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] State Key Laboratory of Information Security, Institute of Information Engineering, CAS, Beijing, P.R. China
[2] Data Assurance and Communication Security Research Center, CAS, Beijing, P.R. China
[3] University of Chinese Academy of Sciences, Beijing, P.R. China
[4] George Mason University, Fairfax, VA, USA

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.

Keywords: 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 sensitive data from the system. It is critical to perform malware analysis on the newest emerging malware and immediately update anti-malware tools on the smartphones

There are two generic types of dynamic 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 same OS as the malware. This approach is efficient since it can use abundant OS context information and directly call the kernel functions to study malware's behaviors. However, it is vulnerable to armored malware such as rootkits that modify kernel structures and functions to defeat the analysis. For

[⋆] Corresponding author.

M. Kutyłowski and J. Vaidya (Eds.): ESORICS 2014, Part I, LNCS 8712, pp. 202–218, 2014.

the out-of-the-box approach, the malware analysis tools are installed in an isolated execution environment, which is securely separated from the targeted OS environment. For instance, Virtual Machine Introspection (VMI) [1–6] runs a suspicious OS in one VM and the analysis tools in another VM. This method needs to reconstruct the internal structures of OS kernel 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 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 bugs and vulnerabilities that may be exploited by malware to compromise the hypervisor and then 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; however, they are not available on mobile processors. Fortunately, the ARM processors, which have been widely used on smartphones, provide a system level isolation solution with 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*, which 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. A memory acquisition module called *TrustDumper* is installed in the secure domain to perform memory dump and malware analysis of the Rich OS. TrustZone can ensure the TrustDumper is securely isolated from the Rich OS, so that a compromised Rich OS cannot compromise the memory acquisition module.

When the Rich OS has crashed or some suspicious behaviors have been detected in the Rich OS, TrustDump ensures a reliable system switch from the normal domain to the secure domain by pressing a hardware button on the smartphone to trigger a non-maskable interrupt (NMI) to the ARM processor. The NMI guarantees that a malicious Rich OS cannot launch attacks to block or intercept the switching process. Since the secure domain has the access privilege to the memory and registers in the normal domain, TrustDumper can freely access the physical RAM memory and the CPU states of the Rich OS. When the system switches into the secure domain, the Rich OS is frozen, so the malware has no time to clean its attacking traces. Besides checking the OS kernel integrity and perform online malware analysis, TrustDumper can send the memory dump and CPU states to a remote machine for further analysis. A hash value of the memory dump is also calculated and sent to verify a correct data transmission. The remote machine can use various powerful memory forensics tools to uncover the malicious behaviors recorded in the memory dump.

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. Since TrustDumper is self-contained, a full-featured OS is not required to be installed in the secure domain. Moreover, TrustDump is OS agnostic and we do not need any changes to the

Rich OS, which satisfies the smartphone forensic principle of extracting the digital evidence without altering the data contents. We build a TrustDump prototype on Freescale i.MX53 QSB.

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 a small memory acquisition module in the secure domain. We do not need to install a hypervisor or root the OS in the normal domain.
- We implement a TrustDump prototype on Freescale i.MX53 QSB. A non-maskable interrupt (NMI) is constructed for ensuring a reliable switching from the normal domain to the secure domain in 1.7 us.

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 and conclude the paper in Section 8.

2 Background

2.1 TrustZone Overview

TrustZone [15, 16] is a system-wide approach to provide hardware-level isolation on ARM platforms. It's supported by a wide range of processors including 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 the normal domain, so it can access the resources 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. 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 domain and the secure domain. If the normal domain requests to switch to the secure domain, the CPU must first enter the monitor mode. The system bus also contains a bit to indicate the state of the bus transaction. Thus, normal peripherals can only perform normal transactions, but not the secure transactions.

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-/write transactions to only interrupts configured as non-secure and allowing secure transactions to all interrupts regardless of security configurations. By default, the TZIC uses Fast Interrupt FIQ as secure interrupt and uses Regular Interrupt IRQ 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. The signal can be either 0 or 1, and each pin of GPIO contributes a bit in the GPIO block. The GPIO can be used to trigger interrupts to the TZIC; however, if the source is masked off in the GPIO, the corresponding interrupt request cannot be forwarded.

3 Threat Model and Assumptions

On a TrustZone-enabled ARM platform, when the Rich OS crashes due to system failure, the Rich OS may not be able to send a secure interrupt to switch the system into the secure domain. When the Rich OS has been compromised, an armored malware can intercept the switch request and fake a memory acquisition process with a “Man in the Middle” attack. It is critical to ensure that TrustDump is securely activated to perform reliable memory dump. Since a malicious Rich OS may target at compromising the memory acquisition module to defeat the memory acquisition process, we must protect the integrity of the TrustDump.

We assume the attacker has no physical access to the smartphone. The ROM code is secure and cannot be flashed. The smartphone has the TrustZone hardware support, which is used to protect the memory acquisition module in the secure domain.

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, while a self-contained software module 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 capable of performing online analysis such as OS integrity checking and Rootkit detection after filling the semantics gap. Also, the acquired memory and CPU registers can be transmitted to a remote computer for logging and further analysis.

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

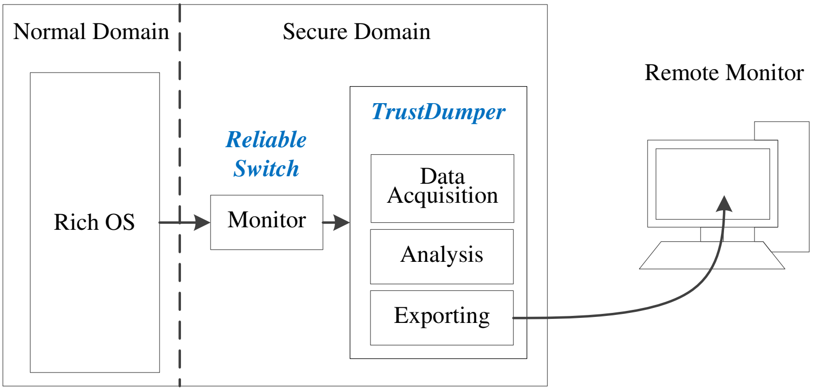

Fig. 1. The System Framework of TrustDump

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 seems simple, but the source code customized to run in the secure domain cannot be directly executed in the normal domain. Since there is no open source Linux kernel 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 do not need to install a full-featured OS in the secure domain, which dramatically reduces the TCB of the system.

4.2 Reliable Switching

A reliable switching into the secure domain is the prerequisite for a reliable memory acquisition. We must ensure the switching will happen 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.

TrustZone provides two ways to enter the secure domain from the normal domain: *SMC* instruction and *Secure Interrupt*. The 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, the SMC instruction may not be called after the crash happens. Alternatively, secure interrupts 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.

Non-maskable interrupt (NMI) has been widely used and deployed on mobile platforms [20, 21], which can trigger one NMI by pressing a button or a combination of several buttons. Since the Rich OS cannot block or intercept NMI, we can use one NMI to enforce the system switching. However, for mobile platforms that do not have dedicated NMI (e.g., Freescale i.MX53 QSB [22]), we solve this problem by configuring one secure interrupt as the NMI.

4.3 Data Acquisition and Transmission

The software module in the secure domain has access privileges to the 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 outside the Rich OS, it has to fill the semantic gaps. Based on the knowledge of the kernel data structures, the analysis module can reconstruct the context of the Rich OS and then perform malware analysis tasks in the secure domain, such as verifying the integrity of the Rich OS and detecting 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 a serial port or a network card. A hash value of the acquired memory is also transmitted to verify the data transmission process. Since the DMA and the peripherals may be used by the Rich OS when the switching happens, their states should be saved 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 staying. Thus, a malicious Rich OS cannot launch Denial of Service attacks to block or intercept the switching. After the NMI being triggered, TrustDump will freeze the Rich OS, so the malware in the Rich OS has no chance to clean its traces.

The TrustDumper has the privilege to access all the memory and CPU registers of the Rich OS, so it may 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.

5 Implementation

We implement a prototype using Freescale *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 the serial port to a Thinkpad-T430 laptop that runs Ubuntu 12.04 LTS. Our TrustDump prototype contains only 450 lines of code.

5.1 Deployment of TrustDump

Since we cannot find any open source OS working in the normal domain, we have to port an Android OS from the secure domain to the normal domain based on the Board Support Package (BSP) published by Adeneo Embedded [23]. Next, we deploy the TrustDumper in the secure domain.

The OS code running in the secure domain cannot execute in the normal domain without proper modification. Since the normal domain has a lower privilege than the secure domain, there are some peripherals that cannot be accessed from the normal domain. For instance, 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 the low power mode. To run Android in the normal domain, we develop a pair of secure I/O functions, *secure_write* and *secure_read*, to enable the normal domain to access the peripherals in the secure domain.

The function definitions are shown in Listing 1. *secure_write* writes 32-bit `data` to the physical address `pa`. Similarly, *secure_read* reads from the physical address `pa` and returns the result. Each peripheral on the i.MX53 QSB has certain configuration registers, which are usually accessed as physical addresses on the board. A *Whitelist* is maintained in the secure domain to store all the registers that the normal domain can access through these two secure I/O functions.

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

Note all the three components are critical to the reliable switching. The compromise of any of the three components will result in an unreliable switching. If the source of the interrupt can be masked by the Rich OS or the Rich OS just blocks all the FIQs to the ARM processor, then the switching to the secure domain will be blocked. To prevent those attacks, we construct an NMI using GPIO-2 interrupt. We first set the GPIO-2 interrupt as a secure interrupt in TZIC. 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. Moreover, through configuring the registers of ARM processor, we set the FIQ requests to be handled in the secure domain.

To minimize the impacts on the access of the Rich OS to other peripherals that share the same access privilege with GPIO-2, we propose a method to enable *Fine-grained Access Control*. Also, to minimize the impacts on the functionalities of other peripherals, we propose a method to enable *Fine-grained Interrupt Control*. It can differentiate the interrupts that share the same interrupt number and distribute them to dedicated handlers in different domains.

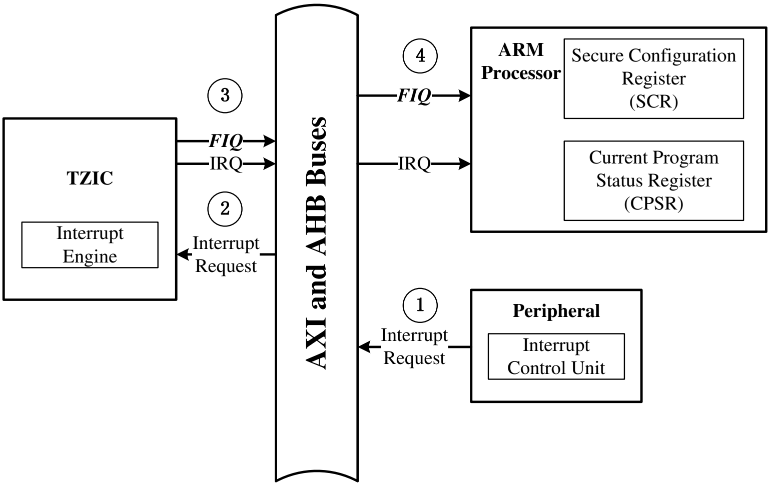

Fig. 2. The Control Flow of Reliable Switching

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 attached to the fifteenth pin of the second GPIO: GPIO-2.

First, the interrupt type of GPIO-2 is set as *secure* in Interrupt Security Registers (TZIC_INTSEC). This prevents the normal domain from accessing the GPIO-2 interrupt configuration in the TZIC. Second, we set the *F* bit in CPSR to 0 to enable FIQ exception. We also set the *FW* bit in SCR to 0 to ensure the FIQ enable (*F*) bit in CPSR cannot be modified by the normal domain. After the configuration of these two bits, the normal domain cannot block the FIQ request to the ARM processor. Third, we set the *FIQ* bit in SCR to 1 to enforce the ARM processor to branch to the monitor mode on an FIQ exception. This step ensures that the FIQ request to secure domain cannot be intercepted or blocked by the normal domain. Finally, we disable the non-secure access to GPIO-2 in CSU so that the interrupt unit of GPIO-2 cannot be configured by the normal domain.

When the ARM processor branches to the monitor mode in the secure domain after the secure interrupt happens, the CPU executes the instruction located in the vector table of the monitor mode at the offset of `0x1C`. After the memory acquisition finishes, the CPU executes the instruction: `subs pc, lr, #4` to return to the normal domain.

Fine-Grained Access Control. The secure domain and the normal domain have different access control policies over the peripherals. The secure domain can access the peripherals belonging to the normal domain, but not vice versa. CSU determines which domain a peripheral belongs to, so we can set access control policies of peripherals by setting the corresponding registers in CSU. We configure GPIO-2 as secure peripheral to prevent the normal domain from accessing it.

However, this simple access control management forces several peripherals to share the same access control policy. For instance, in our prototype, user-defined button 1 and 2 are two pins of GPIO-2 and share the same access policy. We use them in different domains: button 1 is the source of NMI and button 2 is used as the Home Key for the Rich OS. If we disable the non-secure access to user-defined button 1, the non-secure access to button 2 will be denied too, which disables the Home Key in the normal domain.

To solve this problem, we develop a fine-grained access control that sets the peripherals sharing the same policy as secure and releases those peripherals needed in the normal domain by adding them into a *Whitelist*. The Rich OS uses the secure I/O functions described in Listing 1 to access the released peripherals. In this way we can protect the source of NMI from the normal domain without constraining the access of the normal domain to other devices.

Fine-grained Interrupt Control. There is only one interrupt number for all the 32 pins of GPIO-2; however, each pin will generate the same interrupt number 52. Therefore, after we construct the NMI, button 2 will generate the same FIQ request as button 1 does. When the user-defined button 1 is dedicated to trigger an NMI, button 2 will trigger the same NMI, instead of serving as the Home Key as designed in the Rich OS. We solve this problem by developing a fine-grained interrupt control to distribute the interrupts generated by these two buttons to different handlers.

No matter which button is pressed, CPU goes into the secure domain first. Because the functions of the Rich OS cannot be called in the secure domain, the request of button 2 will be forwarded to the normal domain to call the functions of the Rich OS instead of being processed locally as button 1 does. The FIQ exception handler of the Rich OS receives the request and calls the corresponding operation codes in the Rich OS. The entry of FIQ exception is at a static address `0xFFFF01C`. The FIQ mode is not used by the Rich OS, so we can freely use the FIQ exception handlers.

The program flow of hardware interrupts in TrustDump is depicted in Figure 3. The IRQ exception asserted by non-secure interrupt is handled in the Rich OS. The IRQ exception handler gets the number of the pending interrupt from TZIC and gives it to the operation codes.

Upon FIQ request asserted by a secure interrupt, the system will switch to the FIQ exception entry of the secure domain according to the configuration of the TZIC. The FIQ exception handler of the secure domain figures out the source of interrupt through the interrupt control unit of GPIO-2. If the interrupt is an NMI, the handler clears the interrupt status in the TZIC to prevent re-entry. Next, it goes into TrustDumper to perform memory acquisition and analysis. At last, the system returns to the Rich OS.

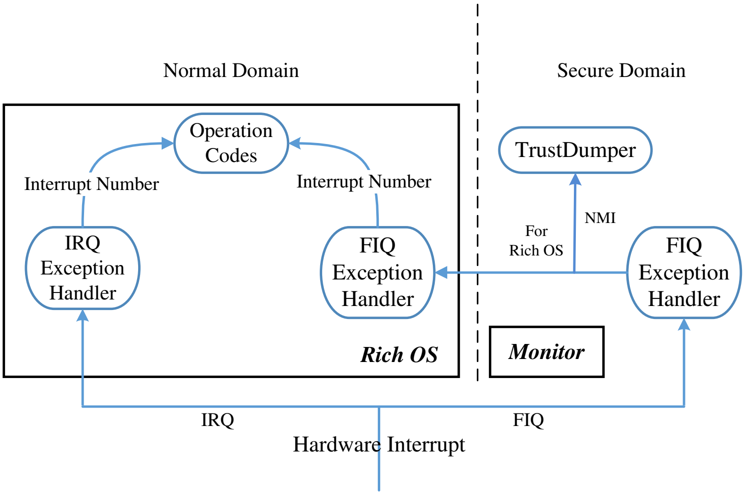

Fig. 3. Program Flow of Interrupt

If the source of the FIQ exception is for the Rich OS, the handler masks the interrupt by setting the interrupt mask register (IMR) in GPIO-2. It stops the interrupt request to TZIC and thus clears the interrupt status in TZIC to prevent re-entry after entering the Rich OS. Besides, masking the interrupt in the handler keeps the interrupt status in the interrupt control unit of GPIO-2, which is used to distinguish different pins of GPIO-2 by the Rich OS. Since the Rich OS can access the interrupt control unit of GPIO-2 to determine which pin generates the interrupt, it can locate the source after receiving an interrupt number 52.

Because the secure domain will not be re-entered, the context of the normal domain stored in the secure domain must be restored before the system jumps to the FIQ handler of the Rich OS. The handler is entered by changing CPU mode to FIQ mode and jumping to the entry of FIQ exception in the normal domain.

In case of return, the FIQ exception handler saves the CPU context first. Then it calls the operation codes in the Rich OS with the interrupt number 52. The operation codes find the source of the interrupt and take the corresponding actions according to the interrupt number. In our prototype, the action function is `mx3_gpio_irq_handler`, which further checks which pin of GPIO generates the interrupt.

As we have masked off the source bit, the function ignores the interrupt and returns directly without doing anything due to failure to pass the mask status judgment.

We enforce the function to bypass the mask status judgment when button 2 is triggered by `or`-ing the corresponding bit with `1` in the judgment statement. With the mask status judgment passed, the action of the user-defined button 2 is taken in the normal domain. After the codes finish running, system returns to the handler. The handler then recovers the stored context and starts exception return by executing the instruction: `subs pc, lr, #4`.

5.3 TrustDumper

The TrustDumper is responsible for acquiring the physical memory and the CPU registers of the Rich OS, performing simple online analysis, and then transmitting the acquired data to a remote machine for further analysis.

Data Acquisition and Transmission. ARM processors have banked registers: one copy for the normal domain and the other copy for the secure domain. In the monitor mode, the processor uses the copy for the secure domain but can also access the copy for the normal domain.

Since the secure domain can access the physical memory of the normal domain, the TrustDumper can directly access the Rich OS's physical address. However, to access the virtual addresses in the Rich OS, the TrustDumper must walk the page tables of the Rich OS to get the corresponding physical addresses. The physical base address of the page table is saved in the Translation Table Base Register (TTBR).

Memory dumping involves transmitting RAM memory to the peripherals. Because this data transmission is time-consuming, we take advantage of the DMA on the board. Since DMA has its own processing core and internal memory, the application processor can continue working on other tasks while the memory is being dumped. The DMA core executes routines that are stored in the internal RAM to perform DMA operations. Before transmitting, the TrustDumper saves the current state of the DMA, exporting the state of the processing core and the routines from the internal RAM to an unused system RAM on the board. Then it downloads the memory dumping code and the corresponding context to the internal RAM. After that, the TrustDumper triggers the DMA and starts to dump memory to the peripherals. When the data transmission is done, an interrupt will be generated for the TrustDumper to restore the core state and DMA internal RAM from the system RAM on the board. In our prototype, we use the serial port as the peripheral to transmit the RAM memory to a remote laptop. In our future work, we will add other peripherals such as network card in our system.

Integrity Checking and Rootkit Detection. In our prototype, the analysis module is capable of checking the integrity of kernel code and detecting rootkits. We provide two implementations, one hardware-based solution and one software-based solution, of SHA-1 algorithm to check the integrity of Android kernel.

We leverage the Symmetric/Asymmetric Hashing and Random Accelerator (SAHARA) of i.MX53 QSB, a security co-processor that 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, to perform hardware-based hash. Since not all ARM platforms have a hardware security accelerator, we also provide a software-based SHA-1 implementation by porting the open source project PolarSSL [24] to i.MX53 QSB. The memory operations and output functions of SHA-1 algorithm in PolarSSL are modified to accommodate the bare-metal environment of the secure domain. Since the performance of hardware hash is better than software hash, we use the hardware to check the kernel integrity.

To calculate a hash value, the start address and length of the target code is required. Theres a static offset between the physical address and the virtual address of the continuous kernel code. In our prototype, the virtual start address of kernel is `0x80004000` and the offset is `0x10000000`, so the physical start address is `0x70004000`. The length of the kernel is case-sensitive, varying from different versions of kernel. Yet after the kernel has been compiled, the length is fixed. In TrustDump, the length is `9080836` bytes.

Our prototype can also detect rootkits that hide malicious processes. Figure 4 illustrates the list of process in linux kernel 2.6.35. In linux, a process is represented by the struct named `task_struct`, which includes the process number (`pid`) and the memory descriptor of the process (`mm`). All the processes are linked by the struct `list_head`, a doubly linked list in `task_struct`. Becasue `task_struct` is a component of the struct `thread_info`, the address of the `task_struct` corresponding to the current running process can be located through the `thread_info`, which is located at (`stack pointer &(0x1FFF)`). Therefore, through retrieving the doubly linked list, all the information of the processes are listed and can be checked to discover the hidden malicious processes.

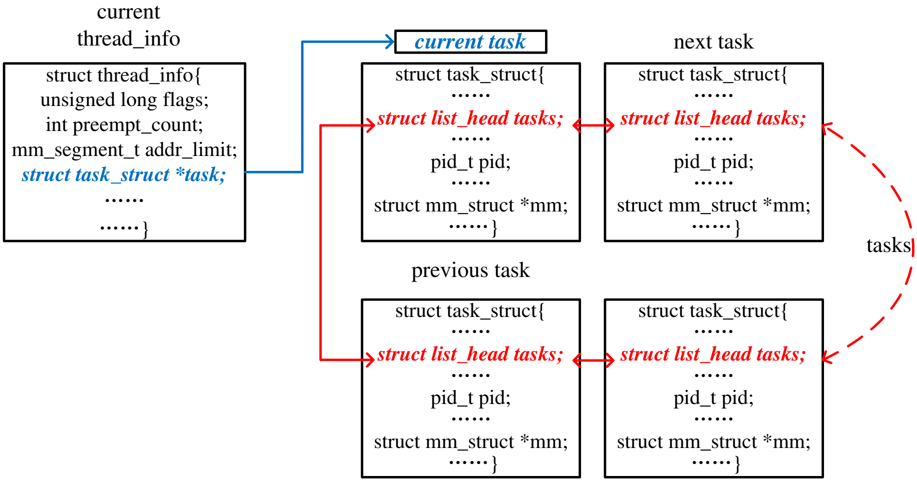

Fig. 4. Process List

6 Performance Evaluation

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

6.1 NMI Switching Time

We measure the time of entering TrustDump with NMI and SMC instruction for comparison. For NMI measurement, since the performance monitor can only be started by software, there is no way to start the performance monitor at the exact time when the button is pressed. It cannot be done to directly measure the time from triggering of the interrupt to handling it in the secure domain. To start the performance monitor right before the NMI is triggered, we assert the NMI in the software-based way. On our board, software can trigger the NMI by writing the NMI interrupt number into the Software Interrupt Trigger Register (TZIC_SWINT) of TZIC. Therefore, we measure the time from writing to the register to receiving the request in the secure domain to evaluate the NMI performance. The result shows that switching time using NMI is 1.7 us, which is neglectable. We also measure the switching time using the SMC instruction by measuring the time from invoking the SMC instruction to receiving the request in the secure domain. The average switching time using SMC instruction is 0.3 us. This is shorter than the time of using NMI because it takes more time for the request of NMI to be transferred to the processor. However, the switching time using NMI is still very small and almost imperceptible. Moreover, using NMI is more reliable than using the SMC instruction to enforce a domain switch.

6.2 Memory Dumping Time

There are two ways to read and send RAM memory content to peripherals: CPU and DMA. In TrustDump, we choose DMA to free the burden of dumping memory from CPU. However, our experimental results show that the memory dumping time using DMA is almost as fast as that of using CPU.

To make the result more convincing, we pick four scales of memory content size: 10 B, 100 B, 1 KB, and 10 KB. For 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 dumping speed: bit rates. The bit rates of each scale are shown in Table 1. We can see that DMA performs as fast as CPU. Based on the result, it will take approximately 13.14 minutes in average to dump Android Kernel of 9080836 bytes to a laptop through the serial port. The bottleneck of the speed is the limited baud rate, which is 115200, of the serial port. The performance can be improved by using other faster peripherals, such as the Ethernet and wireless device. Since it requires to develop new device drivers in the secure domain, we put them into our future work.

Table 1. Memory Dumping Performance

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

6.3 Analysis Performance

We conduct experiments on both the software-based and hardware-based implementations. The result shows that the time to calculate the kernel hash is 1.56 ms by hardware, and 578.6 ms by software. The performance of hardware hash guarantees that TrustDumper can be invoked frequently to perform kernel integrity checking when using the hardware-based solution. Though the software-based solution may be too slow for frequent OS integrity checking, it can be used when the Rich OS crashes or is compromised.

Besides kernel integrity checking, TrustDumper can detect hidden processes. We deploy a real rootkit Suterusu [25] that can hide processes in the Rich OS for evaluation. Suterusu performs system call inline hooking on arm platform to hide user-specified processes. Whenever the *ls* or *top* command is called in linux terminal, Suterusu hooks the functions and deletes the information of the hidden malicious processes from the result. TrustDump can successfully detect the rootkit by traversing all the processes of the Rich OS in 2.13 ms. According to the implementation in 5.3, TrustDumper running in the monitor mode needs to access the stack pointer of the user mode to obtain the pointer of the current *thread_info* in Rich OS. Because the user mode and the system mode of the CPU share the same stack pointer, and changing between the monitor mode and the system mode can be easily done by modifying the Current Program Status Register (CPSR), we access the stack pointer of the system mode instead. With the stack pointer, we can traverse all the processes listed in Figure 4 as described in 5.3. By comparing the result with what we get using command *ls* or *top*, we can find the processes hidden by the Suterusu.

7 Related Work

Memory acquisition techniques on smartphones can be classified into two categories: the software-based solutions and the hardware-based solutions. A software-based memory acquisition solution typically relies on either an OS running on the bare metal to acquire its own memory or a hypervisor to acquire the memory of one VM. Without /dev/mem support in the Android kernel, Linux Memory Extractor (LiME) has been developed as a loadable kernel module in Android to directly dump the memory to the SD card or over the network [26]. It requires rooted devices to insert the module into the kernel. Based on LiME, another work called DMD [27] can acquire the volatile memory of Android. Moreover, DDMS [28] provided by Android SDK can also be used to get memory information. On smartphones, the Android Recovery Mode [29] can give the user a root privilege and bypass the passcodes to acquire the OS memory; however, it requires a reboot before the memory acquisition.

In recent years, Hypervisors have been developed and enabled on ARM platforms [30, 31] with hardware support. Thus, the virtual machine inspection techniques [1] can also be implemented on the smartphones to protect the memory acquisition module from being tampered by the malicious OS. All above software-based solutions are efficient and easy to use. However, since they rely on the Android OS or a hypervisor to acquire the RAM memory, they cannot ensure a reliable memory acquisition when the OS/hypervisor has been compromised.

Hardware-based techniques usually utilize dedicated hardware components to directly access the memory through physical addresses [32], where the OS has been totally bypassed. JTAG [33] and chip-off technique [34] can be used to achieve memory acquisition; however, it works only if a JTAG debug port is identified on the smartphones. Moreover, most deployed OSes deny the debugging requests from JTAG to protect its own security. The cost of the equipment and the destructive nature of chip-off technique make it difficult to be used widely. Gianluigi Me et al. [35] propose a removable memory card based solution to overcome the heterogeneity of the tools adopted to retrieve smartphone contents. The existing hardware-based solution is more secure and reliable. However, it usually demands certain dedicated extra hardware components that may not be available on all smartphone platforms. Fortunately, the ARM processors, which have been widely used on smartphones, now provide a system level isolation solution with a hardware security support called *TrustZone* [15, 16]. TrustZone can ensure a trusted execution environment to protect the memory acquisition module and provide enough access privileges to access the Rich OS memory. Our work is based on TrustZone.

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 between the two domains. 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 a hash value of the Android kernel in 1.56 ms.

Acknowledgment. This work is partially supported by National 973 Program of China under award No. 2014CB340603. Dr. Kun Sun's work is supported by U.S. Army Research Office under Grant W911NF-12-1-0060.

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, pp. 128–138 (2007)
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, pp. 586–600 (2012)

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, pp. 297–312 (2011)
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, pp. 51–62 (2008)
6. Deng, Z., Zhang, X., Xu, D.: Spider: stealthy binary program instrumentation and debugging via hardware virtualization. In: ACSAC, pp. 289–298 (2013)
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 2012, p. 29. USENIX Association (2012)
8. McCune, J.M., Parno, B., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: an execution infrastructure for tcb minimization. In: EuroSys, pp. 315–328 (2008)
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, pp. 143–158 (2010)
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, p. 14. USENIX Association (2012)
11. Zhang, F., Leach, K., Sun, K., Stavrou, A.: Spectre: A dependable introspection framework via system management mode. In: DSN, pp. 1–12 (2013)
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, pp. 38–49 (2010)
13. Wang, J., Stavrou, A., Ghosh, A.: Hypercheck: A hardware-assisted integrity monitor. In: Jha, S., Sommer, R., Kreibich, C. (eds.) RAID 2010. LNCS, vol. 6307, pp. 158–177. Springer, Heidelberg (2010)
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, pp. 375–388 (2011)
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. 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)
27. Sylve, J., Case, A., Marziale, L., Richard III, G.G.: Acquisition and analysis of volatile memory from android devices. Digital Investigation 8(3-4), 175–184 (2012)
28. Google: Using ddms for debugging, http://developer.android.com/tools/debugging/ddms.html
29. Stevenson, A.: Boot into Recovery Mode for Rooted and Un-rooted Android devices, http://androidflagship.com/605-enter-recovery-mode-rooted-un-rooted-android
30. Dall, C., Nieh, J.: Kvm for arm. In: Proceedings of the 12th Annual Linux Symposium (2010)
31. Dall, C., Nieh, J.: Kvm/arm: The design and implementation of the linux arm hypervisor. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014 (2014)
32. Carrier, B.D., Grand, J.: A hardware-based memory acquisition procedure for digital investigations. Digital Investigation 1(1), 50–60 (2004)
33. Breeuwsma, I.M.F.: Forensic Imaging of Embedded Systems Using JTAG (Boundary-scan). Digit. Investig. 3(1) (March 2006)
34. Jovanovic, Z., Redd, I.D.D.: Android forensics techniques. International Academy of Design and Technology (2012)
35. Me, G., Rossi, M.: Internal forensic acquisition for mobile equipments. In: IPDPS, pp. 1–7 (2008)