TrustDump: Reliable Memory Acquisition on Smartphones

He Sun[1,2,3,4], Kun Sun[4], Yuewu Wang[1,2], Jiwu Jing[1,2], and Sushil Jajodia[4]

[1] Data Assurance and Communication Security Research Center, CAS
[2] State Key Laboratory of Information Security, Institute of Information Engineering, CAS
[3] University of Chinese Academy of Sciences
[4] George Mason University

Abstract. With the wide usage of smartphones in our daily life, new malware is emerging to compromise the mobile OS and steal the sensitive data from the mobile applications. Anti-malware tools should be continuously updated via static and dynamic malware analysis to detect and prevent the newest malware. Dynamic malware analysis depends on a reliable memory acquisition of the OS and the applications running on the smartphones. In this paper, we develop a TrustZone-based memory acquisition mechanism called *TrustDump* that is capable of reliably obtaining the RAM memory and CPU registers of the mobile OS even if the OS has crashed or has been compromised. The mobile OS is running in the TrustZone's normal domain, and the memory acquisition tool is running in the TrustZone's secure domain, which has the access privilege to the memory in the normal domain. Instead of using a hypervisor to ensure an isolation between the OS and the memory acquisition tool, we rely on ARM TrustZone to achieve a hardware-assisted isolation with a small trusted computing base (TCB) of about 450 lines of code. We build a TrustDump prototype on Freescale i.MX53 QSB. It can reliably acquire and transmit the kernel memory to a remote machine and calculate a hash value of the Linux kernel in 1.56ms.

Key words: TrustZone, Non-Maskable Interrupt, Memory Acquisition

1 Introduction

Smartphones have been widely used to perform both personal and business transactions and process sensitive data with various OEM or third-party mobile applications. However, due to the large code size and complexity of the mobile OS kernel, a malicious code can exploit known and unknown kernel vulnerabilities to compromise the mobile OS and steal the sensitive data from the mobile applications. It is critical to continuously perform malware analysis on the newest emerging malware and immediately update the anti-malware tools on the smartphones

There are two generic types of malware analysis methods: *in-the-box* approach and *out-of-the-box* approach. For the in-the-box approach, all the anti-malware and debugging tools are installed in the OS. This approach is efficient since it can use the OS context and call the kernel functions to study the malware's behaviors. However, it is vulnerable to malware running at the same OS level, such as rootkits that can modify the kernel functions. For the out-of-the-box approach, the malware analysis tools

are installed in an isolated execution environment from the targeted OS environment. For instance, Virtual Machine Introspection (VMI) [1–6] runs a suspicious OS in one VM, and runs the analysis tools in another VM, which can introspect the suspicious VM from outside. This method needs to know the internal structures of the OS in order to fill the semantic gaps. Recently, Yan et al. [7] extend the out-of-the-box malware analysis approach to Android smartphones using a customized QEMU emulator.

All the VMI based malware analysis solutions rely on a trusted hypervisor, which should not easily crash or be compromised. However, due to the large size of the hypervisor, it may contain a number of vulnerabilities that may be exploited by the malware to compromise the hypervisor and the malware analysis VM. VT-x/SVM [8–10] and System Management Mode (SMM) [11–14] on x86 architecture can be used to create an isolated instruction level execution environment for out-of-the-box malware analysis. Fortunately, the ARM processors, which are widely used on smartphones, also provide a system level isolation solution using a hardware security support called *TrustZone* [15, 16], which divides the mobile platform into two isolated execution environments, *normal domain* and *secure domain*. The OS running in the normal domain is usually called Rich OS, and the one running in the secure domain is called secure OS.

In this paper, we develop a TrustZone-based reliable memory acquisition mechanism called *TrustDump* that is capable of obtaining the RAM memory and CPU registers of the Rich OS even if the Rich OS has crashed or has been compromised. TrustDump does not require to install a hypervisor installed in the normal domain. The Rich OS running in the normal domain is the target for memory dump and malware analysis by a memory acquisition module called TrustDumper, which is installed in the secure domain. Since TrustDumper is self-contained, we don't need to install a full-feature OS in the secure domain. When there is only one OS running on the ARM platform, it is usually running in the secure domain. Thus, we need to first port the Rich OS to run in the normal domain and then install the TrustDumper in the secure domain. TrustZone can ensure the TrustDumper is securely isolated from the Rich OS, so that a compromised Rich OS cannot compromise the memory acquisition module.

To reliably obtain the Rich OS's memory dump, TrustDump ensures a reliable system switch from the normal domain to the secure domain even if the Rich OS has crashed. Moreover, TrustDump guarantees that a malicious Rich OS cannot launch Denial of Service (DoS) attacks to block or intercept the switching process. We use a hardware button on the smartphone to trigger a non-maskable interrupt (NMI) to the ARM processor, which then initiates the switching process. Since the secure domain has the access privilege to the memory in the normal domain, the TrustDumper can access the physical RAM memory and the CPU states of the Rich OS. On the other hand, we must prevent attackers from misusing our technique to acquire the memory and uncover sensitive data such as passwords and encryption keys. We propose to use a PIN number to authenticate the user when the system switches to the secure domain. To achieve a better understanding on what happened on a compromised smartphone, the memory acquisition module sends the memory dump to a remote machine. A hash value of the memory dump is also sent to verify a correct transmission. The remote machine can use all types of powerful memory forensics tools to uncover the malware behaviors recorded in the memory dump and CPU registers.

TrustDump can acquire the RAM memory and CPU registers of the Rich OS even if the Rich OS crashes. Those raw data contain detailed system state information, and they can be combined with the memory dump automatically generated by the OS to further the analysis. When the user notices any malicious behaviors in the Rich OS, it can reliably acquire the RAM memory of the Rich OS by pressing a physical button. A malicious Rich OS cannot block the non-maskable interrupt being sent to the ARM processor to trigger the memory acquisition module. The Rich OS is frozen after the system switches into the secure domain, so it has no chance to clean its attacking traces. TrustDump is OS agnostic. Since TrustDump does not need to make any changes to the Rich OS kernel, rooting the Rich OS is not required. Moreover, it strictly follows the smartphone forensic principle of extracting the digital evidence without losing or altering the data contents.

In summary, we make the following contributions in this paper.

- We design a hardware-assisted memory acquisition mechanism named TrustDump to reliably acquire the RAM memory and CPU registers of the OS on smartphones, even if the OS has crashed or has been compromised.
- The trusted computing base (TCB) of TrustDump is small, only consisting of the memory acquisition module in the secure domain. We don't need to install a hypervisor or root the OS.
- We implement a TrustDump prototype on Freescale i.MX53 QSB. We port the Rich OS to work in the normal domain and run the memory acquisition module in the secure domain. We construct a non-maskable interrupt (NMI) for ensuring a reliable switching with minimum impacts on the Rich OS.

The remainder of the paper is organized as follows. Section 2 introduces background knowledge. Section 3 describes the threat model and assumptions. We present the framework in Section 4. A prototype implementation is detailed in Section 5. Section 6 discusses the experimental results. We describe related works in Section 7. Finally, we conclude the paper in Section 8.

2 Background

2.1 TrustZone Overview

TrustZone [16, 15] is a system-wide approach to provide hardware-level isolation on ARM platforms. It's supported by a wide range of processors: Cortex-A8 [17], Cortex-A9 [18] and Cortex-A15 [19]. It creates two isolated execution domains: *secure domain* and *normal domain*. The secure domain has a higher access privilege than normal domain, so it can access the resource of the normal domain such as memory, CPU registers and peripherals, but not vice versa. There's an *NS* bit in the CPU processor to control and indicate the state of the CPU - 0 means the secure state and 1 means the normal state. The system bus also contains a bit to indicate the state of the bus transaction. Thus, normal peripherals can only transmit normal transactions, but not the secure transactions. There's an additional CPU mode, *monitor mode*, which only runs in the secure domain regardless of the value of the *NS* bit. The monitor mode serves as a gatekeeper between the normal and secure domains. If the normal domain requests to switch to the secure domain, the CPU must first enter the monitor mode.

2.2 TrustZone Aware Interrupt Controller (TZIC)

The TZIC is a TrustZone enabled interrupt controller, which allows complete and independent control over every interrupt connected to the controller. It receives interrupts from peripheral devices and routes them to the ARM processor. The TZIC provides secure and non-secure transaction access to those interrupts, restricting non-secure read and write transactions to only those interrupts configured as non-secure and allowing secure transactions read and write capability to all interrupts regardless of security configuration. It supports priority masking, overall enable/disable of secure or non-secure interrupts, and access to raw interrupt state and pending transactions. By default, the TZIC uses Fast Interrupt FIQ as secure interrupt and uses IRQ Interrupt as non-secure interrupt. There are three exception vector tables associated with the normal domain, the secure domain, and the monitor mode, respectively.

2.3 General Purpose Input/Output (GPIO)

The GPIO provides general-purpose pins that can be configured as either input or output. It can be connected to the physical buttons, LED lights, and other signals through an I/O multiplexer. As the signal can be either 0 or 1, each pin of GPIO contributes a bit in the GPIO block. The GPIO can trigger interrupts to the TZIC. However, if the source is masked off in the GPIO, the corresponding interrupt request cannot be sent to TZIC.

3 Threat model and Assupmtions

On an ARM TrustZone platform, when the Rich OS crashes due to system failure or malicious behaviors, the system may not enter the secure domain even after receiving a secure interrupt. A reliable memory acquisition must prevent this type of "Denial of Service" attacks. When the Rich OS has been compromised, it can intercept the switch request and fake a memory acquisition process in a "Man in the Middle" attack. It is critical to ensure that the TrustDump is really entered after triggering the NMI. The Rich OS may compromise the memory acquisition module in the secure domain to break the memory acquisition process. Therefore, we must protect the integrity of the TrustDump.

We assume the adversary has no physical access to the smartphone. It cannot perform hardware attacks, e.g., accessing the MicroSD card. The ROM code is secure and cannot be flashed. We assume the smartphone has TrustZone hardware security support. We trust the hardware isolation mechanisms of TrustZone and use it to protect the memory acquisition from the Rich OS.

4 TrustDump Framework

Figure 1 shows the TrustDump framework using ARM TrustZone hardware security support. The Rich OS running in the normal domain is the target for memory acquisition, and a self-contained software called TrustDumper in the secure domain is responsible for data acquisition, data analysis, and data transmission of the Rich OS's memory

and CPU registers. After a reliable switching from the normal domain to the secure domain, a data acquisition module is responsible for reading the RAM memory and CPU registers of the Rich OS without any support from the Rich OS. TrustDump is flexible to either perform simple analysis such as checking the integrity of the OS kernel or run complicated malware analysis tools after filling the semantics gap. Then, the acquired memory and CPU registers can be transmitted to a remote computer.

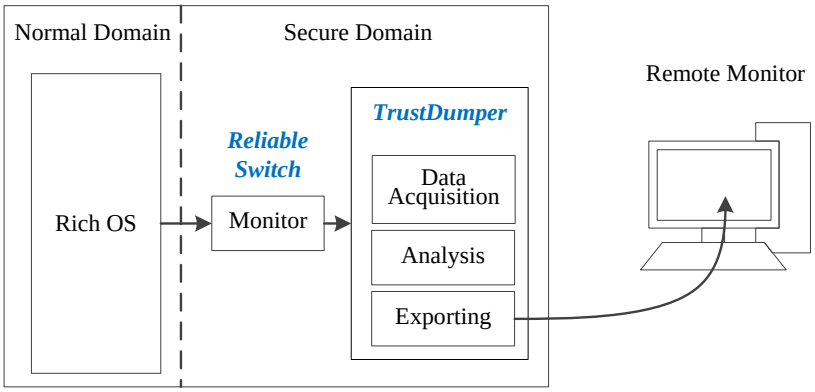

Fig. 1. The System Framework

4.1 TrustDumper Deployment

When there is only one OS running on the ARM platform, it is usually running in the secure domain. In our system, since the Rich OS is running in the normal domain, we need to port the Rich OS to the normal domain and then install the TrustDumper in the secure domain. The work of porting Rich OS to the normal domain sounds simple, but the source code customized to run in the secure domain cannot be executed in the normal domain without modification. Since there's no open source OS available for running in the normal domain on real platform, we have to port Android OS from the secure domain to the normal domain by ourselves. We allocate a sealed memory region for the secure domain to run the TrustDumper. TrustZone guarantees that the normal domain cannot access the sealed memory. Since TrustDumper is self-contained, we don't need to install a full-featured OS in the secure domain. It dramatically reduces the complexity of system implementation.

4.2 Reliable Switching

A reliable switching into the secure domain is the prerequisite for memory acquisition. We must ensure the system will be switched from the normal domain to the secure domain per the user's requests even if the Rich OS is compromised or simply crashes.

First, the system can be safely switched into the secure domain when the Rich OS crashes. In other words, we cannot rely on the Rich OS to initiate the switching process even if the Rich OS is secure and trusted. Second, our system should prevent a malicious Rich OS from launching Denial of Service attacks to block or intercept the switching request.

A non-maskable interrupt (NMI) can be used to achieve a reliable switching, since the Rich OS cannot block or intercept the NMI. NMI has been widely used and deployed on mobile platforms [20, 21], and we can configure one NMI triggered by pressing a button or a combination of several buttons for switching the system into the secure domain. For mobile platforms that do not have dedicated NMI but support TrustZone (e.g., Freescale i.MX53 QSB [22]), we can use one secure interrupt as the NMI.

TrustZone provides two ways to enter the secure domain from the normal domain: *SMC* instruction and *Secure Interrupt*. SMC instruction is a privileged instruction that can only be invoked in the Rich OS's kernel mode. However, when the Rich OS is malicious, it can block or intercept the secure monitor call that uses the SMC instruction. Moreover, when the Rich OS crashes, SMC instruction may not be called before the crash happens. Secure interrupt of TrustZone can be called to switch from the normal domain to the secure domain. TrustZone uses the fast interrupt FIQ as the secure interrupt and uses the normal IRQ interrupt as the normal interrupt. Since we don't need to use all the FIQ in the secure domain, we can reserve one FIQ as the NMI to enforce a reliable domain switching.

4.3 Data Acquisition and Transmission

The software in the secure domain has access privilege to entire physical memory of the normal domain. Moreover, it can access all the banked CPU registers, which are critical to fill the semantic gaps for malware analysis. When the system enters the secure domain, the Rich OS in the normal domain is frozen.

Our system supports both online malware detection and offline malware analysis. For online malware detection, since the analysis module runs out of the Rich OS, it has to fill the semantic gaps. Based on the knowledge of the kernel data structures, the analysis module first reconstructs the context of the Rich OS and then runs malware analysis tools in the secure domain. It can verify the integrity of the Rich OS and detect rootkits. For offline analysis, since we need to transmit a large amount of acquired RAM memory (e.g., 1GB in Freescale i.MX53 QSB) to a remote computer, DMA is used to transfer data from RAM memory to communication peripherals such as serial port or network card. A hash value of the acquired memory is calculated and transmitted to verify the data transmission process. When the DMA and the peripherals are being used by the Rich OS when the switching happens, their states should be saved before the memory dumping and restored afterward.

4.4 System Security

With the NMI triggered by a physical button, TrustDump can safely switch the system from the normal domain to the secure domain no matter what state the Rich OS is at. Thus, a malicious Rich OS cannot launch Denial of Service attacks to block or intercept

the switching. When the system switches into the secure domain, the Rich OS is frozen, so the malware in the Rich OS cannot clean its traces.

The TrustDumper has the privilege to access all the memory and CPU registers of the Rich OS, so it can check the integrity of the Rich OS and detect various malware such as rootkits in the Rich OS. Since the TrustDumper in the secure domain is securely isolated from the Rich OS by TrustZone, a compromised Rich OS cannot compromise the memory acquisition modules. To prevent a malicious user from misusing our mechanism, we use a PIN number to authenticate the user when the system is triggered by the specific NMI to switch to the secure domain.

5 Implementation

We implement a prototype using Freescale's *i.MX53 QSB*, a TrustZone-enabled mobile System on Chip (SoC) [22]. i.MX53 QSB has an ARM Cortex-A8 1 GHz application processor with 1 GB DDR3 RAM memory and a 4GB MicroSD card. We deploy Android 2.3.4 in the normal domain. The development board is connected through a serial port to a Thinkpad-T430 laptop that runs Ubuntu 12.04 LTS. The TrustDump prototype can securely acquire the Android kernel memory and CPU registers, check the Android kernel integrity and detect potential rootkits that modify the kernel data or structures, and transmit the acquired kernel memory to the laptop. The TCB of TrustDump contains only 456 lines of code.

5.1 Deployment of TrustDump

Since we cannot find any open source OS working in the normal domain of a hardware platform, we have to port Android OS from secure domain to normal domain based on the Board Support Package (BSP) published by Adeneo Embedded [23]. We deploy the TrustDumper in the secure domain to isolate it from the Rich OS.

Because the normal domain has a lower privilege than the secure domain, the source code running in the secure domain may not execute in the normal domain without proper modification. There are some peripherals that cannot be configured to be accessed from normal domain. For example, the Deep Sleep Mode Interrupt Holdoff Register (DSMINT) can only be accessed in the secure domain. However, the Rich OS needs DSMINT to hold off the interrupts before entering low power mode. To make Android run in the normal domain, we develop a pair of secure I/O functions, *secure_write* and *secure_read*, to access the peripherals.

The two functions are called in the normal domain to request the secure domain to help access the peripherals. Each peripheral on the i.MX53 QSB is controlled by the registers in it. Each register is expressed as an address in the memory map of the board. Both functions give access privilege escalation to the normal domain on certain registers. The function definition is shown in Listing 1. The target address is physical address and the data is 32 bits. *secure_write* is to write parameter `data` to physical address `pa`. *secure_read* is to read `data` from the physical address `pa` and return the result. There's an SMC instruction in these two functions to deliver the request to secure domain. The *Whitelist* stores all the registers to which the normal domain is allowed

to access with access privilege escalation. Upon receiving a request, the secure domain checks if the address requested is in the *Whitelist*. If yes, the secure domain helps access the address per the request. If not, it directly returns without doing anything.

Listing 1. Definition of *secure_write* and *secure_read*

```
void secure_write(unsigned int data, unsigned int pa);
unsigned int secure_read(unsigned int pa);
```

5.2 Reliable Switching

To ensure a reliable switching from the normal domain to the secure domain, we reserve a secure interrupt (FIQ) of TrustZone to serve as non-maskable interrupt (NMI). Figure 2 shows the switching process, which involves three components, namely, peripheral device, TZIC, and the ARM processor. The switching contains four steps. First, a peripheral device as the source of interrupt makes the interrupt request. Second, the interrupt request will be sent to TZIC. Third, based on the type of the interrupt (FIQ or IRQ), TZIC asserts the corresponding exception to the ARM processor. To trigger a reliable switching, the interrupt request must be an FIQ. Finally, after receiving an FIQ, the ARM processor switches to secure domain according to the setting of the Secure Configuration Register (SCR) and Current Program Status Register (CPSR).

Note all the three components are critical to a reliable switching. The compromise of any of the three components will result in the compromise of the reliable switching. If the source of the interrupt can be managed and then masked by the Rich OS, or Rich OS just blocks all the FIQs to the ARM processor, then the switch to the secure domain will be blocked. We construct an NMI using GPIO-2. An interrupt can be configured as either secure or non-secure in TZIC. The non-secure transaction cannot access the secure interrupt. To make an NMI, the GPIO-2 interrupt should be set as secure interrupt first. Then we use the peripheral access privilege control in Central Security Unit (CSU) to isolate the peripheral from the normal domain. It guarantees the normal domain cannot configure the peripheral. At last, by configuring the registers of ARM processor, we make the FIQ request to be handled in the secure domain.

To minimize the impacts on the access of the Rich OS to other peripherals sharing the same access privilege with GPIO-2, we propose a method to enable *Fine-grained Access Control*. To minimize the impacts on the functionalities of other peripherals, we propose a method to enable *Fine-grained Interrupt Control*. It differentiates the interrupts sharing the same interrupt number and distributes them to different handlers in different security domains.

Non-maskable GPIO-2 Secure Interrupt In our prototype, we use the user-defined button 1 on the board to trigger reliable switching to the secure domain. There are seven GPIOs from GPIO-1 to GPIO-7 on our board. The user-defined button 1 is connected to the second GPIO: GPIO-2. The interrupt number of GPIO-2 is 52. The button is a binary signal and is attached to the fifteenth bit (Data[14]) of GPIO-2.

The interrupt type is set in Interrupt Security Registers (TZIC_INTSEC). Each interrupt corresponds to a bit in the register. We set the bit to 0 in TZIC_INTSEC to mark the

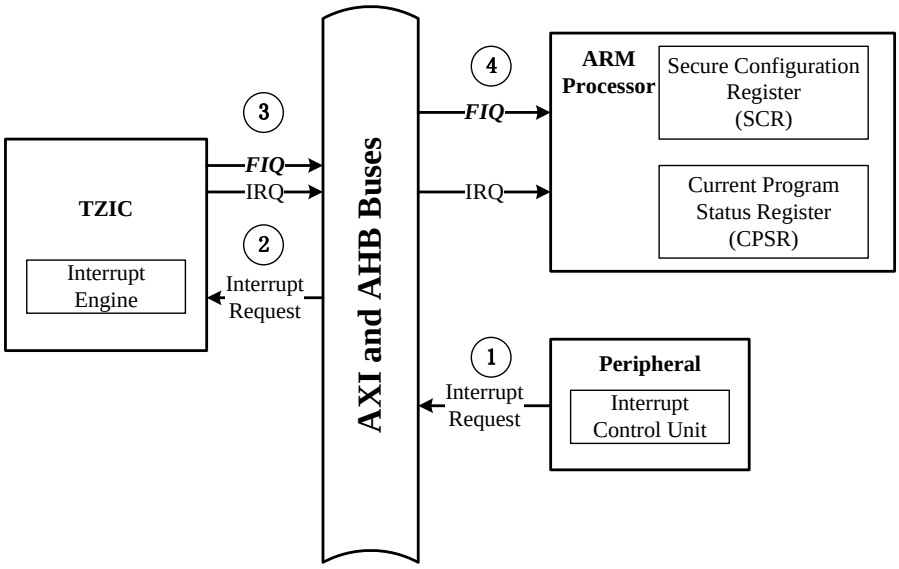

Fig. 2. The Control Flow of Reliable Switching

GPIO-2 interrupt secure. Because many of the security features can only be accessed in the secure domain, we create the NMI before launching the normal domain. First, we configure the interrupt of GPIO-2 as secure in TZIC_INTSEC. This prevents the normal domain from accessing the GPIO-2 interrupt configuration in TZIC. Second, we set the *FIQ* bit in SCR to 1 to ensure the FIQ exception will be branched to the monitor mode. We also set the *FW* bit in SCR to 0 to ensure the FIQ enable (*F*) bit in CPSR won't be modified by normal domain. Then we set the *F* bit in CPSR to 0 to enable FIQ exception before switching to the normal domain. This step ensures that the FIQ exception cannot be blocked to ARM processor by codes running in the normal domain. At last, we disable the non-secure access to GPIO-2 in CSU. It isolates GPIO-2 from normal domain. Thus the interrupt unit of GPIO-2 cannot be configured again by codes running in the normal domain.

As soon as the secure interrupt happens, TZIC generates FIQ request to ARM processor and the ARM processor is routed to the secure domain. The entry of the secure interrupt is in the vector table of monitor mode at the offset of `0x1C`. Upon receiving the interrupt, CPU changes to the monitor mode and jumps to the entry automatically. When the memory acquisition finishes, the CPU executes the instruction: `subs pc, lr, #4` to return to the Rich OS.

Fine-grained Access Control We isolate GPIO-2 from the normal domain in CSU to prevent the normal domain from configuring its interrupt control unit. TrustZone creates two isolated domains with different privileges. Both privileges have their own access control policy to the peripherals. The access privilege of peripherals is managed

by CSU, which sets access control policies between bus masters and bus slaves in bus transaction on i.MX53 QSB. It sets peripheral access policy to determine the master privileges required to access each peripheral. The non-secure master cannot access the secure peripherals. There are registers whose bits correspond to the peripherals. Setting access control policies is done by setting the corresponding bits in registers. It is worth mentioning that the CPU is one of the masters and it generates its own security signal through *NS* bit. So when *NS=0*, CPU can access all the peripherals; while when *NS=1*, CPU can only access the non-secure peripherals.

However, this access control management is too rigid and provides limited discrimination among peripherals. Some peripherals share a same access control policy, their access control is binding. For example, user-defined button 1 only corresponds to one bit of the data of GPIO-2. Other bits share the same access policy with the button and may be connected to non-secure peripherals. If we deny the non-secure access to user-defined button 1, the non-secure access to other bits of GPIO-2 will be denied all together. It harms the usability of GPIO-2.

We use fine-grained access control to solve the problem. We set the configuration-binding peripherals to be only accessible by the secure domain but disclose those peripherals to the normal domain by adding them to a *Whitelist*. In the Rich OS, every time we want to access the disclosed peripherals, we use the secure I/O functions described above. In this way we protect the *NMI* while not influencing the access of the normal domain to other devices.

Fine-grained Interrupt Control If the other bits of GPIO-2 don't use interrupt to operate, fine-grained access control would work for the Rich OS. However, when it comes to interrupts, it is not the case. There's only one interrupt number for all the 32 bits of GPIO-2. Each bit can generate the No.52 interrupt. The 15th bit of GPIO-2 is dedicated to NMI and its handler is in the secure domain while the handlers of other bits' interrupts may be in the Rich OS. For example, in our prototype user-defined button 2 is connected to Data[15] of GPIO-2. And it's for the home key of Android. The Rich OS returns to the home screen of Android when the button is pressed. These two user-defined buttons generate the same interrupt in TZIC. Therefore, after the construction of NMI, button 2 now generates FIQ request together with button 1. When button 2 is pressed, CPU goes into secure domain first. To better control the interrupt and maintain the functionality of button 2, we provide an approach to support fine-grained interrupt control.

When CPU goes into the secure domain after pressing button 2, we forward the FIQ request generated by button 2 back to the Rich OS. The entry of IRQ exception of the normal domain has already been used for interrupt handling by the Rich OS. Therefore, we leverage the entry of FIQ exception of the normal domain, which is left unused by the Rich OS, to handle the forwarded request. As the operation code of button 2 is already in Rich OS, we reuse it with the interrupt number 52.

The program flow of hardware interrupts on the board is depicted in Figure 3. The interrupts configured as non-secure in TZIC will assert IRQ request. The IRQ exception are handled in the Rich OS. The IRQ exception handler gets the number of pending in-

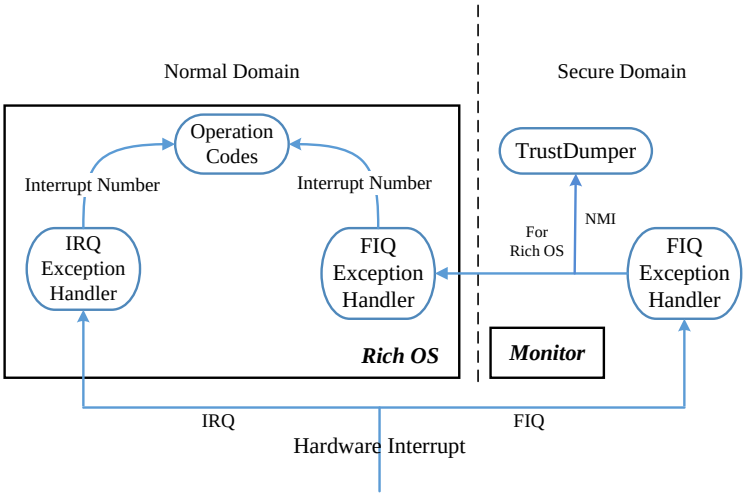

Fig. 3. Program Flow of Interrupt

terrupt from TZIC and gives it to the operation codes, which will perform corresponding action.

The interrupt configured as secure in TZIC asserts FIQ request. Upon secure interrupt assertion, system switches to the FIQ exception entry of secure domain unconditionally. The FIQ exception handler of secure domain gets the interrupt number from TZIC and knows the source bit of interrupt through interrupt control unit of GPIO-2. If the interrupt is dedicated to NMI, the handler clears the interrupt status in TZIC to prevent re-entry. Next, it starts to do the memory acquisition and analysis. After that, the system returns to the normal domain for Rich OS to gain control.

When the source of the interrupt is not for NMI, the handler masks the interrupt bit of the source in GPIO-2. This automatically stops the interrupt request to TZIC and thus clears the interrupt status in TZIC. Masking, not clearing the interrupt in the handler prevents re-entry after entering Rich OS and keeps the interrupt status in the interrupt control unit of GPIO-2, which is used to distinguish the source bit of GPIO-2 by Rich OS. Hence, we just need to transfer the interrupt number to Rich OS because it will access the interrupt control unit of GPIO-2 to determine which bit generates the interrupt. After that, change CPU mode to FIQ mode and jump to the entry of FIQ exception in normal domain. Because secure domain will not be re-entered, before jumping to FIQ handler of Rich OS the context of the Rich OS at the point when the interrupt happened should be restored. In this way, the program flow acts as if the FIQ request is routed to the Rich OS without the interference of secure domain.

The functions of Rich OS cannot be called without the knowledge of the address of the function. The functions cannot be simply called by using the symbol of Rich OS in secure domain as Rich OS is compiled separately from the software in secure domain. It needs an intermediary to receive the request from secure domain and call

the operation codes. We leverage the FIQ exception handler of Rich OS to accomplish this goal. The entry of FIQ exception is addressed at `0xFFFF01C`. It's easy for secure domain to jump to because it's fixed. Because the handler runs in FIQ mode and the mode is not used by Rich OS, we can make it a bridge between secure domain and Rich OS without influencing the operation of Rich OS. We add initialization code of FIQ mode and the handler of FIQ exception to Rich OS. Rich OS sets up a 16KB-stack for FIQ mode when the system boots up and then the FIQ mode is ready for use. After getting the interrupt number from TZIC, the FIQ exception handler saves the context of Rich OS. As the operation codes of interrupts run in SVC mode, the FIQ exception handler saves its own context and changes mode to Supervisor. Then it calls operation codes in Rich OS with the interrupt number. After the codes finish running, program returns to the FIQ exception handler of Rich OS. The handler then recovers the context at the point when the interrupt happened and Rich OS wakes up.

After getting the interrupt number, the interrupt operation codes find the corresponding action code. In our prototype, it's the function `mx3_gpio_irq_handler`. The function first checks which bit of GPIO generates the interrupt. As we mask the source bit, the function will ignore the interrupt and return directly without doing anything. So we let the function bypass the mask status judgment by `or`-ing the corresponding bit with `1` in the mask status judgment code. We don't need to bypass the interrupt status judgment as the interrupt status bit keeps high as long as the interrupt condition meets. With the mask status judgment passed, the action of the user-defined button 2 is taken. Of course, the action of user-defined button 1 is lost as the button is used for NMI.

5.3 TrustDumper

TrustDumper can get the information of CPU state and physical memory of the Rich OS, perform some analysis, and then send the acquired data to a remote machine.

Data Acquisition and Transmission Some of the CPU state information is stored in banked registers: one copy for normal domain and the other copy for secure domain. When running, each domain uses its own copy of the registers. In monitor mode, CPU runs in secure domain using the secure copy of registers but can access the non-secure copies of the registers by setting the *NS* bit to 1.

Secure domain can access the physical memory of normal domain directly. Hence, TrustDump can get the data of Rich OS with physical address mapping. To get the data of the virtual address, TrustDump has a translation block to walk the page tables of Rich OS and get the mapping information. The physical base address of page table is in Translation Table Base Register (TTBR). With access to TTBR of normal domain and according to the two-level virtual memory system of Rich OS, TrustDump walks the translation table to translate virtual address of normal domain to physical address.

Memory dumping involves transferring data of RAM to the peripherals. Both CPU and DMA can do the work. Because data transferring is time-consuming and DMA goes faster when it comes to large scale of data, we take advantage of the DMA on the board. Using DMA, CPU can continue working while the memory is being dumped. DMA has its own processing core and memory. It stores the scripts used by Rich OS

in its internal RAM. Before transferring, TrustDump stores the state of the core and the scripts in internal RAM. It exports these data to an unused area of OCRAM on the board. Then it downloads the dumping execution program and the corresponding context to the internal RAM. After that, TrustDump triggers the DMA and starts to dump memory to the peripherals. CPU can do other work at the time of transferring. After the transfer is done, an interrupt is generated. Then TrustDump restores the core state and internal memory in DMA internal RAM from OCRAM. Note only the area of internal RAM to be used by the analysis is saved to OCRAM. This reduces the saving time. In our prototype, we use the serial port for memory dumping and the output of serial is connected to a laptop. Other peripherals can also be used, e.g., the network card and wireless module.

Integrity Checking and Rootkit Detection In our prototype, the analysis module is capable of checking the integrity of kernel code and detecting some rootkits. We use SHA-1 algorithm of SAHARA to check the integrity of Android kernel. SAHARA Security Accelerator (SAHARA) of i.MX53 QSB is a security co-processor which implements block encryption algorithms (AES, DES, and 3DES), hashing algorithms (MD5, SHA-1, SHA-224, and SHA-256), a stream cipher algorithm (ARC4), and a hardware random number generator. The static kernel code is continuous in physical address space. There's an offset from the physical address to the virtual address. In our prototype the offset is `0x10000000` and the start virtual address of kernel is `0x80004000` in our prototype. The start physical address is `0x70004000`. The length of the kernel depends on the code of kernel and varies from different versions of kernel. Yet after the kernel has been compiled, the length is certain. Given the start address and the length, the hash of kernel can be calculated in analysis module.

We implement SHA-1 hash in two ways. One is in software way and the other is to leverage the hardware accelerator of i.MX53 QSB. Because not every platform has a hardware accelerator, we provide software-implemented hash. The source of SHA-1 algorithm is from open source project PolarSSL [24]. We re-implement the memory operation and output functions to accommodate the code to the bare-metal environment of secure domain. The performance of hardware hash is better than software hash. Therefore, we use the hardware accelerator in our prototype.

Our prototype can detect rootkits that hide processes. Figure 4 illustrates the list of process in linux kernel 2.6.35. A process is represented by a struct: `task_struct`. It includes the process number (`pid`) and the memory descriptor of the process (`mm`). The `task_struct` of the current running process is in the struct: `thread_info`. The address of the `thread_info` can be located at (`stack pointer &(0x1FFF)`). All the processes are linked by struct:`list_head`. These `list_head`s construct a doubly linked list and each `list_head` stores the address of the `list_head` of the previous and next task.

Because the members of data structure struct are continuous in virtual memory space, the pointer to the struct data structure plus the offset of the member in the struct is the address of the member. The address of `task_struct` is the address of `thread_info` plus the offset $0xC$. Therefore, starting from the `thread_info` of current thread, all the information of processes are listed through the doubly linked list.

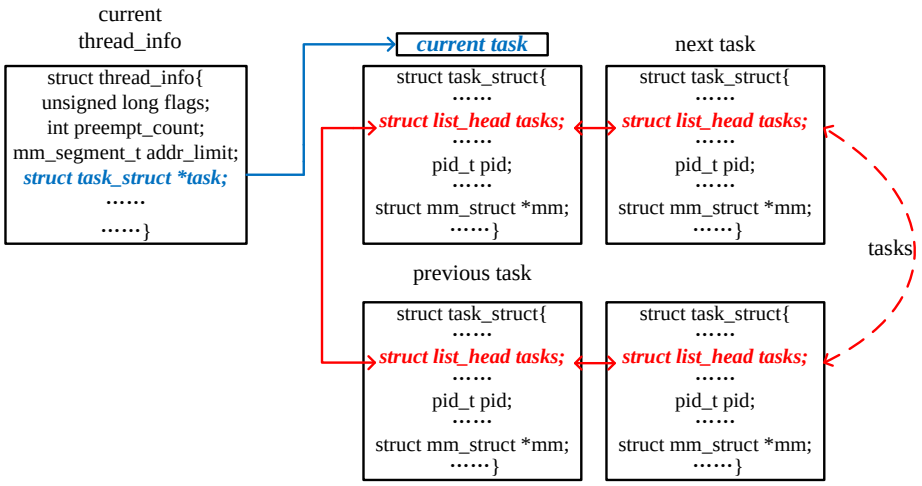

Fig. 4. Process List

6 Performance Evaluation

We evaluate the performance of TrustDump in three aspects: NMI switching performance, analysis performance, and memory dumping performance. We use the performance monitor in Cortex-A8 to count the CPU cycles and then convert the cycle to time by multiplying 1 ns per cycle. We conduct each experiment 50 times and report the average.

6.1 NMI Switching performance

We measure the time of entering TrustDump with *NMI* and SMC instruction respectively for comparison. It's difficult to measure the time from triggering the interrupt to handling it because cycle counter cannot be started at the exact time when the button is pressed. To get the performance of *NMI*, we use the Software Interrupt Trigger Register (TZIC_SWINT) in TZIC to assert the *NMI* in a software way. Writing the interrupt number of the NMI triggers the interrupt to assert. We measure the time from writing the register to receiving the request in TrustDump. The average time of entering TrustDump using *NMI* is 1.7 us. We measure the SMC instruction performance by calculating the time from invoking SMC instruction to receiving the request in TrustDump. The average time using SMC instruction is 0.3 us. It's shorter than the time of *NMI* because it takes time to transfer the interrupt request from the peripheral to TZIC and then to CPU. However, the time of *NMI* in whole is small and imperceptible by user. Moreover, *NMI* is more reliable and more flexible than the SMC instruction.

6.2 Analysis Performance

We deploy rootkit Suterusu [25] for evaluation. Suterusu performs system call inline hooking on arm platform to hide specific processes. Each time you use the *ls* and *top* command in the terminal, suterusu hooks the function and deletes the hidden process information from the result. We can detect the rootkit by traversing all the processes. To get the pointer of current process's *thread_info*, we need to obtain the stack pointer of user mode in monitor mode. In the monitor mode, we change CPU mode to system mode by directly modifying Current Program Status Register (CPSR) and after the mode change save the *r13* register to one of $\{$*r0*,*r1*,...,*r12*$\}$. Then we modify the CPSR to return to monitor mode. The system mode has the same stack pointer as user mode and can go into monitor mode by modifying the CPSR directly. With the stack pointer, we can traverse all the processes along the process list. When comparing the result with what we get using command *ls* or *top*, we can detect hidden processes by a rootkit.

The time to detect Suterusu is 2.13 ms; however, this time depends on the number of processes, and in our prototype there are 75 processes. The time to calculate kernel hash is 1.56 ms by hardware, and 578.6 ms by software. Those times vary with length of the kernel, and our kernel length is `9080836` Bytes. Both the time is negligible and TrustDump can be invoked frequently while not influencing the performance of Rich OS in the normal domain. However, calculating hash through software is much slower than the hardware and the user may experience the Rich OS hang.

Table 1. Memory Dumping Performance

Scale($Byte$)	Bit Rate(bit/s)	
	SDMA	CPU
10	92178.12	92178.49
100	92163.38	92165.45
1K	92163.01	92163.43
10K	92163.09	92163.11

6.3 Memory Dumping Performance

Both DMA and CPU can dump memory to peripherals. We choose DMA in TrustDump to free the burden of dumping memory from CPU. Our result shows that the memory dumping performance of DMA isn't worse than CPU. To compare the performance between DMA and CPU, we pick four scales of memory size: 10B, 100B, 1KB and 10KB. In each scale, we conduct the experiments 50 times for DMA and CPU respectively. We take the average value and divide the result with the scale to get the bit rate. The bit rate at these four scales is listed in Table 1. We can see that DMA performs as fast as CPU. It take 13.14 minutes in average to dump Android Kernel of `9080836` Bytes to a laptop through serial port. The bottleneck of the speed is the limited baud rate of serial port.

The performance of memory dumping can be improved by using other peripherals, e.g., the Ethernet and wireless device. It is included in our future work.

7 Related Work

Memory acquisition techniques are through software or hardware in nature. A software memory acquisition is typically achieved by accessing virtual memory system and the file system inside Rich OS, or by monitoring the physical address space of Rich OS in a virtual machine which runs Rich OS. Hardware techniques target at the physical storage medium directly with the help of dedicated hardware [26].

JTAG [27] and chip-off technique [28] can be used to achieve memory acquisition; However, memory acquisition using JTAG is only possible when a JTAG debug port is identified on the embedded device. What's more, many operating system denies the debugging request of JTAG to protect its own security. The cost of the equipment and the destructive nature of chip-off technique keeps it away from being used.

The hardware way is more efficient and reliable. However, introducing a dedicated hardware on the platform incurs more cost and not all platforms pre-install the dedicated hardware. The software way getting memory through Rich OS is more practical and popular. However if Rich OS compromises the data acquired isn't trustworthy. The out-of-the-box approach of virtual machine [29, 5, 30–32] can keep the acquisition module from being tampered by Rich OS. As we all know, adding a hypervisor under Rich OS reduces the efficiency of the whole system. Besides, the TCB of hypervisor is large. Hardware-assisted approach is also used for out-of-the-box memory acquisition [33, 34, 13]. The out-of-the-box approach has two weak points: (1) The semantic gap between Rich OS and acquisition module; (2) The entrance to the acquisition module. A hypercall for virtual machine or a CPU assembly instruction [8, 9, 14] to enter the acquisition module can be intercepted by Rich OS. Hypersentry [12] can provide not only reliable entrance but also stealthy invocation. However, IPMI used in Hypersentry is not available on ARM platform.

On smartphones, the Android Recovery Mode can give the user root privilege and bypass the passcodes to acquire the memory of Rich OS, but it needs to reboot the device before acquisition. Live memory acquisition in Android also draws much attention. Linux Memory Extractor (LiME) module is introduced into Android kernel to implement live memory acquisition [35]. Based on LiME, another work called DMD [36] is carried out to acquire the volatile memory of Android. Gianluigi Me et.al [37] presented a removable memory card based solution to overcome the heterogeneity of the tools adopted to retrieve smartphone contents. Besides the above solutions, DDMS [38] provided by Android SDK can also be used to get memory information. Because all the work above rely on the Rich OS to operate, they cannot defend against a compromised kernel and then guarantee the reliability of the memory acquisition.

8 Conclusions

Based on ARM TrustZone technology, we propose a reliable memory acquisition mechanism named TrustDump on Smartphone to perform forensic analysis and facilitate

malware analysis. TrustDump installs an Android OS in the normal domain and the memory acquisition module in the secure domain, and it relies on TrustZone to ensure a hardware-assisted isolation. TrustDump ensures the reliability of the memory acquisition with a non-maskable interrupt, which prevents user's request from being intercepted or blocked by a malicious Rich OS. We propose fine-grained access control and fine-grained interrupt control techniques to minimize the impacts on the Rich OS. Our prototype on i.MX53 QSB can enter TrustDump and begin memory dumping in 1.7 us and calculate the hash value of kernel in 1.56 ms.

References

1. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: NDSS. (2003)
2. Jiang, X., Wang, X., Xu, D.: Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In: ACM Conference on Computer and Communications Security. (2007) 128–138
3. Fu, Y., Lin, Z.: Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In: IEEE Symposium on Security and Privacy. (2012) 586–600
4. Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J.T., Lee, W.: Virtuoso: Narrowing the semantic gap in virtual machine introspection. In: IEEE Symposium on Security and Privacy. (2011) 297–312
5. Dinaburg, A., Royal, P., Sharif, M.I., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: ACM Conference on Computer and Communications Security. (2008) 51–62
6. Deng, Z., Zhang, X., Xu, D.: Spider: stealthy binary program instrumentation and debugging via hardware virtualization. In: ACSAC. (2013) 289–298
7. Yan, L.K., Yin, H.: Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In: Proceedings of the 21st USENIX Conference on Security Symposium. Security'12, USENIX Association (2012) 29–29
8. McCune, J.M., Parno, B., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: an execution infrastructure for tcb minimization. In: EuroSys. (2008) 315–328
9. McCune, J.M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V.D., Perrig, A.: Trustvisor: Efficient tcb reduction and attestation. In: IEEE Symposium on Security and Privacy. (2010) 143–158
10. Martignoni, L., Poosankam, P., Zaharia, M., Han, J., McCamant, S., Song, D., Paxson, V., Perrig, A., Shenker, S., Stoica, I.: Cloud terminal: secure access to sensitive applications from untrusted systems. In: Proceedings of the 2012 USENIX conference on Annual Technical Conference, USENIX Association (2012) 14–14
11. Zhang, F., Leach, K., Sun, K., Stavrou, A.: Spectre: A dependable introspection framework via system management mode. In: DSN. (2013) 1–12
12. Azab, A.M., Ning, P., Wang, Z., Jiang, X., Zhang, X., Skalsky, N.C.: Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In: ACM Conference on Computer and Communications Security. (2010) 38–49
13. Wang, J., Stavrou, A., Ghosh, A.K.: Hypercheck: A hardware-assisted integrity monitor. In: RAID. (2010) 158–177
14. Azab, A.M., Ning, P., Zhang, X.: Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In: ACM Conference on Computer and Communications Security. (2011) 375–388

15. ARM: TrustZone Introduction. http://www.arm.com/products/processors/technologies/trustzone/index.php
16. Alves, T., Felton, D.: Trustzone: Integrated hardware and software security. ARM white paper **3**(4) (2004)
17. ARM: Cortex-A8 Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/DDI0344K_cortex_a8_r3p2_trm.pdf
18. ARM: Cortex-A9 Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388f/DDI0388F_cortex_a9_r2p2_trm.pdf
19. ARM: ARM Cortex-A15 MPCore Processor Technical Reference Manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0438i/index.html
20. ARM: Interrupt Behavior of Cortex-M1. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0211a/index.html
21. ARM: Cortex-M4 Devices Generic User Guide. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0553a/Cihfaaha.html
22. Freescale: Imx53qsb: i.mx53 quick start board. http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=IMX53QSB&tid=vanIMXQUICKSTART
23. Adeneo Embedded: Reference BSPs for Freescale i.MX53 Quick Start Board. http://www.adeneo-embedded.com/en/Products/Board-Support-Packages/Freescale-i.MX53-QSB
24. Paul Bakker: PolarSSL. https://polarssl.org/
25. Michael Coppola: Suterusu Rootkit: Inline Kernel Function Hooking on x86 and ARM. http://poppopret.org/2013/01/07/suterusu-rootkit-inline-kernel-function-hooking-on-x86-and-arm/
26. Carrier, B.D., Grand, J.: A hardware-based memory acquisition procedure for digital investigations. Digital Investigation **1**(1) (2004) 50–60
27. Breeuwsma, I.M.F.: Forensic Imaging of Embedded Systems Using JTAG (Boundary-scan). Digit. Investig. **3**(1) (March 2006)
28. Jovanovic, Z., Redd, I.D.D.: Android forensics techniques. International Academy of Design and Technology (2012)
29. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: NDSS. (2003)
30. Litty, L., Lagar-Cavilla, H.A., Lie, D.: Hypervisor support for identifying covertly executing binaries. In: USENIX Security Symposium. (2008) 243–258
31. Hofmann, O.S., Dunn, A.M., Kim, S., Roy, I., Witchel, E.: Ensuring operating system kernel integrity with osck. In: ASPLOS. (2011) 279–290
32. Seshadri, A., Luk, M., Qu, N., Perrig, A.: Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In: SOSP. (2007) 335–350
33. Liu, Z., Lee, J.H., Zeng, J., Wen, Y., Lin, Z., Shi, W.: Cpu transparent protection of os kernel and hypervisor integrity with programmable dram. In: ISCA. (2013) 392–403
34. Jr., N.L.P., Fraser, T., Molina, J., Arbaugh, W.A.: Copilot - a coprocessor-based kernel runtime integrity monitor. In: USENIX Security Symposium. (2004) 179–194
35. Heriyanto, A.P.: Procedures and tools for acquisition and analysis of volatile memory on android smartphones. In: Proceedings of The 11th Australian Digital Forensics Conference, SRI Security Research Institute, Edith Cowan University, Perth, Western Australia (2013)
36. Sylve, J., Case, A., Marziale, L., III, G.G.R.: Acquisition and analysis of volatile memory from android devices. Digital Investigation **8**(3-4) (2012) 175–184

37. Me, G., Rossi, M.: Internal forensic acquisition for mobile equipments. In: IPDPS. (2008) 1–7
38. Google: Using ddms for debugging. http://developer.android.com/tools/debugging/ddms.html