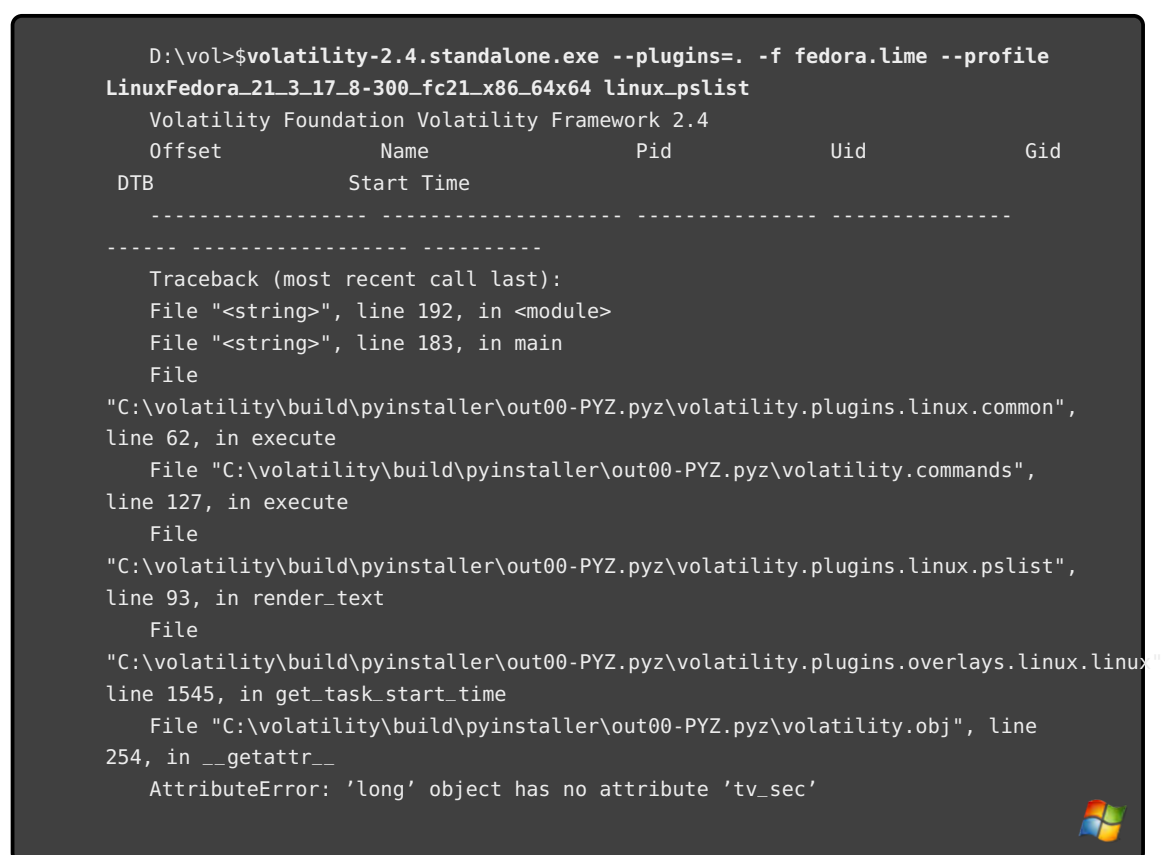
**Figure 3.9:** Fedora: Initiate memory acquisition

is shown in fig. 3.7. After these changes the VM's state is saved to snapshot  $S_2$  for the RAM acquisition to come.

Compiling the LiME and Volatility modules is similar to Debian (cmp. fig. 3.8). For this and the following GNU/Linux distributions `zip` can be leveraged to pack the Volatility profile.

The system's state is frozen in snapshot  $S_3$ . A memory dump is acquired from the snapshot  $S_2$  (cmp. fig. 3.9).

As we can see in fig. 3.10 Volatility fails at handling the memory dump.



```
D:\vol>$volatility-2.4.standalone.exe --plugins=. -f fedora.lime --profile
LinuxFedora_21_3_17_8-300_fc21_x86_64 linux_pslist
Volatility Foundation Volatility Framework 2.4
Offset          Name                Pid          Uid          Gid
DTB             Start Time
-----
Traceback (most recent call last):
  File "<string>", line 192, in <module>
  File "<string>", line 183, in main
  File
"C:\volatility\build\pyinstaller\out00-PYZ.pyz\volatility.plugins.linux.common",
line 62, in execute
  File "C:\volatility\build\pyinstaller\out00-PYZ.pyz\volatility.commands",
line 127, in execute
  File
"C:\volatility\build\pyinstaller\out00-PYZ.pyz\volatility.plugins.linux.pslist",
line 93, in render_text
  File
"C:\volatility\build\pyinstaller\out00-PYZ.pyz\volatility.plugins.overlays.linux.linux",
line 1545, in get_task_start_time
  File "C:\volatility\build\pyinstaller\out00-PYZ.pyz\volatility.obj", line
254, in __getattr__
AttributeError: 'long' object has no attribute 'tv_sec'
```

**Figure 3.10:** Fedora: List processes with Volatility fails

We were not able to solve this issue and submitted a bug report to the Volatility project [76]. At the time of this writing the issue was being examined but had not been solved yet.

### 3.2.3 Manjaro GNU/Linux

With the *Manjaro Linux Xfce 0.8.11 (64 bit)* distribution [46] we face another package manager: The system is updated with `pacman -Syu`. Again we save this state with a

snapshot  $S_1$ . VirtualBox host and guest can share their clipboards by default. But for file exchanges initiated from the Windows host by the `VBoxManage.exe` command, on the guest `VBoxService` must be started. For the modules to compile and to create the Volatility profile the kernel headers and `dwarfdump` have to be installed. (cmp. fig. 3.11)

```
$ sudo pacman -S linux316-headers libdwarf
<snip>
$ sudo VBoxService
```



**Figure 3.11:** Manjaro: Install dependencies

```
$ cd ~
$ unzip LiME-master.zip
<snip>
$ cd LiME-master/src
$ make
<snip>
mv lime.ko lime-3.16.7.3-1-MANJARO.ko

$ cd ~
$ unzip volatility-master.zip
<snip>
$ cd volatility-master/tools/linux/
$ make
<snip>
dwarfdump -di module.ko > module.dwarf
<snip>
$ nm /usr/lib/modules/3.16.7.3-1-MANJARO/build/vmlinux > System.map-'uname -r'
$ zip -j Manjaro_Xfce_0.8.11_3.16.7.3-1-MANJARO.zip module.dwarf System.map-'uname
-r'
<snip>
```



**Figure 3.12:** Manjaro: Compile LiME and create Volatility profile

```
$ sudo insmod lime-3.16.7.3-1-MANJARO.ko "path=/home/foo/manjaro.lime format=lime"
```



**Figure 3.13:** Manjaro: Initiate memory acquisition

Compilation of the LiME and Volatility modules works as expected. Only when it becomes time to pack the Volatility profile does the missing `System.map` file in the `/boot` folder need to be addressed. This is because Manjaro standard kernels don't come with this file. Instead the required symbol table can be extracted from the `vmlinux` file by leveraging the `nm` command. (cmp. fig. 3.12)

While the memory acquisition performs without further issues (see command in fig. 3.13),

```
D:\vol>$volatility-2.4.standalone.exe --plugins=. -f manjaro.lime --profile LinuxManjaro-Xfce-0-8-11-3-16-7-3-1-MANJAROx64
linux-pslist
Volatility Foundation Volatility Framework 2.4
Offset      Name      Pid      Uid      Gid      DTB      Start Time
-----
0xffff88003d9f8000 systemd    1         0         0         0x000000003c553000
0xffff88003d9f8a30 kthreadd  2         0         0         -----
0xffff88003d9f9460 ksoftirqd/0 3         0         0         -----
0xffff88003d9f9e90 kworker/0:0 4         0         0         -----
0xffff88003d9fa8c0 kworker/0:0H 5         0         0         -----
0xffff88003d9fb2f0 kworker/u2:0 6         0         0         -----
0xffff88003d9fbd20 rcu_preempt 7         0         0         -----
0xffff88003d9fc750 rcu_sched  8         0         0         -----
0xffff88003d9fd180 rcu_bh     9         0         0         -----
0xffff88003d9fdbb0 migration/0 10        0         0         -----
0xffff88003d9ff010 khelper    12        0         0         -----
0xffff88003db00000 kdevtmpfs 13        0         0         -----
<snip>
0xffff8800374b0000 gvfs-gphoto2-vo 6505      1000      100      0x0000000037465000
0xffff88003a55c750 gvfs-mtp-volume 6638      1000      100      0x0000000037496000
0xffff8800374b32f0 gvfs-afc-volume 6650      1000      100      0x000000003750b000
0xffff88003757c750 gvfsd-trash  6696      1000      100      0x0000000037442000
0xffff88003c1af010 obex-data-serve 7827      1000      100      0x000000003a679000
0xffff88003a771460 krfcommnd  7940      0         0         -----
0xffff88003a770a30 xfce4-terminal 14211     1000      100      0x000000003cace000
0xffff88003bd61e90 gnome-pty-helpe 14432     1000      100      0x000000003a7bb000
0xffff88003a7765e0 bash      14438     1000      100      0x000000001c001000
0xffff880037579460 VBoxService 17186     0         0         0x000000003a725000
0xffff8800374b1e90 sudo      837       0         100      0x000000003b354000
0xffff8800374b65e0 insmod    838       0         0         0x000000003a614000
```

Figure 3.14: Manjaro: List processes with Volatility

Volatility's `linux_pslist` module presents an unexpected output. Though processes are listed, there is no information in the "Start Time" column. (cmp. fig. 3.14). This is a bug in Volatility at the time of this writing and the following two experiments show the same behavior. This very special bug and a solution will be discussed later in section 3.2.7.

### 3.2.4 OpenSUSE GNU/Linux

The fourth GNU/Linux distribution in this chapter is *openSUSE 13.2 (64 bit)*. It presents the fourth package management engine named *ZYpp*. User interfaces are `zypper` on the command line and *YaST* as the graphical frontend. The system is updated to an up-to-date software level with the command `zypper update`. This state gets locked in snapshot  $S_1$ .

```
$ sudo VBoxService
<snip>
$ sudo zypper install gcc kernel-desktop-devel-3.16.7-7.1 libdwarf-tools
<snip>
```




Figure 3.15: OpenSUSE: Install dependencies

```
$ cd ~
$ unzip LiME-master.zip
<snip>
$ cd LiME-master/src
$ make
<snip>
mv lime.ko lime-3.16.7-7-desktop.ko

$ cd ~
$ unzip volatility-master.zip
<snip>
$ cd volatility-master/tools/linux/
$ make
<snip>
dwarfdump -di module.ko > module.dwarf
<snip>
$ zip -j OpenSUSE_13.2_3.16.7-7-desktop.zip module.dwarf /boot/System.map-`uname -r`
<snip>
```




Figure 3.16: OpenSUSE: Compile LiME and create Volatility profile

As with Manjaro Linux the full VirtualBox guest additions features have to be enabled with the `VBoxService` command. For the steps to come the compiler, kernel headers, and `dwarfdump` have to be installed. (cmp. fig. 3.15) With these requirements fulfilled there are no more hurdles to take in order to prepare LiME and Volatility (cmp. fig. 3.16) and save the system's state to snapshot  $S_2$ .

```
$ sudo insmod lime-3.16.7-7-desktop.ko "path=/home/foo/opensuse.lime
format=lime"
```



**Figure 3.17:** OpenSUSE: Initiate memory acquisition

Memory acquisition of the snapshot  $S_1$  itself does not differ from the other experiments (cmp. fig. 3.17).

As mentioned before, LiME is able to write the acquired memory dump not only to a local drive but also over TCP. Although the Linux experiments describe only the former the latter was also tested. For the network option LiME creates a listening socket on the target system and starts acquisition as soon as the network connection has been established from the examiner's workstation. In the special case of openSUSE the *SUSE firewall* arrives activated in the standard installation and blocks any incoming requests by default. Thus a port has to be opened with an appropriate entry in `/etc/sysconfig/SuSEfirewall2` before the acquisition can begin. One way to achieve this is using the command line is shown in fig. 3.18. However, both modifying the firewall rules and writing a dump to disk is negatively affecting the integrity of the forensic analysis. The examiner has to decide whether it is more important to make sure that no network buffers are overwritten or to avoid changes on disk.

```
$ sed -i "s|\\(FW_SERVICES_EXT_TCP=\\\".*\\\")|\\1 4444\\\"|;s|\\\" |\\\"|\"
/etc/sysconfig/SuSEfirewall2
$ /sbin/SuSEfirewall2 start
```



**Figure 3.18:** OpenSUSE: Open port 4444 in SUSE Firewall

### 3.2.5 Ubuntu GNU/Linux

The final tested distribution is *Ubuntu Desktop 14.04.1 LTS (64 bit)*. Compared with the Debian experiment (cmp. section 3.2.1) only three differences were observed:

- VirtualBox guest additions have to be installed
- the `zip` command is available
- Volatility's `linux_pslist` plugin does not show start times

### 3.2.6 Cross compilation

We have shown how to act if a development system similar to the one to be examined is available. In an artificial environment with cloneable VMs or, as we did, with the

ability to create snapshots this is easy to accomplish. But if these ideal conditions aren't given, cross compilation might be a solution. That is, the LiME LKM and the Volatility debug module each compile against the target's kernel headers instead of those of the current system. In order to compile with the correct settings the `.config` file has to be extracted from either the target's kernel header files package or the appropriate kernel image package. The same applies for the `System.map` file which is needed to create the Volatility profile.

Ligh et al. [42, pg. 587 et seqq.] demonstrate cross compiling by building a Volatility profile for an Ubuntu kernel. At this point we show how to create a LiME LKM and a Volatility profile for the above mentioned Fedora 21 VM (cmp. section 3.2.2).

Initially, this task was attempted on a Debian 7.7 (stable) system. Due to a missing suitable compiler finally an Ubuntu 14.04 system was used instead. However, both rely on the Debian Advanced Packaging Tool (APT). As there is no APT package repository available for the Fedora packages required, a web research is done and leads to the "Fedora buildsystem" offering file downloads for building `kernel-3.17.8-300.fc21` [70]. The `kernel-devel` RPM package contains all required files:

- Configuration for compiling: `config-3.17.8-300.fc21.x86_64`
- Symbol table used by the kernel: `System.map-3.17.8-300.fc21.x86_64`
- Header files: `/usr/src/kernels/3.17.8-300.fc21.x86_64`

Basically RPM package payloads come as cpio archives [55]. One gains access by extracting the payload with `rpm2cpio` and pipes the output stream to the `cpio` command for extracting.

Before the modules can become compiled, their `Makefile`s have to be edited so that they point to the Fedora header files. The Fedora 21 kernel has been compiled with gcc version 4.9 and the configuration includes an option `-fstack-protector-strong` which was introduced with this compiler version. Our first choice as development system Debian 7.7 (stable) just comes with `gcc-4.7`. Though Ubuntu 14.04 only has `gcc-4.8` installed, it offers an additional `gcc-4.9` testing package for installation. The new `Makefile`s and the new compiler version have to be added as parameters to the `make` command.

The whole process is shown in fig. 3.19.

We have shown how to compile a LiME module and a Volatility profile in case that there is no copy of the target system available. Although the development system and the target are both standard Linux systems we still had to carefully gather and assemble all the required tools and files. In addition, a major disadvantage of this method is the unavailability of a test system.

### 3.2.7 Volatility timekeeper bug

As mentioned in sections 3.2.3 to 3.2.5 the `linux_pslist` plugin essentially showed the process list but failed to include the data in the "Start Times" columns. No error

```

$ # >>>> prepare Ubuntu with gcc-4.9 and dwarfdump
$ sudo add-apt-repository ppa:ubuntu-toolchain-r/test
<snip>
$ sudo apt-get update
<snip>
$ sudo apt-get install gcc-4.9 dwarfdump
<snip>
$
$ # >>>> get devel package and check existence of .config and System.map
$ mkdir ~/xc && cd $_
$ wget https://kojipkgs.fedoraproject.org/packages/kernel/3.17.8/300.fc21/x86_64/
  kernel-devel-3.17.8-300.fc21.x86_64.rpm
<snip>
$ rpm2cpio kernel-devel-3.17.8-300.fc21.x86_64.rpm | cpio -t | egrep '\.config'
./usr/src/kernels/3.17.8-300.fc21.x86_64/.config
75312 blocks
$ rpm2cpio kernel-devel-3.17.8-300.fc21.x86_64.rpm | cpio -t | grep 'System.map'
./usr/src/kernels/3.17.8-300.fc21.x86_64/System.map
75312 blocks
$ # >>>> extract RPM package
$ rpm2cpio kernel-devel-3.17.8-300.fc21.x86_64.rpm | cpio -id
<snip>
$
$ # >>>> edit LiME Makefile
$ cd ~/LiME/src/
$ cat Makefile | \
> sed '/KVER ?=/i KDIR ?= ~/xc/usr/src/kernels/3.17.8-300.fc21.x86_64' | \
> sed 's/$(shell uname -r)/3.17.8-300.fc21.x86_64/' | \
> sed 's:/lib/modules/$(KVER)/build:$(KDIR):' > Makefile.fedora
$ # >>>> compile LiME module
$ make CC='/usr/bin/gcc-4.9' -f Makefile.fedora
<snip>
mv lime.ko lime-3.17.8-300.fc21.x86_64.ko
$
$ # >>>> edit Volatility Makefile
$ cd ~/volatility/tools/linux/
$ cat Makefile | \
> sed 's:KDIR ?= /:KDIR ?= ~/xc/usr/src/kernels/3.17.8-300.fc21.x86_64:' | \
> sed 's:$(KDIR)/lib/modules/$(KVER)/build:$(KDIR):' > Makefile.fedora
$ # >>>> compile Volatility module and create profile
$ make CC='/usr/bin/gcc-4.9' -f Makefile.fedora
<snip>
dwarfdump -di module.ko > module.dwarf
<snip>
$ sudo zip -j Fedora_21_3.17.8-300.fc21.x86_64.zip module.dwarf
  ~/xc/usr/src/kernels/3.17.8-300.fc21.x86_64/System.map
<snip>
$ # >>>> result
$ find ~ -type f -name '*fc21*'
/home/foo/LiME/src/lime-3.17.8-300.fc21.x86_64.ko
/home/foo/volatility/tools/linux/Fedora_21_3.17.8-300.fc21.x86_64.zip

```



Figure 3.19: Cross compile LiME and Volatility for Fedora 21



message pointed to a reason for this unexpected behavior. Volatility commands can be supplemented by a debug option `-d`. With this we accessed the following two error messages associated with the processes:

```
Requested symbol wall_to_monotonic not found in module kernel
Requested symbol total_sleep_time not found in module kernel
```

The two mentioned symbols `wall_to_monotonic` and `total_sleep_time` are part of a C structure `timekeeper`. This structure is defined in Volatility's `/tools/linux/module.c`. The code originates from Linux' `kernel/time/timekeeping.c` until kernel version 3.6 [25]. Starting with Linux kernel version 3.7 the structure is defined in the header file `/include/linux/timekeeper_internal.h` [24, 26].

In consequence we forked the GitHub Volatility project, added our findings and submitted a pull request (see fig. 3.20) which has been accepted by the project's maintainers [75].

	✱	@@ -404,6 +404,11 @@ struct slab slab;
404	404	<code>#endif</code>
405	405	
406	406	<code>#if LINUX_VERSION_CODE &gt; KERNEL_VERSION(2,6,31)</code>
	407	<code>+#if LINUX_VERSION_CODE &gt;= KERNEL_VERSION(3,7,0)</code>
	408	<code>/* Starting with Linux kernel 3.7 the struct timekeeper is defined in</code>
		<code>include/linux/timekeeper_internal.h */</code>
	409	<code>+#include &lt;linux/timekeeper_internal.h&gt;</code>
	410	<code>+#else</code>
	411	<code>/* Before Linux kernel 3.7 the struct timekeeper has to be taken from</code>
		<code>kernel/time/timekeeping.c */</code>
407	412	
408	413	<code>typedef u64 cycle_t;</code>
409	414	
	✱	@@ -465,6 +470,7 @@ struct timekeeper {
465	470	<code>seqlock_t lock;</code>
466	471	<code>};</code>
467	472	
	473	<code>+#endif</code>
468	474	
469	475	<code>struct timekeeper my_timekeeper;</code>
470	476	
	✱	

Figure 3.20: Volatility fix in `tools/linux/module.c` [75]

### 3.2.8 Recapitulation

We showed for five different GNU/Linux distributions how to initiate a memory analysis. No systems was like the other. The processes had to be tailored for each target. Table 3.1 shows the characteristics we encountered.

Despite all disparities the process in general was similar and memory acquisition was

**Table 3.1:** Differences noticed during GNU/Linux experiments

GNU/Linux	Debian	Fedora	Manjaro	OpenSUSE	Ubuntu
Version	7.7	21	0.8.11	13.2	14.04.1
Kernel	3.2.0	3.17.8	3.16.7	3.16.7	3.13.0
Guest additions	OK	install	VBoxService	VBoxService	install
Package manager	APT <code>apt-get</code>	RPM <code>yum</code>	pacman <code>pacman</code>	Libzypp <code>zypper</code>	APT <code>apt-get</code>
Dependencies					
DKMS	—	dkms	—	—	—
Compiler	—	gcc	—	gcc	—
Headers	—	kernel-devel	linux316-headers	kernel-desktop-devel-3.16.7-7.1	—
dwarfdump	dwarfdump	libdwarf-tools	libdwarf	libdwarf-tools	dwarfdump
Firewall	○	○	○	●	○
System.map	<code>/boot/...</code>	<code>/boot/...</code>	<code>nm vmlinux</code>	<code>/boot/...</code>	<code>/boot/...</code>
zip	<code>7z</code>	<code>zip</code>	<code>zip</code>	<code>zip</code>	<code>zip</code>
Volatility	OK	error	no time	no time	no time

always feasible. Apart from the Fedora related bug mentioned in section 3.2.2 we could demonstrate Volatility’s successful operation.

Both the unsolved Fedora bug and the solved timekeeper bug showed that the forensic practitioner has to know his open source tools in depth. If you are pressed for time it is not productive to open an issue and wait for volunteers to solve it.

Compared to the previous Windows experiment this section illustrated an increased complexity. The previously mentioned preconditions remain valid and are broadened. On the target system we need:

- access to a terminal
- administrator privileges

Additionally we need a development system for (cross-)compiling the LiME and Volatility modules with:

- Linux header files of the target system
- compiler `.config` settings from either the target’s
  - `/proc/config.gz`
  - header files
  - kernel image package
- symbol table `System.map` from either the target’s
  - `/boot/System.map-...`

- header files
- kernel image package
- kernel `vmlinux` (extracted with `nm`)
- specific version of C compiler
- `dwarfdump`

Now that it is clear how LiME and Volatility can be used to examine a Linux system we will have a look at Linux kernel based Android in the following section.

## 3.3 Android Emulator

Most papers about Android memory forensics deal with Android Virtual Devices (AVDs). On the one hand this is understandable as an AVD is much easier to handle than a real device. This is on account of the many barriers we will discuss in the next chapter 4. A caveat on the other hand is that the AVD kernel by default is not LKM enabled. Thus a new kernel has to be compiled in order to use a LiME module on an AVD.

There used to be an easy way to dump an AVD's memory from the command line. Sylve et al. [64] describe how they tested LiME's accuracy on the Goldfish<sup>1</sup> device with the QEMU Monitor command `pmemsave`. But as of February 2014 the Android SDK Emulator does not support QEMU Monitor anymore [7]. Instead one could compile the Android kernel for ARM's Versatile Express platform [12] and have it executed directly by QEMU which will not be further discussed in this document.

In this section we create an AVD and compile a LKM enabled Goldfish kernel with which to run the AVD. Lastly we adapt the steps from the former Linux experiments.

All software required for this experiment such as `android`, `emulator` and `adb` are included in the Android SDK Tools [8]. For convenience we add the location of these commands to the `$PATH` variable. Furthermore the Android Open Source Project (AOSP) provides the Goldfish kernel sources [9] and the correct toolchain for cross compiling to the ARM processor architecture [6].

### 3.3.1 Compile Goldfish kernel

First we create the AVD with the command `android create avd` (cmp. fig. 3.21) and start it with `emulator`. A look at `/proc/version` then presents the exact point of development of the kernel which is in this case the commit ID `9ac497f`. The compiler used was `gcc version 4.7 (GCC)`.

---

<sup>1</sup>The emulated Android hardware's codename is "Goldfish".

```

$ echo no | android create avd -n test -t 'android-21' -b 'default/armeabi-v7a' -c 2G
-f
Android 5.0.1 is a basic Android platform.
Do you wish to create a custom hardware profile [no]
Created AVD 'test' based on Android 5.0.1, ARM (armeabi-v7a) processor, with the
following hardware config:
hw.cpu.model=cortex-a8
hw.lcd.density=240
hw.ramSize=512
hw.sdCard=yes
vm.heapSize=48
$ emulator -avd test &
$ adb shell cat /proc/version
Linux version 3.4.67-01413-g9ac497f (ghackmann@ghackmann.mtv.corp.google.com) (gcc
version 4.7 (GCC) ) #1 PREEMPT Mon Jul 7 13:02:28 PDT 2014

```



Figure 3.21: Create Android Virtual Device (AVD)

```

$ mkdir -p ~/android/test-goldfish && cd $_
$ # >>>> get the configuration file from AVD
$ adb pull /proc/config.gz
121 KB/s (10361 bytes in 0.083s)
$ gunzip config.gz
$ # >>>> get the toolchain
$ git clone
https://android.googlesource.com/platform/prebuilts/gcc/linux-x86/arm/arm-eabi-4.7
<snip>
$ find . -name *-gcc
./arm-eabi-4.7/bin/arm-eabi-gcc
$ # >>>> get the kernel sources and checkout correct commit
$ git clone https://android.googlesource.com/kernel/goldfish.git
<snip>
$ cd ~/android/test-goldfish/goldfish
$ git checkout 9ac497f
<snip>
HEAD is now at 9ac497f... Merge branch 'android-3.4' into android-goldfish-3.4
$ # >>>> prepare some constants
$ export ARCH=arm
$ export SUBARCH=arm
$ export CROSS_COMPILE=~/.android/test-goldfish/arm-eabi-4.7/bin/arm-eabi-
$ export CoresPlus1=$((`grep -c processor /proc/cpuinfo` + 1))
$ # >>>> add loadable module support to config
$ make clean && make mrproper
$ cp ../config .config
$ make menuconfig
<snip>

```




Figure 3.22: AVD: Compile Goldfish kernel (1/3)



```

$ make modules_prepare
<snip>
$
$ # >>>> compile kernel
$ make -j$CoresPlus1
<snip>
SYSMAP System.map
<snip>
Kernel: arch/arm/boot/zImage is ready
$
$ # >>>> save System.map
$ cp System.map ../System.map
<snip>
$
$ # >>>> start AVD with new kernel
$ emulator -avd test -kernel ~/android/test-goldfish/goldfish/arch/arm/boot/zImage &

```

Figure 3.24: AVD: Compile Goldfish kernel (3/3)

This is enough knowledge to compile a customized Goldfish kernel. From the running AVD we gather the configuration file. We download the toolchain [6] and kernel [9] and checkout the correct git commit from the latter. With `make menuconfig` we add the ability to load and unload modules to the `.config`. Then we `make` the new kernel and finally start the AVD with it. The whole process is shown in figs. 3.22 to 3.24.

### 3.3.2 Compile LiME and Volatility for Goldfish kernel

Cross compiling LiME and Volatility has been done for Fedora GNU/Linux before (cmp. fig. 3.19) but now we have not only to consider a foreign kernel but also an alternative processor architecture. Thus each of the `Makefile`s has to become quite modified in order to find the kernel sources and the toolchain. (See listings 3.1 to 3.2.) The previously created `.config` file is used again. For creating the Volatility profile the `System.map` generated during the kernel's compilation process has been put aside.

Both modules can now be compiled and the Volatility profile created as depicted in fig. 3.25.

For memory acquisition the LiME module is pushed over the ADB to the AVD. The command `insmod` activates the module (cmp. fig. 3.26) and the dump is transferred over TCP to the waiting forensic workstation where it can be examined with Volatility (cmp. fig. 3.27).

Listing 3.1: AVD/Makefile.LiME.cross

```

1  obj-m := lime.o
2  lime-objs := tcp.o disk.o main.o
3
4  KDIR := ~/android/test-goldfish/goldfish/
5  KVER := goldfish
6
7  PWD := $(shell pwd)
8  CCPATH := ~/android/test-goldfish/arm-eabi-4.6/bin
9
10 default:
11     $(MAKE) -C $(KDIR) M=$(PWD) modules
12     $(CCPATH)/arm-eabi-strip --strip-unneeded lime.ko
13     mv lime.ko lime-$(KVER).ko
14
15     $(MAKE) tidy
16
17 tidy:
18     rm -f *.o *.mod.c Module.symvers Module.markers modules.order \*.o.cmd \
19         \*.ko.cmd \*.o.d
20     rm -rf \.tmp_versions
21
22 clean:
23     $(MAKE) tidy
24     rm -f *.ko

```

Listing 3.2: AVD/Makefile.Volatility.cross

```

1  obj-m += module.o
2
3  KDIR := ~/android/test-goldfish/goldfish/
4  CCPATH := ~/android/test-goldfish/arm-eabi-4.6/bin
5
6  -include version.mk
7
8  all: dwarf
9
10 dwarf: module.c
11     $(MAKE) ARCH=arm CROSS_COMPILE=$(CCPATH)/arm-eabi- -C $(KDIR) \
12         CONFIG_DEBUG_INFO=y M=$(PWD) modules
13     dwarfdump -di module.ko > module.dwarf

```

```
$ # >>>> compile LiME loadable module
$ cd ~/android/test-goldfish/
$ git clone https://github.com/504ensicsLabs/LiME.git
<snip>
$ cp Makefile.LiME.cross LiME/src/Makefile
$ cd LiME/src
$ make clean && make
<snip>
$ mv lime.ko lime-goldfish.ko
cp lime-goldfish.ko ../../
$ # >>>> create Volatility profile
$ cd ~/android/test-goldfish/
$ git clone https://github.com/volatilityfoundation/volatility.git
<snip>
$ cp Makefile.Volatility.cross volatility/tools/linux/Makefile
$ cd volatility/tools/linux/
$ make
<snip>
dwarfdump -di module.ko > module.dwarf
$ zip -j Android_Goldfish_3.4.67-01413-g9ac497f.zip module.dwarf ../../System.map
adding: module.dwarf (deflated 90%)
adding: System.map (deflated 74%)
$ cp Android_Goldfish_3.4.67-01413-g9ac497f.zip ../../..
$ cp Android_Goldfish_3.4.67-01413-g9ac497f.zip
  ../../volatility/plugins/overlays/linux/
```




Figure 3.25: AVD: Compile LiME and create Volatility profile

```
$ cd ~/android/test-goldfish/
$ adb push lime-goldfish.ko /sdcard/lime.ko
41 KB/s (5268 bytes in 0.123s)
$ adb forward tcp:4444 tcp:4444
$ adb shell insmod /sdcard/lime.ko "path=tcp:4444 format=lime" &
$ nc localhost 4444 > goldfish.lime
$ ls -lgGh goldfish.lime
-rw-r--r-- 1 513M Feb  8 17:20 goldfish.lime
```



Figure 3.26: AVD: LiME TCP transfer



```

$ cd ~/android/test-goldfish/volatility/
$ export
  VOLATILITY_LOCATION=file:///home/hotblack/android/test-goldfish/goldfish.lime
$ export VOLATILITY_PROFILE=LinuxAndroid_Goldfish_3_4_67-01413-g9ac497fARM
$ python vol.py linux_pslist
Volatility Foundation Volatility Framework 2.4
Offset      Name                Pid  Uid  Gid  DTB      StartTime
0xde81bc00  init                  1  0    0    0x1ebc4000 2015-02-08 16:09:23 UTC+0000
0xde81b800  kthreadd              2  0    0    -0x1      2015-02-08 16:09:23 UTC+0000
0xde81b400  ksoftirqd/0          3  0    0    -0x1      2015-02-08 16:09:23 UTC+0000
<snip>
0xd100ac00  ndroid.exchange       1104 10027 10027 0x9918000 2015-02-08 16:14:39 UTC+0000
0xc98c8000  sh                    1135 0     0    0x9a28000 2015-02-08 16:15:02 UTC+0000
0xc99ff800  insmod                1137 0     0    0x9a40000 2015-02-08 16:15:02 UTC+0000

```



Figure 3.27: AVD: Start Volatility

### 3.3.3 Recapitulation

Each time a copy of volatile memory is to be acquired, there are several prerequisites to be fulfilled. We only had a few little of them with Windows (cmp. section 3.1) and a lot more with Linux (cmp. section 3.2). In this section with a virtualized Android the complexity is again increased.

On the target system we require:

- access to a shell via ADB
- root privileges
- to create a LKM enabled kernel

Additionally we need an Android development system for cross compiling the LiME and Volatility modules with:

- Android SDK Tools [8]
- Android kernel sources for the specific hardware platform and Android version; for the latter the exact same git commit as stated by `/proc/version` is needed
- toolchain for cross compiling to the ARM processor architecture
- compiler `.config` settings from the target's `/proc/config.gz`
- symbol table `System.map` generated by kernel compilation
- `dwarfdump`

### 3.4 Conclusion

We acquired the RAM of diverse virtualized systems: one Windows 7 VM, five GNU/Linux VMs, and one AVD. From these experiments we gained essentially three insights:

1. Complexity in terms of time, effort, and required skills increases from Windows to Linux to Android.
2. Actual real, “live” memory capture isn’t possible for Linux kernel based systems.
3. Know your tools!

We document the rising complexity by comparing selected experiment details (see also table 3.2<sup>2</sup>).

For all cases escalated privileges were required in order to access memory. While Windows investigators can resort to pre-built binaries, specialists for Linux based systems have to identify the explicit version and configuration of the target system before they can tailor a proper LiME module. The time necessary for that defeats a “live” memory capture in terms of “immediately after seizure”.

**Table 3.2:** Comparison of the VM experiments

Process step	Windows	GNU/Linux	Android
<b>Common aspects</b>			
Detailed knowledge about the exact OS version and configuration is required	○	●	●
Modules have to be compiled against the target’s kernel	○	●	●
Cross compilation always necessary	○	○	●
Find compatible compiler version	○	●	●
<b>Memory acquisition</b>			
Software has to be tailored for target	○	●	●
Root permissions required	●	●	●
<b>Memory analysis</b>			
Volatility profiles have to be tailored	○	●	●

The same is valid for Volatility profiles. Volatility comes with almost all Windows profiles out-of-the-box and even does a good job at proposing the correct Windows version just from the memory dump. Due to the vast number of possible Linux kernel versions and configurations the examiner has to create Volatility profiles for every newly explored Linux or Android kernel.

Regarding the examined virtualized systems, Android (with LKM support) only differs from Linux by the imperative to cross compile. The following chapter will show how real Android smartphones differ in further aspects.

<sup>2</sup>See table 4.7 at the conclusion of the following chapter for an expanded version which includes Android smartphones.

An issue which must not be underestimated is the ability to troubleshoot the utilized OSS. There is no right to prompt support or bug-fixes. If you work under time pressure you have to support yourself. In fact the use of community driven software includes some moral commitment to play a part in the respective project.



# 4

## REAL MACHINES – ANDROID SMARTPHONES

---

The preceding chapter traced an arc from the straightforward memory analysis of a Windows system across more complex explorations involving several GNU/Linux distributions and culminated in an analysis using an Android Virtual Device. Common to all of these systems was that they were virtualized and employed unhardened standard installations. Thus full root access was available by default. Now we turn towards real Android smartphones which vary in many aspects from the previously examined virtual Goldfish device.

The selection process was primarily subject to budget restrictions. We used our own phones, attractive offers from online auctions and classifieds as well as a borrowed device. The mixture (cmp. table 4.1) is interesting in that we have a balanced range of Android versions. The age of the devices varies from an ancient 2009 HTC Magic to a cutting-edge 2015 Sony Xperia Z3. The midfield mainly is made up of Samsung Galaxy mainstream devices which are still popular and widely used as of this writing.

Google's Nexus devices are not only popular among security researchers. This is because they are easy to unlock and root which is prior condition for many tasks. Hence we purchased a Nexus S (see fig. 4.1) to commence our examination of real devices.

We were not able to perform a completely successful, realistic forensic live memory research with any of our phones with respect to a) root without reboot, b) acquiring RAM with LiME, and c) have Volatility handle the dump. Each device presented its own obstacles which we depict in this chapter.

**Table 4.1:** Tested Android devices

Manufacturer Model	Android	Kernel	Build
<b>HTC</b>			
Magic	2.2.1	2.6.32.9-27237-gbe746fb	FRG83D
<b>Google (Samsung)</b>			
Nexus S (GT-I9023)	2.3.6	2.6.35.7-gf5f63ef	GRK39F
	4.0.4	3.0.8-g6656123	IMM76D
	4.1.2	3.0.31-g5894150	JZO54K
<b>Samsung</b>			
S III (GT-I9300)	4.3	3.0.31-2713958	JSS15J.I9300XXUGNG3
S III LTE (GT-I9305)	4.2.2 CyanogenMod 10.1.3-i9305	3.0.64-CM-g9c2e2bc	cm_i9305-userdebug 4.2.2 JDQ39E eng.jenkins.20130923 .145024 test-keys
S III mini (GT-I8190)	4.1.2	3.0.31-1332988	JZO54K.I8190XXAMG4
S III mini (GT-I8200N)	4.2.2	3.4.5-2818574	JDQ39.I8200NXXUAOA1
<b>Sony Ericsson</b>			
Xperia Mini Pro (SK17i)	4.0.4	2.6.32.9-perf	4.1.B.0.587
<b>Sony</b>			
Xperia Z3 (D6603)	4.4.4	3.4.0-perf-g1b1963a- 02930-g23f7791	23.0.1.A.5.77

**Figure 4.1:** Google Nexus S (Samsung GT-I9023): After three HTC devices (the G1, myTouch and Nexus One) the Nexus S was the first Google designed Android phone manufactured by Samsung Electronics. It was released in December 2010 and introduced Android 2.3 “Gingerbread” to the market. The most recent official update was Android 4.1.2 “Jelly Bean” in October 2012.



## 4.1 Google Nexus S: Loss of volatile memory

In this section we prepare a Google Nexus S with the latest official factory image Android 4.1.2 “Jelly Bean”. While unlocking and rooting the user’s data is deleted. We did not find a solution for rooting the booted phone without the need to also reboot. We acquire the phone’s memory, demonstrate successful Volatility usage and finally have a look at the differences compared to the previous AVD experiment.

When working with Android devices one is confronted with multiple code names. The Google Nexus S (see fig. 4.1) was manufactured by Samsung under the model name “GT-I9023”. Google’s code name for the model is “soju” which is needed to identify the correct factory images (cmp. section 4.1.2). As a minor consequence of Android device diversification there are also derived models “sojua” (850MHz version, i9020a), “sojuk” (Korea version, m200), and “sojus” Nexus S 4G (d720). The device’s code name is “crespo” which is later referenced in the recovery image (cmp. section 4.1.1). According to the “sojus” model there is also a device with code name “crespo4g”. The mainboard is referenced under the code name “herring”. All this information and more about the target device can be gathered from the file `/system/build.prop` (see fig. 4.2).

```
$ adb shell "cat /system/build.prop" | grep product
ro.product.model=Nexus S
ro.product.brand=google
ro.product.name=soju
ro.product.device=crespo
ro.product.board=herring
ro.product.cpu.abi=armeabi-v7a
ro.product.cpu.abi2=armeabi
ro.product.manufacturer=samsung
ro.product.locale.language=en
ro.product.locale.region=US
<snip>
```



Figure 4.2: Nexus S: `build.prop`

### 4.1.1 Unlock boot loader

Before anything like flashing a ROM or rooting can be tackled, the device’s boot loader (section 2.3.1) first needs to be unlocked.

If the Nexus S is powered off it can be booted into boot loader mode by pressing the volume up and power buttons simultaneously. Now the boot loader can be unlocked with the command `fastboot oem unlock`<sup>1</sup> sent from an USB connected computer (see fig. 4.3). A security question on the phone needs confirmation, warning that a “factory data reset” wipes all “personal data” from the phone (see figs. 4.6a to 4.6c).

<sup>1</sup> `fastboot` is part of the Android SDK.

```
$ fastboot oem unlock
...
OKAY [ 20.267s]
finished. total time: 20.270s
```



**Figure 4.3:** Nexus S: Unlock boot loader

At this point latest it becomes obvious that this common way of rooting is not adequate for volatile memory acquisition. First, the volatile memory is erased by switching off the phone. Second, all additional non-volatile personal artifacts are deleted as a matter of principle. Nevertheless, we tread this path in order to draw the connecting line from an AVD to real world devices.

### 4.1.2 Flash factory image

Google offers three “Factory Images ‘soju’ for Nexus S (worldwide version, i9020t and i9023)” [28]:

2.3.6 (GRK39F) Gingerbread

4.0.4 (IMM76D) Ice Cream Sandwich

4.1.2 (JZO54K) Jelly Bean

With the phone in boot loader mode the factory image of choice can be flashed by calling a single included script `flash-all.sh` (cmp. fig. 4.4).

```
$ wget https://dl.google.com/dl/android/aosp/soju-jzo54k-factory-36602333.tgz
<snip>
$ tar -zxvf soju-jzo54k-factory-36602333.tgz
<snip>
$ cd soju-jzo54k
$ ./flash-all.sh
<snip>
```



**Figure 4.4:** Nexus S: Flash factory image

At this stage the phone is completely reset. The user is greeted by the Android welcome screen and has to perform an initial configuration such as choosing a language, timezone, etc.


### 4.1.3 Flash custom recovery and root device

The goal is to install the application SuperSU by XDA developer Chainfire [15, 16]. In order to be able to sideload SuperSU on the Nexus S a custom recovery image is needed



because the default one (see fig. 4.6d) does not offer this. The recovery image of choice is Team Win Recovery Project (TWRP) [66, 67] which is installed with `fastboot`. With TWRP SuperSU can be sideloaded by selecting this option on the phone and finally starting the installation from ADB (see figs. 4.6e to 4.6i). A reboot completes the process. (For the commands see fig. 4.5.)

```
$ wget http://techerrata.com/file/twrp2/crespo/openrecovery-twrp-2.8.5.0-crespo.img
<snip>
$ fastboot flash recovery openrecovery-twrp-2.8.5.0-crespo.img
sending 'recovery' (5872 KB)...
OKAY [ 0.790s]
writing 'recovery'...
OKAY [ 0.908s]
finished. total time: 1.701s
$ wget http://download.chainfire.eu/696/SuperSU/UPDATE-SuperSU-v2.46.zip
<snip>
$ adb sideload UPDATE-SuperSU-v2.46.zip
Total xfer: 1.41x
```



**Figure 4.5:** Nexus S: Unlock boot loader; install TWRP2 and SuperSU

#### 4.1.4 Compile LiME and Volatility modules

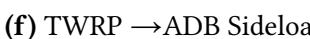
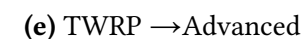
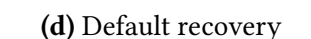
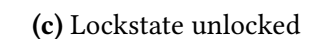
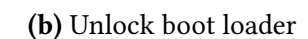
Up to now we unlocked the Nexus S boot loader, flashed a fresh factory image to the phone, flashed a custom recovery image and escalated permissions to root by sideloading SuperSU. That gives us a device with which we can test memory acquisition. A few details about our target are disclosed at “Settings” → “About phone”:

Model number	Nexus S
Android version	4.1.2
Baseband version	I9023XXKI1
Kernel version	3.0.31-g5894150 android-build@vpbs1 ) #1 PREEMPT Mon Sep 10 14:10:13 PDT 2012
Build number	JZO54K

The kernel version’s last seven digits `5894150` are the commit ID of the kernel sources. We will need this later in this section to checkout the kernel from GitHub.

In order to get info from and transfer the LiME kernel module to the phone we need to enable “USB debugging” (found in “System settings” → “Developer options”). While this is the standard form of communication with an AVD, a real Android device forces the user to confirm two security questions (“Allow development settings?”, “Allow USB debugging?”) underlining the security weakening impact of this action.

With `adb shell` we now can get additional information from `/proc/version` such as the originally used compiler (see fig. 4.7).



54

```
$ adb shell cat /proc/version
Linux version 3.0.31-g5894150 (android-build@vpbs1.mtv.corp.google.com) (gcc version
4.6.x-google 20120106 (prerelease) (GCC) ) #1 PREEMPT Mon Sep 10 14:10:13 PDT 2011
```

Figure 4.7: Nexus S: `/proc/version`

At this point we are missing two important files which were available in the Linux experiments (3.2) and which we created during Goldfish kernel compilation for the AVD experiment (3.3): the kernel `.config` and the `System.map`. They were both not available in any of the Android devices we examined. Thus we are forced to compile a kernel exactly the same way as our target device's kernel was built in order to obtain a correct `System.map`. Before we can compile the kernel, the LiME module, and the Volatility module, we have to `make` the correct kernel configuration used by the manufacturer.

The URIs to the kernel sources for all Nexus devices are listed at the AOSP [9]. In table 4.2 the Nexus S / Crespo information is shown.

Table 4.2: Nexus S: Figuring out which kernel to build (excerpt from [9])

Device	Binary location	Source location	Build configuration
...	...	...	...
crespo	device/samsung/crespo/kernel	kernel/samsung	herring_defconfig
...	...	...	...

Here the locations are always prefixed by `https://android.googlesource.com/`. We download the Samsung kernel tree [4] with `git clone`. From `/proc/version` we know that a gcc version 4.6 compiler has to be used. We download the appropriate prebuilt toolchain from the AOSP [5]. After adding the compiler to the `$PATH`, we check that the version is identical to `/proc/version`. Finally the `.config` has to be built from `herring_defconfig` and the kernel can be compiled. As a by-product the `System.map` is created which we put aside. All these commands are shown in fig. 4.8.

Similar to the process depicted for the Goldfish device in section 3.3.2 both the `Makefile`s of LiME and Volatility have to be adapted to suit the planned cross compiling. That is, the paths to kernel sources and toolchain have to be edited. Both `Makefile`s can be reviewed in listings 4.1 to 4.2.

Compilation of both the LiME and Volatility module as well as creation of the Volatility profile do not vary significantly from the AVD process (3.3.2). The corresponding bash interaction is documented in fig. 4.9.

### 4.1.5 RAM acquisition and analysis

When it comes to RAM acquisition the LiME module is pushed over the ADB to the Nexus S. Now we experience a constraint due to an Android security feature we already

```

$ # >>>> download prebuilt toolchain and check compiler version
$ git clone
    https://android.googlesource.com/platform/prebuilts/gcc/linux-x86/arm/arm-eabi-4.6

<snip>
$ export PATH=/media/ultrabay/thesis/nexuss/arm-eabi-4.6/bin/:$PATH
$
$ arm-eabi-gcc --version
arm-eabi-gcc (GCC) 4.6.x-google 20120106 (prerelease)
<snip>
$ # >>>> download Samsung kernel sources and checkout correct commit
$ git clone https://android.googlesource.com/kernel/samsung
<snip>
$ cd samsung
$ git checkout 5894150
<snip>
$ # >>>> prepare environment variables for kernel build
$ export ARCH=arm
$ export SUBARCH=arm
$ export CROSS_COMPILE=arm-eabi-
$ make herring_defconfig
<snip>
$ export CoresPlus1=$((`grep -c processor /proc/cpuinfo` + 1))
$ make -j$CoresPlus1
<snip>
$ cp System.map ..
$ cd ..

```




Figure 4.8: Nexus S: Downloads for kernel compilation

## Listing 4.1: NexusS/Makefile.Volatility.cross

```

1  obj-m += module.o
2
3  KDIR := /media/ultrabay/thesis/nexuss/samsung/
4  CCPATH := /media/ultrabay/thesis/nexuss/arm-eabi-4.6/bin
5
6  -include version.mk
7
8  all: dwarf
9
10 dwarf: module.c
11     $(MAKE) ARCH=arm CROSS_COMPILE=$(CCPATH)/arm-eabi- -C $(KDIR) \
        CONFIG_DEBUG_INFO=y M=$(PWD) modules
12     dwarfdump -di module.ko > module.dwarf

```

Listing 4.2: NexusS/Makefile.LiME.cross

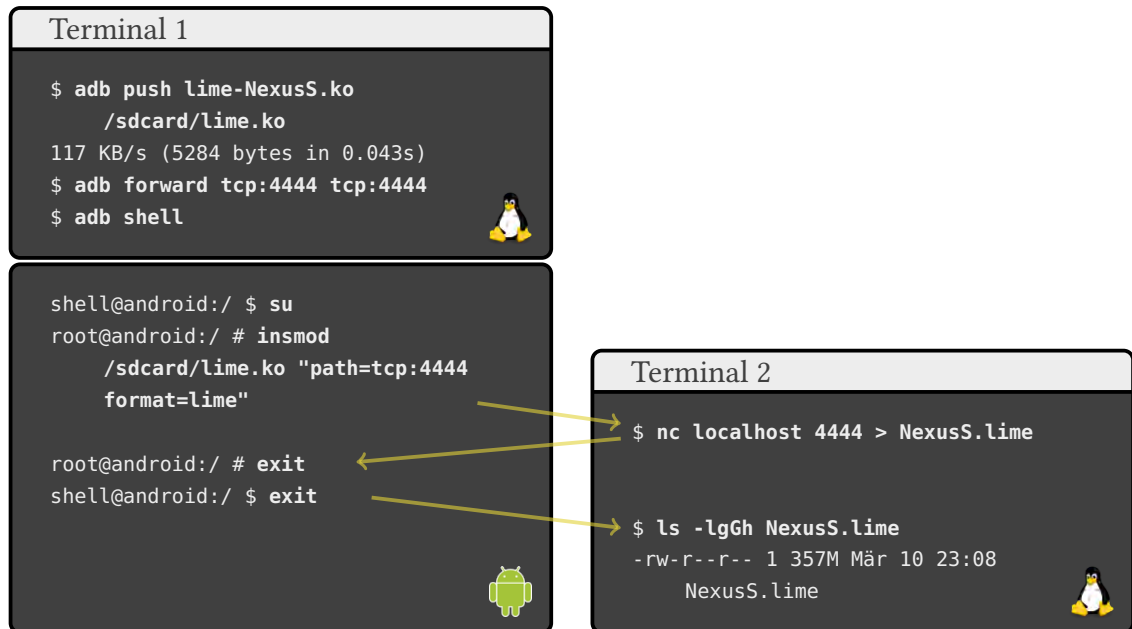
```
1  obj-m := lime.o
2  lime-objs := tcp.o disk.o main.o
3  KDIR := /media/ultrabay/thesis/nexuss/samsung/
4  KVER := NexusS
5  PWD := $(shell pwd)
6  CCPATH := /media/ultrabay/thesis/nexuss/arm-eabi-4.6/bin
7
8  default:
9      $(MAKE) -C $(KDIR) M=$(PWD) modules
10     $(CCPATH)/arm-eabi-strip --strip-unneeded lime.ko
11     mv lime.ko lime-$(KVER).ko
12
13     $(MAKE) tidy
14
15 tidy:
16     rm -f *.o *.mod.c Module.symvers Module.markers modules.order \*.o.cmd \
17         \*.ko.cmd \*.o.d
18     rm -rf \.tmp_versions
19
20 clean:
21     $(MAKE) tidy
22     rm -f *.ko
```

```
$ # >>>> compile LiME loadable module
$ git clone https://github.com/504ensicsLabs/LiME.git
<snip>
$ cp Makefile.LiME.cross LiME/src/Makefile
$ cd LiME/src
$ make clean && make
<snip>
$ cp lime-NexusS.ko ../../
$ cd ../../
$ # >>>> create Volatility profile
$ git clone https://github.com/volatilityfoundation/volatility.git
<snip>
$ cp Makefile.Volatility.cross volatility/tools/linux/Makefile
$ cd volatility/tools/linux/
$ make
<snip>
$ zip -j Android_NexusS_3.0.31-g5894150.zip module.dwarf ../../../../System.map
<snip>
$ cp Android_NexusS_3.0.31-g5894150.zip ../../../../
$ cp Android_NexusS_3.0.31-g5894150.zip ../../../../volatility/plugins/overlays/linux/
$ cd ../../../../
```



Figure 4.9: Nexus S: Compile LiME and create Volatility profile

predicted in section 2.3.2. Other than with the AVD we cannot send the `insmod` command directly from our Linux bash. We don't have root permissions on the target. Instead we log on to the Nexus S with `adb shell` and start the LiME LKM interactively on the smartphone. Hence we need a second Linux terminal in order to control the data transfer over TCP. See fig. 4.10 for an visualization of the two terminals' interplay.



**Figure 4.10:** Nexus S: LiME TCP transfer

Due to the previously created profile Volatility is finally able to analyze the just acquired Nexus S memory dump. Figure 4.11 shows the execution of the `linux_pslist` plugin.

```
$ cd volatility
$ export VOLATILITY_LOCATION=file:///media/ultrabay/thesis/nexuss/NexusS.lime
$ export VOLATILITY_PROFILE=LinuxAndroid_NexusS_3_0_31-g5894150ARM
$ python vol.py linux_pslist
Volatility Foundation Volatility Framework 2.4
Offset      Name                Pid  Uid  Gid  DTB          StartTime
0xe7824000  init                 1  0    0    0x57058000  2015-03-10 21:51:09 UTC+0000
0xe7824380  kthreadd             2  0    0           -0x1 2015-03-10 21:51:09 UTC+0000
0xe7824700  ksoftirqd/0         3  0    0           -0x1 2015-03-10 21:51:09 UTC+0000
<snip>
0xe0ec0000  kworker/0:0          1355 0    0           -0x1 2015-03-10 22:03:04 UTC+0000
0xe0ec1500  kworker/u:0          1359 0    0           -0x1 2015-03-10 22:05:27 UTC+0000
0xe0ec1c00  sh                   1364 0    0    0x318fc000  2015-03-10 22:05:46 UTC+0000
```

**Figure 4.11:** Nexus S: List processes with Volatility

In order to perform a “realistic” memory acquisition we would have had to lock the boot loader with `fastboot oem lock` after having flashed the stock ROM in section 4.1.2. For

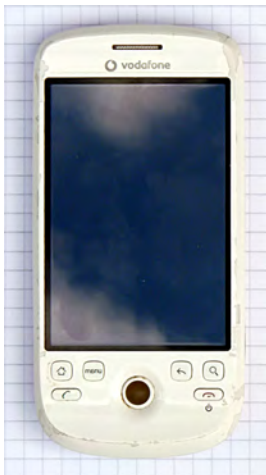
this setup we did not find any way to root the phone without loss of the volatile memory. The literature discusses FROST [31, 47] (cmp. section 2.3.3) which could be used here to acquire the RAM. But the need to unlock the device would destroy/wipe the otherwise persistent memory.

In the following section we will demonstrate rooting a locked smartphone without the need to reboot.

## 4.2 HTC Magic: Root without loss of volatile memory

In this section we demonstrate a memory preserving rooting method.

The phone in fig. 4.12 is a HTC Magic 32B. Several versions of this smartphone were build. We can identify the exact model by its appearance: The key characteristics are the white color with a silver trim, the lack of a 3.5mm TRS jack for headphones and the Vodafone logo.



**Figure 4.12:** HTC Magic 32B: The HTC Magic from 2009 is the successor of the HTC Dream and hence the second commercially released Android phone. Two different hardware platforms 32A and 32B exist for this phone. They vary in the included RAM and processor as well as the need for different boot images and wireless LAN kernel modules. The most recent official update was Android 2.2.1 “Froyo” in December 2010.

No lock screen was activated when we bought the phone. Thus we can look at “Settings” → “About phone” for a few details about our target:

Model number	HTC Magic
Android version	2.2.1
Baseband version	62.50SJ.20.17U_2.22.28.25
Kernel version	2.6.32.9-27237-gbe746fb android-build@apa26 #1
Build number	FRG83D

After enabling USB debugging we get more hardware details from `/system/build.prop` while `/proc/version` discloses the compiler version (fig. 4.13).

Even though the Android version 2.2.1 was officially distributed over the air (OTA) the HTC developer website [32] does not provide the kernel sources. Only the two preceding



```

$ adb shell "cat /system/build.prop" | grep product
ro.product.model=HTC Magic
ro.product.brand=vodafone
ro.product.name=vfpioneer
ro.product.device=sapphire
ro.product.board=sapphire
ro.product.cpu.abi=armeabi
ro.product.manufacturer=HTC
ro.product.locale.language=en
ro.product.locale.region=GB
# ro.build.product is obsolete; use ro.product.device
ro.build.product=sapphire
$ adb shell cat /proc/version
Linux version 2.6.32.9-27237-gbe746fb (android-build@apa26.mtv.corp.google.com) (gcc
version 4.4.0 (GCC) ) #1 PREEMPT Thu Jul 22 15:50:12 PDT 2010

```



**Figure 4.13:** HTC Magic: `build.prop` and `/proc/version`

Android versions are listed for the HTC Magic (cmp. table 4.3). We found rumors about the HTC Magic 32B hardware platform being identical with the predecessor HTC Dream. But HTC only lists kernel sources for the same Android versions as for the HTC Magic (cmp. table 4.4). Thus we abandoned the plan to acquire a RAM dump.

**Table 4.3:** HTC Magic: Available kernel sources at htcdev.com

Device	Carrier	Region	Type	Kernel	Android	Size	Description
Magic	Rogers	CA	CRC	2.6.29	v2.1	73.4 MB	
Magic	N/A	N/A	N/A	2.6.27	v1.5	49.5 MB	

**Table 4.4:** HTC Dream: Available kernel sources at htcdev.com

Device	Carrier	Region	Type	Kernel	Android	Size	Description
Dream	Sprint	WWE	MR	2.6.29	v2.1	74.0 MB	
Dream	Sprint	WWE	CRC	2.6.27	v1.5	49.7 MB	
Dream	N/A	N/A	N/A	2.6.27	v1.5	49.75 MB	


Nevertheless, the HTC Magic is ideal for demonstrating Sebastian Krahmer’s “Rage Against The Cage” exploit from August 2010 [41].

The exploit is about local root privilege escalation. Actually privileges are not really escalated but their successful drop is prevented. The Android Debug Bridge daemon (adb) (cmp. section 2.3.1) starts with root privileges. After some initialization, it drops its privileges with the `setuid()` method. A failure could have been recognized by the `setuid()` return value but Android 2.2.1 doesn’t check this. Here is the exploit’s attack vector.




```
$ wget http://stealth.openwall.net/xSports/RageAgainstTheCage.tgz
<snip>
$ tar zxvf RageAgainstTheCage.tgz
<snip>
$ cd RageAgainstTheCage
$ adb devices
List of devices attached
HT973KF02465    device


$ adb shell
```



```
$ exit
```




```
$ adb push rageagainstthecage-arm5.bin /data/local/tmp/rageagainstthecage
97 KB/s (5392 bytes in 0.054s)
$ adb shell
```




```
$ cd /data/local/tmp
$ ./rageagainstthecage
[*] CVE-2010-EASY Android local root exploit (C) 2010 by 743C


[*] checking NPROC limit ...
[+] RLIMIT_NPROC={878, 878}
[*] Searching for adb ...
[+] Found adb as PID 64
[*] Spawning children. Dont type anything and wait for reset!
[*]
[*] If you like what we are doing you can send us PayPal money to
[*] 7-4-3-C@web.de so we can compensate time, effort and HW costs.
[*] If you are a company and feel like you profit from our work,
[*] we also accept donations > 1000 USD!
[*]
[*] adb connection will be reset. restart adb server on desktop and re-login.
```



```
$ adb kill-server
$ adb shell
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
```



```
#
```



**Figure 4.14:** HTC Magic: Gain temporary root permissions with the Rage-Against-The-Cage-Exploit. In the beginning the '\$' prompt indicates normal shell user privileges (red circle). After the exploitation the '#' prompt indicates a root shell (green circle).

Android’s kernel defines a resource limit `RLIMIT_NPROC` which is the maximum number of threads that can be created for the user ID of the calling process. The exploit forks off the `adbd` process until `RLIMIT_NPROC` is reached and further forking fails. Although each fork exits immediately, they still exist as Zombie processes for a certain time and count towards `RLIMIT_NPROC`.

When `fork()` fails, the exploit kills the ADB daemon which in turn restarts. Again initial privileges are on root level and `setuid()` gets called. Now `setuid()` tries both to decrement the number of root processes and to increment the count of shell user processes. It fails because the `RLIMIT_NPROC` is already reached. As `adbd` does not check the return value, it keeps on running with root privileges.

In practice you have to download the exploit, copy it to the phone and execute it. Then kill the PC’s adb server and reconnect to the phone. The whole process is documented in fig. 4.14. No reboot is needed. We did not research the consequences of the brute-force-forking for the memory.

An important feature of Krahmer’s exploit is the availability of its source code. One can inspect the code and compile it in order to obtain a trustworthy binary. This is not always given as can be seen in the rooting method shown in the following section 4.3

## 4.3 Sony Xperia Mini Pro – Root and memory dump

With the Sony Xperia Mini Pro (see fig. 4.18) we almost can run through a complete forensic live analysis. Only the very last step fails as Volatility does not accept the memory dump.

### 4.3.1 Rooting with Eroot

As before the phone has no lock screen. We gather initial information about our target (“Settings” → “About phone”):

Model number	SK17i
Android version	4.0.4
Baseband version	8x55A-AAABQOAZM-203028G-77
Kernel version	2.6.32.9-perf BuildUser@BuildHost #1
Build number	4.1.B.0.587

After enabling USB debugging more information is available over the ADB (see fig. 4.15).

So far a crucial code name has not yet appeared: For compiling the kernel we find the compiler configuration under the code name “smultron”.

Our internet research for rooting tools led to a recommendation of “Eroot” in the XDA forum [48]. Eroot is a Windows binary and offered at several URLs. At least one of those

```
$ adb shell "cat /system/build.prop" | grep product
ro.build.product=SK17i
ro.product.brand=SEMC
ro.product.name=SK17i_1249-1887
ro.product.device=SK17i
##### Values from product package metadata #####
ro.semc.product.model=SK17i
ro.semc.product.name=Xperia mini pro
ro.semc.product.device=SK17
ro.product.model=SK17i
ro.product.board=
ro.product.cpu.abi=armeabi-v7a
ro.product.cpu.abi2=armeabi
ro.product.manufacturer=Sony Ericsson
ro.product.locale.language=en
ro.product.locale.region=GB
<snip>
$ adb shell cat /proc/version
Linux version 2.6.32.9-perf (BuildUser@BuildHost) (gcc version 4.4.3 (GCC) ) #1
PREEMPT Wed Jul 4 12:32:24 2012
```



**Figure 4.15:** Sony Xperia Mini Pro: `build.prop` and `/proc/version`

we tried was definitely malware. In the following we describe the version which is linked to from the XDA forum [22]. For execution we used a virtualized Windows 7.

The phone has to be prepared with two settings:

“Settings” → “Developer options” → “USB debugging” and

“Settings” → “Xperia” → “Connectivity” → “USB connection mode” → “Mass storage mode (MSC)”.

In order to make sure we have all necessary ADB drivers in place we install Sony’s “PC Companion” [59]. Having connected the phone over an USB cable, Eroot can be started.

Eroot presents a Chinese user interface. We show screenshots with a translation to English in fig. 4.16. Running the software without knowing Chinese is like flying blind and hence not confidence inspiring. On the one hand we have no idea how the software works and not least of all whether it is reliable. A basic level of trust is only given by the XDA forum’s reputation. A serious problem could arise if an expert witness had to justify the use of such tools in court.

However after a few seconds Eroot announces successful rooting. The Xperia has a new application logo “Superuser” (fig. 4.18). A quick test with an ADB shell validates the expected result (fig. 4.17).



**Figure 4.16:** Sony Xperia Mini Pro: Eroot screens (with translations Chinese to English)

### 4.3.2 Preparing memory acquisition

Sony is exemplary in offering the sources for their Android smartphones, even for older models. The appropriate files for the Xperia Mini Pro can be downloaded from Sony’s “Open source archive for build 4.1.B.0.587” [58].

Things are different for Android prebuilt toolchains. At the beginning of this section we learned that the gcc compiler in version 4.4.3 was used to build the kernel (see fig. 4.15).

```
$ adb shell
```

```
shell@android:/ $ su
shell@android:/ # exit
shell@android:/ $ exit
```

Figure 4.17: Sony Xperia Mini Pro: root shell

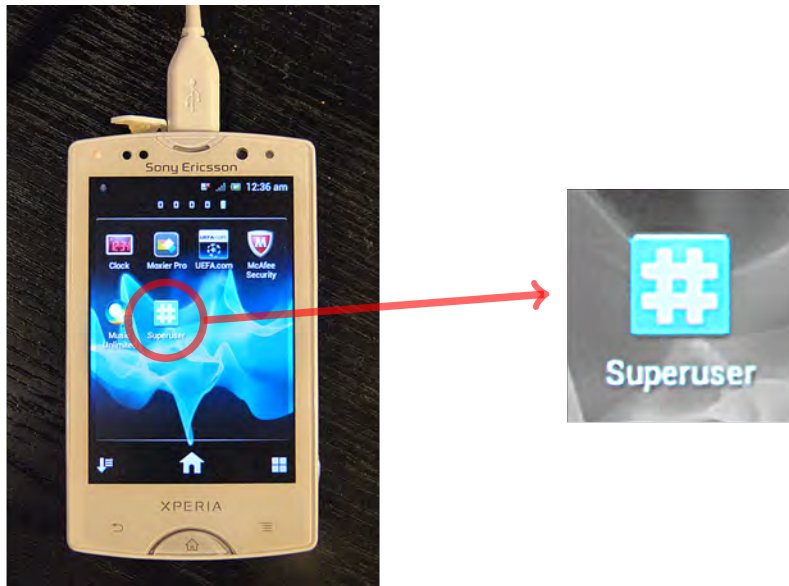


Figure 4.18: Sony Xperia Mini Pro: Superuser installed by Eroot

The AOSP does not keep an archive of deprecated toolchains. The oldest available as of this writing is version 4.6 which we used for the Nexus S kernel in section 4.1. If we try to use this toolchain all we achieve are errors. Internet research led us to XDA developer DooMLoRD’s git repository “android\_prebuilt\_toolchains” [18] which includes a “GCC 4.4.3 toolchain from CyanogenMod repo”.

Before the kernel can be compiled we have to find the correct `*defconfig` for the Xperia Mini Pro. None of the code names in the kernel’s `/arch/arm/configs` `*defconfig` files fits to the information we gathered so far. Another line of inquiry via the internet led us to the “SEMC Blog” [57], an unofficial Sony Ericsson blog. Here the “Model name: Xperia mini pro” is linked to the “Project name: SK17i” and the “Codename: Smultron”. The latter is used to name the `*defconfig` file.

With these information the kernel can be compiled as shown in fig. 4.19.

The `Makefile`s for LiME and Volatility as well as compiling their modules and creating the Volatility profile are straightforward. Hence we shifted the corresponding code to the annex listings 7.1 to 7.2.

```

$ # >>>> download prebuilt toolchain and check compiler version
$ git clone https://github.com/DooMLoRD/android_prebuilt_toolchains.git
<snip>
$ export PATH=/media/ultrabay/thesis/xperiamini/android_prebuilt_toolchains/arm-
  eabi-4.4.3/bin/:$PATH
$ arm-eabi-gcc --version
arm-eabi-gcc (GCC) 4.4.3
<snip>
$ # >>>> download Sony kernel sources
curl -O http://dl-developer.sonymobile.com/code/copylefts/4.1.B.0.587.tar.bz2
<snip>
$ tar -xjf 4.1.B.0.587.tar.bz2
$ # >>>> prepare environment variables for kernel build
$ export ARCH=arm
$ export SUBARCH=arm
$ export CROSS_COMPILE=arm-eabi-
$ # >>>> make .config
$ cd kernel
$ find . -type f -path '*/arm/configs*' -name '*smultron*'
./kernel/arch/arm/configs/semc_smultron_defconfig
$ make semc_smultron_defconfig
<snip>
$ # >>>> compile kernel
$ export CoresPlus1=$((grep -c processor /proc/cpuinfo + 1))
$ make -j$CoresPlus1
<snip>
$ cp System.map ..
$ cd ..

```




Figure 4.19: Sony Xperia Mini Pro: Kernel compilation

```

$ cd volatility
$ export VOLATILITY_LOCATION=file:///media/ultrabay/SonyXperiaMiniPro.lime
$ export VOLATILITY_PROFILE=LinuxAndroid_SonyXperiaMiniPro_2_6_32_9-perfARM
$ python vol.py linux_pslist
Volatility Foundation Volatility Framework 2.4
Offset      Name                               Pid      Uid      Gid      DTB      ...
-----
No suitable address space mapping found
Tried to open image as:
<snip>
LimeAddressSpace: lime: need base
<snip>
LimeAddressSpace: Invalid Lime header signature
<snip>

```




Figure 4.20: Sony Xperia Mini Pro: List processes with Volatility fails

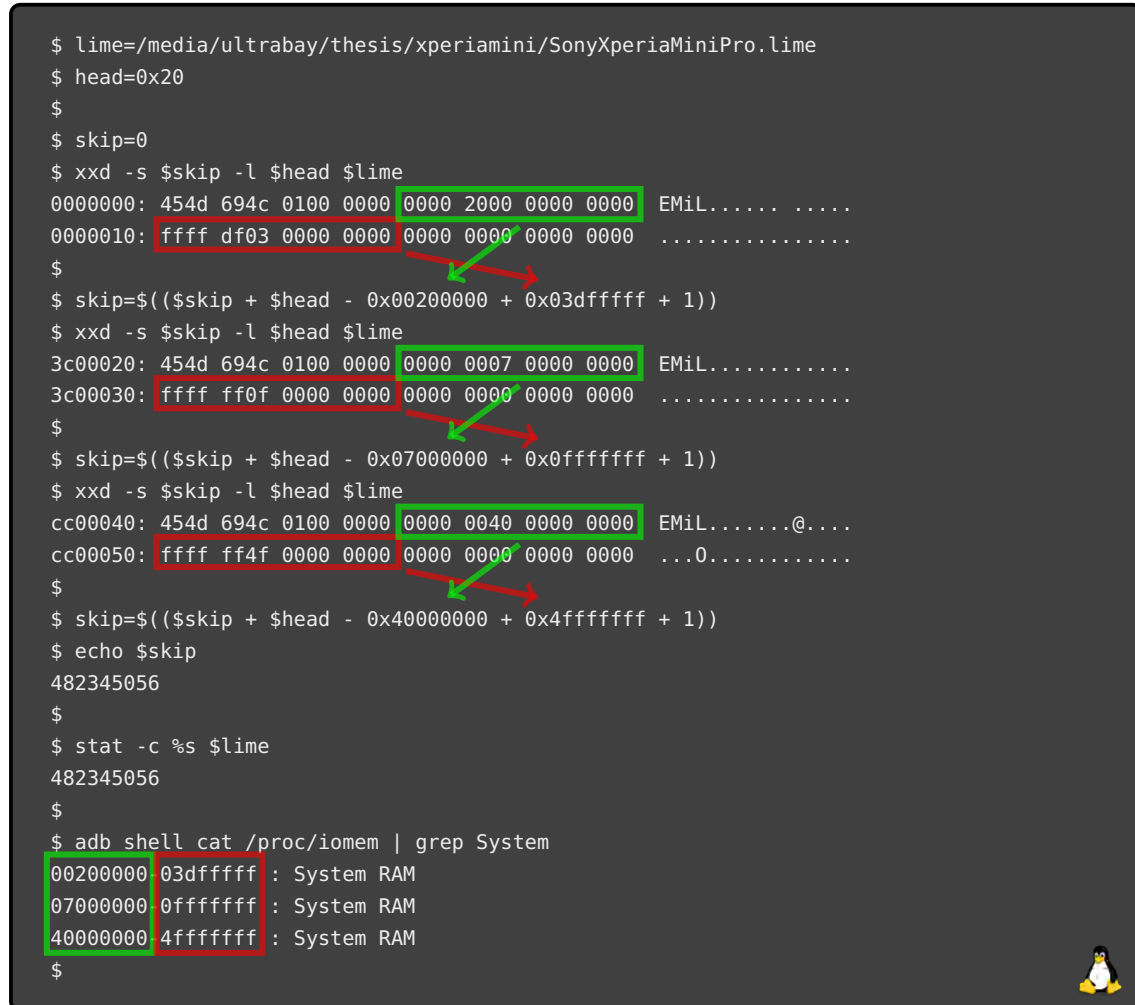
### 4.3.3 Memory acquisition and analysis

Except for one little variation we can dump the Xperia's memory just as before with the Nexus S (section 4.1.5). The commonly used path for pushing the LiME module `/sdcard/` is only accessible with root permission which the ADB doesn't have (cmp. section 2.3.2). Thus, we switch to `/data/local/tmp/` in order to plant the LKM on the phone. (Due to the similarity with the Nexus S process the corresponding shell commands can be looked-up in the annex fig. 7.2.)

```

$ lime=/media/ultrabay/thesis/xperiamini/SonyXperiaMiniPro.lime
$ head=0x20
$
$ skip=0
$ xxd -s $skip -l $head $lime
00000000: 454d 694c 0100 0000 0000 2000 0000 0000  EMiL.....
00000100: ffff df03 0000 0000 0000 0000 0000 0000  ....
$
$ skip=$(( $skip + $head - 0x00200000 + 0x03dfffff + 1))
$ xxd -s $skip -l $head $lime
3c000200: 454d 694c 0100 0000 0000 0007 0000 0000  EMiL.....
3c000300: ffff ff0f 0000 0000 0000 0000 0000 0000  ....
$
$ skip=$(( $skip + $head - 0x07000000 + 0x0ffffff + 1))
$ xxd -s $skip -l $head $lime
cc000400: 454d 694c 0100 0000 0000 0040 0000 0000  EMiL.....@....
cc000500: ffff ff4f 0000 0000 0000 0000 0000 0000  ...0.....
$
$ skip=$(( $skip + $head - 0x40000000 + 0x4ffffff + 1))
$ echo $skip
482345056
$
$ stat -c %s $lime
482345056
$
$ adb shell cat /proc/iomem | grep System
00200000 03dfffff : System RAM
07000000 0ffffff : System RAM
40000000 4ffffff : System RAM
$

```



**Figure 4.21:** Sony Xperia Mini Pro: Check memory dump segments

Finally Volatility's plugins should be able to work with the memory dump. Contrary to our expectations Volatility fails with an error message (fig. 4.20). Experience gained during this work has shown that this error message can have various reasons. We first ensure that the memory dump's format can be excluded as the source of the error. As we did before in section 2.2.1 we walk through the memory sections in the dump file and check the section headers. We compare the number of segments and their address ranges

with the corresponding `/proc/iomem` entries on the phone. This check (see fig. 4.21) leads us to believe that the dump’s contents are accurate.

At this point we end the investigation due to time constraints. The next step would be a check of the Volatility profile. Still, the toolchain—even if compiling the kernel and modules did not cause any concern—might differ from the one we actually needed. We also do not exclude the possibility that the compiler `.config` file differs from the one originally used by the manufacturer. Last but not least one could open an issue with the Volatility project on github.

## 4.4 Sony Xperia Z3: Too new to be targeted

Sony’s Xperia Z3 is the most up-to-date smartphone we examined. The information was copied from “Settings” → “About phone”:

Model number	D6603
Android version	4.4.4
Baseband version	8974-AAAAANAZQ-00022-21
Kernel version	3.4.0-perf-g1b1963a-02930-g23f7791 BuildUser@BuildHost #1 Tue Nov 25 11:03:01 2014
Build number	23.0.1.A.5.77

The only applicable rooting tool we found was *giefroot* [81]. It should root a Sony Xperia Z3 despite locked boot loader. Its description does not mention the need to reboot nor an obligatory loss of data. But it assumes “*Firmware < October 2014 (kernel and system)*” which is not the case here. With the given firmware build 23.0.1.A.5.77 a way to get root access might be a prior firmware downgrade to its predecessor build 23.0.A.2.93 which of course is not practicable as it would destroy the evidence on the device.

No further research was performed for this phone.

## 4.5 Identical twins

Many smartphones are manufactured in several flavours due to marketing considerations and/or the regulatory characteristics of the target market. The fragmented Android ecosystem (cmp. [19, chapter 1]) furthermore causes carrier specific developments of the same device varying in OS version, software and sometimes appearance. As we have seen in the previous sections, it is very important for a successful live analysis to know exactly what device and what OS version is to be examined. Given an Android smartphone with screen lock enabled and USB debugging disabled the only way to get certainty about the exact model is via its physical appearance. Using the example of two (or more precisely four) smartphone models, we show that this task is not always feasible in practice.



### 4.5.1 Samsung Galaxy S III / S III LTE

Samsung's 2012 flagship smartphone is the Galaxy S III. We know of four different versions listed in table 4.5.

**Table 4.5:** Samsung Galaxy S III models

Model	Year	Processor	RAM	LTE	Android <sup>a</sup>
S III (GT-I9300)	2012	Quad-core 1.4 GHz Cortex-A9	1 GB	○	4.3
S III Neo (GT-I9301)	2014	Quad-core 1.4 GHz Cortex-A7	1.5 GB	○	4.4
S III LTE (GT-I9305)	2012	Quad-core 1.4 GHz Cortex-A9	2 GB	●	4.4
S III AT&T (SGH-I747M)	2012	Dual-core 1.5 GHz Krait	2 GB	●	4.4

<sup>a</sup> Latest official Android version as of March 2015.

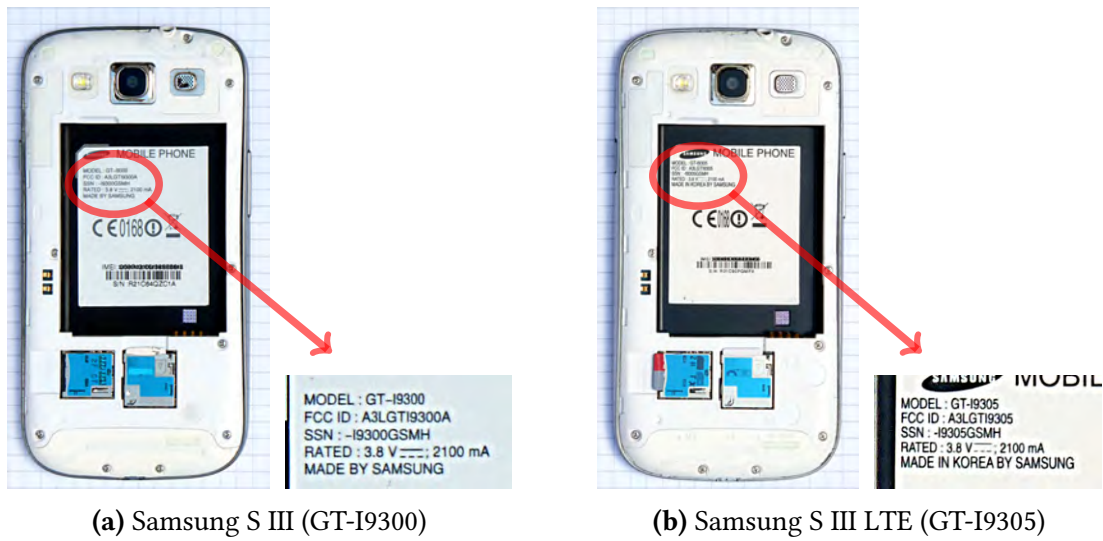
At least our two models S III (GT-I9300) and S III LTE (GT-I9305) appear as almost identical twins. The only difference we spotted was a LTE logo on the I9305's back cover (fig. 4.22). As it is not uncommon for the back cover to be exchanged to personalize its appearance, the LTE logo is not a reliable differentiator.

Every smartphone we dealt with disclosed its specific model on a label beneath the battery. The S III labels are shown in fig. 4.23. Of course, as a practical matter, we cannot gather knowledge about a booted smartphone by removing the battery as this would wipe out the device's volatile memory.



**Figure 4.22:** Samsung S III and S III LTE: Only difference is the LTE logo on the back cover

Identifying a smartphone model by its appearance does not mean to know its OS version. In the case of the above mentioned S III LTE (GT-I9305) one could expect the latest official Android version 4.4. Actually this particular phone runs with a custom ROM CyanogenMod 10.1.3-i9305.



**Figure 4.23:** Samsung S III and S III LTE: Labels under battery allow model identification

### 4.5.2 Samsung Galaxy S III mini(s)

The Samsung Galaxy S III mini is an example of a visually identical twin. It was released, amongst others, in two versions, one with and the other without Near Field Communication (NFC). First released in 2012, Samsung updated this model in 2014 with a new chip set and a newer Android version. All four are listed in table 4.6. The external case however has not changed at all as we can see in fig. 4.24.

**Table 4.6:** Samsung Galaxy S III mini models

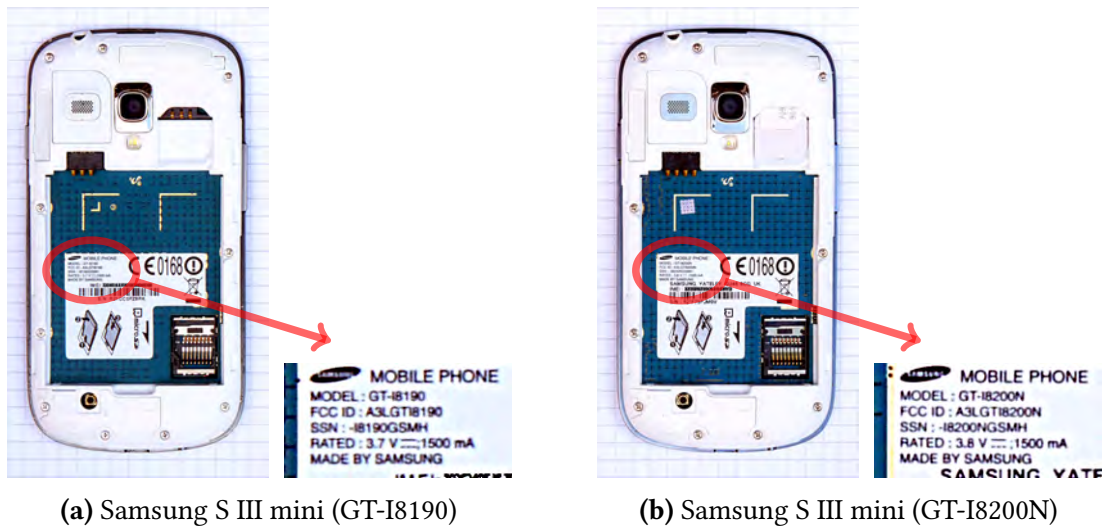
Model	Year	SoC	RAM	NFC	Android <sup>a</sup>
S III mini (GT-I8190)	2012	ST-Ericsson NovaThor U8500 1 GHz	1 GB	○	4.1.2
S III mini (GT-I8190N)	2012	ST-Ericsson NovaThor U8500 1 GHz	1 GB	●	4.1.2
S III mini (GT-I8200)	2014	Marvell PXA986 1.2 GHz	1 GB	○	4.2.2
S III mini (GT-I8200N)	2014	Marvell PXA986 1.2 GHz	1 GB	●	4.2.2

<sup>a</sup> Latest official Android version as of March 2015.

As before with the bigger sibling S III a label hidden by the battery (fig. 4.25) offers information about the specific model at hand.



**Figure 4.24:** Samsung S III mini (GT-I8190 and GT-I8200N): Not distinguishable from each other



**Figure 4.25:** Samsung S III mini (GT-I8190 and GT-I8200N): Labels under battery allow model identification

## 4.6 Conclusion

### 4.6.1 10 steps to Android smartphone RAM analysis

We have shown the steps needed in order to successfully use LiME and Volatility for Android smartphone forensics. These are:

- disable or sidestep lock screen
- enable USB debugging
- identify device: hardware, related code names, OS version
- find and download kernel sources
- find and download appropriate toolchain
- figure out which compiler `.config` was used
- cross compile kernel, LiME, Volatility; create Volatility profile
- root the device while preserving RAM
- transfer LiME module using USB interface and acquire RAM
- run Volatility and be prepared to solve issues

While forensically examined an Android smartphone should be cut off from any network connections. In order to exchange data with the device the preferred method is via the USB interface. This can either be done by booting the device into download mode or, on a booted device, by enabling USB debugging. Booting would destroy the memory content and is therefore not an option. A cold boot attack such as FROST may be possible, albeit fraught with risk. Besides, FROST causes loss of persistent memory content if the boot loader is locked. USB debugging is disabled by default for security reasons. In order to enable it, interaction with the UI is needed. Thus a lock screen can successfully prevent access to the device's settings menu.

With access to the USB interface the examiner queries detailed information about the device's hardware and Android version. In addition, software for RAM acquisition can be placed on the device using the ADB and the memory dump can be pulled over the same channel.

Each combination of hardware and Android version requires its own specific LiME LKM and Volatility profile. With knowledge of the hardware version and its related code names it should be possible to find the kernel sources for the device's Android version. Although the open source license obligates the manufacturers to make the kernel sources publicly available one cannot not find all the source code online. Additionally, the right toolchain has to be found for compiling the kernel and modules. Even if the kernel sources come with compiler configurations for some hardware, it cannot be excluded that the manufacturer actually used a different `.config`. If the sources, toolchain, and

kernel configuration harmonize, the kernel and the modules can be compiled and the Volatility profile created.

Execution of LKMs requires root privileges. Rooting an Android smartphone is often possible. But most rooting solutions force at least a RAM erasing reboot if not unlocking the boot loader which in turn erases even persistent user data. So the examiner has to find a rooting method preserving the volatile memory which is not always available. Furthermore, the solution used should be fully understood so that it can be explained in court.

Having solved all these prerequisites, running LiME and transferring the memory dump is straightforward.

Finally, Volatility can throw a last set of obstacles in one's path. Linux and especially Android analysis is not as perfected as is the case under Windows. Running Volatility on Android dumps means that one is likely to run into issues.

#### 4.6.2 Virtualized vs. real Android

Android Virtual Devices are very convenient for research purpose. All sources are available and the VM can become designed as needed. When dealing with real devices, sources and tools have to be assembled and additional security mechanisms have to be overcome. Table 4.7 is an expanded version of table 3.2 including results of this chapter.

**Table 4.7:** Comparison of the VM and real phone experiments

Process step	Windows VM	GNU/Linux VM	Android AVD	Android Smartphone
<b>Common aspects</b>				
Detailed knowledge about the exact OS version and configuration is required	○	●	●	●
Modules have to be compiled against the target's kernel	○	●	●	●
Cross compilation always necessary	○	○	●	●
Research on kernel sources and kernel configuration required	○	○	○	●
Kernel has to be compiled in order to get kernel symbols	○	○	○	●
Find compatible compiler version	○	○	●	●
Lock screen has to be overcome	○	○	○	●
<b>Memory acquisition</b>				
Software has to be tailored for target	○	●	●	●
Root permissions required	●	●	●	●
Rooting	○	○	○	●
<b>Memory analysis</b>				
Volatility profiles have to be tailored	○	●	●	●

We have seen that the entire process from preparation to acquisition is likely to require significant time. Time that the forensics practitioner may not have due to his workload. But even if he had sufficient time there is another time related issue. In the following chapter 5 we try if the continuously changing memory content of the device under investigation increases the time pressure for the examiner.

# 5

## MEMORY ENDURANCE – EVIDENCE EROSION

---

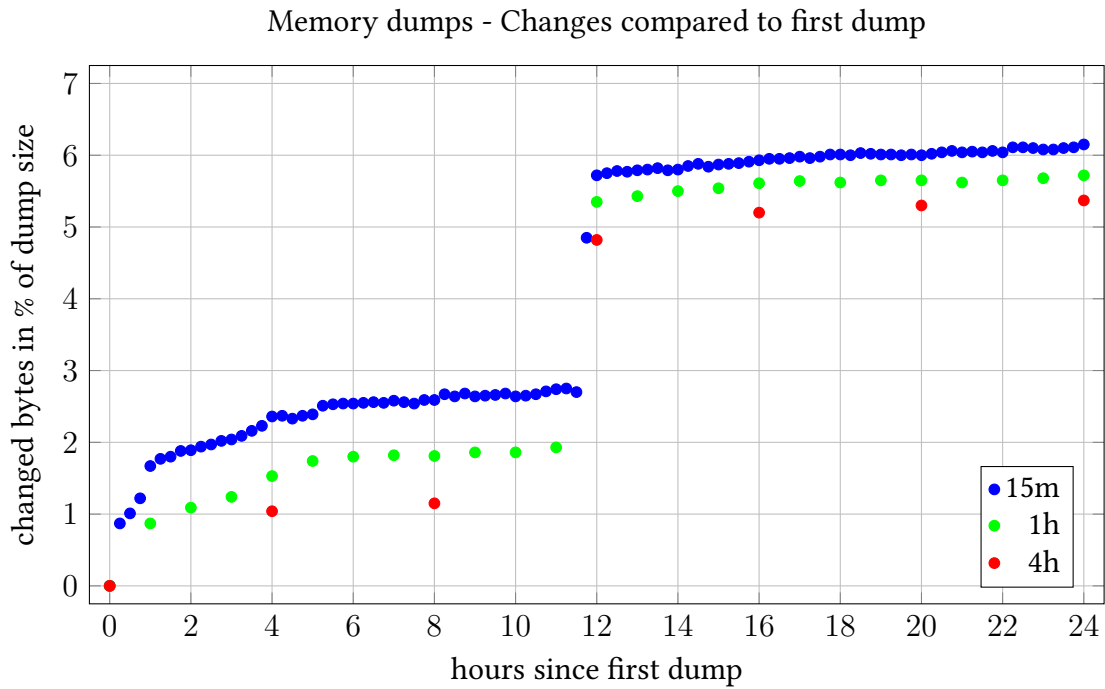
We have seen that it might take significant time until everything is prepared to acquire a memory dump from an Android device. During this time the device continues to operate. This means the memory is constantly changing. The examiner will try to reduce this as far as possible. It is standard procedure to instantly isolate the piece of evidence from any radio communication such as cellular radio, WLAN, Bluetooth, NFC, etc. by caging it in a Faraday bag or the like. At the same time the power supply has to be kept alive.

What is happening during this time in the device's memory is highly dependant on the installed and running software. In order to discover the least extent of evidence memory erosion on a device we performed endurance tests with both an AVD and a Nexus S. The goal was to acquire and compare several memory dumps over a period of 24 hours. The experiment was executed three times with changing intervals of 15 minutes, one hour and four hours.

### 5.1 Memory endurance test with AVD

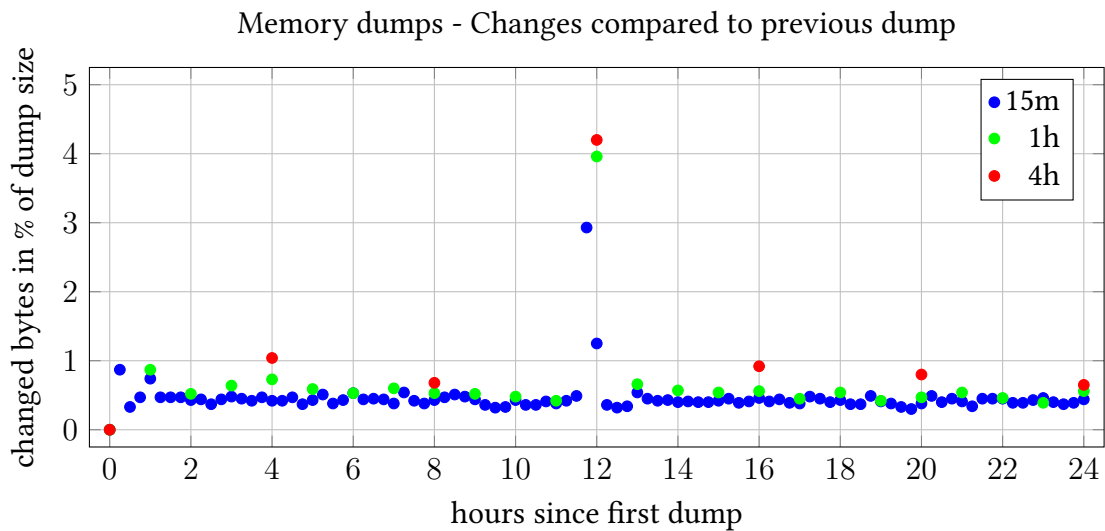
Simulating a Faraday cage, the still operating device was disconnected from all networks. The AVD was created by a bash script as were the dumps acquired in pre-set intervals. (See listing 7.3 in the appendix.)

As one would expect every subsequent dump differed increasingly from the initial dump.



**Figure 5.1:** Change of Goldfish memory over 24 hours (view 1)

Figure 5.1 illustrates these slight but steadily accumulating changes over a period of 24 hours. During the first hour there are comparative big differences. Because it was anticipated that the device works quite a lot right after it is first activated the script was configured to wait 40 minutes between booting and taking the first dump in order to reduce this effect.



**Figure 5.2:** Change of Goldfish memory over 24 hours (view 2)

A remarkable observation is a peak of memory activity in the middle of the experiment.



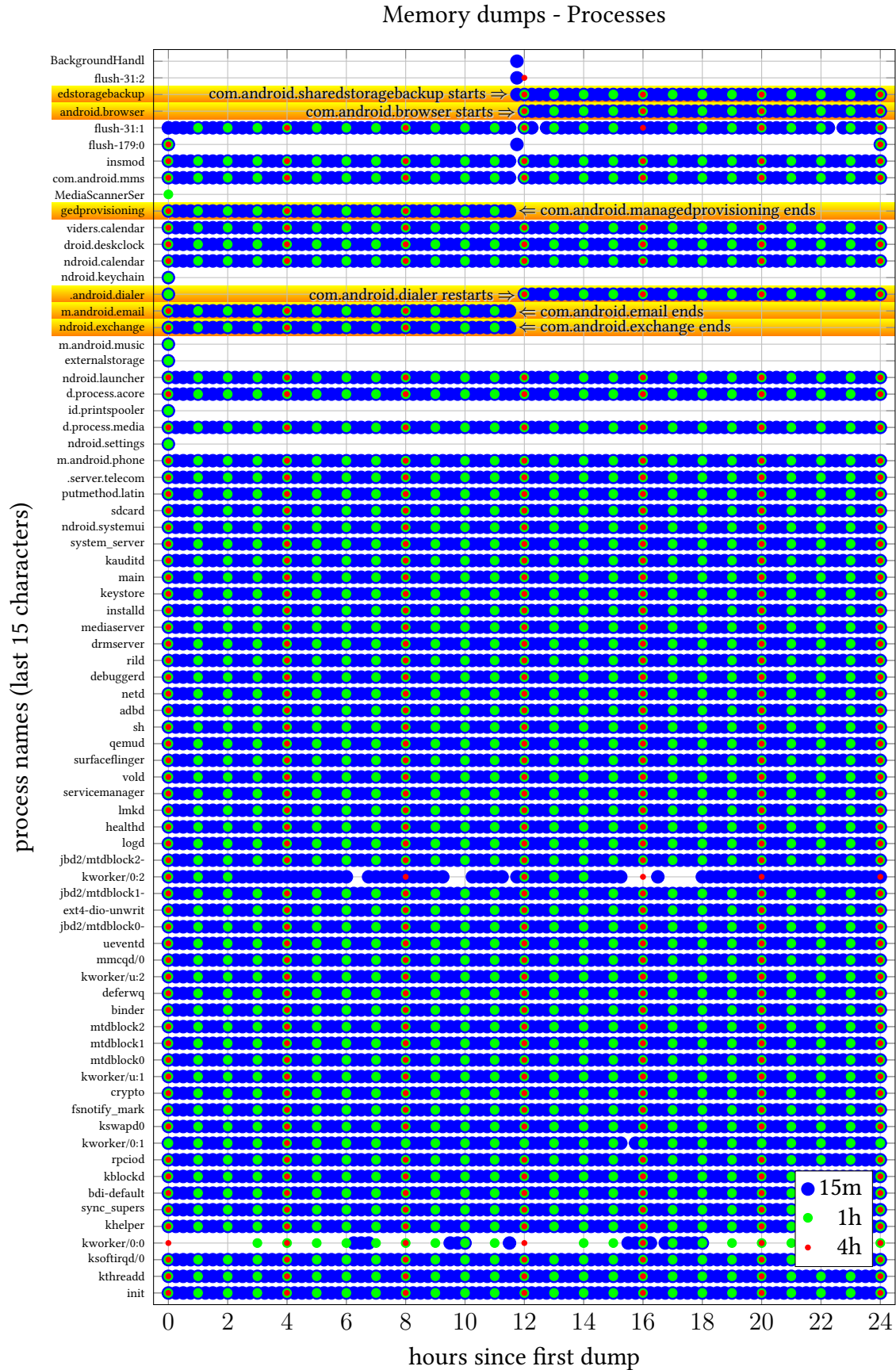


Figure 5.3: Change of Goldfish memory over 24 hours (processes)

All three passes showed this peculiarity at the same time and with comparable values. Figure 5.2 gives a different view by way of a comparison of every dump with its preceding dump. But what triggered this peak in memory changes? Leveraging Volatility's `linux_pslist` plugin we see about 70 processes. With the help of a short shell script (listing 7.5) we concatenate each dump's process lists and present the resulting data in a chart (fig. 5.3). In order to maintain readability the process IDs have been eliminated from the chart. Fragmentary representations of processes – namely the `kworker` and `flush` processes – appear with multiple IDs. Processes that appear to be running all the time generally kept their process IDs. The sole exception was creating the dump itself which is based on the two processes `sh` and `insmod`.

Figure 5.3 reveals a reproducible change of processes unique to the examined 24 hour experiments. Three programs stop:

- `com.android.exchange`
- `com.android.email`
- `com.android.managedprovisioning`

And three programs start:

- `com.android.dialer`
- `com.android.browser`
- `com.android.sharedstoragebackup`

Studying the mechanism behind this behavior goes beyond this scope of this thesis. Further work could be spent on collecting such effects in order to include them in an evaluation of the impact of malware.

The second remarkable observation can be read from fig. 5.2. During the 15-minutes-pass the difference between two consecutive dumps is predominantly constant. A valid forecast would be to have four times the amount of changes between two consecutive 1-hour-dumps and 16 times the amount of changes between two consecutive 4-hour-dumps. Instead, fig. 5.2 shows minor differences in the values of all three passes. If the peaks in the chart's middle are masked the average difference between two consecutive dumps is 2.3 MB, 2.9 MB, and 4.4 MB for the 15 minute, 1 hour, and 4 hour passes. We conclude logically that the vast majority of memory activity is caused by LiME's acquisition process, namely the `sh` and `insmod` threads.

## 5.2 Memory endurance test with Nexus S

We repeated the endurance test with the Nexus S. Only the default applications are installed in order to get a picture of the minimum expected activity. Immediately after initializing the phone the airplane mode is switched on.

### 5.2.1 Preperation

Each of the three runs begins with flashing a fresh stock ROM and rooting as in sections 4.1.1 to 4.1.3. Half an hour before the first dump is taken the device is started and initialized with the following settings:

Language:	English (United States)
Insert SIM card:	Skip
Select Wi-Fi:	Skip
Got Google? (enter Gmail account):	No
Make it Google (get Google account):	Not now
Google and location:	disable all
Date & time:	Central European Time
This phone belongs to...:	Skip

System settings → Developer options	
Developer options:	ON
Allow development settings?	OK
USB debugging:	Check
Allow USB debugging?	OK

Press and hold On/Off-Button for 2 sec.

Airplane mode	tap
---------------	-----

As before with the AVD the memory acquisition is script driven. With the standard AVD daemon on the phone the `adb shell insmod` command is not allowed. The error message is: “adb cannot run as root in production builds”. This constraint can be solved by temporarily replacing the ADB daemon by *adbd Insecure* from XDA developer Chainfire [13] (see also section 2.3.2). The corresponding `adb install` command is shown in fig. 5.4.

```
$ adb install adbd-Insecure-v2.00.apk
2690 KB/s (752250 bytes in 0.273s)
pkg: /data/local/tmp/adbd-Insecure-v2.00.apk
Success
rm failed for -f, No such file or directory
```

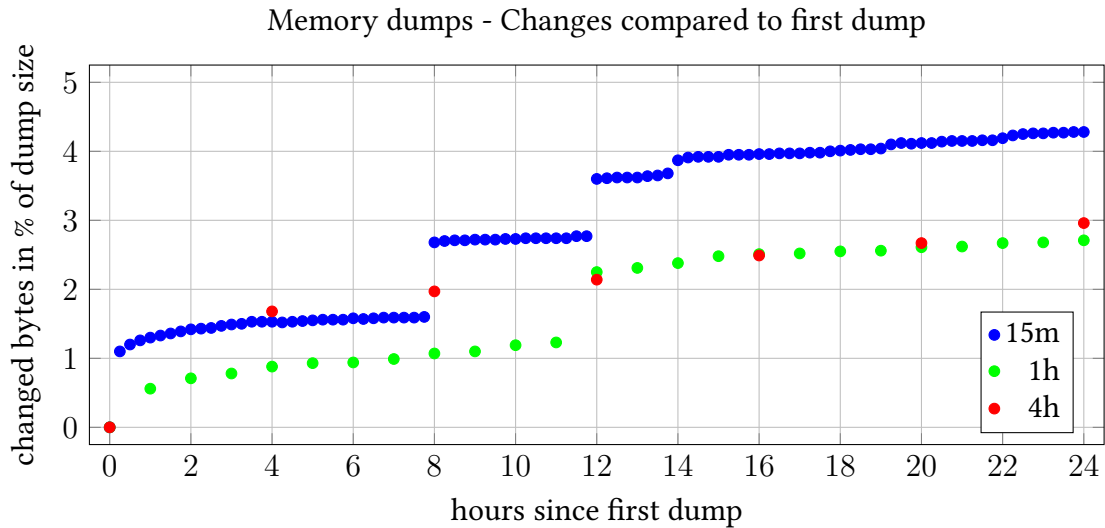


**Figure 5.4:** Nexus S: Install “adbd Insecure”

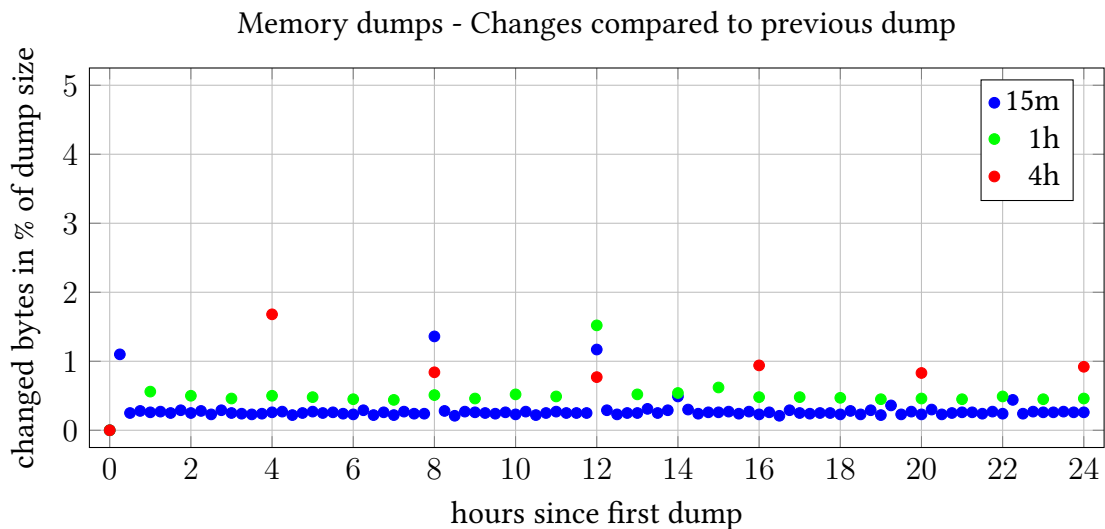
The app “adbd Insecure” must be started manually and the option “Enable insecure adbd” has to be checked.

### 5.2.2 Results

When compared to the AVD experiment we find similar results. Though the Nexus S possesses 50% more process names than its virtualized sibling (107 vs. 74), both systems' memory changes are comparable in total numbers (fig. 5.5 and fig. 5.6). Figure 5.8 and the lower part of fig. 5.7 document that again most processes occur in every memory dump of each pass.



**Figure 5.5:** Change of Nexus S memory over 24 hours (view 1)



**Figure 5.6:** Change of Nexus S memory over 24 hours (view 2)

The main difference between the two experiments is the variance between the 15 minute, 1 hour, and 4 hour passes with respect to when the processes ran. While the AVD showed a very homogeneous picture concerning this matter there are some process names on the

Nexus S which do not appear in every pass. Furthermore, their start and end times are more erratic. The most eye-catching processes are marked in fig. 5.7. Table 5.1 depicts their varying occurrence. The whole picture is less definite than with the AVD.

**Table 5.1:** Nexus S: Occurrence of processes

Process name	Occurrence
	15m / 1h / 4h
com.android.inputmethod.dictionarypack	●●●
com.android.musicfx	○●●
com.android.defcontainer	○●●
com.android.partnersetup	●●●
com.google.android.googlequicksearchbox	○●●
com.google.android.talk	●○○
com.google.android.music:main	●○○
com.android.providers.calendar	●●●
com.google.android.apps.maps:LocationFriendService	●○○
com.google.android.calendar	●●●
eu.chainfire.adbd	●●●
com.android.settings	●●○
com.android.voicedialer	○●●
com.google.android.gm	●○○
com.android.vending	●●●
com.google.android.deskclock	●○○
android.process.acore	●●●

## 5.3 Conclusion

Both experiments indicated a moderate change in memory over time. We concluded that the vast majority of memory activity is caused by LiME's acquisition process. LiME as it has been utilized is designed for one time use. That is, it is started with `insmod` and ends all activities after having saved/transferred the memory dump. Hence, for each new run we connected to the device, unloaded the module and loaded it again. The resulting memory footprint could be reduced if the LiME module were expanded by the ability to do multiple time triggered dumps by itself.

Furthermore, we saw self-acting processes starting and ending without a distinguishable pattern of behavior. Some of these actions correlate temporally. We saw this very clearly and reproducibly in the AVD experiment where it appeared that three processes were exchanged by three other processes.

The Nexus S behaved more erratically than the virtual Goldfish device. This proves our statement that an AVD is more predictable and therefore more convenient to use for research.

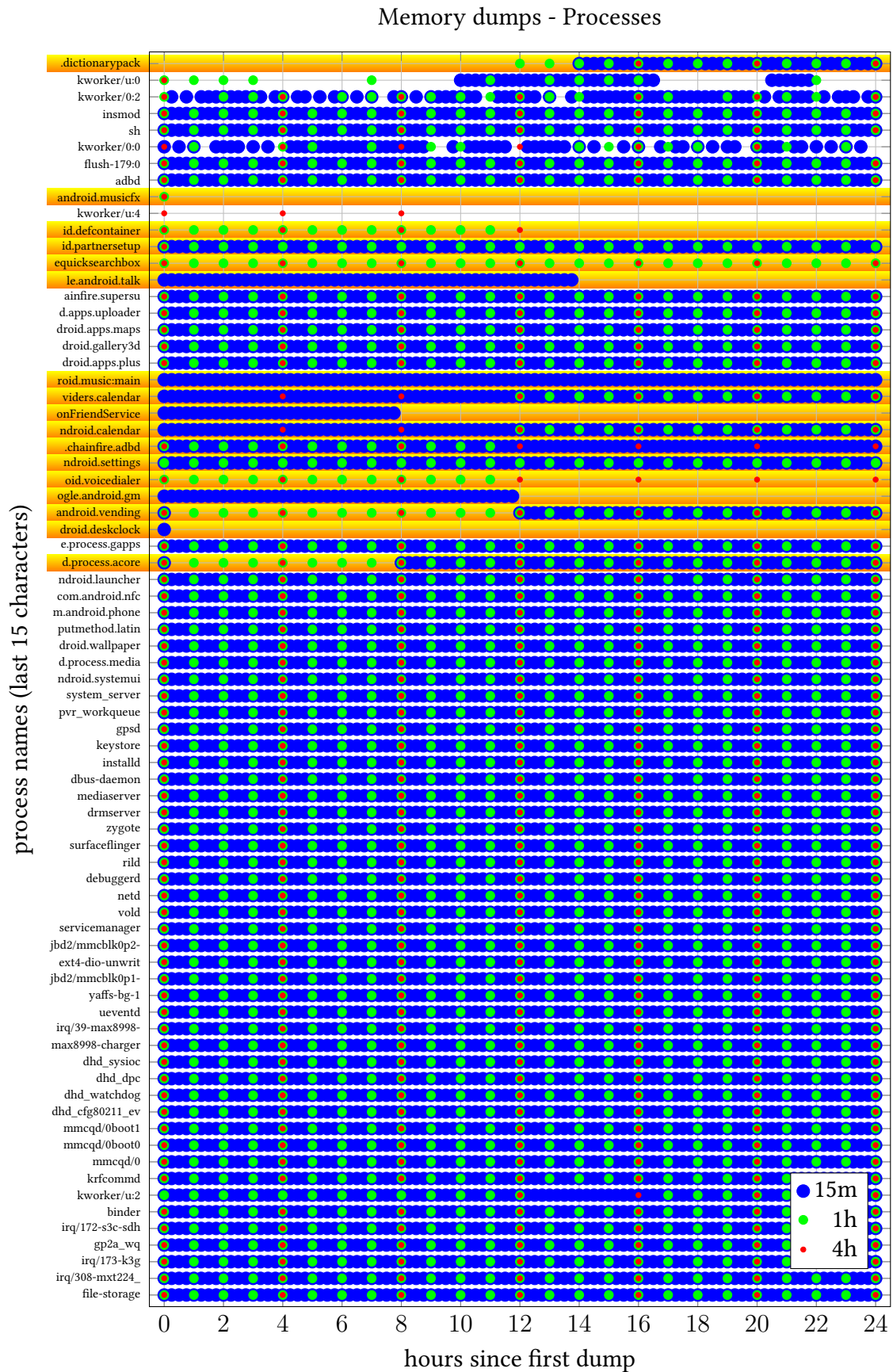
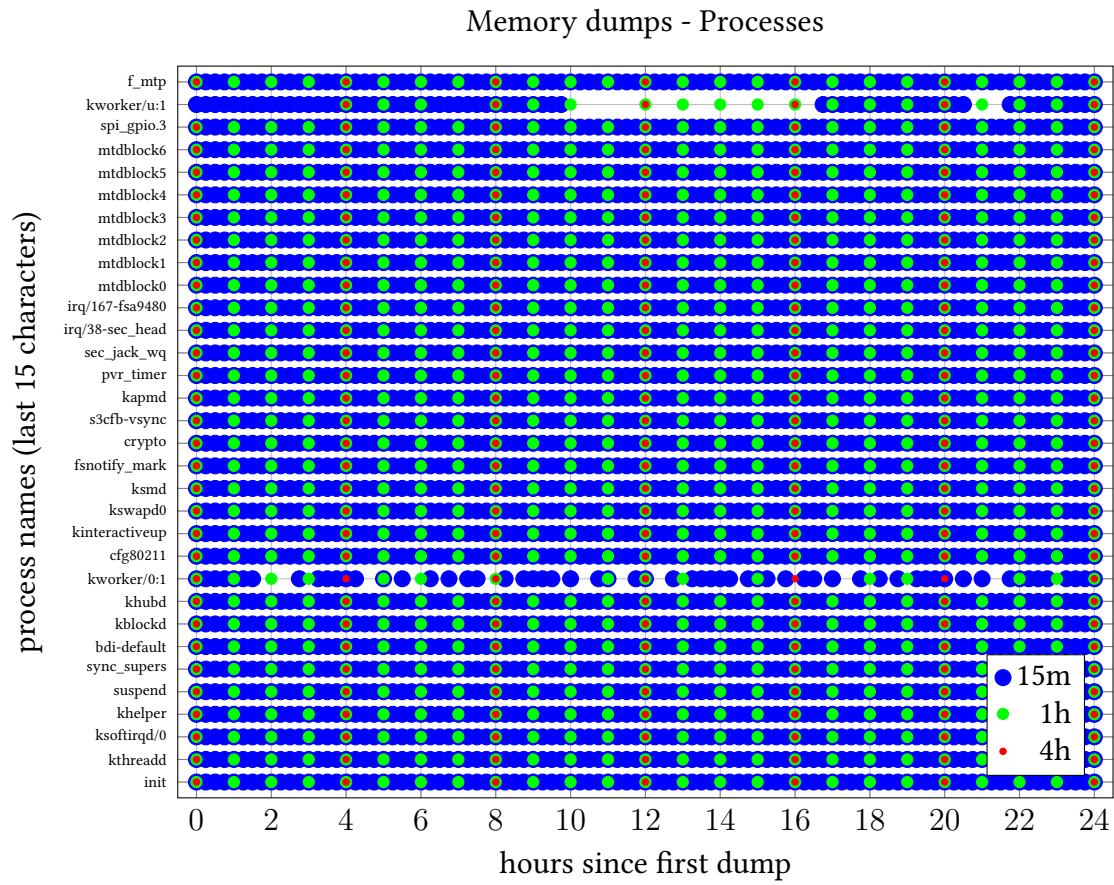


Figure 5.7: Change of Nexus S memory over 24 hours (upper 75 processes)



**Figure 5.8:** Change of Nexus S memory over 24 hours (lower 32 processes)





## CONCLUSION AND FUTURE WORK

---

Finally, we want to draw conclusions about the work which has been done during this thesis and derive proposals for future research.

Academic research papers often look at a very specific question. If an answer is found resulting from a specific experimental set-up the underlying assumption is considered proven. It is beyond dispute that forensic analysis of Android volatile memory is possible and—if successful—of potential value to law enforcement. Many papers present their results on Android memory structures and application artifacts (cmp. 1.3, 2.3.4). We have also created and described in detail experimental set-ups with both a virtualized and a real but prepared Android device resulting in successful memory acquisition.

In a further step we examined whether Android smartphone live forensics with LiME and Volatility can be said to be feasible in general. “Live” in this case has two meanings. First, it refers to a booted device. Second, the memory should be acquired in a comparatively short period. Here our conclusion is negative. We described the increasing complexity of the memory acquisition process when moving from Windows to Linux to Android as well as from virtualized to real devices. Additional complexity is added by the need to create Volatility profiles. Regarding Android smartphones the whole process is so slow, that—even if successful—time effective acquisition is unlikely. On top of that there are other obstacles which often force a failure (cmp. 4.6.1). Moreover, we have seen that trying to overcome these obstacles can endanger the device’s volatile RAM and even its persistent memory.

There are also logistical obstacles. Kernel sources are not always available or do not match those used by the actual phone. Perhaps the needed toolchain is deprecated and no longer available. Or even more trivial: The required USB or charger cables are not at hand. In addition, Android might throw other obstacles into the examiner's path. Android offers some features allowing the smartphone's owner to secure his device and the enclosed personal data against unauthorized access. Security aware users will take care to not leave the boot loader unlocked, not to root their phone permanently, and keep the developer options disabled. Additionally, they will allow Android to encrypt as much of the persistent memory as possible, and enable the lock screen with an effective password. These hurdles are hard or impossible to overcome even if one were not attempting to preserve the device's volatile memory.

A question we did not discuss is how to interpret the data gathered from a rooted smartphone with no security features enabled. Could the owner, in this way, disclaim accountability for any incriminating data found on his device?

Besides the complexity due to Android itself the reliability of tools used for the analysis also has an impact on the success of an investigation. We noted two important facets. On the one hand the utilized tools must be fully trustworthy. That is, they are open source, the examiner understands their source code, and they build their own binaries. Alternatively the tool might be certified by a trusted authority. On the other hand the tool should ideally be bug free. It is well known that software cannot be programmed without bugs. But if it is crucial for an investigation then the technical support for the tool must be fast and effective or else the forensics practitioner will need to be able to help himself.

In summary, we addressed the following problems: fully understanding the rooting technique (4.2), an obscure Chinese rooting windows binary (4.3.1), a problem with LiME that was promptly solved by the developer on request (2.2.1), a bug in Volatility that we solved ourselves (3.2.7), and a still unsolved issue in the Volatility bug tracker (3.2.2).

Since time effective memory acquisition is unlikely the question comes up if an acceptable time frame can be defined. We developed an approach to evaluate memory erosion on an Android smartphone but the test series were insufficient for a final rating. The decay of the original memory state is highly dependant on the active threads. We assume that the vast majority of the observed changes are due to the repeatedly unloaded and loaded LiME module. We therefore propose that LiME be re-designed with the ability to carry out multiple dumps at pre-defined intervals.

## APPENDICES

---

### 7.1 Sony Xperia Mini Pro

Here are the `Makefile`s for LiME and Volatility as well as the commands required to compile their modules and to create the Volatility profile as used in section 4.3.2.

Listing 7.1: SonyXperiaMiniPro/Makefile.Volatility.cross

```
1  obj-m += module.o
2
3  KDIR := /media/ultrabay/thesis/xperiamini/kernel/
4  CCPATH := \
5      /media/ultrabay/thesis/xperiamini/android_prebuilt_toolchains/arm-eabi-4.4.3/bin
6  -include version.mk
7
8  all: dwarf
9
10 dwarf: module.c
11     $(MAKE) ARCH=arm CROSS_COMPILE=$(CCPATH)/arm-eabi- -C $(KDIR) \
12         CONFIG_DEBUG_INFO=y M=$(PWD) modules
13     dwarfdump -di module.ko > module.dwarf
```

```

$ # >>>> compile LiME loadable module
$ git clone https://github.com/504ensicsLabs/LiME.git
<snip>
$ cp Makefile.LiME.cross LiME/src/Makefile
$ cd LiME/src
$ make clean && make
<snip>
$ cp lime-XperiaMiniPro.ko ../../
$ cd ../../
$ # >>>> create Volatility profile
$ git clone https://github.com/volatilityfoundation/volatility.git
<snip>
$ cp Makefile.Volatility.cross volatility/tools/linux/Makefile
$ cd volatility/tools/linux/
$ make
<snip>
$ zip -j Android_SonyXperiaMiniPro_2.6.32.9-perf.zip module.dwarf ../../../../System.map
<snip>
$ cp Android_SonyXperiaMiniPro_2.6.32.9-perf.zip ../../../../
$ cp Android_SonyXperiaMiniPro_2.6.32.9-perf.zip
  ../../../../volatility/plugins/overlays/linux/
$ cd ../../../../

```




Figure 7.1: Sony Xperia Mini Pro: Compile LiME and create Volatility profile

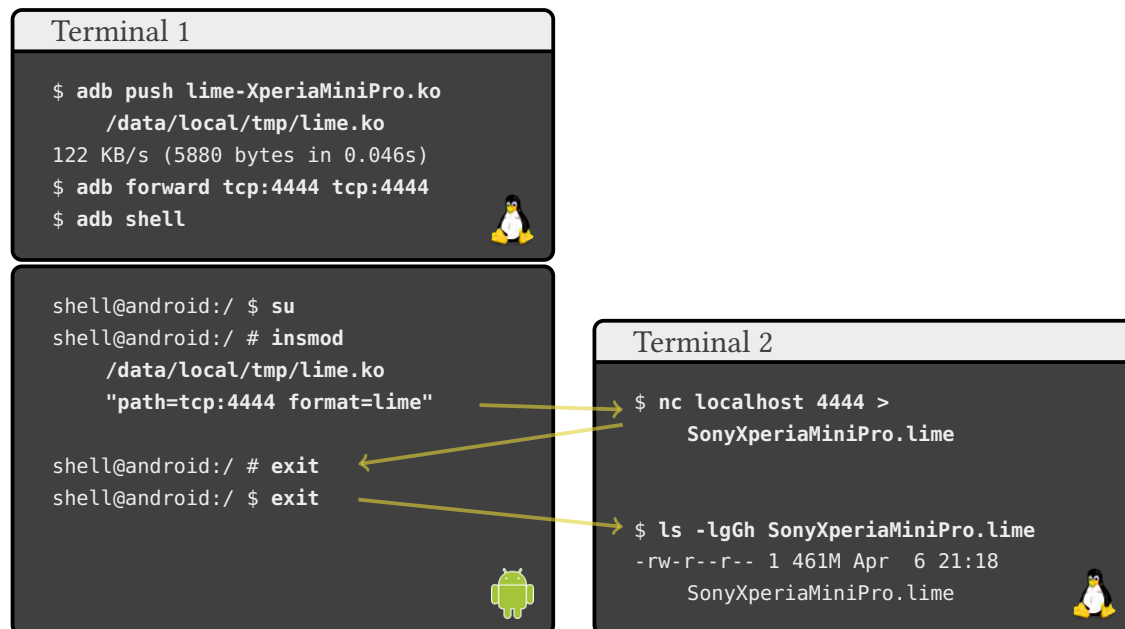


Figure 7.2: Sony Xperia Mini Pro: LiME TCP transfer

Listing 7.2: SonyXperiaMiniPro/Makefile.LiME.cross

```
1  obj-m := lime.o
2  lime-objs := tcp.o disk.o main.o
3
4  KDIR := /media/ultrabay/thesis/xperiamini/kernel/
5  KVER := XperiaMiniPro
6
7  PWD := $(shell pwd)
8  CCPATH := \
    /media/ultrabay/thesis/xperiamini/android_prebuilt_toolchains/arm-eabi-4.4.3/bin
9
10 default:
11     $(MAKE) -C $(KDIR) M=$(PWD) modules
12     $(CCPATH)/arm-eabi-strip --strip-unneeded lime.ko
13     mv lime.ko lime-$(KVER).ko
14
15     $(MAKE) tidy
16
17 tidy:
18     rm -f *.o *.mod.c Module.symvers Module.markers modules.order \*.o.cmd \
        \*.ko.cmd \*.o.d
19     rm -rf \.tmp_versions
20
21 clean:
22     $(MAKE) tidy
23     rm -f *.ko
```

## 7.2 Memory endurance test scripts

The experiments in chapter 5 are script driven. So is the concatenation of the process lists.

## 7.3 AVD: Bash script for multiple memory dumps over time

This bash script (listing 7.3)

- creates an AVD
- starts the AVD at a defined time
- pushes a LiME module to the AVD
- acquires memory dumps at a defined time interval during a defined time

Chapter 5 describes the usage.

Listing 7.3: multidump-goldfish.sh

```

1  #!/bin/bash
2
3  # call me like
4  #
5  #           |start |start | dump   | start the |emu |netcat|
6  #           |AVD  |first | every x | last dump |port|port |
7  #           |     |dump  | seconds | x seconds |   |   |
8  #           |     |     |         | after the |   |   |
9  #           |     |     |         | first one |   |   |
10 # ./multidump_goldfish.sh 183000 190000 $((15*60)) $((12*60*60)) 5560 6560
11
12 # global constants =====
13 t1=$1          # time to start AVD [hhmmss]                e.g. 134000
14 t2=$2          # time to start dump routines [hhmmss]       e.g. 135900
15 timeInterval=$3 # time interval in sec.;                    e.g. 600sec = 10min
16 timeTotal=$4    # total time in sec;                        e.g. 12 * 60 * 60sec = 12hrs
17 portEmu=$5      # port for emulator;                        even integer between 5554 and 5680
18 portNc=$6       # port for netcat;                          e.g. 4444
19 pathKernel="${HOME}/android/test-goldfish/goldfish/arch/arm/boot/zImage"
20 pathLiME="${HOME}/android/test-goldfish/lime-goldfish.ko"
21 pathDump="/media/usb0/android/endurance"
22
23 # derived global constants -----
24 dumpsCount=$((($timeTotal / $timeInterval + 1)) # count of dumps
25 timeInterval5=$(printf "%05d" $timeInterval) # string with len 5; e.g. '00600'
26 idAvd="AVD"$timeInterval5 # AVD name; e.g. 'AVD00600'
27 idEmu="emulator-$portEmu" # AVD serial; e.g. 'emulator-5554'

```

```

27
28 # log function =====
29 log() {
30     timestamp=$(date +%Y-%m-%d %T)
31     echo $timestamp $1
32     echo $timestamp $1 >> multidump_goldfish_$timeInterval5.log
33 }
34
35 # check if emulator has boot completed =====
36 emu_boot_completed() {
37     # is there a device?
38     if [ $(adb devices | grep -c "$idEmu.*device") = 1 ]; then
39         echo "                waiting for boot complete"
40         # check boot complete
41         stateEmu=$(adb -s $idEmu shell getprop init.svc.bootanim)
42         if [ ${stateEmu:0:7} = "stopped" ]; then return 0; else return 1; fi
43     else
44         echo "                waiting for device"
45         return 1
46     fi
47 }
48
49 # wait for time =====
50 wait_until() {
51     log "wait until ${1:0:2}:${1:2:2}:${1:4:2}"
52     while [ $(date +%H%M%S) -lt "$1" ]; do ;; done
53 }
54
55 # MAIN #####
56 log "START"
57
58 log "create AVD (name = $idAvd)"
59 echo no | android create avd -n $idAvd -t 'android-21' -b 'default/armv7a' -c \
    128M -f
60
61 wait_until $t1
62
63 log "start emulator (emu id = $idEmu)"
64 emulator -avd $idAvd -port "$portEmu" -kernel $pathKernel &
65
66 log "wait for bootcomplete and SD card to become writable"
67 until emu_boot_completed; do
68     sleep 5
69 done
70
71 log "push LiME module"
72 adb -s $idEmu push $pathLiME /sdcard/lime.ko
73
74 log "forward tcp port"
75 adb -s $idEmu forward tcp:"$portNc" tcp:"$portNc"
76

```

```
77 wait_until $t2
78
79 log "acquire $dumpsCount memory dumps; interval = $((($timeInterval / 60))min"
80 for i in `seq 1 $dumpsCount`; do
81     j=$(printf "%03d" $i)                # counter for file name
82
83     while [ $(( $(date +%s) % $timeInterval )) != 0 ]; do
84         :
85     done
86
87     log "loop $j - insmod"
88     adb -s $idEmu shell insmod /sdcard/lime.ko "path=tcp:$portNc format=lime" &
89     sleep 5
90
91     dumpfile="$pathDump/goldfish.$idAvd.$j.lime"
92     log "loop $j - netcat writes to $dumpfile"
93     nc localhost "$portNc" > $dumpfile
94     sleep 5
95     ls -lG $dumpfile
96
97     log "loop $j - rmmod"
98     adb -s $idEmu shell rmmod lime.ko
99 done
100
101 log "kill AVD"
102 adb -s $idEmu emu kill
103
104 log "END"
105 # END #####
```



## 7.4 Nexus S: Bash script for multiple memory dumps over time

This bash script (listing 7.4) is a revised version of the previously used script (listing 7.3). For the Nexus S experiment the AVD management is stripped.

Chapter 5 describes the usage.

Listing 7.4: multidump-nexuss.sh

```

1  #!/bin/bash
2
3  # call me like
4  #           |start | dump   | start the |netcat|
5  #           |first | every x | last dump |port  |
6  #           |dump  | seconds | x seconds |      |
7  #           |      |      | after the |      |
8  #           |      |      | first one |      |
9  # ./multidump_nexuss.sh 205900 $((15*60)) $((24*60*60)) 4444
10
11 # global constants =====
12 tstart=$1      # time to start dump routines [hhmmss]          e.g. 135900
13 timeInterval=$2 # time interval in sec.;                      e.g. 600sec = 10min
14 timeTotal=$3   # total time in sec;                          e.g. 12 * 60 * 60sec = 12hrs
15 portNc=$4      # port for netcat;                            e.g. 4444
16 pathLime="${HOME}/Dokumente/M18/Experiment-NexusS/lime-NexusS.ko"
17 pathDump="/media/ultrabay/thesis/dumps/endurance_nexuss_"
18
19
20 # derived global constants -----
21 dumpsCount=$((($timeTotal / $timeInterval + 1)) # count of dumps
22 timeInterval5=$(printf "%05d" $timeInterval) # string with len 5; e.g. '00600'
23 pathDump=$pathDump$timeInterval5
24 mkdir -p $pathDump
25
26
27 # log function =====
28 log() {
29     timestamp=$(date +"%Y-%m-%d %T")
30     echo $timestamp $1
31     echo $timestamp $1 >> multidump_nexuss_$timeInterval5.log
32 }
33
34
35 # wait for time =====
36 wait_until() {
37     log "wait until ${1:0:2}:${1:2:2}:${1:4:2}"
38     while [ $(date +%H%M%S) -lt "$1" ]; do ;; done
39 }
40

```

```
41
42 # MAIN #####
43 log "START"
44
45 log "push LiME module"
46 adb push $pathLiME /sdcard/lime.ko
47
48 log "forward tcp port"
49 adb forward tcp:$portNc tcp:$portNc
50
51 wait_until $tstart
52
53 log "acquire $dumpsCount memory dumps; interval = (($timeInterval / 60))min"
54 for i in `seq 1 $dumpsCount`; do
55     j=$(printf "%03d" $i)                                # counter for file name
56
57     while [ $(( $(date +%s) % $timeInterval )) != 0 ]; do
58         :
59     done
60
61     log "loop $j - insmod"
62     adb shell insmod /sdcard/lime.ko "path=tcp:$portNc format=lime" &
63     sleep 5
64
65     dumpfile="$pathDump/NexusS.$j.lime"
66     log "loop $j - netcat writes to $dumpfile"
67     nc localhost "$portNc" > $dumpfile
68     sleep 5
69     ls -lG $dumpfile
70
71     log "loop $j - rmmod"
72     adb shell rmmod lime.ko
73 done
74
75 log "END"
76 # END #####
```

## 7.5 Bash script for merging process lists of multiple memory dumps

This bash script (listing 7.5) runs Volatility's `linux_pslist` module on every memory dump of the previously executed endurance experiments. The process information is complemented by the experiment's time interval, the dump's file name and the point of time during the experiment. The resulting data is ready to become analyzed, e.g. using a pivot table or a chart like fig. 5.3.

Listing 7.5: multidump-goldfish-pslist.sh

```

1  cd ~/android/test-goldfish/volatility
2  export VOLATILITY_PROFILE=LinuxAndroid_Goldfish_3_4_67-01413-g9ac497fARM
3
4  outFile=~/.Dokumente/M18/Experiment-Android/pslist.csv
5  rm -f $outFile
6
7  for ti in 15 60 240; do # minutes
8      tt=$( printf "%05d" $(( $ti * 60 )) )
9      dumpDir=/media/ultrabay/dumps/endurance_goldfish_"$tt"/goldfish.AVD"$tt".*.lime
10
11     i=0
12     for f in $( ls $dumpDir ); do
13         echo $f
14         fb=$(basename $f)
15         m=$((i*$ti)) # minute
16         python vol.py -f $f linux_pslist | \
17             awk 'BEGIN{OFS=";"} /^0x/ {print "'$ti'", "'$fb'", "'$m'", $1, $2, $3, $4, $5, $6, $7" \
18                 "$8" "$9;}'' >> $outFile
19         i=$((i+1))
20     done
21 done

```

## 7.6 Memory endurance test results

The charts 5.1, 5.2, 5.5, and 5.6 are based on the figures of the following tables.

**Table 7.1:** Memory endurance test / AVD / 4 hour pass / reference is first dump

minute	bytes changed total	bytes changed [% of total mem.]
0	0	0
240	5591789	1,04
480	6158423	1,15
720	25851890	4,82
960	27913876	5,2
1200	28451937	5,3
1440	28819017	5,37

**Table 7.2:** Memory endurance test / AVD / 1 hour pass / reference is first dump

minute	bytes changed total	bytes changed [% of total mem.]
0	0	0
60	4662219	0,87
120	5831746	1,09
180	6633632	1,24
240	8204905	1,53
300	9330212	1,74
360	9654156	1,8
420	9763171	1,82
480	9743002	1,81
540	9963914	1,86
600	10003027	1,86
660	10348731	1,93
720	28717012	5,35
780	29145013	5,43
840	29546085	5,5
900	29757185	5,54
960	30104193	5,61
1020	30287428	5,64
1080	30172251	5,62
1140	30321891	5,65
1200	30317213	5,65
1260	30186848	5,62
1320	30315768	5,65
1380	30469253	5,68
1440	30735427	5,72

**Table 7.3:** Memory endurance test / AVD / 15 minutes pass / reference is first dump

minute	bytes changed total	bytes changed [% of total mem.]	minute	bytes changed total	bytes changed [% of total mem.]
0	0	0	735	30862620	5,75
15	4688249	0,87	750	31011767	5,78
30	5433981	1,01	765	30950977	5,77
45	6565447	1,22	780	31059067	5,79
60	8984552	1,67	795	31142652	5,8
75	9525785	1,77	810	31262810	5,82
90	9684784	1,8	825	31094337	5,79
105	10076135	1,88	840	31153569	5,8
120	10150356	1,89	855	31404091	5,85
135	10391839	1,94	870	31554122	5,88
150	10562144	1,97	885	31345527	5,84
165	10851326	2,02	900	31512295	5,87
180	10950271	2,04	915	31556441	5,88
195	11233142	2,09	930	31632693	5,89
210	11605734	2,16	945	31742159	5,91
225	11957284	2,23	960	31815720	5,93
240	12653393	2,36	975	31961030	5,95
255	12726652	2,37	990	31945512	5,95
270	12511557	2,33	1005	31971900	5,96
285	12730305	2,37	1020	32080159	5,98
300	12834514	2,39	1035	32010325	5,96
315	13476980	2,51	1050	32097087	5,98
330	13579688	2,53	1065	32252640	6,01
345	13656034	2,54	1080	32254448	6,01
360	13620686	2,54	1095	32212096	6
375	13672958	2,55	1110	32379237	6,03
390	13770293	2,56	1125	32303825	6,02
405	13673778	2,55	1140	32267818	6,01
420	13856176	2,58	1155	32262579	6,01
435	13769826	2,56	1170	32207028	6
450	13646450	2,54	1185	32276945	6,01
465	13881922	2,59	1200	32187205	6
480	13902113	2,59	1215	32344136	6,02
495	14313580	2,67	1230	32447484	6,04
510	14166672	2,64	1245	32511624	6,06
525	14368791	2,68	1260	32427936	6,04
540	14198379	2,64	1275	32466656	6,05
555	14203337	2,65	1290	32437920	6,04
570	14269102	2,66	1305	32517932	6,06
585	14409110	2,68	1320	32449890	6,04
600	14166655	2,64	1335	32811601	6,11
615	14212859	2,65	1350	32795303	6,11
630	14353651	2,67	1365	32773854	6,1
645	14523867	2,71	1380	32631051	6,08
660	14711688	2,74	1395	32631473	6,08
675	14760301	2,75	1410	32775821	6,1
690	14498514	2,7	1425	32820671	6,11
705	26047637	4,85	1440	33022698	6,15
720	30714254	5,72			

**Table 7.4:** Memory endurance test / AVD / 4 hour pass / reference is previous dump

<b>minute</b>	<b>bytes changed total</b>	<b>bytes changed [% of total mem.]</b>
0	0	0
240	5591789	1,04
480	3642042	0,68
720	22559966	4,2
960	4956342	0,92
1200	4317541	0,8
1440	3508055	0,65

**Table 7.5:** Memory endurance test / AVD / 1 hour pass / reference is previous dump

<b>minute</b>	<b>bytes changed total</b>	<b>bytes changed [% of total mem.]</b>
0	0	0
60	4662219	0,87
120	2781617	0,52
180	3424356	0,64
240	3925577	0,73
300	3194048	0,59
360	2834567	0,53
420	3206473	0,6
480	2818755	0,53
540	2788734	0,52
600	2570652	0,48
660	2231148	0,42
720	21285051	3,96
780	3560843	0,66
840	3081331	0,57
900	2903830	0,54
960	2993463	0,56
1020	2396729	0,45
1080	2889583	0,54
1140	2251562	0,42
1200	2504750	0,47
1260	2887732	0,54
1320	2461653	0,46
1380	2111427	0,39
1440	3026763	0,56

**Table 7.6:** Memory endurance test / AVD / 15 minutes pass / reference is previous dump

minute	bytes changed total	bytes changed [% of total mem.]	minute	bytes changed total	bytes changed [% of total mem.]
0	0	0	735	1908779	0,36
15	4688249	0,87	750	1726158	0,32
30	1766839	0,33	765	1838446	0,34
45	2540347	0,47	780	2897437	0,54
60	3963329	0,74	795	2418197	0,45
75	2528374	0,47	810	2233396	0,42
90	2527385	0,47	825	2322874	0,43
105	2498068	0,47	840	2140853	0,4
120	2299376	0,43	855	2192948	0,41
135	2350178	0,44	870	2168885	0,4
150	2003077	0,37	885	2156991	0,4
165	2362114	0,44	900	2257769	0,42
180	2590071	0,48	915	2398902	0,45
195	2403147	0,45	930	2083314	0,39
210	2268839	0,42	945	2223224	0,41
225	2532641	0,47	960	2456516	0,46
240	2250767	0,42	975	2216389	0,41
255	2264988	0,42	990	2343337	0,44
270	2535136	0,47	1005	2108697	0,39
285	1967472	0,37	1020	2020392	0,38
300	2286962	0,43	1035	2564042	0,48
315	2739157	0,51	1050	2395072	0,45
330	2015281	0,38	1065	2160793	0,4
345	2309768	0,43	1080	2319362	0,43
360	2847534	0,53	1095	2010525	0,37
375	2340056	0,44	1110	1972744	0,37
390	2422620	0,45	1125	2611746	0,49
405	2356460	0,44	1140	2194505	0,41
420	2033034	0,38	1155	2060524	0,38
435	2898074	0,54	1170	1785183	0,33
450	2275792	0,42	1185	1598544	0,3
465	2064225	0,38	1200	2014144	0,38
480	2298719	0,43	1215	2617637	0,49
495	2541383	0,47	1230	2122984	0,4
510	2722764	0,51	1245	2426916	0,45
525	2599470	0,48	1260	2217664	0,41
540	2360064	0,44	1275	1820832	0,34
555	1923782	0,36	1290	2407546	0,45
570	1729440	0,32	1305	2406757	0,45
585	1770289	0,33	1320	2389418	0,45
600	2288814	0,43	1335	2091833	0,39
615	1912878	0,36	1350	2103100	0,39
630	1918256	0,36	1365	2311912	0,43
645	2191831	0,41	1380	2486386	0,46
660	2026017	0,38	1395	2153748	0,4
675	2280942	0,42	1410	1973384	0,37
690	2656804	0,49	1425	2116290	0,39
705	15755542	2,93	1440	2372255	0,44
720	6691815	1,25			





# References

- [1] 504ENSICS Labs. LiME Linux Memory Extractor, 2014. URL <https://codeload.github.com/504ensicsLabs/LiME/zip/master>. [Online; accessed 30 Dec. 2014].
- [2] Aisha Ibrahim Ali-Gombe. Volatile memory message carving: A “per process basis” approach. *University of New Orleans*, January 2012. URL <http://scholarworks.uno.edu/cgi/viewcontent.cgi?article=2614&context=td>. [Online; accessed 23 Mar. 2015].
- [3] Tiago Alves and Don Felton. TrustZone: Integrated Hardware and Software Security — Enabling Trusted Computing in Embedded Systems, 2004. URL [http://unsyncopated.com/mirror/information\\_quarterly\\_trustzone.pdf](http://unsyncopated.com/mirror/information_quarterly_trustzone.pdf). [Online; accessed 3 Apr. 2015].
- [4] Android Open Source Project. Kernel tree for Samsung systems on Android., 2012. URL <https://android.googlesource.com/kernel/samsung/>. [Online; accessed 9 Mar. 2015].
- [5] Android Open Source Project. platform/prebuilts/gcc/linux-x86/arm/arm-eabi-4.6/, 2012. URL <https://android.googlesource.com/platform/prebuilts/gcc/linux-x86/arm/arm-eabi-4.6/>. [Online; accessed 5 Feb. 2015].
- [6] Android Open Source Project. platform/prebuilts/gcc/linux-x86/arm/arm-eabi-4.7/, 2012. URL <https://android.googlesource.com/platform/prebuilts/gcc/linux-x86/arm/arm-eabi-4.7/>. [Online; accessed 5 Feb. 2015].
- [7] Android Open Source Project. GitHub | stub monitor implementation, February 2014. URL [https://github.com/android/platform\\_external\\_qemu/commit/aa1180ca05774398245953deb306c0e25829afee](https://github.com/android/platform_external_qemu/commit/aa1180ca05774398245953deb306c0e25829afee). [Online; accessed 5 Feb. 2015].
- [8] Android Open Source Project. Download Android Studio and SDK Tools, 2015. URL <https://developer.android.com/sdk/>. [Online; accessed 2 Feb. 2015].
- [9] Android Open Source Project. Building Kernels, 2015. URL <https://source.android.com/source/building-kernels.html>. [Online; accessed 2 Feb. 2015].
- [10] Dimitris Apostolopoulos, Giannis Marinakis, Christoforos Ntantogian, and Christos Xenakis. Discovering authentication credentials in volatile memory of android mobile devices. In *Collaborative, Trusted and Privacy-Aware e/m-Services*, pages 178–185. Springer, April 2013. URL <http://cgi.di.uoa.gr/~xenakis/Published/49-I3E-2013/2013-I3E-AMNX.pdf>. [Online; accessed 23 Mar. 2015].

## References

---

- [11] ARM. TrustZone, 2014. URL <http://www.arm.com/products/processors/technologies/trustzone/index.php>. [Online; accessed 3 Apr. 2015].
- [12] ARM. Versatile Express Product Family, 2014. URL <http://www.arm.com/products/tools/development-boards/versatile-express/index.php>. [Online; accessed 8 Feb. 2015].
- [13] Chainfire. SuperSU – CF-Root download page, November 2014. URL <http://forum.xda-developers.com/showthread.php?t=1687590>. [Online; accessed 5 Apr. 2015].
- [14] Chainfire. Pry-Fi, February 2014. URL <https://play.google.com/store/apps/details?id=eu.chainfire.pryfi>. [Online; accessed 4 Apr. 2015].
- [15] Chainfire. SuperSU – CF-Root download page, 2015. URL <http://download.chainfire.eu/696/SuperSU/UPDATE-SuperSU-v2.46.zip>. [Online; accessed 6 Mar. 2015].
- [16] Chainfire. [STABLE][2015.02.13] SuperSU v2.46, February 2015. URL <http://forum.xda-developers.com/showthread.php?t=1538053>. [Online; accessed 6 Mar. 2015].
- [17] Wolfgang Denk. Das U-Boot – the Universal Boot Loader, September 2014. URL <http://www.denx.de/wiki/U-Boot/WebHome>. [Online; accessed 4 Apr. 2015].
- [18] DooMLoRD. Android prebuilt toolchains, November 2011. URL [https://github.com/DooMLoRD/android\\_prebuilt\\_toolchains](https://github.com/DooMLoRD/android_prebuilt_toolchains). [Online; accessed 6 Apr. 2015].
- [19] Joshua J Drake, Zach Lanier, Collin Mulliner, Pau Oliva Fora, Stephen A Ridley, and Georg Wicherski. *Android Hacker's Handbook*. John Wiley & Sons, 2014.
- [20] eCos. RedBoot – a complete bootstrap environment for embedded systems, 2013. URL <http://ecos.sourceware.org/redboot/>. [Online; accessed 4 Apr. 2015].
- [21] Nikolay Elenkov. *Android Security Internals*. No Starch Press, 2015.
- [22] Eroot. Root has become so easy to make, 2013. URL [https://www.dropbox.com/s/jnwl5r3fnsssbuu/Eroot\\_V1.3.4.exe](https://www.dropbox.com/s/jnwl5r3fnsssbuu/Eroot_V1.3.4.exe). [Online; accessed 6 Apr. 2015].
- [23] AAA Foundation for Traffic Safety. Using naturalistic driving data to assess the prevalence of environmental factors and driver behaviors in teen driver crashes, March 2015. URL <https://www.aaafoundation.org/sites/default/files/2015TeenCrashCausationReport.pdf>. [Online; accessed 25 Mar. 2015].
- [24] Free Electrons. Linux Cross Reference – Linux/include/linux/timekeeper\_internal.h – Kernel Version 3.7, 2015. URL [http://lxr.free-electrons.com/source/include/linux/timekeeper\\_internal.h?v=3.7](http://lxr.free-electrons.com/source/include/linux/timekeeper_internal.h?v=3.7). [Online; accessed 30 Jan. 2015].
- [25] Free Electrons. Linux Cross Reference – Linux/kernel/time/timekeeping.c – Kernel Version 3.6, 2015. URL <http://lxr.free-electrons.com/source/kernel/time/timekeeping.c?v=3.6>. [Online; accessed 30 Jan. 2015].

- 
- [26] Free Electrons. Linux Cross Reference – Linux/kernel/time/timekeeping.c – Kernel Version 3.7, 2015. URL <http://lxr.free-electrons.com/source/kernel/time/timekeeping.c?v=3.7>. [Online; accessed 30 Jan. 2015].
- [27] Gartner Inc. Gartner Says Smartphone Sales Surpassed One Billion Units in 2014, March 2015. URL <https://www.gartner.com/newsroom/id/2996817>. [Online; accessed 26 Mar. 2015].
- [28] Google. Factory Images for Nexus Devices, March 2015. URL <https://developers.google.com/android/nexus/images>. [Online; accessed 7 Mar. 2015].
- [29] Google: Our mobile Planet. Activities on smartphones in 2013, March 2015. URL <http://goo.gl/22w7Ce>. [Online; accessed 26 Mar. 2015].
- [30] Google Security Research. Issue 222: Windows: Local WebDAV NTLM Reflection Elevation of Privilege, March 2015. URL <https://code.google.com/p/google-security-research/issues/detail?id=222>. [Online; accessed 27 Mar. 2015].
- [31] Christian Hilgers, Holger Macht, Tilo Müller, and Michael Spreitzenbarth. Post-mortem memory analysis of cold-booted android devices. In *IT Security Incident Management & IT Forensics (IMF), 2014 Eighth International Conference on*, pages 62–75. IEEE, 2014. URL <https://www1.informatik.uni-erlangen.de/filepool/publications/android.ram.analysis.pdf>. [Online; accessed 12 Apr. 2015].
- [32] HTC. File Search – Kernel Source Code, Binaries and Updates for HTC Android Phones, 2015. URL <http://www.htcdev.com/devcenter/downloads>. [Online; accessed 5 Apr. 2015].
- [33] Eric Huber. A Fistful of Dongles – Andrew Case’s Professional Biography, August 2014. URL <https://web.archive.org/web/20140828092903/http://www.ericjhuber.com/2014/08/afod-blog-interview-with-andrew-case.html>. [Online; accessed 20 Mar. 2014].
- [34] Hung, Jerry. UniversalAndroot – 1-click root for N1, August 2010. URL <http://forum.xda-developers.com/showthread.php?t=747598>. [Online; accessed 4 Apr. 2015].
- [35] Hunt, David. PiPhone – A Raspberry Pi based Smartphone, April 2014. URL <http://www.davidhunt.ie/piphone-a-raspberry-pi-based-smartphone/>. [Online; accessed 14 Apr. 2015].
- [36] IMDb. Werner Herzog – From One Second to the Next, 2013. URL <http://www.imdb.com/title/tt3108864/>. [Online; accessed 25 Mar. 2015].
- [37] International Data Corporation (IDC). Worldwide Quarterly Mobile Phone Tracker – Smartphone OS Market Share, Q4 2014, 2015. URL <http://www.idc.com/prodserv/smartphone-os-market-share.jsp>. [Online; accessed 24 Mar. 2015].

- [38] International Data Corporation (IDC). Worldwide Quarterly Mobile Phone Tracker – Smartphone Vendor Market Share, Q4 2014, 2015. URL <http://www.idc.com/prodserv/smartphone-market-share.jsp>. [Online; accessed 24 Mar. 2015].
- [39] Idan Kamara. explainshell.com – Write down a command-line to see the help text that matches each argument, 2015. URL <http://explainshell.com/>. [Online; accessed 10 Mar. 2015].
- [40] Mattias Kihlman. The Android boot process from power on, June 2009. URL <http://www.androidenea.com/2009/06/android-boot-process-from-power-on.html>. [Online; accessed 1 Oct. 2014].
- [41] Sebastian Krahmer. Droid2, August 2010. URL <http://c-skills.blogspot.co.uk/2010/08/droid2.html>. [Online; accessed 27 Mar. 2015].
- [42] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. John Wiley & Sons, 2014.
- [43] Carsten Maartmann-Moe. Inception, October 2011. URL <http://www.breaknenter.org/projects/inception/>. [Online; accessed 30 Dec. 2014].
- [44] Holger Macht. Live memory forensics on android with volatility. *Friedrich-Alexander University Erlangen-Nuremberg*, January 2013. URL [https://www1.informatik.uni-erlangen.de/filepool/publications/Live\\_Memory\\_Forensics\\_on\\_Android\\_with\\_Volatility.pdf](https://www1.informatik.uni-erlangen.de/filepool/publications/Live_Memory_Forensics_on_Android_with_Volatility.pdf). [Online; accessed 23 Mar. 2015].
- [45] Jeganathan Manikandan. Android Boot Time Optimization – Android boot process, September 2013. URL <http://embien.com/blog/android-boot-process-and-optimization/>. [Online; accessed 1 Oct. 2014].
- [46] Manjaro Linux. Manjaro Linux – A Linux distribution which is based on Arch Linux, 2014. URL [http://kent.dl.sourceforge.net/project/manjarolinux/release/0.8.11/xfce/manjaro-xfce-0.8.11-x86\\_64.iso](http://kent.dl.sourceforge.net/project/manjarolinux/release/0.8.11/xfce/manjaro-xfce-0.8.11-x86_64.iso). [Online; accessed 30 Dec. 2014].
- [47] Tilo Müller and Michael Spreitzenbarth. Frost: Forensic recovery of scrambled telephones. In *Applied Cryptography and Network Security, 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25–28, 2013, Proceedings*, pages 373–388. University of Calgary, 2013. URL <https://www1.cs.fau.de/filepool/projects/frost/frost.pdf>. [Online; accessed 17 Mar. 2015].
- [48] Nabeel. One Click Root [4.1.B.0.587/.431/ 4.0.2.A.0.62] All 2011 Xperias, April 2013. URL <http://forum.xda-developers.com/showthread.php?t=2219781>. [Online; accessed 7 Apr. 2015].
- [49] Openmoko. Qi – a lightweight replacement for the U-Boot bootloader, February 2012. URL <http://wiki.openmoko.org/wiki/Qi>. [Online; accessed 4 Apr. 2015].

- 
- [50] Michael Ossmann and Kyle Osborn. Multiplexed Wired Attack Surfaces, 2013. URL <https://greatscottgadgets.com/infiltrate2013/ossmann-osborn-bhusa2013-whitepaper.txt>. [Online; accessed 15 Apr. 2015].
- [51] Ketan Parmar. In Depth: Android Boot Sequence / Process, November 2012. URL <http://www.kpbird.com/2012/11/in-depth-android-boot-sequence-process.html>. [Online; accessed 1 Oct. 2014].
- [52] Pew Research Center. U.S. Smartphone Use in 2015, April 2015. URL [http://www.pewinternet.org/files/2015/03/PI\\_Smartphones\\_0401151.pdf](http://www.pewinternet.org/files/2015/03/PI_Smartphones_0401151.pdf). [Online; accessed 02 Apr. 2015].
- [53] Bitkom Research. 44 millionen deutsche nutzen ein smartphone (press release in german), March 2015. URL [http://www.bitkom-research.de/WebRoot/Store19/Shops/63742557/5512/871D/66D0/C7BE/CB7E/C0A8/2BB9/ABB5/BITKOM-Presseinfo\\_Smartphone\\_Nutzung\\_25\\_03\\_2015\\_final.pdf](http://www.bitkom-research.de/WebRoot/Store19/Shops/63742557/5512/871D/66D0/C7BE/CB7E/C0A8/2BB9/ABB5/BITKOM-Presseinfo_Smartphone_Nutzung_25_03_2015_final.pdf). [Online; accessed 26 Mar. 2015].
- [54] Matt Richtel. *A Deadly Wandering – A Tale of Tragedy and Redemption in the Age of Attention*. William Morrow/HarperCollins Publishers, 2014.
- [55] rpm.org. RPM container file format specification, August 2009. URL <http://www.rpm.org/wiki/DevelDocs/FileFormat>. [Online; accessed 30 Jan. 2015].
- [56] Patrick Schulz. Code protection in android. *Rheinische Friedrich-Wilhelms-Universität Bonn, Germany*, June 2012. URL [https://net.cs.uni-bonn.de/fileadmin/user\\_upload/plohmann/2012-Schulz-Code\\_Protection\\_in\\_Android.pdf](https://net.cs.uni-bonn.de/fileadmin/user_upload/plohmann/2012-Schulz-Code_Protection_in_Android.pdf). [Online; accessed 25 Mar. 2015].
- [57] SEMC Blog. Unofficial Sony Ericsson blog, 2015. URL [http://semcblog.com/other\\_codenames/](http://semcblog.com/other_codenames/). [Online; accessed 6 Apr. 2015].
- [58] Sony. Open source archive for build 4.1.B.0.587, August 2012. URL <http://developer.sonymobile.com/downloads/xperia-open-source-archives/open-source-archive-for-build-4-1-b-0-587/>. [Online; accessed 6 Apr. 2015].
- [59] Sony. PC Companion, 2015. URL <http://support.sonymobile.com/de/tools/pc-companion/>. [Online; accessed 6 Apr. 2015].
- [60] Spadgenske, Tyler. Build Your Own Smartphone, April 2015. URL <http://www.instructables.com/id/Build-Your-Own-Smartphone/>. [Online; accessed 14 Apr. 2015].
- [61] Pasquale Stirparo, Igor Nai Fovino, and Ioannis Kounelis. Data-in-use leakages from android memory—test and analysis. In *2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 701–708. IEEE, 2013. URL <http://www.computer.org/csdl/proceedings/wimob/2013/9999/00/06673433.pdf>. [Online; accessed 3 Apr. 2015].

- [62] Matthieu Suiche. MoonSols DumpIt goes mainstream!, July 2011. URL <http://www.moonsols.com/2011/07/18/moonsols-dumpit-goes-mainstream/>. [Online; accessed 29 Dec. 2014].
- [63] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Sushil Jajodia. Trustdump: Reliable memory acquisition on smartphones. In *Computer Security – ESORICS 2014 – 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7–11, 2014. Proceedings, Part I*, pages 202–218. Springer, 2014. URL <http://www.cs.wm.edu/~ksun/csci780-f14/papers/0-trustdump.pdf>. [Online; accessed 23 Feb. 2015].
- [64] Joe Sylve, Andrew Case, Lodovico Marziale, and Golden G Richard. Acquisition and analysis of volatile memory from android devices. *Digital Investigation*, 8(3): 175–184, February 2012. URL <http://www.504ensics.com/uploads/publications/android-memory-analysis-DI.pdf>. [Online; accessed 23 Mar. 2015].
- [65] Joseph T Sylve. Android memory capture and applications for security and privacy. *University of New Orleans*, December 2011. URL <http://scholarworks.uno.edu/cgi/viewcontent.cgi?article=2348&context=td>. [Online; accessed 23 Mar. 2015].
- [66] Team Win. TWRP 2.8, February 2015. URL <http://teamw.in/project/twrp2>. [Online; accessed 6 Mar. 2015].
- [67] Team Win. TWRP2 for Crespo, February 2015. URL <http://techerrata.com/browse/twrp2/crespo>. [Online; accessed 6 Mar. 2015].
- [68] The Debian Project. Download Debian 7.7.0 Wheezy stable CD 1, 2014. URL <http://cdimage.debian.org/debian-cd/7.7.0/amd64/iso-cd/debian-7.7.0-amd64-CD-1.iso>. [Online; accessed 30 Dec. 2014].
- [69] The Fedora Project. Download Fedora 21 Workstation, 2014. URL [http://download.fedoraproject.org/pub/fedora/linux/releases/21/Workstation/x86\\_64/iso/Fedora-Live-Workstation-x86\\_64-21-5.iso](http://download.fedoraproject.org/pub/fedora/linux/releases/21/Workstation/x86_64/iso/Fedora-Live-Workstation-x86_64-21-5.iso). [Online; accessed 30 Dec. 2014].
- [70] The Fedora Project. Buildsystem: Information for build kernel-3.17.8-300.fc21, 2015. URL <http://koji.fedoraproject.org/koji/buildinfo?buildID=603039>. [Online; accessed 26 Jan. 2015].
- [71] Vrizlynn LL Thing, Kian-Yong Ng, and Ee-Chien Chang. Live memory forensics of mobile phones. *Digital Investigation*, 7:S74–S82, August 2010. URL <http://dfwrws.org/2010/proceedings/2010-309.pdf>. [Online; accessed 23 Mar. 2015].
- [72] Volatility Foundation. Volatility profiles for Linux and Mac OS X, 2014. URL <https://github.com/volatilityfoundation/profiles>. [Online; accessed 1 Jan. 2015].

- 
- [73] Volatility Foundation. An advanced memory forensics framework <http://volatilityfoundation.org/>, 2014. URL <https://codeload.github.com/volatilityfoundation/volatility>. [Online; accessed 1 Jan. 2015].
- [74] Wächter, Philipp. Samsung gt-i9023 google nexus s memory acquisition and analysis, August 2014. URL <http://lists.volatilesystems.com/pipermail/vol-users/2014-August/001281.html>. [Online; accessed 20 Mar. 2015].
- [75] Wächter, Philipp. Volatility: Fix for kernel 3.7+ in linux/module.c, February 2015. URL <https://github.com/volatilityfoundation/volatility/commit/65cbc6681b1afbdc62f46bf4e61277495d30a18>. [Online; accessed 2 Feb. 2015].
- [76] Wächter, Philipp. Volatility: Issue #161 — linux\_pslist with Fedora 21, January 2015. URL <https://github.com/volatilityfoundation/volatility/issues/161>. [Online; accessed 4 Apr. 2015].
- [77] Wikipedia. Zygote – wikipedia, the free encyclopedia, 2014. URL <http://en.wikipedia.org/w/index.php?title=Zygote&oldid=623694822>. [Online; accessed 12 Oct. 2014].
- [78] Christos Xenakis and Christoforos Ntantogian. Acquisition and Analysis of Android Memory (CyNC 2013 – Cybercrime Network Conference – 8th–10th December, 2013 at the O’Brien Centre for Science, UCD, Dublin, Ireland), December 2013. URL <http://www.ucd.ie/cci/cync/Acquisition%20and%20Analysis%20of%20Android%20Memory.pdf>. [Online; accessed 23 Mar. 2015].
- [79] Zalewski, Michal. Memfetch is a simple utility to dump all memory of a running process, October 2003. URL <http://lcamtuf.coredump.cx/soft/memfetch.tgz>. [Online; accessed 3 Apr. 2015].
- [80] Lenny Zeltser. How miscreants hide from browser forensics, March 2015. URL <http://digital-forensics.sans.org/blog/2015/03/24/hide-from-browser-forensics>. [Online; accessed 25 Mar. 2015].
- [81] zxz0O0. giefroot — A tool to root your device using CVE-2014-7911 (by Keen Team) and CVE-2014-4322 (by zxz0O0), January 2015. URL <http://forum.xda-developers.com/crossdevice-dev/sony/giefroot-rooting-tool-cve-2014-4322-t3011598>. [Online; accessed 2 Apr. 2015].

