DEVLORE: Device Interrupt Protection for Confidential VMs

Andrin Bertschi*, Supraja Sridhara*, Mark Kuhne,
Benedict Schlüter, Friederike Groschupp, Clément Thorens, Nicolas Dutly,
Srdjan Capkun, and Shweta Shinde

ETH Zurich, Universitätstr. 6, 8092 Zurich, Switzerland
{firstname.lastname}@inf.ethz.ch

Abstract. Modern confidential computing executes sensitive computation in an abstraction called confidential VMs and protects from the hypervisor, host OS, and other co-resident VMs. It has been shown that an attacker can inject malicious interrupts to break the confidentiality and integrity of confidential VMs. We present DEVLORE, a device interrupt isolation mechanism that protects confidential VMs from interrupt manipulation attacks. Our design employs a delegate-but-check strategy by offloading interrupt management to the hypervisor, but adds correctness checks in the trusted software. We prototype our design on Arm Confidential Computing Architecture (CCA). We evaluate it on Arm FVP to demonstrate four diverse devices attached to confidential VMs and report costs on a Rock 5B board. Our case studies show the feasibility of real-world use cases and that DEVLORE incurs minimal overheads of 0.06% for typical integrated GPU applications.

Keywords: System Security · Confidential Computing · Trusted Execution Environments · Arm CCA · Interrupts

1 Introduction

Arm-based platforms are deployed in consumer electronic devices (e.g., phones, cars, laptops), edge devices, and the cloud. Applications running on these platforms typically use integrated devices (e.g., display, sensors, integrated GPUs) to perform sensitive tasks (e.g., inference, authentication) on-device or at edge. Cloud providers offer integrated accelerators connected to Arm CPUs (e.g., Grace-Hopper and Grace-Blackwell integrate Arm processors with state-of-the-art GPUs [51,52]). In these settings, it is important to safeguard users' sensitive computation on the platform from untrusted software (e.g., hypervisor executing on personal devices) and untrusted providers (e.g., cloud management).

Confidential computing enables users to execute their computation in isolation from privileged untrusted software. Arm Confidential Computing Architecture (CCA), similar to Intel TDX and AMD SEV-SNP, provides a confidential

* These authors contributed equally to this work.

virtual machine (CVM) abstraction with a new hardware extension called realm management extensions (RME) [39,6,14]. CCA isolates CVMs from each other and from untrusted hypervisors and host OSes. To achieve this, CCA introduces two new worlds called the realm and the root world. The root world has the highest privilege and exclusively executes the trusted firmware including the monitor. The realm world houses the CVMs and a Realm Management Monitor (RMM). These new worlds are in addition to two existing worlds: the normal/non-secure world for the host hypervisor and the secure world for trusted OS and apps.

Arm CCA-based CVMs executing in the realm world may need access to integrated devices on both consumer devices and the cloud. For example, to use a keypad or biometric sensor to perform authentication or to use an integrated GPU for computation on sensitive data in the cloud. However, CCA prohibits devices from accessing CVM memory. Recent works address this gap by allowing devices and CVMs to directly access each others' memory. They enforce memory isolation guarantees to ensure that device access does not compromise the CVM security [68,62,58]. On the other hand, recent works have shown that interrupt attacks from a malicious hypervisor can compromise the CVM execution [60,59].

To address this gap, we aim to enforce interrupt isolation—malicious interrupts injected by the attacker are never delivered to the CVMs, while ensuring that benign device interrupts are delivered correctly. Interrupt isolation presents challenges that are different from device memory isolation. For memory isolation, each device belongs to exactly one CVM. So, once the device is attached via memory setup, the hardware enforces the isolation without further software intervention. In contrast, the interrupt controller and its state is one resource that needs to be shared and multiplexed between the hypervisor and all the CVMs. Further, we not only need to isolate the interrupt configuration but also runtime delivery and acknowledgment.

We consider several potential designs to achieve this primitive of interrupt isolation. Our design exploration shows that obvious solutions are insufficient. Further, since enforcing the primitive entails reasoning about the entire interrupt lifecycle (registration, physical to virtual conversion, delivery, priority, acknowledgment), we have to weigh the design options for each phase. We isolate both physical and virtual interrupts without any software changes to existing device drivers. Our insight is to delegate the interrupt management and delivery to the untrusted hypervisor while the trusted software checks if the hypervisor misbehaves. Our approach minimizes software changes because of this delegate-but-check strategy while maintaining low performance overheads.

We prototype our design, called Devlore, on the Fixed Virtual Platform (FVP), a CCA-enabled emulator provided by Arm to demonstrate feasibility, correctness, and compatibility with hardware specification and software stack [11]. Since CCA is not yet publicly available in CPUs [17,16,20], we retrofit Devlore changes to a RK3588 board based on OpenCCA [20] to estimate Devlore performance. Devlore changes to the hypervisor, firmware, and the kernel (host and guest) are minimal (≈7.5k LoC), and without device driver changes.

We test the functional correctness of DEVLORE using our prototype on the FVP with four devices of varying complexity and isolation requirements: GPU, UART, Keyboard/Mouse, and LEDs/Buttons. Further, our three case-studies on the FVP show that DEVLORE has non-invasive impact on applications (IRC, terminal browser, text editor). Our measurements on the Rock 5B board performance prototype show that DEVLORE overheads are minimal. First, we run interrupt stress workloads using Mali GPU and UART on the Rock 5B board. Our experiments show that under sustained interrupt load, DEVLORE only incurs overheads up to 1%. Next, our case study using the glmark2 GPU benchmark that models typical integrated GPU applications shows that DEVLORE overhead drops to only 0.06%. DEVLORE is available at [4].

Contributions. The main contributions of this paper are:

- ***Interrupt Isolation.*** We design interrupt isolation to bridge the gap between memory and interrupts for integrated devices in CVMs on Arm CCA.
- ***Delegate-but-check.*** Our design, DEVLORE, delegates the interrupt management to the untrusted hypervisor while the trusted software checks its correctness.
- ***Devices.*** DEVLORE isolates device interrupts with no modifications to applications or device drivers and limited overheads. We demonstrate DEVLORE with devices and case studies evaluated on both the Arm FVP and an Arm board.

2 Problem & Insights

We motivate and scope the need for interrupt isolation.

Background: Interrupt Controller. An interrupt controller (GIC in Arm terminology) is a hardware component that delivers physical interrupt requests (IRQs) to cores. With virtualization, the hypervisor virtualizes the interrupts for VMs. To do this, the hypervisor leverages hardware support for virtual interrupt controllers (vGIC in Arm terminology). By programming a per-VM vGIC, the hypervisor can provide each VM with dedicated views of the interrupt controller. Through the vGIC, the VM receives and processes virtual interrupts. We give more comprehensive background in Sec 4.

2.1 Injecting Malicious Device Interrupts

We discuss how an attacker can inject fake interrupts to mount interrupt attacks.

Recent Interrupt-Based Attacks. Recent works have shown that a malicious hypervisor can use interrupt injections to compromise CVM security [60,59]. They observe that confidential computing solutions (e.g., TDX, SNP, CCA) delegate CVM management (e.g., scheduling, memory management, interrupt management) to the untrusted hypervisor. To manage interrupts and virtualize them for CVMs, the hypervisor controls the interrupt controller that delivers interrupts to the cores. So, the hypervisor can program the GIC to inject

interrupts directly into the CVMs at any point during their execution. Since the CVM is not aware of this, similar to Iago attacks [24], it breaks execution integrity. Next, we discuss potential threats to CVMs connected to devices.

Untrusted Devices. On any given Arm platform, an integrated device can only assert its designated interrupt line. So, integrated devices cannot send physical interrupts to fake interrupts from other devices that might be attached to CVMs (see Fig 1.a). Therefore, even if an attacker controls an integrated device they cannot use it to inject fake interrupts. Next, we discuss threats from untrusted management software (e.g., hypervisors) via interrupts [60,59].

Untrusted Management Software. The hypervisor manages the interrupts for the CVMs and can abuse it in two ways: (i) configure the GIC to trigger fake physical interrupts (Fig 1.b); and (ii) use virtual interrupt injection through the vGIC system registers to deliver fake virtual interrupts of either type (Fig 1.c). Next, we analyze device drivers to understand how they handle interrupts.

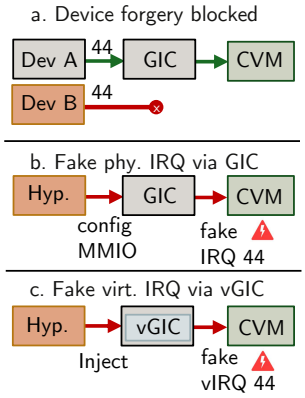

Fig. 1. Attacks. (a) Devices cannot change their physical interrupt numbers. Hypervisor injects fake physical (b) and virtual (c) interrupt (IRQ 44).

2.2 Driver Examples

To understand how device drivers handle interrupts, we manually analyzed the open-source drivers of popular integrated devices in Linux.

Counter. The Linux kernel counter subsystem implements an interrupt-driven counter. This counter can use any interrupt source (e.g., temperature sensor, proximity sensor) as an event source and increments a counter when it receives an interrupt (Lst 1.1). The counter implementation can be used by user-space applications to be notified when the counter is incremented, e.g., when an event has occurred for a fixed number of times (Line 5). Assume that the interrupt source for this counter is connected to a CVM and this counter is used by a CVM application. The untrusted hypervisor can maliciously inject interrupts, trigger this interrupt handler, and increment the counter to trigger the CVM application event condition eventually. Thus, using malicious interrupt injection, the hypervisor successfully tricks the CVM into believing an event occurred when, in reality, it did not, breaking its execution integrity.

```
1  irqreturn_t interrupt_cnt_isr(int irq, void *dev_id) {
2      struct counter_device *ctr = dev_id;
3      struct interrupt_cnt_priv *priv = counter_priv(ctr);
4      atomic_inc(&priv->count);
5      counter_push_event(ctr, COUNTER_EVENT_CHANGE_OF_STATE, 0);
6      return IRQ_HANDLED; }
```

Listing 1.1. Linux interrupt-driven counter.

Wireless Device Boot-up. Qualcomm WCNSS chip is used in a wide range of devices with Arm cores for wireless communications and is vulnerable to the untrusted hypervisor interrupt injection attacks. Concretely, the WCNSS driver used for verified firmware loading expects interrupts to indicate when the device firmware is successfully loaded and ready to use (see Lst 1.2) [49].

```
1  irqreturn_t wcnss_ready_interrupt(int irq, void *dev) {
2     struct qcom_wcnss *wcnss = dev;
3     complete(&wcnss->start_done); /* Change to ready */
4     return IRQ_HANDLED; }
```

Listing 1.2. Ready interrupt handler in WCNSS driver.

If the hypervisor injects an interrupt to trigger this handler before the device is ready, it tricks the driver into transitioning the device state to ready. This may have undesirable effects (e.g., CVM may use an outdated or corrupted WiFi firmware image). Thus, the hypervisor may break the driver execution integrity using only malicious interrupt injection.

2.3 Threat Model and Scope

We trust the platform and all of its components, including the interrupt controller (GIC). Like previous works [58,68], we assume that the hardware integration of the devices on the platform conforms to the specification and is bug-free, i.e., devices are only accessible through the specified address ranges and can only assert their assigned interrupts. We assume that all software executing in the normal and secure worlds is untrusted. We trust the CCA hardware, assume that all trusted CCA software (i.e., the RMM and monitor) is implemented according to CCA specifications, and assume mutual distrust between CVMs. For a given CVM, we consider all software executing in CVM (i.e., operating system, device drivers and runtimes, applications) as trusted. Protecting against side-channels and microarchitectural attacks is orthogonal and we consider them out of scope.
Problem Statement. We aim to provide guarantees for interrupts from integrated devices delivered to a CVM. Concretely, we ensure that the device interrupts delivered to a CVM in the presence of an untrusted hypervisor is equivalent to interrupts that would be delivered if the hypervisor was trusted. This guarantees that fake interrupts injected by a malicious hypervisor cannot compromise CVM security.

3 Potential Solution

Our analysis of the device drivers in the Linux kernel shows that in addition to pure interrupts, the devices use two other types of event notification mechanisms: (1) only poll memory-mapped regions, or (2) use interrupts in conjunction with memory-mapped region reads. In both these cases, the interrupt handler reads or checks isolated MMIO memory before processing the interrupt. This isolated MMIO memory is not accessible to the malicious hypervisor. Therefore, if the

drivers check some state in the device MMIO memory before processing the interrupts, they can prevent the interrupt attacks from Sec 2.

If all device drivers adopt one of the two mechanisms above and the MMIO regions are isolated, they can thwart malicious interrupt injection. However, this solution is infeasible in some cases, such as the interrupt-driven counter, which only depends on an interrupt source and is device agnostic. Because of the device-agnostic nature of this counter, reading from an MMIO space is not possible, as different devices have different MMIO spaces and behaviors. Therefore, supporting this interrupt-driven counter in a CVM such that the hypervisor cannot corrupt it requires architectural guarantees for interrupt isolation. For devices where such memory-mapped reads would be possible (e.g., Qualcomm WCNSS chip), doing so may incur hardware changes to the device to write to the register.

To adopt this solution, we will need to change any device that simply raises an interrupt when an event occurs to also write to a memory-mapped region. Further, we will need to change all existing device drivers where such a check is feasible. Doing this will incur significant changes to the hardware and manual driver patching. More importantly, *even one device left unchanged or one driver left unpatched* will leave the CVM vulnerable to interrupt attacks. We observe that such invasive hardware and driver changes are not necessary if we can provide architectural guarantees for interrupt delivery. Therefore, in DEVLORE we take a principled approach to ensure isolated interrupt delivery to the CVMs.

4 Background: Arm CCA and Interrupt Management

We give a more detailed overview of Arm CCA, explain the interrupt behavior of integrated devices, and how they are virtualized for CVMs.

Arm CCA. To enable Arm CCA, the Arm ISA adds support for Realm Management Extensions (RME). RME enables computation to execute in one of four worlds: Normal, Realm, Secure, and Root world. These worlds are isolated from one another and have distinct physical address spaces (Tab 1). RME enforces this isolation through hardware mechanisms called Granule Protection Checks (GPC). GPCs look up Granule Protection Tables (GPTs), which map physical addresses to the worlds. GPTs are stored in protected root world memory, and programmed by the root world Monitor. Apart from the worlds, the Arm architecture enables privilege levels called exception lev-

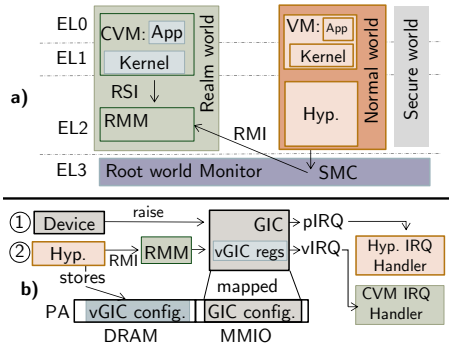

Fig. 2. (a) Arm CCA. (b) Arm interrupt architecture with CCA. 1: Device raises physical interrupt (pIRQ) to hypervisor. 2: Hypervisor invokes RMM with vGIC configuration to virtualize CVM interrupts.

els (EL) from EL0 to EL3. Typically, EL0 executes apps, EL1 operating system kernels, and EL2 hypervisor. The root world executes the Monitor at EL3, the highest privilege level. Software in EL2 can trigger a world switch by issuing a Secure Monitor Call (SMC) to the monitor (Fig 2.a).

CVMs. Confidential VMs (CVMs) execute in realm world EL0-1 under the oversight of the Realm Management Monitor (RMM) that executes in realm world EL2. The RMM does not actively manage CVM lifecycle. Instead, it validates and enforces operations requested by the normal world hypervisor. For this, it exposes the Realm Management Interface (RMI) to the hypervisor via SMC calls. The hypervisor then uses RMIs to handle CVM lifecycle (creation, expansion, deletion), and for scheduling. The RMM isolates the CVMs using stage-two MMU translation tables (S2 tables). Similar to the RMIs, the RMM exposes the Realm Service Interface (RSI) to its CVMs to handle their requests.

To/From	**Root**	**Realm**	**Normal**	**Secure**
Root	Yes	No	No	No
Realm	Yes	Yes	No	No
Normal	Yes	Yes	Yes	Yes
Secure	Yes	No	No	Yes

Table 1. Arm CCA memory access rules. The columns (From) denote the security state of the processing element that originates the memory access request. The rows (To) denote physical address spaces.

Arm Interrupt Management. The Generic Interrupt Controller (GIC) delivers physical interrupts to CPU cores based on its configuration. The hypervisor accesses the GIC through a memory-mapped configuration interface and can selectively enable/disable interrupts, set interrupt affinities, and assign interrupt priorities. With virtualization, guest VMs can only handle virtual interrupts. So, the hypervisor virtualizes physical interrupts from the GIC by programming the virtual GIC (vGIC). The vGIC provides each VM with a virtual view of the interrupt controller. To inject a virtual interrupt, the hypervisor programs up to n vGIC system registers, where n depends on the GIC implementation. If multiple interrupts are pending, the hypervisor programs the vGIC with the highest priority interrupts first. On VM entry, the vGIC delivers the programmed virtual interrupts to the guest.

CVM Interrupt Handling. Under standard CCA, the untrusted normal world hypervisor remains responsible for managing and virtualizing CVM interrupts. When a physical device interrupt arrives while a CVM is running, the CPU must not switch directly to the hypervisor, as this could leak CVM state. Instead, the interrupt first traps to the RMM, which saves the CVM state and only then forwards control to the hypervisor (see (1) in Fig 2.b). For security, CCA prevents the hypervisor from directly programming the CVM vGIC system registers. To inject virtual interrupts into the CVM, the hypervisor must invoke the RMM with a vGIC configuration, and the RMM programs the vGIC on its behalf, before entering a CVM (see (2) in Fig 2.b). In its current design, the RMM cannot verify if these virtual interrupts correspond to genuine physical interrupts. As a result, the hypervisor can inject arbitrary interrupts into a CVM.

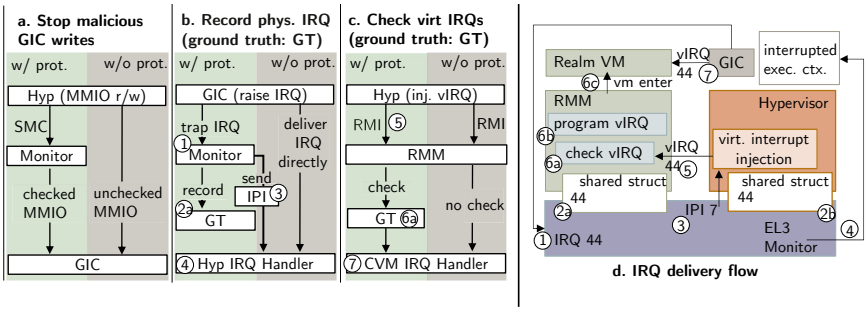

Fig. 3. DEVLORE solution overview. (a-c): DEVLORE ensures interrupt isolation by preventing malicious GIC writes, recording authentic physical interrupts as ground truth (GT), and checking hypervisor-injected virtual interrupts against GT. (d): Interrupt delivery flow. Steps (1,2ab,3,4) record the physical interrupt, Step (5) virtualizes it, and Steps (6a,b,c,7) check and deliver it. Example for IRQ 44: (1) IRQ 44 traps to the monitor; (2a,b) the monitor writes 44 to the RMM and hypervisor shared data structures; (3) it notifies the hypervisor via software interrupt 7 (IPI); (4) it returns to the interrupted application; (5) the hypervisor schedules the guest and sends the virtual interrupt configuration to the RMM; (6a,b) the RMM checks and programs vGIC system registers; (6c) it enters the CVM; (7) the GIC fires virtual interrupt 44.

5 Solution Overview

First, we provide an intuition behind DEVLORE and outline its three steps. The three steps are visualized in Fig 3a-c. Recall that on Arm platforms, untrusted integrated devices cannot fake physical interrupts from other devices as they can only assert their designated interrupt lines. The hypervisor can inject physical interrupts only by writing to the GIC directly. So, by constraining the hypervisor capability to maliciously write to the GIC, we guarantee that all registered physical device interrupts originate solely from their respective devices. Using this observation, in DEVLORE we first **prevent malicious GIC writes (Sec 5.1)** from the hypervisor to ensure physical device interrupt authenticity. However, physical interrupt authenticity alone is not sufficient. We also need to isolate the virtual interrupts that are delivered to the CVM. In CCA, the hypervisor performs interrupt management. So, the hypervisor can always inject fake virtual interrupts to the CVMs. To prevent malicious virtual interrupt injection, we only allow registered virtual interrupt injection if there was a corresponding authentic physical interrupt from the device. CVMs must *register* specific device interrupts to isolate from the hypervisor using DEVLORE. We call these interrupts the CVM *registered interrupts*. To isolate these interrupts, DEVLORE **records all registered physical device interrupts (Sec 5.2)** to create a ground truth. Then, when the hypervisor injects virtual interrupts, DEVLORE **checks the virtual interrupts (Sec 5.3)** against this ground truth and ensures that every registered interrupt delivered to the CVM corresponds to an authentic physical interrupt.

5.1 Prevent Malicious GIC Writes

Our first step is to prevent a malicious hypervisor from injecting physical interrupts by writing directly to the GIC (see Fig 3.a). DEVLORE stops the hypervisor from directly writing to the GIC memory by marking the GIC memory as root world and using the monitor to check any updates to the GIC configuration. This ensures the physical interrupt authenticity, i.e., any physical device interrupt that arrives at the cores is guaranteed to only be from the device.

5.2 Recording Physical Interrupts

After establishing the authenticity of physical device interrupts, we need a way to record these when they arrive at the cores (see Fig 3.b). By default, the GIC sends a physical interrupt to any core that is active. In CCA, all these interrupts are forwarded to the hypervisor for management. So registered interrupts must first be delivered to trusted software (e.g. CVM, RMM, monitor), recorded there, and only then forwarded to the hypervisor. We examine our design options.

Option 1: Deliver to CVM. We can consider delivering the physical interrupt directly to its CVM. However, this is not feasible as the CVM runs in a virtualized environment and cannot directly handle physical interrupts. Instead, on receiving the physical interrupt the hardware forces the CVM to exit and traps to the RMM (in EL2). Therefore, the CVM cannot record these physical interrupts.

Option 2: Deliver to RMM. Physical interrupts always trap to EL2. We can ensure that the registered physical interrupts are always delivered to the RMM which executes in realm EL2. To isolate interrupt delivery to RMM, the interrupt must only target cores executing in the realm world. However, ensuring this is not straightforward. In CCA, the hypervisor is responsible for all scheduling decisions. Thus, any core may be executing in any world at any time. To ensure reliable physical interrupt delivery to the RMM, we need to get around this dynamic scheduling behavior. For this, we can pin a core that always executes the RMM and program the GIC to only deliver the registered device interrupts to this core [23]. With this the RMM can record all physical interrupts that arrive, and then forward the interrupt to the hypervisor. While this approach allows us to record the interrupts, it wastes a core (pinning) and creates a bottleneck.

Option 3: Change the GIC Hardware. In Option 2, we require core pinning because the GIC cannot route specific interrupts exclusively to cores in the realm world. This could be avoided by extending the GIC hardware with a realm-world mechanism analogous to the secure world. The secure world can configure the GIC to deliver specific interrupts only to cores in the secure world. However, the GIC is a highly optimized piece of hardware and to propose changes to it, we need access to the hardware and tools which we do not have. More importantly, these hardware changes will not be sufficient to ensure virtual interrupt isolation (see Sec 5.3). Therefore, DEVLORE records interrupts without hardware changes.

DEVLORE **Solution: Deliver to Monitor.** By default, the GIC provides a mechanism to ensure that specific interrupts are only delivered to EL3, i.e., the monitor. We configure the GIC via the monitor so that all registered interrupts

are delivered to the monitor (Step 1 in Fig 3.b). Upon arrival, the monitor records the physical interrupt in a realm data structure (Step 2), forwards it to the hypervisor (Step 3), and returns the interrupt context (Step 4). This data structure serves as the ground truth for virtual interrupts.

5.3 Checking Virtual Interrupts

After authenticating recorded physical interrupts, we need to isolate and check the virtual interrupts (Fig 3.c). As before, we discuss the main design options.
Option 1: Move Interrupt Management to the RMM. CCA employs a delegate-but-check strategy for all CVM management. So, it lets the hypervisor perform interrupt management for CVMs. We can move the virtual interrupt management to the RMM such that the hypervisor has no opportunity to perform malicious interrupt delivery. Now, the RMM is responsible for all virtual interrupt management, including the non-registered interrupts (e.g., timers, IPIs). Moving this to the RMM is non-trivial because these interrupts are tightly coupled with hypervisor CVM scheduling and would require invasive changes.

Instead, we can move only the registered interrupt management to the RMM. To do this, the monitor records the physical interrupts and then forwards all registered interrupts to the RMM. Then, when the CVM is scheduled, the RMM can program the CVM vGIC system registers to inject the virtual interrupts. We observe that the same vGIC system registers are also used by the hypervisor to forward other non-registered virtual interrupts (e.g., timers, IPIs) to the CVM. Because the number of vGIC system registers are limited, this approach leads to contention between the RMM and the hypervisor for the same registers. More importantly, the RMM now has to reason about priority of interrupts for all the interrupts (registered and non-registered) that are to be sent to the CVM. This further complicates the design and requires the RMM to hold some interrupts to be sent to the CVM in a queue introducing queue management complexity. We conclude that this approach, while feasible, is not practical and breaks CCA delegate-but-check strategy.
DEVLORE **Solution: Check in the RMM.** By default, CCA does not allow the hypervisor to directly program the CVM vGIC system registers. Instead, every time the hypervisor schedules a CVM, it can request the RMM to program the CVM vGIC to inject the virtual interrupts (Step 5 in Fig 3.c). In DEVLORE, the RMM rigorously checks that the virtual interrupt injection is benign. To perform this check, the RMM looks up the data structure with authentic physical interrupts from the monitor (Step 6a). This way, the RMM can distinguish between the case where the hypervisor is injecting a virtual interrupt corresponding to an authentic physical interrupt versus a malicious virtual interrupt. Thus, DEVLORE ensures that the hypervisor cannot inject arbitrary virtual interrupts while preserving ordering and interrupt priorities as we explain in detail in Sec 6.2. DEVLORE design ensures that even if the interrupt management remains in the hypervisor, it cannot break CVM security (see security analysis in Sec 7).

6 Design

We detail DEVLORE interrupt isolation mechanism.
Assumptions. DEVLORE uses techniques from existing works to attach devices to a CVM, ensure runtime memory isolation, and detach devices from the CVMs [62,13,58]. For security, these prior works provide the below guarantees.

G_{id}: *Identity.* Attackers cannot forge device identities and physical memory used to access a device is fixed by the platform integration.
G_a: *Device Attach.* Devices are attached to a CVM in a trusted state (including device reset and attestation).
G_{mem}: *Memory Isolation.* Only the device and the CVM it is attached to can access the device memory.
G_d: *Device Detach.* Detaching devices from CVM will not compromise CVM security (e.g., leak residual device state).

Interrupt Registration. DEVLORE extends the device attach process of prior works to allow CVMs to enable DEVLORE interrupt isolation [58,13,62]. Generally, a single device may generate multiple physical interrupt for different functions. For example, a Mali GPU can raise a total of three interrupts, one each for power management, on page-faults, and job completion. By default, the hypervisor provides the virtual interrupt information to CVMs in the form of a mapping from `(devID,function)` to virtual interrupt number. With DEVLORE, the platform manufacturer provides a trusted mapping from `(devID,function)` to platform-specific physical interrupt number. We observe that the tuple `(devID,function)` is unique on any given platform. So, the RMM can use this tuple to securely manage interrupt registration. Concretely, we have to ensure that although the physical interrupt numbers vary across platforms and the hypervisor may choose different virtual interrupt numbers for CVMs, the hypervisor cannot abuse this to tamper with the virtual-to-physical interrupt bindings.

When a CVM wants to register an interrupt, it requests the RMM with `(devID,function)` and its virtual interrupt number. On receiving this request, the RMM first checks that the device is attached to the requesting CVM before proceeding. If the check passes, the RMM now needs the physical interrupt that corresponds to the `(devID,function)` in the CVM request. For this, the RMM looks up the trusted platform-specific mapping from `(devID,function)` to physical interrupt numbers. This look up ties the CVM-specific virtual interrupt number to the platform-specific physical interrupt number. The RMM uses this physical interrupt number to invoke the monitor to register the interrupt for DEVLORE isolation. Once registered, the RMM stores the virtual interrupt number to facilitate the RMM checks on the virtual interrupt as explained in Sec 6.2. Finally, the RMM adds the registration information (i.e., `(devID,function)`, physical and virtual interrupt number) to the CCA attestation report.

On device detach, the RMM checks if interrupt isolation was enabled for the device. If so, the RMM invokes the monitor to update the GIC configuration to de-register the interrupt isolation.

6.1 Ensuring Interrupt Isolation with DEVLORE

For registered device interrupts, DEVLORE guarantees interrupt isolation (configuration, delivery, acknowledgment) under an untrusted hypervisor. Recall from Sec 5 that DEVLORE proceeds in 3 steps: establishing authenticated physical interrupts, recording the authenticated physical interrupts as ground truth, and checking virtual interrupts against the ground truth.

Authenticated Physical Interrupts. In Arm CCA, the hypervisor manages the GIC by default and can maliciously modify its configuration to inject interrupts. To circumvent this threat, DEVLORE moves the GIC configuration into the root world and protects it using the monitor checks. To maintain hypervisor functionality, DEVLORE introduces a GIC configuration interface for the hypervisor to use in the monitor (Fig 3.a). Concretely, any hypervisor access to the GIC configuration raises an access fault. In the hypervisors fault handler, DEVLORE invokes the monitor to perform the operation. On each such request, the monitor checks that the hypervisor is not trying to reconfigure the GIC for a registered interrupt. If the check passes, the monitor allows the configuration request through. With this, DEVLORE can ensure that all registered physical interrupts that the GIC injects are authentic, i.e., from their respective devices. However, physical interrupt authenticity alone does not guarantee interrupt isolation; DEVLORE must also verify virtual interrupt delivery.

Recording Registered Interrupts. Recall that once DEVLORE establishes physical interrupt authenticity, it uses this as the ground truth to check that the hypervisor benignly virtualizes them for the CVM. To create the ground truth, DEVLORE has to record all the registered physical interrupts that the GIC fires. For this, it configures the GIC such that all registered device interrupts trap to the monitor. This lets the monitor record their arrival. In DEVLORE, the monitor stores this information in the RMM-accessible realm memory and notifies the hypervisor that this interrupt arrived by writing to normal world hypervisor-accessible memory. Now, all that remains is to ensure that DEVLORE checks this information to guarantee benign virtual interrupt delivery.

Delegate-but-check Virtual Interrupt Management. As in CCA, the hypervisor still manages interrupts for the CVMs in DEVLORE. So, the hypervisor should virtualize the device interrupts that the monitor trapped for the CVMs. To enable this, DEVLORE changes the hypervisor to first check if there is any pending interrupt from the monitor every time the hypervisor executes. If there are pending interrupts, the hypervisor injects the corresponding virtual interrupts into the CVM. According to CCA specifications, the hypervisor sends the vGIC configuration to the RMM. With DEVLORE, the RMM checks this configuration and then programs the vGIC to fire the virtual interrupts to the CVM, on CVM entry.

6.2 Checking Virtual Interrupts

Next, we explain checks that the RMM has to perform with the monitor cooperation. We start with simple checks that are insufficient and iterate over them to build towards our full set of checks. Fig 4 shows each check with an example.

Check #1: Total Count. To begin with, assume that the RMM performs a simple count-based check when the hypervisor injects virtual interrupts into a CVM. Concretely, the RMM computes the expected number of pending interrupts per CVM from the monitor recorded physical interrupts and decrements this count on each virtual injection. This bounds injections to neither too many nor too few (see Fig 4 #1), but does not prevent the hypervisor from injecting an interrupt number the device has never raised.

Check #2: Origination from a Physical Interrupt. To stop the hypervisor from arbitrarily injecting virtual interrupts, the RMM should tie the authenticity of the physical interrupts to the virtual interrupts. For this, the RMM requires the monitor to write the registered physical interrupt numbers that arrive to a list in realm memory. Then, when the hypervisor injects the virtual interrupt, the RMM ensures that for each virtual interrupt programmed into the vGIC there is a corresponding entry in the monitor list before forwarding it. In CCA, when the hypervisor injects the virtual interrupt via the RMM, it can send n (where n depends on the GIC implementation) distinct interrupts to the vGIC (see background Sec 4). So, for each injection request, the RMM checks that all of the interrupts in the request are authentic. Tying the virtual interrupt to physical interrupts stops the hypervisor from arbitrarily injecting other interrupts but cannot ensure interrupt ordering (see Fig 4 #2).

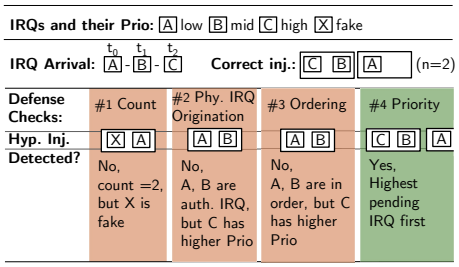

Fig. 4. Device interrupts A,B,C are registered by a CVM for DEVLORE secure interrupt delivery. vGIC window size n=2. DEVLORE achieves interrupt isolation by enforcing checks #2, #3, #4.

Check #3: Ordering. DEVLORE needs to store an ordered list of physical interrupts when they arrive at the monitor to guarantee that the hypervisor cannot reorder them. Then, the RMM can use this ordered list to check the virtual interrupts that the hypervisor wants to inject. We observe that because the hypervisor can inject n distinct interrupts at once to the CVM, DEVLORE does not need to guarantee ordering for these n distinct interrupts. If the RMM programs n distinct interrupts into the vGIC, they become pending together on CVM entry. So the RMM uses the monitor ordered list to check ordering with a window size of n. This prevents the hypervisor from reordering the interrupts that are outside this window (Fig 4 #3). However, this alone cannot achieve DEVLORE interrupt isolation. During benign operation, the hypervisor injects virtual interrupts based on the priority of all the pending interrupts. Thus, time-based ordering cannot preserve the priority.
Check #4: Priorities. To check virtual interrupt priorities, DEVLORE requires the CVM to assign priorities to its registered device interrupts during device attach. The RMM uses this information to create per-CVM ordered priority lists and checks the virtual interrupt injections from the hypervisor against them,

so that the hypervisor always injects the highest-priority pending interrupts (Fig 4 #4). This check is sufficient to guarantee that the interrupt behavior observed by a CVM is equivalent to some benign execution. The hypervisor cannot inject or reorder interrupts in a way the CVM would not already tolerate. The guarantee holds even if the hypervisor maliciously delays CVM scheduling until a high-priority interrupt arrives to effectively never inject a lower-priority interrupt. Since a benign hypervisor provides no scheduling guarantees to a CVM, any malicious delay is indistinguishable from some benign scheduling (Fig 5). Together, checks #2, #3, and #4 achieve DEVLORE interrupt isolation.

6.3 Acknowledging Interrupts

The above DEVLORE checks ensure isolated interrupt configuration and delivery. Despite these checks, the hypervisor still needs to acknowledge interrupts by writing to the memory-mapped registers for functionality. DEVLORE marks the memory-mapped register for acknowledgment as root memory, thus revoking the hypervisor access and forcing SMC. When the hypervisor tries to acknowledge any interrupt using a monitor call (SMC), the monitor first checks if this interrupt is isolated by DEVLORE. If not, the monitor performs the acknowledgment. On the other hand, for interrupts that DEVLORE isolates, it has to ensure that they are acknowledged appropriately for preserving the hypervisor functionality (Appx B.2).

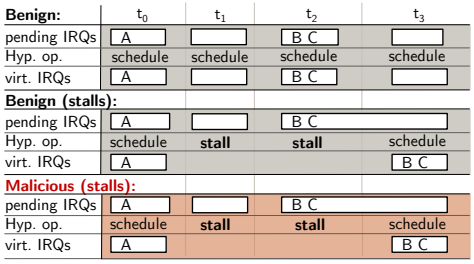

Fig. 5. Top: Benign CVM scheduling, Middle: benign with stalls, Bottom: malicious stalls but indistinguishable from middle trace, stalls were already expected. Hyp. op: Hypervisor operation (schedule or stall): A,B,C: interrupts

Put together, DEVLORE mechanisms for blocking malicious physical injection to establish interrupt authenticity, recording authentic physical interrupts to create the ground truth, and rigorous checking of virtual interrupt injections against the ground truth ensure interrupt isolation for CVMs.

7 Security Analysis

We argue why DEVLORE is secure against different adversaries. Prior works provide the guarantees (G_{id}, G_a, G_{mem}, G_d) for device lifecycle management and memory isolation. In DEVLORE we rely on these guarantees for security, as without them DEVLORE interrupt protection will not hold. While we assume that device lifecycle management and memory isolation are secure, for completeness we explain the attacks mitigated by the guarantees in Appx A. Next, we explain how the attacks on interrupt isolation are mitigated in DEVLORE.

GIC Configuration and Interrupt Injection Attacks. The hypervisor may try to reprogram the GIC to bypass monitor traps, alter interrupt priorities, or acknowledge interrupts directly, all of which would undermine DEVLORE interrupt security guarantees. DEVLORE protects the GIC configuration in root memory and checks all hypervisor accesses to the configuration stopping such attempts. The hypervisor can mount Iago attacks on DEVLORE interfaces by invoking the monitor GIC configuration interface (see Fig 3.a) with arbitrary addresses in the root world to read or write from [24]. DEVLORE stops such attempts by checking that the address range the hypervisor requests to operate on using this interface is in the GIC configuration space. Besides the memory-mapped GIC configuration, the Arm architecture contains system registers for GIC configuration. The hypervisor can try to use these registers to compromise DEVLORE interrupt isolation. However, DEVLORE traps all registered interrupts to EL3. On Arm, the hypervisor in normal world EL2 cannot use the system registers to configure these EL3 interrupts, thus stopping such attacks.

The hypervisor can stall CVM scheduling (Fig 5) and wait for a high priority interrupt to arrive. It can use this malicious scheduling to not inject lower priority interrupts that had arrived earlier. However, we observe that this interrupt injection behavior could also occur during CVM execution with a benign hypervisor as hypervisors do not provide scheduling guarantees for CVMs. Thus, this malicious stalling of the CVM is safe and does not lead to an interrupt injection behavior that diverges from what the CVM normally expects. The hypervisor can try to inject an interrupt multiple times (e.g., 16 times) with one vGIC programming request. This is not possible, as the vGIC injects any given interrupt only once on every realm entry. Therefore, while the vGIC can be programmed to fire n distinct interrupts at once it cannot be programmed to fire the same interrupt multiple times at once. Finally, the hypervisor cannot interrupt the RMM mid-check, as the RMM masks all interrupts during execution.

An attacker-controlled co-resident CVM can try to inject interrupts to other CVMs. This is not possible as each CVM can only program its local vGIC and cannot access vGIC of other CVMs, because the RMM does not expose any APIs for this. An attacker can use a device to fake victim device interrupts. But the hardware integration of the device on the platform is trusted and is part of attestation report. Each device interrupt line is unique and tied to the hardware: software attackers cannot use devices to fake other device interrupts. In Sec 9.1, we perform a comprehensive security evaluation on Arm FVP to show that DEVLORE mitigates the attacks discussed here.

8 Implementation

We prototype DEVLORE on the FVP and change 3054 LoC in the trusted firmware v2.8 (TF-A) which we use as the monitor [8], 1188 LoC in RMM v0.2.0 [15], and 2706 LoC in the Linux kernel v6.2.0 (cca-full/rfc-v1) [54] which we use as the host and guest OS (guest: 60 LoC, host: 2646 LoC). The Arm reference implementation uses KVM in the Linux kernel and a virtual machine

manager (VMM) called kvmtool to build and deploy CVMs which we reuse. We change VMM with 590 LoC to add device support. Our modifications require no application or device-driver changes. We introduce 6 new interfaces for interrupt protection in monitor and RMM. The full API list is shown in Appx B.1.

Changes to Platform Boot. Like previous works, we create an additional GPT during boot for the device-side GPT (GPT_d) to allow devices access to CVM memory (Appx B.1). To ensure memory isolation, we move the SMMU configuration and memory to the root world [62,68]. We also move GIC configuration to the root world during boot. Further, we modify the RMM to store platform device information, i.e. interrupt-to-device mappings. During runtime, the RMM uses this information to check CVM interrupt registrations.

Interrupt Configuration. To isolate physical interrupts, we move the GIC memory-mapped configuration space to root. So, all hypervisor GIC configurations cause granule protection faults (GPF). We modify the GPF handler to invoke the monitor (`smc_gic_config`) to check and perform the configuration. We implement an SMC (`smc_prot_int`) for the RMM to invoke for registering interrupts for isolation with DEVLORE to ensure that they always trap to EL3. We first configure the GIC to trap all Group 0 interrupts to the monitor in EL3 and then program the GIC entry for the registered interrupts as belonging to Group 0. This ensures that these interrupts always trap to EL3 (Step 1, Fig 3.d).

Interrupt Delivery. We allocate a shared data structure for the RMM (realm) and hypervisor (normal) to track pending protected interrupts. The monitor populates these for each registered interrupt trap (Steps 2a–b in Fig 3.d) and notifies KVM using a software interrupt (Step 3–4). KVM injects pending interrupts into CVM (Step 5) by requesting vGIC system register programming, which the RMM checks and forwards to the CVM (Step 6).

Interrupt Acknowledgment. If possible, (i.e., without causing an interrupt storm), the monitor acknowledges any registered interrupts immediately before invoking KVM (Steps 1-3). Otherwise, after the virtual interrupt fires in the guest (Step 7) and the guest handler completes, the RMM traps the guest and invokes the monitor (`smc_ack_int`) to acknowledge the corresponding physical interrupt (Sec 6.3 and Appx B.2).

9 Evaluation

We evaluate DEVLORE with two complementary prototypes: a functionality prototype on the Arm FVP and a performance prototype based on OpenCCA [20] and a Rockchip RK3588 Armv8.2 board. The functionality prototype validates DEVLORE compatibility with Armv9 and Arm CCA, but is limited to software simulation; the performance prototype estimates DEVLORE overhead on currently available Armv8 hardware. As no public CCA hardware is available today, our evaluation, like previous works [62,68,77,58,20] provides a best-effort estimate of DEVLORE functionality and performance on future CCA systems. We defer FVP functionality and compatibility results to Appx C and focus the

main text on security evaluation, quantitative performance overheads, and end-to-end case studies.

Across all experiments, we compare a baseline configuration without interrupt protection (**B**) against a configuration with interrupt protection and DEVLORE changes (**I**). In both configurations, we boot the platform, launch a realm world CVM, attach integrated devices, and measure interrupt-related behavior.

9.1 Security Evaluation on Arm FVP

We validate that DEVLORE enforces correct interrupt isolation with 4 checks. These checks instantiate adversarial behaviors from Sec 7 and empirically validate that our implementation enforces DEVLORE security guarantees.

(a) Benign Interrupts. We run device workloads (GPU, UART, Mouse & Keyboard, LED & Button) that generate up to 10k interrupts and verify that all these device-raised interrupts reach the CVM correctly, as expected.

(b) Malicious Virtual Interrupts. We repeat the benign workloads but additionally modify the hypervisor to inject 10k virtual interrupts that have no corresponding physical interrupt of a protected device. For this, we create a kernel module that invokes `kvm_vgic_inject_irq` on the KVM instance of the CVM (174 LoC). We report that DEVLORE rejects all injected malicious interrupts in the RMM and only delivers virtual interrupts to the CVM that are backed by a recorded physical interrupt, as expected.

(c) Malicious Physical Interrupts. We repeat the benign workloads but additionally modify the hypervisor to write to the GIC configuration to raise 10k physical interrupts for a protected device (Set-Pending register, `GICD_ISPENDR` on the Distributor MMIO of the GIC, 168 LoC). We report that these malicious writes trap to the monitor and are blocked from reaching the GIC, as expected.

(d) Malicious Memory Accesses. We configure the FVP SMMU test engine [12] to issue memory accesses to the MMIO region of the protected device, including the registers for interrupt acknowledgement. The test engine is not attached to the CVM, so all accesses are unauthorized. We report that our implementation blocks these accesses with GPT violations. In addition, we modify the hypervisor to read from the MMIO region of the protected device directly, and report that these accesses are likewise rejected with GPT faults (176 LoC).

9.2 Performance Prototype

We estimate DEVLORE overheads on an Armv8 board with OpenCCA [20]. We apply DEVLORE on top of OpenCCA tree (TF-A v2.11, RMM v0.5, Linux v6.12) [19] and run on a Rock 5B RK3588 SoC, 16 GB of RAM, and Mali G610. We boot the platform on 4 Cortex-A55 cores at 1.8 GHz (`userspace` governor).

Impact on Boot and Normal Operations. DEVLORE adds 2.4% overhead (221 ms) to the platform boot. This increases total boot time from 8.9 s to 9.1 s. Most of this overhead comes from SMMU setup for memory isolation (+177.7 ms), while DEVLORE's GIC configuration (i.e., protecting the GIC in

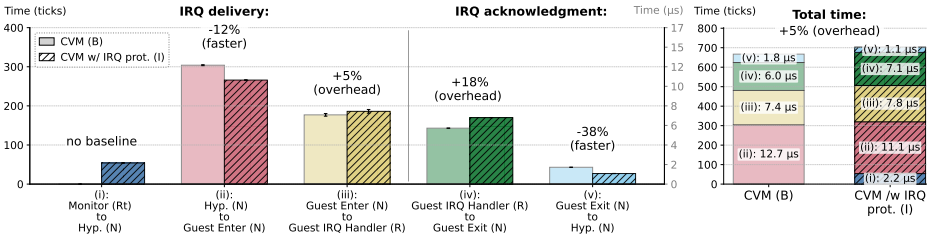

Fig. 6. GPU interrupt-flow breakdown on the Arm board under DEVLORE. Left: stage times for (i) monitor IRQ trap and hypervisor notification (Steps 1-4 in design Fig 3), (ii) hypervisor schedules guest (Step 5 in Fig 3), (iii) guest entry, RMM checks, IRQ handler (Steps 6-7 in Fig 3), (iv) guest exits back to hypervisor, (v) end-of-interrupt acknowledgement in hypervisor. Bars show baseline CVM **B** and CVM with DEVLORE IRQ protection (**I**); percentages indicate overhead of **I** over **B**. No baseline exists for (i), since without DEVLORE the IRQ directly reaches (ii). Right: total interrupt-handling time across (i)-(v). N: Normal, Rt: Root, R: Realm. 1 tick = 41.67 ns. Lower is better.

root world) adds only 0.1 ms during BL31. During Linux boot with **I**, we observe 1816 traps (445 reads, 1371 writes) for protected GIC configuration; each trap takes 2.5 µs, which is negligible relative to the 4.8 s kernel boot time. To evaluate non-interrupt workloads, we run sysbench [41] on a normal world VM and a CVM under **B** and **I**. As expected, DEVLORE has no measurable impact on workloads that do not involve device interrupts (details in Appx C.5).

Interrupt Lifecycle Breakdown. We measure the cost of a single protected interrupt, from when the device raises a physical interrupt until the hypervisor processes the end-of-interrupt acknowledgement, and break this path into interrupt delivery (ⓘ–ⓘⓘⓘ) and acknowledgement (ⓘⓥ–ⓥ) in Fig 6. We use the Mali G610 GPU on the Arm board, which generates frequent interrupts and requires a complex driver stack [10]. Averaged across 1000 interrupts, the end-to-end interrupt time in **B** is 669 ticks (27.9 μs, 95% CI: $\pm$0.2 μs), while **I** increases this to 703 ticks (29.3 μs, 95% CI: $\pm$0.2 μs). This corresponds to a 5% overhead. The stage-wise breakdown in Fig 6 shows that this overhead comes from recording in the monitor ⓘ, during authenticity checks in the RMM on guest entry ⓘⓘⓘ, and guest exit for end-of-interrupt acknowledgement in the monitor ⓘⓥ. This is expected because DEVLORE adds bookkeeping and checks in these stages. In contrast, DEVLORE is faster when the hypervisor schedules the guest with the new virtual interrupt ⓘⓘ, and when it completes post end-of-interrupt bookkeeping ⓥ. This is because, with DEVLORE, the monitor manages the physical interrupt state (disable on trap, re-enable on acknowledgment) rather than leaving it to the hypervisor.

Sustained Interrupt Load. We examine whether these overheads accumulate under sustained device activity. For this, we run an interrupt stress test using both the GPU and UART device. We attach each device to their own CVM and configure the device to generate a high frequency of interrupts. For GPU, we use Mali Kuft Midgard benchmark suite [10]. For UART, we modify the Synopsys

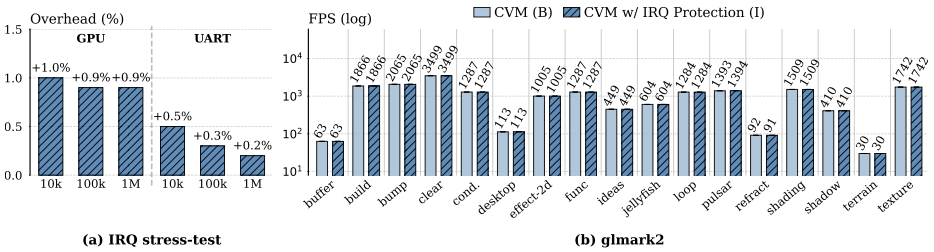

Fig. 7. Arm-board evaluation. Left: GPU and UART IRQ stress test; Y-axis shows overhead of **I** over **B**. Lower is better. Right: glmark2 Frames/sec (GPU) for **B** and **I** in log scale. Higher is better.

DesignWare 8250 driver to run in internal loopback mode. This way, it raises a physical interrupt on each write to the transmit register. Under continuous load, the overhead stays within 1% (stdev below 1.2% of mean). This indicates that per-interrupt costs do not accumulate at scale (see Fig 7, left).

9.3 Case Studies

We validate DEVLORE with real-world apps on the FVP and board.

FVP. On FVP, we run three interactive UART-based applications inside a CVM: a terminal browser, an IRC client and a text editor. Because interactive actions are difficult to reproduce consistently across runs, we use these experiments only as qualitative validation. This demonstrates DEVLORE compatibility and usability compared to executing them in the realm world without interrupt protection. We provide videos for both **B** and **I** in [2,3]; details in Appx C.3.

Board. We evaluate our design under realistic GPU workloads using glmark2 [35], a suite of userspace workloads that capture the mixed compute, memory, and driver interactions of typical integrated GPU apps. The workloads generate 2.5k – 32k GPU interrupts (configuration in Appx C.4). Enabling DEVLORE interrupt protection across all 17 benchmarks has no measurable impact on the end-to-end throughput (see Fig 7, right). Frame rates for CVM with and without DEVLORE are almost identical. The geometric mean slowdown across all benchmarks is 0.06%. So, DEVLORE interrupt protections does not add measurable overhead beyond the baseline realm cost, indicating no end-to-end performance overheads for CVMs.

10 Related Work

Arm CCA and Device Protection. Prior works on Arm CCA [68,62,58] [23,77,47,34,78,74,79,26,25], offer better abstractions [79,25,26,23] or new security guarantees [78,74]. Cage and Strongbox build support for CCA CVMs and TrustZone trusted apps respectively, to securely compute on integrated Arm

GPUs—without trusting the GPU driver and runtime to reduce the TCB [68,28]. DEVLORE instead keeps drivers and runtimes in the CVM TCB and, like Portal, supports a wide array of integrated devices in CVMs [58]. Similarly, Acai connects TEE-enabled PCIe accelerators to CVMs [62], and Arm hardware extension for Device Assignment (RME-DA) allows such accelerators to directly access CVM memory [13]. Neither applies to integrated devices, but DEVLORE reuses techniques from Portal, Acai, and RME-DA for device lifecycle and memory isolation, and extends them with interrupt isolation as its new contribution. TrustZone-based works enable protected device access [36,64,63,45,75,46,44,37], but do not reason about VMs or virtual interrupt delivery. DEVLORE targets CCA and reasons about multiplexing interrupts across CVMs. Ongoing verification of CCA firmware [47,34] can be extended to DEVLORE.

Integrated Device Protection in Other Architectures. On RISC-V, prior works attach devices to enclaves with hardware changes [18,61]. DEVLORE uses CCA's analogous hardware-based isolation. Others let trusted hardware create different views of memory like GPTs in CCA [42,33,21] or hardware-domains [73]. Intel SGX-based trusted I/O [70,48,32,40,29] and prior enclave defenses against malicious non-device interrupts [27,22] target userspace enclaves, whereas DEVLORE isolates device interrupts for CVMs.

Interrupt Attacks. Side-channel attacks exploit unprotected timers, page-faults, IPIs [65,55,66,72,38,67,69,56], and fake CPU exceptions, system calls, and exploit race conditions [60,59,43] These interrupts are raised by the CPU (e.g., page-faults, system calls, IPIs), whereas DEVLORE focuses on device interrupts. Prior works have studied how statistical information on integrated device interrupts (e.g., interrupt timing, # interrupts) can compromise confidentiality [30].

Interrupt Isolation on Arm. For *interrupt configuration*, prior works on TrustZone [36,28,44] isolate integrated device interrupts by marking the GIC configuration as secure world. However, in DEVLORE, we do not trust the secure world which can directly program the GIC to maliciously fire interrupts. So, departing from these works and for security in the CCA setting, DEVLORE marks GIC configuration as root. On the other hand, for *interrupt delivery*, like the above prior works, DEVLORE also marks interrupts as belonging to Group 0 (i.e., secure world) to trap them in EL3 [28,36,44]. Recent CCA works need interrupt isolation for one interrupt ID (e.g., GPU or timer) and resort to the same mechanisms as works on TrustZone [68,76] i.e., marking the interrupt as secure world Group. Thus, they have to trust the secure world, which DEVLORE avoids. Other works on Arm CCA have proposed delegating interrupt management to the RMM for performance when defending against side-channel adversaries [23]. In their design, they pin CVMs to cores and to reduce the number of context switches to the hypervisor by using the RMM to manage the CVM timer interrupts. This is equivalent in security to the existing CCA threat model that does not block malicious interrupt injection from the hypervisor. As the hypervisor still manages all other interrupts, their design would need DEVLORE interrupt isolation. DEVLORE is not specific to one interrupt ID, instead isolates device in-

terrupts across multiple devices and CVMs thus addressing the gap in the CCA threat model.

TDISP, Intel TDX, and AMD SEV-SNP. TDISP is a PCIe feature that allows direct attachment of TEE-enabled devices to CVMs and does not reason about device interrupts. TDX-Connect and SEV-TIO are TDISP-compliant implementations specific to Intel and AMD, respectively. They protect CVM-bound interrupts by deploying trusted firmware and software to check virtual interrupt injections [39,6,7]. But, like CCA, they only allow or deny all device interrupts without granular interrupt isolation. DEVLORE insights can be applied to these other architectures, but with careful analysis.

11 Conclusion

In this paper, we present DEVLORE, a device interrupt isolation mechanism for CVMs. Our delegate-but-check strategy for interrupt isolation, allows the hypervisor to manage the GIC but strictly monitors the interrupt life cycle (configuration, delivery, acknowledgment). Our evaluation demonstrates this on the Arm FVP and an Arm Rock 5B Board.

References

1. DEVLORE Extended Version (2026)
2. DEVLORE: Recording of Case Studies CVM Baseline (B) (2026)
3. DEVLORE: Recording of Case Studies CVM with Interrupt Protection (I) (2026)
4. DEVLORE Sourcecode (2026), https://sectrs-devlore.github.io/
5. Allegretta, C., Schulenberg, B.: GNU nano text editor (1999-2025)
6. AMD: AMD SEV-SNP: Strengthening VM Isolation with Integrity protection and more (2020)
7. AMD: Linux SVSM (Secure VM Service Module) (2022-2024)
8. Arm: Arm Trusted Firmware-A Documentation (2013-2025)
9. Arm: NoMali: A simple Mali 6xx/7xx register interface model (2015-2016)
10. Arm: Bifrost: Open Source Kernel Drivers (2016-2025)
11. Arm: Fast Models Fixed Virtual Platforms (FVP), Reference Guide (2017-2023)
12. Arm: SMMUv3TestEngine, FVP Reference Guide (2017–2024)
13. Arm: Arm Realm Management Extension (RME) System Architecture (2021-2023)
14. Arm: Arm Confidential Compute Architecture (ARM-CCA) (2022-2025)
15. Arm: Trusted Firmware Implementation of the RMM (2022-2025)
16. Arm: Powering Microsoft's Azure Cobalt 200 with Arm Neoverse CSS V3: The next generation of Arm-based compute for the AI era (2025)
17. Azure: Announcing Cobalt 200: Azure's next cloud-native CPU (2025)
18. Bahmani, R., Brasser, F., Dessouky, G., Jauernig, P., Klimmek, M., Sadeghi, A.R., Stapf, E.: CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In: USENIX Security (2021)
19. Bertschi, A., Shinde, S.: OpenCCA Manifest commit 07fcc51 (2025)
20. Bertschi, A., Shinde, S.: OpenCCA: An Open Framework to Enable Arm CCA Research. In: SysTEX (2025)

21. Brasser, F., Gens, D., Jauernig, P., Sadeghi, A.R., Stapf, E.: Sanctuary: ARMing TrustZone with User-space Enclaves. In: NDSS (2019)
22. Busi, M., Noorman, J., Van Bulck, J., Galletta, L., Degano, P., Mühlberg, J.T., Piessens, F.: Securing interruptible enclaved execution on small microprocessors. ACM Trans. Program. Lang. Syst. **43**(3) (2021)
23. Castes, C., Baumann, A.: Sharing is Leaking: Blocking transient-execution attacks with core-gapped confidential VMs. In: ASPLOS (2024)
24. Checkoway, S., Shacham, H.: Iago attacks: Why the system call API is a bad untrusted RPC interface. In: ASPLOS (2013)
25. Chen, J., Mi, Z., Xia, Y., Guan, H., Chen, H.: CPC: Flexible, Secure, and Efficient CVM Maintenance with Confidential Procedure Calls. In: USENIX ATC (2024)
26. Chen, J., Zhou, Q., Yan, X., Jiang, N., Jia, X., Zhang, W.: CubeVisor: A Multi-realm Architecture Design for Running VM with ARM CCA. In: ACSAC (2024)
27. De Clercq, R., Piessens, F., Schellekens, D., Verbauwhede, I.: Secure interrupts on low-end microcontrollers. In: IEEE ASAP (2014)
28. Deng, Y., Wang, C., Yu, S., Liu, S., Ning, Z., Leach, K., Li, J., Yan, S., He, Z., Cao, J., Zhang, F.: StrongBox: A GPU TEE on Arm Endpoints. In: ACM CCS (2022)
29. Dhar, A., Puddu, I., Kostiainen, K., Capkun, S.: ProximiTEE: Hardened SGX Attestation by Proximity Verification. In: CODASPY (2020)
30. Diao, W., Liu, X., Li, Z., Zhang, K.: No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis. In: IEEE S&P (2016)
31. Dickey, T.: LYNX - The Text Web-Browser (1997-2025)
32. Eskandarian, S., Cogan, J., Birnbaum, S., Brandon, P.C.W., Franke, D., Fraser, F., Garcia, G., Gong, E., Nguyen, H.T., Sethi, T.K., Subbiah, V., Backes, M., Pellegrino, G., Boneh, D.: Fidelius: Protecting User Secrets from Compromised Browsers. In: IEEE S&P (2019)
33. Feng, E., Feng, D., Du, D., Xia, Y., Zheng, W., Zhao, S., Chen, H.: sIOPMP: Scalable and Efficient I/O Protection for TEEs. In: ASPLOS (2024)
34. Fox, A.C., Stockwell, G., Xiong, S., Becker, H., Mulligan, D.P., Petri, G., Chong, N.: A Verification Methodology for the Arm® Confidential Computing Architecture: From a Secure Specification to Safe Implementations. Proceedings ACM on Programming Languages **7**(OOPSLA1) (2023)
35. Frantzis, A., Barker, J., Smith, B.: glmark2, v2023.01 (2010-2025)
36. Groschupp, F., Kuhne, M., Schneider, M., Puddu, I., Shinde, S., Capkun, S.: It's TEEtime: Secure Interrupt Isolation for Normal-world Enclaves on Arm. IACR TCHES (2026)
37. Han, S.K., Jang, J.: MyTEE: Own the Trusted Execution Environment on Embedded Devices. In: NDSS (2023)
38. He, W., Zhang, W., Das, S., Liu, Y.: SGXlinger: A New Side-channel Attack Vector Based on Interrupt Latency Against Enclave Execution. In: IEEE ICCD (2018)
39. Intel: Intel Trust Domain Extensions (Intel TDX) (2023-2025)
40. Jang, Y., Keem, S.: SGX-USB: Secure USB I/O Path for Secure Enclaves. HICSS (2024)
41. Kopytov, A.: Sysbench, CPU: 200k Primes, Memory: 1 GB (64k Blocks), Threads: 8, Mutex: 8 (2006-2025)
42. Ku, S.C.: What's in RISC-V IOPMP. RISC-V Forum: Security (2021)
43. Lee, Y., Min, C., Lee, B.: ExpRace: Exploiting Kernel Races through Raising Interrupts. In: USENIX Security (2021)
44. Lentz, M., Sen, R., Druschel, P., Bhattacharjee, B.: SeCloak: ARM TrustZone-Based Mobile Peripheral Control. In: MobiSys (2018)

45. Li, W., Luo, S., Sun, Z., Xia, Y., Lu, L., Chen, H., Zang, B., Guan, H.: VButton: Practical Attestation of User-Driven Operations in Mobile Apps. In: MobiSys (2018)
46. Li, W., Ma, M., Han, J., Xia, Y., Zang, B., Chu, C.K., Li, T.: Building Trusted Path on Untrusted Device Drivers for Mobile Devices. In: APSys (2014)
47. Li, X., Li, X., Dall, C., Gu, R., Nieh, J., Sait, Y., Stockwell, G.: Design and Verification of the Arm Confidential Compute Architecture. In: OSDI (2022)
48. Liang, H., Li, M., Chen, Y., Jiang, L., Xie, Z., Yang, T.: Establishing Trusted I/O Paths for SGX Client Systems With Aurora. IEEE Transactions on Information Forensics and Security (2020)
49. Linux: Linux WCNSS driver (2016-2025)
50. Machado, F.: Simple cross-platform Console IRC Client (2011-2018)
51. Microsoft: Microsoft and NVIDIA partnership continues to deliver on the promise of AI (2024)
52. Nvidia: NVIDIA Puts Grace Blackwell on Every Desk and at Every AI Developer's Fingertips (2025)
53. Paul Vu, T.: Method and Apparatus for Controlling Interrupt Storms (US Patent 7,120,717 B2, Oct 2006)
54. Poulose, S.: [RFC] Support for Arm CCA VMs on Linux (2023)
55. Puddu, I., Schneider, M., Haller, M., Capkun, S.: Frontal attack: Leaking Control-Flow in SGX via the CPU frontend. In: USENIX Security (2021)
56. Rauscher, F., Gruss, D.: Cross-Core Interrupt Detection: Exploiting User and Virtualized IPIs. In: ACM CCS (2024)
57. Rosdahl, J.: miniircd: A (very) simple Internet Relay Chat server (2011-2025)
58. Sang, F., Lee, J., Zhang, X., Kim, T.: PORTAL: Fast and Secure Device Access with Arm CCA for Modern Arm Mobile System-on-Chips (SoCs). In: IEEE S&P (2025)
59. Schlüter, B., Sridhara, S., Bertschi, A., Shinde, S.: WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP. In: IEEE S&P (2024)
60. Schlüter, B., Sridhara, S., Kuhne, M., Bertschi, A., Shinde, S.: Heckler: Breaking Confidential VMs with Malicious Interrupts. In: USENIX Security (2024)
61. Schneider, M., Dhar, A., Puddu, I., Kostiainen, K., Čapkun, S.: Composite Enclaves: Towards Disaggregated Trusted Execution. IACR TCHES (2022)
62. Sridhara, S., Bertschi, A., Schlüter, B., Kuhne, M., Aliberti, A., Shinde, S.: ACAI: Protecting Accelerator Execution with Arm Confidential Computing Architecture. In: USENIX Security (2024)
63. Sun, H., Sun, K., Wang, Y., Jing, J.: TrustOTP: Transforming Smartphones into Secure One-Time Password Tokens. In: ACM CCS (2015)
64. Sun, H., Sun, K., Wang, Y., Jing, J., Wang, H.: TrustICE: Hardware-Assisted Isolated Computing Environments on Mobile Devices. In: IEEE/IFIP DSN (2015)
65. Van Bulck, J., Piessens, F., Strackx, R.: Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In: ACM CCS (2018)
66. Van Bulck, J., Weichbrodt, N., Kapitza, R., Piessens, F., Strackx, R.: Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In: USENIX Security (2017)
67. Van Schaik, S., Seto, A., Yurek, T., Batori, A., AlBassam, B., Genkin, D., Miller, A., Ronen, E., Yarom, Y., Garman, C.: SoK: SGX.Fail: How Stuff Gets eXposed. In: IEEE S&P (2024)
68. Wang, C., Zhang, F., Deng, Y., Leach, K., Cao, J., Ning, Z., Yan, S., He, Z.: CAGE: Complementing Arm CCA with GPU Extensions. In: NDSS (2024)

69. Wang, W., Chen, G., Pan, X., Zhang, Y., Wang, X., Bindschaedler, V., Tang, H., Gunter, C.A.: Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In: ACM CCS (2017)
70. Weiser, S., Werner, M.: SGXIO: Generic Trusted I/O Path for Intel SGX. In: CODASPY (2017)
71. Wikipedia: Today's featured article (2025)
72. Xu, Y., Cui, W., Peinado, M.: Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In: IEEE S&P (2015)
73. Yao, Z., Seyed Talebi, S.M., Chen, M., Amiri Sani, A., Anderson, T.: Minimizing a Smartphone's TCB for Security-Critical Programs with Exclusively-Used, Physically-Isolated, Statically-Partitioned Hardware. In: MobiSys (2023)
74. Ye, Z., Zhou, L., Zhang, F., Jin, W., Ning, Z., Hu, Y., Qin, Z.: FortifyPatch: Towards Tamper-Resistant Live Patching in Linux-Based Hypervisor. In: ACM ISSTA (2024)
75. Ying, K., Ahlawat, A., Alsharifi, B., Jiang, Y., Thavai, P., Du, W.: TruZ-Droid: Integrating TrustZone with Mobile Operating System. In: MobiSys (2018)
76. Zhang, C., Zeng, J., Zhang, Y., Ahmad, A., Zhang, F., Jin, H., Liang, Z.: The HitchHiker's Guide to High-Assurance System Observability Protection with Efficient Permission Switches. In: ACM CCS (2024)
77. Zhang, Y., Hu, Y., Ning, Z., Zhang, F., Luo, X., Huang, H., Yan, S., He, Z.: SHELTER: Extending Arm CCA with isolation in user space. In: USENIX Security (2023)
78. Zhang, Y., Zhang, F., Luo, X., Hou, R., Ding, X., Liang, Z., Yan, S., Wei, T., He, Z.: SCRUTINIZER: Towards Secure Forensics on Compromised TrustZone. In: NDSS (2025)
79. Zhou, Q., Cao, W., Jia, X., Liu, P., Zhang, S., Chen, J., Xu, S., Song, Z.: RContainer: A Secure Container Architecture through Extending ARM CCA Hardware Primitives. In: NDSS (2025)

A Security Analysis: Memory Isolation

We detail attacks on device lifecycle management and memory isolation mitigated by guarantees (G_{id}, G_a, G_{mem}, G_d) DEVLORE assumes from prior works. **Attach/Detach Device.** The untrusted hypervisor can try to map a device to a physical address space it controls before attaching it to a CVM. This is not possible as the physical addresses that the devices are mapped to are fixed for the hardware platform and cannot be changed by software (G_{id}). Next, a hypervisor can try to emulate devices or assign the wrong device to a CVM which is stopped by ensuring unforgeable device identities (G_{id}). Concretely, DEVLORE checks the physical addresses that the hypervisor maps for the device using trusted platform information in the monitor to detect and stop such attacks. The hypervisor can also try to assign a device in a state that compromises the CVM when it is attached to it. To stop such attacks, DEVLORE soft resets the device before attaching it to the CVM (G_a). Similarly, the hypervisor can request DEVLORE to attach a device to a CVM when the CVM does not expect to have a device; this is stopped by G_a. DEVLORE does not allow such requests and expects the CVM to request a device attach. Finally, the hypervisor can try to detach a device while a CVM is using it to leak device data; this is mitigated by G_d.

Co-Resident CVMs. Attacker-controlled co-resident CVMs can try to directly access the device MMIO or DMA memory. An attacker can also try to set up realm memory regions overlapping with device memory of the victim CVM, but this would violate G_{mem}. DEVLORE adds all device memory to the CVM. When memory is added to a CVM, CCA ensures that this memory only belongs to one CVM and isolates it by programming the S2 tables in the RMM. Thus, other CVMs will not have valid mappings for the device memory and cannot access it.
SMMU Configuration and Attacker-controlled Devices. Hypervisor can try to change SMMU S2 table mappings to allow attacker-controlled devices access to victim device memory but this would violate G_{mem}. To stop these attacks and guarantee G_{mem}, DEVLORE protects the SMMU configuration in the root world to check and stop these attempts. Malicious devices can try to directly access victim device memory, but are stopped by G_{mem}. Concretely, DEVLORE ensures that the attacker-controlled device S2 tables will not have valid mappings to access victim device memory.

B Implementation Details

Additional implementation details are available in the extended version [1].

B.1 Device Attach and Detach

DEVLORE adapts the device attach process of prior works to allow CVMs to enable DEVLORE interrupt isolation [58,13,62]. CCA does not allow devices direct access to CVM memory. As in those works, DEVLORE uses 2 GPTs to attach

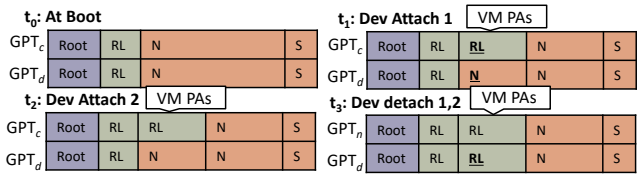

Fig. 8. DEVLORE's 2 GPTs. t0: Create and populate 2 GPTs (GPT_c for cores, and GPT_d for devices). t1: Update GPT_d on device attach to allow devices access to CVM memory by marking CVM as normal world (N) in GPT_d. t2: No updates. t3: Update GPT_d to revoke device access to CVM memory by marking it back to realm world (RL).

devices to CVMs (Fig 8). We create GPT_c for the cores and GPT_d for the devices (i.e., the SMMU GPT). In GPT_d, CVM memory shared with the device is marked as normal world, while in GPT_c the same memory is marked as realm world.

Table 2. DEVLORE new and existing API for device lifecycle management, memory isolation, and interrupt isolation. RSI and RMI: Implemented in RMM. SMC: Implemented in monitor.

API	Status	Description
`rsi_attach_dev`	New	Request attach device to CVM
`rsi_detach_dev`	New	Request detach device from CVM
`rmi_dev_finalize`	New	Hyp. finished adding device memory to realm
`rmi_data_create`	Modify	Update SMMU if devices attached to CVM
`rmi_data_destroy`	Modify	Update SMMU if devices detached to CVM
`rmi_granule_delegate`	Modify	Update 2 GPTs
`rmi_granule_undelegate`	Modify	Update 2 GPTs
`smc_ack_int`	New	Acknowledge interrupt
`smc_gic_config`	New	R/W to GIC configuration
`smc_prot_int`	New	Mark interrupts as protected

Attach. To allow CVMs to register specific device interrupts we implement a new RSI call (`rsi_attach_dev`) in the guest Linux kernel and the RMM, which the CVM invokes with the device details (i.e., ID, memory-mapped GPAs, registered device interrupts). The RMM checks this request by looking up platform information (e.g., physical address ranges for device MMIO, device interrupt numbers) and then forwards the request to the hypervisor. Then, the hypervisor transitions all memory for the device to realm world (`rmi_granule_delegate`), adds this memory to the CVM (`rmi_data_create`), and invokes `rmi_dev_finalize`. In response, the RMM checks that the hypervisor has installed the correct mappings for the CVM based on the CVM request. For memory isolation, the RMM invokes the monitor to configure the SMMU and marks the CVM memory as normal world in GPT_d and soft resets the device. Then, for registered interrupts, the RMM invokes the monitor (`smc_prot_int`) to configure it and returns to the CVM indicating that the device attach is complete.
Detach. The CVM can request a device detach (`rsi_detach_dev`). Tab 2 summarizes the APIs.

B.2 Interrupt Acknowledgment

A benign hypervisor typically acknowledges physical interrupts as soon as it receives an interrupt and then injects a virtual interrupt to the VM i.e., before servicing the interrupt. This is to ensure progress in the system—if the acknowledgment is delayed for a particular interrupt, all further interrupt invocations for this id will be blocked. However, the benign hypervisor should not use the above greedy acknowledgment approach for special types of interrupts. Specifically, certain devices constantly re-trigger the same interrupt if an interrupt is acknowledged too early i.e., the handler in the VM has not performed its operations. For example, the keyboard writes pending characters to its own memory and sends an interrupt. However, if the physical interrupt is acknowledged before the VM handler consumes the characters the keyboard thinks that the interrupt

was dropped and resends it. If the gap between the hypervisor's greedy acknowledgment and the VM's end of interrupt handler is large, the interrupts keep arriving at the GIC, which injects them into the unblocked cores one after the other. This is referred to as an interrupt storm [53].

The hypervisor knows the type of interrupt from the platform device information and uses this information to decide if it is safe to perform greedy acknowledgments for cases where it is safe. If it is not safe, the hypervisor first services the interrupt and injects a virtual interrupt to the VM. Then, the hypervisor waits for the virtual interrupt acknowledgment from the VM and only then performs a physical interrupt acknowledgment. So, when the hypervisor requests to acknowledge registered interrupts, DEVLORE relies on the device tree information. Based on the platform device information, DEVLORE either allows greedy acknowledgment or waits for the CVM to acknowledge the virtual interrupt and only then allows a physical interrupt acknowledgment (`smc_ack_int`).

C Evaluation Details

To evaluate the functionality and security of DEVLORE, we prototype it on Arm's simulator i.e., Fixed Virtual Platform (FVP). In this appendix we detail the setup and results of our evaluation. Then, we provide additional details of our performance evaluation on the Arm board.

C.1 FVP Setup

We validate DEVLORE on the Arm Fixed Virtual Platform (FVP) version `Base RevC 2xAEMvA 11.20_15 Linux64`. We configure it with 4 AEM cores with RME and 3 GB of RAM. The FVP is a simulator without accurate microarchitectural behavior, so we only validate DEVLORE functionality and compatibility with it, not performance.

C.2 Device Setups on FVP

We attach the following devices to the FVP to test DEVLORE compatibility with unmodified device drivers. The next paragraphs describe the devices and their workloads (Tab 3). Because the FVP is not cycle accurate, and because asynchronous interrupts like timers can occur irregularly, neither cycle nor event counts are a reliable performance metric. We use FVP's Model Trace Interface (MTI) to count events. As per our experimental measurements on the FVP, in the baseline **B**, guest entry and exit follow a 1:1 ratio: each RMI entry has a matching SMC exit invocation. With **I**, this changes because DEVLORE interrupt handling adds extra monitor calls: for each interrupt, the RMM invokes the root-world monitor to acknowledge the interrupt.

Keyboard. We use the FVP X11 visualization component that translates keyboard and mouse input on the x86 host to an emulated PL050 PS/2 peripheral on the FVP. We eliminate typing speed effects by using an x86 helper application

Table 3. Device-driver compatibility on the Arm FVP. DEVLORE supports unmodified drivers in CVM for all tested peripherals. IRQ: Driver uses interrupts. LED and Button do not generate interrupts, thus do not need interrupt protection.

Device	IRQs	Driver Mod.	Protected w/ DEVLORE	Setup & Benchmark
GPU	✓	No	✓	G76 kbase driver Midgard test suite
UART	✓	No	✓	Amba-pl011 driver (nano, browser, irc)
PS2/Mouse & Keyboard	✓	No	✓	Amba-kmi driver (move mouse, type n keys)
LED & Button	✗	No	N/A	Read button state, flash LED

that uses the X11/Xlib library to control keyboard input. For each key press, the FVP generates a physical interrupt. We run workloads that generate 1000 and 10000 interrupts.

GPU. The FVP supports a limited non-functional implementation of a Mali G76 GPU (No-Mali) [9]. The GPU cannot perform any computation but only exposes an MMIO layout and can send interrupts. So, we run the official Midgard Mali Kutf IRQ test suite [10]. The test suite triggers an MMIO write via the Mali driver and prompts the GPU to send 10000 interrupts.

UART. We attach the FVP UART PrimeCell PL011 peripheral to the VMs in our setup. This device facilitates serial communication which we control using an X11/Xlib library. We trigger workloads that generate 1000 and 10000 interrupts.

LED & Button. The FVP includes 8 LEDs and 8 User DIP switches (buttons) which can be controlled by software using system registers `SYS_LED` and `SYS_SW` respectively. To test their functionality, we write 1000 random values to these registers and measure the total number of interrupts.

Mouse. On the FVP, the mouse works by capturing the mouse input in the FVP visualization component. Moving the mouse from the left to the right side of the screen raises 53 interrupts. A single mouse click triggers 2 interrupts. While we can verify that the mouse is functional, we cannot benchmark it as we are unable to perform this operation (for moving the mouse, or clicking) without user interaction.

C.3 Case Study Setup on FVP

Below we detail our setup for each of the FVP case studies.

Terminal Based Browser. We cross-compile Lynx browser v2.9.2 [31] for Arm. We attach the UART to the realm VM and browse en.wikipedia.org in Lynx. On the landing page, we navigate to the *From Today's featured Article* section and follow 4 subsequent links [71].

IRC Client. We cross-compile a Console IRC Client [50] and connect to Miniircd Server v2.3 [57]. The server has 2 other users logged in. We exchange 6 messages with a user.

Text Editor. We cross-compile Nano text editor v8.1 [5] and add a new DNS entry to `/etc/hosts`.

Table 4. Glmark2 with all scenes in their standard configuration for 20k frames.

Scene	Description	# Interrupts
buffer	Vertex buffer object throughput	2528
build	Shader compilation and setup	18678
bump	Per fragment lighting with bump mapping	20030
clear	Framebuffer clear performance	20035
conditionals	Shader branching and flow control	12906
desktop	Window compositing and blending effects	24793
effect2d	2D post processing filters	10085
function	Function call overhead in shaders	12902
ideas	Dynamic multi object scene rendering	14977
jellyfish	Animated mesh deformation workload	6164
loop	Shader loop execution stress test	12886
pulsar	Procedural animation and texture sampling	14212
refract	Environment mapped refraction rendering	25394
shading	Lighting and material computation cost	15124
shadow	Shadow mapping performance	16351
terrain	Height mapped terrain rendering	32335
texture	Texture sampling bandwidth	17514

C.4 Case Study Setup on Board

We cross-compile glmark2 [35] version 2023.01 for aarch64 Linux and run it in surfaceless mode on the Mali G610 GPU (OpenGL ES 3.1, Mesa 25.0.7-2). All 17 scenes execute unmodified in their standard configuration at 1920x1080 resolution for 20'000 frames and 30 runs in each environment: a realm-world CVM (**B**) and a realm-world CVM with DEVLORE interrupt protection (**I**). The GPU frequency is fixed to 800 MHz, and DVFS is disabled to avoid frequency-scaling effects. We use glmark2 in the default configuration with: buf=32, r=8, g=8, b=8, a=8, depth=24, stencil=0, multisample anti-aliasing=0. Tab 4 enumerates the benchmarks.

C.5 Non Interrupt Benchmark Setup

We run sysbench v1.0.20 [41] on Arm board (CPU: 200k Primes, Memory: 1 GB (64k Blocks), Threads: 8, Mutex: 8) in both a CVM and normal world VM with (**I**) and without (**B**) DEVLORE changes. As expected, since no interrupts are involved, DEVLORE changes incur no measurable overheads (see Fig 9).

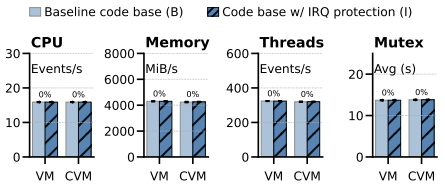

Fig. 9. Sysbench Benchmark on Arm board. No measurable impact on non interrupt workloads.