Post-Mortem Memory Analysis of Cold-Booted Android Devices

Christian Hilgers Holger Macht Tilo Müller Michael Spreitzenbarth
Department of Computer Science, Friedrich-Alexander-University of Erlangen-Nuremberg

Abstract

As recently shown in 2013, Android-driven smartphones and tablet PCs are vulnerable to so-called cold boot attacks. With physical access to an Android device, forensic memory dumps can be acquired with tools like FROST [1] that exploit the remanence effect of DRAM to read out what is left in memory after a short reboot. While FROST can in some configurations be deployed to break full disk encryption, encrypted user partitions are usually wiped during a cold boot attack, such that a post-mortem analysis of main memory remains the only source of digital evidence. Therefore, we provide an in-depth analysis of Android's memory structures for system and application level memory. To leverage FROST in the digital investigation process of Android cases, we provide open-source Volatility [2] plugins to support an automated analysis and extraction of selected Dalvik VM memory structures.

Keywords

Android Forensics, Cold Boot Attack, Post-mortem Analysis, Memory Analysis, Dalvik VM, Volatility Plugins

I. Introduction

Today, most of us store sensitive data like emails, photos, calendar entries and contact lists on mobile devices like smartphones and tablet PCs. This information is of growing importance for the digital investigation process of mobile devices, e.g., to recover GPS movement profiles, recent phone calls, or even photos from the scene. In practice, however, this information often remains closed to forensic examiners because modern platforms like iOS and Android provide end users with a security feature for *full disk encryption* (FDE) [3]. If secure user PINs or passwords are chosen, such that brute force attacks become virtually impossible, encrypted user partitions cannot be accessed without knowing the user credentials. If in such a case the suspect is not able or willing to cooperate, the encrypted data on disk remains closed, potentially hiding important evidence of the case.

A. Motivation

While the user data partition of mobile devices is often encrypted today, data in RAM is generally not encrypted. Although iOS and Android provide features for encrypting data on disk, data in RAM must remain unencrypted for technical reasons. For example, the performance drawback of encrypting main memory is indefensible for mobile end user products. As a consequence, it seems reasonable for forensic examiners to retrieve digital evidence from RAM rather than from disk. Considering that mobile devices like smartphones and tablets are switched off only seldom, chances are likely that important data resides in the unencrypted RAM of a device during the time of its confiscation.

However, for a long time it was unclear whether a smartphone's RAM can be analyzed with physical access to a target device only. If user access to a device would be required to analyze RAM, i.e., if the user PIN or password would be required, the ease of traditional disk forensics is usually preferred to more complex memory forensics. But in the beginning of 2013, a toolset named FROST [4], [1] was published proving the possibility of so-called *cold boot attacks* [5] against Android-driven devices. Cold boot attacks read out RAM with only physical access to a system by rebooting it with a custom bootloader and then retrieving everything that is left in main memory after boot.

While live analysis techniques for main memory are common, e.g., for the case of incident response in virtual machines, cold boot attacks allow only for a *post-mortem analysis* of main memory. Since cold boot attacks compulsorily require the target device to be rebooted during the attack, only a single memory dump can be acquired for the analysis. After the attack, the target device stops working and hence, live analyses after cold boot techniques is impossible.

B. Contributions

From a high-level perspective, the digital forensic process can be divided into five separate tasks: *data recovery*, *data analysis*, *extraction of evidence* and the *preservation* and *presentation* of evidence [6]. Our work includes two of these individual tasks as a contribution: (1) analyzing Android memory images in general and (2) extracting specific evidence from them. However, the latter is only touched by demonstrating what can be of interest from a legal perspective by providing four exemplary Volatility plugins [2], but we do not cover all information that is present in RAM. However, our Volatility plugins give user-friendly access to forensically important data in the RAM, including the phone call history, the last user input, and metadata of photos like GPS coordinates.

The data recovery step, i.e., the acquisition of Android memory images, is done separately by FROST [1]. In the original publication of FROST, the authors mentioned that a post-mortem analysis of main memory images is possible, but they focused on breaking disk encryption of Galaxy Nexus devices. Breaking disk encryption with FROST is only possible if the bootloader of a target device is already unlocked *before* the access, since otherwise the encrypted user partition gets wiped during the attack. In other words, breaking disk encryption with FROST is not possible in most cases, because bootloaders are locked by default and get unlocked manually only seldom. Therefore, building upon the results of FROST, we ignore the case of disk encryption and focus on information that can directly be retrieved from Android's main memory structures. To this end, we give an in-depth analysis of Android's Dalvik VM and selected application level structures.

C. Related Work

Although the concept of a *post-mortem analysis* of main memory has already been mentioned, e.g., regarding Windows memory structures by Vidas [7], the term *memory forensics* [8] is often mentioned in the same breath with *live forensics* [9]. That is why most related work in this field can be classified as live analysis. However, except for the fact that a post-mortem analysis has to deal with a static memory image taken after power was cut, the methods of both analysis techniques most closely correspond. For example, identifying Dalvik and application memory structures in RAM is an important challenge for both disciplines.

Live forensics have been discussed in multiple papers. Hay et al. [10] look at the topic of live forensics from a higher level, drawing a concrete distinction between static and live analysis. They outline the different possibilities for live analysis, also considering, but not solely, memory analysis. However, they do not provide an operating system specific solution, particularly not for Android. The same applies to the paper "Diagnosing Your System without Killing it First" by Adelstein [9]. Adelstein as well as Hay define live analysis as the process of taking a snapshot from a running system without shutting it down. This contradicts the method of cold boot attacks, where the system must be rebooted first.

Memory forensics for the identification of data type structures in RAM has been a research topic for several years, mostly focusing on x86 systems like Linux and Windows, some on embedded devices, but only little on Android. Having forensic memory images available for further processing, high-level information represented by C structures (in-kernel information) or Java objects (Android applications) must be analyzed. Yen et al. [11] as well as Urrea [8] focus on that area. The latter describes the underlying concepts of a concrete Linux distribution by outlining kernel structures relevant for memory management which can be used to retrieve evidence. Urrea uses `dd` to read out physical memory at runtime from `/proc/mem`., thus performing a typical live analysis.

This simple way of physical memory retrieval is considered flawed by Sylve et al. [12] as it alters the evidence in an intrusive way. Instead, Sylve et al. developed their own solution capturing memory from Linux- and Android-based systems. They also illustrate how basic kernel data can be acquired with the help of Volatility. Their paper "Acquisition and Analysis of Volatile Memory from Android Devices" targets the field of *Android live memory forensics* and is considered the most relevant work for us. However, Sylve et al. concentrate on the live acquisition of main memory but do not consult post-mortem analysis of RAM after cold boot attacks.

In the paper "Live Memory Forensics of Mobile Phones", Thing et al. [13] describe a method for analyzing Android memory images in regard to communication. They developed a tool called `memgrab` to capture all memory regions belonging to a specific process at runtime. These memory regions can then be searched for known patterns corresponding to chat messages. However, Thing et al. do not intend to solve the problem of acquiring physical main memory dumps, as we do with FROST. Finally, Leppert [14] showed another way for Android memory analysis with just looking at the heap of specific, running applications.

Whether cold boot attacks can serve as a reliable source for forensic memory images has been answered for x86 systems. In their study "An in-depth Analysis of the Cold Boot Attack", Carbone et al. [15] ask if the cold boot attack can be used for sound forensic memory acquisition and come to a largely positive answer. Similar results for x86 have been shown by Gruhn and Müller in their paper "On the Practicability of Cold Boot Attacks" [16].

D. Paper Outline

The remainder of this paper is structured as follows: In Sect. II, we give background information about FROST, Android and the Volatility framework. In Sect. III, we describe the process of acquiring forensic main memory images from Android devices with the help of toolsets like FROST and LiME. In Sect. IV, we introduce Volatility plugins we have implemented to automate the process of analyzing Dalvik memory in general. In Sect. V, we then present four Volatility plugins we have implemented to analyze specific Android applications. And in Sect. VI, we discuss possible anti-forensics techniques that owners and manufacturers can take to complicate the process of investigating memory images and to defeat our method. Finally, in Sect. VII, we summarize our work and give a brief conclusion.

II. Background

We now provide detailed information about cold boot attacks against Android devices by means of FROST (Sect. II-A). We then give necessary background information about the Android platform (Sect. II-C) as well as about the Volatility framework (Sect. II-C).

A. Cold Boot Attacks

As shown by Müller and Spreitzenbarth [4], [1], forensic examiners with physical access to an encrypted Android phone can recover all or part of its data using cold boot attacks [5]. *Cold booting* a device technically means to briefly cycle power off and on, without allowing the OS to shut down properly. Hence, cold boot attacks cannot be used for live analyses but for post-mortem analyses. After cold booting a device, main memory contents are not lost because DRAM chips of PCs and smartphones exhibit a behavior called the *remanence effect* [17], [18]. The remanence effect says that RAM contents fade away over time rather than disappearing all at once, and that they fade more slowly at lower temperatures, i.e., the colder RAM chips are, the longer the memory contents persist. That is why cold boot attacks are typically more practical when a target device has been cooled down.

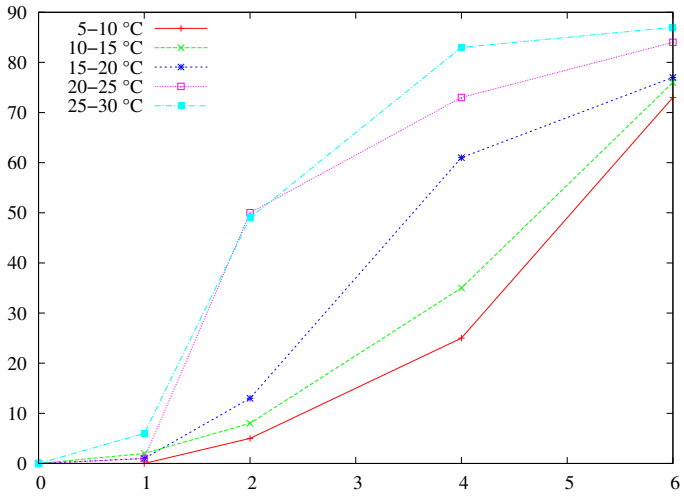

Figure 1. Bit error ratio (y-axis) in dependence of time in seconds (x-axis) and temperatures [1]

In practice, cold booting an Android device requires an examiner to replug its battery briefly, because smartphones and tablet PCs usually have no reset button. The battery must be removed for less than a second, or otherwise the bits in RAM begin to decay and parts of the data in RAM get lost. To increase this *remanence interval*, and the success rate of cold boot attacks, Müller and Spreitzenbarth suggested to put the target device into a $-15\ ^{\circ}C$ freezer for 60 minutes before replugging its battery. The operating temperature of a Galaxy Nexus phone, which is usually around $30\ ^{\circ}C$, then decreases to less than $10\ ^{\circ}C$, significantly reducing the risk of decayed bits. For example, below $10\ ^{\circ}C$ a *bit error rate* (BER) of 0 % can be achieved by replugging the battery in less than a second, whereas higher temperatures rapidly yield to unreliable results (see Figure 1).

To exploit this behavior in the digital investigation process of Android cases, the recovery tool FROST (*Forensic Recovery of Scrambled Telephones*) has been developed. If an examiner gains access to an encrypted Android device that is running but locked, it is possible to reconstruct personal information from RAM with FROST. As already shown in the original publication, this information may include personal messages, calendar entries, and photos. However, in the original publication, a focus was put on identifying the disk encryption key in main memory for breaking Android FDE. Contrary to that, we focus on an in-depth analysis of Android's memory structures. The reason is that FROST must be installed into the recovery partition of an Android device via USB before it can be booted, and this step necessarily wipes out the user partition unless the bootloader was unlocked *before* the access – which is usually not the case in practice. Hence, disk encryption can only be broken with FROST if the bootloader was already unlocked. But once the encrypted user partition is wiped, FROST can still be used to acquire forensic memory dumps because Android's main memory is *not* wiped during the installation of FROST.

B. The Android Platform

Android originates from the equally named *Android, Inc.*. The company founded Android and was later purchased by Google in 2005. In 2007, it was presented to the public by the *Open Handset Alliance*, consisting of companies like HTC, Samsung, Qualcomm, Texas Instruments, and last but not least, Google. In October 2008, the first publicly available phone running the Android platform was released. Android is a software stack consisting of a Linux kernel, a middleware layer, a Java like virtual machine called *Dalvik Virtual Machine* (DVM), some core applications like internet browsers and messaging applications, as well as third party applications making use of the available application framework.

At the bottom of the Android software stack, Android is powered by a Linux kernel. All higher layers rely on the kernel's core services such as security, memory management, process management, the network stack, and the driver model [19]. Although Android is based on a mainline Linux kernel, it is extended by a set of patches. The changes made to the standard Linux kernel include bug fixes, kernel infrastructure improvements, new hardware support, and standalone kernel enhancements for higher layer elements such as applications. However, none of the kernel changes are of immediate relevance for both memory acquisition and memory analysis as we do within this paper. There is just one feature added to the kernel and considered indirectly related to this paper – the *Low Memory Killer*, as opposed to the *Out of Memory Killer* in standard Linux kernels. As soon as a system runs out of memory, the *Out of Memory Killer* sacrifices and kills one or more processes to free up memory. In contrast to that, the *Low Memory Killer* kills processes belonging to an application before the system exhibits negative effects.

The central software component of Android is the DVM; on Android, every application runs in its own DVM instance. It is a similar concept as the Java VM, with a few, but significant differences. For example, the byte code created from the source files are compiled into *dex*-files instead of *class*-files. Those are optimized for target devices like smartphones or tablet computers. The *dex*-files are created by a tool called *dx* which compiles and optimizes multiple Java *class*-files into a single file. Together with a configuration file `AndroidManifest.xml`, and non-source-code files like images and layout descriptions, the *dex*-file is packaged into an *Android Package (apk)* file [20]. Basically, an *apk*-file is a ZIP-compatible file representing a single application.

Android's source code is released under an open source license via the *Android Open Source Project* (AOSP). This includes the kernel source and other higher level components. Being able to read the source code of Android and the DVM implementation is essential for this research project, because it enables us to gain deep knowledge about how data structures lay out in memory. For the purpose of development, debugging, testing, and system profiling, the *Android Software Development Kit* is provided. Besides the API libraries to build Java applications, it includes developer tools such as the *Android Debug Bridge* (adb), or the *Dalvik Debug Monitor Server* (DDMS). We make use of these tools at a later point in this paper.

C. The Volatility Framework

After memory acquisition, i.e., after acquiring a dump file that represents the physical memory of the target system (see Sect. III), we intend to extract data artifacts from it. Without an in-depth analysis of Android's memory structures, we would only be able to extract known file formats like JPEG (with tools like *PhotoRec*) or simple ASCII strings that are stored in a contiguous fashion (with tools like *strings*). This approach is very limited as it can be used for any disk or memory dump but does not focus on OS and application specific structures. As we intend to extract whole data objects from the Android system, we make use of the popular forensic investigation framework for volatile memory, in short *Volatility* [2].

Volatility is a "volatile memory artifact extraction utility framework" that is completely open source, released under the GNU General Public License and written in Python. At the time of writing, Volatility contains official support for Microsoft Windows, Linux and Mac OS X. Starting from version 2.3, it also contains support for the ARM architecture, and thus for Android. In this paper, we use a preliminary, but fully functional version with ARM support. Given a memory image, Volatility can extract running processes, open network sockets, memory maps for each process, and kernel modules. Volatility has a public API and comes with an extendable plugin system which makes it easy to write new code, to support new operating systems, or to add support for extracting additional artifacts.

III. Memory Dump Acquisition

We now provide details for the process of memory dump acquisition. In Sect. III-A, we focus on the LiME module which is used by cold boot attacks with FROST. In Sect. III-B, we focus on a different way of memory acquisition, particularly heap dump acquisition, that cannot be used in practical forensic cases. However, this method offers us a convenient way to learn details about Android's heap structures, and knowledge about these details can eventually be applied to entire physical memory dumps.

A. Memory Dump Acquisition with FROST and LiME

To acquire a dump of the entire physical memory we use the afore mentioned toolset FROST [1], a modified recovery image for Android phones. FROST uses an open-source kernel module called LiME [21] to dump memory. LiME can be loaded into Linux kernels, such as those running on Android devices, to dump the physical memory either to a local file or over the network. LiME is the first module to allow full memory captures from Android devices [12]. To get more forensically sound results than with standard Linux tools, the authors payed special attention to minimizing interaction between kernel and user mode applications during the acquisition process.

In order to acquire the physical memory from the operating system, the LiME module makes use of a kernel structure called *iomem_resource* (see Listing 1) to get physical memory address ranges. Each *iomem_resource* has a field named *start* which marks the start of the physical memory and a field named *end*, which marks the end. Furthermore, the specific I/O memory resources, which represent the physical memory regions, are tagged by a field *name* which must be set to "System RAM" to omit memory mapped I/O regions. Memory images can either be written to an SD card attached to the target device or can be dumped via TCP over USB to a host computer.

Listing 1. Kernel Structure *iomem_resource*

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

The LiME module offers three different image formats that can be used to save a captured memory image on disk. The format that is used is determined by a parameter passed at the command line during load time of the module. The *raw* image format simply concatenates all system RAM ranges and writes them to a file or socket handler. The second format is called *padded* and also includes non-system address ranges in the output. However, those ranges do not contain their original content but are replaced with null. This causes the output to become much larger than it actually is. The third format, called *lime*, is discussed in more detail as it is our format of choice. The *lime* format has been especially developed to be used in conjunction with Volatility. It is supposed to allow easy analysis with Volatility and a special address space has been added to deal with this format. Every memory dump based on the *lime* format has a fixed-size header, containing specific address space information for each memory range. This eliminates the need for having additional paddings just to fill up unmapped or memory mapped I/O regions. The LiME header specification is listed in Listing 2.

Listing 2. LiME Image Format Header

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

In order to dump a physical memory image in the *lime* format with FROST, the target device needs to stay connected to the host computer with USB after the installation of FROST. Afterwards, a TCP tunnel can be created with port forwarding on both the host and the target device, and the target can be induced to start dumping physical memory pages over this tunnel. With the help of the FROST GUI, these steps can be done by just clicking *"RAM Dump via USB"* in the FROST main menu. If the guidelines given in the paper about FROST [4] are followed, i.e., if the target device is cooled down to below $10\ ^{\circ}C$, and if the battery is replugged within less than a second, a forensic memory image with a BER of $0\ \%$ can be acquired.

B. Heap Dump Acquisition with DDMS and MAT

In contrast to full memory dumps that can be acquired by tools like FROST and allow to capture everything that is stored in RAM, it is sometimes helpful to acquire a heap dump of a given process. Isolating the heap of a specific process enables us to identify data type structures inside a limited memory region. Note that this approach cannot be applied in practice,

e.g., after confiscating an Android device at the scene, but that it can be used with certain Android applications to learn about their data structures.

The approach of investigating an Android application's heap area is split into the following three basic tasks:

1) Acquisition of the heap dump of a specified application. This can be done with the tool *DDMS* which is provided by the Android SDK. The resulting file has a special format called the *heap profile*.
2) Analysis of this heap profile with a memory analyzer tool such as *Eclipse MAT*.
3) Manually post-processing of the data provided by the memory analyzer.

The result of step 2 is typically a large list of strings originating from all instantiated *java.lang.String* classes found in an application's heap. This list can be post-processed to find patterns which are likely to be data of forensic interest, such as account names, email addresses and passwords. While this is a valid approach, it contains some flaws we try to circumvent with the Volatility approach. For example, acquiring a heap dump is only possible for applications prepared for debugging. When developing Android applications, there is a flag called *android:debuggable* in the application's configuration file. When set to *true*, it causes the application to open a debug port whenever the application is started on the target device. This port can be used by DDMS to acquire a heap dump from an application running on a device which is physically connected to a computer system. While the corresponding debug option is typically set to *true* during the time of development, it is supposed to be disabled when an application is released to the public. If set to *false*, DDMS has no means of acquiring the heap dump, but there is a way to modify the value of the debug option after the application has been installed, which is often sufficient to learn about its data structures. However, this step includes transferring the corresponding *apk*-file to a PC, unpacking it, modifying the Android manifest, as well as repacking and resigning the application.

IV. Memory Dump Analysis

Memory images acquired by means of FROST, as described in the last section, represent the state of a system at the time of acquisition. They contain a whole application's state and all its data, including the one from the virtual machine it is running in. This section now conveys the underlying concepts of the extraction of evidence, i.e., how to parse internal data structures contained in the kernel (see Sect. IV-A) and the DVM (see Sect. IV-B and Sect. IV-C).

A. Analyzing Android Kernel Structures

Before a memory image can be analyzed, a Volatility profile must be created which is passed to the Volatility framework as a command line parameter. Such a Volatility profile is a set of *vtype definitions* and optional symbol addresses that Volatility uses to locate sensitive information, and to parse this information [2]. Basically, a profile is a compressed archive containing two files — `System.map` and `module.dwarf`. The `System.map` file contains the symbol names and addresses of static data structures in the Linux kernel. Depending on a kernel's build configuration, it is typically created at the end of the compile process. For this purpose, the tool `nm` is executed, taking the compressed kernel image *vmlinuz* as a parameter. On all major Linux distributions, the `System.map` file is found in the `/boot` directory alongside with the actual kernel. In the case of Android, that uses a special version of the Linux kernel, it is found in the kernel source tree after kernel compilation. The `module.dwarf` file emerges by compiling a module against the target kernel and extracting the *DWARF* debugging information from it. DWARF is a standardized debugging format used by source level debuggers to establish a logical connection between an output binary and its actual source code [22]. The DWARF debugging information is generated by the compiler and is included in the output binary. In case of reading RAM dumps, it can be exploited to provide valuable information about main memory structures and method layouts; it is used by Volatility for that purpose.

In order to create a `module.dwarf` file, a utility called `dwarfdump` is required. The Volatility source tree contains the directory `tools/linux/` and running `make` in that directory compiles the module and produces the desired DWARF file. Creating the actual profile is done by simply running `zip <profile>.zip <mod_path.dwarf> <sys_path.map>`. The resulting ZIP-file needs to be copied to the Volatility source tree (`volatility/plugins/overlays/linux/`), and then the profile shows up in the profiles section of the Volatility help output, i.e., we are now able to use Volatility together with Android memory dumps.

Although the support of Android in Volatility is quite new, Linux support is not, such that a number of corresponding Linux plugins, that are also working on Android, are already available. For example, the plugin *linux_pslist*, that enumerates all running processes of a system similar to the Linux *ps* command. The plugin *linux_ifconfig* simulates the Linux `ifconfig` command, i.e., it lists the available network interfaces together with their names, IP and MAC addresses. The plugin *linux_route_cache* reads and prints the route cache that stores recently used routing entries in a hash table. The plugin *linux_proc_maps* acquires memory mappings of each individual process, i.e., it lists the virtual memory addresses and access flags of the heap, stack, and dynamically linked libraries mapped into each process. While plugins like *linux_ifconfig*

and *linux_route_cache* are useful to get direct information from the Android system, *linux_proc_maps* is of interest for analyzing DVM components and userland apps, as we do in the next sections.

B. Analyzing Dalvik Virtual Machine Structures

Andrew Case [23] already showed that a suitable entry point for extracting information out of a DVM is the object `DvmGlobals`. It is a structure available to every single DVM instance and contains global data that is shared and used by application processes. `DvmGlobals` contains some meta information for a specific DVM instance, including `loadedClasses` that is required for further processing. This field is a pointer to a hash table containing all loaded classes that are known for this instance. These classes again contain meta information, e.g., about their layout, size, and members which can later be used to access specific class instance data. To summarize, a single DVM instance belonging to one specific process contains (1) a list of all loaded system classes, and (2) specific information about a single class, such as static variables and method names.

Listing 3. Plugin dalvik_find_gdvm_offset

```
class dalvik_find_gdvm_offset(linux_common.AbstractLinuxCommand):
 def calculate(self):
   offset = 0x0
   mytask = None

   for task, vma in dalvik.get_data_section_libdvm(self._config):
     if not self._config.PID:
       if task.comm}"" != "zygote":
         continue
     mytask = task
     break

   proc_as = mytask.get_process_address_space()

   gDvm = None
   offset = vma.vm_start
   while offset < vma.vm_end:
     offset }= 1
     gDvm = obj.Object('DvmGlobals', vm = proc_as, offset = offset)
     if dalvik.isDvmGlobals(gDvm):
       yield (offset - vma.vm_start)
```

The first of our Volatility plugins we want to discuss is called *dalvik_find_gdvm_offset*. As its name might suggest, its purpose is to locate the offset of the `DvmGlobals` object within the data section where `libdvm.so` is mapped into a process' address space. As stated above, this is the base for further DVM analysis and hence, it serves as a base for other plugins. The relevant code snippet is shown in Listing 3. After initializing basic variables, the helper function `get_data_section_libdvm()` is used to iterate over all memory mappings of the given process. It solely returns the memory mappings of the data section of `libdvm.so`. If no specific process ID has been specified as a command line parameter, the first process running in a DVM is used, called *zygote*. Starting from that position, the plugin scans and tests what is likely to be a `DvmGlobals` object. If found, it passes the offset to the output function, for example as follows: `DvmGlobals offset: 0x7c78`. The found offset (here: *0x7c78*) is the offset from the start of the data section of a process and can now be passed to other plugins.

Another plugin we developed is the *dalvik_vms* plugin, which is intended to find all DVM instances and to print information contained in it. It requires the `DvmGlobals` offset to be given in the command line, as described above. The corresponding plugin code in Listing 4 saves the `gDvm` offset given on the command line. However, if no command line parameter is passed on, the *dalvik_find_gdvm_offset* plugin is used internally. The *dalvik_vms* plugin then walks the process mapping of `libdvm.so`, checking if a `DvmGlobals` object can be instantiated and, if successful, passes the task and the object to the output function which then prints the structure members of `gDvm`. Those can easily be accessed due to the available *vtypes*; an exemplary output is given in Listing 5. It lists information about three DVMs, together with their process IDs, and names belonging to them. It also contains information about the number of preloaded classes.

Our next plugin, *dalvik_loaded_classes*, is used to list the information from *dalvik_vms* together with more detailed information. The *dalvik_loaded_classes* plugin lists all preloaded classes from a specific DVM instance together with the class offset, which can later be used to list specific class information with the *dalvik_class_information* plugin (see later). In Listing 6, our plugin code is listed that uses the *linux_proc_maps* plugin to get the correct process mappings for an arbitrary

Listing 4. Plugin dalvik_vms

```
class dalvik_vms(linux_common.AbstractLinuxCommand):
 def calculate(self):
   offset = 0x0

   gDvmOffset = int(self._config.GDVM_OFFSET, 16)

   for task, vma in dalvik.get_data_section_libdvm(self._config):
     gDvm = obj.Object('DvmGlobals', offset = vma.vm_start } gDvmOffset, vm = task.
         get_process_address_space())

     # sanity check: Is this a valid DvmGlobals object?
     if not dalvik.isDvmGlobals(gDvm):
       continue
     yield task, gDvm
```

Listing 5. Example dalvik_vms Output

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

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

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

PID which has been specified on the command line. Then the *dalvik_vms* plugin is utilized to get a list of DVM instances corresponding to the given process ID. Finally, the code walks those tasks and DVMs to get the concrete list of loaded classes. For each of those classes, it constructs a `ClassObject` and passes it to the output function. An example result is given in Listing 7.

Listing 6. Plugin dalvik_loaded_classes

```
class dalvik_loaded_classes(linux_common.AbstractLinuxCommand):
  proc_maps = linux_proc_maps.linux_proc_maps(self._config).calculate()
  DVMs = dalvik_vms.dalvik_vms(self._config).calculate()

  for task, gDvm in DVMs:
    for entry in gDvm.loadedClasses.dereference().get_entries():
      clazz = obj.Object('ClassObject', offset = entry, vm = gDvm.loadedClasses.obj_vm)
      yield task, clazz
```

Listing 7. Example dalvik_loaded_classes Output

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

In addition to the Java descriptor and source file of the class, the important information needed for further analysis is the offset. This is the virtual address of the system class within its process address space and is required to list specific

information about a single class. The information can be gathered by the following plugin: *dalvik_class_information*. This plugin lists concrete information about a specific system class, such as the number of instance fields, the object size in memory, and method names. It is required to parse instance objects because it contains the byte offsets of each instance field and thus, the location in the physical memory image. If the plugin is supplied with a derived class object, it can also list instance fields of arbitrary super classes. A shortened example output is listed in Listing 8; it is the output for a `Ljava/lang/Long` class at virtual address *0x40c378b8*. It has one instance field named `value`, whose value can be found at offset 8 from the beginning of an instance object from the same kind of system class. Besides an `init()` method, it has methods (direct and virtual) from a class representing a long integer, such as `toString()`, `compare()` and `equals()`.

Listing 8. Example dalvik_class_information Output

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

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

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

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

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

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

Listing 9. Plugin dalvik_class_information

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

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

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

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

To get this information, the plugin code in Listing 9 first reads the class offset given on the command line. In the following lines, the address space for a PID given on the command line is stored into the variable `proc_as` which is used to instantiate a `ClassObject` which is then passed to the output function.

C. Analyzing Runtime Objects

Until now, our plugins have just unveiled generic DVM data and information about system classes but no data that might be relevant for real forensic cases. Relevant data are *runtime objects* of applications rather than static class information. Hence, what is required in real cases is the location, i.e., the address, of instantiated objects inside a memory dump. Together with the static information from system class files, relevant data can be extracted.

For that purpose, we developed a plugin called *dalvik_find_class_instance* to scan a memory region for a certain class instance. Due to the fact that new class objects are typically instantiated on the heap, looking for an instance object inside the DVM heap is a good starting point. The DVM heap is mapped into each process address space, and our plugin code, listed in Listing 10, locates the start and end addresses of the corresponding data section by using the helper function *dalvik_get_data_section_dalvik_heap()*. Afterwards, the plugin starts scanning at the beginning of the data section, trying to instantiate an `Object`. This `Object` contains a reference to the desired `ClassObject`. In turn, the `ClassObject`'s `clazz` pointer points to the actual system class, the one given on the command line. If the correct address is found, it is handed over to the output function until the end of the Dalvik heap data section is reached. During our evaluation of various memory images, this method for retrieving instance objects turned out to work reliably.

Listing 10. Plugin dalvik_find_class_instance

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

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

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

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

The first column in Listing 11 contains the system class for which a corresponding instance object must be found. The second column lists the class instances we are trying to locate and shows multiple rows containing different pointers for the class instance. For three reasons, we cannot stop searching when a single pointer has been found: First, not all objects in memory are valid; the reference to the object might still be intact, so that the `clazz` pointer check succeeds but other areas of the object's memory might have been overwritten. If no Java code holds a reference to an object, the garbage collector is free to handle it, including reassignment of the corresponding memory regions or just leaving it in the current state. Second, there might be multiple addresses (references) pointing to the same data object. And third, there might be a huge coincidence that the `clazz` pointer check succeeds although the corresponding memory area never contained an object of the class we were looking for. To be sure that a specific address really contains the desired instance object, the contained data needs additionally to be looked up and verified manually.

Listing 11. Example dalvik_find_class_instance Output

```
$ ./vol.py dalvik_class_information -p PID -c HEX
SystemClass             InstanceClass
----------------------- -------------
0x414e2f60                 0x414e3658
0x414e2f60                 0x4156bec8
[...]
```

V. Volatility Plugins for Selected Applications

The first step to analyze main memory structures of a certain application is to identify its process, e.g., by using the Volatility plugin *linux_pslist* for an overview of all running processes. Either the process in question can be determined by its name, or further investigations are required. For a manual analysis, the plugin *linux_memmap* can be used to get a mapping between the virtual address space of the process and the physical addresses inside the memory dump at hand. Once the PID of a process is known, more information can be gathered with the plugin *dalvik_loaded_classes*, as described previously.

To analyze a selected process, there are basically two approaches to recover sensitive data: Either *top-down*, i.e., a loaded class file is used as starting point to descend to its internal data structures by running *dalvik_class_information* and *dalvik_find_class_instance* as explained above. This approach is preferable if the class name is known or has a revealing name, while the variables inside the class are unknown. The other approach, *bottom-up*, starts from a known value of a data structure to search for the class file in RAM. This approach is sometimes more practical, especially if a Java class name is unknown, but static variables inside the class are known and can easily be traced. For example, this is often the case for static UTF16 unicode strings of an application.

A DVM string object is defined in the file *dalvik/vm/oo/Object.h*. The *instanceData* field of a string object points to an array object, which in turn is also defined in *dalvik/vm/oo/Object.h*. The array object has a field to store the length and a field to store the content of a string. To identify a class by a known string value, a pointer to that string must be traced in memory which points to the first byte of the array object. Usually the string object is stored immediately before the array object, and immediately before the string object another pointer is stored that points to the string object itself. This pointer is then an instance variable of this class we are searching for, taking us to the corresponding class object. The output of the plugin *dalvik_loaded_classes* reveals what kind of class this object is.

In the remainder of this section, we describe four selected applications that we analyzed to recover data relevant for digital evidence: the phone call history (Sect. V-A), the last user input (Sect. V-B), the user PIN or password (Sect. V-C), and the gallery app with metadata about photos (Sect. V-D). Note that we do not show any more code listings at this point but give higher level descriptions only, as our application specific plugins are too long to be listed meaningfully.

A. Phone Call History

One of our goals was to recover the list of recent incoming and outgoing phone calls from an Android memory dump. This list is loaded when the phone app is opened and therefore, to recover that list, the phone app must have been started at least once after the telephone was booted. The responsible process for the phone app, and thereby for the call history, is *com.android.contacts*. This process loads the class file *PhoneClassDetails.java* which models the data of all telephone calls in a history structure. One instance of this class is in memory *per history entry*. The data fields for each instance are typical meta information of a call: type (incoming, outgoing, or missed), duration, date and time, telephone number, contact name, and, if available, an assigned photo of the contact.

To automatically extract and display this metadata, we provide the Volatility plugin called *dalvik_app_calllog*. This plugin accepts the command line parameters *-o* for an offset to the *gDvm* object, *-p* for a process ID, and *-c* for an offset to the class *PhoneClassDetails*. If some of these parameters are known and are passed on to the plugin, the runtime of the plugin reduces significantly. Otherwise the plugin has to search for these values in RAM by itself. For example, if necessary the plugin runs other Volatility modules like *linux_pslist*, *dalvik_loaded_classes* and *dalvik_find_class_instance* to find possible instances of the class *PhoneClassDetails*, as explained above.

B. Last User Input

One requirement to be able to read out user PINs and passwords is that they have been typed in at least once after the telephone was booted, meaning that the screen must have been unlocked once. Assuming that no further inputs were given after unlocking the screen, the PIN is equal to the last user input, which can be found in an Android memory dump as a UTF16 unicode string. The unicode string of the last user input is created by the class *RichInputConnection* within the process *com.android.inputmethod.latin*, and is stored in a variable called *mCommittedTextBeforeComposingText*. This variable is like a keyboard buffer, i.e., it stores the last typed and confirmed key strokes of the on-screen keyboard.

To recover the last user input, we provide a Volatility plugin called *dalvik_app_lastInput*. Actually, this plugin does not only recover PINs but arbitrary user inputs that were given last; this might be an interesting artifact of digital evidence in many cases. Similar to Sect. V-A, the *dalvik_app_lastInput* plugin accepts three command line parameters: the *gDvm* offset, the PID, and the class file offset. If none, or only some, of these parameters are given, the plugin can determine missing values automatically as well.

C. User PIN or Password

Retrieving the phone call history or the last user input from RAM, as well as other string based data including contacts and SMS, is relatively straight forward. More Volatility plugins in the fashion of *dalvik_app_calllog* and *dalvik_app_lastInput* can be implemented by the methods we provide. Somewhat more challenging is the recovery of the user PIN or password assuming that other user inputs were given after unlocking the screen.

As we observed, the PIN is not only stored inside the keyboard buffer but is in RAM at least twice: once as the UTF16 unicode string described above, and once as an ASCII string. The ASCII version of the PIN is stored by the process *keystore*. This process is a system process and does not run within a DVM instance. By means of the Volatility plugin *linux_proc_maps*, we figured out that the password is stored inside the stack area of the *keystore* process, and we were able to limit the position of the PIN to the range of 200 KBytes. In all our experiments, the PIN was located inside this 200 KBytes area of the main memory. Furthermore, some offsets between known values and the PIN inside this area seem to be constant, whereby we were able to implement a reliable method to reconstruct the PIN. Again, we provide this method as a Volatility plugin, called *dalvik_app_password*.

D. Metadata of Photos

Last, we wanted to extract metadata from the photos in the Android gallery app. One requirement again is, that the gallery has been started at least once after the telephone was booted. Starting the gallery, the responsible process is *droid.gallery3d* which loads the class *LocalAlbum*. This class represents a photo or video album, and stores the name of the album as well as a variable named *mItemPath* which in turn points to all items in that album. A single picture is represented by the class *LocalImage* which extends the class *LocalMediaItem*. In this class, forensically relevant metadata of a picture are stored, like its name, its size, the date and the time, and even the GPS coordinates of the place it was taken. As in the previous sections, we provide an automatic Volatility plugin to retrieve this data from memory dumps, called *dalvik_app_pictures*. Note that a picture itself, if it is in RAM, can be recovered by third party tools like *PhotoRec*. But be aware that not all pictures are always present in RAM, as already mentioned by the authors of FROST [1], meaning that only the metadata of all pictures can be reconstructed for sure, but not the contents.

VI. Anti-Forensics

Anti-forensics techniques to defeat cold boot attacks have long been studied in academics, but end user products secure against cold boot attacks are still rare if available at all. One reason is that academic solutions only counteract certain parts of the cold boot problem. For example, the field of *CPU-bound encryption* systems, including tools like TRESOR [24] on x86 and ARMORED [25] on ARM, only hides the disk encryption key inside CPU registers against physical RAM access. Thus, these solutions defeat attacks on FDE, but they do not protect all data in RAM.

For a more powerful anti-forensics technique against cold boot, all data main memory must be encrypted, either in hardware or in software. While hardware encryption of RAM is basically possible, as shown by experiments with FPGA-based PCI cards [26], implementing such a solution must be left to manufacturers [27], [28]. Software-based solutions, on the other hand, as described by Henson and Taylor [29], could be adapted by end users but require expert knowledge for the installation. Apart from smartphones and ARM, *privatecore.com* offers software-based memory encryption for x86 VMs in a solution named *vCage*.

Besides memory encryption, other techniques to defeat cold boot attacks exist, e.g., *Deadbolt* [30]. Basically, Deadbolt comes with two Android installations, one offering only functionality that is most frequently used (like making a phone call), and the other offering Android's full functionality (including emails, internet browsing, and photography). Deadbolt users are advised to switch between both Android modes to preserve mobility and security at the same time. So Deadbolt cannot defeat memory acquisition by cold boot attacks, but it reduces the amount of privacy related data in RAM. According to the authors of Deadbolt, switching between both Android installations requires less time than booting up the phone, but apparently lacks in user-friendliness though.

Cold boot resistant Android apps, that can be purchased from Google's Play Store like any other app, would be more user-friendly. For example, purchasing a cold boot resistant email, calendar or photo app could protect the respective information against cold boot attacks. Technically, cold boot resistant apps must react on the screen lock event in a way that all data is erased from RAM and written back to disk – which is a secure method when FDE is activated and the bootloader is locked. We experimented with Android apps of that kind but observed that the garbage collector of Android's DVM does *not* reliably remove memory contents, although those were explicitly freed or overwritten inside the Java app several times. The only method allowing us to reliably remove data from RAM was to use native code. Native code, however, lacks many libraries for typical Android Java apps, including many GUI and network functionalities. To allow cold boot resistant apps

in future, the DVM must be extended by a mechanism for *secure deallocation* reducing the lifetime of data in memory, as already suggested by Chow et al. in 2005 [31].

Another anti-forensics technique that counteracts practical implementations like FROST, is to wipe out the disk *and* the RAM at the time a bootloader gets unlocked. Such a countermeasure does not defeat all kinds of cold boot attacks, e.g., it does not defeat variants of the cold boot attack where RAM chips are removed physically and plugged into a second analysis PC. But note that on all common Android smartphones, RAM chips are soldered onto the board and cannot be removed to be read out in another smartphone. Hence, wiping out RAM on boot time, or at least when unlocking the bootloader, indeed raises the bar for attackers.

VII. Conclusions

In traditional disk forensics, data on disk is not lost when power is cut because it is stored in a non-volatile fashion, for instance on flash memory. Contrary to that, we dealt with the analysis of volatile memory, in particular with physical memory dumps that we acquired from Android systems. For the step of memory acquisition, we focused on cold boot techniques and the FROST toolset. As opposed to previous publications about Android memory forensics, most notably by Sylve et al. [12], cold boot techniques do not permit live analysis because the target system must compulsorily be halted and rebooted. Instead, we focused on a post-mortem analysis of static memory images that we could acquire with pure physical access. However, the methods for identifying data type structures inside memory dumps are largely the same for both analyses techniques.

In this paper, we first introduced basics to take full physical memory dumps from Android, and then developed general methods to identity Dalvik and application level data structures in these memory dumps. Exemplarily, we developed four Volatility plugins based on our methods to recover personal information that is relevant in legal cases, including the history of recent phone calls, the last given user input, and metadata of photos like GPS coordinates.

Finally, we looked at anti-forensics techniques against the cold boot method. In sum, many ideas to counteract cold boot attacks exist in academia, but none of them is applied in a secure and user-friendly manner on today's mobile devices. That is, tools like FROST cannot be defeated easily by end users today, and so FROST remains an interesting option for the forensic investigation process. Tools like FROST, together with an extensive understanding of Android's memory structure, are an interesting source for digital evidence in the future.

Acknowledgment

We would like to thank *Michael Gruhn* for proofreading previous versions of our paper and for giving us helpful comments to improve this work.

References

[1] T. Müller and M. Spreitzenbarth, "FROST: Forensic Recovery Of Scrambled Telephones," in *International Conference on Applied Cryptography and Network Security (ACNS)*. Banff, Alberta, Canada: Rei Safavi-Naini and Michael Locasto, Jun. 2013.

[2] Volatilesystems. Volatility Project Homepage. last accessed: January 2014. [Online]. Available: https://code.google.com/p/volatility/

[3] Android Open Source Project (AOSP), "Notes on the Implementation of Encryption in Android 3.0," 2011, source.android.com/tech/encryption/.

[4] M. Spreitzenbarth and T. Müller, "Tools and Processes for Forensic Analyses of Smartphones and Mobile Applications," in *7th International Conference on IT Security Incident Management & IT Forensics (IMF)*. German Informatics Society (GI), Mar. 2013.

[5] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest We Remember: Cold Boot Attacks on Encryptions Keys," in *Proceedings of the 17th USENIX Security Symposium*, Princeton University. San Jose, CA: USENIX Association, Aug. 2008, pp. 45–60.

[6] B. D. Carrier, "Defining Digital Forensic Examination and Analysis Tools Using Abstraction Layers," *IJDE*, vol. 1, no. 4, 2003.

[7] T. Vidas, "Post-Mortem RAM Forensics (or Reversing Windows RAM after-the-fact)," in *CanSecWest '07*. Vancouver, Canada: Nebraska University, Apr. 2007.

[8] J. M. Urrea, "An Analysis of Linux RAM Forensics," Master's thesis, Naval Postgraduate School, Monterey California, 2006.

[9] F. Adelstein, "Live forensics: diagnosing your system without killing it first," *Commun. ACM*, vol. 49, no. 2, pp. 63–66, Feb. 2006.

[10] B. Hay, K. Nance, and M. Bishop, "Live Analysis: Progress and Challenges," *Security & Privacy, IEEE*, vol. 7, no. 2, pp. 30–37, 2009.

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

[12] J. Sylve, A. Case, L. Marziale, and G. G. Richard, "Acquisition and analysis of volatile memory from android devices," *Digital Investigation*, vol. 8, no. 3-4, pp. 175 – 184, 2012.

[13] V. L. Thing, K.-Y. Ng, and E.-C. Chang, "Live memory forensics of mobile phones," *Digital Investigation*, vol. 7, Supplement, pp. S74 – S82, 2010, the Proceedings of the Tenth Annual DFRWS Conference.

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

[15] R. Carbone, C. Bean, and M. Salois, "An in-depth analysis of the cold boot attack," DRDC Valcartier, Defence Research and Development, Canada, Tech. Rep., Jan. 2011, Technical Memorandum.

[16] M. Gruhn and T. Müller, "On the Practicability of Cold Boot Attacks," in *The Eighth International Conference on Availability, Reliability and Security (ARES '13)*. Regensburg, Germany: Friedrich-Alexander University of Erlangen-Nuremberg, Sep. 2013, pp. 390–397.

[17] P. Gutmann, "Data Remanence in Semiconductor Devices," in *Proceedings of the 10th USENIX Security Symposium*. Washington, D.C.: USENIX Association, Aug. 2001.

[18] S. Skorobogatov, "Data Remanence in Flash Memory Devices," in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, vol. 3659, University of Cambridge, Computer Laboratory. Springer, 2005, pp. 339–353.

[19] Google Inc. Android Developer Documentation. last accessed: January 2014. [Online]. Available: http://developer.android.com

[20] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, and S. Dolev, "Google Android: A State-of-the-Art Review of Security Mechanisms," *CoRR*, vol. abs/0912.5101, 2009.

[21] J. Sylve, "LiME - Linux Memory Extractor," in *ShmooCon '12*. Washingtion, D.C.: Digital Forensics Solutions, LLC, Jan. 2012.

[22] M. J. Eager and E. Consulting, "Introduction to the DWARF Debugging Format," *Group*, 2007.

[23] A. Case, "Memory Analysis of the Dalvik (Android) Virtual Machine," in *SOURCE Seattle 2011, presentation video and slides*, 2011.

[24] T. Müller, F. Freiling, and A. Dewald, "TRESOR Runs Encryption Securely Outside RAM," in *20th USENIX Security Symposium*, University of Erlangen-Nuremberg. San Francisco, California: USENIX Association, Aug. 2011, pp. 17–17.

[25] J. Götzfried and T. Müller, "ARMORED: CPU-bound Encryption for Android-driven ARM Devices," in *Proceedings of the 8th International Conference on Availability, Reliability and Security (ARES 2013)*, University of Erlangen-Nuremberg. Regensburg, Germany: IEEE Computer Society, Sep. 2013.

[26] A. Würstlein, "Design and Implementation of a Transparent Memory Encryption and Transformation System," Aug. 2012.

[27] S. Gueron, U. Savagaonkar, F. Mckeen, C. Rozas, D. Durham, J. Doweck, O. MULLA, I. Anati, Z. Greenfield, and M. Maor, "Method and Apparatus for Memory Encryption with Integrity Check and Protection against Replay Attacks," Jan. 2013, wO Patent App. PCT/US2011/042,413.

[28] I. Anati, J. Doweck, G. Gerzon, S. Gueron, and M. Maor, "A Tweakable Encrypion Mode for Memory Encryption with Protection against Replay Attacks," Mar. 2012, wO Patent App. PCT/US2011/053,170.

[29] M. Henson and S. Taylor, "Beyond Full Disk Encryption: Protection on Security-Enhanced Commodity Processors," in *International Conference on Applied Cryptography and Network Security (ACNS)*, Banff, Alberta, Canada, Jun. 2013.

[30] A. Skillen, D. Barrera, and P. van Oorschot, "Deadbolt: Locking Down Android Disk Encryption," in *3rd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM 2013)*, Berlin, Germany, Nov. 2013.

[31] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum, "Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation," in *14th USENIX Security Symposium*, Baltimore, USA, Jul. 2005.