Remotely Controlling TrustZone Applications? A Study on Securely and Resiliently Receiving Remote Commands

Shengye Wan
The College of William & Mary
Williamsburg, VA, USA
swan@email.wm.edu

Kun Sun
George Mason University
Fairfax, VA, USA
ksun3@gmu.edu

Ning Zhang
Washington University in St. Louis
St. Louis, MO, USA
zhang.ning@wustl.edu

Yue Li
The College of William & Mary
Williamsburg, VA, USA
yli20@email.wm.edu

ABSTRACT

Mobile devices are becoming an indispensable part of work for corporations and governments to store and process sensitive information. Thus, it is important for remote administrators to maintain control of these devices via Mobile Device Management (MDM) solutions. ARM TrustZone has been widely regarded as the de facto solution for protecting the security-sensitive software, such as MDM agents, from attacks of a compromised rich OS. However, little attention has been given to protecting the MDM control channel, a fundamental component for a remote administrator to invoke the TrustZone-based MDM agents and perform specific management operations. In this work, we design an ARM TrustZone-based network mechanism, called TZNIC, towards enabling resilient and secure access to TrustZone-based software, even in the presence of a malicious rich OS. TZNIC deploys two NIC drivers, one secure-world driver and one normal-world driver, multiplexing one physical NIC. We utilize the ARM TrustZone-based high privilege to protect the secure-world driver and further resolve several challenges on sharing one set of hardware peripherals between two isolated software environments. TZNIC does not require any changes or collaboration of the rich OS. We implement a prototype of TZNIC, and the evaluation results show that TZNIC can provide a reliable network channel to invoke the security software in the secure world, with minimal system overhead on the rich OS.

CCS CONCEPTS

• **Security and privacy → Mobile and wireless security**; **Mobile platform security**.

KEYWORDS

Mobile Device Management, ARM TrustZone, Network Interface Card, Network Driver

WiSec '21, June 28–July 2, 2021, Abu Dhabi, United Arab Emirates

ACM ISBN 978-1-4503-8349-3/21/06…$15.00
https://doi.org/10.1145/3448300.3468501

ACM Reference Format:
Shengye Wan, Kun Sun, Ning Zhang, and Yue Li. 2021. Remotely Controlling TrustZone Applications? A Study on Securely and Resiliently Receiving Remote Commands. In *Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '21), June 28–July 2, 2021, Abu Dhabi, United Arab Emirates.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3448300.3468501

1 INTRODUCTION

Mobile device management (MDM) enables the vendor and corporate administrators to remotely perform essential functions such as remote I/O control [17] and remote patches [19, 40] on managed mobile devices, dramatically reducing the maintenance cost. Typically, a centralized control server sends out remote management commands to the remote mobile devices via a network control channel. The MDM agent installed on the mobile devices is responsible for performing management functionalities after receiving those remote commands.

Traditional MDM solutions are usually implemented by installing and running management agents along with other applications in the OS of mobile devices [39, 48]. However, in recent years, numerous attacks can gain the kernel privilege of mobile OS. For example, 315 new CVEs have been reported on Android in 2019 [22] as the privilege-escalation related vulnerabilities. Once the attacker acquires the kernel privilege, it may manipulate the MDM agent or block the remote commands from reaching the MDM agent.

ARM introduces TrustZone [2] as a hardware security extension to protect security-sensitive applications in a trusted execution environment [16, 17, 32, 33, 51, 55]. TrustZone technique divides the mobile devices into two isolated execution environments, namely, a *normal world* for running a mobile OS (called rich OS) and normal applications and a *secure world* for executing security-sensitive applications. TrustZone-based MDM solutions (e.g., Samsung KNOX [37]) have been developed to protect the integrity and confidentiality of MDM agents by migrating them from the normal world to the secure world. However, it is still challenging to guarantee that the network packets containing remote commands can be received by the MDM agents that are running in the secure world [31, 35]. The main reason is that the MDM agents still rely on the network driver and the untrusted mobile OS in the normal world to receive and forward the remote commands to the secure world, respectively [31, 35, 38]. Unfortunately, a malicious rich OS

can manipulate the network driver to drop the command packets from the remote administrator or refuse to awake the MDM agent after receiving the commands.

One straightforward solution is to add another network peripheral that is dedicated to the usage of the secure world. However, it may not be feasible on many mobile platforms with small form factor and constrained battery power [20, 45]. Given that most TrustZone-enabled mobile devices share the same set of network peripheral (e.g., NIC) between the normal world and the secure world, one intuitive solution is to migrate the network driver into the secure world for handling all network traffic. It can ensure the receiving of remote commands; however, since it needs to forward all regular network traffic to the rich OS, this solution will inevitably introduce an enormous amount of context switches and significant latency into the rich OS. Another solution is to share one network peripheral with two individual network drivers in two worlds. Some previous works [36, 49] adapt this solution in their TrustZone-based systems where the secure world initiates the connections with external servers; however, these solutions are not a good fit for applications that require long-term peripheral access, such as remote command processing in MDM. Particularly, since the remote commands are initiated by the remote servers, the secure world has no knowledge of when the command will be sent out. Therefore, it has to frequently suspend the non-secure driver to check the receiving packets in a timely manner. Compared to the use cases where the overhead is one time overhead, the unique characteristics of MDM may lead to significant overheads over the network services of the normal world. In this paper, we are trying to answer the following question:

> *For the mobile device that only equips one network peripheral (e.g., NIC) for each connection type, how to provide a reliable network for ARM TrustZone secure world while also maintaining the practicality of the rich OS?*

To answer the above question, we design and develop a TrustZone-based network mechanism called TZNIC, which enhances the resilience and security of the secure-world MDM agents on receiving remote management commands. The basic idea is to deploy two network drivers, a full-fledged normal-world network driver inside the rich OS and a customized slim network driver in the secure world, for multiplexing one shared physical network peripheral (e.g., NIC). To achieve continuous data exfiltration and remain stealthy, attackers are well motivated to maintain normal operations of the network driver in the rich OS but disable the secure world's access to the shared network peripheral. In this case, the normal-world driver can conduct the peripheral initialization, provide the software interface (i.e., descriptor ring buffers and SKB buffers), and handle the network traffic without being affected. Meanwhile, since one physical NIC can only connect to one network driver's software interface and the normal-world driver has already provided its interface, the secure-world driver is responsible for securely and resiliently reusing the normal-world interface for receiving remote commands. TZNIC makes zero modification on the rich OS and keeps the normal-world network driver as unmodified.

It is challenging for the secure-world driver to safely and securely access the normal-world interface, considering that the network interface is dynamically allocated in the normal-world memory and the normal world cannot be trusted to correctly provide the information about the memory layout. In other words, we need to fill the semantic gap between the two worlds. To solve this problem, TZNIC utilizes the high-privileged secure world to reliably inspect NIC registers and extract the memory semantic information based on those registers without relying on any untrusted software. After filling the semantic gap, TZNIC can identify and process the received commands from the located normal-world interface.

When sharing one network interface to receive network packets, the two network drivers in two isolated worlds may read the incoming buffer at the same time. Due to this race condition, the rich OS may deliberately delete the command packets before TZNIC reads it. To solve this challenge without incurring hefty overhead to the normal world, we use one core running in secure world to inspect the received packets in normal world and keep other cores running in the normal world. Note the concurrent reading of TZNIC does not interrupt normal-world usage on the network and it has no impact on the usability of the normal world. Moreover, we make two efforts to increase the possibility for the secure-world driver to receive command packets under the race condition. First, the remote server sends out the command packets multiple times to increase the chances for the secure world to receive the command. Second, for any received command packets, we save the packets into secure memory to prevent further manipulation of the attacker.

We implement a prototype of TZNIC on Juno r1 development board [7] and perform a system evaluation. The experimental results show that TZNIC can reliably receive 99.6% of remote commands that are sent out 5 times, even if the rich OS is compromised and intends to block normal-world agents from receiving any command. Moreover, the entire receiving process is transparent to the rich OS and incurs a small overhead. Our system can be deployed on ARM-based processor platforms with the support of a wide range of wired and wireless network devices.

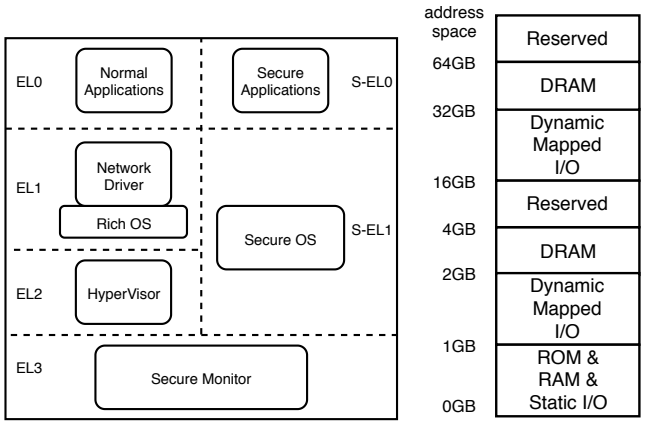
In summary, we make the following contributions.

(1) We propose an ARM TrustZone-based network mechanism called TZNIC, which enables two network drivers residing in the normal world and the secure world, respectively, to concurrently share one physical network peripheral.
(2) By filling the semantic gap, the secure-world network driver can enforce the sharing of the normal-world driver's software interfaces and utilize the shared interface for receiving network commands when the rich OS cannot be trusted.
(3) TZNIC does not require any modification to the rich OS and introduces minor overhead on the normal world. Moreover, since TZNIC does not rely on any assistance from the rich OS, it is transparent and remains stealthy to the rich OS.
(4) We implement a prototype of TZNIC on a development board. The experimental results show that TZNIC can reliably receive the remote management commands with a small system overhead on rich OS.

2 BACKGROUND

2.1 ARMv8-A Architecture

ARMv8-A is the latest ARM architecture with support for 64-bit registers and address space operations. Figure 1 shows the CPU security model and memory model of ARMv8.3-A.



(a) CPU Security Model [8] (b) Memory Model [4]

Figure 1: ARMv8.3-A Architecture

Security Model. ARM architecture provides the TrustZone security feature for the system to operate under two isolated environments: the normal world (also known as non-secure world) and the secure world. The normal world is accessible to the secure world, but not vice versa. On multi-core processor systems, the security setting of each core can be independently configured from the settings of other cores. As shown in Figure 1a, each core can execute instructions in one of six privileges, where *EL0*, *EL1*, *EL2* are used in the normal world and *S-EL0*, *S-EL1*, *EL3* are the secure world privileges. Since each core maintains an independent security status, a multi-core platform may run the secure and non-secure software applications at the same time.

ARMv8-A processors [8] allow the secure world to configure interrupts in two modes: *normal interrupt* and *secure interrupt*. The secure interrupts are always routed to the secure world no matter which world the CPU core is in [5]. Meanwhile, the normal interrupts (e.g., the interrupts generated by NIC) can be routed to either the normal world or the secure world, depending on specific configuration registers [5]. Since in most cases the device is running normal world tasks and the secure world is asleep, the secure interrupt is a key technique to guarantee that the secure-world components can be executed, especially when the normal world is compromised and may decline to invoke the secure world.

Memory Model. ARMv8-A uses a uniform memory address map to provide a consistent physical address to all shared resources, as shown in Figure 1b. The memory can be classified into two main types: *normal memory* and *device memory* [9]. ROM, SRAM, and DRAM belong to the normal memory, which can be configured as either secure memory or non-secure memory [6]. The device memory supports I/O devices, including Static I/O for on-chip peripherals and Dynamic Mapped I/O for general-purpose peripherals (e.g., NIC, mouse, and keyboard). Currently, most general-purpose peripherals treat all read/write access with uniform non-secure privilege, so the normal world and the secure world have the same view and operation privileges on the peripherals' registers [54].

2.2 DMA-Based NICs

Modern NICs handle network packets via either programmed memory mapped input/output (MMIO) or direct memory access (DMA). In MMIO, the NIC provides on-peripheral device memory and then waits for the CPU to exchange packets via device memory. When using DMA, the NIC directly reads and writes the network packets into the normal RAM memory, without involving the CPU. Most modern NICs are DMA-capable since DMA-based operations remove the burden of the CPU on handling the packets and thus dramatically increase the system performance.

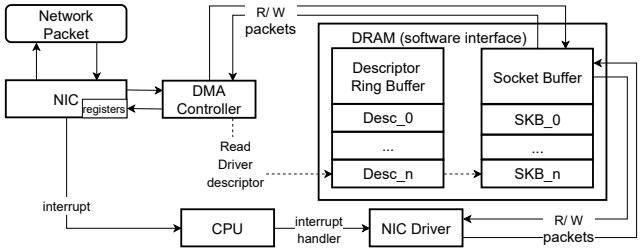


Figure 2: DMA-Based NIC Workflow

Figure 2 shows a typical workflow of the packet transmission between the DMA-capable NIC and its network driver. During the boot-up period, the NIC driver allocates multiple ring-buffer queues to store the descriptors, which are used to coordinate the NIC driver and the NIC on handling network packets asynchronously. Each descriptor points to a socket buffer (SKB) by saving a DMA-purpose physical address. The corresponding SKB buffer is allocated to store the real packet data. The descriptor and SKB data structures composite the *software interface* for NIC to communicate with the driver. When the NIC saves a latest received (RX) packet in memory, it checks the latest RX descriptor ring buffer *Desc_n* to get the available socket buffer *SKB_n*. Then it saves packet to the SKB via DMA Controller and raises an RX-interrupt to notify the CPU.

3 THREAT MODEL

We assume that all mobile peripherals, including the TrustZone-related peripherals and the network peripherals (e.g., NIC), are trusted for containing no known hardware vulnerabilities. Also, we assume TrustZone as trusted to protect the software running in the secure world. During the system boot-up, a trusted boot can ensure the integrity of the kernel images being loaded in both the secure world and the normal world. We assume there is only one DMA-based NIC available on mobile devices, which are equipped with multi-core processors to run secure and non-secure software simultaneously on different cores. Also, we assume the manufacturers have deployed their critical MDM agents in the secure world, and these agents are invoked asynchronously by receiving remote commands from the remote server [37].

We assume the rich OS may be compromised, and the attackers with the kernel privilege can deliberately block the security commands from the remote servers to be received by the security agent installed in the secure world. For instance, attackers can manipulate the driver-related data structures, including descriptor ring buffer and socket buffer (SKB), to hijack the receiving packets and remove

the remote command packets before they are forwarded to the secure world. Meanwhile, we assume attackers do not completely turn off the network peripherals or the network device driver in the rich OS due to the following reasons. First, a remote data-breach attack requires the device to be connected with the network. As most data-breach attackers do not have physical access to the device, they can only remotely leak users' private data to themselves. Second, a phishing attack needs to rely on the network to deceive the users and further collect users' sensitive information, such as login identification and credit cards' details. Third, any Advanced Persistent Threat (APT) cannot afford to fully disable the network since such unreasonable blocking can be easily detected and will break the stealthiness of the attacks. Therefore, the cases that an attacker directly turns off the device or place the device in physical isolation (e.g., a Faraday cage) is out of the scope of this paper.

4 CHALLENGES

After protecting the MDM agents in the secure world, it is challenging to ensure that the services can be invoked with remote commands received from a remote server, since the normal-world software, including the rich OS and the network driver, cannot be trusted to receive the remote server's commands in the normal world and then forward them to the secure world. We make several attempts towards securely and reliably receiving the remote command packets in the secure world.

The first attempt is deploying one complete secure network driver in the secure world to handle the network packets for both the secure world and the normal world and forward the network packets to the normal world. It can guarantee securely receiving the remote control packets. However, since most mobile network traffic happens to the normal world running most normal applications, the secure driver needs to forward all those heavy network traffics to the normal world. Such forwarding requires the processor to conduct the context switch between two worlds frequently, which incurs huge performance overhead and renders this solution impractical. Moreover, it requires to change the network driver in the normal world for the coordination of packet forwarding.

The second attempt is to deploy one complete non-secure driver in the normal world and another complete secure driver in the secure world. Since one NIC can only work with one driver's interface on receiving the network packets at any moment, the secure driver has a higher privilege than the non-secure driver on the usage of NIC. Therefore, when the secure driver tries to receive any network packet from NIC, the non-secure driver is suspended. However, since the secure world does not know the arriving time of the remote commands, it has to frequently suspend the non-secure driver to check the receiving packets in a timely manner and leads to expensive overhead on the normal network services. Therefore, it is not a practical solution to deploy two independently complete network drivers for controlling one shared network interface simultaneously.

Based on the above two failed attempts, we propose a viable solution called TZNIC that deploys a complete network driver in the normal world and multiplexes the normal-world driver's software interface to a slim secure network driver in the secure world. The normal-world driver is responsible for initializing the physical NIC device and providing all software interfaces such as descriptor buffers and packet buffers in the normal world. No changes need to be made on the original normal-world driver. The secure-world driver runs simultaneously with the normal-world driver on multi-core processors, focusing on securely and reliably receiving the remote commands. Instead of allocating another software interface by itself, the slim secure-world driver multiplexes the software interface provided by the normal-world driver. Since the secure-world driver multiplexes the NIC interface of the normal-world driver, this design faces three major challenges.

Challenge-1: Filling semantic gap. Since neither rich OS nor the normal-world driver can be trusted to work collaboratively, the secure-world driver has to figure out how to use the normal world's network interface without getting assistance from the normal-world software. In other words, the secure-world driver needs to fill the semantic gap in locating the critical software interfaces saved in the normal-world memory.

Challenge-2: Resisting interference from the normal world. On a multi-core system, when two network drivers share the same buffer-set, two drivers may read the same buffer simultaneously. Therefore, when the normal-world driver reads out the packets first from the RX interface, those packets cannot be read by the secure-world driver in the secure world any more. More severely, a malicious rich OS may deliberately delete the security-sensitive packets from the shared network software interfaces.

Challenge-3: Being transparent to rich OS. It consists of two requirements. First, the solution should not require any changes on the rich OS or its network driver. Second, it should have a minimal performance impact on the rich OS.

In the next section, we introduce the system design of TZNIC and present how we address all these three challenges in detail.

5 SYSTEM DESIGN

5.1 TZNIC Overview

An overview of TZNIC architecture is shown in Figure 3. During the device boot-up process, a normal-world network driver in the rich OS initializes the NIC. Typically, the initialization includes two major tasks, which are registering the interrupt handler for NIC interrupts and providing software interfaces for NIC to save all the received network packets in the normal-world memory. After the system boots and the normal-world driver initializes the NIC, all cores run in the normal world by default, in consideration of achieving the best of rich OS performance. Meanwhile, we deploy the slim secure-world driver TZNIC with three components: Sec-Awake, Sec-RX, and Sec-Buffer.

Since the secure world is asleep by default, TZNIC needs to first wake up the secure world reliably. Our Secure Activation Module, Sec-Awake, wakes up itself via a secure timer, which raises secure interrupts to switch one CPU core into the secure world. Since the arriving time of a remote packet is usually unpredictable to the secure world, Sec-Awake configures the timer to periodically generate the timer interrupts. After entering the secure world, Sec-Awake invokes the Secure Receiving Module, Sec-RX, to receive the remote server's commands arrived in the normal-world interface. Sec-RX first extracts the normal-world driver's software interface information from NIC on-peripheral registers. Since normal-world

attackers cannot disguise the registers from the secure world's view, Sec-RX is guaranteed to acquire the correct value of registers. Moreover, we present the mechanism to trusted understand the normal-world driver's semantic information based on these registers, which resolves *Challenge-1*.

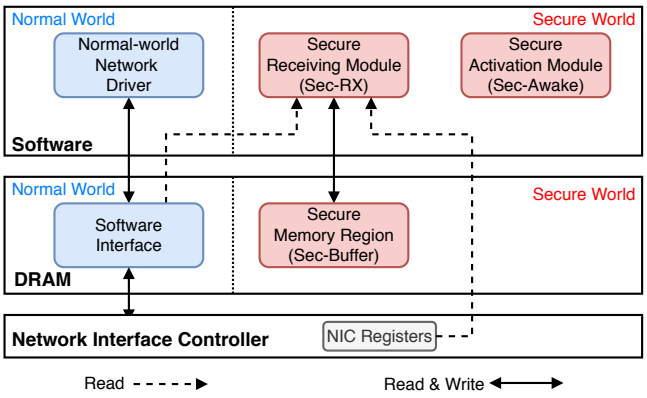


Figure 3: TZNIC Architecture

After that, the receiving module Sec-RX runs in parallel to the normal-world NIC driver on reading the received packets that are saved into the normal-world driver's receiving interface (RX buffers). As the the normal world may discard the secure-world packets benignly or maliciously before Sec-RX reads the packet. Sec-RX requires the packet sender to send the loss-tolerant packets multiple times for increasing the chance to identify the packets. After identifying a secure-world packet, Sec-RX makes a copy to a secure world DRAM region called Sec-Buffer so the normal world cannot touch the packet anymore and hence overcomes the *Challenge-2*. As we presented, the entire process of the secure world is working without modifying or requiring the collaboration of the normal world side, so our design is good to solve *Challenge-3*.

5.2 Secure Activation Module

When all cores are running for the normal-world tasks, a secure-world software can be executed in two scenarios. The first one is that the normal-world rich OS can generate an SMC instruction for asking a specific service deployed in the secure world. As our threat model defines the rich OS as untrusted, TZNIC cannot rely on this instruction to activate itself. The second scenario is the core receives a secure interrupt, which is not allowed to be handled in the normal world and therefore enforces the execution of a secure interrupt handler without relying on any normal-world component. One intuitive design to utilize the secure interrupt is configuring the NIC RX-interrupt as secure, so whenever NIC receives the packet, it will raise the interrupt and invoke TZNIC. However, since most received packets are irrelevant to the secure world, forcing every RX-interrupt to get handled in the secure world will delay the normal-world network tasks and hurts the rich OS performance. Moreover, our design aims to make zero modification on the normal-world network driver, which requires receiving RX-interrupts and handling them accordingly. In consideration of these two limitations, TZNIC does not configure the RX-interrupt as a secure interrupt for getting activation.

To utilize the secure interrupt for achieving reliable activation, we propose the secure activation module, Sec-Awake, which works based on a secure timer interrupt to wake up the secure world independently from the normal world. Under the latest ARMv8-A architecture, each core has a separate secure timer to trigger *Private Peripheral Interrupt (PPI)* for itself [5]. Thus, a secure timer PPI raised by one core is delivered to this core only, and the secure timer interrupt handler can guarantee to switch this core into the secure world. During the system boot, Sec-Awake initializes the secure timer to trigger the first interrupt after the booting process. Also, it registers one secure interrupt handler to ensure the core can enter the secure world. By taking advantage of the secure boot, which verifies the integrity of the secure-world booting images, Sec-Awake is guaranteed to reliably set the secure timer and register the interrupt handler without being attacked.

After successfully entering the secure world, Sec-Awake configures the core's secure timer with a fixed time gap *RX_period* to make sure the TZNIC will wake up in future. Then Sec-Awake invokes the receiving module Sec-RX to inspect and receive the secure-world network traffic properly.

5.3 Secure Receiving Module

The secure receiving module Sec-RX checks all received packets saved in the normal-world memory and filters out remote server's packets into Sec-Buffer. The entire checking and filtering process is expected to be finished without any support from the normal world. Another design goal of Sec-RX is to have a minimal impact on the normal-world driver and the rich OS. On single-core ARM processors, when the system enters the secure world, the rich OS is frozen until the system switches back. Fortunately, modern multi-core ARM processors allow the execution of Sec-RX on one core and the normal-world driver on other cores at the same time.

Since the rich OS cannot be trusted, the secure receiving module has to fill the semantic gap and extract the packets from normal-world software interface by itself. Sec-RX uses the high-privilege provided by the secure world to first access the registers of the shared NIC and then deduce the descriptors' information, such as the memory addresses for saving descriptors, based on the accessed registers. Due to the NIC requires a fixed structure to understand the descriptors provide by different rich OSes or their drivers, for any given NIC, the descriptor stores the socket buffer information in the unified format, and such format cannot be dynamically updated. Based on this feature, once Sec-RX locates the normal-world drivers' descriptors, the module can find out the socket buffer address by reading every descriptor. Finally, it uses the high-privilege again to read the packet saved in the corresponding memory address. The details of extracting such semantic information are presented in Section 7.2. To filter out the secure-world receiving packets from all received packets, Sec-RX allows each MDM agent to register their servers' IP addresses as the whitelist. Then Sec-RX keeps reading each packet's sender IP and checking if any packet's IP address matches any specific server. If there is a match, Sec-RX copies the packet to a pre-allocated secure memory region Sec-Buffer. Since Sec-Buffer is not accessible to the normal world, the

rich OS cannot delete or modify a packet once Sec-RX has retrieved the packet into Sec-Buffer. Moreover, to ensure a reliable reception for each command, Sec-RX requires the remote server to send each remote command with multiple copies and make each copy as a loss-tolerant packet. We present the security analysis of the repeated sending packets in Section 6.

For each wake-up, Sec-RX keeps reading packets from receiving buffers until polling time reaches an upper limit *RX_awake*. A higher limit means Sec-RX can read more packets in one round of wake-up. Meanwhile, the rich OS suffers worse performance with longer *RX_awake*, since rich OS loses one core's computation power to Sec-RX during its wake-up. Therefore, we suggest the numerical value of *RX_awake* in association with the wake-up period *RX_period* should be decided based on the specific target overall performance. We present the theoretical analysis in Section 8.2.

6 SECURITY ANALYSIS

We perform a security analysis of TZNIC against the attacker with complete rich OS privilege. An armored attack can proactively interfere with TZNIC's execution in three different attack vectors, i.e., forging packets, race condition, and NIC-based attacks.

Forging Packets. The rich OS may manipulate the receiving packets in the normal world to pass illegitimate packets into the secure world. To protect against the forged fake packets, the remote server and each mobile device share a secret key to authenticate each command packets. Note that the key is stored in secure memory on mobile, which prevents attacks of normal-world attackers. Besides, each remote command should be sent with with a unique and incremental command ID. The ID is proposed to avoid executing repeated or replayed commands (e.g., replay attacks).

Race Condition. Sec-RX may face a race condition between the normal and the secure worlds on handling the secure-world packets. Since both worlds have the same level of privilege to read from the receiving buffers in the normal world, the normal-world driver may discard secure-world packets from its buffers after the packets are read out. Besides, the rich OS may have chances to delete secure-world packets from the shared memory before Sec-RX reads them out. Due to these race conditions, secure-world packets may be dropped by the normal-world driver or the malicious OS before the secure world wakes up and copies those packets into the secure memory. As the countermeasure of the race condition, we propose the remote server send multiple copies of its packets and make each packet loss-tolerant (e.g., UDP packets) for the secure world. In this case, any packet received by Sec-RX should be independent and good enough to inform the secure-world agents about the required tasks. The suggested number of copies depends on the hardware configuration of the devices, and we present how to decide the duplicate packet number with the numeric examples in Section 8.1.

NIC-based Attacks. Our work focuses on software vulnerabilities, so we consider the hardware issues of the NIC (e.g., manufactured hardware bugs [25]) are out of the scope. Meanwhile, an attacker can still perform all software-related attacks including 1) utilizing the NIC to conduct DMA attacks and 2) exploiting the vulnerabilities of a secure network driver to perform secure-privileged access. First, since TZNIC does not configure the NIC as a secure peripheral, a normal-world attacker cannot manipulate the NIC to perform any direct access on the secure memory. Thus, adopting TZNIC will not introduce new DMA attacks into the secure world.

As for exploiting driver's vulnerabilities, previous researchers have shown that an IO driver can be exploited if the driver contains internal vulnerabilities [50]. Similar to any existing secure-world driver[1], TZNIC proposes to deliver the bug-free software as the best-effort engineering work. Meanwhile, we claim TZNIC exposes a small attacking surface with two security benefits. First, since TZNIC only interacts with the packets that are correctly encrypted by the remote server, the attacker can only use pre-received packets to exploit the secure driver's execution, instead of forging the packets with arbitrary content. Thus, the exploitation's capability and efficiency are highly limited. Second, since TZNIC does not allocate the network interface (e.g., ring buffers) in the secure memory, typical buffer-overflow attacks like [50] will happen within the insecure memory only. Therefore, the sharing interface of TZNIC protects secure-world software and data from these attacks. Finally, we provide an extra discussion in Section 9 as future works to develop multiple network drivers in the secure world. With such a design, the secure world will have better fault tolerance against the vulnerabilities of any single driver.

7 SYSTEM IMPLEMENTATION

We implement a prototype of TZNIC on an ARM Juno r1 development board [7], which has 6 ARMv8-A processor cores and uses Marvell 88e8057-a0-nnb2c000 PCI-E Gigabit Ethernet Controller (Yukon-II NIC) as its network controller.[2] The ARM CoreLink TZC-400 TrustZone Address Space Controller [6] is used to manage the address space as either normal or secure. The TZC-400 supports up to eight separate memory regions with different security settings. A CoreLink GIC-400 Generic Interrupt Controller [3] manages the normal and secure interrupts [5].

The normal world runs OpenEmbedded LAMP OS with Linux kernel version lsk-4.4-armlt in EL1. The OS is deployed with the network driver *sky2* (version 1.30) [27] to work with the Yukon-II NIC. The secure monitor running in EL3 and secure OS running in EL1 are modified based on ARM trusted firmware (ARM-TF) [10]. As a comparison, the normal-world driver contains 5707 LOC; Meanwhile, our implemented slim secure-world driver includes 722 LOC for Sec-RX and other 341 LOC for Sec-Awake, so its total size is 1063 LOC, which is only 18.63% of the original driver. As we presented in Section 10.2, a line of previous works propose the solution to migrate the entire network driver into the secure world. Comparing to these works, adopting TZNIC can introduce the network service with noticeably small TCB size.

7.1 Secure World Initialization

TZNIC reconfigures the memory space for receiving packets. We generate the page table entries for all normal world DRAM physical addresses, so Sec-RX can read any descriptor or packet that saved in the normal world memory. Meanwhile, TZNIC reserves $0x00200000$ bytes of the DRAM as the secure memory at the booting

[1] 29 drivers are implemented in the ARM Trusted Firmware open-source project [10].
[2] Our comprehensive survey in Section 9.1 shows most modern NIC are DMA-enabled and they can be used to serve TZNIC.

stage and sets it as an isolated memory region to serve as Sec-Buffer. The reserved memory is configured as readable, writable, and non-executable for secure world access only. Besides the memory configuration, Sec-Awake configures the secure timer's value during the initialization. The timer is configured to fire the first interrupt after the boot-up process. Sec-Awake controls the wake-up time by operating the per-core register *CNTPS_CVAL_EL1*. We set each wake-up period as 10 *ms*, and the wake-up gap as 80 *ms*. The numbers are calculated in Section 8.2.1 for balancing the network availability and performance overhead.

7.2 Sec-RX Implementation

An overview of the DMA-based NIC receiving workflow is shown in Figure 4. To recover the runtime semantics of the normal world, the secure world would retrieves various RX descriptor ring buffer information from the NIC registers, including the starting address *RX_Start_Addr*, the ring size *RX_Size*, the current offset of the descriptor that the driver is handling *RX_Tail*, and the latest buffer NIC just updated *RX_Head*. When a packet arrives, the NIC gets the latest available descriptor offset saved in *RX_Head*, extracts the SKB address and then saves the packet into corresponding memory. Finally, NIC moves the offset *RX_Head* forward. The registers *RX_Head* is read-only to all software, and so no malicious normal-world attacker can manipulate this value for deceiving TZNIC. Meanwhile, the driver always handles the packet at *RX_Tail* and forwards this register to tell NIC which buffer is the latest free one to use.

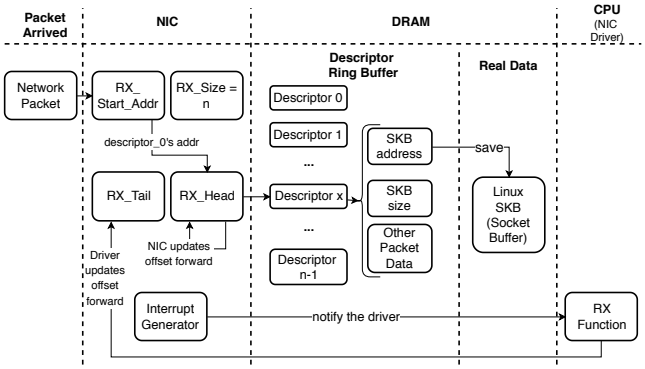


Figure 4: DMA-Based NIC Receiving Workflow

To locate and monitor all normal-world driver's software interface such as the descriptors and RX packets in the normal world, we extract the RX descriptor ring buffer information from these explained registers. All descriptor-related registers can be accessed by applying the offset, which is the sum of three parts: register base address *Y2_B8_PREF_REGS*, the number of the queue, and the register offset. The ring buffer's starting address can be calculated with two registers, namely *PREF_UNIT_ADDR_LO* and *PREF_UNIT_ADDR_HI*. By getting the complete address and corresponding offset, we can map each descriptor unit to the normal-world driver's descriptor structure *sky2_rx_le*. For each located descriptor, we identify the structure's attribute *__le32 addr*, which refers to the SKB address of the real packets. Note the attribute *addr* saves the DMA address (physical address), so the Sec-RX needs to translate the address into the secure world virtual address. Finally, we can read the packet content at the virtual address of *addr*. We implement the remote server to send commands as UDP packets and their entire payloads are encrypted with the AES algorithm.

8 SYSTEM EVALUATION

8.1 Packet Receiving Reliability

To study the reliability of TZNIC, we deploy a python tool Scapy [15] on the remote server to send UDP packets to the device. We send 100 packets as a test round and we evaluate 100 rounds for each scenario to calculate the packet received ratio. TZNIC tries to intercept these UDP packets before they are removed by the normal world. Meanwhile, we use the benchmark *iPerf* [23] as the normal-world application to receive the UDP packets from the same sender with sending configuration in comparison. We first evaluate the reliability of TZNIC under the scenario without race condition, where the rich OS is benign and the software interface provides enough buffers for saving the received UDP packets. In this circumstance, *iPerf* can receive 100% packets without any loss, and TZNIC can also receive 100% of the packets when the module wakes up.

To evaluate the scenario that normal world raises the race condition on the received packets, we test one extreme case that the malicious rich OS attempts to utilize all the normal-world computation power to delete the incoming packets and therefore interfere with the secure world network availability. We deploy a kernel-level attacking program for such interruption purpose. As the baseline of the attacking performance, the attacking program can fully block the benchmark *iPerf* from receiving any packet, which means the attack is strong enough to make any normal-world application unavailable from the network perspective. Even under such disturbance, TZNIC still can receive 67% of the packets from the remote server on average, with the minimum rate as 22% and the maximum receiving rate as 92%. Since the packets are repeatedly sent and loss-tolerant, receiving any copy will invoke the secure-world agent. For instance, if a command packet can be received with a 67% success rate and the server sends out 5 copies, our mechanism has a 99.6% chance to receive this command by the secure world.

Based on the minimum receiving rate, we can tell that even in the worst case, as long as the remote server can send out 5 copies of the command within the *RX_awake*, and keep sending the copies with the duration *RX_period*, then it is promising for TZNIC to receive one valid command. Assuming every remote command is shipped as a full-sized UDP packet with the maximum size 65535 *bytes*, then the connection speed between remote server and mobile device is required to be faster than $\frac{65535\ bytes * 5}{RX\_awake}$. As a numerical example, we assume the connection speed is 33.88 *Mbps*, which is the average USA mobile download speed in 2019 [44]. With this connection speed, TZNIC should configure its wake-up time $RX\_awake \geq 10\ ms$ in order to satisfy the receiving requirement.

8.2 Performance Overhead

In this subsection, we measure the performance impact of TZNIC on the device. Since TZNIC's system overhead varies with its awake time and awake period, we first present a theoretical analysis of

their impacts. We then show the specific overhead introduced by TZNIC when it's awake.

8.2.1 Theoretical Analysis. Assuming a real-world device has been turned on for the total duration T and TZNIC has been executed with the duration T_RX. If we define the device performance when our mechanism is not working to be 100% and the degraded performance with running TZNIC as $Perf\_down$, then we have the overall performance $Perf\_over$ as follows.

$$Perf\_over = \frac{T\_RX * Perf\_down + (T - T\_RX) * 100\%}{T} \quad (1)$$

According to our design, we have the time factor $T\_RX = T * \frac{RX\_awake}{RX\_period}$, and we further define the sleeping time of the secure world $RX\_sleep = RX\_period - RX\_awake$. By considering the wake-up and period conditions, the overall performance of TZNIC is presented as follows.

$$Perf\_over = \frac{RX\_awake * Perf\_down + RX\_sleep * 100\%}{RX\_period} \quad (2)$$

Next, we present how to calculate the ratio of RX_awake to RX_period in order to satisfy the performance requirement. When TZNIC is set to achieve a target performance $Perf\_target$, we have the relationship between RX_awake and RX_period as follows.

$$\frac{RX\_awake}{RX\_period - RX\_awake} = \frac{100\% - Perf\_target}{Perf\_target - Perf\_down} \quad (3)$$

As the Equation 3 indicates, TZNIC can maintain the rich OS with a stable overall performance $Perf\_target$ by properly tuning the time conditions RX_period and RX_awake. As a numeric example, assume we set the target performance $Perf\_target$ = 95%, and we choose the worst degradation performance $Perf\_down \approx 65\%$ as we evaluated in Section 8.2.2, then we know by setting $RX\_period = 8 * RX\_awake$, the rich OS's overall performance is promised to be equal or higher than 95%.

8.2.2 Empirical Study. We first use the tool *iPerf* to evaluate the communication overhead on the normal world. Our experiment shows that TZNIC only introduces negligible overhead no matter the NIC is reading or sending packets in the normal world. This result is reasonable since TZNIC only reads data from the memory using one core for a short period of time, while all other cores are still running the normal network operations in the normal world.

Next, we use the benchmark UnixBench [47] in the normal world to evaluate the overall computation overhead in the normal world. We run the benchmark in two scenarios, with and without TZNIC running in parallel. We consider the normal-world performance without TZNIC as 100% and then normalize the normal-world performance with TZNIC accordingly. For each scenario, we run the UnixBench benchmark with two sets of different configurations. The `1-task` indicates we conduct each task of the benchmark as a one-thread copy. The `6-task` case runs each task with six copies simultaneously on all six cores of our platform. Figure 5 illustrates the performance overhead caused by TZNIC over the normal world.

TZNIC introduces 16.67% performance degradation on the 1-task tests and 23.54% on the 6-task tests. All the 6-task tests suffer more degradation than the 1-task tests since the 6-task tests are supposed to utilize 6 cores simultaneously while the test loses one core when TZNIC is running. We observe that TZNIC affects most benchmark tests with a stable overhead from 11.113% to 18.124% while the `execl` and `file copy` tests suffer the degradation from 29.517% to 38.610%. The reason is that most test sets cannot utilize all CPU resources even for the 6-task cases, but since both the `execl` and `file copy` performance results rely more on the CPU resources, these two sets are affected more by losing even one core. Note that the presented performance degradation represents the $Perf\_down$ in Equation 2, which is only incurred when TZNIC is awake. Meanwhile, as we suggested in Section 8.2.1, in most time TZNIC is asleep, so the overall performance of the mobile is maintained ideally.

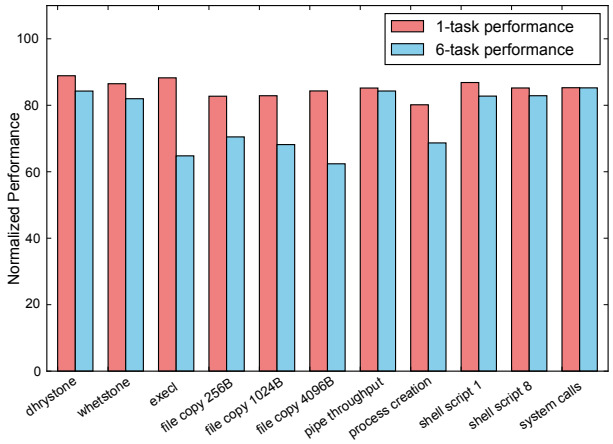


Figure 5: Performance Overhead on Normal World

9 DISCUSSION

We demonstrate the portability of our TZNIC architecture on other ARM-based platforms with various wired and wireless network interfaces. Also, we discuss the limitations of our mechanism and propose potential solutions for resolving those limitations.

9.1 Portability of TZNIC

TZNIC architecture has three particular requirements, namely, multi-core processors, a high-privileged operating mode, and a DMA-based network peripheral. We show that all three requirements can be satisfied in a wide range of mobile devices. First, most ARM-based processors modern processors (e.g., A53, A57, etc.) are designed with the capability to be integrated with multiple cores. Any device equipped with more than one core is qualified to execute the secure-world and normal-world drivers simultaneously. Second, on ARM processors, the TrustZone technique has been widely integrated to provide an isolated execution environment for protecting the integrity and providing the high-privilege vision for TZNIC to inspect the on-peripheral registers and normal-world driver's software interfaces.

Table 1: Porting TZNIC to Other NIC Models

Type	Brand	Model	Chipset	Driver	DMA-Based	Other Chipsets
Wired NIC	Intel	EXPI9301CTBLK	Intel 82574L	intel / e1000e	Yes	82571, ich9lan, pch_lpt, and other 10 chipsets, total 13
Wired NIC	StarTech	ST1000BT32	RTL8110SC	realtek / r8169.c	Yes	RTL8100e, RTL8168cp, RTL8402, and other 32 chipsets, total 35
Wired NIC	Syba	SD-PEX24041	RTL8111F	realtek / r8169.c	Yes	driver has been covered above
Wired NIC	Realtek	RT8111C-PCIE-NIC	RTL8111C	realtek / r8169.c	Yes	driver has been covered above
Wired NIC	D-Link	DGE-530T	DGE-530T	marvell / skge.c	Yes	3Com 3C940, D-Link DGE-530T, and other 11 chipsets, total 13
Wireless NIC	Intel	7260.HMWG.R	Intel 7260	intel / iwlwifi / cfg / 7000.c	Yes	Intel 7260, Intel 3160, Intel 3168, Intel 7265, Intel 7265D, total 5
Wireless NIC	TP-Link	Archer T6E	BCM4352	broadcom / b43	Yes	BCM4306, BCM4311, BCM4318, and other 8 chipsets, total 11
Wireless NIC	Asus	PCE-AC56	BCM4352	broadcom / b43	Yes	driver has been covered above
Wireless NIC	StartTech	300 Mbps N PCI-E	Ralink-RT5392	ralink / rt2x00 / rt2800lib.c	No. Only TX data is sent via DMA	
Wireless NIC	FebSmart	N600	Atheros 802.11n	ath / ath9k	No. Only TX data is sent via DMA	

Third, we conduct a study to confirm that most modern network peripherals work as DMA-based NICs and the detailed result is presented in Table 1. We first identify the top 5 popular wired and top 5 wireless network interface cards according to a list of best sellers in computer networking cards provide by Amazon [1]. For the top 5 most popular wired and wireless network NIC model, we find the related `Driver` information based on their `Chipset`. The column `DMA Packets` shows if the driver and corresponding hardware chipsets work as DMA-based NIC or not. If yes, then TZNIC can be designed to cooperate with the corresponding NIC. Finally, since one driver may support more than one chipset so as long as the driver works as DMA logic, the chipsets in `Other Chipset` also can be extended with TZNIC.

Since we cannot afford to buy all listed network devices, we only check the NIC models whose drivers are open-sourced and supported in the latest Linux kernel downloaded from Github [41] with the git-tag v4.18-rc7. Fortunately, all the drivers of `wired NIC` can be found in the directory of `drivers/net/ethernet` while the drivers of `wireless NIC` can be found in the directory of `drivers/net/wireless`. When a NIC brand has more than one model as the top 5 popular models, we only choose the most popular model for the brand. We skip the NICs that do not provide official documentation on their Linux driver support. We find 8 of the 10 network devices support to work as DMA-based peripherals, and their drivers can cover more than 70 chipsets in total to work as the DMA-based NIC. For those two networking cards that are not working as DMA-based NIC, we find their manufacturers provide other products in DMA fashion, which means these branches have alternative chipsets that can be integrated with TZNIC. For example, even though the driver `ath9k` of Atheros is not a DMA-based driver, another Atheros's driver `ath5k` that is working for Atheros 802.11a/bg Chipset can cooperate with the hardware as DMA-based operation [42]. Also, even the driver `rt2800lib.c` is not a DMA-related driver, another Ralink driver `rt73usb.c` that resides on the folder `<ralink/rt2x00/>` can provide DMA capability.

9.2 Limitations and Future Work

The primary limitation of TZNIC is that the mechanism cannot stop normal-world attacker to fully disable the NIC, such as turning off the card via manipulating the power ON/OFF of the NIC. However, we believe that majority of attackers will leave the network on. Given that the cyber attack nowadays often has a well-planned objective behind, the adversary will most likely leverage the compromised mobile devices to steal data or conduct harm to the owners. The attacker may either have physical access to the device or is attacking remotely. When the attacker has physical access, it can often pull the flash memory out directly, or launch other physical attacks such as putting the phone in a metal box. In this case, there is very little the software can do in terms of network traffic. However, since many of the attacks are carried out remotely, the attackers are well motivated to keep the network communication open, so they can maintain a command and control channel to continuously ex-filtrate sensitive data information from the devices.

Even though we may deploy some monitoring module in the secure world to check and reset the NIC's power-related registers, the normal world still has the same privilege to modify them again, since the NIC cannot differentiate the secure-world's write access from the normal-world's write access. Therefore, one potential solution for resolving this limitation is to utilize extra TrustZone peripheral-related security controllers, such as Central Security Unit (CSU) [43] or TrustZone Protection Controller (TZPC) [2], to prevent the normal world from manipulating the NIC on-peripheral registers and turning off the NIC. With those security controllers, we can deploy a complete network driver in the secure world to initiate and configure the NIC and a slim driver in the normal world to access the NIC. However, nowadays CSU and TZPC controllers are only available on a limited number of development boards, and they have not been integrated as a standard component by ARM TrustZone architecture. We consider to utilize the TrustZone peripheral-related controller with TZNIC as our future work.

Another limitation of TZNIC is that parsing rich packets may introduce complexity into the secure world and thus increase the attack surface. One mitigation is to follow a layered design in the

driver instead of the monolithic driver. Another solution is to make use of the latest hardware-enabled virtualization feature in the TrustZone, which is available from ARMv8.4-A [26]. Alternatively, it is also possible to write the driver using memory-safe languages such as Rust [52]. Regardless of the mitigation technique that can be adapted to harden the secure world, the design of TZNIC remains effective in providing the secure world a configurable level of access to the peripherals in the normal world. Our approach of securely sharing network interface can be combined with either secure virtualization feature or memory-safe languages.

Finally, current TZNIC does not provide the network transmitting function inside the secure world. Unlike the receiving case, if we allow both normal-world and secure-world drivers to write in parallel to the shared buffers for sending out packets, then it may cause concurrent-write issue and crash the rich OS. Current TZNIC requests a secure-world MDM agent to send out network packets via the normal-world network channel, which is the same method used in the existing TrustZone-based MDM solutions [38]. To reliably transmit packets for the secure-world agents, one solution for the secure world is first suspending the entire rich OS and then using the NIC exclusively. As the normal world is frozen, any normal-world attacker cannot disturb the transmitting process of the secure world. However, it may introduce extra overhead and new challenges for writing on the shared TX software interface. We leave it as a future work.

10 RELATED WORK

10.1 High Privileged Operating Modes

Recently, high privileged operating modes than ring 0 have been widely supported in both x86 and ARM processors to isolate security-sensitive code from rich OS and control the normal system resources [2, 28, 29]. The high-privileged mode provides an alternative solution to protect the secure code when a co-processor is not available. For x86 processors, both Intel and AMD support System Management Mode (SMM) in their x86 processors to execute the code with a higher privilege than that running in the protected mode [29]. A number of solutions [12, 13, 56] proposed to use SMM for protecting the network services. For instance, SICE [13] requires the network service to send the integrity attestation to the remote side. However, the network service's availability of SICE is not in the scope of their work. Wang et al. [53] deploy a secure NIC driver within SMM to achieve a reliable network connection. However, it increases the attack surface of the trusted computing base. SMM-Rootkit [24] proposes an SMM-based service that introspects the normal world NIC and driver data by hijacking the NIC's interrupt handler. SMM has been used to achieve out-of-band communications with a dedicated NIC (e.g., HyperSentry [12], Hypercheck [56]). On x86-based desktops, workstations, and laptops, Intel's Active Management Technology (AMT) provides a high privileged operating mode for remote out-of-band management of Intel vPro processors [28]. In recent research works, Intel's Software Guard Extensions (SGX) technique has been used for secure communication in x86-based systems [14, 30]; however, the SGX enclave runs as a user-level process instead of a high-privileged mode like TrustZone. Limited by the privilege setting, adapting TZNIC is out of SGX's capacity.

10.2 TrustZone-Based Network Connection

Besides the x86 architecture, ARM TrustZone has been studied a lot [18, 51, 57] to provide comprehensive security-related supports in different aspects. Along with the increasing of the functionalities, more and more works [17, 32, 33, 36, 38] presented different solutions regarding on protecting the the network services on mobile devices. For instance, TrustZone-based remote attestation enables secure message exchanges between the secure world and the server [33, 38]. A challenge on mobile phones is that there is usually a single NIC that is shared between the normal world and the secure world. One solution is to use the network NIC drive in the normal world to send the encrypted packets created by the secure world, and use the TrustZone to verify and harden the network driver in the normal world. TZ-RKP [11] protects the normal world kernel including the normal world network driver. Li et al. [34] propose building up a trusted path from the normal world network driver to the secure world. A complete secure network driver can be implemented in the secure world [36, 49].

In single-core ARM systems, it must suspend the normal world including its network driver to transmit secure packets. On the multi-core platform, such a complete secure-world driver solution requires sophisticated collaboration from the normal world to yield the NIC to the secure world. In TZNIC, both normal world and secure world share the same physical NIC by running two network drivers on different CPU cores with the same software interface.

Another line of research focuses on the management of the network peripherals instead of utilization. Santos. et al. [21, 46] proposes the idea about utilizing the TrustZone to restrict the peripheral usage via Trust Leases. Brasser. et al. [17] presents the idea to control the normal-world network driver with the secure privilege in the restrict area, for example, a classroom when students are taking tests. SeCloak [32] controls the peripheral's availability by configuring it's security attributes to make sure the peripheral is successfully turned on or off. Unlike these works with the focus on peripherals' ON/OFF regulation, TZNIC's scope include not only the management but also the utilization of these peripherals.

11 CONCLUSION

In this paper, we first identify a remaining challenge on TrustZone-based MDM solutions, namely, no guarantee for the MDM agents to reliably receive commands from remote administrators. We then develop TZNIC, a TrustZone-based network mechanism that shares a single physical NIC between the normal world and the secure world on multi-core ARM processors. By properly exploiting the TrustZone architecture and features of DMA-enabled modern NIC, TZNIC can ensure the secure world for filling the semantic gaps and receiving remote commands simultaneously with the rich OS, remaining self-sufficient and transparent to the rich OS. TZNIC has a small trusted computing base in the secure world. It requires no changes to the existing mobile operating systems, so it is promising to be ported to more mobile devices.

ACKNOWLEDGMENTS

This work was supported in part by US National Science Foundation under grants CNS-1815650, CNS-2038995, CNS-1916926 and US Office of Naval Research under grants N00014-18-2893.

REFERENCES

[1] Amazon. Accessed in June 2018. Best Sellers in Internal Computer Networking Cards. https://www.amazon.com/Best-Sellers-Computers-Accessories-Internal-Computer-Networking-Cards/zgbs/pc/13983711.

[2] ARM. 2009. Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[3] ARM. 2011. CoreLink GIC-400 Generic Interrupt Controller. https://static.docs.arm.com/ddi0471/a/DDI0471A_gic400_r0p0_trm.pdf.

[4] ARM. 2012. Principles of ARM Memory Maps White Paper. http://infocenter.arm.com/help/topic/com.arm.doc.den0001c/DEN0001C_principles_of_arm_memory_maps.pdf.

[5] ARM. 2013. ARM Generic Interrupt Controller Architecture version 2.0. http://docs-api-peg.northeurope.cloudapp.azure.com/assets/ihi0048/b/IHI0048B_b_gic_architecture_specification.pdf.

[6] ARM. 2015. ARM CoreLink TZC-400 TrustZone Address Space Controller. https://static.docs.arm.com/100325/0001/arm_corelink_tzc400_trustzone_address_space_controller_trm_100325_0001_02_en.pdf.

[7] ARM. 2015. Juno ARM Development Platform SoC Technical Reference Manual, Revision: r1p0. https://www.arm.com/files/pdf/DDI0515D1a_juno_arm_development_platform_soc_trm.pdf.

[8] ARM. 2015. Programmer's Guide for ARMv8-A. https://developer.arm.com/documentation/den0024/a.

[9] ARM. 2016. ARMv8-A Memory Systems version 1.0. https://static.docs.arm.com/100941/0100/armv8_a_memory_systems_100941_0100_en.pdf.

[10] ARM. 2018. ARM Trusted Firmware. https://github.com/ARM-software/arm-trusted-firmware.

[11] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 90–102.

[12] Ahmed M Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C Skalsky. 2010. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *ACM CCS*.

[13] Ahmed M Azab, Peng Ning, and Xiaolan Zhang. 2011. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 375–388.

[14] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 8.

[15] Philippe Biondi and the Scapy community. 2018. Scapy's Documentation. http://scapy.readthedocs.io/en/latest/index.html.

[16] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2019. SANCTUARY: ARMing TrustZone with User-space Enclaves.. In *NDSS*.

[17] Ferdinand Brasser, Daeyoung Kim, Christopher Liebchen, Vinod Ganapathy, Liviu Iftode, and Ahmad-Reza Sadeghi. 2016. Regulating arm trustzone devices in restricted spaces. In *ACM MobiSys*.

[18] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. 2020. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA*. 18–20.

[19] Yaohui Chen, Yuping Li, Long Lu, Yueh-Hsun Lin, Hayawardh Vijayakumar, Zhi Wang, and Xinming Ou. 2018. InstaGuard: Instantly Deployable Hot-patches for Vulnerable System Programs on Android. In *NDSS'18*.

[20] CNET. accessed March 2020. Dual-SIM and eSIM on Apple's 2018 iPhones: Everything you need to know. https://www.cnet.com/how-to/dual-sim-and-esim-on-apples-2018-iphones-everything-you-need-to-know/.

[21] Miguel B Costa, Nuno O Duarte, Nuno Santos, and Paulo Ferreira. 2017. TrUbi: A System for Dynamically Constraining Mobile Devices within Restrictive Usage Scenarios. In *ACM MobiHoc*.

[22] CVE Details. 2019. Android CVE Details. https://www.cvedetails.com/product/19997/Google-Android.html.

[23] Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, Kaustubh Prabhu, Mark Ashley, Aaron Brown, Aeneas Jaißle, Susant Sahani, Bruce Simpson, and Brian. Tierney. 2018. iPerf Benchmark. https://iperf.fr.

[24] Shawn Embleton, Sherri Sparks, and Cliff C Zou. 2013. SMM rootkit: a new breed of OS independent malware. *Security and Communication Networks* 6, 12 (2013), 1590–1605.

[25] Project Zero Gal Beniamini. 2017. Over The Air: Exploiting Broadcom's Wi-Fi Stack. https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html.

[26] Matthew Gretton-Dann. 2017. Introducing 2017's extensions to the Arm Architecture. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/introducing-2017s-extensions-to-the-arm-architecture.

[27] Stephen Hemminger. 2005. Sky2 Driver Source Code. https://elixir.bootlin.com/linux/v4.17-rc4/source/drivers/net/ethernet/marvell/sky2.c.

[28] Intel. 2018. Active Management Technology (AMT). https://www.intel.com/content/www/us/en/architecture-and-technology/intel-active-management-technology.html.

[29] Intel. Accessed in June 2020. System Management Mode (SMM). https://en.wikipedia.org/wiki/System_Management_Mode.

[30] Seongmin Kim, Youjung Shin, Jaehyung Ha, Taesoo Kim, and Dongsu Han. 2015. A first step towards leveraging commodity trusted execution environments for network applications. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*. ACM, 7.

[31] Daniel Komaromy. Accessed in June 2020. Unbox Your Phone. https://medium.com/taszksec/unbox-your-phone-part-i-331bbf44c30c.

[32] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. 2018. SeCloak: ARM Trustzone-based Mobile Peripheral Control. In *MobiSys*. ACM.

[33] Wenhao Li, Haibo Li, Haibo Chen, and Yubin Xia. 2015. Adattester: Secure online mobile advertisement attestation using trustzone. In *MobiSys*. ACM.

[34] Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tieyan Li. 2014. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*. ACM.

[35] Linaro. Accessed in June 2020. OP-TEE Documentation. https://optee.readthedocs.io/.

[36] Dongtao Liu and Landon P Cox. 2014. Veriui: Attested login for mobile devices. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*. ACM, 7.

[37] Samsung Electronics Co. Ltd. 2017. White Paper: Samsung Knox Security Solution. https://www.samsungknox.com/docs/SamsungKnoxSecuritySolution.pdf.

[38] Samsung Electronics Co. Ltd. 2018. Get Started with Knox Attestation. https://seap.samsung.com/tutorial/get-started-knox-attestation.

[39] ManageEngine. accessed March 2020. Mobile Device Manager Plus. https://https://www.manageengine.com/mobile-device-management/.

[40] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. 2013. PatchDroid: Scalable Third-party Security Patches for Android Devices. In *ACSAC'13*.

[41] Andy Nguyen. 2021. BleedingTooth: Linux Bluetooth Zero-Click Remote Code Execution. https://google.github.io/security-research/pocs/linux/bleedingtooth/writeup.html.

[42] Minh-Son Nguyen and Quan Le-Trung. 2013. Integration of Atheros ath5k device driver in wireless ad-hoc router. In *Advanced Technologies for Communications (ATC), 2013 International Conference on*. IEEE.

[43] NXP. 2013. Applications Processor Security Reference Manual for i.MX 6SoloLite. https://www.nxp.com/webapp/sps/download/mod_download.jsp?colCode=IMX6DQ6SDLSRM.

[44] Ookla. Accessed in June 2020. 2019 Speedtest U.S. Mobile Performance Report. https://www.speedtest.net/reports/united-states/.

[45] Mitja Rutnik. accessed March 2020. The best dual SIM Android phones to spend your money on. https://www.androidauthority.com/best-dual-sim-android-phones-529470/.

[46] Nuno Santos, Nuno O Duarte, Miguel B Costa, and Paulo Ferreira. 2015. A Case for Enforcing App-Specific Constraints to Mobile Devices by Using Trust Leases.. In *HotOS*.

[47] Ben Smith, Rick Grehan, Tom Yager, and DC Niemi. 2011. Byte-unixbench: A Unix benchmark suite.

[48] SOTI. accessed March 2020. SOTI MOBICONTROL. https://soti.net/mobicontrol.

[49] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. 2015. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *DSN*. IEEE.

[50] Torvalds. 2018. GitHub Linux Kernel. https://github.com/torvalds/linux.

[51] Shengye Wan, Jianhua Sun, Kun Sun, Ning Zhang, and Qi Li. 2019. SATIN: A Secure and Trustworthy Asynchronous Introspection on Multi-Core ARM Processors. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 289–301.

[52] Shengye Wan, Mingshen Sun, Kun Sun, Ning Zhang, and Xu He. 2020. RusTEE: Developing Memory-Safe ARM TrustZone Applications. In *Annual Computer Security Applications Conference*. 442–453.

[53] Jiang Wang, Fengwei Zhang, Kun Sun, and Angelos Stavrou. 2011. Firmware-assisted memory acquisition and analysis tools for digital forensics. In *Systematic Approaches to Digital Forensic Engineering (SADFE), 2011 IEEE Sixth International Workshop on*. IEEE, 1–5.

[54] ARM Developer Website. 2018. Accessing memory-mapped peripherals. https://developer.arm.com/products/software-development-tools/ds-5-development-studio/resources/tutorials/accessing-memory-mapped-peripherals.

[55] Kailiang Ying, Priyank Thavai, and Wenliang Du. 2019. TruZ-View: Developing TrustZone User Interface for Mobile OS Using Delegation Integration Model. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*. ACM, 1–12.

[56] Fengwei Zhang, Jiang Wang, Kun Sun, and Angelos Stavrou. 2014. Hypercheck: A hardware-assisted integrity monitor. *IEEE Transactions on Dependable and Secure Computing* 11, 4 (2014), 332–344.

[57] Ning Zhang, Kun Sun, Wenjing Lou, and Y Thomas Hou. 2016. Case: Cache-assisted secure execution on arm processors. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 72–90.