Digital Investigation xxx (2012) 1–10

Contents lists available at SciVerse ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

Acquisition and analysis of volatile memory from android devices

Joe Sylve [a], Andrew Case [b], Lodovico Marziale [b], Golden G. Richard [a,*]

[a] Department of Computer Science, University of New Orleans, New Orleans, LA 70148, USA
[b] Digital Forensics Solutions, LLC, New Orleans, LA 70130, USA

ARTICLE INFO

Article history:
Received 27 April 2011
Received in revised form 6 September 2011
Accepted 24 October 2011
Available online xxxx

Keywords:
Android
Memory forensics
Memory analysis
Linux
Mobile device forensics

ABSTRACT

The Android operating system for mobile phones, which is still relatively new, is rapidly gaining market share, with dozens of smartphones and tablets either released or set to be released. In this paper, we present the first methodology and toolset for acquisition and deep analysis of volatile physical memory from Android devices. The paper discusses some of the challenges in performing Android memory acquisition, discusses our new kernel module for dumping memory, named *dmd*, and specifically addresses the difficulties in developing device-independent acquisition tools. Our acquisition tool supports dumping memory to either the SD on the phone or via the network. We also present analysis of kernel structures using newly developed Volatility functionality. The results of this work illustrate the potential that deep memory analysis offers to digital forensics investigators.

1. Introduction

The Android operating system now has a substantial share of the mobile market, and is expected to lead the market by the end of 2011 (Eweek, 2011). The mass adoption of Android and its projected growth make it vital that the forensics community be able to properly acquire and analyze evidence from the platform. While a few research efforts have discussed analysis of Android's file system and analysis of process memory, we are not aware of any work to date that completely acquires physical memory and subsequently performs a coherent analysis of the acquired memory for Android devices. Physical memory analysis is vital to investigations, since it contains a wealth of information that is otherwise unrecoverable. This evidence includes objects relating to both running and terminated processes, open files, network activity, memory mappings, and more. Lack of such information can make certain investigative scenarios impossible, such as when performing incident response or analyzing advanced malware that does not interact with non-volatile storage.

In this paper, we explore the technical issues associated with acquiring physical memory captures from Android-based devices as well as subsequent analysis of the data acquired. We present a methodology for acquiring complete memory captures from Android, code to analyze kernel data structures, and scripts that allow analysis of a number of userland and file system-based activities. We believe that Android will continue to require future forensics research and in order to make our results immediately usable to researchers and investigators, we have integrated support for Android memory analysis into the Volatility Memory Analysis framework (Volatility, 2011). Since Volatility is already used extensively in real investigations, to support research in memory forensics, and in a number of training courses, we hope our results will generate further interest in the Android platform.

2. Related work

The presented research encompasses a number of related work areas as it includes acquisition of memory and a number of analysis techniques.

* Corresponding author. Tel.: +1 504 280 6045.
E-mail address: golden@cs.uno.edu (G.G. Richard).

1742-2876/$ – see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2011.10.003

2.1. Linux volatile memory analysis

In the last few years, there has been a substantial amount of memory analysis research targeting Linux. The first systems presented for this purpose were the FATKit (Walters, 2006), (Burdach, 2004), and memparser (Betz, 2005). Inspired by the DFRWS 2008 challenge (DFRWS, 2008), additional efforts were made to extract forensically relevant information from memory captures (e.g., Case et al., 2008). Since then, a number of other research projects have been presented that perform deep analysis of Linux kernel data structures as well as userland information (Case, 2011; Case et al., 2010a, 2010b; Kollar, 2010). The result of these projects is the ability to gather numerous objects and data structures relevant to forensics investigations in an orderly manner. A shortcoming of these projects, however, was their inability to properly handle the vast number of Linux kernel versions and the large number of widely used Linux distributions. Due to the issues investigators face when attempting to analyze one of a large number of Linux kernel versions, a number of recent research projects have attempted to automatically build kernel structure definitions through a combination of static and dynamic analysis (Case et al., 2010a, 2010b; Cozzie et al., 2008; Lin et al., 2010; Slowinska et al., 2011).There has also been recent work by the Volatility developers to automatically generate C kernel structure representations for different Linux kernel versions using debugging information, which is similar to how Volatility handles different versions of the Windows kernel.

While the these projects were able to recover both allocated and de-allocated instances of kernel structures, many of them relied on either following references within data structures or memory scanning using ad-hoc structure signatures. The ability to accurately find data structures to which all references are removed is required in order to find completely freed objects. The problem with current generation scanners, such as those discussed previously, is that the signatures were created based on manual and informal source code review by the project developers. Illustrating serious problems with this approach, including the ease in which malware can bypass such weak signatures, were two publications that used virtual machine introspection and formal methods to construct structure signatures (Dolan-Gavitt et al., 2009; Lin et al., 2011). Using the techniques presented in these publications, forensic investigators are able to scan for instances of data structures with a degree of confidence, since malware is unable to easily bypass the signatures and false negatives and false positives will be minimal.

2.2. Linux memory acquisition

Traditionally, memory captures on Linux were acquired by accessing the */dev/mem* device, which contained a map of the first gigabyte of RAM. This allowed acquisition of 896 MB of physical memory without the need to load code into the kernel. This approach did not work for machines with more than 896 MB of RAM. Due to security concerns, the */dev/mem* device has recently been disabled on all major Linux distributions, as it allowed for reading and writing of kernel memory. In order to capture all physical memory, regardless of size, and to work around the loss of the */dev/mem* device, Ivor Kollar created *fmem* (Kollar, 2010), a loadable kernel module that creates a */dev/fmem* device supporting memory capture. *fmem* has been used in a number of incident response situations and is the defacto Linux memory acquisition tool. Another tool similar to *fmem* is the *crash* (Anderson, 2008) project by Redhat. For reasons we discuss later, the *fmem* module does not work on Android devices.

2.3. Android memory analysis

There are currently three projects that support varying levels of Android memory analysis. The first project, volatilitux (Girault, 2010), provides only limited analysis capabilities, including enumeration of running processes, memory maps, and open files, and does not provide a method to acquire memory from the phone. Our techniques provide both acquisition and analysis capabilities.

The second related work was published in DFRWS 2010 (Thing et al., 2010). This research project avoided the technical issues with capturing physical memory on Android (which we solve in this paper), by focusing on specific, running processes, and using the ptrace functionality of the kernel to dump specific memory regions of a process. The virtual memory captures are then analyzed to discover evidence. While this is a good first step, many important aspects of the Android device's memory are not analyzed, including in-kernel structures, networking information, etc. Another concern is that the approach requires memory to be extracted separately for each process of interest, which requires a number of interactions with the live system and potentially overwrites valuable evidence. We concentrate instead on physical memory acquisition and analysis, which provides a superset of the information contained in the address spaces of individual processes.

Finally, another tool that is capable of extracting process memory is *memfetch* (Zalewski, 2002). This tool dumps a running application's address space, either on demand or when faults (e.g., SIGSEGV) occur. *memfetch* is portable across a variety of Linux distributions, including Android, but cannot acquire physical memory.

3. Acquiring volatile memory

In this section we discuss memory acquisition for Android and our discussion is broken into a number of sections for readability. Section 3.1 explains how to prepare a phone for memory acquisition, Section 3.2 discusses issues with existing acquisition modules, and Section 3.3 discusses portability issues.

3.1. Preparing the phone

Preparation of the phone for memory acquisition requires a number of steps, since Android does not support a memory device that exposes physical memory and furthermore does not provide APIs to support userland memory acquisition applications. This means that

Please cite this article in press as: Sylve J, et al., Acquisition and analysis of volatile memory from android devices, Digital Investigation (2012), doi:10.1016/j.diin.2011.10.003

acquisition of physical memory requires gaining root privileges on the phone so that code can be loaded into the OS kernel to read and export a copy of physical memory. While not ideal, this procedure is commonplace when live forensics analysis is performed on commodity operating systems, virtually all of which have now removed or disabled devices that expose physical memory (e.g., */dev/mem*, *||Device||PhysicalMemory*). Unless Android adds the ability to export memory directly from userland (which is unlikely) or manufacturers include hardware that allows for such access directly through DMA (e.g., FireWire, also unlikely), loading code into the running kernel to dump memory is the only method available to access privileged memory and the memory of all running processes.

The first step in the preparation process, gaining root privileges on an Android phone, commonly referred to as "rooting", is not difficult, as a number of methods exist that allow elevation of a normal user process to root (user id 0) access. Examples of these include "Rage against the Cage" (Kramer, 2010) and a number of NULL pointer dereference exploits (Zinx, 2009). There are valid concerns about using privilege escalation exploits to obtain root privileges, and an investigator should only use rooting techniques that have been verified to work reliably on a particular phone and furthermore, verified not to have undesirable consequences, such as introduction of malicious code. A "rooting toolkit" with verified functionality is therefore a useful component of a live forensic investigator's toolset, along with proper acquisition tools. While this might seem like a radical idea, the situation is not unique to Android devices. For example, if an investigator must obtain a copy of physical memory from a live desktop machine for which no administrator privileges are available, privilege escalation provides the only option for introducing kernel code to facilitate memory dumping.

Once exploited, an Android process continues to execute as root until closed, which provides a vector for loading code into the kernel. The binary containing the exploit can be transferred to the target phone in a number of ways, but the most portable method to transfer files to and from the phone is through the *adb* application that is distributed with the Android SDK. *adb* wraps a host PC-to-phone protocol that allows for transfer of files, execution of commands, and other tasks. Once the exploit is transferred, it can then be executed in the shell to gain root. Of course the entire rooting process can be skipped on phones that were previously rooted by their owner.

3.2. Issues with existing memory acquisition modules

The initial aim of the presented research project was solely analysis of acquired memory. Upon starting the research, it was discovered that existing Linux memory acquisition modules were unusable against Android devices. The first module tested was *fmem*, which is widely used for acquisition on Intel-based machines. The basic operation of *fmem* involves creation of a character device */dev/fmem* that supports read and seek operations backed by physical memory. This allows *dd* and other similar userland applications to read memory from the running operating system. Internally *fmem* works by:

1. Obtaining the starting offset specified by the read operation.
2. Checking that the page corresponding to this offset is physical RAM and not part of a hardware device's address space.
3. Obtaining a pointer to the physical page associated with the offset.
4. Writing the contents of the acquired page to the userland output buffer.

While attempting to use *fmem*, a number of issues were discovered. First, the function used to implement step 2, *page_is_ram*, does not exist on the ARM architecture. This means that the investigator cannot simply specify the entire memory range to be copied as the module would attempt to read from memory-mapped hardware device ranges, which could cause severe instability and potentially crash the phone.

The second issue discovered was that the *dd* application bundled with common Android ROMs does not handle file offsets above 0x80000000 correctly. This is because the Android *dd* uses 32-bit signed integers for offsets and storing 0x80000000 causes a 32-bit signed integer overflow. It then uses a system call to interact with a kernel function that expects a 64-bit signed integer. This means the kernel function receives a sign-extended 64-bit integer, which will obviously produces incorrect results. In the case of 0x80000000, this transforms the address used by the kernel function into 0xFFFFFFFF80000000. This incorrect handling of integers makes *dd* unusable for memory acquisition on a number of Android devices.

Finally, during the testing phase described in Section 4, it was discovered that *fmem* only recovers 80% of the original memory of devices from which it acquires memory. We believe this high percentage of overwritten memory (20%) is due to the fact that *fmem* requires extensive interaction with userland. Particularly when used with *dd*, as is recommended by the *fmem* author, a context switch and userland-to-kernel land copying of data must occur thousands of times during the memory imaging operation.

The other kernel module for memory acquisition, *crash*, faces the same issues with *dd* as it also exposes a device driver to userland. This userland approach also creates the same issues with overwriting excessive memory due to frequent context switching.

3.3. Barriers to device-independent acquisition

One issue that affects all kernel modules for Android phones, including our memory acquisition module, is portability across a wide variety of phone models. Unfortunately, loading kernel modules is a difficult task to perform in a kernel version agnostic manner. When attempting to load a kernel module, if module verification is enabled, the kernel performs a number of sanity checks to ensure that the module was compiled for the specific version of the running kernel. If any of these checks fail, then the kernel refuses to load the module. While module verification is optional, every kernel we tested (see Table 1) enabled it and there is no reason to believe that verification

Please cite this article in press as: Sylve J, et al., Acquisition and analysis of volatile memory from android devices, Digital Investigation (2012), doi:10.1016/j.diin.2011.10.003

Table 1
Phones used as test platforms for our tools.

Model	ROM	Kernel version	Config exported
HTC EVO 4G HW Rev: 0004	OMJ_EVO_2.2_Froyo_v4.0_odexed	2.6.32.15-g59b9e50 #17	Yes
HTC EVO 4G HW Rev: 0003	Stock	2.6.32.17-gee557fd	Yes
HTC EVO 4G HW Rev: 0003	Stock	2.6.35.10-gc0a661b	Yes
Droid Eris	Kaos Froyo	2.6.29-c77FF39d	No
Droid 2	Stock	2.6.32.9-g462500f	No
Android emulator	Stock Goldfish 2.2	2.6.29	Yes

will be disabled on other Android phones. A bypass of the sanity checks is very difficult, since kernel modules are tagged with a number of pieces of information about the kernel they were compiled against. While some of this is superficial information, such as version information and strings that might easily be changed to "trick" the kernel into loading a module, the module also stores CRCs of functions and structures that it requires. Before loading, the kernel reads each symbol in the binary and attempts to match its CRC against the corresponding code in the kernel. Again, if this check fails, then the module does not load. Without the CRC information for particular kernels, the location of which is discussed shortly, successfully loading a module that does not match the required kernel version is extremely difficult, since it would require bruteforcing (on the phone) the kernel CRC values for every symbol used by the module.

To work around the issues related to version-generic kernel modules, a popular root-only Android application, No Dock, attempts to bypass many of the strict checking features (Nodock, 2011). First, the application comes with bare kernel modules compiled against a stock version of each supported kernel for ARM. At load time it first uses *uname* in order to determine the running kernel version and which bare module it should attempt to load. Next, it tries to read */dev/kmem*, a file mapping kernel memory, in order to locate the *vermagic* string. If it is able to read this file and locate the string, it then patches the on-disk module with it in order to satisfy the check. In order to bypass CRC checks, No Dock assumes that by loading a module compiled against the same base kernel that CRC checks will pass. Unfortunately, this is not always the case as functions can change between minor versions and this issue is documented on the referenced page. Therefore No Dock is able to handle a fairly large number of kernel versions, but it can still fail in a number of ways. For example, if */dev/kmem* is not present, then the loader is unable to read the correct version magic string. It will also fail if any of the CRC checks fail. Ultimately, the No Dock approach is promising to increase the number of supported phones for a kernel module, but it is not perfect.

Creating a module for every kernel version that might be deployed on an Android phone is therefore not a trivial task. In order to compile a loadable kernel module, a number of additional files are required, including the kernel source for the installed kernel. While a number of manufacturers release the kernel source for their deployed kernel in order to comply with the GPL, distributors of popular custom ROMs for rooted phones do not include the kernel source with their releases. The lack of access to kernel source also prevents simply bypassing the previously mentioned CRC checks, since the *Modules.symvers* file, which contains the CRCs of all symbols, cannot be obtained.

Module compilation also requires the kernel configuration file (*.config*) that was used when the installed kernel was compiled. Normally there are two ways to acquire this file, the first being from within the kernel sources distributed by the kernel creator and the second from */proc/config.gz* on the running kernel. Our research revealed that while the kernel on some phones provides */proc/config.gz* (see Table 1), it is unavailable on others.

Due to these issues, further research is needed to make a truly kernel version agnostic module. Support for stock kernels on Android phones is fairly straightforward, but procedures to safely bypass the kernel version checking restrictions on custom kernels would have an immense impact on module portability, both for our work and for other useful kernel modules.

4. DMD

In this section we discuss our Android memory acquisition module, named *dmd*, address memory dumping over TCP and to an Android device's SD card, and offer thoughts on the forensic soundness of our approach.

4.1. The acquisition module

In order to support acquisition of kernel memory across all Android devices, we have developed a kernel module that acquires a copy of system RAM with minimal interaction from the investigator. To work around the issues detailed in Section 3.2, our module, *dmd*, takes a different, simpler, and less invasive approach to acquiring memory. Our module works by:

1. Parsing the kernel's iomem_resource structure to learn the physical memory address ranges of system RAM.
2. Performing physical to virtual address translation for each page of memory.
3. Reading all pages in each range and writing them to either a file (typically on the device's SD card) or a TCP socket.

When loading the module, the investigator provides either a directory path to copy the dump to on the host device or a TCP port for the device to listen on. Physical address range information is handled automatically in the

Please cite this article in press as: Sylve J, et al., Acquisition and analysis of volatile memory from android devices, Digital Investigation (2012), doi:10.1016/j.diin.2011.10.003

kernel module. The memory dump is written directly from the kernel to limit the amount of interaction with userspace and in particular, to eliminate the need for userspace data copying programs such as *dd*. This saves a substantial number of system calls and other kernel activity that is necessary when using userland tools such as *dd* and *cat*, which must issue a *read* and *write* call for every block of data requested via the memory device. The module also attempts to avoid the use of kernel file system buffers and network buffers in order to minimize the contamination of volatile memory during the acquisition process.

4.2. Interacting with the developed module

To illustrate the use of the described module, we will now walk through two examples of acquiring memory from an Android device. We will first discuss the acquisition of memory over a TCP connection, followed by a discussion of acquiring a memory dump via the phone's SD card. While these processes should be identical for all Android devices, in our example we will use a rooted HTC EVO 4G, a popular Android phone.

4.2.1. Acquisition of memory over TCP

The first step of the process is to copy the kernel module to the phone's SD card using *adb*. *adb* is the Android Debug Bridge, which supports a number of interactions with an Android device tethered via USB. We then use *adb* to setup a port-forwarding tunnel from a TCP port on the device to a TCP port on the local host. The use of *adb* for network transfer eliminates the need to modify the networking configuration on the device or introduce a wireless peer—all network data is transferred via USB. For the example below, we have chosen TCP port 4444. We then obtain a root shell on the device by using *adb* and *su*. To accomplish this we run the following commands with the phone plugged into our computer and debugging enabled on the device.[1]

```
$ adb push dmd-evo.ko /sdcard/dmd.ko
$ adb forward tcp:4444 tcp:4444
$ adb shell
$ su
#
```

Memory acquisition over the TCP tunnel is then a two-part process. First, the target device must listen on a specified TCP port and then we must connect to the device from the host computer. When the socket is connected, the kernel module will automatically send the acquired RAM image to the host device. The module first sends a fixed-size header, which lists the physical memory address ranges for the device and their corresponding offsets in the image. It then sends an image of each physical address range concatenated together.

In the *adb* root shell we install our kernel module using the *insmod* command. To instruct the module to dump memory via TCP we set the *path* parameter to "tcp", followed by a colon and then the port number that *adb* is forwarding. On our host computer we connect to this port with *netcat* redirect output to a file. When the acquisition process is complete, *dmd* will terminate the TCP connection.

The following command loads the kernel module via *adb* on the target Android device:

```
# insmod dmd path=tcp:4444
```

On the host, the following command captures the memory dump via TCP port 444 to the file "evo.dump":

```
$ nc localhost 4444 > evo.dump
```

4.2.2. Acquisition of memory to the device' SD card

In some cases, such as when the investigator wants to make sure no network buffers are overwritten, disk-based acquisition may be preferred to network acquisition. To accommodate this situation, *dmd* provides the option to write memory images to the device's file system. On Android, the logical place to write is the device's SD card.

Since the SD card could potentially contain other relevant evidence to the case, the investigator may wish to image the SD card first in order to save unallocated space. Unfortunately, some Android phones, such as the HTC EVO 4G and the Droid series, place the removable SD card either under or obstructed by the phone's battery, making it impossible to remove the SD card without powering off the phone (these phones will power down if the battery is removed, even if they are plugged into a power source!). For this reason, the investigator needs to first image the SD card, and then subsequently write the memory image to it. While this process violates the typical "order of volatility" rule of thumb in forensic acquisition, namely, obtaining the most volatile information first, it is necessary to properly preserve all evidence.

Fortunately, imaging the SD card on an Android device that will be subjected to live forensic analysis (including memory dumping) does not require removal of the SD card. Tethering the device to a Linux machine, for example, and activating USB Storage exposes a */dev/sd?* device that can be imaging using traditional means (e.g., using *dd* on the Linux box). Activating USB Storage mode unmounts the SD card on the Android device, so a forensically valid image can be obtained.

With USB Storage mode deactivated we copy the *dmd* kernel module to the device using the same steps described in the last section. When installing the module using *insmod*, we set the *path* parameter to */sdcard* to specify the directory in which the dump should be placed:

```
$ insmod dmd path=/sdcard
```

Once the acquisition process is complete, we can power down the phone, remove the SD card from the phone, and transfer the memory dump to the examination machine. If the phone cannot be powered down, *adb* can also be used to transfer the memory dump to the investigator's machine.

[1] Enabling debugging involves a simple change in the phone's settings.

Please cite this article in press as: Sylve J, et al., Acquisition and analysis of volatile memory from android devices, Digital Investigation (2012), doi:10.1016/j.diin.2011.10.003

4.3. Testing

The developed kernel module was tested against a number of Android phones. Table 1 lists these phones with the model, ROM, and kernel version. Other Android phones are similar, with minor differences in kernel versions.

Since it would be infeasible to test every Android model on the market and the goal of our effort is to provide memory acquisition capabilities for all Android devices, the module was designed to work as simply as possible. The only functionality that the final version of the module relies on is the ability to translate virtual to physical addresses, the ability to write to files from the kernel, and the ability to communicate over TCP. If any of those facilities were broken, the operating system would not operate correctly as these are basic operations necessary for proper operation of the phone. Because we use only basic operating systems services in the *dmd* module, we are confident that the module will work on all Android devices as well as other architectures that support Linux.

Testing was performed using manual analysis of the acquired memory capture as well as testing captures with our developed Volatility functionality, which is discussed in Section 5. All phones tested successfully allowed for acquisition of memory with no observed side effects to continued operation of the device.

4.4. Forensic soundness of acquisition approach

For the developed acquisition approach to be of use to the forensic community, it must meet the basic standards of forensic soundness. Adherence to these guidelines determines if evidence will be admissible in court and usable in other legal settings. While live forensics investigation on any computer inevitably disturbs some volatile data, just as a traditional forensics investigation of a murder scene inevitably disturbs some characteristics of the crime scene, careful steps can be made to minimize the impact. We believe our approach meets basic forensic soundness standards for a number of reasons. First, we attempt to minimize the impact on the target device when transferring data to and from it. Second, only a USB connection with the phone is needed for interaction. Once connected, only a single binary (the kernel module) needs to be transferred and executed to perform the acquisition. Third, loading of the module requires a minimal footprint, as the *dmd* module is very small ($\sim$70 KB) and requires very few kernel functions to acquire memory. As explained previously, minimal interaction with userland is needed beyond loading the module, since all reading and writing of data to files or via the network is handled within the kernel. This saves hundreds of system calls and other function invocations that would otherwise need to be performed.

To quantitatively test the soundness of our module we turned to virtualization. The Android SDK ships with a qemu-based emulator that runs the full Android stack all the way down to the kernel. By launching the emulator with the flags *–qemu –monitor stdio* we are presented with a command line interface that allows us to run commands to interact with the emulator. The *pmemsave* command pauses the execution of the guest operating system running in the emulator, saves a dump of physical memory of the guest operating system, and then continues execution of the guest operating system. This essentially allows us to capture a physical memory snapshot of a virtual Android device. We then use this snapshot to establish "ground truth" in our testing.

For our tests we repeatedly used *pmemsave* to take snapshots of memory on the virtual Android device. When the snapshot was finished we immediately started a capture using *dmd*. We then compared the two images for identical physical memory pages. Our average results for 10 runs of our tests are provided in Table 2.

We were also interested in comparing our results to tools traditionally used in Linux memory acquisition, namely *fmem* and *dd*. However, as we discussed in Section 3.2, *fmem* does not work properly on Android devices. We modified *fmem* to work around the issues we described in step 2 of the *fmem* acquisition process. Our modifications were minimal and only handled how *fmem* determines if an address points to physical RAM. These modifications should not affect the soundness of the capture. Since the Android emulator maps physical RAM starting at address 0, the issues we described with *dd* do not play a factor in acquiring memory from virtual devices (but remain problematic for real devices). We ran the same tests against the modified *fmem* as we did with *dmd*. The results are also recorded in Table 2.

512 MB RAM images collected using *dmd* were consistently over 99% identical to the *pmemsave* snapshots. Since the copying of the image takes time, which allows other running processes to perturb memory during the capture, we feel that this is a very reasonable result. When compared to the modified *fmem* implementation *dmd* shows on average significantly better results: about 99% of pages are correctly captured versus about 80%. We believe this supports our design decision to minimize interactions with userland programs and eliminate the traditional method of exposing a new memory device via a kernel module and then using a userland program such as dd to capture memory contents through this device. Based on our design goals and the results of our testing, we believe that the developed approach meets all the guidelines of a forensically sound process.

5. Analyzing a memory capture

Once memory is successfully acquired from the phone, detailed analysis is needed in order to extract information in a repeatable and useful manner. In this section we discuss code and methods we have developed that fully

Table 2
Average results from 10 runs of our testing procedure.

Method	Total number of pages	Number of identical pages	Percentage of identical pages
dmd (TCP)	131,072	130,365	99.46%
dmd (SD Card)	131,072	129,953	99.15%
fmem (SD Card)	131,072	105,080	80.17%

Please cite this article in press as: Sylve J, et al., Acquisition and analysis of volatile memory from android devices, Digital Investigation (2012), doi:10.1016/j.diin.2011.10.003

reconstruct forensically relevant kernel and userland state through memory analysis on Android devices.

5.1. ARM addressing

While a number of tools exist that are capable of analyzing Linux memory dumps, all of them have been focused on the Intel architecture due to its popularity. With the rise of ARM as the leader in the mobile realm, understanding its architecture will become increasingly necessary for forensics investigators. The main functionality necessary to support ARM from a memory forensics perspective is the ability to translate between virtual addresses and physical addresses offline. This is necessary as memory captures contains data exactly as it is physically laid out in RAM, while symbols from debugging information and pointers within data structures will correspond to virtual addresses. Paging in ARM takes the form of a two-level paging structure that can support a maximum of 4 GB of virtual memory in all currently released architectures.

5.2. Adding volatility support

In implementing our address translation capabilities, we targeted a well-known project, the Volatility Memory Analysis project. To generically support multiple hardware architectures and memory acquisition file formats, Volatility supports the notion of "address spaces", which allow for automatic handling of virtual address to physical offset translations. To add ARM support to Volatility, we implemented a Volatility "address space" for ARM, and verified its accuracy against our testbed of devices as well as the implementation in *crash*. Similar to other address space implementations, besides just address translation, we also support testing of particular page table bits as well as filling in the address space with zeroes when requested pages are not mapped. To obtain the initial page table directory value, we rely on the analyzed phone's *System.map* file to obtain the address of the *swapper* process (pid 0), and then use that information to obtain the initial memory management structure (*swapper->mm)*. This *mm* structure contains a *pgd* member that reveals the virtual address of the initial page directory (PGD). This paging information can then be used to perform initial analysis and to find other valid PGDs.

An interesting challenge in the development of the address space is that, unlike Intel, not all Android devices start RAM mappings from physical address zero. This breaks the assumption that all known memory forensics tools have, which is that offset X of a memory capture file corresponds to physical offset X in memory. To work around this issue without major changes to Volatility, we made the developed address space aware of the necessary physical offsets on a per-phone basis, with this information provided automatically by the *dmd* module in each memory dump. The module determines the necessary offsets for acquiring all physical RAM by reading the kernel's *iomem_resource* structure. This information is also made available in userland through the proc file system by reading */proc/iomem*, which we depict for a typical phone model in Fig. 1. With the address space aware of the physical shifts that are necessary, we can properly translate virtual addresses to physical memory offsets for all ARM-based devices.

5.3. Using volatility plugins

Once the ARM address space translation was successfully implemented, we were able to leverage existing Linux plugins within the Volatility framework to provide

```
# cat /proc/iomem
02b00000-02effffff : msm_hdmi.0
03700000-039fffff : kgsl_phys_memory
03700000-039fffff : kgsl
03a00000-03a3ffff : ram_console
03b00000-03dfffff : msm_panel.0
20000000-2e7fffff : System RAM
20028000-20428fff : Kernel text
2044a000-2058ca13 : Kernel data
30000000-3bffffff : System RAM
a0000000-a001ffff : kgsl_reg_memory
a0000000-a001ffff : kgsl
a0200000-a0200fff : msm_serial_hs_bcm.0
a0300000-a0300fff : msm_sdcc.1
a0400000-a0400fff : msm_sdcc.2
a0500000-a0500fff : msm_sdcc.3
a0800000-a0801000 : msm_hsusb
a1200000-a1200fff : spi_base
a9900000-a9900fff : msm_i2c.0
a9900000-a9900fff : msm_i2c
aa200000-aa2effff : mdp
aa600000-aa600fff : msm_mddi.0
```

Fig. 1. Contents of */proc/iomem* on an HTC EVO 4G, Hardware Revision 4.

Please cite this article in press as: Sylve J, et al., Acquisition and analysis of volatile memory from android devices, Digital Investigation (2012), doi:10.1016/j.diin.2011.10.003

complete analysis capabilities for Android. Fig. 2a–c shows output for a number of Volatility plugins run against an HTC EVO memory dump captured by *dmd*:

Volatility also supports gathering of common memory analysis information, such as memory maps, open files, networking connections, and more. Beyond the existing Volatility Linux functionality, we also developed a number of plugins that will be useful to investigators. We plan on releasing these plugins upon publication of this paper. The first plugin developed was the *linux_iomem* plugin that mimics the contents of the */proc/iomem* file. This plugin works by first obtaining the address of the *iomem_resource*

a

```
python volatility.py -f [memory capture path] --profile=android [plugin options] [plugin name]
```

b

```
# python volatility.py -f /mnt/data/volimgs/androidmem --profile=android linux_mount

Volatile Systems Volatility Framework 1.4_rc1
/dev/block/mtdblock4  /system     yaffs2      ro,relatime
sysfs                 /sys        sysfs       rw,relatime
devpts                /dev/pts    devpts      rw,relatime
/dev/block/dm-1       /mnt/asec/com.rovio.angrybirds-1
                                  Vfat        ro,relatime,nosuid,nodev,noexec
proc                  /proc       proc        rw,relatime
none                  /dev/cpuctl cgroup      rw,relatime
tmpfs                 /mnt/sdcard/.android_secure
                                  tmpfs       ro,relatime
tmpfs                 /dev        tmpfs       rw,relatime
/dev/block/mtdblock6  /data       yaffs2      rw,relatime,nosuid,nodev
tmpfs                 /app-cache  tmpfs       rw,relatime
/dev/block/vold/179:1 /mnt/sdcard vfat        rw,relatime,nosuid,nodev,noexec
none                  /acct       cgroup      rw,relatime
tmpfs                 /mnt/asec   tmpfs       rw,relatime
/dev/block/vold/179:1 /mnt/secure/asec/.android_secure
                                  Vfat        rw,relatime,nosuid,nodev,noexec
/dev/block/mtdblock5  /cache      yaffs2      rw,relatime,nosuid,nodev
/dev/block/dm-0       /mnt/asec/com.com2us.sliceit-1
                                  Vfat        ro,relatime,nosuid,nodev,noexec
```

c

```
# python volatility.py -f /mnt/data/volimgs/androidmem --profile=android
                       linux_task_list_psaux -p 1

python volatility.py --profile=android -f /mnt/data/volimgs/android-full
linux_task_list_psaux
Volatile Systems Volatility Framework 1.4_rc1
Arguments                                                          Pid
init                                                               1
[kthreadd]                                                         2
[ksoftirqd/0]                                                      3
[watchdog/0]                                                       4
```

Fig. 2. a. Using Volatility against an Android memory capture. b. Listing the Android device's mount points. c. Partial process listing (truncated because the complete listing is quite long).

Please cite this article in press as: Sylve J, et al., Acquisition and analysis of volatile memory from android devices, Digital Investigation (2012), doi:10.1016/j.diin.2011.10.003

structure, and then walking all of the children of each parent resource. For the information gathered by this plugin please refer to Fig. 1.

The second plugin developed, *linux_acquire_proc_maps*, is able to selectively acquire memory mappings within a specified userland process and write the corresponding pages to output file(s). The functionality contained in this plugin is a superset of the work presented in (Thing et al., 2010) as it is able to 1) acquire arbitrary memory regions from a specified process without requiring access to the running process, and 2) handle attempted acquisition of unmapped pages. This plugin works by taking the process ID of the target process and the memory range of data to acquire. It then walks the active process list and locates the process of interest. After finding the process, it creates a new address space with the PGD of the target process (*task->mm->pgd).* Once this is complete, the plugin can convert virtual addresses within the process to physical offsets. To obtain the targeted memory regions, it breaks the addresses requested into page-size aligned offsets and then reads from the requested addresses in page-sized chunks. All data read is then written to the output file(s) specified by the user.

5.4. Testing

After acquiring a memory capture from the phones listed in Table 1, the captures were analyzed by the developed Volatility code. This analysis code was tested in two phases. First, the developed address space, the code responsible for translating virtual to physical addresses, was tested using addresses with a static translation. This is possible because the Linux kernel identity maps the kernel and its data starting at virtual address 0xc0000000 on 32-bit systems. Knowing this, our test suite consisted of using the address space's virtual to physical function, *vtop*, and ensuring that the proper physical offset was returned. For example, the first check made is that a call to *vtop* with address 0xc0004000 returns a physical offset of 0x4000. Similar calls are made for a number of statically mapped virtual addresses.

Once the address space was capable of properly translating virtual addresses, we then tested the existing Volatility Linux plugins as well as our own. Existing plugins such as process listing, open file enumeration, and *netstat* were tested to ensure that core parts of the kernel could be properly analyzed. We then developed the three previously described plugins. Each of these plugins was tested against a known good set of data. For *linux_iomem*, this included comparing results with the contents of */proc/iomem*. For *linux_acquire_proc_maps*, testing was done by first obtaining specific process address mappings from the phone using *memfetch* (Zalewski, 2002), and then acquiring the same address range with the Volatility plugin. These captures were then compared to ensure they contained the same data.

6. Conclusions and future work

In this paper, we have presented methods that obtain complete captures of volatile memory from Android devices along with subsequent analysis of that data in both userland and the kernel. To our knowledge, this is the first published work on accurate physical memory acquisition and deep memory analysis of the Android kernel's structures. The developed kernel analysis support allows the popular Volatility framework to be used when analyzing data, via our implementation of ARM-specific support.

There are a number of areas of future work spawned by the research we've described. The first is performing Android-specific memory analysis against popular applications as well as the Dalvik virtual machine itself. Since Dalvik controls all userspace applications, generic analysis of its runtime virtual machine could support automated analysis of all Android applications. This includes the Phone, Contacts, MMS, and every application installable through the Android Market.

The second area of future work, which is hampered the issues discussed in Section 3, is developing a generic kernel module that can be loaded into any Android kernel without additional cross-compilation steps. Development of this module presents no specific challenge with respect to functionality, since no kernel version-specific facilities are necessary to acquire and export memory, but bypassing the kernel's CRC checks (essentially, to indicate "trust us—loading this module is OK") without the corresponding *Modules.symvers* appears to be very difficult. Despite these difficulties, we are continuing to search for a solution.

References

Anderson D. Crash. Available, http://people.redhat.com/anderson/crash_whitepaper/; 2008.

Androidnothize NoDock – testing and compatibility. Available, http://sites.google.com/site/androidnothize/no-dock/testing-comp; 2011.

Betz C. MemParser, http://sourceforge.net/projects/memparser/; 2005.

Burdach M. Idetect, http://forensic.seccure.net/tools/idetect.tar.gz; 2004.

Case, A. "De-anonymizing live CDs through physical memory analysis," presented at the Blackhat DC Security Conference, Washington D.C., 2011.

Case A, et al. FACE: automated digital evidence discovery and correlation. Digital Investigation 2008;5:S65–75.

Case A, et al. Treasure and tragedy in kmem_cache mining for live forensics investigation. Digital Investigation 2010a;7:S41–7.

Case A, et al. Dynamic recreation of kernel data structures for live forensics. Digital Investigation 2010b;7:S32–40.

Cozzie, A., et al. "Digging for data structures." Proceeding of 8th symposium on operating system design and implementation (OSDI'08), 2008.

DFRWS. Forensics challenge, www.dfrws.org/2008/challenge/index.shtml; 2008.

Dolan-Gavitt, B., et al. "Robust signatures for kernel data structures," ACM conference on computer and communications security, 2009.

EWeek. Android ships 33M smartphones to lead world: canalys. Available, http://www.eweek.com/c/a/Mobile-and-Wireless/Android-Ships-33M-Smartphones-to-Lead-World-Canalys-162803/; 2011.

Girault E. Volatilitux; 2010.

Kollar I. Forensic RAM dump image analyser. Prague: MASTERS, Department of Software Engineering, Charles University; 2010a.

Kollar I. fmem. Available: http://hysteria.sk/~niekt0/foriana/fmem_current.tgz; 2010.

Kramer S. Rage against the cage. Available, http://c-skills.blogspot.com/2010/08/droid2.html; 2010.

Lin, Z., et al. "Automatic reverse engineering of data structures from binary execution," 17th annual network and distributed system security symposium (NDSS), 2010.

Lin, Z., et al. "SigGraph: brute force scanning of kernel data structure instances using graph-based signatures," network and distributed systems security symposium (NDSS), 2011.

Please cite this article in press as: Sylve J, et al., Acquisition and analysis of volatile memory from android devices, Digital Investigation (2012), doi:10.1016/j.diin.2011.10.003

Slowinska, A., et al. "Howard: a dynamic excavator for reverse engineering data structures," 18th annual network & distributed system security symposium (NDSS), 2011.

Thing VLL, et al. Live memory forensics of mobile phones. DFRWS; 2010.

Volatility. https://www.volatilesystems.com/default/volatility; 2011.

Walters A. FATKit: Detecting malicious library injection and upping the "anti". Technical report. 4TF Research Laboratories; 2006.

Zalewski Michal. memfetch, http://lcamtuf.coredump.cx/soft/memfetch.tgz; 2002.

Zinx. Linux Kernel 2.x sock_sendpage() local root exploit (Android Edition). Available, http://www.exploit-db.com/exploits/9477/; 2009.

Please cite this article in press as: Sylve J, et al., Acquisition and analysis of volatile memory from android devices, Digital Investigation (2012), doi:10.1016/j.diin.2011.10.003