A Tool for Volatile Memory Acquisition from Android Devices

Haiyu Yang, Jianwei Zhuge, Huiming Liu, Wei Liu

Chapter 19

A TOOL FOR VOLATILE MEMORY ACQUISITION FROM ANDROID DEVICES

Haiyu Yang, Jianwei Zhuge, Huiming Liu and Wei Liu

Abstract Memory forensic tools provide a thorough way to detect malware and investigate cyber crimes. However, existing memory forensic tools must be compiled against the exact version of the kernel source code and the exact kernel configuration. This poses a problem for Android devices because there are more than 1,000 manufacturers and each manufacturer maintains its own kernel. Moreover, new security enhancements introduced in Android Lollipop prevent most memory acquisition tools from executing.

This chapter describes AMExtractor, a tool for acquiring volatile physical memory from a wide range of Android devices with high integrity. AMExtractor uses `/dev/kmem` to execute code in kernel mode, which is supported by most Android devices. Device-specific information is extracted at runtime without any assumptions about the target kernel source code and configuration. AMExtractor has been successfully tested on several devices shipped with different versions of the Android operating system, including the latest Android Lollipop. Memory images dumped by AMExtractor can be exported to other forensic frameworks for deep analysis. A rootkit was successfully detected using the Volatility Framework on memory images retrieved by AMExtractor.

Keywords: Mobile device forensics, memory forensics, Android, rootkit detection

1. Introduction

The Android operating system is the most popular smartphone platform with a market share of 82.8% in Q2 2015 [7]. The popularity of the operating system makes it vital for digital forensic investigators to acquire and analyze evidence from Android devices. Most digital forensic tools and frameworks focus on extracting user data and metadata from the Android filesystem instead of volatile memory. However, new security enhancements, such as full-disk encryption introduced in Android Ice Cream (version 4.0), make it extremely difficult to recover evidence via filesystem forensics [2].

Volatile memory is valuable because it contains a wealth of information that is otherwise unrecoverable. The evidence in volatile memory includes objects related to running and terminated processes, open files, network activity, memory mappings and more [1]. This evidence could be extracted directly if a full physical memory dump were to be obtained. Often, a full copy of volatile memory is the first, but essential, step in advanced Android forensics and threat analysis.

Volatile memory acquisition from Android devices is challenging. A major challenge is the fragmentation of Android devices – there are more than 24,000 distinct Android devices and 1,294 manufacturers [13]. This fragmentation introduces flaws in Android memory acquisition tools:

- **Availability:** Memory acquisition tools do not work on several devices because they lack certain functionality. LiME, the most popular tool in the Android community, relies on loadable kernel module (LKM) support by target devices. However, many Android devices do not provide this functionality. For example, Google Nexus smartphones do not support loadable kernel modules – attempting to load a kernel module using the `insmod` command produces a "function not implemented" error. Loadable kernel module support is a compile-time option and enabling it requires the kernel to be compiled and the boot partition flashed. Some manufacturers incorporate security enhancement mechanisms that prevent unofficial kernel modules from running. For example, Samsung Galaxy has KNOX that only allows kernel modules with Samsung signatures to be loaded.

- **Compatibility:** It is very difficult to port some memory acquisition tools to new devices. For example, LiME must be compiled against the exact version of the target kernel source code and the exact kernel configuration. However, these conditions are not always met because most mobile phone manufacturers do not release their source code.

- **Accuracy:** Some memory acquisition tools have large forensic impacts on target devices, producing evidence that may not be admissible in court. For example, the `fmem` memory acquisition tool for Linux systems can be used to copy data from kernel mode to user mode. However, it involves frequent copying that may override memory content and contaminate the memory. Other tools that work in user mode only have access to the memory of particular processes; thus, they are incapable of detecting rootkits.

This chapter describes AMExtractor (Android Memory Extractor), a widely applicable tool for acquiring volatile memory from Android devices. AMExtractor has three advantages compared with existing tools. First, AMExtractor uses the `/dev/kmem` device to execute code in kernel space; this bypasses the loadable kernel module restriction and works well on the latest stock ROMs without any modifications. Second, AMExtractor does not need the source code of the target device and it is compatible with most Android operating system versions, including the latest Lollipop. Third, AMExtractor runs in kernel mode. This makes the tool forensically sound – it has minimal impact on target devices because it reads and transmits memory content only in kernel mode and minimizes data copying. Unlike tools that run in user mode, AMExtractor can find information hidden from user mode and it is not affected by rootkits.

AMExtractor was tested on four mobile phone models: (i) Samsung Galaxy Nexus; (ii) LG Nexus 4; (iii) LG Nexus 5; and (iv) Samsung Galaxy S4. Different versions of stock and third-party ROMs as well as the latest stock firmware were tested without any failures. AMExtractor was evaluated by comparing it against LiME and `fmem`. The results demonstrate that the dumped memory is nearly the same as that obtained with LiME. The AMExtractor output was also exported to the Volatility Framework to detect the presence of rootkits. Evidence of malware invisible to traditional security tools was discovered.

2. Related Work

Traditional memory acquisition methods can be classified as: (i) hardware methods; and (ii) software methods [5].

2.1 Hardware Methods

JTAG test pins can be used to retrieve the internal memory of a device. This method was verified on the Nokia 5110 model by Willassen, [20]. However, over and above the difficulty of directly programming JTAG to acquire live memory dumps, not all Android devices have JTAG test pins. Muller developed the FROST framework [12] to retrieve sensitive information, including disk encryption keys from memory using a cold attack. Specifically, FROST is able to read the remaining memory content after a mobile phone maintained at a low temperature is powered off. The limitation of FROST is that it is necessary to unlock the phone and flash the recovery image; this causes the phone to be reset to the factory settings with the loss of all user data. Moreover, FROST relies on the data remanence property to read memory; thus, the memory retrieved may not be an exact copy of the live memory.

2.2 Software Methods

Traditionally, memory content can be acquired from `/dev/mem` devices. However, this approach does not work for mobile phones with RAM in excess of 896 MB [17]. Kollar [8] has developed `fmem`, a loadable kernel module that creates a `/dev/fmem` device supporting memory acquisition. Unfortunately, `fmem` does not work on Android devices by default [17]. Additionally, reading memory from such devices in user space involves too many interactions between user space and kernel space, which can modify the original content.

Sun et al. [16] have implemented a reliable memory acquisition tool called TrustDump. TrustDump is a TrustZone-based memory acquisition tool that can extract the RAM memory and CPU register values of a mobile device even when the operating system has crashed or has been compromised. However, TrustDump is only supported on Freescale i.MX53 QSB, an embedded development board, making it an impractical tool for evidence acquisition from Android devices.

LiME [17] is another popular forensic tool. It is a loadable kernel module that parses the memory mapping structure in a kernel and dumps the memory content to an SD card or transmits it over a TCP connection. By using a direct I/O or kernel socket technique, LiME minimizes its interactions with user and kernel space, thereby providing a more forensically-sound memory dump. LiME uses a custom format to reduce the size of memory images, a feature supported by other memory analysis tools.

However, LiME has some shortcomings. Its portability across a range of smartphone models poses problems with regard to memory forensics [17]. When attempting to load a kernel module, if module verification is enabled (true for every kernel tested in this work), the kernel performs several checks to ensure that the module was compiled for the specific version of the running kernel. LiME needs the kernel source code of the target phone. This constraint cannot always be satisfied because manufacturers tend to delay the publication of source code or never publish their code. However, even if the source code is available, the kernel configuration and toolchains must be exactly the same as for the stock firmware running on the phone. Any change prevents the module from being loaded due to a CRC checksum mismatch. Recompiling

Table 1. Volatile memory acquisition support by commercial tools.

Products	**Status**
Cellebrite	No support, but support is planned
XRY	No support
Oxygen Forensic	No support
Magnet IEF	No support

the kernel and flashing it to a phone appears to be the best approach, but changing the original system is not always acceptable, especially in forensic research. Even worse, recently released phones such as the Google Nexus and Samsung Galaxy series, by default, turn off the loadable kernel module compiling option in their Linux kernels.

Stuttgen and Cohen [15] have proposed a method for loading a module into a Linux kernel without kernel source code by developing a truly version-independent kernel module and modifying it prior to loading. Using the checksum and kernel information retrieved from another existing kernel module, they were able to dynamically modify the module to perform a robust memory acquisition. However, their method is not widely applicable due to the lack of existing kernel modules in Android devices.

With regard to code injection techniques, `devik` and `sd` [4] have proposed a method that dynamically patches a Linux kernel without using a loadable kernel module. Lineberry [9] has presented a similar method. The methods focus on hooking the `syscall` table instead of reading memory content. AMExtractor follows this approach, but modifies it to hook device drivers.

2.3 Commercial Memory Forensics Tools

At this time, the major commercial digital forensic tools do not support volatile memory acquisition from mobile phones. Table 1 shows the status of memory acquisition support by popular commercial tools.

3. AMExtractor Design

Figure 1 presents the AMExtractor architecture. AMExtractor is an ELF file that runs in user space, but requires root privileges to execute code at the kernel level.

Root privileges are indispensable to using AMExtractor. An existing root manager or kernel bug exploit may be used to root Android devices. This prerequisite is relatively easy to satisfy because many users root

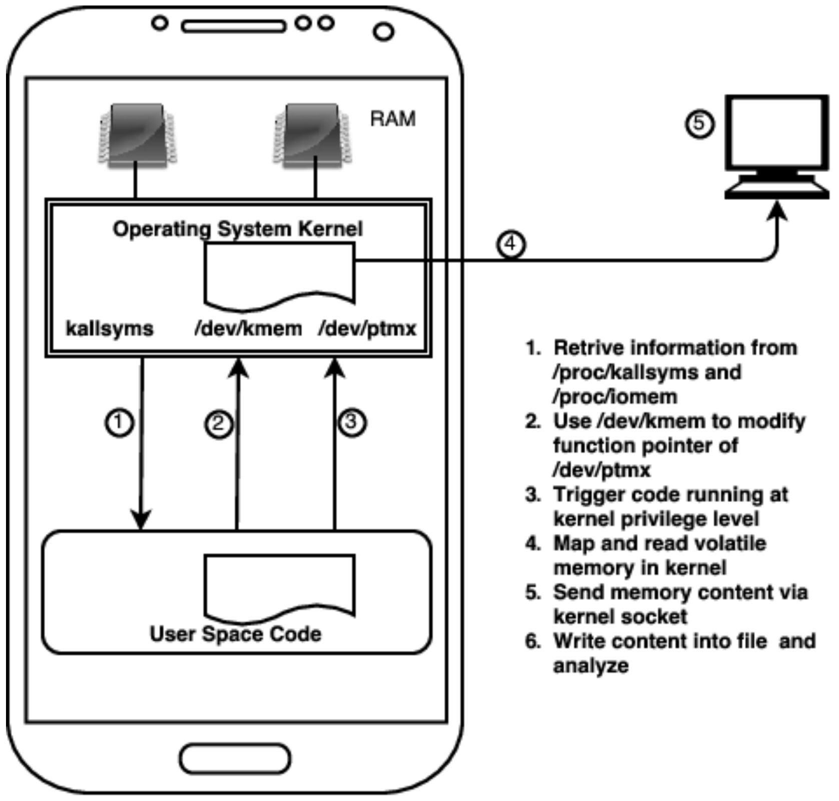
RAM
Operating System Kernel
kallsyms
/dev/kmem
/dev/ptmx
User Space Code
1
2
3
4
5
1. Retrive information from /proc/kallsyms and /proc/iomem
2. Use /dev/kmem to modify function pointer of /dev/ptmx
3. Trigger code running at kernel privilege level
4. Map and read volatile memory in kernel
5. Send memory content via kernel socket
6. Write content into file and analyze

Figure 1. AMExtractor architecture.

their devices [10]. Moreover, many root solutions have been published; at least two universal solutions were presented recently [21, 22] In any case, methods for obtaining root privileges are outside the scope of this research.

The AMExtractor memory dump process involves the following steps:

- **Step 1: Retrieve Information:** AMExtractor reuses some kernel facilities for robustness and compatibility. Information about the running system, including physical memory offset and kernel symbols, is retrieved at runtime. The kernel function address and physical memory layout are retrieved using `/proc/kallsyms` and `/proc/iomem`. Disassembly of the `vm_normal_page` function is also required.

- **Step 2: Modify the Function Pointer:** Having gathered enough system information, AMExtractor needs to modify only one byte of the target kernel. The function pointer points to custom code developed by the authors of this chapter and this function is invoked by a device operation such as `fsync` on `/dev/ptmx`. Note that `/dev/kmem` was chosen to modify the kernel because it is supported by all Android and Linux versions. The impact on the target system caused by the modification is small because only one byte of the target kernel is modified.

- **Step 3: Trigger Code Running at the Kernel Privilege Level:** With the function pointer pointing to the custom code, calling the particular method on a modified device leads to the custom code executing at the kernel privilege level. Unused file operators exposed by the kernel are chosen for minimal changes to the kernel (e.g., operators of `/dev/ptmx`, `/dev/zero` and `/dev/null`). Experiments revealed that the `fsync` operator of `/dev/ptmx` could be used in most cases.

- **Step 4: Map and Read Volatile Kernel Memory:** When the custom code executes at the kernel privilege level, it enumerates pages using the memory management facilities provided by the kernel. Upon enumerating the managed resource in `iomem_resource`, all the system RAM can be found with the starting and ending addresses. For each page in the address range of system RAM, AMExtractor translates the page frame number into the virtual address. The virtual address can be used by a socket read/write function. The extraction process is practically the same as that of LiME. The only difference is that AMExtractor does not rely on the source code of the target device kernel while LiME must be compiled against it.

- **Step 5: Transmit Memory Content via a Kernel Socket:** A kernel socket is used to transmit memory content instead of copying data to user mode. This method minimizes interactions between user space and kernel space, and, thus, has a minimal impact on the target system. The link to a personal computer for memory content transmission can be created by an ADB bridge or Wi-Fi connection and the content sent out via TCP/IP.

- **Step 6: Write Memory Content to a File and Analyze the Content:** The memory content can be received by and written to a file on a separate personal computer for further analysis. The output format of AMExtractor is compatible with many memory analysis toolkits, including Volatility [19].

4. Implementation

This section presents details about the AMExtractor implementation. Using `/dev/kmem` instead of a loadable kernel module strengthens the tool, but introduces some challenges. This section describes these challenges and the methods used to solve them.

4.1 Gathering Information

The first step is to dynamically read the kernel symbols. This is not required for tools using loadable kernel modules because they have already been compiled against the source code. However, AMExtractor needs to know the function addresses before it can use `/dev/kmem`. Fortunately, the Android kernel has a symbol table exported in `/proc/kallsyms` that provides enough information.

The content of `/proc/kallsyms` is a simple plaintext file that is easy to parse:

```
c4508000 T stext
c4508000 T _sinittext
c4508000 T _stext
c4508000 T __init_begin
c450805c t __create_page_tables
c4508060 t __enable_mmu_loc
...
...
```

However, the kernels of most phones restrict the kernel pointer addresses from being printed. Fortunately, `/proc/sys/kernel/kptr_restrict` can be used to turn off this restriction.

4.2 Using /dev/kmem to Deploy the Trigger

While each loadable kernel module has a well-defined entry, tools based on `/dev/kmem` do not. Therefore, a technique for triggering code in kernel space is needed. AMExtractor deploys the trigger by modifying one function pointer to the custom code. The triggering process proceeds as follows:

- Find a device created by a Linux kernel for which `struct file_operations` is writable.
- Modify the function pointer of the structure to the custom code.
- Trigger the custom code by calling the corresponding device function.

AMExtractor uses the `/dev/ptmx` device. This device has an operation structure that is always writable because the method of the device is assigned during booting. For certain versions of the Android kernel, `/dev/zero` and `/dev/null` are suitable alternatives.

4.3 Running Code at the Kernel Privilege Level

Triggering the custom code at the kernel privilege level is straightforward. All that is needed is to open the modified device in the previous step and perform the operation, i.e., call `fsync()` on `/dev/ptmx`.

4.4 Mappping and Reading Kernel Memory

This process is nearly the same as that of LiME. When enumerating pages, the `iomem_resource` structure is traversed, the page frame number is translated to the page pointer and the page is mapped to a virtual address. The difference between AMExtractor and LiME lies in the translation method. Tools based on loadable kernel modules can perform the translation using `pfn_to_page`. However, `pfn_to_page` is a macro that is compiled (inline) into other functions. Therefore, it is impossible to reuse `pfn_to_page` in the target kernel and it is necessary to re-implement the logic of the macro. A hard coded implementation of the macro is also infeasible because there are three different memory models in a Android kernel, corresponding to the different implementations of `pfn_to_page`. It is infeasible to enumerate the three implementations and identify the correct one because a wrong choice causes kernel panic. The problem is solved as follows:

- Find a function that contains the `pfn_to_page` macro.
- Reverse-engineer the binary code of the function.
- Re-implement the logic of the macro.

The following code snippet is the disassembled output of the `pfn_to_page` macro in IDA Pro:

```
C024F344 loc_out
C024F344    LDR     R3, =0xC122AFC0
            ; load address of memory_map into r0
C024F348    LDR     R0, [R3]
C024F34C    ADD     R0, R0, R4, LSL #5
            ; r4 contains page frame number
            ; sizeof(struct page) == 32
C024F350    LDMFD   SP, {R4,R5, R11, SP, PC}; return
C024F354
```

Table 2. Tests of forensic soundness.

Device	**ROM**	**Version**	**LKM**	**LiME**	**AMExtractor**
Galaxy Nexus	Stock ROM	4.3	No	Failed	Successful
Galaxy Nexus	Paranoid	4.4.4	Yes	Success	Successful
Nexus 4	Stock ROM	4.2.2	No	Failed	Successful
Nexus 4	Stock ROM	5.1.1	No	Failed	Successful
Nexus 5	Stock ROM	4.4.4	No	Failed	Successful
Nexus 5	Stock ROM	5.1.1	No	Failed	Successful
Nexus 5	Self-compiled	5.0	Yes	Successful	Successful
Galaxy S4	Stock ROM	5.0	Yes	Failed	Successful

Obviously, the device uses a flat memory model and the size of the structure page is 32.

4.5 Transmitting Memory Content

Using the page pointer provided by `pfn_to_page`, calling `kmap` maps the page with a virtual address that is useful in the kernel socket. Functions `sock_create_kern` and `kernel_sendmsg` can then be used in the kernel without copying data to user space.

5. Experimental Evaluation

Experiments were conducted using AMExtractor on various devices and the extracted memory was analyzed. The experimental results demonstrate that AMExtractor has wide applicability on the latest firmware versions. Also, the extracted memory was successfully analyzed to detect rootkit activities; this is not possible using traditional user space tools.

5.1 Applicability Evaluation

AMExtractor was tested on four phone models: Galaxy Nexus, Nexus 4, Nexus 5 and Samsung Galaxy S4. Various versions of stock ROMs and third-party ROMs were included in the experiments. Also, a custom kernel with a loadable kernel module option was compiled to test LiME.

As shown in Table 2, AMExtractor successfully dumped the memory contents of all the phones shipped with latest ROMs. The Samsung Galaxy S4 with KNOX enabled was also tested. LiME only succeeded on the self-compiled ROM of Nexus 5 and one third-party ROM of Galaxy Nexus. Although the Galaxy S4 enabled a loadable kernel module in stock ROM, the security enhancements provided by KNOX prevented

Table 3. Tests of forensic soundness.

Acquisition Tool	Number of Pages	Identical Percentage
AMExtractor	484,096	99.06%
LiME	484,096	99.46%
`fmem`	484,096	80.17%

an unofficial kernel module from being loaded. In contrast, AMExtractor worked well even when KNOX was operational.

5.2 Integrity Evaluation

Forensic soundness is a critical criterion for evaluating a memory forensic tool. Comparisons of the memory dumped by different tools serves as a good proof of soundness. In the evaluation, the dumped memory contents were compared against the memory of the Android emulator to demonstrate integrity. In order to make LiME and `fmem` work properly, a self-compiled kernel was flashed to Nexus 5. Table 3 presents the results.

Although the memory contents may change during the long dumping process, the memory contents dumped by AMExtractor and LiME were nearly the same. This is not surprising because the same approaches are used to enumerate pages and transmit the memory contents.

6. In-Depth Analysis of Extracted Memory

Volatile memory is valuable in forensic investigations. Efforts have been made to extract information and evidence from volatile memory dumps. A promising application is rootkit detection. Modern malware often uses kernel-level techniques to hide their activities. Kernel rootkits run with the highest operating system privileges. These rootkits can modify the interactions between user mode and kernel mode to cloak themselves. The following code snippet shows a typical Android rootkit that hides its file and process:

```
shell@hammerhead:/ # ps | grep wmr
1|root@hammerhead:/ #
```

Rootkit detection is difficult because a rootkit may be able to subvert the software intended to find it. The sample rootkit above can hide itself when a user issues the `ps` command to list suspicious processes. However, analysis of the full memory dump provided by AMExtractor can reveal evidence of malware activities. Specifically, in the rootkit example above, the package name of the rootkit cannot be found in the `ps` output. However, the rootkit is revealed when the memory extracted by AMExtractor is analyzed:

```
user@PC:~$ vol.py --profile=LinuxGNARM -f ./dump_memory
linux_pslist | grep wmr
Volatility Foundation Volatility Framework 2.4
0xc61d1a40 com.mwr.dz          1657   10064     10064
     0x85170000 2015-09-01 07:24:01
0xc510f840 m.mwr.dz:remote     1670   10064     10064
     0x85dc4000 2015-09-01 07:24:01
```

Note that no processes with the string "`wmr`" in their names are listed because the rootkit has hidden them. However, when Volatility is used to analyze the AMExtractor memory dump, the processes become visible:

```
user@PC:~$ vol.py --profile=LinuxGNARM -f ./dump_memory
linux_check_syscall_arm
Volatility Foundation Volatility Framework 2.4
/*---------------omitted----------------*/
  0xd7 0xc00cdfd0 sys_setfsuid
  0xd8 0xc00ce0ac sys_setfsgid
  0xd9 0xbf004000 HOOKED
  0xda 0xc0188024 sys_pivot_root
  0xdb 0xc0150e44 sys_mincore
  0xdc 0xc014cb38 sys_madvise

/*---------------omitted----------------*/
```

Furthermore, `sys_call_table` can be analyzed using the memory dump provided by AMExtractor. As seen in the output above, `sys_getdents64` was compromised by malware.

Although AMExtractor can be applied to a wide variety of Android devices, the tool has some limitations. The code executed in the kernel is located in user space. The control flow of the kernel may be redirected to custom code by modifying a function pointer. ARM CPUs provide the *Privileged Execute Never* (PXN) permission to prevent such activities. If a target kernel were to fully utilize PXN functionality, AMExtractor would fail. Moreover, AMExtractor relies on `/proc/kallsyms` and `/dev/kmem`. Fortunately, most Android devices, including all the devices tested, do not use PXN [6].

7. Conclusions

The AMExtractor tool is designed to acquire volatile physical memory from Android devices with the latest ROMs and firmware. The tool utilizes `/dev/kmem` to perform memory extractions with high integrity and better applicability than existing tools. Memory images dumped by AMExtractor may be exported to other forensic frameworks for deep analysis. Additionally, AMExtractor supports rootkit detection. In an experiment, a rootkit was successfully detected using the Volatility Framework on volatile memory retrieved by AMExtractor.

Acknowledgement

This research was supported by the National Natural Science Foundation of China under Grant No. 61472209 and by the Tsinghua University Initiative Scientific Research Program (20151080436).

References

[1] D. Apostolopoulos, G. Marinakis, C. Ntantogian and C. Xenakis, Discovering authentication credentials in volatile memory of Android mobile devices, *Proceedings of the Twelfth IFIP WG 6.11 Conference on e-Business, e-Services and e-Society*, pp. 178–185, 2013.

[2] K. Barmpatsalou, D. Damopoulos, G. Kambourakis and V. Katos, A critical review of seven years of mobile device forensics, *Digital Investigation*, vol. 10(4), pp. 323–349, 2013.

[3] T. Cannon and S. Bradford, Into the droid: Gaining access to Android user data, presented at the *Defcon Hacking Conference*, 2012.

[4] `devik` and `sd`, Linux on-the-fly kernel patching without LKM, *Phrack*, vol. 11(58), 2001.

[5] G. Garcia, Forensic physical memory analysis: An overview of tools and techniques, presented at the *TKK T-110.5290 Seminar on Network Security*, 2007.

[6] X. Ge, H. Vijayakumar and T. Jaeger, Sprobes: Enforcing kernel code integrity in the TrustZone architecture, presented at the *Third Workshop on Mobile Security Technologies*, 2014.

[7] International Data Corporation, Smartphone OS market share, 2015 Q2, Framington, Massachusetts (`www.idc.com/prodserv/smartphone-os-market-share.jsp`), 2015.

[8] I. Kollar, Forensic RAM Dump Image Analyzer, Master's Thesis, Department of Software Engineering, Charles University in Prague, Prague, Czech Republic, 2009.

[9] A. Lineberry, Malicious code injection via `/dev/mem`, presented at the *Black Hat Europe Conference*, 2009.

[10] K. Lucic, Over 27.44% users root their phone(s) in order to remove built-in apps, *Android Headlines*, Valencia, California, November 13, 2014.

[11] H. Macht, Live Memory Forensics on Android with Volatility, Diploma Thesis in Computer Science, Department of Computer Science, Friedrich-Alexander University Erlangen-Nuremberg, Erlangen, Germany, 2013.

[12] T. Muller and M. Spreitzenbarth, FROST – Forensic recovery of scrambled telephones, *Proceedings of the Eleventh International Conference on Applied Cryptography and Network Security*, pp. 373–388, 2013.

[13] OpenSignal, Android fragmentation visualized, London, United Kingdom (`opensignal.com/reports/2015/08/android-fragmentation`), 2015.

[14] J. Park and S. Choi, Studying security weaknesses of Android systems, *International Journal of Security and its Applications*, vol. 9(3), pp. 7–12, 2015.

[15] J. Stuttgen and M. Cohen, Robust Linux memory acquisition with minimal target impact, *Digital Investigation*, vol. 11(S1), pp. S112–S119, 2014.

[16] H. Sun, K. Sun, Y. Wang, J. Jing and S. Jajodia, TrustDump: Reliable memory acquisition from smartphones, *Proceedings of the Nineteenth European Symposium on Research in Computer Security*, pp. 202–218, 2014.

[17] J. Sylve, A. Case, L. Marziale and G. Richard, Acquisition and analysis of volatile memory from Android devices, *Digital Investigation*, vol. 8(3-4), pp. 175–184, 2012.

[18] V. Thing, K. Ng and E. Chang, Live memory forensics of mobile phones, *Digital Investigation*, vol. 7(S), pp. S74–S82, 2010.

[19] Volatility Foundation, Volatility Framework (`www.volatilityfoundation.org`), 2016.

[20] S. Willassen, Forensic analysis of mobile phone internal memory, in *Advances in Digital Forensics*, M. Pollitt and S. Shenoi, Springer, Boston, Massachusetts, pp. 191–204, 2005.

[21] W. Xu, Ah! Universal Android rooting is back, presented at the *Black Hat USA Conference*, 2015.

[22] W. Xu and Y. Fu, Own your Android! Yet another universal root, presented at the *Ninth USENIX Workshop on Offensive Technologies*, 2015.