LDR: Secure and Efficient Linux Driver Runtime for Embedded TEE Systems

Huaiyu Yan[†], Zhen Ling[†*], Haobo Li[†], Lan Luo[‡], Xinhui Shao[†], Kai Dong[†]
Ping Jiang[†], Ming Yang[†], Junzhou Luo[†], Xinwen Fu[§]
[†]Southeast University, Email: {huaiyu_yan, zhenling, haobo_li, xinhuishao, dk, jiangping, yangming2002, jluo}@seu.edu.cn
[‡]Anhui University of Technology, Email:lanluo448@gmail.com
[§]University of Massachusetts Lowell, Email: xinwen_fu@uml.edu

***Abstract*—Trusted execution environments (TEEs), like TrustZone, are pervasively employed to protect security sensitive programs and data from various attacks. We target compact TEE operating systems like OP-TEE, which implement minimum TEE internal core APIs. Such a TEE OS often has poor device driver support and we want to alleviate such issue by reusing existing Linux drivers inside TEE OSes. An intuitive approach is to port all its dependency functions into the TEE OS so that the driver can directly execute inside the TEE. But this approach significantly enlarges the trusted computing base (TCB), making the TEE OS no longer compact. In this paper, we propose a TEE driver execution environment—Linux driver runtime (LDR). A Linux driver needs two types of functions, library functions and Linux kernel subsystem functions that a compact TEE OS does not have. The LDR reuses the existing TEE OS library functions whenever possible and redirects the kernel subsystem function calls to the Linux kernel in the normal world. LDR is realized as a sandbox environment, which confines the Linux driver inside the TEE through the ARM domain access control features to address associated security issues. The sandbox mediates the driver's TEE functions calls, sanitizing arguments and return values as well as enforcing forward control flow integrity. We implement and deploy an LDR prototype on an NXP IMX6Q SABRE-SD evaluation board, adapt 6 existing Linux drivers into LDR, and evaluate their performance. The experimental results show that the LDR drivers can achieve comparable performance with their Linux counterparts with negligible overheads. We are the first to reuse functions in both the TEE OS and normal world Linux kernel to run a TEE device driver and address related security issues.**

I. Introduction

Trusted execution environments (TEEs) are extensively utilized to safeguard the integrity and confidentiality of security-sensitive program logic (SPL). Contemporary TEEs employ hardware-based methods, delivering robust security foundations for creating an isolated domain (such as SGX enclaves [1], SEV secure virtual machines [2], TrustZone secure worlds [3], and RISC-V enclaves [4]), which functions as a black box to the external environment. SPLs are typically deployed within these isolated domains to protect against external threats, particularly privileged attacks from root users, operating systems, hypervisors, etc. As a result, TEEs are widely adopted by cloud service providers and embedded system manufacturers to enable secure data processing.

However, due to the absence of sophisticated driver support inside TEEs compared to the comprehensive driver support found in commodity OS kernels, such as the Linux kernel, it is difficult to implement I/O-oriented TEE services. Instead of employing a monolithic design like KNOX [5], embedded TEE OSes, like OP-TEE [6] and SierraTEE [7], take a compact kernel design in order to reduce attack surfaces, and we refer to these TEE OSes as compact TEE OSes. Compact TEE OSes typically maintain a minimal trusted computing base (TCB) with a small line of code (LoC) [8], [9], [10], providing only security-critical services like secure boot, cryptographic operations, and attestation. When it comes to TEE exclusive I/O interactions, many research works choose to port existing device drivers into TEEs [11], [12], which involves a huge amount of repeated engineering efforts. To reduce the engineering efforts introduced by trivial driver porting, Driverlets [13] proposes a general driver-reuse method where a recorder executes a driver inside an emulator like QEMU [14] and records its runtime I/O and DMA behaviors. Then, a replayer replays the recorded behaviors inside TEE to reproduce driver functionalities. However, the request of a sophisticated emulator is hard to satisfy, as only a limited set of hardware board models is currently supported by QEMU [15]. Moreover, such approach of replying recorded driver behaviors exhibits poor performance. Other systems like MyTEE [16] enable secure I/O by reusing existing NW drivers and providing security guarantees through virtualization-based memory isolation as well as temporary supervised privilege escalation. However, since MyTEE I/O operations involve world switches and hypercalls, it incurs heavy overhead for devices with frequent I/O requests. Therefore, we are motivated to explore an approach that reuses the existing device drivers to facilitate secure and efficient driver support inside compact TEE OSes while maintaining a small TCB.

In this paper, we propose a TEE driver execution environment, named Linux driver runtime (LDR), that enables secure and efficient TEE driver support. We focus our design on ARM TrustZone-empowered embedded systems and choose to reuse existing Linux drivers. LDR is a sandbox environment that provides necessary runtime support for driver execution inside the secure OS which works as the operating system inside the TrustZone secure world (SW). In brief, we load a

*Corresponding author: Prof. Zhen Ling of Southeast University, China.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.23009
www.ndss-symposium.org

driver module for a target secure peripheral, which we refer to as a SW driver, into LDR and orchestrate and mediate its interactions with the secure OS as well as the normal world (NW) Linux kernel so that the driver module can properly initialize and operate on the target peripheral. To mitigate the security concerns introduced by the SW driver's dependency on the NW Linux kernel, LDR provides library function (e.g., memory management, I/O operation, etc.) support for the SW driver by reusing the secure OS library functions. As for the remaining ones which the secure OS does not implement, LDR redirects them to the NW Linux kernel yet with security checks on both passed arguments and return values.

To maintain a small TCB and to protect the secure OS from potentially untrusted drivers, LDR isolates SW drivers from the secure OS as well as from each other. To be specific, LDR leverages ARM domain access control (DAC) [3] features to create an isolated execution domain (IED) for each SW driver so that an untrusted driver cannot directly access the secure OS memory or other SW drivers' data during execution. To facilitate secure interactions between SW drivers and the secure OS, LDR creates an IED gate that acts as the secure gateway of the secure OS. Particularly, during SW driver execution, the IED gate intercepts and mediates every secure OS function call issued by a SW driver. Additionally, LDR enforces forward control flow integrity (F-CFI) on these function calls to defend against control-flow hijacking attacks on the secure OS. Finally, we investigate the secure driver state maintenance schemes to prevent sensitive SW driver data from being leaked to the NW during redirected Linux subsystem calls.

LDR design choices have several advantages over existing approaches in terms of 1) *engineering efforts* (§VI-C): Beside driver code reuse, LDR reuses existing secure OS functions and redirects unsupported ones to the NW Linux kernel instead of implementing the driver runtime from the scratch. 2) *performance* (§VI-E, §VI-F): LDR runs compiled driver modules in native speed and therefore outperforms Driverlets that leverages emulated execution. 3) *security* (§VII): LDR enables secure I/O with all device I/O operations exclusively conducted inside the SW. LDR isolates all SW drivers from the secure OS as well as from each other to maintain a small TCB. Meanwhile, LDR mediates each function call issued from SW drivers and enforces security checks on those redirected to the NW Linux kernel.

We develop an LDR prototype and deploy it on a TrustZone-empowered NXP IMX6Q SABRE-SD board in order to demonstrate the feasibility and effectiveness of LDR. To estimate engineering efforts, we generate the SW drivers for diverse sensors, including accelerometer, image processing unit, etc. The results show that the modification to the original Linux driver code is reasonable and most driver code stays untouched. Additionally, we evaluate the LDR performance using these SW drivers via various use cases. The experimental results demonstrate that the LDR SW drivers can exert the full power of the devices for these real-world use cases, with sensor data collection in full sampling speed and only -2.43% streaming speed penalty to high-definition video streaming.

In summary, this paper offers the following contributions:

- We present several key observations on the internal structures of modern Linux drivers. Such observations can minimize the dependency functions of a SW driver, thus simplifying the overall LDR design and reducing engineering efforts.
- We are the first to offer driver runtime environment support by reusing existing TEE functions whenever possible and redirecting the unsupported ones to the NW OS kernel.
- We propose a TEE driver isolation mechanism based on efficient ARM DAC features, which separates untrusted drivers from the secure OS as well as from each other while securely mediating their interactions with the secure OS and the NW Linux kernel through CFI.
- We implement an LDR prototype and evaluate its feasibility and effectiveness in supporting various sensor drivers. The experimental results demonstrate that LDR introduces minimal performance overhead to real-world use cases.

II. Motivation & Observation

In this section, we first present our motivations and investigate two alternative design choices, briefly discussing their merits and limitations. Then, we present our observation and insights on the Linux driver model which can greatly simplify the overall LDR design.

A. Motivations

TrustZone-based secure OSes have limited support for device driver inside the SW, which hinders the deployment of I/O-oriented secure services on TrustZone-enabled platforms. In contrast, the Linux kernel provides sophisticated driver support to almost all devices whose drivers are well maintained by a lot of communities. Therefore, we are motivated to propose an approach that shall realize 1) **Code reuse.** The existing driver code should be reused. We intend to reuse low-layered driver functions related to direct I/O, interrupt handling and utility functions since these functions interact with the peripheral and realize basic driver functionalities. 2) **Security.** To preserve a small TCB size, the modifications to the secure OS should be as small as possible. Additionally, since a SW driver is large in size and may contain potential security flaws, all SW drivers should be isolated from the secure OS rather than being included as part of the TCB. 3) **Good performance.** The SW drivers shall achieve comparative performance with their Linux counterparts.

B. Alternative Designs

Porting Linux Kernel Functions into Secure OS. One straightforward approach is to include driver loadable kernel modules (LKM) and the Linux kernel inside the secure OS, as shown in Figure 1-(a), so that all external functions depended by the driver LKM can be provided by the secure OS. To minimize code base introduced to the secure OS, the original Linux kernel can be trimmed down only to include functions needed by the drivers. During runtime, drivers can directly invoke functions of the trimmed kernel as they do when running inside the Linux kernel. This approach can result in good performance as both the drivers and the trimmed kernel reside in the same address space of the secure OS. Additionally, since the trimmed kernel provides all the functions needed

by the drivers, no driver modification is needed. However, there are several drawbacks involving such approach. Firstly, it is a non-trivial job to trim down the Linux kernel. As shown in §II-C, a driver depends on functions provided by various Linux kernel subsystems. These functions are tightly aggregated with the Linux kernel as well as with each other and it takes huge engineering efforts to extract them. Secondly, though trimmed down, the remaining kernel code base is still very large compared to the secure OS and adding such trimmed kernel to the secure OS can result in tremendous TCB bloating.

Redirecting All Function Calls to Linux Kernel. Another alternative approach is to only put driver LKMs inside the secure OS and redirect all driver dependency function invocations back to the NW Linux kernel, as shown in Figure 1-(b). For each SW driver's dependency function, a SW caller stub and a NW callee stub are generated to pass arguments and synchronize global data objects between the SW and the NW accessed by both the caller and the callee [17]. Meanwhile, a remote procedure call (RPC) proxy checks and validates each redirected call in terms of passed arguments and return values. Such total function call redirection approach does not enlarge TCB since the complex Linux kernel subsystems still reside in the NW and can be realized with relatively small engineering efforts. However, such approach may result in performance downgrade as well as security issues. Firstly, redirecting every single function invocation may result in huge performance overhead for drivers with frequent kernel function calls, e.g., sensor devices with bus I/O at a high sampling rate. To justify such claim, we deploy an MMA8451 accelerometer driver[18] inside the secure OS following this approach. The results show that the sampling rate drops to 405.89, 49.3% lower than the device's default sampling rate of 800. Secondly, such approach may result in security issues. Recall that the kernel I/O library functions primarily help drivers to interact with the underlying devices. Therefore, the untrusted Linux kernel may corrupt the raw data by hooking these I/O functions and the SW cannot acquire integral and genuine raw data.

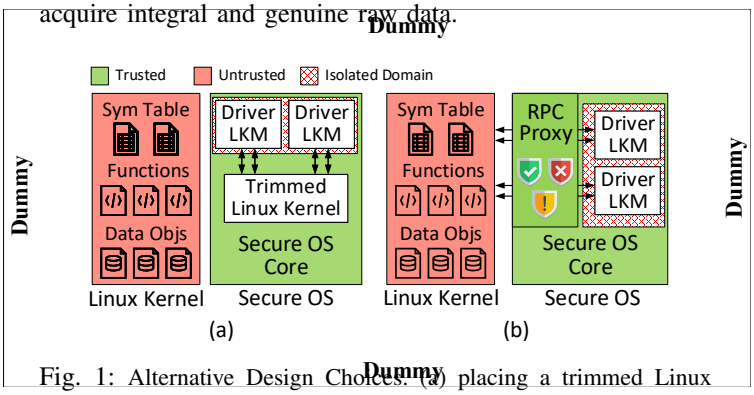


Fig. 1: Alternative Design Choices. (a) placing a trimmed Linux kernel inside the secure OS. (b) redirecting all function calls back to the Linux kernel.

Reflection on Alternative Designs. The fundamental factor resulting in the drawbacks of the above two alternative designs is that a driver LKM depends on so many complicated kernel functions that the runtime environment has to support either by porting or through function call redirection. Therefore, we ask two inspiring questions:

Q1 *Are all driver functions necessarily needed to provide I/O support inside TEE?*

Q2 *Do all these dependency functions need to be either directly provided inside the secure OS or completely redirected to the NW Linux kernel?*

C. Observations

We try to answer **Q1** by figuring out the abstract structure of a Linux driver. In turn, we construct a Linux driver model by analyzing the source code of different Linux drivers [19], [20], [18], [21], focusing on their common internal structures and their interactions with other Linux kernel components. In general, a driver is composed of its state variables and functions, as shown in Figure 2.

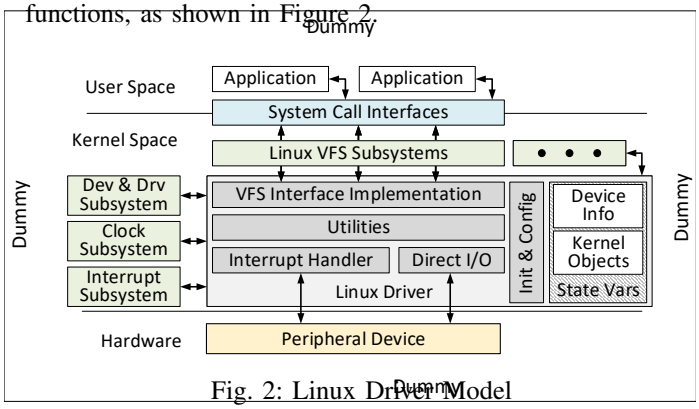


Fig. 2: Linux Driver Model

State Variables. Usually, a Linux driver is implemented with several global data structures holding information vital to the driver. We refer such global data structures as the driver's state variables which contain the driver's device information (e.g., register map and hardware state) and subsystem data objects used when Linux kernel functions are invoked.

Driver Functions. As shown in Figure 2, we classify all driver functions into five categories in terms of their functionalities: 1) **Initialization & configuration.** These functions (i.e. `init()`, `probe()`) are responsible for initializing the driver state variables. The Init & Conf functions set up the device information fields and call kernel subsystem functions with the subsystem data objects as arguments to register the driver to various Linux kernel subsystems. 2) **Direct I/O.** These functions interact directly with the underlying hardware device through bus I/O operations or memory mapped I/O (MMIO). 3) **Interrupt handler.** These functions handle the interrupts issued by the device and they are often registered to the Linux interrupt subsystem during the driver initialization. 4) **VFS interface implementation.** These functions implement VFS-defined interfaces and are registered to the VFS subsystems in form of function pointers [22]. 5) **Utilities.** The rest functions are referred to as utility ones which conduct operations, e.g., processing data read from devices, enabling/disabling a device functionality using I/O functions.

Based on such abstract driver model, we observe that Init & Conf functions are meant to initialize the state variable and never get executed during runtime. Such observation motivates our first observation shown as follows.

Obs 1. There is no need to re-run Init & Conf functions as long as we make a secure and valid copy of the SW driver's state variable from an initialized NW driver.

Moreover, the original driver provides its services to user-space applications through its VFS functions. Though sophisticated, Linux file systems are too heavy for TEEs [23] and the secure OS has its own lightweight mechanism to provide secure OS services to SW applications. Therefore, such observation results in our second observation.

Obs 2. A driver's VFS functions are service-oriented and do not directly interact with the peripherals. Instead, they depend on functions related to I/O, interrupt handling and utilities, which conduct direct interaction with the underlying device.

According to **Obs 1&2**, a driver's Init & Conf and VFS functions can be discarded and LDR no longer needs to support the corresponding dependency functions invoked by these functions (**Q1**). Furthermore, as to be shown in §VI-C, such function discarding will significantly reduce the number of driver dependency functions, especially kernel subsystem functions, and in turn simplify the overall LDR design.

To answer **Q2**, we analyze a driver's dependency functions and categorize them into two classes: 1) **Library functions** which provide various utility primitives (e.g. memory management, primary data processing, I/O operations, etc.) which are common across various OSes. 2) **Kernel subsystem functions** which provide advanced driver management services which are specific to the Linux kernel. The remaining dependency functions are vital to driver's well functioning and it is difficult to port them to the secure OS. Accordingly, we answer **Q2** by presenting our final observation.

Obs 3. Library functions can be directly supported by the secure OS and the Linux kernel subsystem functions need to be redirected to and handled by the NW Linux kernel.

III. Assumptions & Threat Model

We assume that the secure OS is trusted. The NW programs including the Linux kernel and NW drivers are assumed trusted during the system boot phase so that NW drivers can be securely initialized and a legal NW state variable can be generated and used to initialize the corresponding SW driver. After the SW driver is initialized, the whole NW may be compromised at run time and thus become malicious.

It is also assumed that the SW drivers may contain vulnerabilities which an attacker can exploit to launch memory corruption or control flow hijacking attacks, e.g., return-oriented programming (ROP) attacks [24]. We are intended to protect the secure OS from such attacks and maintain a small TCB. Moreover, we aim to ensure code integrity and data privacy for each SW driver and prevent an untrusted driver from corrupting memory regions belonging to other SW drivers. Note that LDR cannot ensure device availability since the malicious NW OS can just shutdown or crash the system, resulting in denial-of-service. In addition, we assume that the underlying hardware's implementation adheres to their specification. Finally, we consider advanced hardware-oriented attacks, e.g., cache-based side channel attacks [25], [26], [27], bus snooping attacks [28], DMA attacks [29] and cold boot attacks [30], to be out of scope.

We assume that the underlying devices can be configured as secure using the TrustZone Protection Controller (TZPC) [31], making them exclusively accessible from the SW. Therefore, data can only be captured inside the SW so as to assure the integrity and validity of the raw data. However, we do not enforce any particular security policy on whether the captured data can only be accessed from the SW or be shared among both NW and SW programs. Developers can make the final decision depending on the particular application scenario. If developers allow NW applications to access the data, the data can be further protected by being encrypted or signed inside the SW and then passed to the NW applications [32].

IV. LDR Design

In this section, we elaborate on the design of our Linux driver runtime for TrustZone-based secure OSes. After presenting an overview of LDR design, we first introduce the LDR driver generation approach to generate the twin drivers from the original Linux driver. Then we justify the correctness of substituting Linux kernel library functions with secure OS ones and demonstrate the basic workflow of the SW driver execution inside LDR. Finally, to protect secure OS from untrusted SW drivers, we present a driver isolation mechanism based on ARM domain access control (DAC) and secure state variable maintenance schemes.

A. LDR Overview

We design the Linux Driver Runtime (LDR), a runtime execution environment for SW driver loadable kernel modules (LKMs). Two phases are involved to run a driver inside LDR, namely the offline phase and the runtime phase, as shown in Figure 3.

Offline Phase: During offline phase, we generate SW drivers and prepare the corresponding driver dependency functions inside LDR. We first investigate a twin driver approach where a pair of twin drivers, namely the NW driver and the SW driver, are generated based on the original Linux driver (§IV-B). Particularly, the NW driver is meant to assist with the SW driver's runtime execution. Additionally, the SW driver is generated by removing Init & Conf and VFS functions of the original driver (**Obs 1&2**). Moreover, based on the dependency functions of the SW driver, LDR provides the corresponding library functions by reusing existing secure OS functions (§IV-C) and generates RPC stubs that redirect subsystem function calls (**Obs 3**).

Runtime Phase: During runtime, the twin drivers work cooperatively to facilitate SW driver support (§IV-D). The SW driver is loaded into LDR, interacts with the secure peripheral devices and provides secure I/O services to other SW components. The corresponding NW driver is a helper driver running inside the NW Linux kernel to assist with SW driver initialization and subsystem call redirection.

Considering that SW drivers themselves have large code base and may contain potential security flaws, we create an isolated execution environment to protect the secure OS from the untrusted SW drivers and prevent an untrusted SW driver from jeopardizing other drivers' code integrity and data privacy. Firstly, we create an isolated execution domain for a set of functionally-related SW drivers and prevent such SW drivers from accessing memory regions belonging to the secure OS and other IEDs (§IV-E1). Secondly, we provide

a carefully-defined IED call gate that intercepts and mediates each dependency function call issued by a SW driver (§IV-E2). Finally, we propose several driver state maintenance schemes that securely maintain both SW and NW driver states so that no sensitive SW driver information is leaked to the Linux kernel during driver initialization and execution (§IV-F).

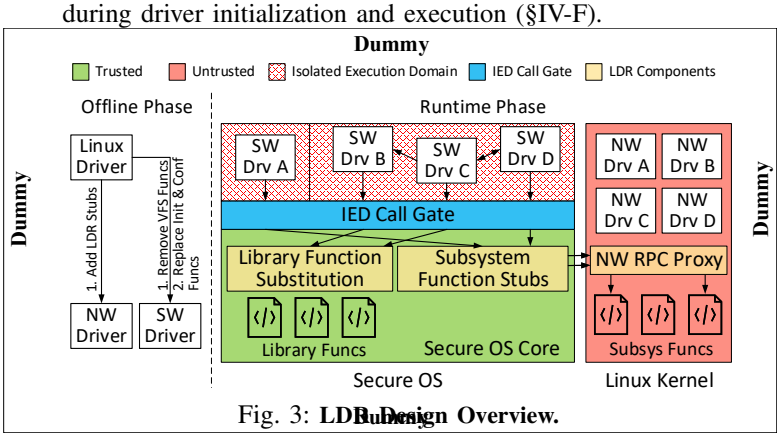


Fig. 3: **LDR Design Overview.**

B. Twin Driver Generation

We propose an LDR driver generation (LDG) approach to generate a pair of twin drivers, namely the SW driver and the NW driver. The SW driver is to interact with the corresponding device and export its services to the secure OS components so that the secure OS can provide the driver services to trusted applications (TA). The NW driver is a NW helper driver used to synchronize the SW state variable and provide data retrieval interfaces to NW applications if needed.

1) SW Driver: The SW driver is generated by recognizing and removing unnecessary functions from the original driver and the basic idea is to remove Init & Conf functions and VFS functions from the driver.

Recall that the Init & Conf functions initialize the driver's state variable by invoking multiple Linux kernel subsystem functions (**Obs 1**). During runtime, the corresponding NW driver is first initialized and its state variable is copied and used as the SW driver's state variable. However, several fields of the NW driver's state variable are invalid inside the SW due to the semantic gap between the NW and SW. Take device register mapping as an example, the returned register mapping address is a Linux kernel virtual address which is invalid inside the SW. Therefore, a new initialization function (`sw_init()`) is added to invoke the driver registration interfaces provided by our light-weighted LDR driver manager to register SW drivers to LDR. Then, another initialization function (`sw_probe()`) is added to initialize the device information fields of the SW driver state variable by invoking the secure OS functions. For example, we invoke the secure OS memory mapping functions to acquire the correct register mapping addresses. The rest fields of the SW state variable (i.e. the subsystem data objects) remain intact with their NW counterparts.

As for VFS functions, we can remove them without jeopardizing the driver's capability to provide I/O functionalities (**Obs 2**) and we present an automatic way to remove these functions in §V-A.

2) NW Driver: The NW driver is also generated from the original Linux driver with minor modifications to initialize the state variables for the SW driver and provide data retrieval interfaces to NW applications if needed. We add extra operations to the original Init & Conf functions, namely `init()` and `probe()`, to pass the initialized subsystem data object fields of the NW state variable to the SW driver. To allow developers to retrieve data from the SW driver if needed, we design a cross-world procedure call (CPC) mechanism to generate a pair of NW and SW CPC stubs. The NW stub is responsible for marshalling the arguments while the SW stub unmarshalls the arguments and invokes the corresponding SW driver data retrieval function. Except the Init & Conf functions and the optional NW stubs, the rest NW driver functions are never executed during runtime.

C. Library Function Substitution

LDR provides library function support for SW drivers by reusing the existing secure OS library functions. To be specific, we wrap the existing secure OS library functions to adapt their function signatures to those declared by the Linux kernel (**Obs 3**). These secure OS library function wrappers are used during SW driver linking phase and eventually invoked by the SW driver during driver execution phase.

We claim that such substitution can achieve functional correctness. Firstly, since a secure OS library function realizes the same functionalities as its Linux kernel counterpart, reusing the existing secure OS library functions can satisfy the SW driver's requirements. Secondly, due to the intrinsic simplicity of the secure OS, a structure defined by the secure OS is not so sophisticated as its Linux kernel counterpart and therefore takes less memory space. For example, the `struct mutex` of the Linux kernel [33] takes 20 bytes while that of OPTEE [34] takes only 12 bytes. Recall the SW driver is built using the Linux `KBuild` system. Therefore, the memory space reserved for each SW driver data object, either global or local, is determined by the corresponding Linux structure definition and is more than what the secure OS function needs. Since the SW driver data objects are only used by secure OS library functions and the reserved memory space is sufficient for function execution, it is safe to use pointers to these SW driver data objects as arguments of the secure OS library functions.

D. Workflow of SW Driver Execution

LDR consists of the *SW driver loader*, the *symbol manger*, the *driver manger* and the *session manager*. Figure 4 illustrates the workflow of LDR, showing how the SW driver is loaded into LDR and how its services are provided to other SW and NW components. During the loading phase, the SW driver's code integrity is enforced through digital signature and the validity of its initial state is ensured through trusted boot [35]. To begin with, a SW driver is signed offline and stored inside the non-volatile storage media such as a flash disk. A NW LDR client application (CA) passes the SW driver ELF image to the *SW driver loader* through shared memory. After verifying the SW driver's signature, the *SW driver loader* sets proper memory access attributes for its sections, resolves its undefined symbols using the *symbol manger* and relocates the dependency function callsites to the resolved function addresses (Fig. 4, ❶). After the SW driver is properly loaded and linked, the SW driver's `sw_init()` is invoked to register the driver information, such as the SW state variable address

and shared memory, to the LDR driver manager. Then, the NW driver is loaded to the NW Linux kernel. Once the NW driver is fully initialized, the SW driver can be notified to invoke `sw_probe()`, thereby copying the whole NW state variable to the SW state variable which is used as the SW driver's initial state. The subsystem data object fields are kept intact while `sw_probe()` initializes the rest fields, so that these fields contain valid device information and can be used by the secure OS library functions (Fig. 4, ❷). Once the SW driver is properly initialized, LDR leverages TZPC to configure the corresponding peripheral as secure device. Its SW I/O functions are exclusively responsible for all I/O interactions with the device, and interrupts issued by the device are handled by its SW interrupt handlers. Additionally, to prevent SW driver's interrupt handler from being interrupted by the NW Linux kernel, LDR dedicates one ARM core to handling secure interrupts and excludes it from NW usage (Fig. 4, ❸). Other secure OS components can access the SW driver services by directly calling its exported I/O and utility functions (Fig. 4, ❹). Moreover, if developers allow the data captured by the underlying device to be used by NW applications, the CPC stubs are leveraged to retrieve the captured data and the *session manager* coordinates the communication between the NW and SW driver (Fig. 4, ❺). Finally, when the SW driver invokes a kernel subsystem function, the invocation is redirected to the NW LDR Linux kernel proxy and the SW state variable is synchronized with the NW state variable (Fig. 4, ❻). We elaborate on the SW state variable maintenance in §IV-F.

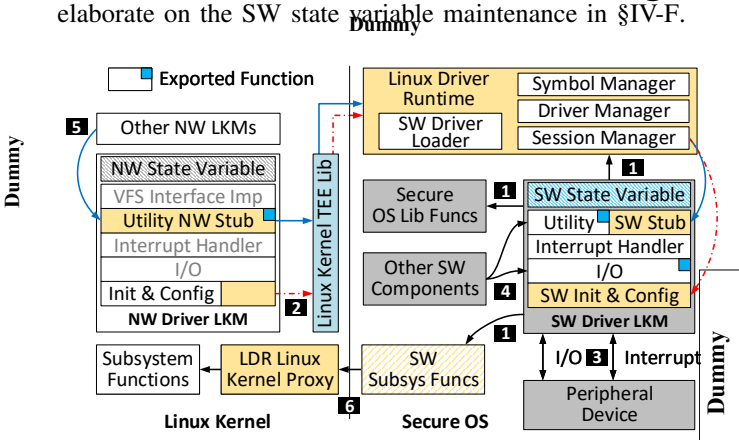


Fig. 4: **Workflow of SW Driver Execution.** White and grey to indicate NW and SW components respectively. Orange to indicate LDR newly-added parts. Blue to highlight state variable. Note that we exclude the security mechanisms to be discussed in §IV-E and §IV-F to illustrate a clear workflow.

E. Driver Isolation

Since a third-party driver may be vulnerable, LDR confines each SW driver inside an isolated execution environment in order to prevent a SW driver from corrupting the secure OS memory as well as other drivers' memory regions. Additionally, to defend against control flow hijacking attacks [24], LDR intercepts each secure OS function invocation issued by SW drivers, and checks its validity.

1) Isolated Execution Domain: LDR creates an isolated execution domain (IED) for each SW driver. As shown in Figure 5, a typical LDR memory layout consists of 1) the secure OS, 2) trusted applications (TA), 3) *several IEDs* that confine each SW driver, and 4) *an IED gate* that provides an entry point for SW drivers to invoke secure OS functions and for the secure OS to enter IEDs. Each IED provides the resident driver with the necessary runtime execution context: a dedicated *IED heap* for dynamic memory allocation and a dedicated *IED stack* which is separated from the secure OS stack and used by the SW driver during execution.

Based on the ARM DAC features, the memory region of each LDR component is assigned a unique domain number. With domain 0 and 1 originally assigned to the secure OS and the TAs, the IED gate is assigned with domain 2 and each IED occupies one remaining domain number. Even though ARM DAC only supports 16 domains in total, leaving domain 3-15 to IEDs, existing research works like EPK [36] and VDom [37] have proposed efficient ways to breakthrough the domain number limitations. With such support, any number of IEDs can be deployed inside LDR.

With DAC, strong memory access control policies are enforced. In general, each IED is isolated from the secure OS and from all the other IEDs while the IED gate is always accessible to all IEDs. During execution, a SW driver can only access memory regions belonging to its containing IED as well as the IED gate. When the SW driver calls a secure OS function *F*, such invocation is intervened by the IED gate which saves the current IED context and opens access to the secure OS. To prevent race conditions where two IEDs are open simultaneously, the secure OS can only serve one IED at a time and other IEDs except the calling one remain inaccessible to defend against confused deputy attacks like Boomerang [38]. During *F*'s execution, the IED heap and stack are used. Since the secure OS is trusted, such arrangement will not undermine the SW driver's code integrity and data privacy. On *F* return, the secure OS invokes the IED gate which restores the IED context and closes access to the secure OS. We do not allow inter-IED driver function invocation. Therefore, for drivers depending on each other, they can be loaded into the same IED and treated as a single isolation entity.

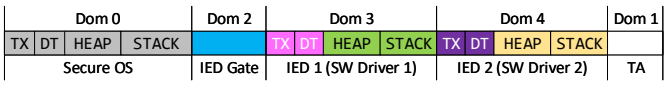


Fig. 5: **LDR SW Driver Isolation.** TX stands for code. DT stands for data.

2) Driver Dependency Function Call Validation: Another key to LDR driver isolation is that each dependency function call is mediated by the IED gate. An attacker may launch control flow hijacking attacks, like ROP attacks [24], to bypass the validation of the IED gate. Therefore, we propose a dependency function call validation scheme to enforce forward control flow integrity (CFI) by leveraging the IED gate to verify all dependency function calls issued by SW drivers.

The basic idea is to hook every dependency function call to the IED gate. During SW driver loading, the *SW driver loader* links each dependency function call to the IED gate instead of directly linking them to the corresponding dependency function call, i.e., secure OS functions or Linux subsystem functions. Meanwhile, each pair of callsite addresses and their target callee addresses, i.e., $\langle callsite, callee \rangle$, is recorded in the secure OS. After linking, `.text` segments of the SW

driver are mapped as non-writable. The SW driver cannot change such memory access attributes via the page table, since the page table locates in the secure OS (domain 0) and is inaccessible from SW drivers. Once a dependency function is invoked, the IED gate retrieves the target callee address according to the recorded $\langle callsite, callee \rangle$ mapping information and invokes the target callee if there is a valid mapping. Such dependency function call validation scheme ensures that every call to dependency functions is intercepted and verified by the IED gate and for each callsite, the SW driver can only invoke the corresponding function recorded during driver linking but none else. We will discuss the security of such dependency function call validation scheme and how it prevents control flow hijacking attacks in §VII-B1.

F. Secure State Variable Maintenance

Since contents of the NW and SW state variable are exchanged during the SW driver initialization and Linux kernel subsystem call redirection, we need to differentiate which fields are allowed to be passed to the NW, thereby protecting the SW state variable. A SW driver's state variable has to be properly initialized before can be used and its contents need to be properly synchronized with its NW counterpart before a redirected Linux kernel subsystem function is called.

1) SW-exclusive Field Labeling: Recall that a SW driver's state variable consists of fields related to device configuration information and subsystem data objects. And such fields can be used by three types of functions: 1) **Driver functions.** Fields are referenced by the SW driver's own functions either through direct reference or as function arguments; 2) **Library functions.** Fields are used as arguments of the secure OS library functions. 3) **Linux kernel subsystem functions.** Fields are used as arguments of kernel subsystem functions and they are redirected to the Linux kernel.

We refer to the subsystem data object fields as *shareable fields* while the remaining fields as *SW-exclusive fields*. Only the *shareable fields* can be passed to the NW while all other fields are only used within the SW. Since the subsystem data objects are used by the Linux kernel to manage the driver and the corresponding devices, they do not contain any sensitive information like device register mapping information. We can pass them to the NW Linux kernel subsystem functions. For other fields that are SW exclusive, LDR provides a mechanism for developers to label these fields by specifying the SW-exclusive field layout inside the SW state variable in form of a bitmap. After compilation, the generated bitmap is attached to the SW driver image as a dedicated data section. Such bitmap is retrieved during SW driver loading and used to differentiate the *shareable fields* and the *SW-exclusive fields* on the fly.

2) State Variable Initialization: For SW driver initialization, two functions are invoked, namely `sw_init()` and `sw_probe()`. To begin with, the SW driver's `sw_init()` function is invoked to register the SW driver to the LDR driver manager that records the SW driver's state variable address as well as the SW-exclusive field layout bitmap. Then, the NW driver is loaded into the NW Linux kernel to initialize the NW state variable with all its kernel data object fields properly initialized by various Linux kernel subsystems. Next, the NW driver notifies the SW driver via CPC to copy the initialized NW state variable to the SW state variable and used as the SW driver's initial state. However, at this point, the rest *SW-exclusive fields* are initialized by the Linux kernel and still adhere to the Linux kernel's semantics. Thus, the SW driver's `sw_probe()` is invoked to conduct SW-exclusive re-definition on these fields so that their contents reflect the secure OS semantics and can be used by secure OS functions.

3) State Variable Synchronization & Shadowing: LDR provides a SW RPC stub for each subsystem function to handle the state variable synchronization and the Linux kernel subsystem call redirection. Before each redirected kernel subsystem function call, SW RPC stub leverages the SW-exclusive field layout bitmap to only synchronize the *shareable fields* between the SW and NW state variable while shadowing *SW-exclusive fields* from the NW. To prevent leaking SW-exclusive information to the NW, only shareable fields of the SW state variable are copied to the corresponding fields of the NW state variable based on the SW-exclusive field layout bitmap. Then, to invoke a Linux kernel subsystem function, the copied NW *shareable fields* are used as arguments of the corresponding subsystem function and their contents can be modified by that function. Once the subsystem function returns, the SW RPC stub synchronizes the modified NW *shareable fields* back to SW state variable with all *SW-exclusive fields* intact.

V. Implementation

We implement an LDR prototype based on OP-TEE OS 3.10.0 [34]. In this section, we first show how to automatically delete unnecessary functions from the original driver by leveraging GCC deadcode elimination features. Then, we introduce how we provide dependency functions for SW drivers.

A. Automatic Function Removal

For function removal, we leverage the GCC deadcode elimination feature to automatically remove VFS and Init & Conf functions. Recall that a SW driver is generated based on the corresponding original Linux driver by removing its unnecessary functions, namely the VFS functions and the original Init & Conf functions. Since these functions are assigned to fields of Linux kernel objects like `platform_driver` and `i2c_driver` in form of function pointers, we can comment out such function pointer field assignment statements and GCC will automatically exclude the corresponding functions as well as their callees recursively from the final object file using its deadcode elimination optimization feature, which is enabled by default (`-O1`). We implement a python script to help developer pinpoint these assignment statements. Note that, since GCC can only exclude internal functions without external callers, we need to ensure that functions to be deleted are declared as static functions.

B. Dependency Function Support

We provide 74 dependency functions to support drivers shown in Table I. Recall that the *symbol manager* provides three types of dependency function support, i.e., the kernel library functions, the Linux kernel subsystem functions and the LDR driver management functions. Among them, 39 existing OPTEE OS library functions or the GCC library functions are wrapped to provide library function support. 7

functions related to clock management and time delay are redirected back to the Linux kernel leveraging the OPTEE `thread_rpc_cmd()` interface [34]. Then, we implement the remaining 28 driver management functions of the LDR driver manager from scratch. Additionally, we implement a SW symbol exportation macro named `TEE_EXPORT_SYMBOL()` to export secure OS functions. A *SW driver loader* is also implemented to load SW driver LKMs and leverage the *symbol manager* to resolve SW driver undefined symbols. The *SW driver loader* links all driver dependency functions to the hook functions (i.e. the IED gate) as discussed in §IV-E2.

VI. Evaluation

In this section, we introduce the evaluation platform setup and conduct various evaluations on the whole LDR system. First, we evaluate the engineering efforts to generate SW drivers. Next, we conduct system benchmarks and measure the latency of Linux kernel subsystem function redirection and the cross-world procedure call. Then, we conduct several case studies using various devices to show that LDR is feasible for real-world use cases. Finally, we compare LDR with other two state-of-the-art TEE-based secure I/O approaches, namely Driverlets[13] and MyTEE[16].

A. Evaluation Platform Setup

We evaluate LDR on an NXP IMX6Q SABRE-SD evaluation board, a powerful evaluation platform equipped with 4 ARM Cortex-A9 cores, 1 GB DRAM and a rich set of peripherals. We use the sensors shown in Table I to evaluate the feasibility of LDR.

TABLE I: Device List

#	Device	Product Module
1	Barometer	MPL3115A2
2	Magnetometer	MAG3110
3	Accelerometer	MMA8451
4	Ambient Light Sensor	ISL29023
5	Thermal Sensor	IMX6Q Thermal Sensor
6	Camera Sensor	OV5640
7	Image Processing Unit	IMX6Q Image Processing Unit

TABLE II: TCB Size Evaluation

TCB Component	Original LoC	Changed	Added	Removed
Secure Monitor	722	8	311	0
Secure OS	274,864	61	3,711	58
IED Gate	0	0	925	0
Total	275,586	69	4,947	58

B. TCB Size Evaluation

We first specify all LDR TCB components and then evaluate LDR TCB size increase in terms of LoC and binary size. The LDR TCB consists of the secure monitor, the secure OS and the IED gate. LDR excludes SW drivers from TCB through sandboxing, as discussed in §II-A. LoC statistics are summarized in Table II and the code counting is conducted using `cloc` [39]. In total, the LoC increases by only 1.80%. The original OPTEE OS image is 557.1KB and the LDR image excluding padding for DAC alignment is 588.6KB. The binary expansion is only 5.65%. Therefore, LDR introduces a reasonably small TCB increase.

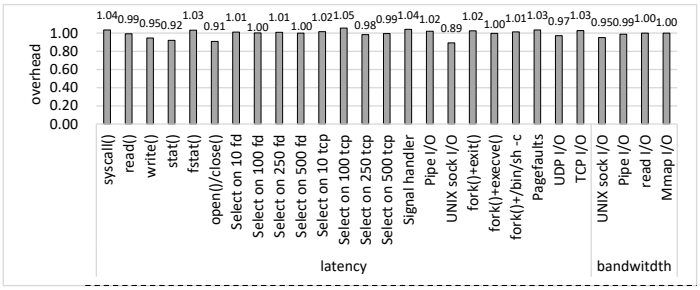


Fig. 6: LMbench Results

C. Analysis of SW Driver Generation Efforts

Generating a SW driver mainly involves deleting unnecessary functions and adding new initialization functions as well as CPC stubs if needed. Table III-(a) summarizes the modification we make to the original Linux drivers. To begin with, the VFS functions and the original Init & Conf functions are deleted automatically. The new initialization functions are responsible for register mapping, interrupt handler registration as well as driver registration. These operations are straightforward and trivial compared to the original Init & Conf functions which involves various Linux kernel subsystem calls. As for the CPC SW stubs, they conduct argument unmarshalling which is easy to be generated. In summary, most of the original driver code are untouched and the SW driver can be produced with reasonable engineering efforts. Furthermore, as shown in Table III-(b), the number of SW driver dependency functions are much less than those of the corresponding Linux driver. Therefore, the development efforts for LDR can be significantly reduced.

D. System Benchmarks

We perform two sets of system benchmarks to evaluate the performance overhead introduced by LDR to both vanilla Linux and the original OPTEE.

1) LMBench: We use vanilla Linux [40] and the original OPTEE OS [34] [1] as the baseline setting. We perform LMbench 3.0 [41] on both the baseline setting and LDR. Results show that despite the absence of one core, our LDR prototype introduces little overhead to most NW Linux services, as shown in Figure 6. This is because the LDR kernel subsystem call proxy does not influence most kernel code path.

2) OPTEE xtest: We use *xtest* [42], a comprehensive test suite shipped with OPTEE, to evaluate both the performance (benchmark tests) and the functionality correctness (regression tests) of the LDR OPTEE OS. We take the vanilla Linux kernel and the original OPTEE OS with 4 cores available to the NW as the baseline setting. We also conduct another experiment with the baseline setting yet with only 3 cores. Finally, we run *xtest* on our LDR prototype. For each experimental setting, we run *xtest* for 10 times and we compute the geometric mean of the elapsed time for each test as the result. Our LDR prototype passes all tests without failures. The experiment results in Figure 7 also show that the absence of one core introduces little performance overhead to all test cases and LDR introduces negligible performance overhead to the SW execution.

[1] Minor modifications are conducted to port the original OPTEE OS to our evaluation board.

TABLE III: SW Driver Evaluation

(a) LoC Statistics

Device	Original Driver		SW Driver LoC					SW Driver
	LoC	Size (byte)	UNT	DEL	CHA	ADD	MOD rate	Size (byte)
ipu	10,742	210,884	10,706	18	18	372	3.80%	114,896
isl29023	794	17,564	748	4	42	75	15.24%	9,720
mag3110	537	16,024	523	11	3	84	18.25%	9,996
mma8451	510	13,520	498	4	8	64	14.90%	9,296
mpl3115A2	325	10,668	312	5	8	85	30.15%	8,944
thermal	779	20,320	773	4	2	78	10.78%	11,376

(b) Dependency Function Number

Original	SW Driver Dep Funcs		
Dep Funcs	Linux	LDR	Total
105	35	7	42
38	13	3	16
42	14	4	18
26	13	4	17
26	14	3	17
42	17	4	21

UNT: untouched code, **DEL**: deleted code, **CHA**: changed code, **ADD**: newly added code. **MOD rate** = *(DEL+CHA+ADD)/(Original Driver LoC)*.

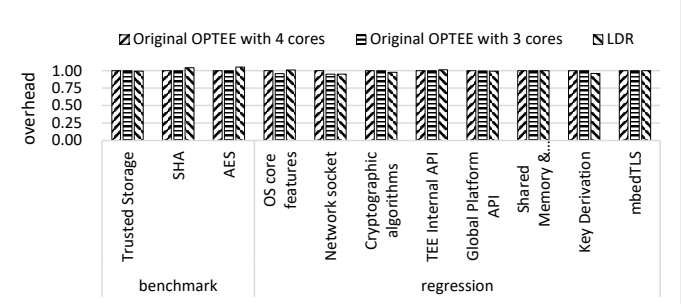


Fig. 7: OPTEE xtest Results

E. Micro Benchmarks

We measure the latency of the Linux subsystem function redirection and the CPC. For each function, we measure the elapsed time of 1,000 continuous calls for 4 rounds and take the average value as the call latency for this function.

1) Linux Subsystem Function Redirection: To evaluate the latency of a redirected Linux subsystem function call, we prepare five functions with 0 to 4 parameters respectively. The results show that the latency of an LDR redirected function call is 11.14 μs, and this means that the LDR Linux function redirection mechanism introduces relatively low latency. Figure 8-(a) also illustrates that the number of arguments has negligible impact on the overall latency.

2) Cross-world Procedure Call (CPC) Mechanism: To measure the CPC latency, we customize a LKM consisting of empty functions with different parameters. We leverage 11 functions with 0 to 10 regular parameters respectively to evaluate the influence of the number of parameters on the call latency. Then, we leverage functions with parameters as pointers to data chunks of different sizes, ranging from 32 bytes to 512 bytes, to evaluate the influence of the size of parameters on the call latency.

Figure 8-(b) shows that the average latency for a CPC-based function call without parameters is 117.78 μs. The average latency for functions with parameters of regular values and data pointers are 140.02 μs and 140.29 μs respectively. Function calls without arguments take less time since the function invocation metadata (e.g., function name) is passed using registers, and for other functions their arguments are passed through shared memory. The results also show that the number or the size of parameters has little influence on the CPC latency, and thus the parameter marshalling and unmarshalling process introduces negligible overheads.

A further observation is that the CPC latency is approximately ten times that of a redirected subsystem function call. Such difference originates from the more complex processes involved in the CPC code path, including OPTEE thread allocation, session establishment/teardown, etc. In comparison, the code path of a redirected subsystem function call is more lightweight as it employs the current OPTEE session. This difference is the result of OPTEE's internal implementation.

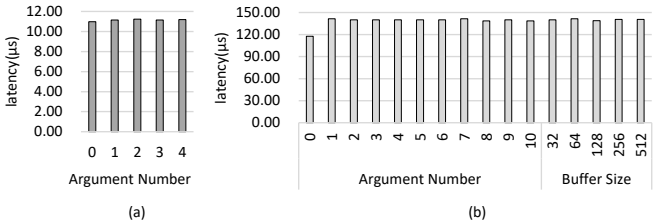


Fig. 8: Latency of Linux Function Redirection and CPC Mechanism

F. Case Studies

We evaluate LDR through real-world case studies using diverse sensor devices in Table I. Furthermore, we demonstrate the robustness of our LDR and empirically verify the feasibility of the secure OS library function substitution approach. First, we read the temperature data using the IMX6Q thermal sensor from the SW. Then, we measure the sampling rate of the barometer, the magnetometer, the accelerometer and the ambient light sensor from the SW. Finally, we leverage the camera sensor and the IMX6Q IPU to conduct evaluations on use cases of image capturing and video streaming. All the devices are driven by their corresponding SW drivers.

1) Temperature Capturing: For the case study on thermal sensor, we focus on the temperature measurements and aim to read the SoC temperature from the SW using the SW thermal driver. We first export the temperature data reading function of the SW thermal driver and then read the temperature data inside the secure OS using the SW thermal driver. The results show that the SW thermal driver can work properly inside LDR with all the temperature data read correctly.

2) Environmental Data Sampling: We use four I2C-based sensor drivers to show that LDR can support sensors with bus I/Os inside the SW. Similar to the thermal sensor, we export data reading functions of each SW driver and leverage a greedy polling method to read data from the sensor by invoking functions exported from the corresponding SW driver.

To evaluate the performance of the SW drivers, we set up two sets of experiments as before and conduct 10 rounds of data reading to measure their sampling rates with each round lasting about 3 seconds. Since the sampling rate of MPL3115

is relatively low, its tests last 10 seconds for each round. The default sampling rate and the measured sampling rate using the LDR SW drivers are listed in Table IV. The results show that the LDR SW drivers can fully exert the performance of the devices without compromising sampling rate.

TABLE IV: Sensor Date Sampling Performance

Device	Default SR	LDR without IED		LDR with IED	
		SR	Difference	SR	Difference
MAG3110	80	83.56	+4.4%	84.82	+6.0%
MMA8451	800	809.61	+1.2%	800.03	+0.0%
ISL29023	11.1	11.47	+3.4%	11.17	+0.6%
MPL3115	1	1.09	+8.7%	1.05	+5.3%

3) Image Capturing & Video Streaming: An OV5640 camera module is physically connected to an Image Processing Unit (IPU) through a MIPI CSI interface on the evaluation board. The IPU processes the captured image and inserts image frames into the V4L2 video buffer. Consequently, in the case study of image capturing and video streaming, we adapt an IMX6Q IPU driver into LDR and leave the OV5640 camera driver inside the Linux kernel. Moreover, since there is neither image processing nor video codec libraries inside the SW, we choose to transmit the captured image frames to the NW applications which we assume to be protected from a malicious NW kernel using techniques investigated in [43], [44], [45], [46], [47].

We adapt an IMX6Q IPU driver into LDR. To begin with, we first discard `ipu_device.c` from the IPU project because it is concerned with VFS operations. Then, we replace the `init()` and `probe()` functions of the SW IPU driver with SW customized ones which create proper SW memory mapping on the IPU registers, conduct SW exclusive initialization on the SW state variable and register IPU interrupt handlers. Additionally, as discussed in §IV-D, we exclude the fourth core (#3) from NW usage and dedicate it to handling interrupts issued from the IPU by setting the interrupt affinity using the generic interrupt controller [48]. Finally, as the IPU driver is required by another NW driver called `mxc_v4l2_capture`, we create CPC stubs for the relevant exported functions in both the NW and SW IPU drivers. These stubs forward invocations to the SW IPU driver during runtime for image frame retrieval.

Image Capturing. First, we evaluate the performance of the SW IPU driver for image capturing with various resolutions (480P, 720P, and 1080P). We use the image capturing tool, i.e., *v4l2-ctl*, to grab images from the camera and store them locally. For both the original IPU driver and the LDR IPU driver, we take three cases of image capturing with 1 shot, 10 shots, and 100 shots, respectively. For each case, we conduct 10 rounds of image capturing experiments and the average time is calculated. As illustrated in Figure 9, the results show that the time differences among the original and the LDR IPU driver are quite small, showing that LDR introduces negligible latency overheads for image capturing.

Video Streaming. Finally, we evaluate the performance of the SW IPU driver for video streaming. We use *FFmpeg* [49] to stream the video using the RTP protocol [50] and the video is encoded in x264 format. Since we implement pure-software video codec without hardware-based multimedia acceleration, the current LDR prototype can only support video streaming

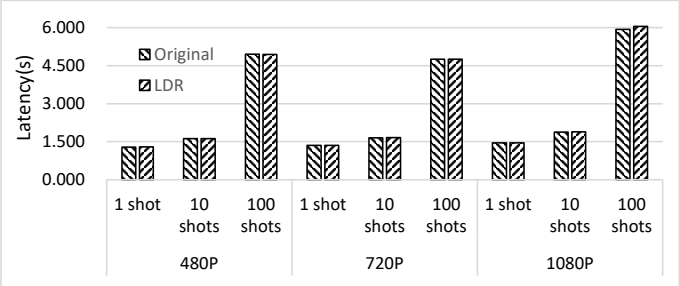


Fig. 9: Latency of Video Capturing under Different Resolution with Both Original and LDR IPU Driver

under resolution of 480P and 720P [51]. Additionally, to get a better streaming performance, we compile FFmpeg with ARM Neon features *enabled*. For both IPU driver setups, we take 10 rounds of video streaming with 400 frames for each round and the resulting streaming statistics is collected, namely FPS, duplicated frames per 100 frames, dropped frames per 100 frames and streaming speed. The streaming speed is an indicator defined by *FFmpeg* to reflect video stream processing performance like codec, etc. and the higher the streaming speed is, the better performance is achieved. The video stream latency is less than 1 second, and the average of each statistics is shown in Table V. The results show that LDR introduces no speed penalty to 480P streaming while only -2.43% speed downgrade to 720P streaming, indicating that LDR is ready for most video streaming tasks.

TABLE V: FFmpeg Streaming using OV5640 under Different Resolution with Both Original and LDR IPU Driver

Res	Driver	FPS	Dup F per 100 F	Drop F per 100 F	Stream Speed	Speed Penalty
480P	Orginal	25	0	9	0.9999×	-
	LDR	25	0	9	0.9999×	0.00%
720P	Orignal	24.6	0	19.5	0.9945×	-
	LDR	24	8.65	4.95	0.9703×	-2.43%

G. LDR vs Other Secure I/O Systems

We compare LDR with other two state-of-the-art secure I/O systems in terms of performance, security trade-offs and engineering efforts.

*1) **LDR vs Driverlets**:* Both LDR and Driverlets[13] aim to facilitate driver support inside TEEs yet take totally different approaches.

Performance. Driverlets provides an Replayer that replays pre-recorded I/O events to reproduce a device functionality. Such approach introduces 11%-270% latency overhead for frame capturing compared to the native Linux driver. In contrast, by directly running compiled driver binary inside the SW IED, LDR drivers can run in native speed as the original drivers. As shown in §VI-F3, LDR outperforms Driverlets with almost no extra latency for frame capturing. Additionally, LDR can support smooth video streaming in 24 FPS, which Driverlets is incapable of due to its heavy overhead.

Security Trade-offs. Driverlets runs completely inside the TEE with no interactions with the NW. Though ensuring strong security guarantees, such arrangement hinders efficient data processing since the TEE is lack of sophisticated libraries. For example, the SW has no image or video codec libraries

and thus cannot support efficient image encoding or video streaming. Although LDR can fully support SW-closed data processing, we also foresee more application diversity by allowing data to be processed by NW applications after proper encryption or digital signing. In effect, LDR is more preferred being combined with widely-studied application protection schemes [43], [44], [45], [46], [47] and secure data sharing mechanisms [52], [53], [54], [55]. With the guarantee on the integrity of data sources provided by LDR, data captured by LDR drivers can be transmitted to a fortified NW user application through a secure channel and get processed there, realizing full-lifetime data protection as well as efficient and diverse data processing.

Engineering Efforts. In general, Driverlets takes less engineering efforts since its record & replay workflow is automated and the generated I/O template is properly generalized for I/O operations with similar behaviors. Nevertheless, since LDR leverages automatic function removal and most driver code remain untouched, the engineering efforts to generate a SW driver is reasonably small.

2) ***LDR vs MyTEE****:* LDR and MyTEE [16] take different approaches to reusing the existing Linux drivers to enable secure I/O inside compact TEE OSes. Specifically, MyTEE leaves NW drivers inside the NW Linux kernel and allows them to access secure peripherals through temporary supervised privilege escalation while LDR enables native TEE drivers by providing a driver runtime inside the SW.

We compare MyTEE with LDR in terms of engineering efforts, security considerations and performance. Both MyTEE and LDR require manual driver code analysis and modification to driver code, and offer similar security guarantees by excluding drivers from the TCB as well as protecting drivers from NW attacks. To investigate the potential performance overhead introduced by MyTEE, we implement a sensor data reading module inside the secure OS and reuse the corresponding drivers inside the NW Linux kernel for device I/O following MyTEE approaches of secure I/O. Since our evaluation board (Cortex-A9) does not support virtualization extensions [56], we build a hypothetical experiment setup where no privilege escalation and I/O logging operations are involved. Note that such experimental setup is only for performance evaluation purposes and can reveal an optimized performance upper bound of MyTEE approaches. We test the sampling rate (SR) of sensors evaluated in §VI-F2 using the sensor data reading module and the results are shown in Table VI. For sensors with relatively low sampling rate (less than 100), MyTEE introduces negligible performance. However, for MMA8451, an accelerometer with high sampling rate, MyTEE can only achieve a sampling rate of 610.19, 23.73% lower than the sampling rate achieved by LDR. The reason is each MyTEE I/O request involves 2 NW & SW switches as well as 2 Linux kernel & hypervisor switches. A lower SR is expected if all MyTEE security mechanisms including privilege escalation and I/O logging are enabled, adding more overhead. In comparison, LDR avoids world switches on performance-critical I/O path and thus gains better performance.

TABLE VI: Sensor Data Sampling Rate (SR) Comparison with MyTEE Approaches

Device	Default SR	LDR SR	MyTEE SR	MyTEE vs LDR
MMA8451	800	800.03	610.19	-23.73%
MAG3110	80	84.82	82.76	-2.43%
ISL29023	11.1	11.17	11.17	0.00%
MPL3115	1	1.05	1.10	4.76%

VII. Security Analysis

In this section, we first discuss how LDR protects SW drivers from NW attacks. Then, we evaluate the effectiveness of LDR security mechanisms against attacks from untrusted SW drivers.

A. Protecting SW Drivers from NW Attacks

LDR protects SW driver from NW attacks in both initialization and execution phases.

1) Driver Initialization Phase: As discussed in §III, both NW and SW drivers are loaded and initialized during the system boot phase. The trusted/secure boot can ensure that the states of the Linux kernel and the NW driver are trusted and valid at boot time. By copying the trusted NW state variable from the NW driver, the SW driver has a legal initial state.

2) Driver Execution Phase: During execution phase, the SW state variable resides inside the SW and its SW-exclusive fields are protected by LDR secure state variable maintenance mechanisms as discussed in §IV-F. The potential attack surfaces introduced by redirected Linux subsystem calls can be analyzed in terms of passed arguments, the function itself and the return value. With the state variable synchronization & shadowing mechanism, LDR ensures that only necessary Linux kernel subsystem objects are synchronized to the NW driver and used as parameters. Therefore, no security-critical driver data is leaked to the NW on each redirected call and the SW driver's data security is enforced.

During execution, the Linux subsystem function cannot access SW memory and thus cannot jeopardize the SW driver's code integrity and data security. Additionally, after the removal of a driver's VFS interface implementation functions and its Init & Conf functions, there are minimum kernel subsystem function calls inside the SW driver and the current LDR prototype only needs to redirect 7 such calls, as discussed in §V. These functions are involved with clock management and time delay. A malicious Linux kernel may close the device's clock source or delay a driver operation, causing denial-of-service, which we exclude in our threat model. Moreover, such DoS attacks do not undermine the SW driver's code integrity and data security.

An attacker may try to craft malicious return values of the redirected Linux kernel subsystem calls. To mediate malicious return values, the LDR's driver dependency function call validation mechanism can be extended with proper return value sanitizing and any violation against the Linux kernel subsystem function specification will be recognized.

B. Protecting Secure OS from Untrusted SW Drivers

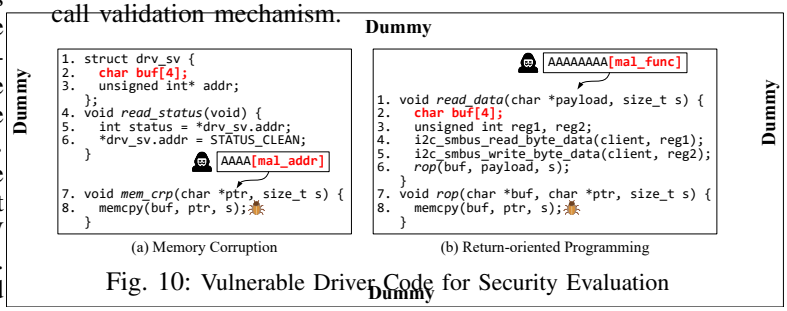
We first conduct theoretical security analysis on the effectiveness of LDR security mechanisms against various attacks issued from untrusted SW drivers. Then, we conduct a case study on a vulnerable driver and show how LDR security mechanisms mediates the corresponding exploits.

1) Theoretical Security Analysis: Since NW applications can invoke SW driver services through CPC, an attacker may exploit a SW driver's vulnerabilities and tries to steal or corrupt the secure OS data by launching various attacks, including memory corruption, code injection, control flow hijacking. ***1) Memory Corruption.*** Our LDR leverages the ARM DAC feature to create an isolated execution domain for each SW driver. During execution, the SW driver can only access memory belonging to its own domain, thus preventing it from corrupting memory regions of the secure OS and other SW drivers. ***2) Code Injection.*** Since the SW driver' data segments are mapped as non-executable and the `.text` segments are non-writable, the attacker cannot launch code injection attacks by dynamically injecting malicious code to the SW driver. Additionally, since the page table resides in the secure OS, the attacker cannot access the page table from the SW driver. Therefore, the attacker is not able to convert the memory attributes enforced by the page table. Furthermore, the attacker may try to invoke memory mapping functions (e.g. `mprotect()`, `remap()`, etc.) to change the page table memory attributes. With the dependency function call validation scheme, every secure OS function call is mediated by the IED gate and any attempted call to these memory attribute manipulation functions is denied. ***3) Control Flow Hijacking.*** Due to the forward control flow integrity enforced by the dependency function validation scheme, the attacker cannot invoke any function other than the callee recorded during SW driver linking from a particular dependency function callsite. Additionally, since the IED gate is atomic, single-threaded and it closes access to the secure OS before transferring control to the SW driver, it is impossible for the attacker to exploit the DACR manipulation ROP gadgets inside the IED gate to revoke the DAC access control policies. Furthermore, developer can be suggested to leverage static code scanning tools to search for these ROP gadgets inside the SW driver binary so that the binary itself does not contain any exploitable DACR modification ROP gadgets.

2) CVE Case Studies: We conduct several case studies to further evaluate the effectiveness of LDR security mechanisms. We create a SW driver containing a buffer overflow vulnerability, as specified in multiple CVEs related with drivers (2021-28972 [57], 2020-12653 [58], 2018-3580 [59]). We try to exploit such vulnerability by launching memory corruption and ROP attacks and verify whether LDR can defeat them.

Memory Corruption. We try to launch memory corruption attacks from the vulnerable driver by leveraging the buffer overflow vulnerability shown in Figure 10-(a). We design a function named *read_status()*, which reads from a device status register (*line 5*) and then clears the status register (*line 6*). The address of the device register, *i.e.*, `addr`, is stored in the driver's state variable (*line 3*). The victim buffer `buf` is defined at *line 4*. The function *mem_crp()* contains a buffer overflow vulnerability without boundary checking on input payload (*line 8*). An attacker can pass a malicious payload to *mem_crp()* and consequently overwrite `addr` with a malicious address, *i.e.*, `mal_addr`. When *read_status()* is invoked afterwards, it will read from and write to `mal_addr`, resulting in memory corruption attacks. We point `mal_addr` to a secure OS object as well as the state variable of another driver residing inside of a different IED. LDR defeats these memory corruption attacks with the DAC-based memory isolation mechanism, defeating memory corruption attacks.

Return-oriented Programming. We try to launch ROP attacks by leveraging the stack overflow vulnerability shown in Figure 10-(b). Function `read_data()` has a local array *i.e.*, `buf`, which is allocated on the function stack frame upon invocation. It also invokes Function `rop()` which contains a stack overflow vulnerability at *line 8* without boundary checking. An attacker can pass a malicious payload to `read_data()` and overwrite the return address stored on the stack with a malicious function address *i.e.*, `mal_func`. When `read_data()` returns, `mal_func` will be invoked, resulting in ROP attacks. We first point `mal_func` to a secure OS function. Such attack fails since the secure OS is configured as non-accessible during SW driver execution with the DAC-based driver isolation mechanism. We then try to exploit the IED gate by pointing `mal_func` to the IED gate entry. This attack also fails since LDR prevents a function from being called from an illegal callsite with the dependency function call validation mechanism.



(a) Memory Corruption (b) Return-oriented Programming

Fig. 10: Vulnerable Driver Code for Security Evaluation

VIII. Discussion

In this section, we first analyze how LDR design choices can be applied to 64-bit ARM processors. Then, we show how more complex devices will influence the design considerations. Finally, we discuss the limitations of the current LDR prototype and future work.

A. LDR with AArch64 Platforms

LDR can be instantiated on 64-bit ARM, aka. AArch64, platforms with minor modifications. Since DAC features have been deprecated for AArch64 processors, debug watchpoints [60] can be leveraged to realize IED on AArch64 platforms. Specifically, before entering an IED, the IED gate configures watchpoints to monitor memory regions belonging to the secure OS, TAs and other IEDs. Any access beyond the current IED memory region and the IED gate will trigger a watchpoint exception, which is handled by the secure OS. On IED exit, the IED gate resets the watchpoints before transferring control to the secure OS.

B. LDR with more Complex Devices

Even though the current LDR prototype does not evaluate more complex devices like GPUs or NICs, we believe that LDR designs can be applied to these devices with additional driver compartmentalization [61], [62]. Modern PCIe device drivers exhibit sophisticated yet complicated internal structures and consist of several intra-driver subsystems. For example, a modern GPU driver involves its own task scheduler, memory manager, components interacting with the Linux Direct Rendering Manager (DRM), etc. Instead of adapting the entire

driver into LDR, we imagine that such driver be divided into cooperative modules and only the security-critical modules such as resource manager be adapted into LDR while leaving modules responsible for user interaction and task scheduling inside the NW Linux kernel. In fact, such design choices have been investigated among GPUs [61], NICs [62] and even middlewares like VMM [63], container managers [64], etc. Considering the security-critical modules are more self-contained, LDR can provide excellent runtime environment for these security-critical modules.

C. LDR with TEEOS-native Drivers

We discuss the feasibility of another alternative design named the TEEOS-native approach in which a Linux driver, called the SW native driver, is directly embedded inside the secure OS and no NW driver is involved. Compared with LDR, such approach introduces more TCB increase. Without NW drivers, the Linux subsystem fields of the SW native driver need to be initialized through invoking multiple Linux subsystem functions. Therefore, extra subsystem redirection functions are needed for each driver as shown in Table VII. In total, another 119 [2] SW Linux subsystem functions need to be supported in LDR. These functions can be supported inside of the secure OS through either porting or redirection. On one hand, porting these functions to the secure OS involves a huge amount of engineering efforts and will introduce significant TCB expansion [13] as discussed in §II-B since these functions are tightly coupled with the Linux kernel. As shown in Table VIII, it is estimated that such porting will introduce 30 kLoC. On the other hand, if the TEEOS-native approach redirects calls upon these functions to the NW Linux kernel as LDR does (§IV-C), another 3,570 LoC is introduced for function redirection stubs. Additionally, for each subsystem function call, the TEEOS-native approach needs to conduct point-to analysis to identify all related objects that can be reached from the passed arguments [65], [17], [66] and synchronize them upon each call. The resulting deep copy [65] and synchronization code will further introduce several thousands of LoC. It is estimated that such redirection will introduce 15 kLoC, as shown in Table VIII.

In contrary, LDR avoids such TCB expansion by generating new SW drivers and leveraging the NW driver as an assistant driver. First, LDR generates a SW driver by removing unnecessary functions, namely Init & Conf and VFS functions, from the original Linux driver (§IV-B), which significantly reduces the number of required Linux subsystem functions. Consequently, the current LDR prototype only needs to support 7 Linux subsystem functions implemented in 185 LoC. Second, LDR leverages the NW driver as a helper driver to synchronize subsystem call arguments. For each redirected subsystem function, only a small number of driver state variable fields need to be synchronized and all related data objects can be found through the NW state variable §IV-F. Additionally, the LDR design is scalable for all subsystem functions without point-to analysis. Finally, LDR reuses `KBuild` to provide a driver development environment familiar to system developers and modularizes SW drivers for flexibility.

[2]Some subsystem functions are invoked by multiple drivers.

TABLE VII: Number of Extra Linux Subsystem Functions Needed for Each Driver with the TEEOS-native Approach

	IPU	ISL	MAG	MMA	MPL	Thermal
# of Subsys Funcs	67	25	28	15	16	25

TABLE VIII: Estimated Total LoC of the TEEOS-native Approach and Comparison with LDR

	TEEOS-native		LDR
	Porting	Redirection	
Estimated Introduced LoC (kLoC)	30	15	5

D. Limitations

Our current LDR prototype has three limitations: 1) A secure OS may not provide all library functions a driver needs. For library functions that the secure OS does not originally support, developers have to implement them. 2) The current LDR prototype does not provide advanced Linux kernel utilities like work queues, kernel threads, etc., therefore, we do not support complex NIC or GPU drivers as discussed in §VIII-B. However, we may delegate these scheduling work to the NW Linux kernel as proposed in TruZ-View [67]. 3) We manually analyse Linux drivers to identify their Init & Conf and VFS functions. However, we believe such process can be automated as investigated in CryptoMPK[68], PtrSplit[65], KSplit[17]. We plan to augment the current SW driver generation process with automatic driver code analysis and generation.

IX. Related Work

TrustZone-based Secure I/O. Many research projects have proposed various approaches for TrustZone-based secure I/O. SuiT [69] and TruZ-View [67] respectively propose trusted user interface (TUI) mechanisms for a touch display screen that allows users to input sensitive data into an application inside the SW. Oath [11] also proposes a TUI mechanism which can defend against physical memory disclosure attacks using the on-chip RAM. TrustSAMP [12] proposes a solution for an audio device to securely process copyrighted audio data by decrypting and playing DRM-related audio inside the SW. However, these systems have to port drivers for the corresponding devices, which involves significant engineering efforts. Driverlets [13] shares the same goal with LDR. Compared to the "record and replay" emulated approach employed by Driverlets, LDR runs compiled real-world driver modules inside the secure OS and thus achieves a much better performance. Meanwhile, LDR also confines drivers in ARM DAC-based sandboxes to prevent potential vulnerabilities of the SW drivers from compromising the TEE OS. MyTEE[16] leverages ARMv8 virtualization extensions to enable isolation primitives on platforms without common TrustZone extensions like TZPC, TZASC, etc. Similar to LDR, MyTEE reuses existing NW drivers inside the Linux kernel for secure I/O through temporary privilege escalation. However, MyTEE may incur heavy performance overhead for devices with freqeunct I/O requests due to world switches as well as kernel-hypervisor switches involved in each I/O operation.

Driver Isolation. The primary objective of most driver isolation research [70], [71], [72], [73], [74], [75], [62] is to confine potential driver crashes and vulnerabilities within a restricted environment, preventing damage to other kernel

components while preserving all driver functionalities. Such restricted environments can be established within user-space processes [70], [71], [73], [74] or virtual machines [72], [75], [62]. To ensure that driver services remain accessible to other system components, glue code is generated to coordinate communication and interaction between the isolated driver and other system components outside the restricted environment. These works assume that the Linux kernel is trusted. However, our LDR design choices have an opposite assumption where the NW including the Linux kernel is untrusted. Consequently, LDR proposes multiple security mechanisms to protect SW drivers from potential attacks issued from the NW. Moreover, these works redirect all function calls to the Linux kernel while LDR achieve better performance by reusing existing TEE OS library functions whenever possible and redirecting the kernel subsystem function calls to the Linux kernel in the NW.

Domain-based Sandboxing. Recently, several research studies have explored efficient user-space sandboxing schemes that leverage domain-based memory access control primitives available on most commodity computer systems. ERIM [76] implements a user-process data encapsulation mechanism based on Intel memory protection key (MPK) through carefully-designed call gates, binary inspection, and binary rewriting. Hodor [77] proposes a new OS abstraction called a protected library using MPK. Unlike ERIM, Hodor employs debug registers to prevent the execution of unsafe MPK manipulation instructions. Other research studies, such as EPK [78], xMP [79], and SeCage [80], utilize hardware virtualization support, namely extended page table (EPT) and `VMFUNC`, to achieve efficient domain switching primitives similar to those supported by MPK. Distinct from these projects, LDR employs ARM DAC to create kernel-level sandboxes, providing secure driver support within TEEs.

X. Conclusion

We present the Linux driver runtime (LDR), a driver LKM execution environment inside TrustZone-based secure OSes that facilitates secure and efficient TEE driver support. We are the first to design and implement a driver execution environment inside compact TEE OSes by reusing existing TEE OS library functions and redirecting kernel subsystem functions to the NW Linux kernel. Additionally, we create a sandbox environment to confine SW drivers and prevent attacks issued from the potentially vulnerable SW drivers. We evaluate the feasibility of LDR on an NXP IMX6QSABRESD board using various on-board devices and the experimental results show that LDR introduces negligible overheads to real-world applications. LDR is now available from: https://github.com/SparkYHY/Linux-Driver-Runtime.

Acknowledgments

We thank the anonymous reviewers for their constructive comments and suggestions that have helped us substantially improve the quality of this paper. This research was supported in part by National Natural Science Foundation of China Grant Nos. 62022024, 61972088, 62232004, 62072103, 62102084, 62072102, 62072098, 61972083, and 62132009, by US National Science Foundation (NSF) Awards 1931871, 1915780, and US Department of Energy (DOE) Award DE-EE0009152, Jiangsu Provincial Key Laboratory of Network and Information Security Grant No. BM2003201, Key Laboratory of Computer Network and Information Integration of Ministry of Education of China Grant Nos. 93K-9, and Collaborative Innovation Center of Novel Software Technology and Industrialization. Any opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, p. 86, 2016.

[2] I. Advanced Micro Devices. (2023) Amd secure encrypted virtualization (sev). [Online]. Available: https://www.amd.com/en/developer/sev.html#:~:text=Usesonekeypervirtual,guestoperatingsystemandhypervisor.

[3] A. Ltd. (2011) Arm architecture reference manual armv7-a and armv7-r edition. [Online]. Available: https://developer.arm.com/documentation/ddi0406/c/

[4] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song, "Keystone: an open framework for architecting trusted execution environments," in *Proceedings of the 15th European Conference on Computer Systems, EuroSys*, 2020.

[5] SAMSUNG. (2023) Stay connected, protected, and productive. discover the knox security platform and business solutions. [Online]. Available: https://www.samsungknox.com/en

[6] Linaro. (2023) Open portable trusted execution environment, optee. [Online]. Available: https://www.op-tee.org/

[7] Sierraware. (2016) Sierratee for arm® trustzone® and mips. [Online]. Available: https://www.sierraware.com/open-source-ARM-TrustZone.html

[8] F. Brasser, D. Gens, P. Jauernig, A. Sadeghi, and E. Stapf, "SANCTUARY: arming trustzone with user-space enclaves," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium, NDSS*, 2019.

[9] J. S. 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 Trans. Dependable Secur. Comput.*, vol. 15, no. 5, pp. 797–810, 2018.

[10] M. H. Yun and L. Zhong, "Ginseng: Keeping secrets in registers when you distrust the operating system," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium, NDSS*, 2019.

[11] D. Chu, Y. Wang, L. Lei, Y. Li, J. Jing, and K. Sun, "Ocram-assisted sensitive data protection on arm-based platform," in *Proceedings of the 24th European Symposium on Research in Computer Security, ESORICS*, 2019.

[12] Y. Li, L. Lei, Y. Wang, J. Jing, and Q. Zhou, "Trustsamp: Securing streaming music against multivector attacks on ARM platform," *IEEE Trans. Inf. Forensics Secur.*, vol. 17, pp. 1709–1724, 2022.

[13] L. Guo and F. X. Lin, "Minimum viable device drivers for arm trustzone," *Proceedings of the 17th European Conference on Computer Systems, EuroSys*, 2022.

[14] QEMU. (2023) Qemu. [Online]. Available: https://www.qemu.org/

[15] Q. Wiki. (2023) Qemu supported machines. [Online]. Available: https://wiki.qemu.org/Documentation/Platforms/ARM

[16] S. Han and J. Jang, "Mytee: Own the trusted execution environment on embedded devices," in *Proceedings of the 30th Annual Network and Distributed System Security Symposium, NDSS*, 2023.

[17] Y. Huang, V. Narayanan, D. Detweiler, K. Huang, G. Tan, T. Jaeger, and A. Burtsev, "Ksplit: Automating device driver isolation," in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2022.

[18] Linaro. (2019) Imx6q mma8451 accelerometer sensor driver. [Online]. Available: https://github.com/nxp-imx/linux-imx/blob/imx_4.14.98_2.3.0/drivers/hwmon/mxc_mma8451.c

[19] ——. (2019) Imx6q image processing unit (ipu) driver. [Online]. Available: https://github.com/nxp-imx/linux-imx/tree/imx_4.14.98_2.3.0/drivers/mxc/ipu3

[20] ——. (2019) Imx6q mag3110 magnetometer sensor driver. [Online]. Available: https://github.com/nxp-imx/linux-imx/blob/imx_4.14.98_2.3.0/drivers/hwmon/mag3110.c

[21] ——. (2019) Imx6q thermal sensor driver. [Online]. Available: https://github.com/nxp-imx/linux-imx/blob/imx_4.14.98_2.3.0/drivers/thermal/imx_thermal.c

[22] R. Love, *Linux Kernel Development*, 3rd ed. Addison-Wesley Professional, 2010.

[23] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. M. Eyers, R. Kapitza, P. R. Pietzuch, and C. Fetzer, "SCONE: secure linux containers with intel SGX," in *Proceeding of 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2016.

[24] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 2:1–2:34, 2012.

[25] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, "Trusense: Information leakage from trustzone," in *Proceedings of the 37th IEEE Conference on Computer Communications, INFOCOM*, 2018.

[26] 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 *Proceedings of the 27th USENIX Security Symposium, USENIX Security*, 2018.

[27] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Proceedings of the 40th IEEE Symposium on Security and Privacy, S&P*, 2019.

[28] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, "Vigilare: toward snoop-based kernel integrity monitor," in *Proceedings of the 19th ACM Conference on Computer and Communications Security, CCS*, 2012.

[29] A. Markuze, A. Morrison, and D. Tsafrir, "True IOMMU protection from DMA attacks: When copy is faster than zero copy," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2016.

[30] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Commun. ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[31] A. Ltd. (2004) Arm trustzone protection controller. [Online]. Available: https://developer.arm.com/documentation/dto0015/a/about-the-trustzone-protection-controller#:~:text=TheTrustZoneProtectionController(TZPC,systeminaTrustZonedesign.

[32] Y. Cheng, X. Ding, and R. H. Deng, "Driverguard: Virtualization-based fine-grained protection on I/O flows," *ACM Trans. Inf. Syst. Secur.*, vol. 16, no. 2, p. 6, 2013.

[33] Bootlin. (2019) Linux kernel source code v4.14.98. [Online]. Available: https://elixir.bootlin.com/linux/v4.14.98/source

[34] Linaro. (2020) Imx optee os 3.10.0. [Online]. Available: https://github.com/nxp-imx/imx-optee-os/tree/imx_4.14.98_2.3.0

[35] Z. Ling, H. Yan, X. Shao, J. Luo, Y. Xu, B. Pearson, and X. Fu, "Secure boot, trusted boot and remote attestation for ARM trustzone-based iot nodes," *J. Syst. Archit.*, vol. 119, p. 102240, 2021.

[36] J. Gu, H. Li, W. Li, Y. Xia, and H. Chen, "EPK: scalable and efficient memory protection keys," in *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC*, 2022.

[37] Z. Yuan, S. Hong, R. Chang, Y. Zhou, W. Shen, and K. Ren, "Vdom: Fast and unlimited virtual domains on multiple architectures," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2023.

[38] 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*, 2017.

[39] AlDanial. (2023) Count lines of code. [Online]. Available: https://github.com/AlDanial/cloc

[40] Linaro. (2020) Imx linux 4.14.98. [Online]. Available: https://github.com/nxp-imx/linux-imx/tree/imx_4.14.98_2.3.0

[41] intel. (2018) lmbench 3.0. [Online]. Available: https://github.com/intel/lmbench

[42] Linaro. (2019) Imx optee test. [Online]. Available: https://github.com/nxp-imx/imx-optee-test/tree/imx_4.14.98_2.3.0

[43] J. Criswell, N. Dautenhahn, and V. S. Adve, "Virtual ghost: protecting applications from hostile operating systems," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2014.

[44] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: secure applications on an untrusted operating system," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2013.

[45] 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, MobiSys*, 2017.

[46] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. S. Dwoskin, and D. R. K. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2008.

[47] A. Baumann, M. Peinado, and G. C. Hunt, "Shielding applications from an untrusted cloud with haven," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2014.

[48] A. Ltd. (2013) Arm generic interrupt controller, v2.0. [Online]. Available: https://developer.arm.com/documentation/ihi0048/latest/

[49] FFmpeg. (2023) Ffmpeg. [Online]. Available: https://github.com/FFmpeg/FFmpeg

[50] Wikipedia. (2023) Real-time transport protocol. [Online]. Available: https://en.wikipedia.org/wiki/Real-time_Transport_Protocol

[51] N. Community. (2013) H-264-decoding-using-ffmpeg-using-gpu-for-display-acceleration. [Online]. Available: https://community.nxp.com/t5/i-MX-Processors/H-264-decoding-using-ffmpeg-using-GPU-for-display-acceleration/m-p/291554

[52] J. S. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, "Secret: Secure channel between rich execution environment and trusted execution environment," in *22nd Annual Network and Distributed System Security Symposium, NDSS*, 2015.

[53] J. Jang and B. B. Kang, "Securing a communication channel for the trusted execution environment," *Comput. Secur.*, vol. 83, pp. 79–92, 2019.

[54] ——, "Retrofitting the partially privileged mode for tee communication channel protection," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 5, pp. 1000–1014, 2020.

[55] J. Wang, Y. Wang, L. Lei, K. Sun, J. Jing, and Q. Zhou, "Trustict: an efficient trusted interaction interface between isolated execution domains on ARM multi-core processors," in *Proceedings of the 18th ACM Conference on Embedded Networked Sensor Systems, Sensys*, 2020.

[56] A. Ltd. (2023) Arm cortex-a processor comparison table. [Online]. Available: https://developer.arm.com/documentation/102826/latest/

[57] NVD. (2023) Cve-2021-28972. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-28972

[58] ——. (2023) Cve-2020-12653. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2020-12653

[59] ——. (2023) Cve-2018-3580. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2018-3580

[60] A. Ltd. (2023) Arm architecture reference manual for a-profile architecture, chapter d2.10 watchpoint exceptions. [Online]. Available: https://developer.arm.com/documentation/ddi0487/ja

[61] Y. Deng, C. Wang, S. Yu, S. Liu, Z. Ning, K. Leach, J. Li, S. Yan, Z. He, J. Cao, and F. Zhang, "Strongbox: A GPU TEE on arm endpoints," in

Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security, CCS, 2022.

[62] V. Narayanan, A. Balasubramanian, C. Jacobsen, S. Spall, S. Bauer, M. Quigley, A. Hussain, A. Younis, J. Shen, M. Bhattacharyya, and A. Burtsev, "Lxds: Towards isolation of kernel subsystems," in *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC*, 2019.

[63] D. Li, Z. Mi, Y. Xia, B. Zang, H. Chen, and H. Guan, "Twinvisor: Hardware-isolated confidential virtual machines for arm," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP*, 2021.

[64] Z. Hua, Y. Yu, J. Gu, Y. Xia, H. Chen, and B. Zang, "Tz-container: protecting container from untrusted OS with ARM trustzone," *Sci. China Inf. Sci.*, vol. 64, no. 9, 2021.

[65] S. Liu, G. Tan, and T. Jaeger, "Ptrsplit: Supporting general pointers in automatic program partitioning," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2017.

[66] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha, "The design and implementation of microdrivers," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2008.

[67] K. Ying, P. Thavai, and W. Du, "Truz-view: Developing trustzone user interface for mobile OS using delegation integration model," in *Proceedings of the 9th ACM Conference on Data and Application Security and Privacy, CODASPY*, 2019.

[68] X. Jin, X. Xiao, S. Jia, W. Gao, D. Gu, H. Zhang, S. Ma, Z. Qian, and J. Li, "Annotating, tracking, and protecting cryptographic secrets with cryptompk," in *Proceedings of the 43rd IEEE Symposium on Security and Privacy, S&P*, 2022.

[69] Y. Cai, Y. Wang, L. Lei, Q. Zhou, and J. Li, "Suit: Secure user interface based on trustzone," in *Proceedings of the 53rd IEEE International Conference on Communications, ICC*, 2019.

[70] S. Boyd-Wickizer and N. Zeldovich, "Tolerating malicious device drivers in linux," in *Proceedings of the 2010 USENIX Annual Technical Conference, USENIX ATC*, 2010.

[71] S. Butt, V. Ganapathy, M. M. Swift, and C. Chang, "Protecting commodity operating system kernels from vulnerable device drivers," in *Proceedings of the 25th Annual Computer Security Applications Conference, ACSAC*, 2009.

[72] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, M. Williamson *et al.*, "Safe hardware access with the xen virtual machine monitor," in *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure, OASIS*, 2004.

[73] V. Ganapathy, A. Balakrishnan, M. M. Swift, and S. Jha, "Microdrivers: A new architecture for device drivers," in *Proceedings of the 11th Workshop on Hot Topics in Operating Systems HotOS*, 2007.

[74] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser, "User-level device drivers: Achieved performance," *J. Comput. Sci. Technol.*, vol. 20, no. 5, pp. 654–664, 2005.

[75] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz, "Unmodified device driver reuse and improved system dependability via virtual machines," in *Proceedings of the 6th Symposium on Operating System Design and Implementation OSDI*, 2004.

[76] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "Erim: Secure, efficient in-process isolation with protection keys (mpk)," in *Proceedings of 28th USENIX Security Symposium, USENIX Security*, 2019.

[77] M. Hedayati and S. Gravani, "Hodor: Intra-process isolation for high-throughput data plane libraries," in *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC*, 2019.

[78] J. Gu, H. Li, W. Li, Y. Xia, and H. Chen, "Epk: Scalable and efficient memory protection keys," in *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC*, 2022.

[79] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "xmp: Selective memory protection for kernel and user space," in *Proceedings of the 41st IEEE Symposium on Security and Privacy, S&P*, 2020.

[80] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2015.

APPENDIX A
AN ILLUSTRATIVE EXAMPLE OF SECURE OS FUNCTION CALL

We use an example where a SW driver is loaded into the IED, initializes itself by invoking `malloc()` to acquire a memory chuck from the IED heap and exits, to illustrate how driver isolation is enforced during the SW driver execution. First of all, the *SW driver loader* loads a SW driver into the IED heap with its `.text` segments mapped as `R-X` and its data segments mapped as `RW-`. Then, the *SW driver loader* invokes the SW driver's `sw_init()` function through the IED gate (Fig. 11-(b)). The IED gate switches from the secure OS stack to the IED stack and configures the secure OS and TA as non-accessible by setting the corresponding DACR bits as `NA` (Fig. 11-(c)). When the SW driver invokes `malloc()`, such secure OS function call is hooked to the IED gate where the invocation's validity is checked. If the secure OS function call is valid, the IED gate opens access to the secure OS and the secure OS function still uses the IED stack to avoid complex and heavy context switching. Then the secure OS allocates a dynamic memory chuck on the IED heap and returns to the IED gate (Fig. 11-(d)). Next, the IED gate closes access to the secure OS and further returns to the SW driver (Fig. 11-(e)). Finally, the SW driver finishes its initialization and returns back to the *SW driver loader* through the IED gate which makes the secure OS accessible and switches back to the secure OS stack (Fig. 11-(f)).

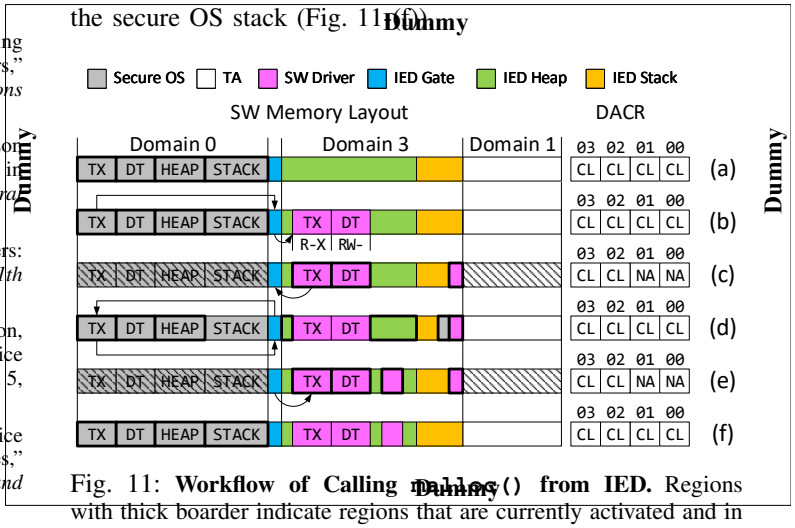


Fig. 11: **Workflow of Calling `malloc()` from IED.** Regions with thick boarder indicate regions that are currently activated and in control. TX stands for code. DT stands for data. IED gate is assigned to domain 2. IED is assigned to a new domain 3 and its memory layout consists of the IED heap and the dedicated stack.

APPENDIX B
FUNCTION REDIRECTION DURING INTERRUPT HANDLING

We observe that the SW interrupt handler may invoke some NW callback functions when handling the fast interrupt request (FIQ) triggered by the underlying device. Since the OP-TEE OS `thread_rpc_cmd()` interface cannot be used inside the interrupt context, we implement a FIQ Linux callback (FLC)

method to facilitate such NW callback invocation during SW FIQ handling.

The general FLC workflow is shown in Figure 12. To begin with, the SW interrupt handler's call to the Linux callback is hooked to the FLC SW module during SW driver linking. During runtime, the FLC SW module intercepts such call once the Linux callback is invoked (Fig. 12, ❶). The FLC SW module retrieves the callback address as well as the arguments and saves the current context of the FIQ handler (i.e. register states). Next, the FLC SW module encapsulates the callback address and the arguments into a shared memory buffer and then issues a secure monitor call (SMC) to trap into the secure monitor (SM) (Fig. 12, ❷). The SM turns on our customized FLC flag, saves the current NW state and jumps to the FLC NW module's entry point, i.e., `ldr_fiq_callback()`, through ARM exception return mechanism [3] (Fig. 12, ❸). The FLC NW module decapsulates the arguments, invokes the target callback (Fig. 12, ❹) and saves the return value back into the argument buffer. Then, it traps into the SM through an SMC (Fig. 12, ❺). Once taking control, the SM clears the FLC flag, restores the NW back to its original state and returns to the FLC SW module's return point, i.e. `callback_ret_entry()` (Fig. 12, ❻). The FLC SW module restores the FIQ handler context and saves the return value of the callback in R0 to emulate a typical function return (Fig. 12, ❼). Finally, the interrupt handler resumes and finishes what's left for FIQ handling (Fig. 12, ❽).

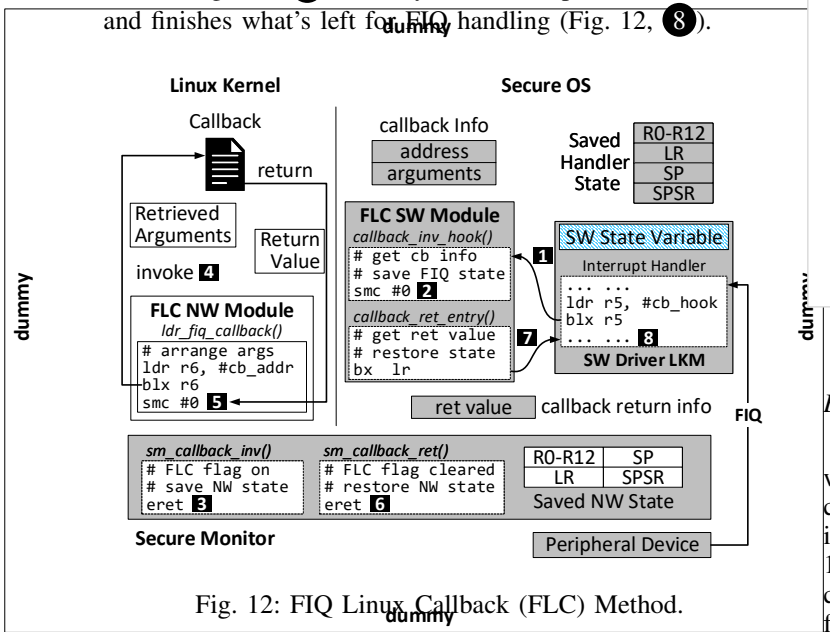


Fig. 12: FIQ Linux Callback (FLC) Method.

APPENDIX C
PERFORMANCE IMPLICATIONS OF LDR SECURITY MECHANISMS ON MICRO BENCHMARKS

We investigate the performance implications of LDR security mechanisms on the latency of the Linux subsystem function redirection and the cross-world procedure call.

A. Linux Subsystem Function Redirection

To evaluate how LDR security mechanisms influence the latency of a redirected Linux subsystem function call, we carry out three sets of experiments: 1) **baseline**. This set is used as the comparison group and is aimed to measure the basic function redirection latency without any security mechanisms. 2) **dependency function call validation**. This set enables the DAC-based driver isolation. We leverage drivers without state variables to measure the net latency introduced by dependency function call validation mechanism. 3) **state variable synchronization & shadowing**. This set is used to measure the latency introduced by state variable synchronization & shadowing mechanism. The results show that the latency of a basic redirected function call is 5.82 μs. When the DAC-based isolation is enabled inside LDR, an extra latency of 2.11 μs is introduced by the dependency function call validation mechanism on average. Moreover, the state variable synchronization mechanism introduces another latency of 3.21 μs.

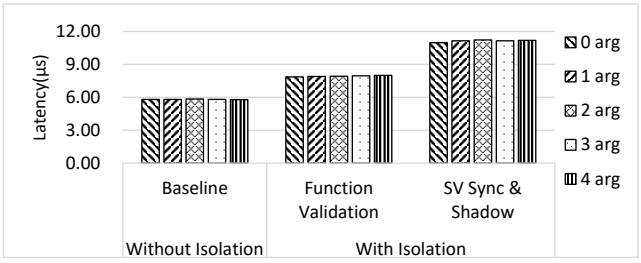


Fig. 13: Latency of Linux Function Redirection Mechanism

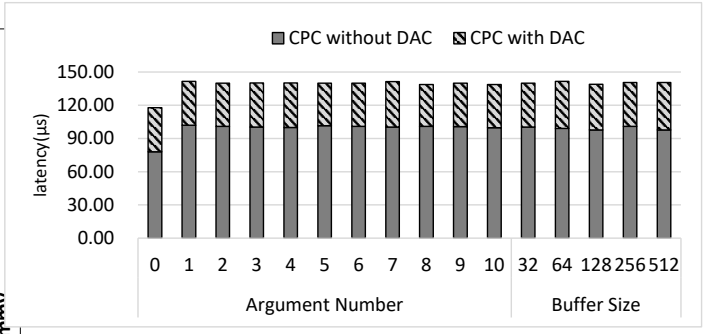


Fig. 14: Latency of CPC Mechanism

B. Cross-world Procedure Call (CPC) Mechanism

As for the CPC latency, we conduct two sets of experiments with one set to evaluate the basic CPC latency acting as comparison group and the other set to evaluate the latency introduced by LDR SW driver isolation mechanism. Figure 14 depicts that the average latency for a CPC-based function call without parameters is 77.91 μs. The average latency for functions with parameters of regular values and data pointers are 100.71 μs and 99.16 μs respectively. The LDR SW driver isolation mechanism introduces an extra 39.91 μs latency for all kinds of CPC calls on average.

TABLE IX: FFmpeg Streaming using OV5640 with different core and interrupt handling arrangements as well as security mechanisms. The resolution is set to 480P.

IPU Driver Arrangment	FPS	Dup F per 100 F	Drop F per 100 F	Stream Speed	Speed Penalty
Original (4 Cores)	24.9	0.00	1.80	0.9788×	-0.00%
Original (3 Cores)	24.6	0.08	1.43	0.9770×	-0.18%
LDR without IED	24.5	0.33	1.30	0.9771×	-0.17%
LDR with IED	24.0	0.98	1.05	0.9747×	-0.42%

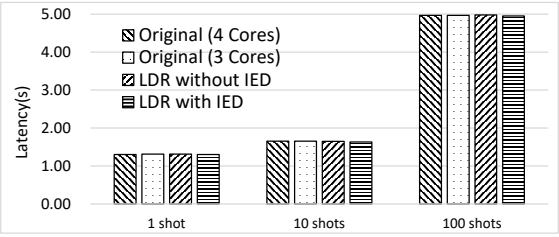


Fig. 15: Latency of video capturing with different core and interrupt handling arrangements as well as security mechanisms. The resolution is set to 480P.

APPENDIX D
PERFORMANCE IMPLICATIONS OF CORE ARRANGEMENT AND LDR SECURITY MECHANISMS ON IPU DRIVER

As mentioned in Section §IV-D, we have to allocate core exclusively for managing end-of-frame (EoF) interrupts initiated by the IPU upon completing a frame transmission, thus excluding it from NW usage. Consequently, we examine the performance implications of this core allocation on both image capture latency and video streaming performance. Additionally, we also investigate the performance implications of LDR security mechanisms, namely IEDs. We take 4 sets of experiments with different core and interrupt handling setups, which are **1)** Original IPU driver with 4 cores. **2)** Original IPU driver with 3 cores. **3)** LDR IPU driver without IED with SW exclusive EoF handling. **4)** LDR IPU driver with IED with SW exclusive EoF handling. To emphasize the effects of the core arrangement, we maintain a resolution of 480P for all subsequent experiments. Specifically, for video streaming experiments, we re-compile FFmpeg with ARM Neon features *disabled* by passing "`--disable-neon`" option for compilation.

For image capturing, the results displayed in Figure 15 reveals that the differences among various core and interrupt handling arrangements are minimal. The time taken for a single frame shot ranges from 1.30s to 1.32s, while for 10 shots, it varies between 1.64s and 1.66s, and for 100 shots, it spans from 4.95s to 4.98s.

For video streaming, the average of each statistic is presented in Table V. When disabling ARM Neon features, there is an average streaming speed decrease of -2.32%. We use the case where the original IPU driver is employed with four cores available to the rich OS as the baseline. The other 3 arrangements result in streaming speed penalties ranging from -0.42% to -0.17%. These findings demonstrate that reserving one core for EoF interrupt handling causes a negligible overall performance downgrade. Furthermore, the LDR security mechanisms have minimal impact on overall performance, with only a -0.25% streaming speed decrease compared to the case where LDR security features are disabled.