Ninja: Towards Transparent Tracing and Debugging on ARM

Zhenyu Ning and Fengwei Zhang
Wayne State University
$\{$*zhenyu.ning, fengwei*$\}$*@wayne.edu*

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. 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.

1 Introduction

Malware on the mobile platform exhibits an explosive growth in recent years. To solve the threat of the malicious applications, a variety of tools have been proposed for malware detection and analysis [18, 22, 37, 44, 45, 52, 55, 56]. 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 [18, 45, 52] are based on emulation or virtualization technology, a series of anti-emulation and anti-virtualization techniques [29, 36, 48] have been developed to challenge them. These techniques show that the emulation or virtualization can be easily detected by footprints like string properties, absence of particular hardware components, and performance slowdown. The hardware-assisted virtualization technique [17, 50] can improve the transparency of the virtualization-based systems; however, this approach still leaves artifacts on basic instruction execution semantics that could be easily detected by malware [39].

To address this challenge, researchers study the malware on bare-metal devices via modifying the system software [22, 37, 44, 55] or leveraging OS APIs [15, 56] 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 [2, 40] to hardware-assisted virtualization analysis systems [19, 20, 32], and then to bare-metal analysis systems [30, 31, 41, 54]. 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 [29, 36, 39, 48] 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 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.

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. Moreover, we evaluate the instruction-skid problem and show that it has little influence on our system.

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

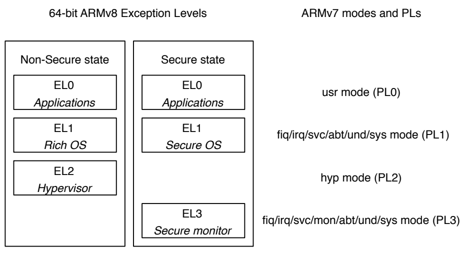


Figure 1: The ARMv8 and ARMv7 Architectures.

- 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 and the instruction and system call tracing is immune to timing attacks.

2 Background

2.1 TrustZone and Trusted Firmware

ARM TrustZone technology [12] 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 [8], and the exception return instruction `eret` is used to switch back to the normal domain from the secure domain after the exception is handled.

Figure 1 shows the difference between the ARMv8 and the ARMv7 architectures. In the new architecture, ARM removes the execution modes in ARMv7 and renames the Privilege Level (PL) to Exception Level (EL). The term EL indicates the level where an exception can be handled and all ELs except EL0 can handle exceptions. Any exception occurs in a certain level could only be handled in the same level or a higher level.

The names of the system registers in 64-bit ARMv8 architecture contain a suffix that indicating the lowest EL at which the register can be accessed. For example, the name of the `PMEVCNTR_EL0` register indicates that the lowest EL to access this register is EL0. Similarly, the registers with suffix `EL3` can only be accessed in EL3.

ARM Trusted Firmware [7] (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.

2.2 PMU and ETM

The Performance Monitors Unit (PMU) [8] is a feature widely implemented in both x86 and ARM architectures [42], 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 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) [11] 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.

According to Futuremark [23], 21 of the most popular 50 smartphones and tablets are equipped with ARM Cortex-A5x or Cortex-A7x series processors, in which the PMU and ETM components are included.

3 Related Work

3.1 Transparent Malware Analysis on x86

Ether [20] leverages hardware virtualization to build a malware analysis system and achieves high transparency. Spider [19] is also based on hardware virtualization, and it focuses on both applicability and transparency while using memory page instrument to gain higher efficiency. Since the hardware virtualization has transparency issues, these systems are naturally not transparent. LO-PHI [41] 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 [54] increases the transparency by involving System Manage Mode (SMM), a special CPU mode in x86 architecture. It leverages PMU to monitor the program execution and switch into SMM for analysis. Comparing with MalT, NINJA improves in the following aspects: 1) The PMU registers on MalT are accessible by privileged malware, which breaks the transparency by checking the values of these registers. By leveraging TrustZone technology, NINJA configures needed PMU registers as secure ones so that even the privileged malware in the normal domain cannot access them. 2) MalT is built on SMM. However, SMM is not designed for security purpose such as transparent debugging (originally for power management); frequent CPU mode switching introduces a high performance overhead (12 μs is required for a SMM switch [54]). NINJA is based on TrustZone, a dedicated security extension on ARM. The domain switching only needs 0.34 μs (see Appendix B). 3) Besides a debugging system, NINJA develops a transparent tracing system with existing hardware. The instruction and system call tracing introduce negligible overhead, which is immune to timing attacks while MalT suffers from external timing attack.

BareCloud [31] and MalGene [30] focus on detecting evasive malware by executing malware in different environments and comparing their behavior. There are limitations to this approach. Firstly, it fails to transparently fetch the malware runtime behavior (e.g., system calls and modifications to memory/registers) on a bare-metal environment. Secondly, it assumes that the evasive malware shows the malicious behavior in at least one of the analysis platforms. However, sophisticated malware may be able to detect all the analysis platforms and refuse to exhibit any malicious behavior during the analysis. Lastly, after these tools identify the evasive malware from the large-scale malware samples, they still need a transparent malware analysis tool which is able to analyze these evasive samples transparently. NINJA provides a transparent framework to study the evasive malware and plays a complementary role for these systems.

3.2 Dynamic Analysis Tools on ARM

Emulation-based systems. DroidScope [52] rebuilds the semantic information of both the Android OS and the Dalvik virtual machine based on QEMU. CopperDroid [45] is a VMI-based analysis tool that automatically reconstructs the behavior of Android malware including inter-process communication (IPC) and remote procedure call interaction. DroidScibe [18] uses CopperDroid [45] to collect behavior profiles of Android malware, and automatically classifies them into different families. Since the emulator leaves footprints, these systems are natural not transparent.

Hardware virtualization. Xen on ARM [50] migrates the hardware virtualization based hypervisor Xen to ARM architecture and makes the analysis based on hardware virtualization feasible on mobile devices. KVM/ARM [17] 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 [39].

Bare-metal systems. TaintDroid [22] 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 [44] extends the idea of TaintDroid on the most recent Android Java virtual machine Android Runtime (ART). VetDroid [55] reconstructs the malicious behavior of the malware based on permission usage, and it is applicable to taint analysis. DroidTrace [56] uses `ptrace` to monitor the dynamic loading code on both Java and native code level. BareDroid [34] provides a quick restore mechanism that makes the bare-metal analysis of Android applications feasible at scale. 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.

3.3 TrustZone-related Systems

TZ-RKP [13] runs in the secure domain and protects the rich OS kernel by event-driven monitoring. Sprobes [51] provides an instrumentation mechanism to introspect the rich OS from the secure domain, and guarantees the kernel code integrity. SeCReT [28] is a framework that enables a secure communication channel between the normal domain and the secure domain, and provides a trust execution environment. Brasser *et al*. [14] use TrustZone to analyze and regulate guest devices in a restricted host spaces via remote memory operation to avoid misusage of sensors and peripherals. C-FLAT [1] fights against control-flow hijacking via runtime control-flow verification in TrustZone. TrustShadow [25] shields the execution of an unmodified application from a compromised operating system by building a lightweight runtime system in the ARM TrustZone secure world. The runtime system forwards the requests of system services to the commodity operating systems in the normal world and verifies the returns. Unlike previous systems, NINJA leverage TrustZone to transparently debug and analyze the ARM applications and malware.

4 System Architecture

Figure 2 shows the architecture of NINJA. The NINJA consists of a target executing platform and a remote debugging client. In the target executing platform, Trust-

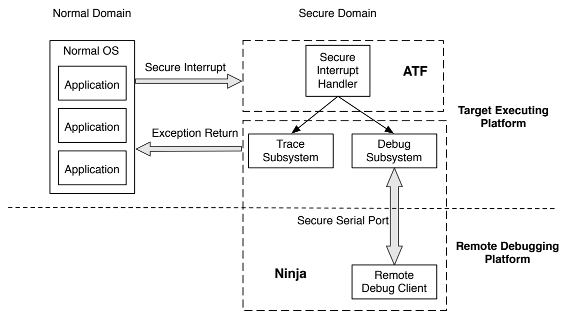


Figure 2: Architecture of NINJA.

Zone 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 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 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 response 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`.

4.1 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) [5] 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 [43] 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`.

4.2 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, and Android API tracing. We achieve the instruction and system call tracing via hardware component ETM, and the Android API tracing with help of PMU registers.

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 configure the ETM to use the signal of PMU event `EXC_SVC` as the external input.

4.3 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.

5 Design and Implementation

We implement NINJA on a 64-bit ARMv8 Juno r1 board. There are two ARM Cortex-A57 cores and four ARM Cortex-A53 cores on the board, and all of them include the support for 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. Note that NINJA is compatible with commercial mobile devices because it relies on existing deployed hardware features.

5.1 Bridge the Semantic Gap

As with the VMI-based [27] and TEE-based [54] 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.

5.1.1 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 running 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 running process in the OS.

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

5.1.2 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 [52] 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

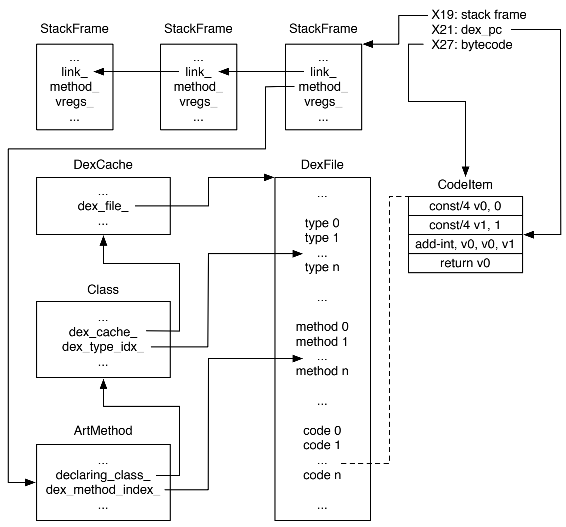


Figure 3: Semantics in the Function `ExecuteGotoImpl`.

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 manual 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 3 shows the semantics in the function `ExecuteGotoImpl`. By combining registers `X21` and `X27`, we can locate the current 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.

5.2 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 4.1, 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`.

5.3 The Trace Subsystem

5.3.1 Instruction Tracing

NINJA uses ETM embedded in the CPU to trace the executed instructions. Figure 4 shows the ETM and related components in Juno board. The funnels shown in the figure are used to filter the output of ETM, and each of them is controlled by a group of CoreSight Trace Funnel (CSTF) registers [9]. The filtered result is then output to Embedded Trace FIFO (ETF) which is controlled by Trace Memory Controller (TMC) registers [10].

In our case, as we only need the trace result from the core 0 in the Cortex-A57 cluster, we set the `EnS0` bit in CSTF Control Register of funnel 0 and funnel 2, and clear other slave bits. To enable the ETF, we set the `TraceCaptEn` bit of the TMC CTL register.

The ETM is controlled by a group of trace registers. 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 stop the trace, we first clear the `EN` bit of `TRCPRGCTLR` register to disable

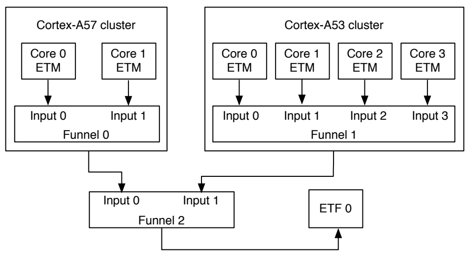


Figure 4: ETM in Juno Board.

ETM and then set the `StopOnFl` bit and the `FlushMan` bits of `FFCR` register in the TMC registers to stop the ETF. To read the trace result, we keep reading from `RRD` register until `0xFFFFFFFF` is fetched. Note that the trace result is an encoded trace stream, and we use an open source analyzer `ptm2human` [26] to convert the stream to a readable format.

5.3.2 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 4.2, 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 5.3.1.

5.3.3 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 5.1.2, each Java method is interpreter 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 5.1.2 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.

5.4 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 [3]. 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 table in Appendix A shows the supported debugging commands. The information about symbols in both bytecode and machine code are not supported at this moment, and we consider it as our future work.

5.4.1 Single-instruction Stepping

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 a 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 5.3.3. With the configuration, NINJA pauses the execution of the target after the execution of each instruction and waits for the debugging commands.

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.

5.4.2 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 [54] 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.

5.4.3 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 to 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 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.

5.5 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 5 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 5, the current 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.

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

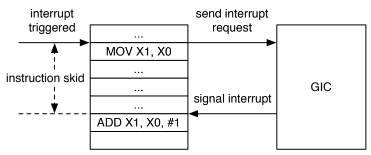


Figure 5: Interrupt Instruction Skid.

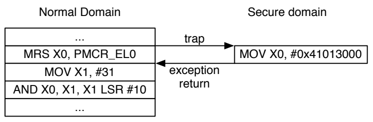


Figure 6: Protect the `PMCR_EL0` Register via Traps.

The skid problem is a well-known problem [42, 49] and affects NINJA since the current 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 the processor. Although the skid problem cannot be completely eliminated, the side-effect of the skid does not affect our system significantly, and we provide a detailed analysis and evaluation in Section 7.5.

6 Transparency

As NINJA is not based on the emulator or other sandboxes, the anti-analysis techniques mentioned in [29, 36, 48] cannot detect the existence of NINJA. Moreover, other anti-debugging techniques like anti-ptrace [53] 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.

6.1 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.

6.1.1 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 6 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`).

6.1.2 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 TrustZone Address Space Controller (TZASC) [12], 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 0x9ffffffff 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.

6.2 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 [8]. Other than these timers, the Juno board is equipped with an additional Real Time Clock (RTC) component PL031 [6] and two dual-timer modules SP804 [4] 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 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 [8]. 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 (see Appendix B) 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 [31] and MalGene [30], 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 [20, 54].

7 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 1. Since NINJA only relies on the ATF, the table shows that the Trusted Computing Base (TCB) of NINJA is much smaller than existing systems.

7.1 Output of Tracing Subsystem

To learn the details of the tracing output, we write a simple Android application that uses Java Native Interface to read the /proc/self/status file line by line (which can be further used to identify whether ptrace is enabled) and outputs the content to the console. We use instruction trace of the TS to trace the execution of the application, and also measure the time usage. The status file contains 38 lines in total, and it takes about 0.22 *ms* to finish executing. After the execution, the ETF contains 9.92 *KB* encoded trace data, and the datarate is approximately 44.03 MB/s. Next, we use ptm2human [26] to decode the data, and the decoded trace data contains 1341 signpost instructions (80 in our custom native library and the others in libc.so). By manually introspect the signpost instructions in our custom native library, we can rebuild the whole execution control flow. To reduce the

Table 1: Comparing with Other Tools. The source lines of code (SLOC) of the TCB is calculated by `sloccount` [47] based on Android 5.1.1 and Linux kernel 3.18.20.

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

	NINJA	TaintDroid [22]	TaintART [44]	DroidTrace [56]	CrowDroid [15]	DroidScope [52]	CopperDroid [45]	NDroid [38]
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

storage usage of the ETM, we can use real-time continuous export via either a dedicated trace port capable of sustaining the bandwidth of the trace or an existing interface on the SoC (e.g., a USB or other high-speed port) [11].

7.2 Tracing and Debugging Samples

We pickup two samples `ActivityLifecycle1` and `PrivateDataLeak3` from DroidBench [21] project and use NINJA to analyze them. We choose these two specific samples since they exhibit representative malicious behavior like leaking sensitive information via local file, text message, and network connection.

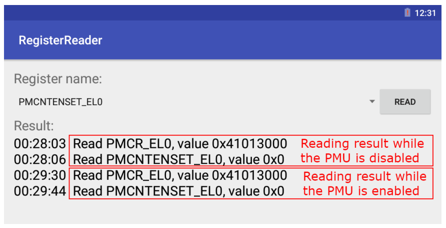
Analyzing *ActivityLifecycle1*. To get an overview of the sample, we first enable the Android API tracing feature to inspect the APIs that read sensitive information (source) and APIs that leak information (sink), and find a suspicious API call sequence. In the sequence, the method `TelephonyManager.getDeviceId` 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 *PrivateDataLeak3*. Similar to the previous analysis, the Android API tracing helps us to find a suspicious API call sequence consisting of the methods `TelephonyManager.getDeviceId`, `Context.openFileOutput`, and `SmsManager.sendTextMessage`. As the Android uses the system calls `sys_openat` to open a file and `sys_write` to write a file, we set breakpoints at the address of these calls. Note that the second parameter of `sys_openat` represents the full path of the target file and the second parameter of `sys_write` points to a buffer writing to a file. Thus, after the breakpoints are hit, we see that sample writing IMEI 353626078711780 to the file `/data/data/de.ecspride/files/out.txt`. The API `SmsManager.sendTextMessage` uses `binder` to achieve IPC with the lower-layer `SmsService` in Android system, and the semantics of the IPC is described in CopperDroid [45]. By intercepting the system call `sys_ioctl` and following the semantics, we finally find the target of the text message "+49" and the content of the message 353626078711780.

7.3 Transparency Experiments

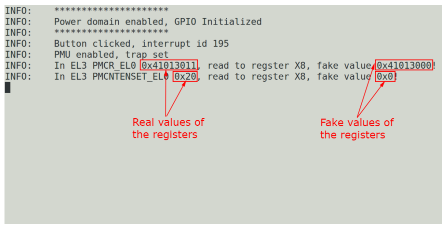
7.3.1 Accessing System Instruction Interface

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 the result is shown in the upper rectangle of Figure 7a. The last bit of the `PMCR_EL0` register and the value of the `PMCNTENSET_EL0` register are 0, which means that all 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. From the console output shown in Figure 7b, we see that the access to the registers is successfully trapped into EL3. And the output shows that the real values of the `PMCR_EL0` and `PMCNTENSET_EL0` registers are 0x41013011 and 0x20, respectively, which indicates that the counter `PMEVCNTR5_EL0` is enabled. However, the lower rectangle in Figure 7a shows that the value of the registers fetched by the application keep unchanged.



```
00:28:03 Read PMCR_EL0, value 0x41013000      Reading result while
00:28:06 Read PMCNTENSET_EL0, value 0x0       the PMU is disabled
00:29:30 Read PMCR_EL0, value 0x41013000      Reading result while
00:29:44 Read PMCNTENSET_EL0, value 0x0       the PMU is enabled
```

(a) Reading PMU Register in an Application.



```
INFO:    *********************
INFO:    Power domain enabled, GPIO Initialized
INFO:    *********************
INFO:    Button clicked, interrupt id 195
INFO:    PMU enabled, trap set
INFO:    In EL3 PMCR_EL0 0x41013011, read to regster X8, fake value 0x41013000!
INFO:    In EL3 PMCNTENSET_EL0 0x20, read to regster X8, fake value 0x0!
```

(b) EL3 Output in the Secure Console.

Figure 7: Accessing System Instruction Interface.

This experiment shows that NINJA effectively eliminates the footprint on the system instruction interface.

7.3.2 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 the output is shown in the upper rectangle in Figure 8a. Note that the 5th to the 8th bytes are mapped as the `TRCPRGCTLR` register and the `EN` bit, which indicates the status of the ETM, is the last bit of the register. In the upper rectangle, the `EN` bit 0 shows that the ETM is disabled. Next, we enable the instruction tracing feature of NINJA and reload the kernel module. The lower rectangle in Figure 8a shows that the content of the memory fetched by the module remains the same. However, in the Figure 8b, the output from EL3 shows that the memory of the ETM has changed. This experiment shows that we successfully hide the ETM status change to the normal domain, and NINJA remains transparent.

7.3.3 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.

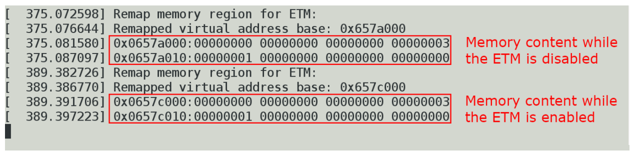
Table 2: The TS Performance Evaluation Calculating 1 Million Digits of π.

	Mean	STD	# Slowdown
Base: Tracing disabled	2.133 s	0.69 ms	
Instruction tracing	2.135 s	2.79 ms	$\sim$ 1x
System call tracing	2.134 s	5.13 ms	$\sim$ 1x
Android API tracing	149.372 s	1287.88 ms	$\sim$70x

7.4 Performance Evaluation

In this section, we evaluate the performance overhead of the trace subsystem due to its automation characteristic. 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.

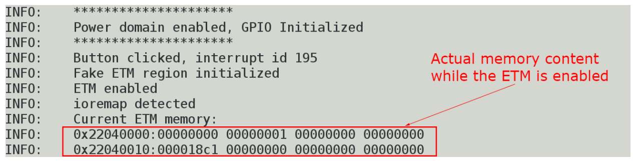
To learn the performance overhead on the Linux binaries, we build an executable that using an open source π calculation algorithm provided by the GNU Multiple Precision Arithmetic Library [46] to calculate 1 million digits of the π for 30 times with the tracing functions disabled and enabled, and the time consumption is shown in Table 2. Since we leverage ETM to achieve the instruction tracing and system call tracing, the experiment result shows that the ETM-based solution has negligible overhead — less than 0.1%. In the Android API tracing, the overhead is about 70x. This overhead is mainly due to the frequent domain switch during the execution and bridging the semantic gap. To reduce the overhead, we



```
[  375.072598] Remap memory region for ETM:
[  375.076644] Remapped virtual address base: 0x657a000
[  375.081580] 0x0657a000:00000000 00000000 00000000 00000003
[  375.087097] 0x0657a010:00000001 00000000 00000000 00000000
[  389.382726] Remap memory region for ETM:
[  389.386770] Remapped virtual address base: 0x657c000
[  389.391706] 0x0657c000:00000000 00000000 00000000 00000003
[  389.397223] 0x0657c010:00000001 00000000 00000000 00000000
```

Memory content while the ETM is disabled
Memory content while the ETM is enabled

(a) Reading ETM Memory Region.



```
INFO:    *********************
INFO:    Power domain enabled, GPIO Initialized
INFO:    *********************
INFO:    Button clicked, interrupt id 195
INFO:    Fake ETM region initialized
INFO:    ETM enabled
INFO:    ioremap detected
INFO:    Current ETM memory:
INFO:    0x22040000:00000000 00000001 00000000 00000000
INFO:    0x22040010:000018c1 00000000 00000000 00000000
```

(b) EL3 Output in the Secure Console.

Figure 8: Memory Mapped Interface.

Table 3: The TS Performance Evaluation with CF-Bench [16].

	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	~ 1x	18673	1095	~ 1x	21349	1011	~ 1x
System call tracing	25360	774	~ 1x	18664	1164	~ 1x	21342	911	~ 1x
Android API tracing	6452	24	~ 4x	122	4	~ 154x	2654	11	~ 8x

can combine ETM instruction trace with data trace, and leverage the trace result to rebuild the semantic information and API usage offline.

To measure the performance overhead on the Android applications, we use CF-Bench [16] downloaded from Google 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. The result in Table 3 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. 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.

7.5 Skid Evaluation

In this subsection, we evaluate the influence of the skid problem to NINJA. Since the instruction tracing, system call tracing, and memory read/write do not involve PMI, these functionalities are not affected by the skid problem. In ART, each bytecode is interpreted as an array of machine code. Our bytecode stepping mechanism recognizes the corresponding bytecode once it is executing any machine code in the array, i.e., the skid problem affects the bytecode stepping if and only if the instruction shadow covers all the machine code for a bytecode. We evaluate the listed 218 bytecode opcode [24] on the Android official website, and it shows that the shadow region cannot cover the machine code for any of them. Thus, the bytecode stepping does not suffer from the skid problem. For a similar reason, the skid problem has no influence on the Android API tracing.

However, the native code stepping and the breakpoint

Table 4: Instructions in the Skid Shadow with Representative PMU Events.

Event Number	Event Description	# of Instructions	
		Mean	STD
0x81-0x8F	Exception related events that firing after taking exceptions	0	0
0x11	CPU cycle event that firing after each CPU cycle	2.73	2.30
0x08	Instruction retired event that firing after executing each instruction	6.03	4.99

are still affected, and both of them use instruction retired event to overflow the counter. Since the skid problem is due to the delay between the interrupt request and the interrupt arrival, we first use PMU counter to measure this delay by CPU cycles. Similar with the instruction stepping, we make the PMU counter to count `CPU_CYCLES` event and initialize the value of the counter to its maximum value. Then, the counter value after switching into EL3 is the time delay of the skid in CPU cycles. The results of 30 experiments show that the delay is about 106.3 CPU cycles with a standard deviation 2.26. As the frequency of our CPU is 1.15GHz, the delay is about 0.09μs. We also evaluate the number of instructions in the skid shadow with some representative PMU events. For each event, we trigger the PMI for 30 times and calculate the mean and standard deviation of the number of instructions in the shadow. Table 4 shows the result with different PMU events. Unlike the work described in [42], the exception related events exhibits no instruction shadow in our platform, and we consider it is caused by different ARM architectures. It is worth noting that the number of instructions in the skid shadow of the CPU cycle event is less than the instruction retired event. However, using the CPU cycle event may lead to multiple PMIs for a single instruction since the execution of a single instruction may need multiple CPU cycles, which introduces more performance overhead but with more fine-grained instruction-stepping. In practice, it is a trade off between the performance overhead and the debugging accuracy, and we can use either one based on the requirement.

8 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 [33] 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 [27].

The protection mechanism mentioned in Section 6.1 helps to improve transparency when the attackers try to use PMU or ETM registers, and using shadow registers [35] 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 thus affects NINJA's transparency. To address this problem, we need to fully 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 [54] on x86 which handles interrupt locally may help to mitigate the problem by reducing the response time.

9 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. 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.

10 Acknowledgements

We would like to thank our shepherd, Manuel Egele, and the anonymous reviewers for their valuable comments and feedback. Special thanks to He Sun, who offers early discussion about the project. We also appreciate Saeid Mofrad, Leilei Ruan, and Qian Jiang for their kindly review and helpful suggestions.

References

[1] ABERA, T., ASOKAN, N., DAVI, L., EKBERG, J.-E., NYMAN, T., PAVERD, A., SADEGHI, A.-R., AND TSUDIK, G. C-FLAT: Control-flow attestation for embedded systems software. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS'16)* (2016).

[2] ANUBIS. Analyzing Unknown Binaries. `http://anubis.iseclab.org`.

[3] 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`.

[4] ARM LTD. ARM Dual-Timer Module (SP804) Technical Reference Manual. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0271d/DDI0271.pdf`.

[5] ARM LTD. ARM Generic Interrupt Controller Architecture Specification. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0048b/index.html`.

[6] ARM LTD. ARM PrimeCell Real Time Clock Technical Reference Manual. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0224b/DDI0224.pdf`.

[7] ARM Ltd. ARM Trusted Firmware. https://github.com/ARM-software/arm-trusted-firmware.

[8] ARM Ltd. ARMv8-A Reference Manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0487a.k/index.html.

[9] ARM Ltd. CoreSight Components Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0314h/DDI0314H_coresight_components_trm.pdf.

[10] ARM Ltd. CoreSight Trace Memory Controller Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0461b/DDI0461B_tmc_r0p1_trm.pdf.

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

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

[13] Azab, A. M., Ning, P., Shah, J., Chen, Q., Bhutkar, R., Ganesh, G., Ma, J., and Shen, W. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS'14)* (2014).

[14] Brasser, F., Kim, D., Liebchen, C., Ganapathy, V., Iftode, L., and Sadeghi, A.-R. Regulating ARM TrustZone devices in restricted spaces. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'16)* (2016).

[15] Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. 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] Chainfire. CF-Bench. https://play.google.com/store/apps/details?id=eu.chainfire.cfbench.

[17] Dall, C., and Nieh, J. KVM/ARM: The design and implementation of the linux ARM hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)* (2014).

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

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

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

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

[22] 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 (OSDI10)* (2010).

[23] Futuremark. Best Smartphones. http://www.futuremark.com/hardware/mobile.

[24] Google Inc. Dalvik bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html.

[25] Guan, L., Liu, P., Xing, X., Ge, X., Zhang, S., Yu, M., and Jaeger, T. TrustShadow: Secure execution of unmodified applications with ARM trustzone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'17)* (2017).

[26] Hwang, C.-C. ptm2human. https://github.com/hwangcc23/ptm2human.

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

[28] Jang, J. S., Kong, S., Kim, M., Kim, D., and Kang, B. B. SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment. In *Proceedings of 22nd Network and Distributed System Security Symposium (NDSS'15)* (2015).

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

[30] 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).

[31] Kirat, Dhilung and Vigna, Giovanni and Kruegel, Christopher. Barecloud: Bare-metal analysis-based evasive malware detection. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)* (2014).

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

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

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

[35] Nethercote, N., and Seward, J. 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).

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

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

[38] Qian, C., Luo, X., Shao, Y., and Chan, A. T. 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).

[39] SHI, H., ALWABEL, A., AND MIRKOVIC, J. Cardinal pill testing of system virtual machines. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)* (2014).

[40] SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS'08)* (2008).

[41] SPENSKY, C., HU, H., AND LEACH, K. LO-PHI: Low-observable physical host instrumentation for malware analysis. In *Proceedings of 23rd Network and Distributed System Security Symposium (NDSS'16)* (2016).

[42] SPISAK, M. 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).

[43] SUN, H., SUN, K., WANG, Y., AND JING, J. 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).

[44] SUN, M., WEI, T., AND LUI, J. 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).

[45] TAM, K., KHAN, S. J., FATTORI, A., AND CAVALLARO, L. CopperDroid: Automatic reconstruction of Android malware behaviors. In *Proceedings of 22nd Network and Distributed System Security Symposium (NDSS'15)* (2015).

[46] THE GNU MULTIPLE PRECISION ARITHMETIC LIBRARY. Pi with GMP. `https://gmplib.org/`.

[47] UBUNTU. sloccount. `http://manpages.ubuntu.com/manpages/precise/man1/compute_all.1.html`.

[48] VIDAS, T., AND CHRISTIN, N. Evading Android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (AsiaCCS'14)* (2014).

[49] VOGL, S., AND ECKERT, C. Using hardware performance events for instruction-level monitoring on the x86 architecture. In *Proceedings of the 2012 European Workshop on System Security (EuroSec12)* (2012).

[50] XEN PROJECT. Xen ARM with virtualization extensions. `https://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions`.

[51] XINYANG GE, H. V., AND JAEGER, T. SPROBES: Enforcing Kernel Code Integrity on the TrustZone Architecture. In *Proceedings of the 2014 Mobile Security Technologies (MoST'14)* (2014).

[52] 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).

[53] YU, R. Android packers: facing the challenges, building solutions. In *Proceedings of the Virus Bulletin Conference (VB'14)* (2014).

[54] ZHANG, F., LEACH, K., STAVROU, A., AND WANG, H. 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.

[55] ZHANG, Y., YANG, M., XU, B., YANG, Z., GU, G., NING, P., WANG, X. S., AND ZANG, B. 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).

[56] 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).

A Debugging Commands

Command	Description
rr	Output the value of all general purpose registers `X0` to `X30`, the stack pointer, and the program counter.
rw `n` `v`	Write 64-bit value `v` to the register named `n` and output the name the register and its new value.
mr `a` `s`	Output the content of the memory starting from 64-bit virtual address `a` with size `s`. If the virtual address does not exist, output `Incorrect address`.
mw `a` `v`	Write 8-bit value `v` to the 64-bit virtual address `a` and output the address and the 8-bit value stored in the address. If the virtual address does not exist, output `Incorrect address`.
ba `a`	Add a breakpoint at the 64-bit virtual address `a` and output the address. If the virtual address does not exist, output `Incorrect address`.
bd `a`	Delete the breakpoint at the 64-bit virtual address `a` and output the address. If the virtual address or breakpoint does not exist, output `Incorrect address`.
bc	Clear all the breakpoints and output `succeed`.
n	Step to the next instruction and output the instruction.
nb	Step to the next Java bytecode and output the bytecode.
nm	Step to the next Java method and output the call stack.
c	Continue the execution after a breakpoint and output `continued`.

B Domain Switching Time

We use the PMU counter to count the `CPU_CYCLES` event and calculate the elapsed time by the delta of the value and the frequency of the CPU. First we read the PMU counter twice continuously and calculate the elapsed cycles, and the difference in CPU cycles indicate the time elapsed between the two continuous PMU read instructions. Then we insert an `SMC` instruction between the two read instructions to trigger a domain switching with NINJA disabled, and the difference of the CPU cycles represents the round trip time of the domain switching in ATF. At last, we measure the CPU cycles with NINJA enabled, and this time consumption includes the time consumption of both ATF and our customized exception handler. To avoid the bias introduced by the CPU frequency scaling, we set the minimum scaling frequency equal to the maximum one to ensure that the CPU is always running in the same frequency. The results of 30 experiments are shown in the following table.

ATF Enabled	NINJA Enabled	Mean	STD	95% CI
		0.007 μs	0.000 μs	[0.007 μs, 0.007 μs]
✓		0.202 μs	0.013 μs	[0.197 μs, 0.207 μs]
✓	✓	0.342 μs	0.021 μs	[0.349 μs, 0.334 μs]