Huber *et al. EURASIP Journal on Information Security* (2016) 2016:17
DOI 10.1186/s13635-016-0041-4

EURASIP Journal on Information Security

RESEARCH Open Access

A flexible framework for mobile device forensics based on cold boot attacks

Manuel Huber[1*], Benjamin Taubmann[2], Sascha Wessel[1], Hans P. Reiser[2] and Georg Sigl[3]

Abstract
Mobile devices, like tablets and smartphones, are common place in everyday life. Thus, the degree of security these devices can provide against digital forensics is of particular interest. A common method to access arbitrary data in main memory is the cold boot attack. The cold boot attack exploits the remanence effect that causes data in DRAM modules not to lose the content immediately in case of a power cut-off. This makes it possible to restart a device and extract the data in main memory.
In this paper, we present a novel framework for cold boot-based data acquisition with a minimal bare metal application on a mobile device. In contrast to other cold boot approaches, our forensics tool overwrites only a minimal amount of data in main memory. This tool requires no more than three kilobytes of constant data in the kernel code section. We hence sustain all of the data relevant for the analysis of the previously running system. This makes it possible to analyze the memory with data acquisition tools. For this purpose, we extend the memory forensics tool Volatility in order to request parts of the main memory dynamically from our bare metal application. We show the feasibility of our approach on the Samsung Galaxy S4 and Nexus 5 mobile devices along with an extensive evaluation. First, we compare our framework to a traditional memory dump-based analysis. In the next step, we show the potential of our framework by acquiring sensitive user data.

Keywords: Cold boot attack, Digital forensics, Data acquisition, Mobile device security, Android, Smartphones, Virtual machine introspection

1 Introduction

Our today's mobile devices store sensitive data about their owners in both volatile and non-volatile memory. These include contact details, user credentials, personal and business emails, pictures, location history, or perhaps even the user's health states. That is why mobile devices are of special interest for forensic investigations. Modern mobile device operating systems, like Android and iOS, increase the confidentiality of data by encrypting the file system. The utilized Full Disk Encryption (FDE) key is therefore stored in main memory at runtime, leaving it prone to extraction with a cold boot attack [1]. Moreover, an attacker can retrieve sensitive user data exposed in RAM, including passwords, key material, pictures, and further confidential information [2, 3].

Cold boot attacks exploit the remanence effect of Dynamic Random Access Memory (DRAM), which causes data to fade slowly in memory [4, 5]. Thus, it is possible to reboot or to reset a device and to acquire the data in main memory. A cold boot attack can be initiated by either shortly removing the battery from the device or by triggering a hardware reset functionality. A forensic tool deployed on the target device conducts the data acquisition. The amount of unimpaired bytes in memory after the restart reflects the quality of an attack. This primarily depends on time, temperature, and on the utilized forensic tool. The longer the timespan between the power cut-off and restarting the device, the more bits degrade in RAM. Cooling down the physical memory mitigates the amount of data loss by the remanence effect. Additionally, not allowing the forensics tool to overwrite important content during the acquisition process is essential. An often neglected factor in the data acquisition process is the overwriting of kernel memory when booting forensics tools on the device, like in the Forensic Recovery Of Scrambled Telephones (FROST) framework [1]. In particular, this

*Correspondence: manuel.huber@aisec.fraunhofer.de
[1] Fraunhofer Research Institute AISEC, Munich, Germany
Full list of author information is available at the end of the article

includes the system state, for example, the list of running processes and their mapping in physical memory.

In this paper, we propose a novel framework that requires overwriting only a minimal amount of bytes in RAM and does not initialize device memory. We sustain all the data relevant for the analysis of the previously running system by overwriting no more than 3 KB of constant data in the kernel code section. Therefore, we obtain a genuine memory dump, which is easier to analyze. This is because the kernel structures required to interpret the memory dump are unaltered. The RAM still contains the previous system state, such as the kernel structures and page tables of the Memory Management Unit (MMU). An example are FDE keys we recognize based on their surrounding kernel structures. Preserving these structures, we can efficiently analyze the contents of the RAM with memory forensics tools. Furthermore, we can inspect the contents of device memory to investigate the security provided by hardware features like the ARM TrustZone [6]. Our framework does not require root privileges on the running device. Instead of utilizing a full-fledged Linux kernel, like in the FROST framework, we boot the mobile device with a minimalistic Bare Metal Application (BMA). The BMA provides a communication interface to a host system via the Universal Asynchronous Receiver Transmitter (UART) interface. The communication interface makes it possible to request memory dynamically. Data acquisition tools utilizing this interface can be leveraged for the forensic analysis on the host system [7, 8].

Our contributions are the following:

- The concept of a flexible framework for mobile device forensics based on cold boot attacks.
- The development of a minimal, easily portable BMA. The BMA provides a communication interface using the serial port for dynamically requesting memory from the device.
- The extension of the memory forensics tool Volatility with an implementation of the communication interface for data acquisition.
- A practical demonstration of the feasibility of our approach. We therefore implement a prototype for the Samsung Galaxy S4 mobile device and port the framework to the Nexus 5 device.
- An extensive evaluation of the proposed framework. We compare the cold boot-based analysis with traditional memory dump analysis. We also show the potential of our framework by acquiring sensitive data in a concrete-use case.

The remainder of this paper is organized as follows. In Section 2, we present related work in the fields of forensics and cold boot attacks. In Section 3, we provide background information about the interpretation problem of raw data obtained from memory dumps. We elaborate the concepts and architecture of our framework in Section 4. In Section 5, we describe the implementation of our framework. We give an explanation on the device-specific realization for the Samsung Galaxy S4 device and describe the framework's portability in Section 6. We then evaluate our forensic architecture in Section 7 and conclude in Section 8, summarizing our work.

2 Related work

In this section, we present related work that addresses the problem of data acquisition from main memory, especially in the context of mobile devices. We also provide a brief overview regarding forensics on Android devices and on countermeasures to cold boot attacks.

A preliminary approach to cold boot attacks for acquiring memory of a previously running system was to force a reboot where memory is fully preserved [9]. The preserved memory is then leveraged to circumvent OS authentication mechanisms in order to even recover the state of the previously running system. The first cold boot attack was published by Halderman et al. in [10], where they show that it is possible to extract data from main memory on the x86 architecture after a short interruption of the power supply. This also works when a DRAM module is moved to another host computer. They show that the rate of the degradation of volatile memory can be drastically reduced by cooling down the RAM module. For data acquisition, the tool *bios memimage* boots directly from a USB flash drive or via Preboot eXecution Environment (PXE) [11]. The tool transmits the content of the RAM via network to another investigation host or stores it on a USB flash drive. Additionally, it provides features to find and fix corrupted RSA and AES keys in a memory dump.

Based on the approach for desktop computers, the cold boot attack found its adoption on the ARM architecture. With FROST [1], Müller et al. show that cold boot attacks are feasible on Android phones. On those devices, it is not possible to boot external sources. In addition, the RAM module is non-removable because it is integrated into the System on a Chip (SoC). Their approach is to force a restart of a running device by interrupting the connection to the battery. Afterwards, they boot the already installed FROST framework from the recovery partition by pressing the corresponding buttons on the device to trigger the recovery mode. The FROST framework loads an entire Linux kernel and features a kernel module that searches for AES keys in main memory. The FROST boot image is installed on the recovery partition. This circumvents the restrictions in booting external sources. Therefore, the bootloader must get unlocked, which usually triggers a routine that formats the user data partition.

A drawback of the FROST framework is that it overwrites the heap of the previously running kernel, because the framework boots a full-fledged Linux kernel. This includes information like structures of the MMU, the list of running processes and the memory mappings of processes to physical locations. Additionally, the kernel likely reinitializes Input/Output (IO) devices, which resets the corresponding device memory.

Cold boot attacks do not permit live forensics, as the system halts for a short moment and then reboots the device. It is nonetheless possible to access the device memory directly on a running device. The two most obvious ways are to either read directly from `/dev/mem` or to use tools like the Linux Memory Extractor (LiME) kernel module [12]. One problem is that both approaches require root permissions [13]. It is only possible to bypass that problem by exploiting security flaws in processes that have root access. The other problem is when using kernel modules, the running kernel has to be capable of loading custom kernel modules.

Another option to access main memory is the use of devices that offer Direct Memory Access (DMA). For the x86 architecture, this was shown for the PCIe [14], Firewire [15], and Thunderbolt [16] interface. Those interfaces are in general not available on mobile devices. However, mobile devices often have a Joint Test Action Group (JTAG) interface for debugging purposes. This provides full access to main memory at runtime. In [17], the author uses the JTAG interface to exploit the baseband of a smartphone. The RIFF Box [18] is a device that makes it easy to retrieve a memory dump or even to read or write the memory on the internal flash drive via the JTAG interface.

Digital forensics goes back to approaches on early computers decades ago [7] and nowadays finds its adoption on Android devices. In [19], the authors use the process trace system call to stop and resume processes and to create memory dumps of their address spaces. This is useful when data remains in memory only for a very short period. This happens, for example, when it is loaded and erased in only one routine.

In [2, 20], Apostolopoulos et al. search for authentication credentials in the process memory of applications. They use the Dalvik Debugging Monitor Server (DDMS) tool [20] and the LiME kernel module [12]. The authors execute both analyses on running mobile phones with root privileges.

Hilger et al. show a forensic application that uses the memory of cold booted devices in [21]. They create tools based on the FROST framework to analyze the heap of the Dalvik Virtual Machine. With this approach, they are able to obtain critical data, for example, the phone call history, the last user input, and passwords [22].

Research on mitigating cold boot attacks mainly focuses on protecting the FDE key against the attack [23–26]. In [24], the authors relocate the disk encryption key from RAM to the CPU registers of the ARM microprocessor. This goes back to [23], where they develop the approach for keeping the disk encryption key in registers for the x86 architecture. The work in [26] describes a software-based approach to protect the key while being in a private mode that allows using basic device functionality. In [27], the approach is to encrypt user data in RAM when the device switches to the screen-locked state. The utilized key is stored on the ARM SoC rather than in DRAM. There is however only little research on the application of the cold boot attack for sophisticated forensic analysis or for the inspection and improvement of security features. Anti-forensics techniques, e.g., [28] for the x86 architecture, aim to defeat memory acquisition modules by manipulating the physical address space layout. The research of how to apply cold boot attacks for forensics is still an ongoing research topic [5, 29].

3 Data interpretation

In order to interpret the raw data in main memory, we require meta information. The problem of data interpretation is the *semantic gap* in the fields of Virtual Machine Introspection (VMI) [30]. The required meta information depends on the OS, the kernel version, and its configuration. This information is necessary to determine the location of kernel structures and which components those structures include.

The kernel data structures are important for a full analysis in order to reconstruct the list of running processes and their memory mappings. In FROST, the full Linux kernel is booted and overwrites the data of the former running kernel. This causes a high amount of crucial information loss and results in non-reconstructible data.

In our architecture, we utilize the tool Volatility for data interpretation [31]. Volatility is an easily extendable open source framework for memory forensics. The framework consists of a collection of tools for acquiring memory, for bridging the semantic gap and for the extraction of relevant information. Volatility provides a decent amount of plug-ins that obtain detailed information on the target system. For example, the plug-in *linux_pslist* retrieves a list of running processes. The framework supports different operating systems (Microsoft Windows, Mac OS X, Linux, and Android) and architectures (x86 and ARM).

Volatility requires supplementary data about the target system. *Volatility profiles* reflect this supplementary data. Profiles contain the metadata about the structures and debug symbols of the kernel running on the target system. This information can be automatically extracted from the kernel source code.

4 Framework architecture

The goal of our approach is to construct a minimalistic module, which occupies only the smallest amount of memory in RAM. In doing so, we preserve the structures of the formerly running Android kernel when booting the module. We hence develop a minimalistic application, the BMA, that does not require a Linux kernel at runtime.

In order to obtain data from the mobile device, we use its serial interface. The corresponding driver is very efficiently implementable in terms of memory, compared to other hardware devices that transfer data from the mobile device, such as the USB interface. In general, our driver consists of reading and writing from a dedicated register of the UART interface.

In our framework, we only request relevant data from the target device (see Section 5.2) and assume that the contents in RAM do not degrade during the analysis (see Section 7). The utilized forensics tool is equipped with the decision logic about the relevant information in RAM. Figure 1 depicts an overview of our architecture. The illustration expresses that our proposed architecture consists of two parts: the minimalistic BMA on the mobile device and a forensic framework on a host system.

The left side of Fig. 1 clarifies that we map the BMA to the code segment of the previously running kernel. The BMA boots directly on the target system without requiring any other dependencies. The BMA implements a serial driver in order to receive and process commands from the forensics host. A command includes a physical start address and the amount of bytes the BMA must read and return. We propose a forensic framework for data acquisition and analysis on the host system as the second part of the architecture, as depicted on the right side of Fig. 1. The forensic framework directly requests data from the serial interface connected with the device via a serial cable. We use a simple protocol for the channel between the host and the BMA. The forensic framework uses data acquisition modules, for example, in order to retrieve the list of running processes. We explain the details of this process in Section 5.

For locating a process' virtual address, two steps are required during the analysis of a memory dump:

- Translation of virtual to physical addresses in main memory. This is hardware-dependent and requires the information stored in the page tables of the MMU.
- Translation of the physical address to an offset inside the dump. This depends on the storage format and requires the meta information for memory segment mapping inside the dump.

As shown in Fig. 1, the virtual address translation layer determines the virtual-to-physical address mapping. The data request layer then translates the physical address to an offset. This layer makes use of a UART driver to request the data from the BMA on the mobile device.

On the one hand, this architecture allows to gather full, genuine memory dumps from devices for later analysis. On the other hand, we can start a live analysis of the RAM. In both ways, we can reliably identify confidential data, ranging from FDE keys to vast amounts of confidential user data. The flexibility of the architecture allows for custom extensions, depending on the goal of an investigator. Volatility is our choice of forensics tool on the host, but other tools that systematically analyze memory dumps can be taken into account. Utilizing exhaustive search tools or plug-ins to harness the BMA over the serial interface results in poor performance. Since the kernel code segment is way larger than the size of our BMA, we can extend the BMA and its communication protocol. One such possibility is to establish algorithms that search for known patterns or structures, such as FDE keys [3, 32], and other AES and RSA key structures [11]. The advantage is that memory can be rapidly accessed from within the BMA. Another extension of the BMA can be to introduce a simple compression mechanism to the protocol.

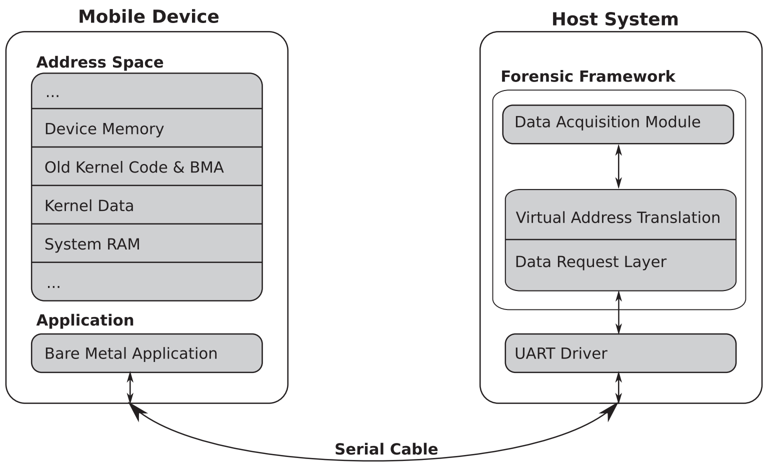

Fig. 1 Schematic overview of the forensic architecture showing the BMA communicate with the forensic framework on the host

This increases the performance of the data requests, for example, when an application requests large amounts of data which mostly consist of zerobyte chunks.

5 Implementation

For the realization of the framework, we first describe the implementation of our BMA. We then elaborate the extension of the memory forensics tool Volatility which we use for data acquisition and analysis.

5.1 Bare metal application

There are two ways to execute a cold boot attack. The first one is to move the memory module to another host machine with an installed forensics tool. The second possibility is to halt the system and to abruptly restart the device in order to directly boot a forensics tool on the device itself, which is the BMA in our case. Only the second approach is feasible on mobile devices, because the memory is integrated into the SoC and cannot be removed.

The main tasks of the BMA are to initialize the UART connection and to process incoming data requests. We utilized a large portion of the code for initializing the UART connection directly from the Qualcomm Linux kernel [33]. The BMA waits for incoming requests on the UART port. In the next step, the BMA reads and parses these requests. We implemented the parsing routine for incoming commands in a minimal way, saving allocated memory. After parsing, the BMA requests the corresponding data from RAM and writes it to the UART interface.

In contrast to the Linux kernel, our BMA does not initialize any device and only configures the UART interface. We thus preserve device memory, which might be altered by the set-up routine of a device driver of the Linux kernel. The BMA allows to request arbitrary memory chunks up to a full image of the RAM. The UART interface provides only a low data transfer rate, which makes the extraction of a full dump of main memory inconvenient.

As an extension, we implemented an FDE key search functionality based on the kernel structures wrapping this key, close to [32]. This functionality also comes with a simple heuristic that identifies keys even when bit flips occur. We trigger this functionality with a further command in the protocol to make the BMA search and return the keys found.

5.2 Volatility extension

We execute the forensic analysis of the contents of main memory with the memory forensics tool Volatility. The tool commonly requests data from memory dumps, e.g., acquired with the LiME module.

With the concept of Address Spaces (ASs), Volatility provides the possibility to translate addresses. In our case, this is the translation of virtual to physical addresses and determining the correct position of an address in a memory dump. Depending on the use case, Volatility allows to combine and stack ASs. We applied the *ARM Address Space* for virtual to physical address translation in our concept [22]. According to our architecture, we extended Volatility with the implementation of a custom AS that requests data over the serial port, instead of from a memory dump file. Furthermore, we require a *Volatility profile* for the specific device under analysis [21].

5.2.1 The serial address space

We developed a new Volatility AS for data acquisition via the serial port. We call this AS the *serial address space*, which resembles the *file address space*. The latter requests the contents from a local memory dump file. Instead, the serial AS implements the protocol required to request data from the BMA via the UART interface. The serial AS opens the serial port upon its initialization. Consequently, when a plug-in requests data from an address, the serial AS directly passes the request to the BMA on the target device. In order to do so, the serial AS writes the dump command with the start and end address of the request to the UART interface. The BMA receives and then parses the dump command. The BMA reads the requested data from memory and writes it back to the UART interface. The serial AS then receives the requested data and returns it to overlying ASs. As some address requests are frequently made, such as for reading the page tables, we equipped the serial AS with a cache that stores the data of former requests. In case a request occurs again, we immediately return the data from the cache instead of consulting the BMA. This likely decreases the duration of the analysis many times over.

Figure 1 depicts the different layers used by a standard Volatility plug-in, such as *linux_pslist*. The plug-in retrieves the list of running processes by following the linked list of task structures, called `task_struct`, in the Linux kernel. Since our target system is on the ARM architecture, Volatility makes use of the *ARM address space* for virtual to physical address translation. The serial AS sends the request from the layer above to the BMA via the UART interface. The BMA reads the requested memory and returns it via the UART interface to Volatility.

5.2.2 The volatility profile

Volatility stores the meta information required to interpret the data in main memory in a Volatility profile. The profile bridges the semantic gap (see Section 3). For example, it provides a map of the data structures in memory. A profile strongly depends on the OS and the corresponding kernel of the target device. In order to create a Linux profile, the source code of the deployed kernel is required (see Section 6).

6 Device-specific realization

We selected the Samsung Galaxy S4 GT-I9505 device for our specific implementation case. This device is a commonly used mobile phone and provides a serial port. We deployed the CyanogenMod Android 4.4.4 distribution (version 11-20141008-SNAPSHOT-M11-jflte) on our target device. For the Volatility profile, we utilized the corresponding kernel sourcecode (kernel version 3.4.104-cyanogenmod-g42b4b50-dirty) [33]. We describe the integration of the BMA onto the device's recovery partition by wrapping it into an Android boot image. Then, we describe the boot procedure in order to launch the BMA. To be able to connect the BMA with a host system, we also describe our hardware setup. We show that our solution is easy to port to other devices that offer a serial interface by the example of the Nexus 5 device.

```
...
2a03f664-2a03f6a4 : pc-cntr
2a03f720-2a04071f : tz_{l}og.0
80200000-87dfffff : System RAM
80208000-80f8e523 : Kernel Code
8111a000-817a6da3 : Kernel Data
89000000-8d9fffff : System RAM
8ec00000-8fdfffff : System RAM
8ff00000-9fdfffff : System RAM
a6700000-fe1fffff : System RAM
fff00000-fffefff : System RAM
```

Listing 1 Truncated output of `/proc/iomem` on a Samsung Galaxy S4 device.

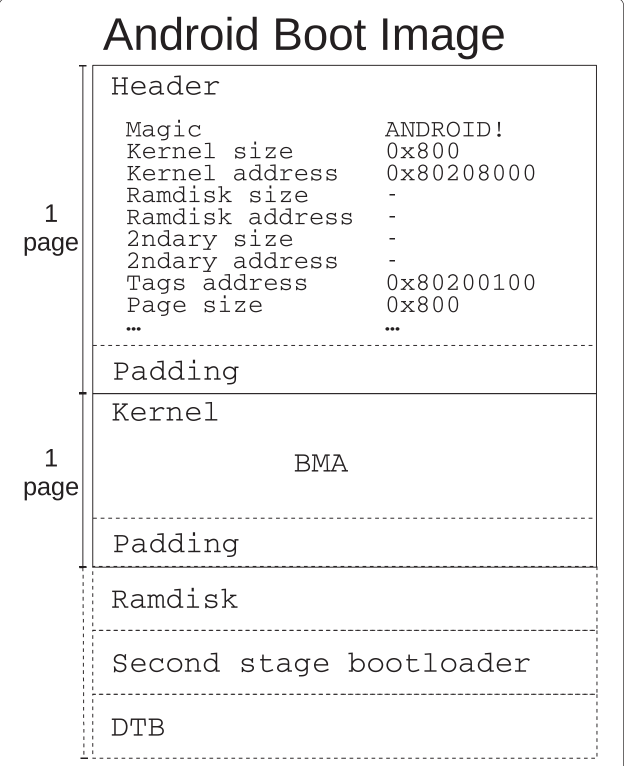

Fig. 2 The BMA wrapped into an Android boot image

6.1 Wrapping and deploying the BMA

In order to deploy the BMA on the recovery partition of the mobile device, we wrapped the BMA into an Android boot image. This makes the device's bootloader capable of loading this image. Our generated boot image contains the configuration for the mapping of the BMA in memory at runtime.

Figure 2 illustrates the structure of the boot image. An Android boot image comes along with a preliminary header of one flash page size. The header always has an 8-byte magic start value (`ANDROID!`). The header contains the necessary fields of different sizes to advise the bootloader about the image's structure. In the kernel address field, we store the address the BMA gets mapped to (`0x8020800`). The flash page size of the Samsung Galaxy S4 device is set to 2048 bytes (`0x800`). The size of the kernel, here the BMA, is one such page. The BMA is located right after the boot image header where the compressed kernel binary can normally be found. Our boot image neither contains a ramdisk, a second stage bootloader, nor a Device Tree Blob (DTB). The remaining fields in the header are optional and can remain empty. This depends on the device's bootloader (see Section 7.3).

Our configuration maps the BMA to the address `0x80208000`, where the code segment of the formerly running Linux kernel is located (see Listing 1). Furthermore, the initial code of the BMA sets the BMA's 1024-byte stack adjacent to itself, i.e., also in the kernel code section. In doing so, we overwrite no more than 3 KB of memory completely located in the kernel code segment. This segment solely contains constant data deduced from kernel code.

In order to launch the BMA, we flash the generated Android boot image to the recovery partition of the device. We execute this step after the short power cut-off, respectively, the hardware reset of the device.

6.2 Booting the BMA

In order to deploy and boot the BMA on the target device, we need to be aware of the device's boot procedure. Figure 3 illustrates a schematic representation of the boot process. The illustration describes the different boot stages between pressing the power button and booting the recovery image. The bootloader boots our pre-deployed boot image from the recovery partition and places the BMA inside the former kernel code segment. In order to do so, the bootloader reads out the values inside the boot image header and also checks the header's magic value.

The common boot procedure requires the following steps. When turning on a device, the system initializes the hardware and executes the routines stored in the boot

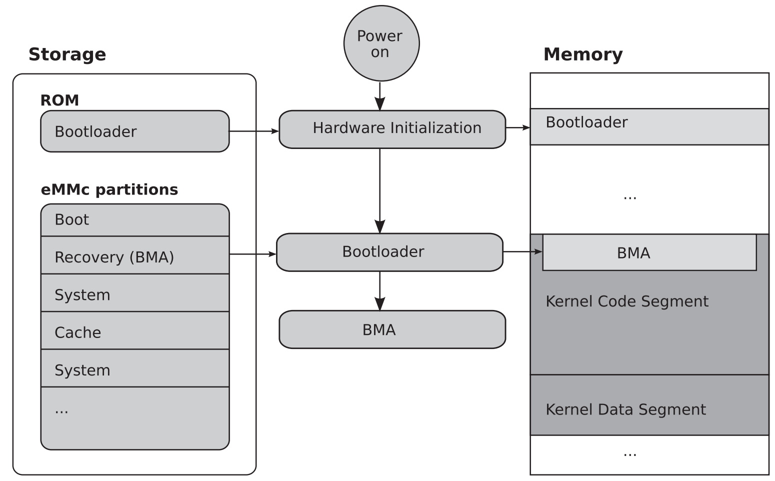

Fig. 3 Boot procedure starting the BMA from the recovery partition on a cold booted device

ROM. These routines load the bootloader. Upon its execution, a routine in the bootloader checks the status of the hardware buttons to identify which boot mode to trigger.

In general, most Android devices can boot in three different modes: *normal*, *recovery*, and *download*.

Figure 3 describes the scenario where a startup into *recovery mode* takes place. The purpose of the *recovery mode* is to update, install, or repair an Android system. In order to trigger the *recovery mode*, a special key combination must be pressed when switching on the device. On the Samsung Galaxy S4, this is the *Volume Up* and *Power* button.

The *normal mode* is the default boot mode. This mode starts the Android OS when switching on a device the common way. In order to do so, the bootloader starts the kernel stored on the boot partition of the embedded Multi-Media Controller (eMMC).

The *download mode* allows to directly write data to partitions via USB, e.g., to the recovery partition. In order to trigger the *download mode* on the Galaxy S4, the *Volume Down* and *Power* buttons must be pressed. We used the tool *Heimdall* [34] to write the BMA via the USB interface to the device's recovery partition in this mode.

6.3 Hardware setup

We realized the connection between the test device and the host system, a common PC, via the Samsung Anyway Jig [35]. The jig serves as a universal maintenance tool for various devices produced by Samsung. It is equipped with a D-Sub DB-25 connector and can be connected to the supported device via a proper adapter cable. To connect the jig to the test device, we used a custom Micro-USB to D-Sub DB-25 cable. We established the connection to the host PC via the RS-232 interface. The Samsung Anyway Jig Adapter connects the GND and ID-line of the Micro-USB of the device port with a resistor. This configures the micro-USB port such that it acts as a serial interface port. We configured the dimension of the resistor with the DIP-switches of the Samsung Anyway Jig. After connecting to the UART port, the whole boot process of a device can be monitored because the bootloader writes debug information to the serial interface. Depending on the kernel command line parameters, the kernel can also output its debug information over the serial interface.

6.4 Portability

We verified the easy portability by realizing the framework on the Nexus 5 device. We did not extend Volatility, as the only requirement is a new profile based on the Nexus 5 kernel code. For the BMA, only three changes to our created boot image were necessary.

First, the BMA had to be remapped in memory. According to the different iomem layout of the Nexus 5 device, the first system RAM segment starts at address zero and the kernel code segment is located at `0x8000`. Second, the base offset value of the UART registers in the BMA had to be adjusted. This is because the UART core is mapped to a different location in memory compared to the Samsung Galaxy S4 device. We could completely reuse the serial driver of the BMA for the device.

Third, the bootloader on the Nexus 5 device expects a DTB. Otherwise, the bootloader refuses to boot up. The bootloader locates the DTB in the boot image based on an offset at a specific address in the boot image's kernel image. We modified the BMA at that address with an offset that points straight after the BMA. This allowed us to append a minimal DTB to the BMA, which the bootloader accepts. The bootloader relocates the DTB to the tags address. We do not overwrite crucial data, because at that location, former constant valued DTBs can be found. The bootloader required no ramdisk or second stage bootloader. To flash the boot image to the device, we used the tool *Fastboot* [36]. The connection to the UART interface is established via the device's headphone socket. Therefore, we crafted a 3.3V USB to serial cable that we soldered onto a headphone jack.

7 Evaluation

In this section, we evaluate our proposed framework. We first measure the amount of data that degrades due to the cold boot attack and show the feasibility of our approach in Section 7.1. In Section 7.2, we demonstrate the application of our framework on the Samsung Galaxy S4 device using Volatility. We compare the Volatility analysis on a traditional LiME memory dump with a cold boot-based analysis using our BMA. In the next step, we extract sensitive user data from a cold booted device based on a concrete-use case in order to demonstrate the potential of our framework. In Section 7.3, we discuss the implementational aspects of the framework.

7.1 Loss of information

We evaluate the loss of information with our architecture considering three aspects: the decay of information through the device restart, the duration of the analysis, and the BMA's size.

7.1.1 Decay based on the cold boot attack

We executed the two different types of the cold boot attack to evaluate the amount of data exposed to degradation during the power cut-off:

1. Momentary removal of the battery and restart of the device once the battery re-inserted.
2. Power button press for a few seconds while the phone is still running, causing a hardware-based reset.

Note that we executed both attacks at a temperature of approximately 20° C. The results improve when cooling down the phone and its memory modules, as described in [1, 10, 37].

Before rebooting the device, we wrote an array of 10,000 known bytes to main memory with a kernel module printing the physical start address of the array. Afterwards, we read the contents of the array's physical address and counted the unimpaired bytes. We executed the analysis 25 times. Table 1 depicts the corresponding results. In case of the *fast cold boot attack*, we re-inserted the battery as fast as possible. In case of the *slow cold boot attack*, we re-inserted the battery after approximately 1 s.

Table 1 Number of successfully retrieved bytes of a 10,000-byte array with different cold boot attacks

Attack type	Min	Max	Average
Fast cold boot	9983	9998	9991
Slow cold boot	26	9521	3379
Reset cold boot	9998	10,000	9999

The depicted results mainly show that a non-reset-based cold boot attack does not always return reliable results as it depends on multiple factors like the temperature and the abilities of an adversary. In case of the fast cold boot attack, we retrieved 9991 of the 10,000 known bytes in average. Despite the only weak degradation, the application of Volatility plug-ins became difficult (see Section 7.2). When conducting a slow cold boot attack, the degradation proceeded quickly. In this case, the successful application of Volatility plug-ins got infeasible.

However, the reset-based attack provided the better results during our tests. This scenario is more reliable as it does not depend on how fast the battery can be re-inserted. In most cases, we retrieved all of the bytes successfully and had occasional bit flips only in very few test cases. This shows that even in case of the reset attack, where we did not remove the battery at all, the memory occasionally decays. In normal cases, where we retrieved all of the bytes correctly, the application of Volatility plug-ins was always successful.

In our test set-up, it was also possible to successfully extract data with our architecture from the device when it was restarted twice. A second restart is required on the Samsung Galaxy S4 device to reboot into recovery after flashing the BMA onto recovery partition. On the Nexus 5 device, the bootloader allows to immediately boot from a partition after flashing.

7.1.2 Decay during the analysis

As we do not completely save the data in main memory at once, we rely on the data to remain intact on the target device during the whole analysis process. To demonstrate that this requirement is given, we executed our attacks multiple times 15 min after the device has booted the BMA. As expected, we retrieved exactly the same unaltered data between the test requests. Furthermore, we inspected memory dumps on Nexus 5 devices running a virtualization architecture that comes along with FDE using dm-crypt [38]. Gathering the dump of the 2 GB RAM with the BMA required about 42 h. We compared this dump to a LiME dump created shortly before, finding that there was no decay during the analysis. As an example, we were able to recover the FDE keys in the LiME dump, as well as in the BMA's dump. In addition, we were able to quickly return the keys using the BMA's FDE key search functionality. For this reason, we may assume that data does not decay any further as long as the BMA executes and memory is supplied with power.

7.1.3 Information loss based on the size of the BMA

Another important source of data loss is the amount of data in RAM that the BMA occupies. The boot image containing our BMA has a size of 4 kB and the size of the stack that the BMA sets up is 1024 bytes. The bootloader loads the image header of size 2048 bytes to a fixed location in memory. Based on the information in the header, the bootloader maps the BMA of the same size to the kernel code section. Thus, we overwrite no more than 3 kB of memory in the kernel code section. This is less than the previously running kernel's size. The size of the CyanogenMod boot image for the Galaxy S4 device is about 5.9 MB. The kernel is compressed and extracted before it gets started. The kernel code segment forms a set of constant and known bytes. The segment does not change between different runs of the system. This means that the overwritten bytes do not impair relevant data as the code segment of the formerly running kernel is mapped to this address range (see Section 5.1).

7.2 Volatility plug-ins

We compare the application of multiple Volatility plug-ins with the BMA to a traditional LiME dump-based analysis in the first part. With this comparison, we show that we are able to similarly analyze the data retrieved from the BMA compared to previously recorded memory dumps on the running system. Thereby, we also focus on the performance impact and caching effect during the BMA analysis. In the second part, we demonstrate the potential of our framework by retrieving sensitive user data through an analysis with various plug-ins.

7.2.1 Comparison with LiME dump analysis

We created a memory dump on the running phone with the LiME kernel module as a reference right before we executed the cold boot attack. Then, we reset the device to execute the Volatility plug-ins with our framework using the BMA running on the phone. Afterwards, we restarted the device and pulled the LiME dump to execute the same set of plug-ins on the dump file. Table 2 lists the results and runtimes of the plug-in applications on the LiME dump and on our BMA in combination with the reset attack.

The plug-in *linux_pslist* extracts a list of running processes. The resulting entries of our measurements differed between the dump file (230 entries) and the cold boot analysis (225 entries) by solely five more threads. This comes from the LiME kernel module creating these threads during the acquisition process. The analysis with the BMA took 24.23 s, whereas the LiME dump analysis took 3.54 s.

The plug-in *linux_iomem* extracts the map of the system's memory for physical devices. The results received from the cold boot-based analysis were equal to the memory dump analysis. As in the scenario before, the runtime of the plug-in was longer in case of the BMA application. Compared to the LiME dump analysis with a duration of 2.18 s, the analysis with the BMA took 8.97 s. According to Table 2, the application of other plug-ins yields comparable runtime differences between the BMA and the LiME dump analysis.

The plug-in *linux_proc_maps* returns the memory mappings of a single process. This renders results similar to contents in `/proc/<pid>/maps`. For our measurements, we requested the mappings of the `init` and the `rild` process. The latter is responsible for the radio functionality of an Android phone. In both cases, the measurements returned exactly the same results: nine entries in case of the `init` process and 156 entries in case of the `rild` process.

We finally requested the stack and heap memory segments of the `rild` and the `init` process with the plug-in *linux_dump_maps*. The amount of data in bytes was, for both processes, the same for the stack and heap. The data of the stack of the `init` process turned out to be consistent between the two acquisition methods. The same holds for the `rild` process.

In every test case, the required time for executing a plug-in which operates on the memory using the BMA was significantly higher. This emerged as a result of the low transfer rate of the UART interface. The average transfer rate we measured with our hardware was at about 11.25 KB/s when we request large chunks of data. This speed reduces when plug-ins make lots of small data requests during the analysis due to the BMA's protocol overhead. For our purposes, the low transfer rate was acceptable, since the plug-ins terminated within less than 45 s.

```
Offset NamePid Uid Gid ...
0xc000e000 init 1 0 0
0xde9da400 keystore 227 1017 1017
0xdd25ac00 d.process.acore 798 10003 10003
0xdcea3000 m.android.phone 828 1001 1001
0xdbf8e000 m.android.email 1480 10032 10032
0xdbeb7c00 droid.gallery3d 1505 10035 10035
0xdc1a8400 ndroid.exchange 1522 10033 10033
0xdcd0d000 ndroid.contacts 1689 10003 10003
0xdbb99c00 mod.filemanager 2028 10022 10022
0xdca5bc00 android.browser 2364 10020 10020
```

Listing 2 Truncated output of the plug-in *linux_pslist* acquired during a user session.

The speed strongly increases due to the caching functionality in our serial AS, which buffers previous requests. Data once requested from the device is thereby stored in the cache. Caching was, in particular, useful for plug-ins that accessed the same sets of addresses frequently, such as *linux_lsof*. Furthermore, all the plug-ins frequently requested only small amounts of bytes at a time from

Table 2 Different Volatility plug-ins and their runtime using a dump file and the BMA

Plug-in	Results (LiME dump)	Results (BMA)	Time (LiME dump)	Time (BMA)
linux_pslist	230 entries	225 entries	03.54 s	24.23 s
linux_iomem	138 entries	138 entries	02.18 s	08.97 s
linux_proc_maps (init)	9 entries	9 entries	02.00 s	06.03 s
linux_dump_maps (init, heap)	340.0 KB	340.0 KB	01.98 s	35.84 s
linux_dump_maps (init, stack)	139.3 KB	139.3 KB	01.94 s	07.02 s
linux_proc_maps (rild)	156 entries	156 entries	05.38 s	18.55 s
linux_dump_maps (rild, heap)	380.9 KB	380.9 KB	02.05 s	41.48 s
linux_dump_maps (rild, stack)	139.3 KB	139.3 KB	02.06 s	08.29 s

the memory during the analysis. For example, the plug-in *linux_pslist* requested about 93 KB of data in total, 4 bytes in average, and due to caching we reduced this amount to about 12 KB. The plug-in *linux_iomem* requested about 58 KB of data in total, 9 bytes in average, and due to caching, we reduced this amount to about 30 KB. The application of other plug-ins yields comparable results. Considering the time for dump file creation in other approaches, our framework can even provide faster results.

7.2.2 *Acquisition of sensitive user data*

```
Pid FD Path
828 0 /dev/null
828 70 pipe:[12680]
828 71 /data/data/com.android.providers.telephony/
data-bases/mmssms.db
828 88 anon_{i}node:[4225]
828 89 /data/data/com.android.providers.telephony/
data-bases/telephony.db
```

Listing 3 Truncated output of the plug-in *linux_lsof* acquired during a user session.

We show the potential of our framework for straight data acquisition at the example of a real user session. Since we already showed how we retrieve FDE keys with the BMA, we focus on other sensitive assets in the following. There is way more confidential information to detect, which is potentially never made persistent and can only be found in RAM. Therefore, we created a potential usage scenario where the user enters confidential data on the phone, which is masked in the following. After the scenario, we reset the device, flashed and booted the BMA. Then, we started an investigation using various Volatility plug-ins.

The scenario starts by booting the phone, using it for approximately 15 min and ends after leaving it idle for about 1 min. During that time, the user carries out the following activity:

- Create a new contact *Secret Contact* with phone number *017** in the contacts application.
- Synchronize a previously set-up exchange account within the mail application.
- Create a draft short message *Top Secret Short Message Draft* to *Secret Contact* using the messenger application.
- After a while, edit the stored short message draft to *Top Secret Message* and send the message.
- Visit webpages with the browser and use search engines. Log-in to pages with a user account and enter confidential data, such as *Top Secret Information.*
- With the filemanager, create a new file */data/secret.-txt* and edit the file with the content *Top Secret Text.*

As a first step of the analysis, the investigator with physical access to the device retrieves the process list with *linux_pslist* (see Listing 2). The full list has 239 entries in total. The amount of processes that an investigator suspects private data to be contained is way smaller. Inspecting the open file handles of the phone process `com.android.phone` with the plug-in *linux_lsof* reveals the potential sensitive file *mmssms.db*. The truncated list is depicted in Listing 3. In total, the plug-in finds 90 open files, but most of them can be left out of consideration.

```
h13Top Secret Short Message Draft
Top Secret Message
004917*
Top Secret Short Message Draft
```

Listing 4 Truncated output of a file acquired with the plug-in *linux_find_file* during a user session.

Using the plug-in *linux_find_file*, we searched the corresponding inode and retrieved the cached file contents of about 103 KB. By dumping the strings of the read file, we obtained about 105 strings. This helped us to quickly recognize the recipient, the initial draft, and the edited message, see the truncated output in Listing 4. The plug-in *linux_lsof* is especially useful for determining open files of processes, such as logs.

As a next step of the analysis, we retrieved the memory segments of the process `com.android.exchange` using the plug-in *linux_proc_maps*. We suspected relevant data of the process to be located in the processes' heap segment. Listing 5 shows the output cut to the lines containing the keyword heap.

```
PID Start EndFlags Pgoff Major Minor
Inode Path
1522 0x41a22000 0x41a2a000 rw- 0x0 0 0
0 [heap]
1522 0x41e82000 0x61a2a000 rw- 0x0 0 4
8872 /dev/ashmem/dalvik-heap
```

Listing 5 Snippet of the output of the plug-in *linux_proc_maps* for the Android exchange process.

Using the plug-in *linux_dump_maps*, we retrieved the heap segments. The string output of the Dalvik-heap segment quickly revealed the mail account's username and password separated by a semicolon *firstname.lastname@*.de:**. Even though the Dalvik-heap segment seems to be large, the request for the segment was quickly handled, because Volatility recognizes that the segment is sparsely allocated.

We conducted the same steps for the process `android.process.acore`, which serves as Android's contact provider, for `com.cyanogenmod.filemanager` and for `com.android.browser`. Inside anonymous memory segments, we were able to find the contact *Secret Contact* with phone number *017**. The browser's memory segments contain vast amounts of loaded websites, user account names, search queries, and text entered in webmail and social media pages. This made it possible to recover entered data, such as *Top Secret Information*. The Dalvik-heap of the filemanager exposes the filename */data/secret.txt* and its content *Top Secret Text*.

In a further step of the investigation, we inspected the data in the routing table cache with the plug-in *linux_route_cache*. We recovered the hosts we recently connected to during our browsing session, such as our webmail page (see Listing 6).

```
Interface Destination Gateway
--------- --------------- -------
wlan0 131.159.0.91 10.144.207.1
wlan0 173.194.112.136 10.144.207.1
wlan0 131.159.0.91 10.144.207.1
lo10.144.207.3910.144.207.39
```

Listing 6 Snippet of the output of the plug-in *linux_route_cache* acquired during a user session.

In order to successfully and efficiently carry out an analysis, the investigator has to be aware of where Android processes store their relevant data. Open-file handles and the Dalvik-heap are the most probable locations to expose such data. We were in knowledge of the data we were searching for in our scenario. However, relevant processes and data can be relatively quickly identified and filtered from memory dumps.

7.3 Aspects of the implementation

As we read memory from a cold booted device, we need to be aware of the decay of data. In case of corrupted data, this means for example that the pointers in the `task_struct` of the Linux kernel cannot be correctly dereferenced. This causes the forensic analysis to fail at some point, because previously running tasks cannot be detected. In order to treat these cases, we propose to extend forensics tools to work in combination with corrupted data acquired by a cold boot attack. Heuristics can help fix invalid pointers or to at least ignore them. However, our data remains in almost all of our test cases intact so that we did not have to deal with this problem. This is due to the reset attack where the battery is not removed. The feasibility of the reset attack depends on whether the specific device offers the hardware reset functionality or not.

Another important aspect is that the target device must air a UART port. A lot of devices have it even though it is not visible at first glance. The UART port is often integrated into the micro-USB port or the headphone socket.

Care has to be taken considering the bootloader. Using the Samsung Galaxy S4 device, the bootloader accepted a simply crafted boot image, but the requirements changed for the Nexus 5 bootloader. To figure out what the bootloader requires is not always obvious, but open source bootloader code helps to recognize such requirements [39]. Fortunately, most of the mobile device bootloaders work similarly. Nevertheless, it is possible that bootloaders are capable of overwriting volatile data, which would represent an inevitable problem.

The deployment of the BMA onto the device either requires write access to the recovery partition at runtime or a device where the bootloader can be unlocked for flashing the BMA. However, write access is only possible with root privileges. In case the bootloader is locked, it has to be unlocked before flashing partitions. Unlocking the bootloader normally leads to erasing all user data on persistent storage. Persistent memory can then no longer be recovered. But with our method, volatile memory remains unimpaired when unlocking the bootloader and we do not require root privileges on the phone. This means that we are still able to recognize crucial contents of the previous session in RAM, which were possibly never made persistent.

We expect that our framework can be used for further topics because our implementation is easy to extend and can be easily ported to other devices. The framework can be used, for example, to evaluate whether it is possible to access application memory running in the secure world of the TrustZone [6].

8 Conclusions

In this paper, we presented a forensic framework for mobile devices based on the cold boot attack. In contrast to other state-of-the-art implementations, we do not boot a full-fledged Linux kernel on the target device. Instead, we boot our easily portable minimal BMA, which occupies no more than 3 KB in the RAM. The BMA preserves the data structures of the previously running kernel and does not reset device memory. As we only overwrite constant data in the kernel code section, this ensures that all of the important kernel data remains available for analysis. The BMA provides a serial communication interface. This interface allows to dynamically request parts of the main memory. Forensic analysis can thus be conducted on the host system. For this purpose, we extended Volatility with a serial communication module for the analysis of the target device's memory.

We realized the framework for the Samsung Galaxy S4 and ported it to the Nexus 5 device in order to demonstrate the feasibility of our approach. In our evaluation, we compared our cold boot-based analysis with traditional memory dump analysis using Volatility showing proper results. We have shown that our BMA allows to request full, genuine memory dumps and to efficiently gather vital information based on the sustained kernel structures, such as FDE keys and further confidential data.

Acknowledgements
The research leading to these results was supported by the "Bavarian State Ministry of Education, Science and the Arts" as part of the FORSEC research association.

Authors' contributions
MH, supported by BT, developed the concept and design of the framework. MH and BT both implemented and applied the prototypes for the framework. MH worked on the acquisition and analysis of data in order to evaluate the framework. Both authors elaborated on the manuscript and revised it until it reached its final state. SW, HR, and GS have been involved in revising the manuscript carefully for important intellectual content. GS actively supervised the project from which the foundations of this work stem. All authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Fraunhofer Research Institute AISEC, Munich, Germany. [2]University of Passau, Passau, Germany. [3]Technische Universität München, Munich, Germany.

Received: 11 January 2016 Accepted: 26 July 2016
Published online: 11 August 2016

References

1. T Müller, M Spreitzenbarth, in *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*. ACNS'13. FROST: forensic recovery of scrambled telephones (Springer, Berlin, Heidelberg, 2013), pp. 373–388
2. Evaluating the privacy of Android mobile applications under forensic analysis. Comput. Secur. **42**, 66–76 (2014). doi:10.1016/j.cose.2014.01.004. http://www.sciencedirect.com/science/article/pii/S0167404814000157
3. T Pettersson, *Cryptographic key recovery from Linux memory dumps*. (Presentation, Chaos Communication Camp, Finowfurt near Berlin, Germany, 2007)
4. Peter Gutmann, in *Proceedings of the 10th Conference on USENIX Security Symposium - Vol. 10*. SSYM'01. Data remanence in semiconductor devices (USENIX Association, Berkeley, CA, USA, 2001)
5. M Gruhn, T Müller, in *Eighth International Conference on Availability, Reliability and Security (ARES '13)*. On the practicability of cold boot attacks (IEEE Computer Society, Washington, DC, USA, 2013), pp. 390–397
6. ARM security technology: building a secure system using TrustZone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. Accessed Dec 2015
7. R Agarwal, S Kothari, in *Information Science and Applications*. Lecture Notes in Electrical Engineering, vol. 339. Review of digital forensic investigation frameworks (Springer, Berlin, Heidelberg, 2015), pp. 561–571
8. A Hoog, *Android forensics: investigation, analysis and mobile security for Google Android*. (Elsevier, Amsterdam, Netherlands, 2011)
9. EM Chan, JC Carlyle, FM David, R Farivar, RH Campbell, in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08. BootJacker: Compromising computers using forced restarts (ACM, New York, NY, USA, 2008), pp. 555–564
10. AJ Haldermanand, SD Schoen, N Heninger, C William, P William, JA Calandrino, AJ Feldman, J Appelbaum, EW Felten, Lest we remember: cold-boot attacks on encryption keys. Commun. ACM. **52**(5), 91–98 (2009)
11. Center for Information Technology Policy at Princeton University, Memory Research Project Source Code (2015). https://citp.princeton.edu/research/memory/code
12. J Sylve, Android Mind Reading: Memory Acquisition and Analysis with DMD and Volatility. ShmooCon'12 (2012)
13. Azimuth Security: Dan Rosenberg: Re-visiting the Exynos memory mapping bug. http://blog.azimuthsecurity.com/2013/02/re-visiting-exynos-memory-mapping-bug.html. Accessed Dec 2015
14. C Devine, G Vissian, in *Proceedings of SSTIC '09*. Compromission physique par le bus PCI (Thales Security Systems, Ulm, Germany, 2009)
15. M Becher, M Dornseif, CN Klein, in *CanSecWest*. FireWire: all your memory are belong to us, (2005)
16. C Maartmann-Moe, Adventures with Daisy in Thunderbolt-DMA-land: Hacking Macs through the Thunderbolt interface (2012). http://www.breaknenter.org/2012/02/adventures-with-daisy-in-thunderbolt-dma-land-hacking-macs-through-the-thunderbolt-interface
17. R-P Weinmann, in *Proceedings of the 6th USENIX Conference on Offensive Technologies*. WOOT'12. Baseband attacks: remote exploitation of memory corruptions in cellular protocol stacks (USENIX Association, Berkeley, CA, USA, 2012), pp. 2–2
18. Rocker team flashing interface: RIFF Box. http://riffbox.org. Accessed Dec 2015
19. VLL Thing, K-Y Ng, E-C Chang. Live memory forensics of mobile phones, vol. 7 (Elsevier, Amsterdam, Netherlands, 2010), pp. 74–82
20. D Apostolopoulos, G Marinakis, C Ntantogian, C Xenakis, in *Collaborative, Trusted and Privacy-Aware e/m-Services*. IFIP Advances in Information and Communication Technology, volu. 399. Discovering authentication credentials in volatile memory of Android mobile devices (Springer, Berlin, Heidelberg, 2013), pp. 178–185
21. C Hilgers, H Macht, T Müller, M Spreitzenbarth, in *Proceedings of the 2014 Eighth International Conference on IT Security Incident Management & IT Forensics*. IMF '14. Post-mortem memory analysis of cold-booted android devices (IEEE Computer Society, Washington, DC, USA, 2014), pp. 62–75
22. J Sylve, A Case, L Marziale, GG Richard, Acquisition and analysis of volatile memory from android devices. Digital Invest. **8**(3–4), 175–184 (2012)
23. T Müller, FC Freiling, A Dewald, in *Proceedings of the 20th USENIX Conference on Security*. SEC'11. TRESOR runs encryption securely outside RAM (USENIX Association, Berkeley, CA, USA, 2011), pp. 17–17
24. J Götzfried, T Müller, in *Proceedings of the 2013 International Conference on Availability, Reliability and Security*. ARES '13. ARMORED: CPU-bound encryption for Android-driven ARM devices (IEEE Computer Society, Washington, DC, USA, 2013), pp. 161–168
25. T Müller, B Taubmann, FC Freiling, in *Proceedings of the 10th International Conference on Applied Cryptography and Network Security*. ACNS'12. TreVisor: OS-independent software-based full disk encryption secure against main memory attacks (Springer, Berlin, Heidelberg, 2012), pp. 66–83
26. A Skillen, D Barrera, PC van Oorschot, in *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. SPSM '13. Deadbolt: Locking down Android disk encryption (ACM, New York, NY, USA, 2013), pp. 3–14
27. P Colp, J Zhang, J Gleeson, S Suneja, E de Lara, H Raj, S Saroiu, A Wolman, in *Proceedings of the 20th Int. Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. Protecting data on smartphones and tablets from memory attacks (ACM, New York, NY, USA, 2015), pp. 177–189

28. N Zhang, K Sun, W Lou, S Hou, YT Jajodia, in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS '15. Now you see me: hide and seek in physical address space (ACM, New York, NY, USA, 2015), pp. 321–331
29. R Carbone, C Bean, M Salois, An In-Depth Analysis of the Cold Boot Attack: Can It Be Used for Sound Forensic Memory Acquisition? (2011)
30. PM Chen, BD Noble, in *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*. HOTOS '01. When virtual is better than real (IEEE Computer Society, Washington, DC, USA, 2001), p. 133
31. The Volatility Foundation: open source memory forensics. http://volatilityfoundation.org. Accessed Dec 2015
32. Joey Hewitt: GitHub - Scintill/keysearch: Search an Android RAM dump for Linux Dm-crypt (incl. LUKS/cryptsetup) Keys. https://github.com/scintill/keysearch. Accessed Dec 2015
33. CyanogenMod: CyanogenMod Android kernel Samsung Jf. https://github.com/CyanogenMod/android_kernel_samsung_jf. Accessed Dec 2015
34. Glass Echidna: Heimdall. http://glassechidna.com.au/heimdall. Accessed Dec 2015
35. XDA Developers: What is the Samsung Anyway Jig? http://xda-developers.com/what-is-the-samsung-anyway-jig. Accessed Dec 2015
36. Google: ADB Fastboot Install - A script to install ADB & Fastboot on Mac OS X and/or Linux. https://code.google.com/p/adb-fastboot-install. Accessed Dec 2015
37. S Lindenlauf, H Hofken, M Schuba, in *10th International Conference on Availability, Reliability and Security (ARES '15)*. Cold boot attacks on DDR2 and DDR3 SDRAM (IEEE, 2015), pp. 287–292. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7299495
38. M Huber, J Horsch, M Velten, M Weiß, S Wessel, in *11th International Conf. on Information Security and Cryptology - Inscrypt*. A secure architecture for operating system-level virtualization on mobile devices (Springer, Berlin, Heidelberg, 2015)
39. CodeAurora Forum: (l)ittle (k)ernel based Android bootloader. https://codeaurora.org/blogs/little-kernel-based-android-bootloader. Accessed Dec 2015