Abstract—With the wide usage of smartphones in our daily life, new malware is emerging to compromise the mobile OS and then steal or manipulate sensitive data from mobile applications. Forensic analysis tools demand a reliable and trustworthy memory acquisition of the operating systems running on the smartphones for further digital forensic analysis. However, a compromised OS may launch denial of service attacks to prevent a valid memory acquisition by forensic examiners. In this paper, we develop a TrustZone-based memory acquisition mechanism called TrustDump that is capable of reliably and securely obtaining the RAM memory and CPU registers of the mobile OS even if the OS has crashed or been compromised. TrustDump is isolated from the mobile OS by TrustZone. Instead of using a hypervisor to ensure the 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. TrustDump can include basic online analysis modules to catch malware in an early stage. Moreover, the acquired memory and register data can be sent to a remote server through a fast Micro-USB port for real-time forensic analysis when the OS runs or a slow serial port for further forensic analysis when the OS has crashed. A trusted graphical user interface is integrated in the TrustZone to authenticate the user and prevent the misuse of our memory acquisition tool. We build a TrustDump prototype on Freescale i.MX53 QSB.

Index Terms—Memory forensic analysis, TrustZone, non-maskable interrupt, reliable memory acquisition.

I. 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. Therefore, it is critical to perform malware analysis on the emerging mobile OS via malware analysis and immediately update anti-malware tools on the smartphones.

Malware analysis techniques can be classified into two categories: static malware analysis and dynamic malware analysis [1]. With static malware analysis [2]–[4], a program image stored on non-volatile storage is being studied without actual execution. It can reveal the behavior of a program under unusual conditions; however, as the source code of the program is not always available and the binary-form code is complicated, it is impossible to fully predict the behavior of a program. With dynamic malware analysis [5]–[8], a program is studied when its code is loaded into memory and being executed, so the analysis tool can monitor how the program interacts with the network, file system, and memory. In this way potential malware can be identified by analyzing the actual behavior of the program.

There are two main 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 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) [9]–[12] runs a suspicious OS in one VM and the analysis tools in another VM or the hypervisor. This method needs to reconstruct the internal structures of OS kernel to fill the semantic gaps. Recently, Yan and Yin [13] 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 known and unknown vulnerabilities that may be exploited by malware to compromise both the hypervisor and the malware analysis VM. VT-x/SVM [14]–[16], System Management Mode (SMM) [17], [18], and the newest SGX [19] on ×86 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 ARM, the most popular mobile processor. Fortunately, the ARM processors provide a system-level isolation solution with a hardware security support called TrustZone [20], [21], which divides the mobile platform into two isolated execution environments, the normal domain and the secure domain. The OS running in the normal domain is usually called the Rich OS, and the one running in the secure domain is called the 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 acquire the memory of the Rich OS, perform online malware analysis, and transmit the acquired data to a remote server for further analysis. 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 smartphones to trigger a non-maskable interrupt (NMI) to the ARM processor. This hardware-based 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 chance to clean its attacking traces.

TrustDump is capable of checking the OS kernel integrity immediately after acquiring the Rich OS’s memory. For further malware analysis, TrustDumper can send the memory dump and CPU states to a remote machine, which can then use various powerful memory forensics tools [22]–[24] to uncover the malicious behaviors recorded in the memory dump. When the Rich OS does not hang, we can use a fast Micro-USB port in the Rich OS to transmit the memory data to the remote server. A hash value of the memory dump is calculated in the secure domain before sending it along with the memory dump to the remote server, so that the integrity of the memory data being transmitted can be verified even if the Rich OS is malicious. Moreover, when the Rich OS crashes or the Micro-USB port is not available in the Rich OS, we can use a slow but trusted serial port in the secure domain to transmit the data.

Besides its reliability, TrustDump can also ensure a trustworthy memory acquisition. Instead of using a hypervisor to ensure the 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 490 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. Since we do not need to change any code in the Rich OS, it satisfies the smartphone forensic principle of extracting the digital evidence without altering the data contents. Moreover, to prevent our memory acquisition mechanism from being misused by adversaries to steal the sensitive memory data in the Rich OS, we integrate a self-contained Touchscreen driver in the secure domain to authenticate the user before performing the memory operations.

We build a TrustDump prototype on Freescale i.MX53 QSB [25], [26] and verify the effectiveness of TrustDump on reliable and trustworthy memory acquisition. We verify the security and performance of TrustDump on Android version 2.3.4 in the normal domain. Since the architecture of TrustDump is OS agnostic, it can work well with all major versions of Android.

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

- We design a hardware-assisted memory acquisition mechanism named TrustDump to reliably and trustworthily acquire the RAM memory and CPU registers of the mobile OS on smartphones, even if the mobile OS has crashed or been compromised.
- The trusted computing base (TCB) of TrustDump is small, only consisting of a small memory acquisition module and basic malware analysis functions in the secure domain. We do not need to install a hypervisor or a kernel module in the Rich OS.
- The acquired memory data can be transmitted to a remote server through either a fast Micro-USB port when the Rich OS still runs or a slow serial port when the Rich OS hangs.
- 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 \(\mu\)s.

This paper is extended from a preliminary conference paper [27] to enhance the usability and security, and also increase the performance of the developed system. More specifically, the main differences between these two versions are summarized as follows:

- Since the booting sequence of a TrustZone-enabled operating system is different from an ordinary mobile OS, this paper includes more details about the booting process of our system.
- This paper introduces a new self-contained GUI module in the system to enhance the usability and security of our system. First, it provides basic I/O supports that enable users to communicate with the mobile device. Second, a trusted GUI is required to prevent malicious users from misusing our system to steal sensitive information from the Rich OS. Since the GUI module in the Rich OS cannot be trusted, the self-contained GUI module is in the secure domain.
- This paper improves the memory dumping speed by using a Micro-USB port to transmit the memory data to a remote laptop. Compared to the serial port based data transmission solution, we reduce the transmission time by 40 times, so that it can be used to support continuous OS integrity checking.
The remainder of the paper is organized as follows. Section II introduces background knowledge on TrustZone. Section III describes the threat model and assumptions. We present the TrustDump framework in Section IV. A prototype implementation is detailed in Section V. Section VI discusses the experimental results. We describe related works in Section VIII and conclude the paper in Section IX.

II. BACKGROUND

A. TrustZone Overview

TrustZone [20], [21] 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 [28], Cortex-A9 [29] and Cortex-A15 [30]. It creates two isolated execution domains: the secure domain and the normal domain. The secure domain has a higher access privilege than the normal domain. Namely 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, the monitor mode, which only runs in the secure domain regardless of the value of the NS bit. The monitor mode serves as a gatekeeper controlling migration between the normal domain and the secure domain. When 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.

B. 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 access all interrupts regardless of security configurations. By default, the TZIC uses Fast Interrupt Request (FIQ) as secure interrupt and uses Interrupt Request (IRQ) as non-secure interrupt. There are three exception vector tables associated with the normal domain, the secure domain, and the monitor mode, respectively.

C. 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 generated.

III. 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 request to switch the system into the secure domain. Thus, the memory acquisition module in the secure domain cannot be triggered to reliably obtain the Rich OS’s memory and CPU registers.

When the Rich OS has been compromised, a malware can launch Denial-of-Service attacks to block the switch requests for entering the secure domain. In addition, an armored malware can intercept the switch request and fake a memory acquisition process with a Man-in-the-middle attack. Therefore, it is critical to ensure that TrustDump is securely activated to perform a trustworthy memory dump. Moreover, when we use the Micro-USB port in the untrusted Rich OS to transmit the acquired data, we must guarantee that the data has not been manipulated by the Rich OS.

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 TrustDump code. Also, we must prevent attackers from misusing our technique to acquire the memory and uncover sensitive data such as passwords and encryption keys from smartphones owned by other people.

We assume 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. The Rich OS may be compromised by adversaries, but the code in the secure domain is secure and can be trusted.

IV. TRUSTDUMP FRAMEWORK

Figure 1 shows the TrustDump framework using ARM TrustZone security extension. 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. Finally, the acquired memory and CPU registers can be transmitted to a remote monitor for logging and further malware analysis.
A. TrustDump Deployment

A TrustZone-enabled platform always boots from the secure domain. Hence 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 first port the Rich OS to the normal domain and then install the TrustDumper in the secure domain. Since there is no open source project supporting Linux kernel to run in the normal domain on a real platform, we have to port Android OS from the secure domain to the normal domain. The idea 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. We allocate a sealed memory region for the secure domain to run the TrustDumper, and 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.

The boot sequence of TrustDump is depicted in Figure 2. After the system powers on, the code in the ROM runs first. The ROM will load the secure bootloader into the secure domain and transfer control to the secure bootloader. The secure bootloader is responsible for loading the monitor and TrustDumper into the secure domain, initializing the secure domain, constructing the page tables, setting up exception vector table, and configuring the security privilege of the normal domain. The secure bootloader also loads the non-secure bootloader of the Rich OS into the normal domain. Next, the execution jumps to the monitor that switches the system from the secure domain to the normal domain. After the normal domain is entered, the non-secure bootloader starts to bootstrap the Rich OS. At last, the Rich OS is running in the normal domain.

B. Reliable Switching

A reliable switching into the secure domain is a prerequisite for a reliable memory acquisition. We must ensure that the switching process will happen per the user’s request even if the Rich OS has been 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 or any applications running in the Rich OS to initiate the switching process even if the Rich OS is secure. Second, our system should prevent a malicious Rich OS from launching Denial of Service attacks or Man-in-the-middle attacks to block or intercept the switching requests, respectively.

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 before the crash happens. Alternatively, secure interrupts of TrustZone can be called to switch from the normal domain to the secure domain. By default, TrustZone uses the fast interrupt request FIQ as the secure interrupt and uses the interrupt request IRQ as the normal interrupt.

Non-maskable interrupt (NMI) has been used and deployed on mobile platforms [31], [32], where one NMI can be triggered 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 [25]) or all NMIs have been occupied, we solve this problem by configuring one secure interrupt as the NMI. Since the secure interrupt is well supported in TrustZone-enabled platforms, our NMI configuration method can be widely applied on all TrustZone-enabled ARM platforms.

C. Data Acquisition and Transmission

When the system enters the secure domain, the Rich OS in the normal domain is frozen. The software 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.

Our system supports both online malware detection and offline malware analysis. For online malware detection, since the analysis module in the secure domain 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 immediately perform some 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 Micro-USB port. If the Rich OS does not crash, we can use a fast Micro-USB port in the Rich OS to transmit the memory data to the remote server. Since the Rich OS cannot be trusted, a secure hash value of the memory dump is calculated in the secure domain before being sent along with the memory dump to the remote server. When the

Fig. 2. Boot Sequence of TrustDump.
Micro-USB port becomes unavailable due to Rich OS crash or compromise, we can use a slow but trusted serial port in the secure domain to transmit the data. Since DMA is shared by both worlds, TrustDumper may not directly use the DMA when it is being occupied by the Rich OS. However, through our experiments, we see that the DMA conflicts rarely happen. Even when this happens, we can always fall back to the serial port based solution that does not rely on the DMA.

D. User Authentication

Since the acquired memory and CPU registers contain private and sensitive information, it should not be accessed without a valid user authentication. One simple solution is to use a passcode to authenticate the user when the system is triggered by the NMI to switch to the secure domain. It requires a reliable and trusted graphical user interface (GUI) for the user to input the passcode, but we cannot rely on the GUI provided by the Rich OS. First, when the Rich OS crashes, its Touchscreen driver cannot be used, so we cannot depend on it for the user to reliably input the passcode. Second, when the Rich OS is malicious, it can manipulate the GUI to steal the passcode [33].

To solve this problem, we integrate a self-contained Touchscreen driver to provide a trusted GUI in the secure domain. After a NMI is triggered, our GUI module first checks with the user if the NMI was triggered intentionally or not. If triggered unintentionally, it returns to the Rich OS without any memory acquisition operation. Otherwise, it will ask the user to input the passcode for authentication. The input passcode can only be stored and accessed in the secure domain. Since the GUI module resides in the secure domain, it cannot be intercepted or tampered by the Rich OS. Moreover, the GUI still works even if the Rich OS crashes.

V. Implementation

We implement a prototype using Freescale i.MX53 QSB, a TrustZone-enabled development board [25]. i.MX53 QSB has an ARM Cortex-A8 1 GHz application processor with 1 GB DDR3 RAM memory and a 4GB MicroSD card. It has a 4.3 inch seiko LCD with touchscreen. We deploy Android 2.3.4 in the normal domain. The development board is connected through the serial port and the Micro-USB port to a Thinkpad-T430 laptop that runs Ubuntu 12.04 LTS.

A. Android Kernel Porting

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 [34]. Next, we develop and 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 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 help the normal domain to access the peripherals in the secure domain.

The functions definition is 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 can be accessed in the normal domain through these two secure I/O functions.

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

B. Reliable Switching

Since the development board in our prototype does not provide an NMI, we reserve a secure interrupt (FIQ) of TrustZone to serve as the NMI to ensure the reliable switching. Figure 3 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 is 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 Rich OS to access 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 corresponding handlers in different domains.

1) Non-Maskable GPIO-2 Secure Interrupt: An NMI can be either triggered periodically or triggered by event. Entering the TrustDumper periodically can ensure that the Rich OS will be inspected from time to time. The periodic inspection can be achieved by setting a secure timer to generate NMI at a fixed time interval. Alternatively, an event-based interrupt enables random entrance to the secure domain, so that the user can inspect the Rich OS at any time. The event-based interrupt can be achieved by triggering an NMI with a physical peripheral, such as a button.

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 15th pin of the second GPIO (i.e., 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 $F$ bit in SCR to 0 to prevent the FIQ enable ($F$) bit in CPSR from being modified by the normal domain. After setting these two bits, we can ensure that the normal domain cannot block the FIQ requests to the ARM processor. Third, we set the $F$ 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 switches to the monitor mode after receiving a secure interrupt, the CPU executes the instruction located in the vector table of the monitor mode at the offset of $0 \times 1C$. After the memory acquisition finishes, the CPU executes the instruction `subs pc, lr, #4` to return to the normal domain.

2) Fine-Grained Peripheral 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. Since CSU determines which domain a peripheral belongs to, we set the 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.

This simple access control solution makes multiple peripherals share the same access control policy. For instance, in our prototype, user-defined button 1 and 2 are two GPIO-2 pins that share the same access policy. We use button 1 as the source of NMI and button 2 as the Home Key in the Rich OS. If we disable the non-secure access to user-defined button 1, the non-secure access to button 2 that controls the Home Key in the normal domain is disabled too. 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 peripherals. Thus, we can protect the source of NMI from the normal domain without constraining the access of the normal domain to other devices.

3) Fine-Grained Interrupt Control: Since there is only one interrupt number shared by all 32 pins of GPIO-2, each pin will generate the same interrupt numbered 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. 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 in the Rich OS is at a static address $0 \times FFFF01C$. Since FIQ mode is not used by the Rich OS, we have the freedom to use the FIQ exception entry.

Figure 4 shows the control flow of hardware interrupts in TrustDump. The IRQ exception asserted by non-secure interrupt is directly handled in the Rich OS. The IRQ exception handler retrieves the number of the pending interrupt from TZIC and gives it to the operation codes.
Upon an FIQ request asserted by a secure interrupt, the system switches to the FIQ exception entry of the secure domain according to the configuration of the TZIC. The FIQ exception handler of the secure domain can tell 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 TZIC to prevent re-entry and then goes into TrustDumper to perform memory acquisition and analysis. At last, the system returns to the Rich OS.

If the source of the FIQ exception is destined 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. Moreover, it masks the interrupt in the handler to keep the interrupt status in the interrupt control unit of GPIO-2, which can be used to distinguish different pins of GPIO-2. 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 numbered 52. Because the secure domain does not allow reentrancy, 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.

When the system switches back to the normal domain, 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 perform corresponding functions according to the interrupt number. In our prototype, the action function is \texttt{mx3_gpio_irq_handler}, which checks which pin of GPIO generates the interrupt.

\section*{C. 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.

1) Data Acquisition: 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).

In a Linux-based system, only one process is running at a specific time. If TrustDump intends to access the memory resource of the running process, it walks the page table of the process based on the physical base address of the page table in TTBR. However, if TrustDump wants to access the memory resource of a stopped process, it must use the page table of the stopped process. The physical base address of that page table is stored in the memory descriptor, \texttt{mm_struct}, of the stopped process. With the base address of the page table, the physical address corresponding to the virtual address in a stopped process can be obtained by walking the page table of the stopped process.

2) Memory Dumping: 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 in its internal RAM to perform DMA operations. TrustDumper first checks the state of the DMA. If the DMA is still working on the task of the normal domain, TrustDumper falls back to use the CPU to copy the data. Only when the DMA state is idle can TrustDumper use the DMA. Then it saves the current state of the DMA by 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 provide two types of communication peripherals, the serial port and the Micro-USB port, to transmit the acquired RAM memory to a remote laptop.

\begin{itemize}
  \item \textit{a) Using serial port:} We implement a serial port driver in the secure domain to transmit the acquired memory data of the Rich OS to the laptop connected to our board through a USB to serial adapter. The communication between these two parties is managed by \texttt{Minicom}, a software for Unix-like operating systems which sets up a serial console in the laptop to send or receive the serial port data. The serial port driver of TrustDump has only 81 Lines of Code, but it can support the output data in strings, hexadecimal, and decimal formats. It can accept input from the serial console, too. The driver is self-contained and well protected in the secure domain, so it can work even if the Rich OS is malicious or crashes.
  \item \textit{b) Using Micro-USB:} The serial port driver is simple and trusted; however, the data transmission rate is low. Therefore, it is not sufficient to use the serial port to transit the data collected by a high frequent memory dump request. To solve this problem, we extend our prototype to use the fast Micro-USB port to achieve a much higher transmission rate than the serial port. Micro-USB port is widely used on smartphones; however, the hardware controller of the Micro-USB port is more complex than the serial port. Since it will dramatically increase the TCB if we include a Micro-USB driver in the secure domain, we instead leverage the Micro-USB driver in the Rich OS to help transmit the dumped memory.
\end{itemize}

The data flow of Micro-USB based memory dumping procedure is depicted in Figure 5, where only TrustDumper and the laptop are trusted. TrustDumper dumps the memory...
data into the shared buffer in a daemon process of the Rich OS. Then the Micro-USB driver in the Rich OS transmits the memory data in the buffer to a laptop through the Micro-USB to USB adapter. Since the memory dumping is performed in the Rich OS, to protect the integrity of the memory dump, an encrypted hash value of the memory dump will be sent together with the memory dump. The secret key is stored and protected on the non-volatile secure storage of i.MX53 QSB, namely, the electrically-programmable poly fuses (e-Fuses) [26]. Once written, the e-Fuses cannot be modified. Moreover, the e-Fuses can only be accessed by the secure domain. After receiving the memory dump, the laptop first decrypts the encrypted message to get the hash value. Then it compares the value with the hash calculated from the received memory dump. The memory dump is intact when these two values are equal.

**Shared Buffer Between Domains:** The shared buffer is in the data segment of a daemon process that runs in the Rich OS. Upon being launched, the daemon process reserves a 1MB virtual address space as the shared buffer for receiving the memory data from TrustDumper. However, not all the virtual address space requested by `malloc` are allocated physical memory at once. To manage the physical memory resource efficiently, Linux kernel only maps the first page (4KB) to physical memory when `malloc` is invoked. The other pages in the 1MB virtual address space will be mapped to physical address space only when the page is accessed by raising page fault. To ensure the kernel allocates physical memory for the entire shared buffer, the daemon process continues to access each page in the shared buffer. It can be effectively done by writing to only one virtual address in each page within a loop, which will only end after the buffer is filled by TrustDumper.

TrustDumper locate this shared buffer automatically upon being triggered. It first finds the start address of the buffer by parsing the memory map of the daemon process. In Android, each process has its own unique memory map that is stored in the process descriptor, `task_struct`. Thus, this memory map can be found by traversing all the processes in the system (the traversing process is detailed in Section V-C3) and comparing the names with the name of the daemon process. In our prototype, the name of the daemon process is “daemon”. A typical memory map of the daemon process is in Listing 2. In the memory map, each line stands for a virtual memory area (VMA), which consists of several pages. The basic information of the VMA, including the range, the privilege, offset in the mapped file and the name, is also listed. The VMA that has no name in the fourth line of the map is for the shared buffer. From the range we can tell that the VMA is larger than 1MB. Moreover, the start address of the buffer obtained through `malloc`, which is `0x2aaac008`, is not equal to the start address of the VMA. There is a `0x1008` offset from the start address of the buffer to that of the VMA. Therefore, the start address of the shared buffer is the start address of the VMA plus the `0x1008` offset. In order to manipulate the shared buffer in the secure domain, TrustDumper translates the virtual address of the shared buffer to physical address by walking the page table of the daemon process.

```plaintext
start   end     privilege offset   name             
00008000-000a6000 r-xp 00000000 /daemon
000ad000-000b1000 rw-p 0009d000 /daemon
000b1000-000db000 rw-p 00000000 [heap]
2aaba000-2abab000 rw-p 00000000
7ece2000-7ed03000 rw-p 00000000 [stack]
```

Listing 2. A typical memory map of the daemon process.

In our system, we pre-allocate 1MB memory buffer to simplify the memory dumping through Micro-USB port. Since the size of the shared buffer is fixed, the export module in TrustDumper only needs to know the start address of the memory region that is going to be dumped. The address is assigned by the user and stored at the head of the shared buffer by the daemon process. Therefore, the export module reads the 32-bit address data from the beginning of the shared buffer and copies the memory content starting from that address into the shared buffer. To dump more than 1MB data, the daemon process continuously increases the start address of the memory to be dumped by 1MB and waits for TrustDumper to fill. The process continues until all the requested memory data is dumped. An alternative option is to dynamically allocate 1MB memory in the Rich OS only when the memory dumping is demanded. However, this dynamic allocation solution may suffer from both Denial of Service and TOCTTOU attacks since the Rich OS may be malicious. Therefore, we choose the static allocation solution. Note the reserved 1MB memory can be released when the memory dumping is done.
Data Transmission Through Android Debug Bridge: After detecting the content change of the shared buffer, the daemon process starts to transmit the data to the laptop. When an Android device is connected to a laptop through a Micro-USB to USB adapter, the laptop can control the Android device through Android Debug Bridge (ADB) [35]. The ADB is a command line tool, and an ADB server that runs on the laptop manages the commands to send request to the Android device. The ADB daemon running on the Android device is in charge of the communication with ADB server. It receives the request from the ADB server and interacts with the daemon process. With the help of the ADB, the laptop can remotely access the console of the Android system by invoking adb shell command in its own console. In the remote console, the user can access the file system and execute programs on the Android device. Therefore, we can launch the daemon process through a remote console on the laptop. Since in our prototype the printf of the C library is implemented to print information to the console, the daemon process calls printf to transmit the content of the shared buffer to that remote console.

The usage of Micro-USB driver in the Rich OS may suffer from denial-of-service (DoS) when the Rich OS is malicious. If the Rich OS is malicious, it can manipulate the Micro-USB driver and block its data transmission. Moreover, when the Rich OS crashes, the Micro-USB driver may not be available and its behavior is unpredictable. However, when the DoS happens, we can always use the slow serial port to transmit memory dump.

3) Integrity Checking and Rootkit Detection: In our prototype, the analysis module is capable of checking the integrity of kernel code and detecting certain types of 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 [36] to the secure domain. The memory operations and output functions of SHA-1 algorithm in PolarSSL are modified to accommodate the bare-metal environment of the secure domain. The hardware solution has a better performance than the software solution.

To calculate a hash value, the start address and length of the target code is required. There is 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 code is \(0x80004000\) and the offset is \(0x1000000\), so the physical start address is \(0x70004000\). The length of the kernel varies from different versions. 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 by traversing the process list in the system. Figure 6 illustrates the list of process in Linux kernel 2.6.35. In Linux, a process is represented by the process descriptor, 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. Since 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.

D. Self-Contained GUI Driver in the Secure Domain

The development board that we use supports many types of graphical I/O devices, including VGA monitor, LVDS monitor,
HDMI monitor, and Parallel LCD that can all be connected to the board for display. Besides, LVDS monitor and LCD can be used as Touchscreen for user input. LVDS monitor needs more hardware module supports than the LCD. In our prototype, we use the LCD as the I/O device.

A reliable and trusted driver of the I/O device is required for achieving a reliable GUI. To ensure the GUI is secure and reliable from a malicious or crashed Rich OS, we implement a self-contained GUI driver in the secure domain. The touchscreen driver provides limited functionalities, only to satisfy the basic I/O need of TrustDump. The driver is kept simple to contain only 273 lines of code excluding the raw images. The GUI driver consists of two parts: the LCD driver for display and the touchscreen driver for user input.

1) **LCD Driver for Display:** The core of the display system on the development board is an Image Processing Unit (IPU), which manages the data flow from the display framebuffer to the external display device. It can also conduct image processing and manipulation on the data flow. The IPU supports different input pixel formats. In our prototype, there are 800×480 pixels in the framebuffer and the pixel format is RGB565. The size of the framebuffer, the pixel format and the location of the framebuffer are all set in the Channel Parameter Memory (CPMEM). The IPU is also responsible for communicating with display device, either directly or through a bridge (such as a TV encoder or an LVDS display bridge). The IPU is directly connected to the LCD as the display interface of the IPU supports the input format of the LCD. The display pixel format of the LCD is 24-bit parallel RGB and the resolution is 800×480. The display driver controls both the IPU and the LCD to guarantee the picture displayed on the LCD is rendered by TrustDump.

The display driver is kept simple to reduce the size of TCB, only containing basic display functionalities. It does not provide complicated image processing functions such as frame rotation or planes combination. The display driver can generate a limited number of pictures that are necessary for TrustDump. The raw pictures for display are stored in the secure domain. The display driver copies the needed raw pictures to different places in the framebuffer and sets the background to black color (the value of which in RGB565 is 0×0000). By dynamically generating the frames, the size of the stored raw pictures in the secure domain is reduced.

The Rich OS and TrustDump have different display framebuffers. The framebuffer of the Rich OS is in the normal domain, while the framebuffer of TrustDump is stored in the secure domain. Therefore, the TrustDump’s framebuffer is well protected from being tampered by the normal domain. After the IPU is enabled, it parses the metadata of the framebuffer from the CPMEM and directs the data in the framebuffer to the display device automatically. Therefore, there is no need to back up any framebuffers between domain switching. Only the metadata of the framebuffer in the IPU needs to be backed up when TrustDump is entered. The privilege of the IPU should also be changed to secure in order to access the framebuffer in the secure domain and then changed back to non-secure before returning to the Rich OS. Since the Rich OS and TrustDump share the IPU, TrustDump will store the state of IPU, including the metadata of the framebuffer and other hardware configurations, and recover it before returning to the Rich OS.

Figure 7 shows an authentication screen of TrustDump. This screen pops out every time the user presses the user-defined button 1 to trigger TrustDump. This screen is composed of three raw pictures: the “TrustDump” logo, the “Return to Android” button and the soft keyboard. The other part of the screen is set to black. There is a “Return to Android” button in the screen to avoid the user wasting time waiting for the dumping to finish when TrustDump is triggered unintentionally.

2) **Touchscreen Driver for User Input:** The LCD is equipped with a 4-wire resistive touchscreen that is connected to the Power Management Integrated Circuit (PMIC). The PMIC collects the voltage of the touchscreen and converts the analog voltage to digital value that represents the X-Y coordinate of the touch on the screen. The value of each coordinate is stored in a register in PMIC and its value is zero when there is no touch event. Every time a touch happens, an interrupt arises. The interrupt handler then gets the user’s intention according to the location where the touch happens and processes the touch event. Since the touchscreen is used both by the Rich OS and TrustDump, there are two different interrupt handlers: one in the normal domain and the other in the secure domain. After TrustDump is triggered by NMI, the interrupt of the touchscreen is set to secure (FIQ). Then when there is a touch on the screen, an interrupt rises in the secure domain. The GUI module of TrustDumper reads the X-Y coordinate from the registers in PMIC and proceeds according to the user’s intention. After all the activities end in the secure domain, the interrupt of the touchscreen is configured back to non-secure.
TABLE I
MEMORY DUMPING PERFORMANCE (SERIAL PORT)

<table>
<thead>
<tr>
<th>Scale (Byte)</th>
<th>Bit Rate (bit/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>92178.12</td>
</tr>
<tr>
<td>100</td>
<td>92163.38</td>
</tr>
<tr>
<td>1K</td>
<td>92163.01</td>
</tr>
<tr>
<td>10K</td>
<td>92163.0</td>
</tr>
</tbody>
</table>

VI. 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.

A. NMI Switching Time

We measure the switching time delays of entering TrustDump using NMI and SMC instruction, respectively. When using NMI, since the performance monitor can only be started by software, it is difficult to start the performance monitor at the exact time when the physical button is pressed. Thus, we cannot 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 a 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 the switching time using NMI is 1.7 μs, which is small and negligible.

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 μs. This is shorter than the time of using NMI since it takes more time for the NMI request to be transferred to the processor. However, the switching time using NMI is still very small. Moreover, using NMI is more reliable than using the SMC instruction to enforce a domain switch.

B. 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.

1) Using Serial Port: For memory dumping using the serial port, we pick four scales of memory content size: 10 Bytes, 100 Bytes, 1 KB, and 10 KB. For each scale, we conduct the experiments 50 times. We take the average value and divide the result with the scale to get the dumping bit rates, which are shown in Table 1. We can see that the bit rate is steady at 92163 bit/s. It takes approximately 13.14 minutes in average to dump Android Kernel of 9080836 bytes to a laptop through the serial port. Though it is slow and can hardly be used for continuous kernel integrity checking, it is a reliable channel to dump the memory when the Rich OS is compromised or crashes.

2) Using Micro-USB: The experimental results verify that it is much faster for transmitting the data using Micro-USB port than using the serial port. The dumping process of using Micro-USB is broken into three steps:

   1) TrustDumper fills the shared buffer.
   2) The hash of the shared buffer is calculated in the secure domain.
   3) The daemon process prints the buffer to a remote laptop.

We run the experiments 50 times and report the average value. The time consumption of each step is listed in Table II. It takes 2.16 seconds in total to transmit a 1MB data block. It consists of 20.08 ms to fill the buffer, 0.18 ms to generate the hash value of the buffer, and 2.138 s to transmit 1MB data to the remote laptop. The average switching time (1.7 μs in VI-A) can be ignored as it is much smaller compared to the other switching times. To dump more than 1MB memory data, TrustDump needs to be entered multiple times. Since TrustDump is entered by pressing the user-defined button 1, it is hard for a person to catch the exact time to press the button as soon as the last memory dumping is finished. To solve this problem, we remove the human factor by triggering TrustDump through SMC instruction. Note we make this replacement only for evaluation purpose. The result shows that it takes approximately 19.53 seconds in average to dump an Android Kernel of 9080836 bytes to a laptop through the Micro-USB port. The data rate using Micro-USB port is about 40 times faster than that using the serial port.

C. Online Memory Analysis

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. Though the software-based solution may be too slow for frequent OS integrity checking, it can be used when the hardware hash capability is not supported on the mobile platform.

Besides kernel integrity checking, TrustDumper can detect hidden rootkit processes. We deploy a real rootkit Suterusu [37] that can hide processes in the Rich OS. 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. TrustDumper can successfully detect the rootkit
by traversing all the processes of the Rich OS in 2.13 ms. According to the implementation in V-C, 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 the Rich OS. Because the user mode and the system mode of the CPU share the same stack pointer and it is easy to change between the monitor mode and the system mode by modifying the Current Program Status Register (CPSR), we can access the stack pointer of the system mode instead. With the stack pointer, we can traverse all the processes listed in Figure 6 as described in V-C. By comparing the result with what we get using command ls or top, we can successfully identify the processes hidden by the Suterusu.

VII. DISCUSSION

A. Security Analysis and Limitations

One of our major design goals is to ensure a memory dumping even if the Rich OS is compromised or simply crashes. With the NMI triggered by a physical button, TrustDump can reliably migrate 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 DoS attacks to block or intercept the switching. If the Rich OS has crashed, the NMI can migrate the system into the secure domain too. After the NMI being manually triggered, since the Rich OS is frozen, the malware in the Rich OS has no chance to clean its traces.

The TrustDumper in the secure domain has the privilege to access all the memory and CPU registers of the Rich OS, so it is capable of checking the integrity of the Rich OS and detecting various malware such as rootkits in the Rich OS. Since the TrustDumper is securely isolated from the Rich OS by TrustZone, a compromised Rich OS cannot compromise the TrustDumper.

Since the acquired memory and CPU registers contain private information, they should not be accessed without authentication. To prevent a user from misusing our mechanism, we use a passcode to authenticate the user when the system is triggered by the NMI to switch to the secure domain. The password is input through a trusted GUI module in the secure domain. The GUI module is self-contained and does not rely on any code in the Rich OS. Since the passcode is stored in the secure domain, only the owner of the device has the access to the passcode.

One major limitation of TrustDump is to require the battery power always on during the memory acquisition process. When the Rich OS shuts down, the system is triggered by the NMI to switch to the secure domain. If the Rich OS has crashed, the NMI can migrate the system into the secure domain too. After the NMI being manually triggered, since the Rich OS is frozen, the malware in the Rich OS has no chance to clean its traces.

B. Usage of TrustDump

TrustDump focuses on acquiring RAM memory and CPU registers for performing offline static malware analysis, while it can be further extended to support online dynamic malware analysis such as rootkit detection as shown in Section V-C3.

Most dynamic malware analysis tools assume that the Linux Kernel can be trusted, so they can use semantic information provided by the OS to perform dynamic malware analysis. Our system assumes that the Rich OS may be compromised and thus cannot be trusted, so we rely on filling the semantic gaps in the secure domain to perform out-of-box malware analysis. Previous bare-metal forensic analysis tools can be adapted for ARM platforms to work in TrustDump. Note that since smartphones are resource-constrained devices, it may be too expensive to run heavy-weight online forensic analysis tools on smartphones. Instead, TrustDump can dump the memory information outside the smartphones for offline analysis that can exploit all variants of powerful malware analysis tools on external high-performance platforms.

VIII. 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. A standard Android kernel can output memory information with the help of /dev/mem and /dev/kmem. Without 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 [46]. It requires rooted devices to insert the module into the kernel. Based on LiME, another work called DMD [47] can acquire the volatile memory of Android. Moreover, DDMS [48] provided by Android SDK can also be used to get memory information. On smartphones, the Android Recovery Mode [49] 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 with hardware support. Thus, the virtual machine inspection techniques 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, where the OS has been totally bypassed. Reina et al. [57] leverage the isolated environment in x86 platform to dump physical memory of a running system. JTAG [58] and chip-off technique [59] 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. Me and Rossi [60] 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 some 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-wide isolation solution with a hardware security support called TrustZone [20], [21]. TrustZone can ensure a trusted execution environment to protect the memory acquisition module and provide enough access privileges to access the Rich OS memory. Several TrustZone-based systems (e.g., Mobicore/Trustonics [61], Trusted Logic [62], ObCs [63], [64], and KNOX [65]) have been developed to enhance the security of mobile device. MobiCore [61] is a secure Operating System for TrustZone-enabled ARM controllers including ARM1176 or CortexA8/A9. It provides development tools called Trustlets for third-party application developer.

A number of research efforts have been made on TrustZone. Jang et al. [66] build a secure channel between the secure domain and the process in the normal domain to protect the session key from a compromised Rich OS. Since when Micro-USB port is used the memory dump in the shared buffer is plaintext, TrustDump does not need a secure channel to protect the memory dump. Beside, without the help of TrustZone, a process in the Rich OS can still dump memory using the Micro-USB port [67]; however, the memory dump will be tampered if the Rich OS is compromised. Azab et al. [68] perform real-time kernel protection of the normal domain from the secure domain. Santos et al. [69] construct a trusted language runtime that supports C# in the secure domain. Marforio et al. [70] deploy the smartphones supporting TrustZone as location tokens for payments. The payments are protected by the secure domain.

IX. CONCLUSIONS

Based on ARM TrustZone technology, we propose a reliable and trustworthy 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 a self-contained 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. Moreover, we can prevent attackers from misusing our mechanism to steal the sensitive information from the Rich OS. Our prototype on i.MX53 QSB can enter TrustDump and begin memory dumping in 1.7 ms, calculate a hash value of the Android kernel in 1.56 ms and finish transmitting a kernel size of 9MB in 19.53 s.

ACKNOWLEDGMENT

The authors would like to thank Ozgur Sinanoglu and other reviewers for their insightful comments and feedback.

REFERENCES

Kun Sun received the Ph.D. degree in computer science from North Carolina State University. He was a Senior Research Scientist with Intelligent Automation Inc. He was a Research Professor with George Mason University. He was a member of the Technical Staff with Bell Laboratories, Lucent Technology, in 2000. He is currently an Assistant Professor with the College of William and Mary. He has over ten years working experience in both industry and academia. His research focuses on systems and network security.

Yuewu Wang received the Ph.D. degree from Graduate University, Chinese Academy of Sciences (CAS), in 2008. From 2008 to 2011, he was an Assistant Professor with Data Assurance and Communication Security, CAS. In 2012, he joined the Institute of Information Engineering, CAS, along with Data Assurance and Communication Security. He is currently an Associate Professor of the Institute of Information Engineering at CAS. He has authored 20 referred technical papers. His research interest is in the field of information security, including mobile system security, network attack simulation, and information system security evaluation.

Jiwu Jing received the degree from the Department of Electronics Engineering, Tsinghua University, in 1987, and the M.Sc. and Ph.D. degrees from the Graduate School, University of Science and Technology of China, in 1990 and 2003, respectively. He is currently a Professor of the Institute of Information Engineering, Chinese Academy of Sciences (CAS). He is the Chief Engineer of the Institute of Information Engineering and the Director of Data Assurance and Communication Security, CAS. He is also the Group Leader of Expert Group on Information Security of National High Technology Research and Development Program of China (863 Program). He has authored 90 referred technical papers. His main research interest is in the field of information security including public key infrastructure, identity management, fault tolerance, mobile security, information system, and network protection. His research has won the National Science and Technology Progress Awards, Science and Technology Progress Award of the Chinese Ministry of Electronics Industry, and the Science and Technology Progress Award of CAS.