This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2018.2883027, IEEE Transactions on Information Forensics and Security

Hardware-assisted Transparent Tracing and Debugging on ARM

Zhenyu Ning and Fengwei Zhang

***Abstract*—Existing malware analysis platforms leave detectable fingerprints like uncommon string properties in QEMU, signatures in Android Java virtual machine, and artifacts in Linux kernel profiles. Since these fingerprints provide the malware a chance to split its behavior depending on whether the analysis system is present or not, existing analysis systems are not sufficient to analyze the sophisticated malware. In this paper, we propose NINJA, a transparent malware analysis framework on ARM platform with low artifacts. NINJA leverages a hardware-assisted isolated execution environment TrustZone to transparently trace and debug a target application with the help of Performance Monitor Unit and Embedded Trace Macrocell. These hardware features help NINJA to achieve transparency while avoiding heavy performance overhead. NINJA does not modify system software and is OS-agnostic on ARM platform. We implement a prototype of NINJA (i.e., tracing and debugging subsystems), and the experiment results show that NINJA is efficient and transparent for malware analysis. An improved fast system restoration mechanism is also designed to facilitate the continuous malware analysis.**

***Index Terms*—ARM, transparent, tracing and debugging**

I. INTRODUCTION

MALWARE on the mobile platform exhibits an explosive growth in recent years, and a variety of tools have been proposed for malware detection and analysis [1], [2], [3], [4], [5], [6], [7], [8]. However, sophisticated malware, which is also known as evasive malware, is able to evade the analysis by collecting the artifacts of the execution environment or the analysis tool, and refuses to perform any malicious behavior if an analysis system is detected.

As most of the existing mobile malware analysis systems [1], [5], [6] are based on emulation or virtualization technology, a series of anti-emulation and anti-virtualization techniques [9], [10], [11] have been developed to challenge them. These techniques show that the emulation or virtualization can be detected by footprints like string properties, the absence of particular hardware components, and performance slowdown. The hardware-assisted virtualization technique [12], [13] improves the transparency of the virtualization-based systems; however, this approach leaves artifacts on instruction execution semantics that could be detected by malware [14].

To address this challenge, researchers study the malware on bare-metal devices via modifying the system software [2], [3], [4], [7] or leveraging OS APIs [8], [15] to monitor the runtime behavior of malware. Although bare-metal based approaches eliminate the detection of the emulator or hypervisor, the artifacts introduced by the analysis tool itself are still detectable by malware. Moreover, privileged malware can even manipulate the analysis tool since they run in the same environment. How to build a transparent mobile malware analysis system is still a challenging problem.

This transparency problem has been well studied in the traditional x86 architecture, and similar milestones have been made from emulation-based analysis systems [16], [17] to hardware-assisted virtualization analysis systems [18], [19], [20], and then to bare-metal analysis systems [21], [22], [23], [24]. However, this problem still challenges the state-of-the-art malware analysis systems.

We consider that an analysis system consists of an *Environment* (e.g., operating system, emulator, hypervisor, or sandbox) and an *Analyzer* (e.g., instruction analyzer, API tracer, or application debugger). The *Environment* provides the *Analyzer* with the access to the states of the target malware, and the *Analyzer* is responsible for the further analysis of the states. Consider an analysis system that leverages the emulator to record the system call sequence and sends the sequence to a remote server for further analysis. In this system, the *Environment* is the emulator, which provides access to the system call sequence, and both the system call recorder and the remote server belong to the *Analyzer*. Evasive malware can detect this analysis system via anti-emulation techniques and evade the analysis.

To build a transparent analysis system, we propose three requirements. Firstly, the *Environment* must be isolated. Otherwise, the *Environment* itself can be manipulated by the malware. Secondly, the *Environment* exists on an off-the-shelf (OTS) bare-metal platform without modifying the software or hardware (e.g., emulation and virtualization are not). Although studying the anti-emulation and anti-virtualization techniques [9], [10], [11], [14] helps us to build a more transparent system by fixing the imperfections of the *Environment*, we consider perfect emulation or virtualization is impractical due to the complexity of the software. Instead, if the *Environment* already exists in the OTS bare-metal platform, malware cannot detect the analysis system by the presence of the *Environment*. Finally, the *Analyzer* should not leave any detectable footprints (e.g., files, memory, registers, or code) to the outside of the *Environment*. An *Analyzer* violating this requirement can be detected.

In light of the three requirements, we present NINJA [1], a transparent malware analysis framework on ARM platform

Z. Ning and F. Zhang are with the COMPASS lab, Department of Computer Science, Wayne State University, Detroit, MI 48202, USA. Email: zhenyu.ning@wayne.edu; fengwei@wayne.edu

[1] A NINJA in feudal Japan has invisibility and transparency ability

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2018.2883027, IEEE Transactions on Information Forensics and Security

based on hardware features including TrustZone technology, Performance Monitoring Unit (PMU), and Embedded Trace Macrocell (ETM). We implement a prototype of NINJA that embodies a trace subsystem with different tracing granularities and a debug subsystem with a GDB-like debugging protocol on ARM Juno development board. Additionally, hardware-based traps and memory protection are leveraged to keep the use of system registers transparent to the target application. The experimental results show that our framework can transparently monitor and analyze the behavior of the malware samples. Moreover, NINJA introduces reasonable overhead. We evaluate the performance of the trace subsystem with several popular benchmarks, and the result shows that the overheads of the instruction trace and system call trace are less than 1% and the Android API trace introduces 4 to 154 times slowdown.

In addition, as the malware sample may tamper the current system state, which can lead to an inaccurate analysis result of the next sample, a fast restoration mechanism is required for continuous malware analysis. Previous restoration mechanisms either require a system reboot [24] or require special hardware components [22], [25]. We also implement a prototype of an improved fast restoration mechanism which leverages TrustZone and ETM data address trace to selectively restore memory and Network File System (NFS) to swap file system for speeding up the restoration. The experiments show that our fast restoration mechanism can restore the system in 0.029s to 2.160s on NXP i.MX53 Quick Start Board. Our experiments with π calculation [26] and memory benchmark [27] also show that the memory changed by a program is only a small portion of the whole memory, which indicates that the selective memory restoration is more effective than the full memory restoration.

The main contributions of this work include:

- We present a hardware-assisted analysis framework, named NINJA, on ARM platform with low artifacts. It does not rely on emulation, virtualization, or system software, and is OS-agnostic. NINJA resides in a hardware isolation execution environment, and thus is transparent to the analyzed malware.
- NINJA eliminates its footprints by novel techniques including hardware traps, memory mapping interception, and timer adjusting. The evaluation result demonstrates the effectiveness of the mitigation and NINJA achieves a high level of transparency.
- We implement a prototype of an improved fast and complete restoration mechanism that selectively restores memory, remotely swaps network file system, and completely restores registers to a clean state.
- We implement debugging and tracing subsystems with a variety of program analysis functionalities. NINJA is capable of studying kernel- or hypervisor-level malware. The tracing subsystem exhibits a low performance overhead. Our evaluation results show that the instruction tracing and system call tracing are immune to timing attacks.

This paper is an extended version of our previous work [28] published in USENIX Security 2017. Based on that work, we implement a fast restoration mechanism to facilitate the continuous malware analysis. We also improve the functionality and usability of the trace and debug subsystem. The main differences between these two versions are summarized as follows:

- We improve previous fast restoration mechanism with selective memory restoration, runtime file system switching, and complete context recovery. The selective memory restoration and runtime file system switching help to improve the performance of system restoration while the complete context recovery mitigates the incompleteness of previous restoration systems.
- This paper introduces data address trace in the trace subsystem which allows analysts to learn the target memory address of memory read/write instructions. The data address trace is helpful in many different use cases such as fast restoration (see Section V-E), dynamic taint analysis [29], and inferring encryption keys [30].
- The usability of trace subsystem is improved via introducing address range and process ID filters. These filters help analysts to focus on the interested processes and memory addresses, and greatly reduce the noise in the trace result. Moreover, we add six more stepping modes for step-by-step debugging including speculative-execution-related stepping that may help analyze recent Meltdown [31] and Spectre [32] attacks.

II. BACKGROUND

A. TrustZone and Trusted Firmware

ARM TrustZone technology [33] introduces a hardware-assisted security concept that divides the execution environment into two isolated domains, i.e., secure domain and non-secure domain. Due to security concerns, the secure domain could access the resources (e.g., memory and registers) of the non-secure domain, but not vice versa. In ARMv8 architecture, the only way to switch from normal domain to secure domain is to trigger a secure exception [34], and the exception return instruction `eret` is used to switch back to the normal domain from the secure domain after the exception is handled.

ARM Trusted Firmware [35] (ATF) is an official implementation of secure domain provided by ARM, and it supports an array of hardware platforms and emulators. While entering the secure domain, the ATF saves the context of the normal domain and dispatches the secure exception to the corresponding exception handler. After the handler finishes the handling process, the ATF restores the context of the normal domain and switches back with `eret` instruction. ATF also provides a trusted boot path by authenticating the firmware image with several approaches like signatures and public keys.

B. PMU and ETM

The Performance Monitors Unit (PMU) [34] is a feature widely implemented in both x86 and ARM architectures [36], which leverages a set of performance counter registers to calculate CPU events. Each architecture specifies a list of common events by event numbers, and different CPUs may also

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2018.2883027, IEEE Transactions on Information Forensics and Security

maintain additional event numbers. A Performance Monitor Interrupt (PMI) can be triggered while a performance counter register overflows. Note that the PMU is a non-invasive debug feature that does not affect the performance of the CPU.

The Embedded Trace Macrocell (ETM) [37] is another non-invasive debug component in ARM architecture. It traces instructions and data by monitoring instruction and data buses with low performance impact. Actually, ARM expects that ETM has no effect on the functional performance of the processor. The ETM generates an element for executed signpost instructions that could be further used to reconstruct all the executed instructions. The generated elements are encoded into a trace stream and sent to a pre-allocated buffer on the chip.

III. Related Work

A. Transparent Malware Analysis on x86

Ether [19] and Spider [18] leverage hardware virtualization to build a malware analysis system and achieves high transparency. Since the hardware virtualization has transparency issues, these systems are naturally not transparent. LO-PHI [23] leverages additional hardware sensors to monitor the disk operation and periodically poll memory snapshots, and it achieves a higher transparency at the cost of incomplete view of system states. MalT [24] leverages PMU to monitor the program execution and increases the transparency by involving System Manage Mode (SMM). However, it still suffers from external timing attack, and the PMU registers are not well protected. BareCloud [22] and MalGene [21] focus on detecting evasive malware by executing malware in different environments and comparing their behavior, but how to transparently fetch and study the malware behavior still needs to be solved.

B. Dynamic Analysis Tools on ARM

Emulation-based systems. DroidScope [6] rebuilds the semantic information of both the Android and the Dalvik virtual machine based on QEMU. CopperDroid [5] is a VMI-based analysis tool that reconstructs the behavior of Android malware including inter-process communication (IPC) and remote procedure call interaction. DroidScibe [1] uses CopperDroid [5] to collect behavior profiles of malware and automatically classifies them into different families. Due to the emulator's footprints, these systems are natural not transparent.
Hardware virtualization. Xen on ARM [13] migrates the hardware virtualization based hypervisor Xen to ARM architecture and makes the analysis based on hardware virtualization feasible on mobile devices. KVM/ARM [12] uses standard Linux components to improve the performance of the hypervisor. Although the hardware virtualization based solution is considered to be more transparent than the emulation or traditional virtualization based solution, it still leaves some detectable footprints on CPU semantics while executing specific instructions [14].
Bare-metal systems. TaintDroid [2] is a system-wide information flow tracking tool. It provides variable-level, message-level, method-level, and file-level taint propagation by modifying the original Android framework. TaintART [4] extends the idea of TaintDroid on the most recent Android Java virtual machine Android Runtime (ART). VetDroid [7] reconstructs the malicious behavior of the malware based on permission usage, and it is applicable to taint analysis. DroidTrace [8] uses `ptrace` to monitor the dynamic loading code on both Java and native code level. BareDroid [38] provides a quick restore mechanism that makes the bare-metal analysis of Android applications feasible at scale. Malton [39] adopts multi-layer monitoring, information flow tracking, and efficient path exploration to ART to achieve a comprehensive view of the malicious behavior. PackerGrind [40] and DexLego [41] work against the packers to obtain the real behavior of Android applications. Although these tools attempt to analyze the target on real-world devices to improve transparency, the modification to the Android framework leaves some memory footprints or code signatures, and the ptrace-based approaches can be detected by simply check the `/proc/self/status` file. Moreover, these systems are vulnerable to privileged malware.

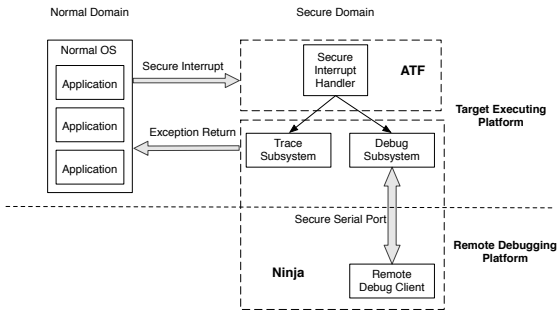

Figure 1: Architecture of NINJA.

C. System Restoration

BareBox [42] restores the system memory via overriding the OS memory with a previous snapshot and recovers the file system with help of a mirror disk. BareCloud [22] requires a cluster of hardware-based modular worker units and uses Logical Volume Manager(LVM)-based snapshots to restore the system. MalT [24] restores the system memory and context via rebooting, and recovers the file system by monitoring the disk operations. Bolt [25] uses the similar approach with BareBox to recover the system memory and leverages hardware features of flash storage devices to restore the file system. Unlike these systems, NINJA utilizes hardware-trace-based selective memory restoration to boost the memory restoration and restores the file system via runtime file system switch.

IV. System Architecture

Figure 1 shows the architecture of NINJA. The NINJA consists of a target executing platform and a remote debugging client. In the target executing platform, TrustZone provides hardware-based isolation between the normal and secure domains while the rich OS (e.g., Linux or Android) runs in the normal domain and NINJA runs in the secure domain. We setup a customized exception handler in Exception Level 3 (EL3) to handle asynchronous exceptions (i.e., interrupts) of our interest. NINJA contains a Trace Subsystem (TS) and a Debug Subsystem (DS). The TS is designed to transparently trace the execution of a target application, which does not

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2018.2883027, IEEE Transactions on Information Forensics and Security

need any human interaction during the tracing. This feature is essential for automatic large-scale analysis. In contrast, the DS relies on human analysts. In the remote debugging platform, the analysts send debug commands via a secure serial port and the DS then responds to the commands. During the execution of an application, we use secure interrupts to switch into the secure domain and then resume to the normal domain by executing the exception return instruction `eret`.

A. Reliable Domain Switch

Normally, the `smc` instruction is used to trigger a domain switch by signaling a Secure Monitor Call (SMC) exception which is handled in EL3. However, as the execution of the `smc` instruction may be blocked by privileged malware, this software-based switch is not reliable.

Another solution is to trigger a secure interrupt which is considered as an asynchronous exception in EL3. ARM Generic Interrupt Controller (GIC) [43] partitions all interrupts into secure group and non-secure group, and each interrupt is configured to be either secure or non-secure. Moreover, the GIC Security Extensions ensures that the normal domain cannot access the configuration of a secure interrupt. Regarding to NINJA, we configure PMI to be a secure interrupt so that an overflow of the PMU registers leads to a switch to the secure domain. To increase the flexibility, we also use similar technology mentioned in [44] to configure the General Purpose Input/Output (GPIO) buttons as the source of secure Non-Maskable Interrupt (NMI) to trigger the switch. The switch from secure domain to normal domain is achieved by executing the exception return instruction `eret`.

B. The Trace Subsystem

The Trace Subsystem (TS) provides the analyst the ability to trace the execution of the target application in different granularities during automatic analysis including instruction tracing, system call tracing, Android API tracing, and data address tracing. We achieve the instruction, system call tracing, and data address tracing via hardware component ETM, and the Android API tracing with help of PMU registers. Note that the Android API tracing is designed specifically for Android, while the instruction and system call tracing are OS-agnostic.

By default, we use the GPIO button as the trigger of secure NMIs. Once the button is pressed, a secure NMI request is signaled to the GIC, and GIC routes this NMI to EL3. NINJA toggles the enable status of ETM after receiving this interrupt and outputs the tracing result if needed. Additionally, the PMU registers are involved during the Android API trace. Note that the NMI of GPIO buttons can be replaced by any system events that trigger an interrupt (e.g., system calls, network events, clock events, and etc.), and these events can be used to indicate the start or end of the trace in different usage scenarios.

Another advanced feature of ETM is that PMU events can also be configured as an external input source. In light of this, we specify different granularities of the tracing. For example, we trace all the system calls by configuring the ETM to use the signal of PMU event `EXC_SVC` as the external input.

C. The Debug Subsystem

In contrast to the TS, the Debug Subsystem (DS) is designed for manual analysis. It establishes a secure channel between the target executing platform and the remote debugging platform, and provides a user interface for human analysts to introspect the execution status of the target application.

To interrupt the execution of the target, we configure the PMI to be secure and adjust the value of the PMU counter registers to trigger an overflow at a desired point. NINJA receives the secure interrupt after a PMU counter overflows and pauses the execution of the target. A human analyst then issues debugging commands via the secure serial port and introspects the current status of the target following our GDB-like debugging protocol. To ensure the PMI will be triggered again, the DS sets desirable values to the PMU registers before exiting the secure domain.

Moreover, similar to the TS, we specify the granularity of the debugging by monitoring different PMU events. For example, if we choose the event `INST_RETIRED` which occurs after an instruction is retired, the execution of the target application is paused after each instruction is executed. If the event `EXC_SVC` is chosen, the DS takes control of the system after each system call. Meantime, the event `INST_SPEC` event helps the DS pause the execution after an instruction is speculatively executed.

V. DESIGN AND IMPLEMENTATION

We implement NINJA on a 64-bit ARMv8 Juno r1 board and a 32-bit ARMv7 NXP i.MX53 Quick Start Board (QSB). The Juno board integrates two Cortex-A57 cores and four Cortex-A53 cores, and each core includes separate PMU, ETM, and TrustZone. Based on the ATF and Linaro's deliverables on Android 5.1.1 for Juno, we build a customized firmware for the board. Since the ETMs of the cores on the Juno board only support instruction tracing, we also implement a prototype of NINJA on the i.MX53 QSB based on an open source project [45] to demonstrate data address tracing. Note that NINJA is also compatible with commercial mobile devices because it relies on existing deployed hardware features.

A. Bridge the Semantic Gap

As with the VMI-based [46] and TEE-based [24] systems, bridging the semantic gap is an essential step for NINJA to conduct the analysis. In particular, we face two layers of semantic gaps in our system.

1) The Gap between Normal and Secure Domains: In the DS, NINJA uses PMI to trigger a trap to EL3. However, the PMU counts the instructions executed in the CPU disregarding to the current running process. That means the instruction which triggers the PMI may belong to another application. Thus, we first need to identify if the current process is the target. Since NINJA is implemented in the secure domain, it cannot understand the semantic information of the normal domain, and we have to fill the semantic gap to learn the current process in the OS.

In Linux, each process is represented by an instance of `thread_info` data structure, and the one for the current

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2018.2883027, IEEE Transactions on Information Forensics and Security

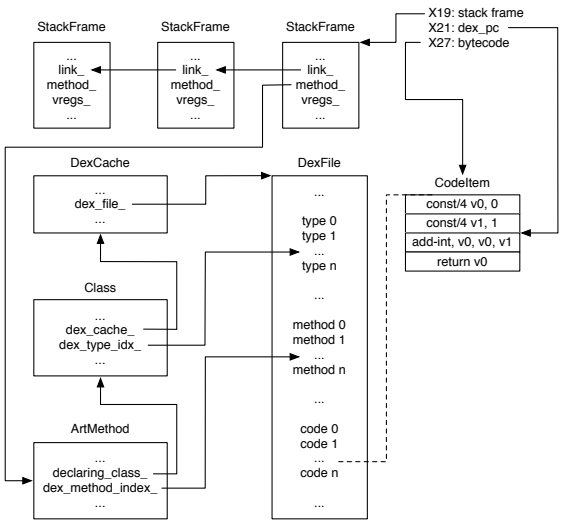

Figure 2: Semantics in the Function `ExecuteGotoImpl`.

process could be obtained by `SP & ~(THREAD_SIZE - 1)`, where `SP` indicates the current stack pointer and `THREAD_SIZE` represents the size of the stack. Next, we can fetch the `task_struct`, which maintains the process information (like pid, name, and memory layout), from the `thread_info`. Then, the target process can be identified by the pid or process name.

2) The Gap in Android Java Virtual Machine: Android maintains a Java virtual machine to interpret Java bytecode, and we need to figure out the current executing Java method and bytecode during the Android API tracing and bytecode stepping. DroidScope [6] fills the semantic gaps in the Dalvik to understand the current status of the VM. However, as a result of Android upgrades, Dalvik is no longer available in recent Android versions, and the approach in DroidScope is not applicable for us.

By manually analyzing the source code of ART, we learn that the bytecode interpreter uses `ExecuteGotoImpl` or `ExecuteSwitchImpl` function to execute the bytecode. The approaches we used to fill the semantic gap in these two functions are similar, and we use function `ExecuteGotoImpl` as an example to explain our approach. In Android, the bytecode of a Java method is organized as a 16-bit array, and ART passes the bytecode array to the function `ExecuteGotoImpl` together with the current execution status such as the current thread, caller and callee methods, and the call frame stack that stores the call stack and parameters. Then, the function `ExecuteGotoImpl` interprets the bytecode in the array following the control flows, and a local variable `dex_pc` indicates the index of the current interpreting bytecode in the array. By manually checking the decompiled result of the function, we find that the pointer to the bytecode array is stored in register `X27` while variable `dex_pc` is kept by register `X21`, and the call frame stack is maintained in register `X19`. Figure 2 shows the semantics in the function `ExecuteGotoImpl`. By combining registers `X21` and `X27`, we can locate the currently executing bytecode. Moreover, a single frame in the call frame stack is represented by an instance of `StackFrame` with the variable `link_` pointing to the previous frame. The variable `method_` indicates the current executing Java method, which is represented by an instance of `ArtMethod`. Next, we fetch the declaring class of the Java method following the pointer `declaring_class_`. The pointer `dex_cache_` in the declaring class points to an instance of `DexCache` which is used to maintain a cache for the DEX file, and the variable `dex_file_` in the `DexCache` finally points to the instance of `DexFile`, which contains all information of a DEX file. Detail description like the name of the method can be fetched via the index of the method (i.e., `dex_method_index_`) in the method array maintained by the `DexFile`. Note that both `ExecuteGotoImpl` and `ExecuteSwitchImpl` functions have four different template implementations in ART, and our approach is applicable to all of them.

B. Secure Interrupts

In GIC, each interrupt is assigned to Group 0 (secure interrupts) or Group 1 (non-secure interrupts) by a group of 32-bit `GICD_IGROUPR` registers. Each bit in each `GICD_IGROUPR` register represents the group information of a single interrupt, and value 0 indicates Group 0 while value 1 means Group 1. For a given interrupt ID `n`, the index of the corresponding `GICD_IGROUPR` register is given by $n \,/\, 32$, and the corresponding bit in the register is $n \bmod 32$. Moreover, the GIC maintains a target process list in `GICD_ITARGETSR` registers for each interrupt. By default, the ATF configures the secure interrupts to be handled in Cortex-A57 core 0.

As mentioned in Section IV-A, NINJA uses secure PMI and NMI to trigger a reliable switch. As the secure interrupts are handled in Cortex-A57 core 0, we run the target application on the same core to reduce the overhead of the communication between cores. In Juno board, the interrupt ID for PMI in Cortex-A57 core 0 is 34. Thus, we clear the bit 2 of the register `GICD_IGROUPR1` ($34 \bmod 32 = 2, 34 \,/\, 32 = 1$) to mark the interrupt 34 as secure. Similarly, we configure the interrupt 195, which is triggered by pressing a GPIO button, to be secure by clearing the bit 3 of the register `GICD_IGROUPR6`.

C. The Trace Subsystem

1) Instruction Tracing: NINJA uses ETM, which is embedded in the CPU and controlled by a group of trace registers, to trace the executed instructions. As the target application is always executed in non-secure EL0 or non-secure EL1, we make the ETM only trace these states by setting all `EXLEVEL_S` bits and clearing all `EXLEVEL_NS` bits of the `TRCVICTLR` register. Then, NINJA sets the `EN` bit of `TRCPRGCTLR` register to start the instruction trace. In regard to stopping the trace, we first clear the `EN` bit of `TRCPRGCTLR` register to disable ETM. To decode the trace result, we use an open source analyzer `ptm2human` [47] to convert the stream to a readable format.

The ETM also supports a variety of filters to narrow the trace, and the narrowed trace helps the analysts focus on only the interested pieces of instructions. NINJA also leverages this feature to achieve multiple restrictions on the instruction tracing.

The address range comparator enables the NINJA to include only a specific address range or exclude a particular address range from the trace. There are four address comparator pairs

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2018.2883027, IEEE Transactions on Information Forensics and Security

in each ETM on the Juno board, and each pair is controlled by two TRCACVR registers to specify an address range. The TRCVIIECTLR register is used to enable these address comparator pairs and control the include or exclude logic. Assume that we aim to restrict the trace in address range 0x0 to 0xF0000000. The INCLUDE bits of the TRCVIIECTLR is set to 1 to enable the address comparator pair 1 while the EXCLUDE bits are left to be 0. To specify the address range in the address comparator pair 1, we set TRCACVR0 and TRCACVR1 registers to 0x0 and 0xF0000000, respectively. With this configuration, the instruction tracing is triggered on in the specified address range.

The ARMv8 architecture [34] uses the CONTEXTIDR_EL1 register to identify the current process of the core. Correspondingly, NINJA traces a single target process or multiple target processes according to the CONTEXTIDR_EL1 register. The address range comparator discussed above traces instructions of different processes in the specified address range. However, in some situations, we are interested in the executed instructions of a specific process (the process of the target application). Thus, we leverage the context ID comparator to achieve the restriction. NINJA first sets the CID bit of TRCCONFIGR to ensure the context ID tracing is enabled. Next, the TRCACATR0 and TRCACATR1 registers are set to 0x904 to add the context ID comparator 0 to the address comparator TRCACVR0 and TRCACVR1. To make the context ID comparator 0 matches the target process, we write the ID of the target process to TRCCIDCVR0 register. Now, the instruct tracing only occurs while instructions in the specified address range of the target process are executing.

2) System Call Tracing: The system call of Linux in ARM platforms is achieved by supervisor call instruction svc, and an immediate value following the svc instruction indicates the corresponding system call number. Since the ETM can be configured to trace the PMU event EXC_SVC, which occurs right after the execution of a svc instruction, we trace the system calls via tracing this event in ETM.

As mentioned in Section IV-B, we can configure the ETM to trace PMU events during the instruction trace. The TRCEXTINSELR register is used to trace at most four external input source, and we configure one of them to trace the EXC_SVC event. In Cortex-A57, the event number of the EXC_SVC event is 0x60, so we set the SEL0 bits of the TRCEXTINSELR register to be 0x60. Also, the SELECT bits of the second trace resource selection control register TRCRSCTLR2 (TRCRSCTLR0 and TRCRSCTLR1 are reserved) is configured to 0 to select the external input 0 as tracing resource 2. Next, we configure the EVENT0 bit of TRCEVENTCTL0R register to 2 to select the resource 2 as event 0. Finally, the INSTEN bit of TRCEVENTCTL1R register is set to 0x1 to enable event 0. Note that the X bit of PMU register PMCR_EL0 should also be set to export the events to ETM. After the configuration, the ETM can be used to trace system calls, and the configuration to start and stop the trace is similar to the one in Section V-C1.

3) Android API Tracing: Unlike the instruction trace and system call trace, we cannot use ETM to directly trace the Android APIs as the existence of the semantic gap. As mentioned in Section V-A2, each Java method is interpreted by ExecuteGotoImpl or ExecuteSwitchImpl function, and ART jumps to these functions by a branch instruction bl. Since a PMU event BR_RETIRED is fired after execution of a branch instruction, we use PMU to trace the BR_RETIRED event and reconstruct the semantic information following the approach described in Section V-A2 if these functions are invoked.

There exist six PMU counters for each processor on Juno board, and we randomly select the last one to be used for the Android API trace and the DS. Firstly, the E bit of PMCR_EL0 register is set to enable the PMU. Then, both PMCNTENSET_EL0 and PMINTENSET_EL1 registers are set to 0x20 to enable the counter 6 and the overflow interrupt of the counter 6. Next, we set PMEVTYPER5_EL0 register to 0x80000021 to make the counter 6 count the BR_RETIRED event in non-secure EL0. Finally, the counter PMEVCNTR5_EL0 is set to its maximum value 0xFFFFFFFF. With this configuration, a secure PMI is routed to EL3 after the execution of the next branch instruction. In the interrupt handler, the ELR_EL3 register, which is identical to the PC of the normal domain, is examined to identify whether the execution of normal domain encounters ExecuteGotoImpl or ExecuteSwitchImpl function. If true, we fill the semantic gap and fetch the information about the current executing Java method. By the declaring class of the method, we differentiate the Android APIs from the developer defined methods. Before returning to the normal domain, we reset the performance counter to its maximum value to make sure the next execution of a branch instruction leads to an overflow.

4) Data Address Tracing: The data address tracing provides the addresses of the data involved in data storing and loading instructions (e.g., str and ldr instructions). These addresses can be used to facilitate the selective memory restoration (see Section V-E), the dynamic taint analysis [29] or help infer the keys of encryption algorithms [30]. Since the data address tracing feature is not available on Juno board, we implement it on the NXP i.MX53 Quick Start Board (QSB) which integrates a Cortex-A8 processor that supports the data address tracing. To demonstrate the OS-agnostic feature of NINJA, we use Ubuntu 12.04 as the rich OS in this implementation.

Although the architecture of the i.MX53 QSB is different from the Juno board, the configuration of the funnels and buffers are similar to that mentioned in Section V-C1. In this section, we only show how to make the ETM in the i.MX53 QSB trace the data addresses.

In the ETMVDCR3 register, NINJA sets the bit 16 to enable the control of excluding memory addresses and the bit 8 to exclude the address range comparator 1. In the address range comparator 1, we exclude all the secure memory addresses by setting the ETMACVR1 and ETMACVR2 to be 0x0 and 0xFFFFFFFF, respectively, and setting the bits [11:10] of ETMACTR1 and ETMACTR2 to be 0b10. The ETMVDEVR register is configured to be 0x6F to enable the data address trace once the ETM is enabled. Due to the different ETM architecture, NINJA uses the etm2human [48] instead of the ptm2human project to decode the trace result.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2018.2883027, IEEE Transactions on Information Forensics and Security

Table I: Representative Stepping Modes in NINJA

PMU Event	Event Description
INST_RETIRED	Fires after an instruction is retired.
BR_RETIRED	Fires after a branch instruction is retired.
BR_MIS_PRED	Counts the mispredicted branch instructions.
L1D_CACHE_LD	Counts the L1 data cache read operation.
L1D_CACHE_REFILL_LD	Counts the L1 data cache refill operation.
LD_SPEC	Fires after a load instruction is speculatively executed.
BR_IMMED_SPEC	Fires after a branch instruction is speculatively executed.

D. The Debug Subsystem

Debugging is another essential approach to learn the behavior of an application. NINJA leverages a secure serial port to connect the board to an external debugging client. There exists two serial port (i.e., `UART0` and `UART1`) in Juno board, and the ATF uses `UART0` as the debugging input/output of both normal domain and secure domain. To build a secure debugging bridge, NINJA uses `UART1` as the debugging channel and marks it as a secure device by configuring NIC-400 [49]. Alternatively, we can use a USB cable for this purpose. In the DS, an analyst pauses the execution of the target application by the secure NMI or predefined breakpoints and send debugging commands to the board via the secure serial port. NINJA processes the commands and outputs the response to the serial port with a user-friendly format. The information about symbols in both bytecode and machine code are not supported at this moment, and we consider it as our future work.

1) Step-by-Step Execution Debugging: The ARMv8 architecture provides instruction stepping support for the debuggers by the `SS` bit of `MDSCR_EL1` register. Once this bit is set, the CPU generates a software step exception after each instruction is executed, and the highest EL that this exception can be routed is EL2. However, this approach has two fundamental drawbacks: 1) the EL2 is normally prepared for the hardware virtualization systems, which does not satisfy our transparency requirements. 2) The instruction stepping changes the value of `PSTATE`, which is accessible from EL1. Thus, we cannot use the software step exception for the instruction stepping. Another approach is to modify the target application's code to generate an SMC exception after each instruction. Nonetheless, the modification brings the side effect that the self-checking malware may be aware of it.

The PMU event `INST_RETIRED` is fired after the execution of each instruction, and we use this event to implement instruction stepping by using similar approach mentioned in Section V-C3. With the configuration, NINJA pauses the execution of the target after the execution of each instruction and waits for the debugging commands.

Besides the instruction-level stepping, the PMU provides a group of other events which are helpful for malware analysis. Table I shows some representative stepping options supported by NINJA via different PMU events. The `BR_RETIRED` event can be used to trace the function calls. The `BR_MIS_PRED`, `LD_SPEC`, and `BR_IMMED_SPEC` events are related to the speculative execution. The recent Meltdown [31] and Spectre [32] attacks abuse the speculative execution to leak the privileged resource, and these speculative-execution-related events may help the analyst step the execution according to the speculative execution and get some insights for detecting these attacks. The `L1D_CACHE_LD` and `L1D_CACHE_REFILL_LD` events can be used to calculate the cache miss ratio, which can be used to detect the ROP attack, Meltdown attack, and Spectre attack [50], [51], [52].

Moreover, NINJA is capable of stepping Java bytecode. Recall that the functions `ExecuteGotoImpl` and `ExecuteSwitchImpl` interpret the bytecode in Java methods. In both functions, a branch instruction is used to switch to the interpretation code of each Java bytecode. Thus, we use `BR_RETIRED` event to trace the branch instructions and firstly ensure the `pc` of normal domain is inside the two interpreter functions. Next, we fill the semantic gap and monitor the value of `dex_pc`. As the change of `dex_pc` value indicates the change of current interpreting bytecode, we pause the system once the `dex_pc` is changed to achieve Java bytecode stepping.

2) Breakpoints: In ARMv8 architecture, a breakpoint exception is generated by either a software breakpoint or a hardware breakpoint. The execution of `brk` instruction is considered as a software breakpoint while the breakpoint control registers `DBGBCR_EL1` and breakpoint value registers `DBGBVR_EL1` provide support for at most 16 hardware breakpoints. However, similar to the software step exception, the breakpoint exception generated in the normal domain could not be routed to EL3, which breaks the transparency requirement of NINJA. MalT [24] discusses another breakpoint implementation that modifies the target's code to trigger an interrupt. Due to the transparency requirement, we avoid this approach to keep our system transparent against the self-checking malware. Thus, we implement the breakpoint based on the instruction stepping technique discussed above. Once the analyst adds a breakpoint, NINJA stores its address and enable PMU to trace the execution of instructions. If the address of an executing instruction matches the breakpoint, NINJA pauses the execution and waits for debugging commands. Otherwise, we return to the normal domain and do not interrupt the execution of the target.

3) Memory Read/Write: NINJA supports memory access with both physical and virtual addresses. The TrustZone technology ensures that EL3 code can access the physical memory of the normal domain, so it is straight forward for NINJA to access memory via physical addresses. Regarding memory accesses via virtual addresses, we have to find the corresponding physical addresses for the virtual addresses in the normal domain. Instead of manually walk through the page tables, a series of Address Translation (AT) instructions help to translate a 64-bit virtual address to a 48-bit physical address[2] considering the translation stages, ELs and memory attributes. As an example, the `at s12e0r` *addr* instruction performs stage 1 and 2 (if available) translations as defined for EL0 to the 64-bit address *addr*, with permissions as if reading from *addr*. The [47:12] bits of the corresponding physical address

[2]The ARMv8 architecture does not support more bits in the physical address at this moment

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2018.2883027, IEEE Transactions on Information Forensics and Security

are storing in the PA bits of the PAR_EL1 register, and the [11:0] bits of the physical address are identical to the [11:0] bits of the virtual address *addr*. After the translation, NINJA directly manipulates the memory in normal domain according to the debugging commands.

E. Improved System Restoration

System restoration is a critical task in continuous malware analysis since the malware in previous analysis session may tamper the system and affect the analysis of the next session. In general, we consider the restoration in three different aspects: memory, file system, and system context. Although some fast restoration systems [22], [24], [25], [42] have been proposed, challenges on all these three aspects.

Challenge 1: Full memory restoration is time-consuming and unreasonable. Previous systems [25], [42] partition the physical memory into different regions and use one of the regions to maintain a clean copy of the OS memory which is further used to restore the OS memory. However, fully restore the whole OS memory is time consuming. As shown in [25], the full memory restoration takes 85% of the whole system restoration time. However, the execution of a program only changes a small portion of the whole memory (as shown in Section VII-C2). As the physical memory of a system increases rapidly nowadays, the full memory restoration would spend much more time on restoring the untainted memory.

Solution: Selective memory restoration. To avoid the cost of copying untainted memory, we aim to learn the specific memory addresses that the program have modified, and the data address trace in TS matches this purpose perfectly. Thus, based on the result of the data address trace discussed in Section V-C4, we implement a selective memory restoration mechanism and only restore the modified memory. Specifically, the i.MX53 QSB contains 1GB physical memory, and we partition it into three regions: a 448MB region for the OS (RG1), a 448MB region for the clean memory copy (RG2), and a 128MB region for the firmware (RG3). Meanwhile, RG2 and RG3 are configured as secure access only via the TrustZone Address Space Controller (TZASC) [33]. After each analysis session, we recover the memory in RG1 with the memory in RG2 via the code in RG3 which is executed in secure domain. Additionally, we enable the data address trace during the analysis and learn the modified memory addresses from the ETM trace result after each analysis session. In the memory restoration process, only these changed memory are restored.

Challenge 2: File system restoration is either costly or requiring special hardware. Most previous systems [22], [24], [42] restore the file system via monitoring the disk write operations, which introduces considerable performance overhead. Bolt [25] uses the hardware feature of flash-based block storage to achieve a fast restoration. However, this requires special flash storage devices and the modification to the firmware of the flash translation layer.

Solution: Runtime file system switching. Runtime file system switching has been proposed on different systems [53], [54], [55], which can be used to efficiently switch to a clean copy of file system after each analysis session. We implement our file system restoration based on the pivot_root system call [53] in Linux to achieve the runtime file system switching. NINJA maintains two clean copies of the file system on a remote file server, and the target executing platform mounts one of them as the file system via NFS. Once the current analysis session is finished, NINJA switches to another copy efficiently via pivot_root, and the restoration of the used copy on the remote file server can be concurrent with the analysis session to speed up the next restoration process. Note that the NFS does not affect the transparency since it is widely used in the popular cloud providers like Amazon AWS [56].

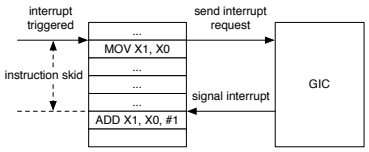

Figure 3: Interrupt Instruction Skid.

Challenge 3: The context restoration is not complete. Bolt [25] restores the system context via recovering the general purpose registers and three other system registers including TTBR, SCTLR and ASID. However, there exist some other system registers which may be manipulated by the malware and affect the system. For example, the TTBCR register determines which of the translation table base registers is used for address translation, and the DACR register defines the access permission of each memory domain. Failing to restore these registers could lead to an unclear copy of the whole system view.

Solution: Complete context restoration. We enumerate all the system registers in the ARMv7 architecture and identify the registers which may be critical to the malware analysis. The values of these registers are restored during the context restoration.

F. Interrupt Instruction Skid

In ARMv8 manual, the interrupts are referred as asynchronous exceptions. Once an interrupt source is triggered, the CPU continues executing the instructions instead of waiting for the interrupt. Figure 3 shows the interrupt process in Juno board. Assume that an interrupt source is triggered before the MOV instruction is executed. The processor then sends the interrupt request to the GIC and continues executing the MOV instruction. The GIC processes the requested interrupt according to the configuration, and signals the interrupt back to the processor. Note that it takes GIC some time to finish the process, so some instructions following the MOV instruction have been executed when the interrupt arrives the processor. As shown in Figure 3, the currently executing instruction is the ADD instruction instead of the MOV instruction when the interrupt arrives, and the instruction shadow region between the MOV and ADD instructions is considered as interrupt instruction skid.

The skid problem is a well-known problem [36], [57] and affects NINJA since the currently executing instruction is not the one that triggers the PMI when the PMI arrives the processor. Thus, the DS may not exactly step the execution of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2018.2883027, IEEE Transactions on Information Forensics and Security

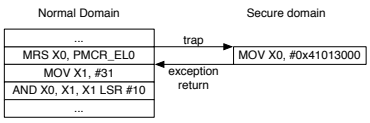

Figure 4: Protect the PMCR_EL0 Register via Traps.

the processor. Although the skid problem cannot be completely eliminated, the side-effect of the skid does not affect our system significantly [28].

VI. Transparency

As NINJA is not based on the emulator or other sandboxes, the anti-analysis techniques mentioned in [9], [10], [11] cannot detect the existence of NINJA. Moreover, other anti-debugging techniques like anti-ptrace [58] do not work for NINJA since our analysis does not use ptrace. Nonetheless, NINJA leaves artifacts such as changes of the registers and the slow down of the system, which may be detected by the target application. Next, we discuss the mitigation of these artifacts.

A. Footprints Elimination

Since NINJA works in the secure domain, the hardware prevents the target application from detecting the code or memory usage of NINJA. Moreover, as the ATF restores all the general purpose registers while entering the secure domain and resumes them back while returning to the normal domain, NINJA does not affect the registers used by the target application as well. However, as we use ETM and PMU to achieve the debugging and tracing functions, the modification to the PMU registers and the ETM registers leaves a detectable footprint. In ARMv8, the PMU and ETM registers are accessible via both system-instruction and memory-mapped interfaces.

1) System-Instruction Interface: The system-instruction interface makes the system registers readable via `MRS` instruction and writable via `MSR` instruction. In NINJA, we ensure that the access to the target system registers via these instructions to be trapped to EL3. The `TPM` bit of the `MDCR_EL3` register and the `TTA` bit of the `CPTR_EL3` register help to trap the access to PMU and ETM registers to EL3, respectively; then we achieve the transparency by providing artificial values to the normal domain. Figure 4 is an example of manipulating the reading to the `PMCR_EL0` register and returning the default value of the register. Before the `MRS` instruction is executed, a trap is triggered to switch to the secure domain. NINJA then analyzes the instruction that triggers the trap and learns that the return value of `PMCR_EL0` is stored to the general-purpose register `X0`. Thus, we put the default value 0x41013000 to the general-purpose register `X0` and resume to the normal domain. Note that the `PC` register of the normal domain should also be modified to skip the `MRS` instruction. We protect both the registers that we modified (e.g., `PMCR_EL0`, `PMCNTENSET_EL0`) and the registers modified by the hardware as a result of our usage (e.g., `PMINTENCLR_EL1`, `PMOVSCLR_EL0`).

2) Memory Mapped Interface: Each of the PMU or ETM related components occupies a distinct physical memory region, and the registers of the component can be accessed via offsets in the region. Since these memory regions do not locate in the DRAM (i.e., main memory), the TZASC, which partitions the DRAM into secure regions and non-secure regions, cannot protect them directly. Note that this hardware memory region is not initialized by the system firmware by default and the system software such as applications and OSes cannot access it because the memory region is not mapped into the virtual memory. However, advanced malware might remap this physical memory region via functions like `mmap` and `ioremap`. Thus, to further defend against these attacks, we intercept the suspicious calls to these functions and redirect the call to return an artificial memory region.

The memory size for both the PMU and ETM memory regions is 64k, and we reserve a 128k memory region on the DRAM to be the artificial PMU and ETM memory. The ATF for Juno board uses the DRAM region 0x880000000 to 0x9fffffff as the memory of the rich OS and the region 0xa00000000 to 0x1000000000 of the DRAM is not actually initialized. Thus, we randomly choose the memory region 0xa00040000 to 0xa00060000 to be the region for artificial memory mapped registers. While the system is booting, we firstly duplicate the values in the PMU and ETM memory regions into the artificial regions. As the function calls are achieved by `bl` instruction, we intercept the call to the interested functions by using PMU to trigger a PMI on the execution of branch instructions and compare the `pc` of the normal domain with the address of these functions. Next, we manipulate the call to these functions by modification to the parameters. Take `ioremap` function as an example. The first parameter of the function, which is stored in the register `X0`, indicates the target physical address, and we modify the value stored at the register to the corresponding address in the artificial memory region. With this approach, the application never reads the real value of PMU and ETM registers, and cannot be aware of NINJA.

B. Defending Against Timing Attacks

The target application may use the SoC or external timers to detect the time elapsed in the secure domain since the DS affects the performance of the processor and communicates with a human analyst. Note that the TS using ETM does not affect the performance of the processor and thus is immune to the timing attack.

The ARMv8 architecture defines two types of timer components, i.e., the memory-mapped timers and the generic timer registers [34]. Other than these timers, the Juno board is equipped with an additional Real Time Clock (RTC) component PL031 [59] and two dual-timer modules SP804 [60] to measure the time. For each one of these components, we manipulate its value to make the time elapsed of NINJA invisible.

Each of the memory-mapped timer components is mapped to a pre-defined memory region, and all these memory regions are writable in EL3. Thus, we record the value of the timer or counter while entering NINJA and restore it before existing NINJA. The RTC and dual-timer modules are also mapped to a writable memory region, so we use a similar method to handle them.

The generic timer registers consist of a series of timer and counter registers, and all of these registers are writable in EL3

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2018.2883027, IEEE Transactions on Information Forensics and Security

except the physical counter register CNTPCT_EL0 and the virtual counter register CNTVCT_EL0. For the writable registers, we use the same approach as handling memory-mapped timers to manipulate them. Although CNTPCT_EL0 is not directly writable, the ARM architecture requires a memory-mapped counter component to control the generation of the counter value [34]. In the Juno board, the generic counter is mapped to a controlling memory frame 0x2a430000-0x2a43ffff, and writing to the memory address 0x2a430008 updates the value of CNTPCT_EL0. The CNTVCT_EL0 register always holds a value equal to the value of the physical counter register minus the value of the virtual offset register CNTVOFF_EL2. Thus, the update to the CNTPCT_EL0 register also updates the CNTVCT_EL0 register.

Note that the above mechanism only considers the time consumption of NINJA, and does not take the time consumption of the ATF into account. Thus, to make it more precise, we measure the average time consumption of the ATF during the secure exception handling and minus it while restoring the timer values. Besides the timers, the malware may also leverage the PMU to count the CPU cycles. Thus, NINJA checks the enabled PMU counters and restores their values in a similar way to the writable timers.

The external timing attack cannot be defended by modifying the local timer since external timers are involved. As the instruction tracing in NINJA is immune to the timing attack, we can use the TS to trace the execution of the target with DS enabled and disabled. By comparing the trace result using the approaches described in BareCloud [22] and MalGene [21], we may identify the suspicious instructions that launch the attack and defend against the attack by manipulating the control flow in EL3 to bypass these instructions. However, the effectiveness of this approach needs to be further studied. Currently, defending against the external timing attack is an open research problem [19], [24].

VII. EVALUATION

To evaluate NINJA, we fist compare it with existing analysis and debugging tools on ARM. NINJA neither involves any virtual machine or emulator nor uses the detectable Linux tools like ptrace or strace. Moreover, to further improve the transparency, we do not modify Android system software or the Linux kernel. The detailed comparison is listed in Table II. Since NINJA only relies on the ATF, the table shows that the Trusted Computing Base (TCB) of NINJA is much smaller than existing systems.

A. Tracing and Debugging Samples

To evaluate NINJA, we use Android 5.1.1 as the rich OS on ARM Juno Board and Ubuntu 12.04 as the rich OS on NXP i.MX53 QSB. On Juno board, we pick up the *ActivityLifecycle1* sample from DroidBench [63] project and use NINJA to analyze it. We choose this specific sample since it exhibits the representative malicious behavior that leaking sensitive information network connection. In regard to the NXP i.MX53 QSB, we use the data address trace in NINJA to analyze a real-world rootkit, *Suterusu* [64].

Analyzing *ActivityLifecycle1*. To get an overview of the sample, we first enable the Android API tracing feature to inspect the APIs that read sensitive information (source) and APIs that leak information (sink), and find a suspicious API call sequence. In the sequence, the method TelephonyManager.getDeviceId and method HttpURLConnection.connect are invoked in turn, which indicates a potential flow that sends IMEI to a remote server. As we know the network packets are sent via the system call sys_sendto, we attempt to intercept the system call and analyze the parameters of the system call. In Android, the system calls are invoked by corresponding functions in libc.so, and we get the address of the function for the system call sys_sendto by disassembling libc.so. Thus, we use NINJA to set a breakpoint at the address, and the second parameter of the system call, which is stored in register X1, shows that the sample sends a 181 bytes buffer to a remote server. Then, we output the memory content of the buffer and find that it is a HTTP GET request to host www.google.de with path /search?q=353626078711780. Note that the digits in the path is exactly the IMEI of the device.

Analyzing *Suterusu*. Since rootkits usually manipulate the text section of the kernel, we apply the data address trace with an address filter to monitor the modification to this memory region. Moreover, to avoid the noises introduced by other processes, we use the CID_WRITE_RETIRED event of PMU to monitor the switch of executing processes. Once *Suterusu* becomes the current process, we enable the data address trace with the corresponding address range and context ID filter. In our experiment, the text section of the kernel is from 0x80032000 to 0x8082A000, and the data address trace helps us find that *Suterusu* attempts to write to a series of addresses included in this range. For example, it aims to write 12 bytes starting from 0x800C9D74. By checking the system symbols, we learn that this address is the start address of function sys_read, which means that *Suterusu* hooks this kernel function by modifying the first several instructions of the function. Our experiment also reveals that some other kernel functions are hooked in the same way.

B. Transparency Experiments

1) Accessing System Instruction Interface: To evaluate the protection mechanism of the system instruction interface, we write an Android application that reads the PMCR_EL0 and PMCNTENSET_EL0 registers via MRS instruction. The values of these two registers represent whether a performance counter is enabled. We first use the application to read the registers with NINJA disabled and check the value of the PMCR_EL0 and PMCNTENSET_EL0 registers which reflect the enable status of the performance counters. At this moment, the value of these registers indicates that the performance counters are disabled. Then we press a GPIO button to enable the Android API tracing feature of NINJA and read the registers again. As the access to the registers is trapped into EL3 and the artificial values are provided, the values of these registers still show that the performance counters are disabled. This experiment shows that NINJA effectively eliminates the footprint on the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2018.2883027, IEEE Transactions on Information Forensics and Security

Table II: Comparing with Other Tools. The source lines of code (SLOC) of the TCB is calculated by `sloccount` [61] based on Android 5.1.1 and Linux kernel 3.18.20.

ATF = ARM Trusted Firmware, AOS = Android OS, LK = Linux Kernel

	NINJA	TaintDroid [2]	TaintART [4]	DroidTrace [8]	CrowDroid [15]	DroidScope [6]	CopperDroid [5]	NDroid [62]
No VM/emulator	✓	✓	✓	✓	✓			
No ptrace/strace	✓	✓	✓			✓	✓	✓
No modification to Android	✓			✓	✓	✓	✓	✓
Analyzing native instruction	✓			✓	✓	✓	✓	✓
Trusted computing base	ATF	AOS + LK	AOS + LK	LK	LK	QEMU	QEMU	QEMU
SLOC of TCB (K)	27	56,355	56,355	12,723	12,723	489	489	489

system instruction interface, and detailed experiment steps can be found in [28].

2) Accessing Memory Mapped Interface: In this section, we take `ioremap` function as an example to evaluate whether the interception to the memory-mapping functions works. As the `ioremap` function can be called only in the kernel space, we write a kernel module that remaps the memory region of the ETM by the `ioremap` function, and print the content of the first 32 bytes in the region. Similar to the approach discussed above, we first load the kernel module with NINJA disabled and check the output memory content. Next, we enable the instruction tracing feature of NINJA and reload the kernel module to output the memory content again. In our experiment, these two memory contents remain the same with help of the artificial memory regions. This experiment shows that we successfully hide the ETM status change to the normal domain, and NINJA remains transparent. Detailed experiment steps can be found in [28].

3) Adjusting the Timers: To evaluate whether our mechanism that modifies the local timers works, we write a simple application that launches a dummy loop for 1 billion times, and calculate the execution time of the loop by the return values of the API call `System.currentTimeMillis()`. In the first experiment, we record the execution time with NINJA disabled, and the average time for 30 runs is 53.16s with a standard deviation 2.97s. In the second experiment, we enable the debugging mode of NINJA and pause the execution during the loop by pressing the GPIO button. To simulate the manual analysis, we send a command `rr` to output all the general purpose registers and then read them for 60s. Finally, a command `c` is sent to resume the execution of the target. We repeat the second experiment with the timer adjusting feature of NINJA enabled and disabled for 30 times each, and record the execution time of the loop. The result shows that the average execution time with timer adjusting feature disabled is 116.33s with a standard deviation 2.24s, and that with timer adjusting feature enabled is 54.33s with a standard deviation 3.77s. As the latter result exhibits similar execution time with the original system, the malware cannot use the local timer to detect the presence of the debugging system.

C. Performance Evaluation

1) Trace Subsystem: We evaluate the performance overhead of the trace subsystem due to its automation characteristic.

Table III: Time Consumption of Calculating 1 Million Digits of π on i.MX53 QSB.

	Mean	STD	95% CI	Slowdown
Base: Tracing disabled	8.682s	0.072s	[8.656s, 8.708s]	
Instruction tracing	8.745s	0.039s	[8.731s, 8.760s]	$\sim$ 1x
System call tracing	8.745s	0.029s	[8.735s, 8.756s]	$\sim$ 1x
Data address tracing	8.748s	0.044s	[8.733s, 8.764s]	$\sim$ 1x

Performance overhead of the debugging subsystem is not noticed by an analyst in front of the command console, and the debugging system is designed with human interaction.

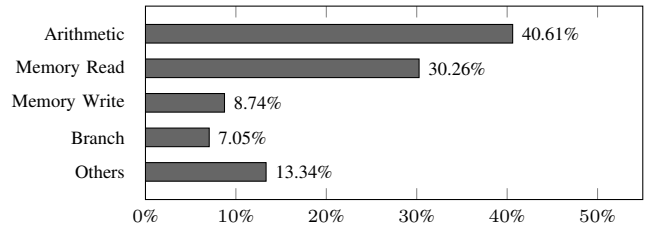

Figure 5: Instruction Distribution of Calculating π.

To learn the performance overhead on the Linux binaries, we build an executable that using an open source π calculation algorithm provided by the GNU Multiple Precision Arithmetic Library [26] to calculate 1 million digits of the π for 30 times on i.MX53 QSB. The time consumptions of the π calculation with and without the tracing functions are shown in Table III. Note that the Android API tracing is not available since we use Ubuntu as the rich OS on this board. As the table shows, the time consumption of the π calculation with tracing disabled is 8.682s, and that of the calculation with instruction tracing enabled, system call tracing enabled, and data address tracing enabled is 8.745s, 8.745s, and 8.748s, respectively. Even in the worst case, the overhead of the ETM-based tracing solution is less than 0.1%. For better understanding of the overhead, we also use the TS to measure the distribution of different instructions executed during the π calculation. As shown in Figure 5, 40.61% of the executed instructions are arithmetic instructions. The percentage of memory read instructions and memory write instructions is 30.26% and 8.74%, respectively. The branch instructions take 7.05 percent of the executed instructions and the percentage of the other instructions is 13.34%.

To measure the performance overhead on the Android applications, we use CF-Bench [65] downloaded from Google

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2018.2883027, IEEE Transactions on Information Forensics and Security

Table IV: The TS Performance Evaluation with CF-Bench [65] on Juno Board.

	Native Scores			Java Scores			Overall Scores		
	Mean	STD	Slowdown	Mean	STD	Slowdown	Mean	STD	Slowdown
Base: Tracing disabled	25380	1023		18758	1142		21407	1092	
Instruction tracing	25364	908	$\sim$ 1x	18673	1095	$\sim$ 1x	21349	1011	$\sim$ 1x
System call tracing	25360	774	$\sim$ 1x	18664	1164	$\sim$ 1x	21342	911	$\sim$ 1x
Android API tracing	6452	24	$\sim$ 4x	122	4	$\sim$ 154x	2654	11	$\sim$ 8x

Play Store. The CF-Bench focuses on measuring both the Java performance and native performance in Android system, and we use it to evaluate the overhead for 30 times on Juno board. Since the data address tracing is not supported on this board, we eliminate it from the experiments. The result in Table IV shows that the overheads of instruction tracing and system call tracing are sufficiently small to ignore. The Android API tracing brings 4x slowdown on the native score and 154x slowdown on the Java score, and the overall slowdown is 8x. This overhead is mainly due to the frequent domain switch during the execution and bridging the semantic gap. To reduce the overhead, we can combine ETM instruction trace with data trace, and leverage the trace result to rebuild the semantic information and API usage offline. Note that we make these benchmarks to be executed only on Cortex-A57 core 0 by setting their CPU affinity mask to 0x1 since NINJA only stays in that core.

2) System Restoration: To measure the performance of the system restoration on the i.MX53 QSB, we use the cycle counter register to count the clock cycles consumed by the restoration. Due to the selective memory restoration mechanism, the restored memory size varies during different analysis sessions. Thus, we also measure the memory restoration time of different changed memory size. Each restoration process is repeated for 100 times, and the average time consumption is reported in Table V. Compared with Bolt [25], the result in Table V shows that the domain switch time and context restoration time of NINJA are slower, which is caused by different testbeds and the restoration of additional registers. However, considering the time consumption of memory restoration and file system restoration takes most of the time during the restoration process, the time used for domain switch and context restoration is ignorable. The full memory restoration times of NINJA and Bolt are similar, but the selective memory restoration makes NINJA 462x faster than Bolt when the size of changed memory is small (1MB). In regard to the file system restoration, NINJA takes 24ms to switch the file system, which is 18x faster than the file system restoration in Bolt. Note that NINJA still needs to restore the restore the dirty file system on the remote server after the file system switching. However, this can be done synchronously with the next malware analysis session since the restoration is performed on the remote server, and the analyst does not need to wait for the restoration. Considering that the memory restoration time takes 85% percentage of the whole system restoration time in Bolt, we are also interested in size of the changed memory during the execution of a program. Specifically, we use the data address trace to record the target address of all memory write instructions during the execution, and further calculate the changed memory size by these addresses. We use this approach to learn the changed memory size of calculating 1 million digits of π, and the result shows that only about 7.96MB memory has been changed during the execution. STREAM [27], a dedicated memory benchmark, is also used in this experiment, and the execution of the benchmark changes 114MB memory in total. This result shows that the actual size of changed memory during the program execution is a small portion of the whole memory, and selective memory restoration would be much more efficient than the full memory restoration.

Table V: Time Consumption of System Restoration (in μs). The symbol \ means that the related data is not reported by Bolt [25].

	Ninja	Bolt [25]
Domain Switch	6.6	1.2
Selective Memory Restoration (1MB)	4,618	\
Selective Memory Restoration (16MB)	75,562	\
Selective Memory Restoration (256MB)	1,214,709	\
Full Memory Restoration (448MB)	2,135,161	2,445,271
File System Restoration	24,351	433,917
Context Restoration	42	23

VIII. DISCUSSION

NINJA leverages existing deployed hardware and is compatible with commercial mobile devices. However, the secure domain on the commercial mobile devices is managed by the Original Equipment Manufacturer (OEM). Thus, it requires cooperation from the OEMs to implement NINJA on a commercial mobile device.

The approach we used to fill the semantic gaps relies on the understanding of the kernel data structures and memory maps, and thus is vulnerable to the privileged malware. Patagonix [66] leverages a database of whitelisted applications binary pages to learn the semantic information in the memory pages of the target application. However, this approach is limited by the knowledge of the analyzer. Currently, how to transparently bridge the semantic gap without any assumption to the system is still an open research problem [46].

The protection mechanism mentioned in Section VI-A helps to improve transparency when the attackers try to use PMU or ETM registers, and using shadow registers [67] can further protect the critical system registers. However, if an advanced attacker intentionally uses PMU or ETM to trace CPU events or instructions and checks whether the trace result matches the expected one, the mechanism of returning artificial or shadow register values may not provide accurate result and affects

transparency. To address this problem, we need to virtualize the PMU and ETM, and this is left as our future work.

Though NINJA protects the system-instruction interface access to the registers, the mechanism we used to protect the memory mapped interface access maybe vulnerable to advanced attacks such as directly manipulating the memory-mapping, disabling MMU to gain physical memory access, and using DMA to access memory. Note that these attacks might be difficult to implement in practice (e.g., disabling MMU might crash the system). To fully protect the memory-mapped region of ETM and PMU registers, we would argue that hardware support from TrustZone is needed. Since the TZASC only protects the DRAM, we may need additional hardware features to extend the idea of TZASC to the whole physical memory region.

Although the instruction skid of the PMI cannot be completely eliminated, we can also enable ETM between two PMIs to learn the instructions in the skid. Moreover, since the instruction skid is caused by the delay of the PMI, similar hardware component like Local Advanced Programmable Interrupt Controller [24] on x86 which handles interrupt locally may help to mitigate the issue by reducing the response time.

IX. CONCLUSIONS

In this paper, we present NINJA, a transparent malware analysis framework on ARM platform. It embodies a series of analysis functionalities like tracing and debugging via hardware-assisted isolation execution environment TrustZone and hardware features PMU and ETM. Since NINJA does not involve emulator or framework modification, it is more transparent than existing analysis tools on ARM. To minimize the artifacts introduced by NINJA, we adopt register protection mechanism to protect all involving registers based on hardware traps and runtime function interception. Moreover, as the TrustZone and the hardware components are widely equipped by OTS mobile devices, NINJA can be easily transplanted to existing mobile platforms. A fast restoration mechanism is also implemented in NINJA to facilitate the continuous malware analysis. Our experiment results show that performance overheads of the instruction tracing and system call tracing are less than 1% while the Android API tracing introduces 4 to 154 times slowdown.

REFERENCES

[1] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro, "DroidScribe: Classifying Android malware based on runtime behavior," *Mobile Security Technologies (MoST'16)*, 2016.

[2] Enck, William and Gilbert, Peter and Cox, Landon P and Jung, Jaeyeon and McDaniel, Patrick and Sheth, Anmol N, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, 2010.

[3] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid Android: Versatile protection for smartphones," in *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC'10)*, 2010.

[4] M. Sun, T. Wei, and J. Lui, "TaintART: a practical multi-level information-flow tracking system for Android RunTime," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*, 2016.

[5] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic reconstruction of Android malware behaviors," in *Proceedings of 22nd Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[6] Yan, Lok Kwong and Yin, Heng, "Droidscope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis," in *Proceedings of the 21st USENIX Security Symposium (USENIX Security'12)*, 2012.

[7] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in Android apps with permission use analysis," in *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*, 2013.

[8] Zheng, Min and Sun, Mingshen and Lui, John CS, "DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability," in *2014 International Wireless Communications and Mobile Computing Conference (IWCMC'14)*, 2014.

[9] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu, "Morpheus: automatically generating heuristics to detect Android emulators," in *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14)*, 2014.

[10] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: hindering dynamic analysis of Android malware," in *Proceedings of the 7th European Workshop on System Security (EurSec'14)*, 2014.

[11] T. Vidas and N. Christin, "Evading Android runtime analysis via sandbox detection," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (AsiaCCS'14)*, 2014.

[12] C. Dall and J. Nieh, "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'14)*, 2014.

[13] Xen project., "Xen ARM with virtualization extensions," https://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions.

[14] H. Shi, A. Alwabel, and J. Mirkovic, "Cardinal pill testing of system virtual machines," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)*, 2014.

[15] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for Android," in *Proceedings of the 1st ACM workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'11)*, 2011.

[16] Anubis, "Analyzing Unknown Binaries," http://anubis.iseclab.org.

[17] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis," in *Proceedings of the 4th International Conference on Information Systems Security (ICISS'08)*, 2008.

[18] Z. Deng, X. Zhang, and D. Xu, "Spider: Stealthy binary program instrumentation and debugging via hardware virtualization," in *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC'13)*, 2013.

[19] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, 2008.

[20] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system," in *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14)*, 2014.

[21] Kirat, Dhilung and Vigna, Giovanni, "MalGene: Automatic extraction of malware analysis evasion signature," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[22] D. Kirat, G. Vigna, and C. Kruegel, "Barecloud: Bare-metal analysis-based evasive malware detection," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)*, 2014.

[23] C. Spensky, H. Hu, and K. Leach, "LO-PHI: Low-observable physical host instrumentation for malware analysis," in *Proceedings of 23rd Network and Distributed System Security Symposium (NDSS'16)*, 2016.

[24] F. Zhang, K. Leach, A. Stavrou, and H. Wang, "Using hardware features for increased debugging transparency," in *Proceedings of The 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015, pp. 55–69.

[25] L. Guan, S. Jia, B. Chen, F. Zhang, B. Luo, J. Lin, P. Liu, X. Xing, and L. Xia, "Supporting transparent snapshot for bare-metal malware analysis on mobile devices," in *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC'17)*, 2017.

[26] The GNU Multiple Precision Arithmetic Library, "Pi with GMP," https://gmplib.org/.

[27] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 1995.

[28] Z. Ning and F. Zhang, "Ninja: Towards transparent tracing and debugging on ARM," in *Proceedings of 26th USENIX Security Symposium (USENIX Security'17)*, 2017.

[29] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of 31st IEEE Symposium on Security and Privacy (S&P'10)*, 2010.

[30] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache attacks on mobile devices," in *Proceedings of the 25th USENIX Security Symposium (USENIX Security'16)*, 2016.

[31] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[32] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[33] ARM Ltd., "TrustZone Security Whitepaper," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html.

[34] ——, "ARMv8-A Reference Manual," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0487a.k/index.html.

[35] ——, "ARM Trusted Firmware," https://github.com/ARM-software/arm-trusted-firmware.

[36] M. Spisak, "Hardware-assisted rootkits: Abusing performance counters on the ARM and x86 architectures," in *Proceedings of the 10th USENIX Workshop on Offensive Technologies (WOOT'16)*, 2016.

[37] ARM Ltd., "Embedded Trace Macrocell Architecture Specification," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0014q/index.html.

[38] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna, "BareDroid: Large-scale analysis of Android apps on real devices," in *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)*, 2015.

[39] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, "Malton: Towards on-device non-Invasive mobile malware analysis for ART," in *Proceedings of 26th USENIX Security Symposium (USENIX Security'17)*, 2017.

[40] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of Android apps," in *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE'17)*, 2017.

[41] Z. Ning and F. Zhang, "DexLego: Reassembleable bytecode extraction for aiding static analysis," in *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'18)*, 2018.

[42] D. Kirat, G. Vigna, and C. Kruegel, "Barebox: Efficient malware analysis on bare-metal," in *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC'11)*, 2011.

[43] ARM Ltd., "ARM Generic Interrupt Controller Architecture Specification," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0048b/index.html.

[44] H. Sun, K. Sun, Y. Wang, and J. Jing, "TrustOTP: Transforming smartphones into secure one-time password tokens," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[45] D. Zhang, "TrustZone project with Linux 2.6.35 and U-Boot (i.MX53QSB)," https://github.com/finallyjustice/liboot-tz.

[46] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, "Sok: Introspections on trust and the semantic gap," in *Proceedings of 35th IEEE Symposium on Security and Privacy (S&P'14)*, 2014.

[47] C.-C. Hwang, "ARM PTM decoder, and ARM ETM v4 decoder," https://github.com/hwangcc23/ptm2human.

[48] A. Shishkin, "ARM's ETM v3 decoder," https://github.com/virtuoso/etm2human.

[49] ARM Ltd., "ARM CoreLink NIC-400 Network Interconnect Technical Reference Manual," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0475g/index.html.

[50] X. Wang and J. Backer, "SIGDROP: Signature-based ROP detection using hardware performance counters," *arXiv preprint arXiv:1609.02667*, 2016.

[51] E. Israel, D. Marx, Y. Alon, A. Gafni, and B. Omelchenko, "Detection of the Meltdown and Spectre vulnerabilities," https://research.checkpoint.com/detection-meltdown-spectre-vulnerabilities-using-checkpoint-cpu-level-technology/.

[52] Capsule8, "Part two: Detecting Meltdown and Spectre by detecting cache side channels," https://capsule8.com/blog/detecting-meltdown-spectre-detecting-cache-side-channels/.

[53] Linux Programmer's Manual, "pivot root," http://man7.org/linux/man-pages/man2/pivot_root.2.html.

[54] IBM Knowledge Center, "Steps for dynamically replacing the sysplex root file system," https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.bpxb200/repcurr.htm.

[55] H. . https://docs.oracle.com/cd/E37670_01/E37355/html/ol_use_case3_btrfs.html. Oracle, "Creating Subvolumes and Snapshots."

[56] Amazon, "Elastic File System," https://aws.amazon.com/efs/.

[57] S. Vogl and C. Eckert, "Using hardware performance events for instruction-level monitoring on the x86 architecture," in *Proceedings of the 2012 European Workshop on System Security (EuroSec12)*, 2012.

[58] R. Yu, "Android packers: facing the challenges, building solutions," in *Proceedings of the Virus Bulletin Conference (VB'14)*, 2014.

[59] ARM Ltd., "ARM PrimeCell Real Time Clock Technical Reference Manual," http://infocenter.arm.com/help/topic/com.arm.doc.ddi0224b/DDI0224.pdf.

[60] ——, "ARM Dual-Timer Module (SP804) Technical Reference Manual," http://infocenter.arm.com/help/topic/com.arm.doc.ddi0271d/DDI0271.pdf.

[61] Ubuntu, "sloccount," http://manpages.ubuntu.com/manpages/precise/man1/compute_all.1.html.

[62] C. Qian, X. Luo, Y. Shao, and A. T. Chan, "On tracking information flows through jni in android applications," in *Proceedings of The 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14)*, 2014.

[63] EC SPRIDE Secure Software Engineering Group, "DroidBench," https://github.com/secure-software-engineering/DroidBench.

[64] M. Coppola, "Suterusu rootkit: Inline kernel function hooking on x86 and ARM," https://poppopret.org/2013/01/07/suterusu-rootkit-inline-kernel-function-hooking-on-x86-and-arm/.

[65] Chainfire, "CF-Bench," https://play.google.com/store/apps/details?id=eu.chainfire.cfbench.

[66] L. Litty, H. A. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries," in *Proceedings of the 17th USENIX Security Symposium (USENIX Security'08)*, 2008.

[67] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, 2007.