MyTEE: Own the Trusted Execution Environment on Embedded Devices

Seungkyun Han
Chungnam National University
yakmg3000@gmail.com

Jinsoo Jang
Chungnam National University
jisjang@cnu.ac.kr

***Abstract*—We propose a solution, MyTEE, that enables a trusted execution environment (TEE) to be built even in worst-case environments wherein major hardware security primitives (e.g., ARM TrustZone extensions for memory access control) are absent. Crafting page tables for memory isolation, filtering DMA packets, and enabling secure IO exist at the core of MyTEE. Particularly for secure IO, we shield the IO buffers and memory-mapped registers of the controllers and securely escalate the privilege of the partial code block of the device drivers to provide permission to access the protected objects. By doing so, the need to host the device driver in the TEE (in whole or in part), which can potentially introduce a new attack surface, is exempted. The proof-of-concept (PoC) of MyTEE is implemented on the Raspberry Pi 3 board, which does not support most of the important security primitives for building the TEE. Additionally, three secure IO examples with the hardware TPM, framebuffer, and USB keyboard are demonstrated to show the feasibility of our approach.**

I. Introduction

The trusted execution environment (TEE) is one of the reasonable security measures for protecting security-critical services and data on embedded devices. Particularly, considering ARM's high market share in mobile and embedded devices (90% for mobile, IoT, and in-vehicle systems), TrustZone, the security extension of ARM processor, is the most potent measure to enable TEEs on embedded devices. It provides various security extensions, such as separating the security states of the CPU, hardware-based memory access control, and secure IO. Using such features, various TEE applications have been proposed [27], [28], [32], [33], [35], [38], [40], [41], [43], [50], [52]–[54], [56], [57].

Despite its effectiveness and adoption in a wide range of research, deployment of the TEE in actual working environments is limited for the following reasons. Embedded devices sometimes are incomplete in their implementation of the hardware components required to build the TEE. For example, the Broadcom BCM2837 SoC [7], which is deployed in commercial devices [16] (e.g., smart hub), does not support TrustZone extensions for memory and peripheral isolation [12]. Even if the hardware components are available, the entire trusted application (TA) development (and deployment) process may require vendor involvement for using proprietary APIs and performing security verification [9]. This can increase the time-to-market for new third-party services.

We propose MyTEE to address the limitations of hosting TEE on embedded devices. It is designed with the harsh assumption that most TrustZone extensions are not supported (other than the security state of the CPU). In other words, TrustZone Address Space Controller (TZASC) and TrustZone Memory Adapter (TZMA) for memory access control, and TrustZone Protection Controller (TZPC) for establishing a secure IO channel, are not supported. The input/output memory management unit (IOMMU) for preventing malicious direct memory access (DMA) is not available either. Without such hardware security primitives, MyTEE isolates the TEE region, prevents DMA attacks, and dynamically builds a secure IO channel between the TEE and peripherals.

Memory protection is achieved by deliberately managing the page tables. The stage-2 page table [15] (i.e., extended page table on x86 [3]) that maps the intermediate physical addresses to physical addresses is leveraged to isolate the TEE from the untrusted OS. Part of MyTEE is implemented as a tiny hypervisor, which can also be compromised. Because stage-2 paging-based protection is not effective once the attacker obtains hypervisor privileges, we also ensure that the page table of the hypervisor does not map the TEE and is immutable. However, even with careful management of the page table, security of MyTEE could still be broken by malicious DMA. To address this, we implemented a DMA filter that traps, verifies, and emulates any memory-mapped IO (MMIO) to the DMA controller. To realize secure IO, we delegated a task to the untrusted OS that sends a request to the peripherals (e.g., TPM commands) instead of porting the device drivers in the TEE. Minimal but essential components for trustworthy communication, such as the buffer for peripheral output and MMIO region for peripheral controllers, are protected by the stage-2 paging. Then, the partial code block of the device driver is given hypervisor privilege to access the secure objects and log the transactions for future validation by the trusted application (TA).

We implemented MyTEE on a Raspberry Pi 3 development board equipped with a Broadcom BCM2837 SoC that does not support TrustZone extensions. OP-TEE [4] and Raspbian OS with Linux 4.15 were hosted as the TEE and the rich execution environment (REE) software platform, respectively. The ARM trusted firmware was patched to enable memory isolation. The DMA filter and MyTEE services (e.g., privilege escalation of a device driver) were implemented as part of the

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.23041
www.ndss-symposium.org

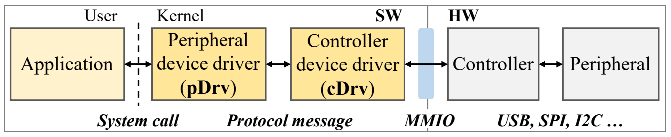


Fig. 1: Communication channel between an application and a peripheral.

TABLE I: Example ARMv8-A based SoCs that lack TrustZone extensions and their target device types.

Vendor	SoC	Secure State	TZPC	TZASC	TZMA	ISA	Device
Broadcom	BCM2837	●	○	○	○	v8.0	I
Unisoc	SC9863A	●	○	○	○	v8.1	M, T
Amlogic	G12A	●	○	◐	○	v8.0	I
NXP	LS1012ASN	●	◐	○	◐	v8.0	I
MediaTek	MT6739, 6765	●	○	○	○	v8.0	M, T
Samsung	Exynos 7570, 7578	●	○	○	○	v8.0	M

● Supported, ◐ Presumably supported (not publicly opened), ○ Not supported, M: Mobile phone, T: Tablet PC, I: IoT device

tiny hypervisor's trap handler. In particular, MyTEE services were invoked by MyTEE APIs that are implanted in the device drivers. Three example applications that build secure IO to a TPM, framebuffer, and a USB keyboard were developed using the APIs. During the performance evaluations, we observed negligible overhead to the overall system. By contrast, overhead of example applications was variable depending on the type of peripheral and the baseline latency. Finally, the contributions of this work can be summarized as follows:

- We propose MyTEE to build and strengthen the TEE, even without the support of mandatory TrustZone hardware extensions such as TZASC and TZPC.
- The secure IO can be established by equipping the existing device drivers with MyTEE APIs instead of hosting them in the TEE. The proposed scheme can be readily extended to build secure IO channels with various peripherals.
- The peripheral and controller device drivers for the hardware TPM, framebuffer, and USB keyboard were analyzed and then protected as MyTEE example applications to demonstrate the feasibility of our approach.

II. Background

A. ARMv8 Architecture

Here, we discuss the security states of the CPU, TrustZone extensions, and virtualization on the ARMv8.0 architecture.

Security states. On a TrustZone-enabled system, there are two CPU security states: secure and non-secure. The non-secure state provides three privilege modes: user, kernel, and hypervisor. The rich execution environment (REE) runs in this state. The security state of the CPU is changed to secure to run security-critical software in the TEE. In general, the user, kernel, and monitor modes are available in the secure state (from ARMv8.4 onward, the hypervisor mode is also supported). The software running in the monitor mode acts as a gatekeeper between the REE and the TEE, and saves and restores the context of each environment. Not limited to the gatekeeper, the monitor mode is leveraged to implement the kernel integrity monitor [27], [32].

TrustZone extensions. In addition to the security states of CPU, TrustZone hardware extensions are defined and can be integrated to a SoC. TZASC and TZMA enable the TEE memory to be isolated from the rest of the system. Specifically, they partition the memory regions and configure different access permissions for each region in accordance with the security state. For example, we can enforce that a memory region allocated to the TEE is accessed only when the CPU is in the secure state. TZPC dynamically changes the security states of peripherals to either secure or non-secure. By using this, a secure IO channel between the peripheral with secure state and the TEE can be established.

Virtualization. ARM defines hardware-assisted virtualization as a mandatory feature from ARMv8.0-A onward. The stage-2 paging translates the intermediate physical address, which is the physical address to the virtual machine view, to the machine address. In addition, access permission to the machine address can be configured in the stage-2 page table entry. Any stage-2 paging fault (e.g., access violation) is trapped to the hypervisor. The base address of the stage-2 page table is set to the hypervisor system register, the virtualization translation table base register (VTTBR). Previous work has leveraged this additional translation layer to isolate security-critical software from untrusted OS [36], [45], [55].

B. Communication with Peripherals

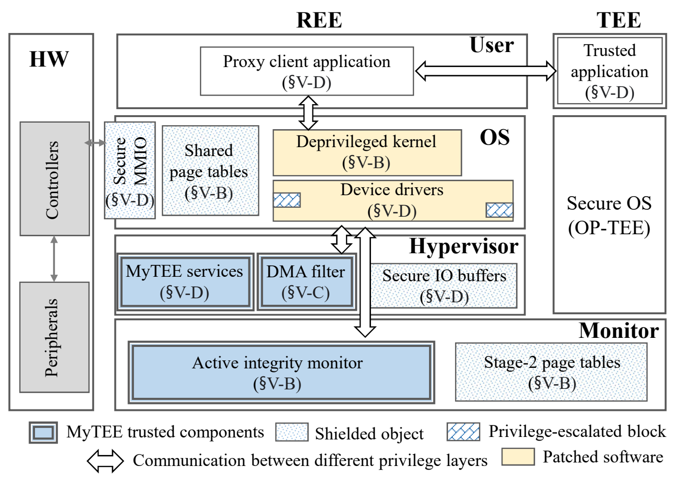
Peripherals communicate with a host machine using various protocols, such as the Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), Peripheral Component Interconnect (PCI), and the Universal Serial Bus (USB). The controller of each protocol performs operations to manage the communication, such as assigning a device address, selecting a channel for communication, and initiating the transactions. Those operations are conducted by setting memory-mapped controller registers.

A user process interacts with device drivers to communicate with peripherals. In general, device drivers for the controller (cDrv) and the peripheral (pDrv) are involved for the communication. The pDrv provides interfaces to the user process, such as `ioctl`, `write`, and `read`. Additionally, it creates a protocol (e.g., USB) message that delivers required information such as the addresses of IO buffer and peripheral.

The cDrv retrieves information from the message and configures the controller through MMIO. In addition, it reads data from the IO buffer and delivers it to the peripheral that is connected to the controller. The response from the peripheral is also read and written to the buffer. These memory operations can be fulfilled by DMA to minimize the CPU burden. For example, the cDrv for an SPI controller can set up a DMA controller such that the source and target of the DMA operations are set to the buffer and SPI FIFO register, respectively. Figure 1 illustrates the communication channel between the application and the peripheral.

III. Motivation

As discussed in Section II-A, various security extensions have been defined and can be integrated into an SoC. *However,*


REE
TEE
HW
User
Proxy client application (§V-D)
Trusted application (§V-D)
Controllers
Peripherals
Secure MMIO (§V-D)
Shared page tables (§V-B)
Deprivileged kernel (§V-B)
OS
Device drivers (§V-D)
Secure OS (OP-TEE)
Hypervisor
MyTEE services (§V-D)
DMA filter (§V-C)
Secure IO buffers (§V-D)
Monitor
Active integrity monitor (§V-B)
Stage-2 page tables (§V-B)
MyTEE trusted components
Shielded object
Privilege-escalated block
Communication between different privilege layers
Patched software
Fig. 2: Active integrity monitor for memory isolation, DMA packet filter, and secure IO constitute MyTEE.

implementation of the security extensions in an SoC is optional; hence, part or all of them are not present in some SoCs. A quick analysis of the latest trusted firmware repositories [5], [18], [21] and consultation with some vendors readily revealed such an incomplete implementation. Table I shows the example SoCs that lack the extensions. BCM2837 [7], which is deployed in production IoT devices [16] such as industrial controllers and smart hubs, does not provide TrustZone extensions. SC9863A [26], MT6739 [20], and Exynos 7570 [19] are hosted in low-end mobile phones [22]–[24] and tablet PCs [25] to reduce costs and target specific markets (e.g., developing countries). On the contrary, even if the extensions are provided, they are generally proprietary, which requires close involvement of SoC vendors in the development and deployment processes of secure services [9] and hence might hamper the time-to-market requirement. MyTEE aims to ameliorate this problem by providing an alternative solution for creating TEEs without depending on hardware security extensions.

IV. Attack Model and Assumption

In addition to the assumption of an untrusted OS, the TrustZone hardware extensions illustrated in Section II-A are assumed to be not implemented in our target system. Therefore, even if the TEE software platform is installed, it is completely unsecure [12].

In contrast, the security states and virtualization of CPU, which are defined as mandatory features on ARMv8-A, are implemented and trusted. We also assume secure boot; thus, the firmware, boot loader, and the OS kernel images are intact at boot time. This can be accomplished by placing the images in the immutable region of an SD card [14] instead of depending on a device key isolated in the TEE, which is not possible in our attack model. Moreover, the host and peripheral hardware are physically isolated and not malicious. Therefore, physical attacks such as tampering with the SD card or performing a cold boot attack [31] are not possible. Finally, side channel attacks [42], [48], [60], [61] are not considered in our attack model.

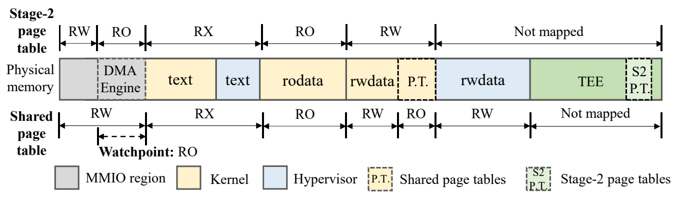


Fig. 3: Physical memory layout and memory permission settings for protecting the TEE and MyTEE components.

V. System Design

A. Overview

MyTEE aims to enable the TEE in a strictly limited environment wherein hardware security extensions are not supported. In other words, open-source TEE platforms such as OP-TEE [6] can be secured by MyTEE even if they are deployed on such limited environments. To support memory protection without TZASC, the active integrity monitor isolates memory using stage-2 paging [15] and de-privileges kernel (Section V-B). Furthermore, the DMA filter that monitors every MMIO transaction to DMA controllers is designed to prevent DMA attacks (Section V-C). For secure IO, the pDrv and cDrv are hardened using MyTEE APIs to shield the IO buffers and memory-mapped registers of the controller and to allow part of the device driver code to securely and seamlessly access them (Section V-D). Figure 2 describes the core components of MyTEE. Notably, MyTEE components running in the hypervisor and monitor modes constitute our trusted computing base (TCB). In particular, filtering DMA and supervising the critical operations enable strict isolation of the trusted components from the untrusted OS. Further trusted services such as secure IO are built based on the isolation.

B. Execution Environments Isolation

MyTEE comprises the TEE and REE components. The tiny hypervisor and MyTEE APIs implanted in the device driver constitute the REE component. The active integrity monitor as the TEE component runs in the monitor mode to emulate critical operation of the kernel and hypervisor. Both components are protected by carefully managing the kernel page tables and stage-2 page tables. First, by removing the mapping for the TEE and the hypervisor from the stage-2 page tables, MyTEE ensures that the untrusted OS cannot access both regions. The kernel text embedding MyTEE APIs calls and the hypervisor text are also enforced to be read-only. The stage-2 page tables are created as part of the secure boot.

For privilege escalation of the device drivers (Section V-D), the hypervisor uses the kernel page table to map its virtual address space instead of managing its own page table. This approach enables the privilege-escalated code block to seamlessly run with the hypervisor privilege. That is, kernel APIs and dynamically allocated kernel objects can be accessed using the kernel virtual addresses in the hypervisor without

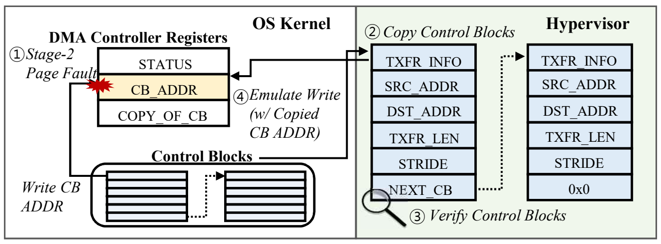


Fig. 4: MyTEE verifies DMA control blocks to prevent malicious DMAs from being triggered by an untrusted OS.

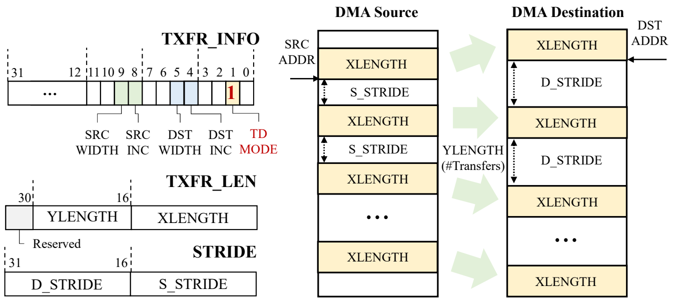


Fig. 5: In DMA two-dimensional (TD) mode, MyTEE checks the CB settings to ensure that the DMA does not access the protected region on each data transmission iteration.

explicitly creating mappings to them in a separate hypervisor page table. To this end, the kernel page table maps the entire REE (including the hypervisor), except the TEE. Nevertheless, the untrusted OS cannot access the hypervisor region due to the stage2-paging-based isolation. Furthermore, to prevent the untrusted OS or compromised hypervisor from manipulating the page tables, the active integrity monitor deprivileges the kernel and hypervisor. Similar to previous work [27], [32], the (shared) kernel page table is set to read-only and the integrity monitor verifies and emulates the page table update; it ensures that a new page table entry does not map to the physical memory that is already allocated to the protected components and invariants (e.g., TEE, hypervisor, kernel text).

Additional security-critical operations, such as managing system registers (e.g., page table base register), are also emulated. In particular, instructions for such critical operations are replaced with the secure monitor call (SMC) for entering the TEE. The stage-2 page tables are protected from the attacker who obtains the hypervisor privilege by placing them in the TEE region that is not mapped to the hypervisor or the untrusted OS. We also ensure that the security-critical instructions that manage the hypervisor system registers are not included in the hypervisor. For example, the instructions that configure VTTBR and the system control register (SCTLR) of hypervisor are removed from the memory once the registers are initialized at boot time. Finally, the watchpoint-based memory protection technique [37] was adopted to monitor privileged malicious access to the DMA controller registers (Section V-D). Figure 3 describes the memory layout with MyTEE.

C. DMA Filter

Due to the absence of TZASC, MyTEE depends on careful page table management and deprivileging for the memory protection of itself and the TEE. This might be vulnerable to DMA attacks because compromised peripherals can arbitrarily read and write any memory region. The DMA attack can be triggered by two different subjects: malicious hardware or the untrusted OS. As elaborated in Section IV, we assume that the host device is physically isolated and that peripherals on the device are not malicious. Thus, a DMA attack performed by the malicious hardware is not considered in our work. In contrast, the untrusted OS can maliciously program the DMA controller to break MyTEE security. To obviate this, MyTEE filters out malicious DMA requests that may lead to the DMA controller performing abnormal operations. Because there are two types of DMA controllers—built-in and external DMA controllers—depending on whether the controller is integrated into an individual device or externally hosted, slightly different monitoring strategies should be applied.

External DMA controller. MyTEE secures the external DMA controller by monitoring every DMA control block (CB). The CB is a data structure that specifies mandatory information for transmitting a data, such as source and destination addresses, data size, and transmission types. As can be seen in Figure 4, because the address of the CB is delivered to the DMA controller through MMIO, MyTEE traps and validates any access to the memory-mapped registers of the DMA controller. Again, stage-2 paging is leveraged to lock the registers. Accesses to the locked registers generate page faults that are trapped by the MyTEE hypervisor. MyTEE first dereferences the CB address and copies the CB to the secure memory. Because several CBs can be chained together by setting the `NEXT_CB`, which presents the 32-bit bus address of the next CB, we continue to dereference the value of `NEXT_CB` and copy the CBs until the `NEXT_CB` set to 0x0 is encountered.

Then, the copied CBs are verified to hinder the DMA from accessing any protected region. The data transmission of DMA is performed in different ways depending on the `TD_MODE` flag setting in `TXFR_INFO`. As illustrated in Figure 5, when the flag is set, the transmission between the source and destination memory buffers is performed in two-dimensional (TD) mode. The source data, the size of which is specified by `XLENGTH`, is transferred to the destination memory. Further, the transmission is performed `YLENGTH` times. `D_STRIDE` and `S_STRIDE` contain signed integers and are added to `SRC_ADDR` and `DST_ADDR`, respectively, to determine the source and destination addresses for the next transmission. The `SRC_WIDTH`, `SRC_INC`, `DST_WIDTH`, and `DST_INC` flags in `TXFR_INFO` also configure the addresses for the next transmission. Specifically, if the `SRC_INC` or `DST_INC` flag is set, each memory read from source or write to destination increases the source and destination addresses by 4 or 32 depending on the settings of `SRC_WIDTH` and `DST_WIDTH`. On the contrary, the normal (non-TD) mode works in an even simpler manner. `TXFR_LEN` as a whole specifies the amount of data to be transferred. `STRIDE` is also ignored in this mode.

Based on the observation of how the CB programs the DMA controller, MyTEE verifies expected accesses by the

DMA in advance to filter out malicious CBs. Any CB that programs the controller to perform the DMA to the immutable region (i.e., kernel text, hypervisor, TEE) is considered malicious and blocked. Finally, the address of the copied CB, instead of the original one, is written to the DMA controller register to remove the race condition between the verification and use. Once the address is written to the controller register, each field of the CB is copied into the read-only registers in the controller (`COPY_OF_CB` in Figure 4). In addition, we allow the DMA to access the secure buffers only when both the source and destination buffers are hosted in secure memory. By doing this, we prevent secrets in the buffer from being leaked or manipulated by malicious DMA.

Built-in DMA controller. The DMA controller can be integrated into other devices. On the Raspberry Pi 3, the USB controller incorporates its own DMA controller. Compared with the external DMA controller, this type of built-in DMA controller has a simpler configuration mechanism. Instead of using the DMA CB that is first located in memory and subsequently subsumed by the DMA controller, it enables an OS to directly update the memory-mapped controller registers that configure the size and address of the data transmission memory. In addition, no memory-to-memory operation occurs with this DMA controller. Either the source or destination is set to the FIFO of the peripheral that is attached to the USB hub. For example, the DMA operation for fetching the USB keyboard input is performed such that the input is read from the FIFO and written to the buffer in host memory. This nature of the built-in DMA controller simplifies the DMA filtering. The controller registers are locked by stage-2 paging to monitor any attempt to update them, but copying the CBs in secure memory is not required. In particular, updates to the registers for configuring the buffer address and its size are investigated to prevent the DMA from accessing the kernel text and data, hypervisor, TEE, and the locked registers themselves. In addition, any DMA access to the secure buffer other than that conducted as part of secure IO transactions is dropped (Section VI-B).

D. Peripheral Isolation

The absence of TZPC and TZASC hinders the establishment of the TrustZone-based secure IO. Here, we illustrate how MyTEE establishes a secure IO channel between the TA and the peripherals without depending on such TrustZone extensions.

1) Design primitives for secure IO: Previous work [40], [41], [52] enabled the secure IO between the TEE and the peripherals by configuring TZPC to set up the security state of peripherals to be secure so that they could access the TZASC-protected memory. In addition, the peripheral device driver was hosted in the TEE. Unfortunately, this approach cannot be applied to a limited environment that does not provide such hardware protection units. Porting the device driver to the TEE also enlarges the attack surface of the TEE. MyTEE addresses these limitations and satisfies the following requirements: *R1) should not bloat the TEE, R2) is easy to use and deploy, R3) ensures IO is verifiable, and R4) should not be abusable.*

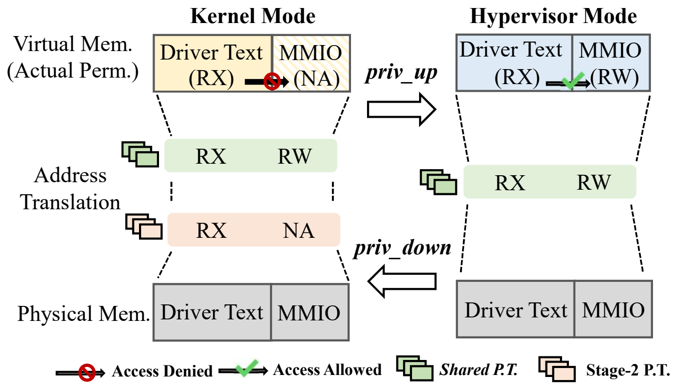


Fig. 6: Temporal privilege escalation of a device driver partition and the combination of shared and stage-2 page tables enable secure IO without bloating the TEE.

2) Temporal privilege escalation: To satisfy *R1*, we revisit the device drivers instead of hosting them in the TEE. The CA functions as a proxy to deliver the request from the TA. As already illustrated in Section II-B, the pDrv, cDrv, and controller are involved in the communication with peripherals. We protect part of them to enable secure IO, namely the memory-mapped registers of controllers and buffers that are allocated to deliver IO requests and responses. Note that depending on the type of peripheral and scenario, the requirements of preserving confidentiality or integrity of the secure buffer differ. For example, both confidentiality and integrity should be ensured for medical data in a framebuffer. By contrast, the TPM command in the buffer is not a secret but should be delivered intact to TPM.

Figure 6 illustrates how MyTEE realizes secure IO. The memory-mapped registers are set to non-accessible from the OS by managing the stage-2 paging. The predefined IO buffers in secure (hypervisor) memory are used on demand, instead of kernel-allocated buffers. Only peripheral hardware and the TA can access them. Even the device drivers that are engaged in secure IO cannot read from or write to these shielded objects. Thus, we temporarily escalate the privileges of code blocks that manage the secure buffer and register so that driver operations are not blocked. As discussed in Section V-B, a page table is shared to map both the kernel and hypervisor. However, because the actual kernel space permission is determined by the combination of settings of the kernel and stage-2 page tables, sharing the page table does not undermine the hypervisor security. That is, we can still enforce different memory permissions between the kernel and hypervisor; the shielded objects are accessible only when the privilege is escalated to the hypervisor.

The privilege escalation is conducted using MyTEE APIs–`mytee_priv_up` and `mytee_priv_down`–which wrap the selected code blocks (Table II). The APIs are implemented as hypercalls. Once the `mytee_priv_up` is invoked, the current CPU mode switches to hypervisor, a new stack is allocated in secure memory, and interrupt is disabled. Although we trust the hypervisor, the privilege-escalated code chunk could be vulnerable, giving an attacker a chance to obtain the arbitrary read/write primitives with hypervisor privilege. This might

TABLE II: MyTEE APIs for secure IO.

Name	Description
mytee_shield_mmio()	Protect the MMIO region using the stage-2 paging
mytee_priv_up()	Escalate the privilege to hypervisor
mytee_priv_down()	Restore the privilege to kernel
mytee_verify_memopr()	Validate the address and size of memory operations
mytee_log_txn()	Create a secure log for the privileged operations

lead to corrupting the DMA controller to collapse the paging-based TEE isolation. We adopted the debug watchpoint-based memory protection [37] to prevent this attack. The watchpoint is configured to monitor the MMIO region for the DMA controller at every privilege escalation. Hence, any write access to the controller generates the watchpoint exception that is trapped and handled by the trusted hypervisor. Finally, the control flow transfers to the instruction following the API invocation without depriveleging to kernel. Thus, the code block after the `mytee_priv_up` executes with hypervisor privilege and hence can access the secure buffer and register. The `mytee_priv_down` is placed at the end of the code block and draws back what is set by the `mytee_priv_up`. Note that to prevent the attacker from abusing these APIs, we enforce that only the drivers that are statically built as part of the kernel text can use the APIs. We check the link register (LR), which is automatically updated with the return address when a function is invoked, to distinguish kernel text from the loadable kernel modules (LKMs).

Thanks to the use of MyTEE APIs, a secure IO channel with a peripheral can be readily established (satisfying *R2*). This helps minimize the modification of the privileged software (i.e., hypervisor) to reflect the peripheral specific logic and data structure. For example, as discussed in Section VI-A, all commands sent to the TPM hardware must be logged for future verification by the TA. The logging is required to understand the semantics of TPM and protocol, such as the TPM command and the SPI message formats. Using MyTEE APIs enables developers to manage such peripheral specific data and logic in the relevant device drivers instead of having to adapt the MyTEE hypervisor.

3) Robustness and security: Because MyTEE API calls in the device drivers are exposed to potential attackers, they can be abused or bypassed. The attacker can simply bypass the API calls by bending the control flow to ensure that the MMIO regions are not shielded. Furthermore, the privilege-escalated code blocks can be abused to send malicious requests to the peripheral. This can be mitigated by enforcing operations in the privileged code blocks to be logged and letting TA verify the operations, which meets *R3*. The `mytee_log_txn` API supports logging, which is invoked in the privilege-escalated block to log the events in secure memory. The operation type (e.g., shielding MMIO registers, writing a command), message to the peripheral, and configuration values of the controllers are logged. The verification can be conducted as follows: check (1) whether the buffer and MMIO regions are shielded during a transaction and (2) whether the request delivered to the peripherals is intact. There could be malicious and multiple accesses to MMIO. However, logging performed in the privileged block is non-bypassable, as long as the API is properly implanted. Once an unexpected log is detected (e.g., TPM command tampering), the TA can simply stop or continue

Listing 1: Original function with `writel`.

```
static int bcm2835_send_data(struct mbox_chan *
    link, void *data){
   ...
   writel(msg, mbox->regs + MAIL1_WRT);
   ...
}
```

Listing 2: Instrumented function with MyTEE APIs.

```
/** A new wrapper function is defined **/
+ mytee_wrapper_writel(u32 msg, u32 mmio_addr)
+ {
+   int ret;
+   ret = mytee_verify_memopr(MAILBOX_WRT, \
+   mmio_addr, sizeof(u32));
+   if (!ret){
+      mytee_log_txn(MAILBOX_WRT, msg);
+      writel(msg, mmio_addr);
+   }
+ }

static int bcm2835_send_data(struct mbox_chan *
    link, void *data)
{
   ...
/** Privilege escalation for accessing **/
/** the locked memory-mapped registers **/
+   mytee_priv_up();
+   mytee_wrapper_writel(msg, mbox->regs + \
+   MAIL1_WRT);
+   mytee_priv_down();
   ...
}
```

iterating the transaction until it is normally achieved.

The memory operation (e.g., `writel`) in the privilege-escalated blocks should not be abused (*R4*). To this end, MyTEE supports the `mytee_verify_memopr` API that verifies the address and size of the memory operation in the privileged block. Because the addresses of the secure objects are deterministic, e.g., the MMIO regions for peripherals are defined and fixed by the SoC manufacturer, the verification is simply conducted by checking whether the memory operation is performed within the valid memory range. Note that all relevant parameters (e.g., addresses of the source and target) and their dereferenced values are marshaled into the secure stack in the hypervisor memory before the verification to prevent time-of-check-to-time-of-use (TOCTOU) attacks.

Furthermore, the privilege-escalated driver chunks could be vulnerable. However, even if the attacker obtains the hypervisor-privileged "write-what-where" primitive by exploiting the vulnerability, the read-only regions are still immutable because the TEE (i.e., active integrity monitor) is the sole anchor to update the page tables. Although the writable objects such as the log can be corrupted, the effectiveness of the attack is confined to the REE as the device driver is hosted outside the TEE; thus, the exploitation cannot directly corrupt the TEE.

Finally, like general TEE services, MyTEE-protected services are also vulnerable to the denial-of-service (DoS) attack. However, as shown in Section VI, MyTEE provides a way to build an indicator for secure IO activation, which prevents masquerading unprotected IO as secure IO. In addition, simultaneous accesses to the IO devices from the REE and TEE are handled in a way that TEE access is always given higher priority. This approach aligns with general TEE-based secure IOs that freeze the REE while conducting the IO from the TEE [46], [52].

4) Instrumentation example: Listing 2 demonstrates an example instrumentation of a device driver with MyTEE. On Raspberry Pi, the OS can communicate with the GPU using the Mailbox protocol [10]. The function `bcm2835_send_data` aims to write a message through MMIO to the Mailbox controller so that it is delivered to the GPU. In our example, the MMIO region is assumed to be already shielded, and we attempt to log every message written to the region. As can be seen in Listing 2, we first put the original `writel` function in the new function, `mytee_wrapper_writel`, with additional MyTEE APIs for verifying and logging the write. One of the input parameters of the APIs, `MAILBOX_WRT`, is a constant value that indicates the current context for logging. It is referenced in the verification function to retrieve a valid address range for the MMIO with the Mailbox controller. The developer can add finer-grained filtering logic to capture only a specific type of request message (e.g., display configuration) that is delivered by the Mailbox. Because the `mytee_wrapper_writel` is wrapped again with `mytee_priv_up` and `mytee_priv_down`, it operates with the hypervisor privilege and thus can access the shielded MMIO region. Note that in this example we do not marshal the parameters for `mytee_wrapper_writel` because their size is 4 bytes; therefore, they are delivered using general registers. However, passing address taken variables as arguments requires marshalling in the secure stack.

VI. Secure IO Applications

In this section, three examples are illustrated to show how the communication channels between the TEE and peripherals, i.e., the hardware TPM, USB keyboard, and framebuffer, can be hardened with MyTEE. Although implementation (particularly how the drivers are instrumented with MyTEE APIs) can be variable depending on peripherals, the MyTEE-supported secure IO can be exemplified as shown in Figure 7.

A. Secure TPM

For PoC of secure TPM, we used SLB 9670 TPM [11] from Infineon. It is a hardware implementation of TPM 2.0 and connected to the GPIO pins on Raspberry Pi 3. The communication between the OS and TPM is performed using SPI protocol. The device drivers for TPM (pDrv) and SPI controller (cDrv) are integrated in the Linux mainstream. Any request from a user application bounding for the TPM is first delivered to pDrv and formatted as a SPI message. Then, the message is written to the TPM through MMIO to the controller. The response from the TPM is also read from the controller through MMIO. Both MMIOs are performed by cDrv.

The secure IO to TPM is built and works as follows. The TA creates a TPM command and delivers it to the proxy CA. In turn, the CA delivers the command with a flag requesting the secure IO to the pDrv. The MMIO region for the SPI

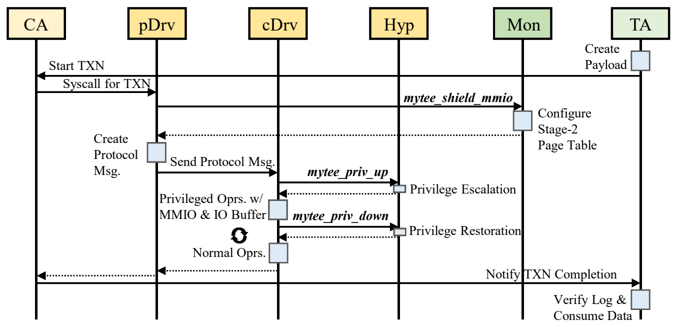


Fig. 7: Communication process for secure IO.

controller is protected by the `mytee_shield_mmio` in the pDrv and this shielding operation is also logged in the secure (hypervisor) memory. After the TPM is initialized by an `init` command sent by the pDrv, the SPI message that embeds the command and flag from the CA is sent to the cDrv. Once the cDrv checks the flag, it performs the privilege-escalated MMIO. To realize this, the `bcm2835_wr` function is instrumented in a similar manner as that illustrated in Listing 2. The function is placed in the new wrapper function, then the `mytee_priv_up` and `mytee_priv_down` APIs are invoked before and after the wrapper function. Additional APIs for parameter marshalling, memory write verification, and command logging are placed in the wrapped function. Specifically, we marshalled the command and its length field in the SPI message. The `bcm2835_wr` function writes the command to the FIFO of the slave (i.e., TPM); one byte is written per iteration in a loop following the SPI transmission protocol. Once all the commands are passed to the TPM, the privilege is reverted back to the kernel.

Reading a response from TPM requires privileged read access to the shielded MMIO region. Therefore, the `bcm2835_rd` function is instrumented in a similar manner. The read operation is also fulfilled with 1 byte granularity. Each byte from the TPM is written to the buffer in secure memory instead of the one in the SPI message. Later, the TA verifies whether its TPM request is managed in a trustworthy manner by checking the logs and then consumes the output in the secure buffer. Note that the header of the TPM response message contains a status flag for the transaction. A response message with a success status delivers the output (e.g., a random number) depending on the type of command. Thus, only responses with success status are filtered and written to the secure buffer. In addition, we also maintain a transaction flag that indicates the completion of a single transaction to prevent further (malicious) requests from being made before the TA consumes the output. The flag is checked as part of the validation of MMIO for writing a TPM command and is cleared by the TA after the consumption of the TPM output.

B. Trusted USB Keyboard

In this section, we first elaborate the communication mechanism between the host and general USB devices. Then, we demonstrate how MyTEE can establish a secure IO with the USB devices by providing an example of hardening communication with a USB keyboard.

1) General USB Peripheral: USB devices can have several configurations. Each configuration can also define several interfaces. For example, a USB headset has interfaces for a microphone and for audio. Each individual interface has at least one endpoint, which is a hardware buffer at the peripheral for communication between the host and the peripheral. There are two types of endpoints: control and data. The control endpoint aims to set up and control the USB device (e.g., LED control on keyboard), and endpoint0 is dedicated for this purpose. The data endpoint is optional and is defined for data transmission. The direction of the endpoint is defined based on the host. For example, the IN endpoint is used for transmission from the peripheral to the host, whereas the OUT endpoint transmits data from the host to the device.

On Linux, the USB Request Block (URB) is a data structure defined for data transmission with USB peripherals. The URB is created by the pDrv and contains all the information required for the transaction (e.g., endpoint, device address, buffer address). Once the cDrv for the USB controller receives the URB from the pDrv, it configures the USB controller based on the information in the URB to start the transaction. In general, USB controllers support several channels for communication with peripherals. Each channel has the same set of configuration registers that are mapped in host memory. As part of bootstrapping a transaction, the cDrv selects an idle channel and configures it based on the information in the URB, such as the device address and endpoint. The buffer address in the URB can also be configured to the controller to enable the DMA between the host and the peripheral.

The completion of the DMA incurs an interrupt, which is trapped by a kernel thread that finally invokes the interrupt handler registered by pDrv. The handling of interrupts is performed based on the updated information in the URB. The handler first checks the status flag that indicates whether the previous transaction triggered by the URB is successful. The URB is then resent to the peripheral if the status flag sets to failure. The handler executes peripheral specific service routines to further handle the URB with a success flag. For example, the URB is referenced to obtain a scan code in the buffer. The handler converts the scan code into the key code based on keymap [8]. Furthermore, the keyboard input is written into a file so that applications can consume the events (e.g., /dev/input/eventX).

2) Trusted Keyboard: We designed the trusted keyboard with MyTEE based on the analysis of USB device drivers in Linux. The following security requirements were considered for the trusted keyboard design: (1) A user should be able to recognize whether secure IO with the keyboard is currently established. (2) Confidentiality and integrity of the communication should be guaranteed.

Control message inspection. We used the LED for the NUM Lock key as an indicator for secure IO activation. Thus, the LED should be turned on only when the TA initiates and requests the trusted keyboard. The TA first sets a trusted keyboard flag in the hypervisor memory and asks the proxy CA to turn on the LED. Then, the CA simply turns on the NUM Lock LED using `xset` utility [17]. This triggers the pDrv of the USB keyboard. In the pDrv, the URB for the control message is created and sent to the cDrv of the USB controller. Note that as illustrated in Section V-C, every DMA packet is monitored. This locks not only the DMA address register but also all controller registers due to the granularity of stage-2 paging, the minimal size of which is 4 KB. Therefore, we can essentially trap any attempt to configure the controller for URB transmission.

Benefiting from this property, we additionally implemented the trap handler in the hypervisor to monitor the channel configuration as well as the DMA packet and thus enable access control to the LED. Any attempt to write the device (peripheral) address and buffer address to the controller registers for the channel configuration is monitored. First, by decoding the value written to the channel characteristic register (`hcchar`), the device address can be retrieved. We compare it with the predefined keyboard address to filter out URBs for other peripherals. Furthermore, the current channel number can be obtained based on the trapped stage-2 page fault address. Specifically, because each channel has the same memory layout of configuration registers that are placed in the contiguous memory, we can obtain the channel number based on the offset between the first channel address and the fault address. The following setup of buffer address, which is used by the DMA controller, also incurs a stage-2 page fault. We can similarly obtain the current channel number based on the fault address. If the current channel is one that was assigned for the keyboard, we can copy and decode the content of the buffer in the secure (hypervisor) memory. MyTEE turns on the LED only when the LED control message is found from the decoded content and the trusted keyboard flag is set.

Keyboard input protection. The key input is also protected by monitoring the MMIO to the controller registers. We retrieve the device address and channel number based on the stage-2 page fault in the same manner as shown for control message inspection. The only difference pertains to how we manage the stage-2 paging fault that is incurred by setting up the buffer address for DMA. The buffer is updated by DMA with a fetched scan code. The integrity and confidentiality of the scan code should be protected. To this end, MyTEE switches the address of the buffer with that of the secure buffer allocated in the hypervisor region when handling the trapped MMIO. By doing so, we can ensure that the untrusted OS cannot access the buffer, but it is still possible to fetch the scan code because the hardware (i.e., DMA controller) can still access the secure buffer.

As part of handling the interrupt due to completion of DMA, the secure buffer is accessed to execute peripheral-specific logic, such as scan code conversion. Therefore, escalated privilege is required to read data from the protected memory. The approach that escalates the privilege of driver code blocks shown in Listing 2 can be adopted to achieve this; that is, the code blocks that access the secure buffer can be wrapped with `mytee_priv_up` and `mytee_priv_down` APIs. For simplicity of implementation, we developed a new function that integrates mandatory and minimal operations for translating and protecting the keyboard input. Then, we wrap the function in its entirety using the MyTEE APIs instead of wrapping the scattered code blocks in the kernel that access the secure buffer. In the function, the scan code is converted to the key code. In addition, the key code is written to the secure memory for future use by the TA.

Note that further operations can be performed by the device driver, beyond the translation of key code. For example, provided that the keyboard supports volume control keys, the driver can notify a key-press event to another (e.g., speaker) device driver. We let the unprivileged part of the driver continue to conduct such tasks in our example implementation. Also note that the interrupts generated due to the key events are not directly handled in the TEE. Instead, the OS and device driver instrumented with MyTEE take over the task. Specifically, each key event is processed in the REE until the CA finishes collecting user input. Then, the key codes stacked in the secure memory are consumed by the TA. Our approach has clear advantages over directly handling the interrupt in the TEE, which requires additional changes to the OS and TEE for forwarding (and handling) interrupts to (and in) the TEE and thus complicates system design and bloats the TEE.

Tracking device address. In our example, the device address should be kept updated for reliable peripheral identification. To this end, we monitor two standard control messages: `GET_DESCRIPTOR` and `SET_ADDRESS`. USB devices have a default device address of 0x0 when they are first connected to the USB port. The OS sends a URB with the `GET_DESCRIPTOR` to get the information such as USB class and protocol of the device. MyTEE hooks this to detect the keyboard connection. Then, the following URB with the `SET_ADDRESS`, which aims to assign a new device address for the device, is further monitored.

C. Trusted Display

In this section, we demonstrate how MyTEE protects the framebuffer for trusted displays. On Raspberry Pi, the videocore (i.e., GPU) manages configuration of the display, such as the framebuffer allocation and display resolution setup. The OS can send a request for the display configuration to the videocore. The inter-processor communication between the CPU and videocore is conducted through the Mailbox controller. Any request or response is exchanged by performing MMIO to the Mailbox controller registers.

For secure IO, we first ensure that the display configuration is not maliciously changed. Toward this end, we lock the memory-mapped registers of the Mailbox controller by leveraging stage-2 paging. The `mytee_shield_mmio` is invoked in the `ioctl` handler for display configuration (i.e., `FBIOPUT_VSCREENINFO`) in the pDrv. Furthermore, the cDrv of Mailbox controller is patched in a similar fashion as that shown in Listing 2 to enable access to the shielded registers under MyTEE supervision. To prevent TOCTTOU attacks, we restrict the display configuration to be performed only once after the Mailbox controller registers are locked. The validity of configuration is checked against the TA's original request right before updating the framebuffer. In general, there are two interfaces for manipulating the framebuffer: `write` and `mmap`. MyTEE supports both mechanisms to realize the trusted output as follows.

`mmap`-based access. The pDrv generally provides the `mmap` mechanism to enable the user application to directly access the framebuffer. To support `mmap` in establishing secure IO, a privileged block with MyTEE APIs is placed at the end of the `mmap` handler in the pDrv. When the proxy CA maps the framebuffer to handle the `mmap` request from the TA, the patched `mmap` handler is triggered. The physical address of the framebuffer is obtained as part of general `mmap` handling process. Then, the privileged block validates the address and size for `mmap` to prevent any of the MyTEE-protected regions from being maliciously mapped to the TA (e.g., Boomerang attack [44]). Furthermore, the valid region is shielded and its (physical) address and size are written to the secure memory. Subsequently, the TA verifies the controller- and framebuffer-related logs and asks the TEE OS to directly map the framebuffer to its address space by referring to the information in the secure memory.

TABLE III: Lines of code (LoC) in C for MyTEE TCB and secure IO examples.

Trusted component		
Monitor*	Hypervisor	Kernel*
517 + 1492 (ASM)	1552 (ASM)	673
Secure IO (driver patch + TA)		
TPM	USB keyboard	Framebuffer
159 + 516	96 + 156	168 + 201

[*] Include TZ-RKP adoption

`write` system call. The `write` handler in the pDrv supports directly writing data to the framebuffer. Enabling this in secure IO requires the system to ensure that the written data cannot be leaked or manipulated by the untrusted OS. To this end, the TA first places the secret data in the secure memory and asks the CA to invoke the `write` system call with dummy data. Similar to the `mmap`-based approach, the `write` handler of pDrv is instrumented such that it entails the privileged code block that shields the framebuffer and validates the memory operations. The secret data is then copied from the secure memory to the shielded framebuffer. Note that because the task of updating the framebuffer is delegated to the instrumented device driver (unlike in the `mmap` support), log verification is performed in the privileged block. Also note that an indicator for successful establishment of a secure IO channel for both approaches can be similarly implemented, as shown in Section VI-B2.

VII. Implementation

The PoC of MyTEE was implemented on a Raspberry Pi 3 development board equipped with Broadcom BCM2837 SoC. The SoC was integrated with a quad-core ARM Cortex-A53 processor, DesignWare USB 2.0 controller [1], DMA controller with 16 channels, and SPI controller. Specifically, as security features, only the security state of the CPU is supported; TrustZone extensions such as TZPC, TZASC, TZMA are missing in this SoC. The OP-TEE and Linux (ver. 4.14) were deployed for the TEE and REE software platforms, respectively. Table III shows the size of trusted components and device driver patches (and TAs) for secure IO. Note that 673 LoC in the kernel and 821 LoC (ASM) in the monitor are added for the TZ-RKP [27] adoption.

A. ARM Trusted Firmware

We updated the ARM trusted firmware to create the stage-2 page tables for isolating TEE and hypervisor from the untrusted OS. Specifically, the memory region for security-critical

TABLE IV: Protected security-critical system registers for OS.

System register	Description
SCTLR	Controls MMU, instruction alignment, etc.
TTBRx	Sets the base address of page tables
TTBCR	Configures memory translation attributes
DACR	Defines access permissions for memory domains

objects (e.g., hypervisor stack, secure buffer) including the tables themselves is not mapped in the stage-2 translation layer. Runtime management of the stage-2 page tables for shielding the MMIO region of controllers on demand is performed in the monitor mode (i.e., `BL31` in the trusted firmware). Furthermore, emulation of critical kernel operations such as the update of page tables and system registers is conducted in the monitor mode as well.

B. OS Kernel and Drivers

The kernel page table is shared with the hypervisor (Section V-D). Thus, the page table format and virtual address range supported by the kernel and hypervisor should be equivalent. As discussed in Section XI-B, the virtual address range supported by the hypervisor mode is smaller than that by the kernel mode when 64-bit virtual addresses are used. By contrast, the supported address range is same with the 32-bit address mode. Two types of page table descriptor formats for this mode are available: short descriptors and long descriptors [2]. However, the hypervisor mode only supports the long descriptor format. Thus, we compiled the kernel as a 32-bit binary with the large physical address extension option (i.e., `CONFIG_ARM_LPAE` in the `.config` file) so that the kernel also uses the long descriptor format.

Once the kernel text, data, and hypervisor are loaded in the memory, they are protected by stage-2 paging. The shielding request is generated by the `init` process that is instrumented to invoke the MyTEE. Critical operations, such as the page table update, are patched with SMC for their verification and emulation by MyTEE. We refer to the Samsung open-source project [13], which provides the kernel patch for TZ-RKP implementation, to locate such critical operations in the kernel source. Table IV lists the critical system registers that are dynamically reconfigured in the kernel at runtime, thereby resulting in the operation for the configuration being emulated instead of simply being removed. For example, `TTBR0` is set at context switches and `SCTLR` is configured to activate the memory alignment trap at every kernel entry from user mode.

C. Hypervisor

The tiny hypervisor is implemented in ASM from scratch. For page table sharing, we configured the hyp translation table base register (`HTTBR`) with the same value in the translation table base register (`TTBR1`) that contains the kernel page table address. This configuration is conducted at boot time and the instruction for setting HTTBR is nullified to ensure that it is not abused by the attacker. The hypercall handler is implemented to support MyTEE APIs such as `mytee_priv_up`. The context of the kernel is stored in the saved program status register (`SPSR`) when the hypercall is invoked. In general, the saved context is restored by using the `eret` instruction when returning to kernel. Instead, we use the `ret` instruction to return to kernel without deprivileging so that the chosen (wrapped) blocks of device drivers run with the hypervisor privilege. Note that the watchpoint-based DMA register protection was not implemented because configuring watchpoints for the 32-bit hypervisor is not supported in the ARM architecture [2] (viz., Section XI-B).

D. OP-TEE

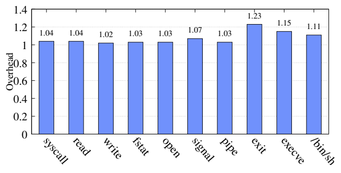
In our use cases, some OS-level services need to be supported for the TA implementation. For example, the page table mapping to the secure memory in the hypervisor region needs to be created to enable the TA to access the secure buffers. To this end, OP-TEE APIs were leveraged instead of adding new system services. In contrast, the peripheral specific logic needed to build secure IO, such as TPM command generation and log verification, was implemented in the TA. Note that we used a GitHub branch for OP-TEE [4] that supports Raspberry Pi OS (i.e., Raspbian). However, there is no technical difficulty in applying MyTEE to the official version of OP-TEE [6].

VIII. Security Analysis

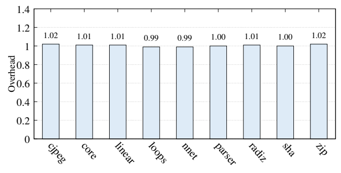
Race condition attack. MyTEE logs operations in the privileged blocks and enables the TA to verify them later. However, because the APIs can be arbitrarily invoked by the untrusted OS, TOCTOU attacks can be conducted to reduce or eliminate the effectiveness of log verification. For example, an untrusted OS can attempt to lead the TPM to overwrite a random number requested by the TA with a known hash value right after the TA verifies the TPM command log. To remove this race condition, we maintain the transaction flag that is set when any transaction is completed and checked before a new command is delivered to the TPM (Section VI-A). In the trusted display example, a similar race condition attack can be conducted by having multiple cores write to the same shielded framebuffer. However, in the privileged block, the source of the write operation to the framebuffer is always fixed to the secure buffer that only can be updated by TA. Thus, exploiting the race condition is not beneficial for the untrusted OS.

Abusing MyTEE APIs. The privilege escalation primitive can be abused to undermine the security of TEE as well as MyTEE itself. Because the TEE is not mapped in the shared page table, the privilege-escalated malicious code cannot directly access the TEE. In addition, the page table is set to read-only, preventing its manipulation. However, the attacker can attempt to update the page table base register (i.e., HTTBR, VTTBR) with a malicious page table that has a mapping to the TEE. Thus, we nullify critical instructions that configure the system registers of hypervisor at boot time. In addition, we ensure that the `mytee_priv_up` API is executed within the static kernel text, not in the dynamically loaded modules (e.g., LKMs) that are potentially malicious. The privileged memory operation can also be abused. To prevent this, we verify the memory operation with the context information that indicates the target peripheral joining the secure IO. By doing so, we can restrict the memory operation to be performed within the predefined regions of controller registers and secure buffers.

Exploiting vulnerabilities. The privileged code block is small enough to be formally verified. Even if the attacker



(a) LMBench



(b) CoreMark-PRO

Fig. 8: LMBench and CoreMark-PRO results of MyTEE normalized to that of Linux. Maximum overhead of 23% was observed with `exit` in LMBench.

hijacks the control flow by exploiting the vulnerability in the privileged block, the TEE is still secure because no critical instructions, such as those for configuring the HTTBR, exist in the executable region. In addition, any illegitimate writing to the DMA controller registers is monitored by leveraging the watchpoint-based memory protection. Colluding with the dynamically loaded kernel modules that contain such critical instructions possibly empowers the attacker. However, this can still be addressed by adding further security facilities for the privilege escalation. For example, we can remove the mapping to all the executable regions other than the static kernel text when the privilege is escalated. We leave this as a future work.

IX. Performance Evaluation

Adopting MyTEE imposes overhead to the OS, due to the stage-2 paging for memory isolation and filtering DMA packets. We measured this overhead by running LMBench and CoreMark-PRO. Additionally, the performance of DMA and three secure IO examples with TPM, USB keyboard, and framebuffer was measured.

A. LMBench

We ran LMBench to measure the rudimentary operations of an OS. Figure 8(a) describes the average latency of 10 iterations of the tests in LMBench. We observed a minimum overhead of 2% and a maximum overhead of 23% for `write` and `exit`, respectively. `execve` and `/bin/sh` also resulted in relatively high overhead of 15% and 11%. These high overheads were incurred due to not only enabling the stage-2 paging but also emulation of critical operations, which are performed in the monitor mode. Specifically, `exit`, `execve`, and `/bin/sh` generate more page faults to fork a new process and more context switches due to relatively longer execution times compared to other tests. Therefore, more overhead for switching CPU modes and emulating the critical operations was imposed.

B. CoreMark-PRO

The CoreMark-PRO benchmark supports five integer and four floating point workloads to measure the performance of the processor and memory subsystem. We ran each workload 10 times and evaluated the average latency. As can be seen in Figure 8(b), the observed overhead was negligible. Most tests were observed with overheads of 1 to 2%. Considering the fact that the average runtime of CoreMark-PRO tests was much longer (approximately 22426×) than that of LMBench, presumably, the overhead was somewhat obscured when the overall runtime increased beyond a certain amount of time.

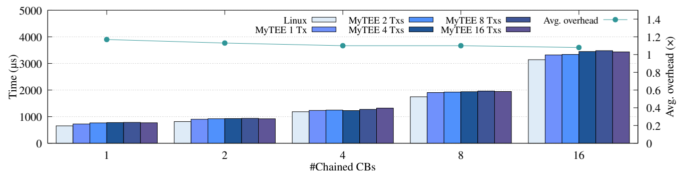
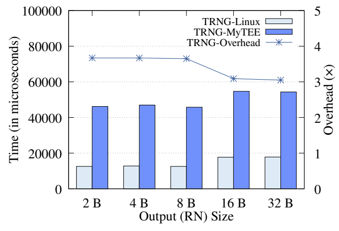
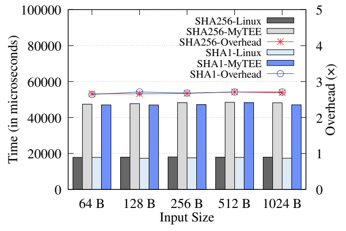


Fig. 9: Performance of memory-to-memory data transmission in DMA TD mode when the CBs are verified by MyTEE. The verification overhead was reduced to 8% with the larger number of chained CBs.



(a) TRNG



(b) Hashing

Fig. 10: Performance of a hardware TPM for random number generation and hashing with SHA1 and SHA256 (in μs.)

C. DMA Performance

The overhead imposed by DMA filtering was evaluated by measuring the time for configuring the DMA controller, which is trapped, verified, and emulated by MyTEE. Specifically, we created CBs that program the DMA controller to perform memory-to-memory data transmission in TD mode. The number of chained CBs and the number of transmissions defined by a single CB both ranged from 1 to 16. Figure 9 presents the results. The time includes the latency for preparing the CBs (e.g., allocating non-cacheable memory) as well as the MMIO with the controller. The performance impact of the number of transmissions by each CB was negligible. By contrast, the average overhead decreased from 17% to 8% as the number of CBs increased. According to our analysis, this is because the time for MyTEE verification—which consists of copying CBs and verifying the expected memory access—was smaller than that for creating CBs; thus, the overhead gradually decreased with the larger number of chained CBs.

D. Trusted Applications

1) Hardware TPM: We evaluated the overhead imposed on TPM resulting from the adoption of MyTEE. Among TPM operations, we specifically measured the latency of random number generation and hashing with SHA256 and SHA1 algorithms. The size of random numbers generated by TPM ranges from 2 bytes to 32 bytes. For hashing, the input sizes range from 64 to 1024 bytes. Each TPM operation was performed 10 times, and the average latency is described in Figure 10. We observed a performance degradation of 3.68×

TABLE V: Performance of updating a framebuffer with write and mmap (in ms.)

Input size	**1.6 MB** Baseline	**1.6 MB** MyTEE	**3.2 MB** Baseline	**3.2 MB** MyTEE	**6.4 MB** Baseline	**6.4 MB** MyTEE
write	51.80	59.02 (1.14×)	93.97	64.46 (0.68×)	149.63	93.32 (0.62×)
mmap	56.44	44.06 (0.78×)	97.15	63.64 (0.66×)	143.10	75.63 (0.53×)

TABLE VI: Performance of a trusted USB keyboard.

Kernel service (in μs.)		**Secure key input application (in sec.)**			
URB trans.	**IRQ handling**	**125**[†]	**250**[†]	**500**[†]	**1000**[†]
94.2 (3.05×)	64.7 (1.33×)	2.8 (1.05×)	6.8 (1.08×)	14.1 (1.09×)	28.8 (1.07×)

[†] # of virtual key inputs

with the generation of a 2-byte random number. However, the performance slightly improved as the output size increased. In contrast, performance degradation by maximum 2.71× was observed in hashing. The overhead was incurred due to additional privileged operations in device drivers such as logging TPM commands and protecting TPM output. Moreover, the latency for communication between the CA and TA is also included in the overhead. In the case of random number generation, a tendency was observed wherein the overhead slightly decreased as the size of the random number increased. This stems from the fact that the overhead imposed by MyTEE is almost invariable, whereas the time taken by TPM for the random number generation is generally affected by the size of the random number. By contrast, because the impact of input size was negligible in the hash operation of TPM, a nearly invariable overhead was observed in the hash tests, regardless of input size.

2) Framebuffer: The overhead of framebuffer protection was evaluated. To this end, we compared the performance of writing data, the size of which ranged from 1.6 MB to 6.4 MB, to the shielded and non-shielded framebuffers. The data was linearly written from the starting point of the framebuffer memory. Both `mmap`-based and `write`-based approaches were evaluated. Table V illustrates the average latency of 10 runs of each test and the resulting overhead. We observed 14% overhead in writing 1.6 MB data with the `write`-based approach. However, writing larger data improved the performance: 32% and 38% with 3.2 MB and 6.4 MB data, respectively. With the `mmap`-based approach, writing to the protected framebuffer outperformed in all test cases. Specifically, we observed a performance improvement of 47% when 6.4 MB data was written to the framebuffer with `mmap`. According to our analysis, the performance improved due to the settings for the privilege escalation of the code block that accesses the protected framebuffer. That is, because the interrupt is disabled in the privileged block, the write operation can be atomically performed. By contrast, in baseline, several context switches occur during the write and the time consumed for the switches was large enough to obscure the overhead of MyTEE.

3) USB Keyboard: The performance of the protected USB keyboard was evaluated in two aspects: (1) overhead of handling a single key stroke in kernel and (2) performance of an application that generates secure key inputs. One cycle of handling a key input in kernel consists of two phases: sending a URB to read the scan code from the keyboard via DMA and handling the interrupt that is raised by the completion of the DMA. The performance of these two phases was separately measured and the average latency of 10 runs for each phase is shown in Table VI. For sending a URB, approximately 3× the amount of time was required due to the adoption of MyTEE. In particular, overhead was incurred by handling stage-2 paging faults to access the shielded MMIO region of the USB controller. As described in Section VI-B2, the faults are monitored to track the currently assigned channel and to configure a secure buffer as the DMA destination. The performance of interrupt handling was evaluated by measuring the execution time of `hid_irq_in` function, which defines several branches for handling URBs depending on their status fields. The status field indicates whether the sent URB was successfully processed, or whether an error was encountered. We only measured the time for handling a successful URB, the `urb->status` field of which was cleared, and observed 33% overhead. As illustrated in Section VI-B2, the new function that translates scan code and is wrapped as a privileged code block was the main cause of the overhead.

In the application benchmark, multiple key inputs between 125 and 1000 were virtually generated. We added 10 ms delay between each key input and wrote the same number of characters to the framebuffer. The overhead of application was between 5% and 9%. We expect that the overhead is reasonable in general secure IO scenarios in that the input size is even smaller (e.g., passcode, OTP) and human input speed is slower and more variable compared to virtual key inputs and thus causes more factors (e.g., interrupt) to obscure the overhead.

X. Related Work

A. Adoption of TrustZone

Previous research has proposed security facilities in the TrustZone-based TEE that are protected by configuring the TZASC and TZMA. TrustShadow [33] hosts a lightweight agent in the TEE that isolates unmodified applications from the untrusted OS while still enabling them to communicate with OS services. TZ-RKP [27] and Sprobes [32] implemented the kernel integrity monitor in the TEE. SeCReT [38] and Ginseng [59] hook untrusted OS services and verified them in the TEE to authenticate the CA and protect the application secrets, respectively. The stealthy malware analyzer [47], software TPM [49], memory dumper [53], and the mobile device management (MDM) service [29] also leverage TrustZone for their isolation. Some research has dynamically reconfigured the TZASC to realize flexible isolation. That is, the TZASC is set up on demand at runtime to shield applications on a per core basis [28] and build additional secure execution environments in the untrusted OS [30], [54]. The TZPC has also been utilized to isolate peripherals for their exclusive use by the TEE. Adattester [40] configured the TZPC to isolate the keypad and display from the untrusted OS, thereby preventing ad frauds. Similarly, TrustOTP [52] proposed a TEE-based onetime password (OTP) by isolating such IO peripherals. Previous research was essentially conducted based on the assumption that TrustZone and its extensions are properly deployed and available to anyone who needs to utilize them. By contrast, MyTEE attempts to assure the security and universality of TEE, even if that assumption is not satisfied. Approaches for TrustZone virtualization [34], [39] and creating a new isolated execution environment [36] can also be revisited to create the TEE without leveraging the TrustZone hardware extensions. However, MyTEE is still advantageous over such approaches in that it enables secure IO without crowding the TEE and is not limited to emulating missing features.

B. *Secure IO*

To build a secure IO channel, previous work has proposed ways to secure a device driver that is an interface to a peripheral. Wimpy Kernel [62] deprivileges the critical part of the USB device driver and protects it as a shielded user process. Trusted Display [58] adds and shields a kernel component that mediates and emulates security-critical accesses to GPU. These works require thorough analysis on device drivers to separate the security-critical logics of device drivers. Furthermore, engineering efforts to port security hypervisor framework [45], [55] to embedded devices is expected. By contrast, MyTEE provides APIs that can be implanted in existing device drivers, instead of partitioning and shielding the device drivers. Furthermore, the adoption of a de-facto standard public TEE platform (e.g., OP-TEE [6]) minimizes the complexity of hypervisor implementation by delegating the task for shielding a user process to the TEE. BitVisor [51] migrates critical parts of the device driver to the hypervisor layer, which also requires considerable analysis and engineering effort and enlarges the trusted computing base (TCB). Tabellion [46] provides the hypervisor and TEE APIs that are invoked in the device drivers to protect IO buffers. We propose a more flexible approach in that the peripheral specific logics (e.g., accessing IO buffers in URBs) that need to be executed in the higher privilege can be integrated as part of the device driver instead of migrating to the privileged software. AdAttester [40] and TrustOTP [52] maintain separate device drivers in the TEE, potentially enlarging the attack surface of the TEE. Furthermore, the presence of TrustZone extensions is assumed, which is not always true depending on SoC design.

XI. Discussion and Future Work

A. *Usability and Security*

Adopting MyTEE to build a secure IO channel requires in-depth analysis of device drivers to pinpoint proper locations for implanting MyTEE APIs. However, other than analyses that would require significant effort depending on the complexity of driver implementation, it is straightforward to equip the driver with MyTEE APIs, as described in Listing 2. One of the concerns with this approach is escalating the privilege of memory operation, which potentially enlarges attack surfaces. To address this, we validate arguments for memory copy operations performed in the privileged block. In addition, because only the simple memory copy APIs, such as `writel`, are limitedly privilege-escalated in our general usage model, we can readily perform formal verification against instrumented (privileged) code blocks, including the APIs. Developers might misuse MyTEE APIs, creating new security holes. For example, missing logging might break the reliability of secure IO. Checking the correctness of privileged block implementations at compile time can be a reasonable solution, but we leave this for future work.

B. *64-bit OS Support*

Although ARMv8 supports both 32-bit and 64-bit virtual addresses, the OS and MyTEE hypervisor are compiled as 32-bit binaries in our PoC. This aims to enable page table sharing between the kernel and hypervisor in our development environment (i.e., Raspberry Pi 3), which is equipped with a Cortex-A53 processor. In particular, the processor implements ARMv8.0 specifications that support different virtual address ranges for the kernel and hypervisor modes. The kernel mode supports two virtual address subranges: the bottom range of `0x0` to `0x0000_FFFF_FFFF_FFFF` and the top range of `0xFFFF_0000_0000_0000` to `0xFFFF_FFFF_FFFF_FFFF`. The 64-bit Linux maps the user and kernel spaces to the bottom and top ranges by maintaining different page tables for each space. By contrast, the 64-bit hypervisor mode supports only the bottom range on ARMv8.0. Thus, even if the hypervisor uses the kernel page table, it cannot map the privilege-escalated device driver that is mapped with the top range of virtual addresses in Linux.

Therefore, we use 32-bit virtual address for the kernel and hypervisor, which can present the entire virtual address space with a single page table. The only requirement is the ability to compile the kernel with the LPAE option, as illustrated in Section VII-B. Unfortunately, the fallback to 32-bit disables the watchpoint-based DMA protection from hypervisor-privileged attacks. Thus, in this case, the MyTEE security depends on whether privileged blocks contain vulnerabilities and thus requires them to be thoroughly verified. However, this restriction can be lifted from ARMv8.1. The virtualization host extensions (VHE) support using two subranges of virtual addresses in hypervisor mode as well. We will explore adopting this feature in MyTEE in our future work.

XII. Conclusion

In this paper, we presented MyTEE to build and strengthen the TEE on embedded devices without depending on the TrustZone hardware extensions that have been assumed to be generally available in previous work. Toward this end, the kernel deprivileging and careful memory management, DMA packet filter, and temporal privilege escalation techniques were proposed. In addition, MyTEE and its example use cases for secure IO were implemented on the Raspberry Pi 3 development board, which lacks mandatory TrustZone extensions for building the TEE. MyTEE is available at: https://github.com/sssecret2019/mytee.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments. This work was supported by Institute for Information & communications Technology Promotion (IITP) Grant funded by the Korea Government (MSIT) (No. IITP-2022-0-01200 and IITP-2020-0-01840), National Research Foundation (NRF) Grant funded by Korea Government (MSIT) (No. NRF-2020R1F1A1058305), and research fund of Chungnam National University.

References

[1] (2014) Usb 2.0 otg controller. "https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/a10_54018.pdf".

[2] (2018, May) Arm architecture reference manual armv8, for armv8-a architecture profile. "https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile".

[3] (2018, May) Enabling intel virtualization technology features and benefits. "https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/virtualization-enabling-intel-virtualization-technology-features-and-benefits-paper.pdf".

[4] (2019) Raspbian with op-tee support. "https://github.com/benhaz1024/raspbian-tee".

[5] (2020) Ls2088ardb: Fixed lpm-20 which got broken after an earlier commit. "https://gitlab.conclusive.pl/devices/whle-ls1/atf/-/tree/bf83b558cd535ae0272b5e5985d1c7d269b5b020/plat/nxp".

[6] (2021, May) About op-tee. "https://optee.readthedocs.io/en/latest/general/about.html".

[7] (2021) Broadcom bcm2837 specs. "https://gadgetversus.com/processor/broadcom-bcm2837-specs/".

[8] (2021) Input event codes. "https://github.com/torvalds/linux/blob/master/include/uapi/linux/input-event-codes.h".

[9] (2021) Knox licenses. "https://docs.samsungknox.com/dev/common/knox-licenses.htm".

[10] (2021) Mailbox property interface. "https://github.com/raspberrypi/firmware/wiki/Mailbox-property-interface".

[11] (2021) Optiga™ tpm application note. "https://www.infineon.com/dgdl/Infineon-App-Note-SLx9670-TPM2.0_Embedded_RPi_DI_SLx-ApplicationNotes-v01_03-EN.pdf?fileId=5546d46267c74c9a01684b96e69f5d7b".

[12] (2021) Raspberry pi 3 op-tee. "https://optee.readthedocs.io/en/latest/building/devices/rpi3.html".

[13] (2021, May) Samsung open source. "https://opensource.samsung.com/uploadList?menuItem=mobile&classification1=mobile_phone".

[14] (2021) Secure boot solution for raspberry pi. "https://www.swissbit.com/en/products/security-products/secure-boot-solution/".

[15] (2021) Stage 2 translation. "https://developer.arm.com/documentation/102142/0100/Stage-2-translation".

[16] (2021) Top 10 commercial products based on raspberry pi compute module. "https://pallavaggarwal.in/10-products-using-raspberry-pi-compute-module/".

[17] (2021) xset(1) - linux man page. "https://linux.die.net/man/1/xset".

[18] (2022) Arm trusted firmware. "https://github.com/ARM-software/arm-trusted-firmware".

[19] (2022) Exynos 7570. "https://semiconductor.samsung.com/processor/mobile-processor/exynos-7-quad-7570/".

[20] (2022) Mt6739. "https://www.mediatek.com/products/smartphones-2/mt6739".

[21] (2022) plat: nxp: Add ota support with policy ota option. "https://gitlab.conclusive.pl/devices/whle-ls1/atf/-/blob/master/plat/nxp/soc-ls1012/soc.c".

[22] (2022) Smartphones with mediatek mt6739 processor. "https://www.kimovil.com/en/list-smartphones-by-processor/mediatek-mt6739".

[23] (2022) Smartphones with samsung exynos 7 octa 7885 processor. "https://www.kimovil.com/en/list-smartphones-by-processor/samsung-exynos-7-octa-7885".

[24] (2022) Smartphones with spreadtrum unisoc sc9863a processor. "https://www.kimovil.com/en/list-smartphones-by-processor/spreadtrum-unisoc-sc9863a".

[25] (2022) Tablets with spreadtrum unisoc sc9863a processor. "https://www.kimovil.com/en/list-tablets-by-processor/spreadtrum-unisoc-sc9863a".

[26] (2022) Unisoc sc9863a review – how is this cheap soc? "https://xiaomiui.net/unisoc-sc9863a-review-19068/".

[27] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "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, 2014, pp. 90–102.

[28] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Sanctuary: Arming trustzone with user-space enclaves." in *NDSS*, 2019.

[29] F. Brasser, D. Kim, C. Liebchen, V. Ganapathy, L. Iftode, and A.-R. Sadeghi, "Regulating arm trustzone devices in restricted spaces," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, 2016, pp. 413–425.

[30] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek, "Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[31] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman, "Protecting data on smartphones and tablets from memory attacks," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. ACM, 2015, pp. 177–189.

[32] X. Ge, H. Vijayakumar, and T. Jaeger, "Sprobes: Enforcing kernel code integrity on the trustzone architecture," *Proceedings of the Third Workshop on Mobile Security Technologies (MoST)*, 2014.

[33] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "Trustshadow: Secure execution of unmodified applications with arm trustzone," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2017, pp. 488–501.

[34] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vtz: Virtualizing arm trustzone," in *In Proc. of the 26th USENIX Security Symposium*, 2017.

[35] J. Jang and B. Byunghoon Kang, "Retrofitting the partially privileged mode for tee communication channel protection," *IEEE Transactions on Dependable and Secure Computing*, 05 2018.

[36] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. B. Kang, "Privatezone: Providing a private execution environment using arm trustzone," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 5, pp. 797–810, 2016.

[37] J. Jang and B. B. Kang, "Selmon: reinforcing mobile device security with self-protected trust anchor," in *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, 2020, pp. 135–147.

[38] J. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, "Secret: Secure channel between rich execution environment and trusted execution environment," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15), San Diego, CA*, 2015.

[39] D. Kwon, J. Seo, Y. Cho, B. Lee, and Y. Paek, "Pros: Light-weight privatized se cure oses in arm trustzone," *IEEE Transactions on Mobile Computing*, vol. 19, no. 6, pp. 1434–1447, 2019.

[40] W. Li, H. Li, H. Chen, and Y. Xia, "Adattester: Secure online mobile advertisement attestation using trustzone," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2015, Florence, Italy, May 19-22, 2015*, G. Borriello, G. Pau, M. Gruteser, and J. I. Hong, Eds. ACM, 2015, pp. 75–88. [Online]. Available: https://doi.org/10.1145/2742647.2742676

[41] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li, "Building trusted path on untrusted device drivers for mobile devices," in *Proceedings of 5th Asia-Pacific Workshop on Systems*. ACM, 2014, p. 8.

[42] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD:

USENIX Association, Aug. 2018, pp. 973–990. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[43] H. Liu, S. Saroiu, A. Wolman, and H. Raj, "Software abstractions for trusted sensors," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 2012, pp. 365–378.

[44] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna, "Boomerang: Exploiting the semantic gap in trusted execution environments," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS'17), San Diego, CA*, 2017.

[45] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 143–158.

[46] S. Mirzamohammadi, Y. M. Liu, T. A. Huang, A. A. Sani, S. Agarwal, and S. E. S. Kim, "Tabellion: Secure legal contracts on mobile devices," in *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 220–233. [Online]. Available: https://doi.org/10.1145/3386901.3389027

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

[48] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 195–209.

[49] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon *et al.*, "ftpm: A software-only implementation of a {TPM} chip," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 841–856.

[50] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using arm trustzone to build a trusted language runtime for mobile applications," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ACM, 2014, pp. 67–80.

[51] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato, "Bitvisor: A thin hypervisor for enforcing i/o device security," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 121–130. [Online]. Available: https://doi.org/10.1145/1508293.1508311

[52] H. Sun, K. Sun, Y. Wang, and J. Jing, "Trustotp: Transforming smartphones into secure one-time password tokens," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 976–988.

[53] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia, "Trustdump: Reliable memory acquisition on smartphones," in *Computer Security-ESORICS 2014*. Springer, 2014, pp. 202–218.

[54] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "Trustice: Hardware-assisted isolated computing environments on mobile devices," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2015, pp. 367–378.

[55] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, "Design, implementation and verification of an extensible and modular hypervisor framework," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 430–444.

[56] J. Winter, P. Wiegele, M. Pirker, and R. Tögl, "A flexible software development and emulation framework for arm trustzone," in *Trusted Systems*. Springer, 2012, pp. 1–15.

[57] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du, "Truz-droid: Integrating trustzone with mobile operating system," in *Proceedings of the 16th annual international conference on mobile systems, applications, and services*, 2018, pp. 14–27.

[58] M. Yu, V. D. Gligor, and Z. Zhou, "Trusted display on untrusted commodity platforms," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 989–1003. [Online]. Available: https://doi.org/10.1145/2810103.2813719

[59] M. H. Yun and L. Zhong, "Ginseng: Keeping secrets in registers when you distrust the operating system." in *NDSS*, 2019.

[60] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "Case: Cache-assisted secure execution on arm processors," in *Security and Privacy, 2016. SP 2016. IEEE Symposium on*. IEEE, 2016.

[61] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng, "Sectee: A software-based approach to secure enclave architecture using tee," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1723–1740.

[62] Z. Zhou, M. Yu, and V. D. Gligor, "Dancing with giants: Wimpy kernels for on-demand isolated i/o," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 308–323.