DiSIT, Computer Science Institute
Università del Piemonte Orientale “A. Avogadro”
Viale Teresa Michel 11, 15121 Alessandria
http://www.di.unipmn.it

Configuration and Use of Android Virtual Devices for the Forensic Analysis of Android Applications (see below for citation details)

C. Anglano, M. Canonico, M. Guazzone (cosimo.anglano@uniupo.it, massimo.canonico@uniupo.it, marco.guazzone@uniupo.it)

TECHNICAL REPORT TR-INF-2017-06-02-UNIPMN
(June 2017)

Research Technical Reports published by DiSIT, Computer Science Institute, Università del Piemonte Orientale are available via WWW at URL http://www.di.unipmn.it/.
Plain-text abstracts organized by year are available in the directory

Recent Titles from the TR-INF-UNIPMN Technical Report Series

2017-01 *A dynamic simulation model for comparing kidney exchange policies*, M. Beccuti, G. Franceschinis, S. Villa, March 2017.

2016-04 *Tracing sharing in an imperative pure calculus*, P. Giannini, M. Servetto, E. Zucca, December 2016.

2016-03 *SUPPORTING DATA COMMUNICATION AND PATIENT ASSESSMENT DURING EMERGENCY TRANSPORTATION*, M. Canonico, S. Montani, M. Striani, September 2016.

2016-02 *TECHNICAL NOTE TO Forensic Analysis of the ChatSecure Instant Messaging Application on Android Smartphones (see below for citation details)*, C. Anglano, M. Canonico, M. Guazzone, September 2016.

2016-01 *Reasoning in a rational extension of SROEL*, L. Giordano, D. Theseider Dupré, May 2016.

2014-02 *A Provenly Correct Compilation of Functional Languages into Scripting Languages*, P. Giannini, A. Shaqiri, December 2014.

2014-01 *An Intelligent Swarm of Markovian Agents*, A. Bobbio, D. Bruneo, D. Cerotti, M. Gribaudo, M. Scarpa, June 2014.

2013-01 *Minimum pattern length for short spaced seeds based on linear rulers (revised)*, L. Egidi, G. Manzini, July 2013.

2012-04 *An intensional approach for periodic data in relational databases*, A. Bottrighi, A. Sattar, B. Stantic, P. Terenziani, December 2012.

2012-03 *Minimum pattern length for short spaced seeds based on linear rulers*, L. Egidi, G. Manzini, April 2012.

2012-02 *Exploiting VM Migration for the Automated Power and Performance Management of Green Cloud Computing Systems*, C. Anglano, M. Canonico, M. Guazzone, April 2012.

2012-01 *Trace retrieval and clustering for business process monitoring*, G. Leonardi, S. Montani, March 2012.

2011-04 *Achieving completeness in bounded model checking of action theories in ASP*, L. Giordano, A. Martelli, D. Theseider Dupré, December 2011.

2011-03 *SAN models of a benchmark on dynamic reliability*, D. Codetta Raiteri, December 2011.

2011-02 *A new symbolic approach for network reliability analysis* , M. Beccuti, S. Donatelli, G. Franceschinis, R. Terruggia, June 2011.

2011-01 *Spaced Seeds Design Using Perfect Rulers*, L. Egidi, G. Manzini, June 2011.

Configuration and Use of Android Virtual Devices for the Forensic Analysis of Android Applications

TECHNICAL NOTE TO

Forensic Analysis of Telegram Messenger on Android Smartphones

(Please cite:

Cosimo Anglano, Massimo Canonico, Marco Guazzone,

"Forensic Analysis of Telegram Messenger on Android Smartphones,"
Submitted for publication.)

Configuration and Use of Android Virtual Devices for the Forensic Analysis of Android Applications[☆,☆☆]

Cosimo Anglano[a], Massimo Canonico[a], Marco Guazzone[a]

(Please, cite as:
Cosimo Anglano, Massimo Canonico, Marco Guazzone,
"Forensic Analysis of Telegram Messenger on Android Smartphones,"
Submitted for publication.)

[a] *DiSIT - Computer Science Institute,*
University of Piemonte Orientale, Alessandria (Italy)

Abstract

We present the forensic analysis of the artifacts generated on Android smartphones by *Instant Messaging* applications, like *Telegram Messenger* and *ChatSecure* applications. *Telegram Messenger* is the official client for the Telegram instant messaging platform, which provides various forms of secure individual and group communication, by means of which both textual and non-textual messages can be exchanged among users, as well as voice calls. *ChatSecure* is a *secure* instant messaging application that provides strong encryption for transmitted and locally-stored data to ensure the privacy of its users.

Specifically, in this technical report, we focus on the *ChatSecure* application as a case study, and we describe how to concretely configure and use the various tools that we rely upon to create and run an *Android Virtual Device* (*AVD*), and to carry out the analysis of its persistent and volatile memory.

Keywords: Mobile forensics, Android, Instant Messaging, Virtual Device

[☆] Please, cite as: *Cosimo Anglano, Massimo Canonico, Marco Guazzone, "Forensic Analysis of Telegram Messenger on Android Smartphones," Submitted for publication.*

[☆☆] Link to publisher: `http://dx.doi.org/10.1016/j.diin.2016.10.001`

Email addresses: `cosimo.anglano@uniupo.it` (Cosimo Anglano), `massimo.canonico@uniupo.it` (Massimo Canonico), `marco.guazzone@uniupo.it` (Marco Guazzone)

1. Configuring and using the Android Emulator, LiME, and Volatility

In this technical report, we illustrate how to configure and use the software tools used to carry out the analysis methodology described in this work, and in particular the *Android Mobile Device Emulator* (AMDE) to create and manage Android Virtual Devices (AVD) (1.1), LiME (1.2) to dump the contents of volatile memory, and Volatility (1.3) to analyze these dumps. We focus on the ARM architecture sinceit is the only one that supports the analysis of both persistent and volatile memory. For our experiments, we follow the approach proposed in [9, 10].

1.1. Configuring and using the Android Mobile Device Emulator

In this work we use AVDs in place of a real device to carry out the experiments. Using an AVD entails two distinct steps, namely: (1) the AVD must be created first, and then (2) it must be started by the AMDE, so that the needed apps and services may be installed and used.

All the software tools required to create AVDs, as well as the AMDE, are included in the *Android SDK Tools* [5] and the *Application Binary Interface* (ABI) for ARM EABI v7a System Image software, that we assume are already installed and properly configured on the machine(s) used for the experiments. [1]

To create an AVD on the machine where the emulator runs, the *android create avd* command needs to be used as reported below (character ’$’ denotes the shell prompt):

```
$ android create avd -n chatSecureTest -t 'android-21' -b '
    default/armeabi-v7a' -c 2G
```

where:

-n chatSecureTest is the name of the AVD;

-t ’android-21’ is the target ID of the new AVD (the characteristics of this target are showed in Listing 1);

-b ’default/armeabi-v7a’ is the Application Binary Interface;

[1] The installation and configuration of these tools is outside the scope of this appendix. Various tutorials explaining how to configure and install Android SDK Tools are available on Android developers web pages [2].

-c 2G is the size of SD card (in this case, it is set to 2 GBytes).

Listing 1: Charateristics of 'android-21' target.

```
id: 5 or ``android-21''
     Name: Android 5.0.1
     Type: Platform
     API level: 21
     Revision: 2
     Skins: HVGA, QVGA, WQVGA400, WQVGA432, WSVGA,
        WVGA800 (default), WVGA854, WXGA720, WXGA800,
        WXGA800-7in
 Tag/ABIs : no ABIs.
```

Once the AVD has been created, it can be used as a real device by means of the AMDE, that provides a GUI allowing the user to interact with it after having started it by means of the following command:

```
$ emulator -avd chatSecureTest &
```

To run the experiments discussed in this work, we install ChatSecure on the running AVD by means of the following commands:

```
$ wget https://guardianproject.info/releases/chatsecure-
   latest.apk
$ adb install chatsecure-latest.apk
```

Furthermore, to extract data generated by ChatSecure from the internal memory of the device, we use the *File Explorer* tool provided by the *Android Device Monitor* [3]. Alternatively, the pull action provided by the Android Debug Bridge can also be used as described below:

```
$ adb pull <remote> <local>
```

where the $< remote >$ and $< local >$ indicate the file/folder to extract, and where to store it on the machine used for the experiments, respectively.

1.2. Configuring and using LiME for volatile memory extraction

The procedure described in the previous section allows the experimenter to extract the data stored in the persistent memory of the device. To extract the contents of volatile memory of an AVD, we resort instead to a different procedure involving LiME, that is arguably the most accurate open-source tool for memory extraction available on Linux systems [7, 6].

LiME consists in a *loadable kernel module* (LKM) that, once loaded into the running kernel, dumps the contents of the volatile memory either on an SD card placed in the device, or over a TCP connection. Therefore, to enable the usage of LiME, the Android kernel running on the AVD must provide loadable modules support.

Unfortunately, the standard AVD kernel (i.e., the default kernel provided with AVDs) does not provide such a support, so to use LiME it is necessary to first configure and compile it (as described in 1.2.1 below), and then to compile LiME as a loadable module for this kernel (as described in 1.2.2 below).

1.2.1. Compiling the Goldfish kernel

To include loadable memory support, the Android kernel (that is named *Goldfish*) must be properly configured and recompiled. To ensure that the recompiled kernel works correctly on the AVD, it is necessary to identify the kernel version running on it, so that the correct source can be used for the recompilation.

The version of the *Goldfish* kernel running on the AVD can be determined by inspecting the contents of the */proc/version* special file on the AVD, that can be done as reported below:

```
$ adb shell cat /proc/version
Linux version 3.4.67-01422-gd3ffcc7-dirty (digit@tyrion.par
    .corp.google.com) (gcc version 4.8 (GCC) ) #1 PREEMPT
    Tue Sep 16 19:34:06 CEST 2014
```

The kernel version is identified by the so-called *point of development*, that in the example above is *gd3ffcc7*.

Once this information is known, it is necessary to (a) download the kernel *config* file from the AVD (this file contains the compilation options for the running kernel), (b) download the *toolchain* [4] containing the tools required for the compilation, (c) download the source code of the correct kernel version that has been just identified, and (d) add loadable module support to the *config* file. These steps are reported below, where character '#' denotes a comment:

```
# >>>>> create the test-goldfish folder
$ mkdir -p ~/android/test-goldfish
$ cd ~/android/test-goldfish
#>>>>> get config.gz file from the emulator and unzip it
$ adb pull /proc/config.gz
$ gunzip config.gz
#>>>>> get the toolchain
$ git clone
https://android.googlesource.com/platform/prebuilts/gcc/
    linux-x86/arm/arm-eabi-4.7
#>>>>> get the kernel sources and checkout the correct
    commit
$ git clone https://android.googlesource.com/kernel/
    goldfish.git
$ cd ~/android/test-goldfish/goldfish
$ git checkout d3ffcc7
#>>>>> prepare the environment for cross-compilation
$ 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
```

The last statement of the listing above, namely *make menuconfig*, opens a configuration menu that allows one to select the *loadable module support* option from a textual menu.

Finally, the re-configured kernel and its modules can be compiled as follows:

```
$ make modules_prepare
#>>>> compile the kernel
$ make -j$CoresPlus1
#>>>> save System.map
$ cp System.map ../System.map
```

and the AVD can be rebooted with the new kernel, that now includes loadable module support, as follows:

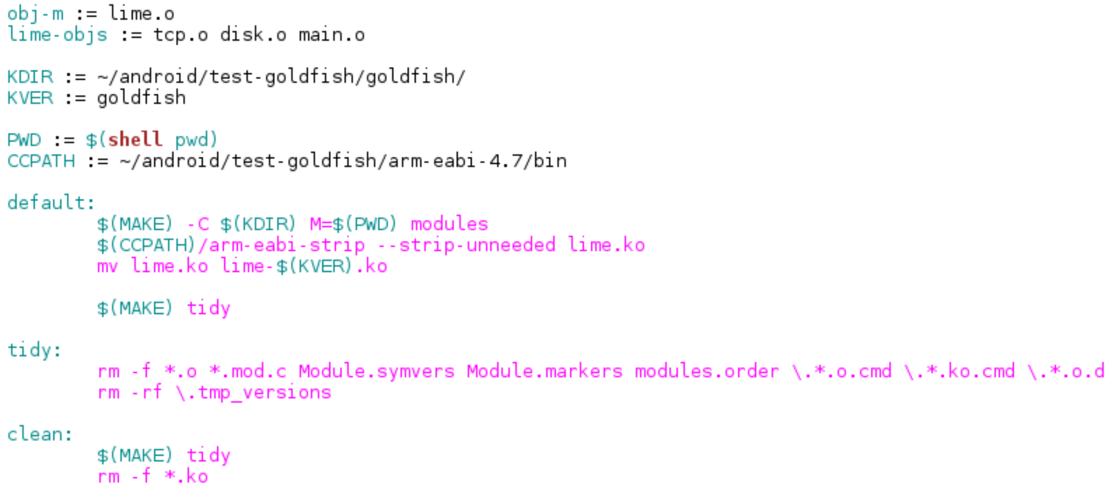
```
# >>>> start AVD with the new kernel
$ emulator -avd chatSecureTest -kernel ~/android/test-
    goldfish/goldfish/arch/arm/boot/zImage &
```

1.2.2. Compiling LiME for the Goldfish kernel and using it for memory acquisition

As mentioned before, LiME consists in a kernel module, that needs to be compiled for the kernel running on the AVD as shown below:

```
# >>>>> compile LiME loadable module
$ cd ~/android/test-goldfish/
$ git clone https://github.com/504ensicsLabs/LiME.git
$ cp Makefile.LiME.coss LiME/src/Makefile
$ cd LiME/src
$ make clean && make
$ mv lime.ko lime-goldfish.ko
```

where the *Makefile.LiME.cross* file is shown in Figure 1.



```
obj-m := lime.o
lime-objs := tcp.o disk.o main.o

KDIR := ~/android/test-goldfish/goldfish/
KVER := goldfish

PWD := $(shell pwd)
CCPATH := ~/android/test-goldfish/arm-eabi-4.7/bin

default:
        $(MAKE) -C $(KDIR) M=$(PWD) modules
        $(CCPATH)/arm-eabi-strip --strip-unneeded lime.ko
        mv lime.ko lime-$(KVER).ko

        $(MAKE) tidy

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

clean:
        $(MAKE) tidy
        rm -f *.ko
```

Figure 1: The *Makefile.LiME.cross* file.

Once the module has been compiled, it is pushed to the AVD using the ADB, and then it is loaded into the kernel by means of the *insmod* command, as shown below:

```
$ cd ~/android/test-goldfish/
$ adb push lime-goldfish.ko /sdcard/lime.ko
$ adb forward tcp:4444 tcp:4444
$ adb shell insmod /sdcard/lime.ko "format=lime path=tcp
    :4444" &
$ nc localhost 4444 > goldfish.lime
```

The parameters passed to *insmod* specify that the dump has the LiME's native format, and that the corresponding data are sent across a TCP connection identified by port 4444, that has been forwarded to the same port of the physical machine on which the emulator is running. At the end of the acquisition, the memory dump is stored in the *goldfish.lime* file on the physical machine where the emulator is running.

1.3. Configuring and using Volatility for volatile memory analysis

Volatility [8] is one of the most popular platforms for the analysis of volatile memory, and supports a wide variety of memory dump formats, processor architectures, and operating systems.

To use Volatility of a specific system (characterized by its processor architecture and operating system), it is necessary to create a *volatility profile* storing the information concerning the data structures, the algorithms, and the symbols that have to be used to correctly parse the memory dumps coming from that system.

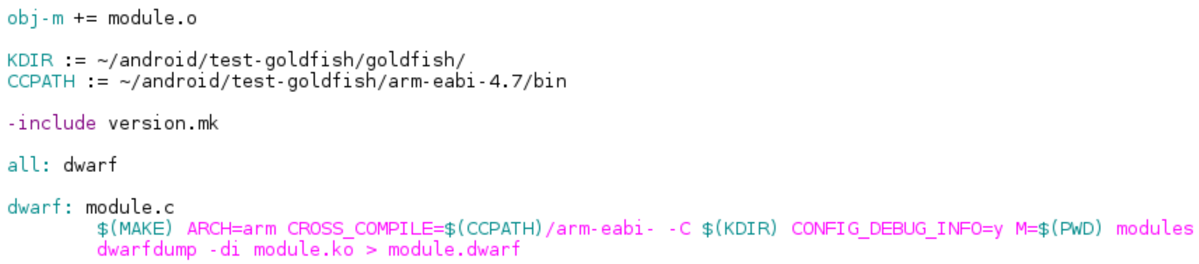
Creating a Volatility Linux profile means generating a set of *VTypes* and a *System.map* file for a particular kernel version and packing those together into one *zip* file.

VTypes can be extracted from the compiled Linux kernel file *vmlinux* if available, otherwise with the *dwarfdump* tool (a tool that parses the debugging information from *ELF* files, such as Linux kernel and Linux modules).

The *System.map* file can be instead created, for the Android system, by compiling the target kernel as discussed below.

First of all, we need to use a makefile to cross-compile Volatility for the ARM processor architecture, as the one shown in Fig. 2 that we use in our work. Then, we create the profile *Android_Goldfish_3.4.67-01413-gd3ffcc7.zip* using the commands listed below:

```
#>>>>> compile Volatility module
$ cd ~/android/test-goldfish/
$ git clone https://github.com/volatilityfoundation/
    volatility.git
$ cp Makefile.Volatility.cross volatility/tools/linux/
    Makefile
```



```
obj-m += module.o

KDIR := ~/android/test-goldfish/goldfish/
CCPATH := ~/android/test-goldfish/arm-eabi-4.7/bin

-include version.mk

all: dwarf

dwarf: module.c
	$(MAKE) ARCH=arm CROSS_COMPILE=$(CCPATH)/arm-eabi- -C $(KDIR) CONFIG_DEBUG_INFO=y M=$(PWD) modules
	dwarfdump -di module.ko > module.dwarf
```

Figure 2: The *Makefile.Volatility.cross* file.

```
$ cd volatility/tools/linux/
$ make
# >>>>> create Volatility profile
$ zip -j Android_Goldfish_3.4.67-01413-gd3ffcc7.zip module.
    dwarf ~/android/test-goldfish/goldfish/System.map
$ cp Android_Goldfish_3.4.67-01413-gd3ffcc7.zip ~/android/
    test-goldfish/goldfish/
$ cp Android_Goldfish_3.4.67-01413-gd3ffcc7.zip ~/android/
    test-goldfish/volatility/plugins/overlays/linux/
```

The last step necessary to run Volatility consists in setting two environment variables called *VOLATILITY_LOCATION* and *VOLATILITY_PROFILE*: the first one has to point to the memory dump file to analyze, while the second one has to point to Volatility profile as follows:

```
$ cd ~/android/test-goldfish/volatility/
$ export VOLATILITY_LOCATION=file://~/android/test-goldfish
    /goldfish.lime
$ export VOLATILITY_PROFILE=LinuxAndroid_Goldfish_3_4_67
    -01413-gd3ffcc7ARM
```

In our work, we use Volatility to search the memory area used by the ChatSecure process for the known passphrase as follows. First, we discovered the *Process ID* (PID) of the ChatSecure process by means of the *linux_psaux* Volatility plugin, that prints the list of active processes, as shown below:

```
$ python vol.py linux_psaux
 Volatility Foundation Volatility Framework 2.5
Pid        Uid       Gid       Arguments
1          0         0         /init
2          0         0         [kthreadd]
3          0         0         [ksoftirqd/0]
```

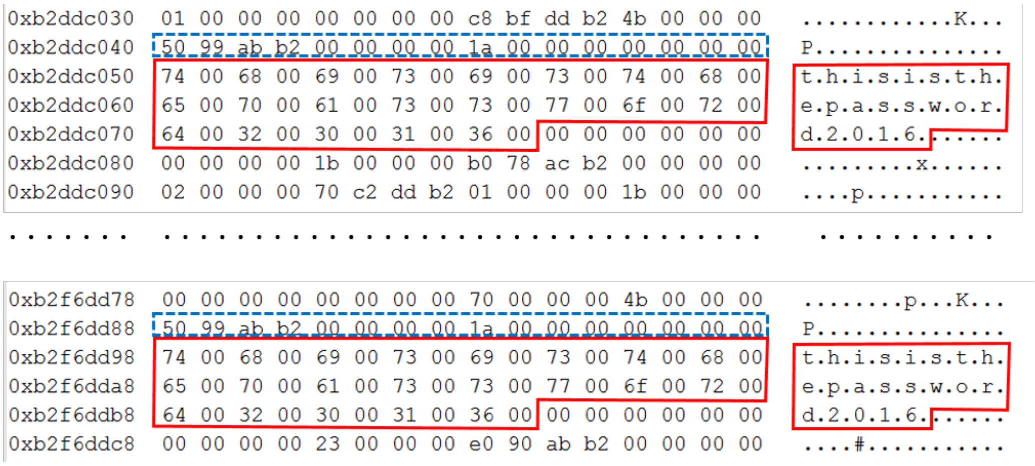
```
...
2876    10060   10060   info.guardianproject.otr.app.im
...
```

From the output of the *linux_psaux* plugin, we see that the PID of ChatSecure is 2876 (the corresponding process is named *info.guardianproject.otr.app.im*).

Then, we can search the memory area associated with the above process by means of the *yarascan* Volatility plugin [1], that is able to scan for pattern or regular expressions anywhere in process or kernel memory. In the example below, we show how *yarascan* can be used to search for the passphrase "*thisisthepassword2016*" (the one shown in Fig. 3) in the memory space of process with PID=2876:

```
$ python vol.py linux_yarascan –Y "thisisthepassword2016" –
    p 2876
```

In the example above, flag -*Y* indicates the pattern to search for, while flag -*p* restricts the scan to the memory area of the specific process.



```
0xb2ddc030  01 00 00 00 00 00 00 00 c8 bf dd b2 4b 00 00 00   ............K...
0xb2ddc040  50 99 ab b2 00 00 00 00 1a 00 00 00 00 00 00 00   P...............
0xb2ddc050  74 00 68 00 69 00 73 00 69 00 73 00 74 00 68 00   t.h.i.s.i.s.t.h.
0xb2ddc060  65 00 70 00 61 00 73 00 73 00 77 00 6f 00 72 00   e.p.a.s.s.w.o.r.
0xb2ddc070  64 00 32 00 30 00 31 00 36 00 00 00 00 00 00 00   d.2.0.1.6.......
0xb2ddc080  00 00 00 00 1b 00 00 00 b0 78 ac b2 00 00 00 00   .........x......
0xb2ddc090  02 00 00 00 70 c2 dd b2 01 00 00 00 1b 00 00 00   ....p...........

.......     ...............................                ..........

0xb2f6dd78  00 00 00 00 00 00 00 00 70 00 00 00 4b 00 00 00   ........p...K...
0xb2f6dd88  50 99 ab b2 00 00 00 00 1a 00 00 00 00 00 00 00   P...............
0xb2f6dd98  74 00 68 00 69 00 73 00 69 00 73 00 74 00 68 00   t.h.i.s.i.s.t.h.
0xb2f6dda8  65 00 70 00 61 00 73 00 73 00 77 00 6f 00 72 00   e.p.a.s.s.w.o.r.
0xb2f6ddb8  64 00 32 00 30 00 31 00 36 00 00 00 00 00 00 00   d.2.0.1.6.......
0xb2f6ddc8  00 00 00 00 23 00 00 00 e0 90 ab b2 00 00 00 00   ....#...........
```

Figure 3: Passphrase in the volatile memory of the device.

References

[1] Alvarez VM. The Yara project. 2016. Available at http://virustotal.github.io/yara/.

[2] Google . Andoid Developers. 2016. Available at https://developer.android.com.

[3] Google . Android Device Monitor. 2016. Available at https://developer.android.com/studio/profile/monitor.html.

[4] Google . Android Open Source Project. 2016. Available at https://source.android.com.

[5] Google . Android SDK Tools. 2016. Available at https://developer.android.com/studio/index.html.

[6] Sylve J. Android memory capture and applications for security and privacy. Master's thesis; University of New Orleans, USA; 2011.

[7] Sylve J, Case A, Marziale L, Richard GG. Acquisition and analysis of volatile memory from android devices. Digital Investigation 2012;8(3–4):175–84. doi:10.1016/j.diin.2011.10.003.

[8] Volatility Foundation . An advanced memory forensics framework. 2016. Available at http://volatilityfoundation.org/.

[9] Wächter P. Practical Infeasibility of Android Smartphone Live Forensics. Master's thesis; University of Friedrich-Alexander, Germany; 2015.

[10] Wächter P, Gruhn M. Practicability Study of Android Volatile Memory Forensic Research. In: Proc. of the 7th IEEE International Workshop on Information Forensics and Security (WIFS). IEEE; 2015. p. 1–6. doi:10.1109/WIFS.2015.7368601.