

DEVLORE: Extending Arm CCA to Integrated Devices

A Journey Beyond Memory to Interrupt Isolation

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

Abstract

Arm Confidential Computing Architecture (CCA) executes sensitive computation in an abstraction called realm VMs and protects it from the hypervisor, host OS, and other co-resident VMs. However, CCA does not allow integrated devices on the platform to access realm VMs and doing so requires intrusive changes to software and is simply not possible to achieve securely for some devices. In this paper, we present DEVLORE which allows realm VMs to directly access integrated peripherals. DEVLORE memory isolation re-purposes CCA hardware primitives (granule protection and stage-two page tables), while our interrupt isolation adapts a delegate-but-check strategy. Our choice of offloading interrupt management to the hypervisor but adding correctness checks in the trusted software allows DEVLORE to preserve compatibility and performance. We evaluate DEVLORE on Arm FVP to demonstrate 5 diverse peripherals attached to realm VMs.

1 Introduction

Arm-based platforms are deployed in consumer electronic devices (e.g., phones, cars, laptops) and the cloud. Applications running on these platforms typically use integrated devices (e.g., display, sensors, integrated GPUs) to perform their sensitive tasks such as on-device inference or authentication. Cloud providers offer integrated accelerators connected to Arm CPUs. For example, Nvidia superchips (e.g., Grace-Hopper, Grace-Blackwell) integrate Armv9-A processors with state-of-the-art GPUs [4] to support AI-models in the cloud. In both 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 allows users to execute their computation in isolation from privileged untrusted software. Arm has announced a new hardware extension called realm management extensions (RME) to natively support confidential

computing. Similar to Intel TDX and AMD SEV-SNP, Arm Confidential Computing Architecture (CCA) provides a virtual machine (VM) abstraction. CCA isolates VMs 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 confidential VMs and a thin Realm Management Monitor (RMM). These new worlds are in addition to two existing worlds: the normal/non-secure world that is reserved for the host hypervisor and the secure world which houses TrustZone-based trusted apps.

CCA isolates CPU accesses to memory based on worlds by using a hardware-based mechanism called granule protection tables (GPTs). For each physical address, the GPT tracks the world it is mapped to. Then, the hardware stops: any world from accessing the root world; the normal and secure world from accessing the realm and the root world. Lastly, the RMM software uses page tables to isolate realm VMs from each other. This architecture summarized in Figure 2(a) reduces the TCB by keeping the hypervisor and management logic in the normal world. The VMs only have to trust the firmware which includes the monitor and the RMM. All non-CPU components including integrated devices are mapped to the normal world to ensure that they cannot access the realm or root memory.

Arm CCA-based VMs executing in the realm world might 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 does not allow any devices access to VM memory. One approach to allow VMs to access devices is to create a shared encrypted memory that is maintained in the normal world. However, this is an undesirable solution due to unavailability of encryption engines in integrated devices, required changes to device drivers, and the performance overhead due to the necessary memory copies.

To this end, we present DEVLORE—a CCA-based design that allows VMs to directly access integrated devices. DE-

* These authors contributed equally to this work.

vLORE allows devices and VMs to directly access each others memory but ensures that such access does not compromise the VM’s confidentiality or integrity. DEVLORE extends the use of GPTs to attach devices to VMs. It then uses a second GPT to stop the untrusted software outside the VM from accessing the device. However, this approach is not sufficient for all devices and incurs large overheads if applied blindly. Device accesses can range from purely memory-mapped IO, to full-fledged DMA, to fine-grained memory-mapped IO and context-based views of the device state. DEVLORE caters to device-specific needs by using the two-GPT approach only when necessary and augments it with DMA-protection for devices that need it. Our two-GPT memory isolation ensures security while preserving performance.

Next, we show that device memory isolation is not enough to guarantee confidential computing. An attacker can abuse interrupts (e.g., inject, reorder, drop) to corrupt benign device drivers executing in VMs. To this end, we propose a novel interrupt isolation design that isolates both physical and virtual interrupts without any software changes to existing OSes and drivers. Our insight is to delegate the interrupt management and delivery to the untrusted hypervisor while the trusted software checks if the hypervisor misbehaves. This approach minimizes software changes, but may incur large overheads if applied blindly to all devices on the platform. To address this, DEVLORE allows configurable isolation for cases where interrupt isolation may not be necessary, just memory isolation is sufficient (e.g., device uses polling instead of interrupts). DEVLORE adapts a suitable isolation based on each interrupt ID registered by the device attached to the VM.

We prototype DEVLORE on a CCA-enabled emulator provided by Arm to demonstrate feasibility, correctness, and compatibility with hardware specification and software stack. DEVLORE changes to the hypervisor, firmware, and guest kernel are minimal (5K LoC) whereas it does not require any changes to device drivers. We demonstrate DEVLORE with 5 devices of varying complexity and isolation requirements: keyboard, button, LED, mouse, and SMMU test engine. We further benchmark DEVLORE with synthetic workloads to stress test our memory and interrupt isolation. For these stress workloads with frequent interrupts DEVLORE reports overheads of 25%. Further, for workloads that stress both frequent DMA and interrupts, DEVLORE reports overheads of 50%

Contributions. The paper makes three main contributions:

- **Integrated Devices.** DEVLORE is the first work that allows Arm CCA-based VMs to directly access integrated devices while maintaining CCA guarantees as well as existing TCB, performance, and functionality.
- **Memory & Interrupt Isolation.** Our novel approach employs two-GPT memory isolation and delegated interrupt isolation.
- **Diverse Device Support.** DEVLORE isolates diverse set

of devices connected to realm VMs without any modifications to applications or device drivers.

2 Background

We first introduce Arm CCA and then explain the memory and interrupt behavior of integrated devices.

2.1 Arm CCA

Prior to Armv9-A, Arm supported executing computation in two worlds (normal and secure) using TrustZone. From Armv9-A, Arm ISA optionally supports Realm Management Extensions (RME) for CCA. It extends the previous two worlds (normal and secure) with two new worlds (realm and root). CCA guarantees that the realm world is not accessible to the normal world. Similarly, the root world is isolated from all other world software. To enforce this isolation, the RME introduces hardware units called Granule Protection Checks (GPCs) where a granule is the smallest addressable chunk of memory. The GPCs filter all memory accesses by looking up a Granule Protection Table (GPT). The GPT maps each granule’s physical address (PA) to a world, or physical address space (PAS) of that granule. The GPT is stored in the root world and only accessible and programmed by monitor which runs in the root world.

Apart from the worlds, the Arm architecture enables privilege levels called exception levels (EL) from EL0 to EL2 in the normal, realm, and secure world. EL3 is the most privileged exception level and only executes the root world software (i.e., the trusted firmware with the monitor). CCA enables running realm VMs (in EL0 and EL1) that are not accessible to the normal world. The untrusted hypervisor executes in the normal world and manages VM resources and scheduling. CCA deploys a Realm Management Monitor (RMM) to execute in the realm world at EL2. All communication from the realm VMs to the hypervisor are routed through the RMM. Concretely, the RMM offers a Realm Service Interface (RSI) to the realm VMs which they use to send requests to the RMM and the hypervisor.

While the GPCs isolate the different worlds from each other, they are insufficient to isolate mutually-distrusting realm VMs. To isolate the realm VMs, the RMM programs their stage-2 translation tables (S2 tables) and enforces an exclusive mapping check i.e., a given physical address in the realm world is only mapped to one realm VM in the S2 tables. To program the S2 tables, the RMM delegates the task of setup and management to the hypervisor, and only performs the exclusive mapping check before programming the S2 tables.

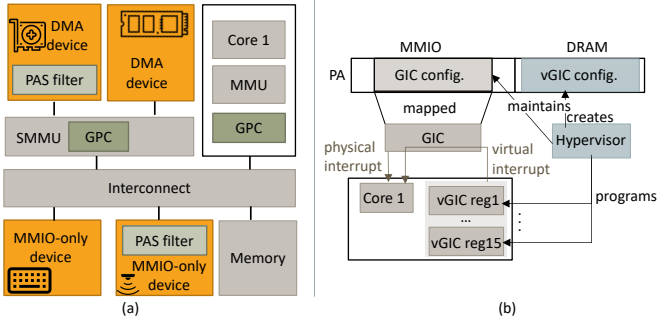


Figure 1: (a) Types of devices and CCA’s GPC and PAS filters for memory protection (b) Arm’s interrupt architecture (with and without CCA) where the hypervisor maintains the GIC configuration, creates and maintains vGIC configuration for VMs, and programs vGIC system registers. The GIC fires physical interrupts and virtual interrupts programmed in the vGIC system registers.

2.2 Integrated Devices

Integrated devices maintain their state in device registers and optionally read/write the main memory. Hypervisors can virtualize devices either by managing them on behalf of the VMs (e.g., virtio) or by enabling direct device passthrough to the VMs. With hypervisor managed virtualized devices, the hypervisor traps on all communication between the VM and device, and perform them on the VM’s behalf. With this approach, the hypervisor is trusted to perform the device operations correctly. In Arm CCA, the hypervisor is not trusted. Therefore, in DEVLORE we use direct device passthrough.

Device Trees and MMIO. Applications communicate and control integrated devices using MMIO and DMA operations. For MMIO to an integrated device, the platform defines the physical addresses that should be used for the device. These physical addresses are fixed and stored as part of the firmware image in a device tree. This device tree defines the device IDs, their interrupts, and the physical addresses used for MMIO. The monitor reads this device tree during platform boot and propagates it to the hypervisor. When the hypervisor launches VMs with devices, the hypervisor provides a virtualized device tree. The VM’s device tree contains the device IDs, the corresponding interrupts, and guest physical addresses (GPAs). The hypervisor ensures that the guest physical addresses in the device tree are mapped to the correct physical addresses in the S2 tables. With CCA, the hypervisor controls the host device tree. Each VM has its own device tree which is a part of its attestation report. Typically, devices have distinct MMIO physical addresses.

Devices can directly access host memory using DMA operations. On the Arm platform, DMA-capable devices are behind an SMMU (equivalent to an IOMMU on Intel/AMD). To isolate the devices to specific VMs, the hypervisor pro-

grams the SMMU S2 tables. With CCA, the SMMU has a GPC that the monitor programs to ensure that devices cannot access realm memory. Unlike the SMMU S2 tables configuration, the SMMU’s GPC configuration is in the root world and not accessible to the hypervisor.

2.3 Interrupts

Next, we explain interrupt management on Arm, both for non-CCA and CCA platforms.

GIC. On Arm platforms, the Generic Interrupt Controller (GIC) delivers interrupts to the cores. Each peripheral has a hardware line to the GIC to assert an interrupt. When an interrupt is asserted, the GIC sends it to a core based on the interrupt’s configuration. Interrupts can be enabled or disabled, routed to any or a specific core, and assigned a priority. The GIC uses this information whenever necessary, e.g., to order multiple pending interrupts. On Arm platforms with virtualization support, the hypervisor manages the GIC, which exposes a memory-mapped configuration address range. The hypervisor can use the GIC configuration space to selectively enable/disable interrupts, set up interrupt affinities, and program interrupt priorities.

vGIC. Each VM on the Arm platform needs a local view of the GIC. To support this, the Arm platform provides virtual GICs (vGIC) that the hypervisor sets up for the VMs. VMs can then configure and manage their vGICs and receive interrupts while being oblivious that they are not interacting with the physical GIC. The vGIC configuration space is virtualized by the hypervisor and is not backed by hardware i.e., the hypervisor creates the view of the configuration space for each VM. Virtual interrupt injection is performed by the hypervisor by programming hardware-backed vGIC system registers. Concretely, when the hypervisor programs the vGIC system registers with interrupt numbers, the hardware fires these interrupts on VM entry.

To virtualize the configuration space, the hypervisor traps when the VM tries to access the vGIC through MMIO (e.g., for changes to interrupt affinity, setting up priorities). Currently, when a physical interrupt for the VM arrives at the GIC, the GIC cannot directly inject it into the VM. If this was allowed, it would break the virtualization abstraction that the vGIC provides. Instead, the hypervisor virtualizes this interrupt and uses the vGIC to send the interrupt to the VM. To do this, the hypervisor traps on physical interrupts to VM cores, programs the vGIC system registers with the corresponding interrupt, and resumes the VM. For performance, there are many (n) system registers that the hypervisor can program to fire n distinct pending interrupts at once. The number of system registers (n) is specific to the GIC implementation. If there are several pending interrupts, the hypervisor programs the vGIC with the highest priority interrupts. On VM entry, the vGIC fires the interrupts programmed by the hypervisor.

Acknowledging Interrupts. 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 id, all further interrupt invocations for this id will be blocked. On the hand, the benign hypervisor should not use the above greedy acknowledgment approach for special type of interrupts. Specifically, certain devices constantly re-trigger the same interrupt ID 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’s 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, who injects them into the unblocked cores one after the other. This is referred to as an interrupt storm [30]. The hypervisor knows the type of interrupt from the device tree 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 interrupts to the VM. Then, the hypervisor waits for the virtual interrupt acknowledgment from the VM and only then performs a physical interrupt acknowledgment.

CCA Interrupt Support. CCA expects the hypervisor to virtualize and manage interrupts for the realm VMs. However, directly switching to the hypervisor when a realm VM’s core receives a physical interrupt can leak the realm VM’s state. So CCA ensures that all physical interrupts to cores executing realm VMs trap to the RMM. The RMM saves the state of the realm VM, and then switches to the hypervisor to virtualize the interrupt. For security, CCA does not allow the hypervisor to directly program the vGIC’s system registers and the RMM clears any values that the hypervisor might have written to these registers before entering a realm VM. Instead, to program the vGIC for the realm VMs, the hypervisor invokes the RMM with the vGIC configuration. On each RMM invocation, the hypervisor can program the vGIC to send at most n distinct pending interrupts. In response, the RMM checks if the vGIC configuration is valid according to Arm’s GIC specification and programs the realm VM’s vGIC system registers. Therefore, currently the RMM allows the hypervisor to inject all interrupts to the realm VM.

3 Integrated Devices in Arm CCA

We outline the security challenges that arise when adding CCA support for integrated devices.

3.1 Threat Model

We trust the platform and all of its components, including the GIC. 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. While we trust the platform’s hardware integration, we do not trust the firmware of all the devices in the platform. Instead, each realm VM can choose which devices to trust and communicate with. The monitor includes the firmware of these devices in the VM’s attestation report. We assume a one-to-one mapping property between a device and the address range used to access it (i.e., the MMIO range). While most devices satisfy this property, we consider devices that do not (e.g., GPIO configurations) as out of scope. We assume that all software executing in the normal and secure worlds are untrusted. We trust the CCA hardware, assume that all trusted CCA software (i.e., the RMM and monitor) are implemented according to CCA specifications, and assume mutual distrust between realm VMs. Protecting against side-channels and microarchitectural attacks is orthogonal to our work and we consider them to be out of scope.

3.2 Memory Accesses

Applications that execute on Arm-based platforms need to use various integrated devices (e.g., display, sensors, integrated GPUs). Integrated devices can either rely purely on MMIO reads and writes from the processors (e.g., sensors) or also use DMA capabilities to access the DRAM (e.g., integrated GPUs). Next, we explain the different types of devices and the security challenges in connecting them to realm VMs.

MMIO-only Devices. Some devices (e.g., sensors, buttons, LEDs) never directly access processor memory. Instead, applications control and communicate with these devices through their memory-mapped space using MMIO operations. To enable this communication, these devices are mapped into the processor’s physical address space. When the VM wants to access the device, CCA’s protections for CPU cores is sufficient to protect MMIO accesses. One can achieve this in two steps. First, map the devices into realm memory addresses to allow CCA’s realm VMs to connect to MMIO-only devices. With this, CCA’s GPCs and S2 tables for the MMU are sufficient to secure these devices. Concretely, the GPCs prevent any normal world software (e.g., hypervisor) from writing to the memory-mapped regions. This blocks all normal world access to the device. Then, the RMM’s S2 tables ensures that only one realm VM has valid translations to the memory-mapped physical addresses. So, if another realm VM tries to access this device, it leads to a translation fault blocking the access. This ensures that only the realm VM that the device is connected to can read/write to the device. Lastly, since these devices never originate accesses to processor memory, the

lack of checks from device to memory is irrelevant.

Devices with Fine-Grained MMIO Access Control. Some devices need fine-grained access control (e.g., to protect 32-bit registers). For this, CCA introduces hardware changes to the devices to implement device-specific protection checks in the form of PAS filters which allow for this fine-grained access control. With these PAS filters, all MMIO accesses from the cores to the device can be filtered on the device eliminating the need for GPCs. While the device's PAS filters perform world-level isolation, they cannot distinguish accesses from different realm VMs. Therefore, the RMM's S2 tables are necessary to ensure that only one realm VM can access the device's MMIO pages. In addition to world level filtering, the devices can use the PAS filters to expose different device views based on the world of the access. For example, a GPS sensor with PAS filters can create 2 views for the normal and realm world. When the normal world reads the memory-mapped region it sees 0 and when the realm world reads the memory-mapped region it sees the GPS co-ordinates. Similarly, an LED with PAS filters can allow the normal world to only read its value while the realm world can read and write to the LED.

DMA Devices. On an Arm CCA platform, any accesses to realm world memory by integrated devices are stopped by the GPCs. This is because the CCA platform treats the transactions originating from these devices as coming from normal world. There is only one way to connect realm VMs with these legacy devices: the hypervisor sets up a shared memory region in the normal world which is accessible to both the realm VMs and the devices. The devices read and write to this shared memory region to perform DMA operations. This shared memory region is accessible to the untrusted hypervisor that can compromise the confidentiality and integrity of the data transferred to and from the device. One option to secure this communication is for the realm VM and the device to decrypt and encrypt all data that they read and write to the shared memory region, often referred to as the bounce buffer design. Concretely, to send data to the device via a DMA write, the realm VM first encrypts the data and copies it to normal world shared memory. Before using this data, the device reads and decrypts the data. This method has two drawbacks. First, the multiple encryption/decryption cycles and data copies incur performance penalties [2]. Second, some devices (e.g., sensors, buttons) may not have cryptographic capabilities to perform encryption and decryption. Therefore, the optimal approach to connecting devices realm VMs is to allow them to directly read/write realm VM's memory. But if we allow these legacy devices direct access to realm memory, they can be used by attackers to compromise CCA security. Additionally, DMA-capable devices have memory-mapped regions used by applications for configuration and control, which means that MMIO operations for these devices also need to be protected.

3.3 Device Interrupts Can Break Realm VMs

Devices are diverse and use different mechanisms to notify processor-bound computation of events based on their needs and capabilities. To better understand these notification mechanisms, we manually analyzed the open-source drivers of popular integrated devices in the Linux kernel. Some devices use only interrupts as the notification mechanism. With CCA, the untrusted hypervisor manages the interrupts for the realm VMs. Currently, CCA does not protect against the untrusted hypervisor arbitrarily injecting interrupts. The hypervisor can inject arbitrary interrupts to compromise the security of the realm VM.

Counter. For example, the Linux kernel's 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, as shown in the Listing 1.

Listing 1: Linux interrupt-driver counter

```
1 irqreturn_t interrupt_cnt_isr
2     (int irq, void *dev_id){
3     struct counter_device *ctx = dev_id;
4     struct interrupt_cnt_priv *priv = counter_priv(ctr);
5     atomic_inc(&priv->count);
6     counter_push_event(ctr,
7         COUNTER_EVENT_CHANGE_OF_STATE, 0);
8     return IRQ_HANDLED;
9 }
```

This 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 realm VM and this counter is used by a realm VM application. The untrusted hypervisor can arbitrarily inject interrupts, trigger this interrupt handler, and increment the counter to trigger the realm VM application's event condition eventually. Therefore, using arbitrary interrupt injection, the hypervisor successfully tricks the VM into believing an event occurred when, in reality, it did not, compromising the integrity of its execution.

Wireless Device Boot-up. Qualcomm's WCNSS chip is used in a wide range of devices with Arm cores for wireless communications and vulnerable to the untrusted hypervisor's interrupt injection attacks. Concretely, the WCNSS driver used for verified firmware loading expects interrupts to indicate when the device's firmware is successfully loaded and ready to use (see Listing 2) [3].

Listing 2: Ready interrupt handler in WCNSS driver

```
1 irqreturn_t wcnss_ready_interrupt
2     (int irq, void *dev){
3     struct qcom_wcnss *wcnss = dev;
4     complete(&wcnss->start_done);
5     return IRQ_HANDLED;
6 }
```

If the hypervisor injects an interrupt to trigger this handler before the device is ready, it tricks the driver into transitions

the device’s state to ready. This may lead to undesirable effects (e.g., VM may use an outdated or a corrupted WiFi firmware image). With this attack, the hypervisor breaks the driver execution without directly accessing the device, only using interrupt injection.

3.4 Device Interrupts Need Memory Protection & More

In addition to pure interrupts, the devices can use two other types of event notification mechanisms. These mechanisms either only poll memory-mapped regions or use interrupts in conjunction with memory-mapped region reads. In either case, the interrupt handler reads or checks MMIO memory, which, if left unprotected, makes the VM vulnerable to the interrupt attacks from Section 3.3.

Polling. Some devices write to their own memory when an event occurs. This device memory is mapped into the processor’s physical address space. On the processor, the device driver or a user-space process continuously reads the memory-mapped region and waits for a specific value. Devices drivers that expect the events to occur frequently (e.g., network drivers built for high throughput) typically use polling. In these cases, the core polling continuously does not waste a lot of cycles. The hypervisor can compromise such devices if the memory-mapped regions that the driver reads is unprotected.

Interrupts Followed by Memory Read. For some devices, the drivers execute an MMIO register read in the interrupt handler to guard against spurious interrupts i.e., interrupts that might have misfired due to an inconsistent state in the device or the processor [5]. For example, the Arm Mali GPU driver’s interrupt handler performs an MMIO read to check the interrupt status. If the check fails, the handler does not process the interrupt and simply returns.

Listing 3: Mali GPU interrupt handler

```
1 irqreturn_t kbase_job_irq_handler
2         (int irq, void *data){
3     ...
4     struct kbase_device *kbdev = kbase_untag(data);
5     ...
6     u32 val = kbase_reg_read(kbdev,
7         JOB_CONTROL_REG(JOB_IRQ_STATUS), NULL);
8     ...
9     if (!val)
10        return IRQ_NONE;
11    ...
12    kbase_job_done(kbdev, val);
13    return IRQ_HANDLED;
14 }
```

In this case, the hypervisor can inject arbitrary interrupts to trigger this handler. However, if the hypervisor cannot compromise this register, the injected interrupt cannot trick the device driver into processing the interrupt, thus stopping the attack.

Memory Isolation is Necessary but not Sufficient. If all device drivers adopt one of the two mechanisms outlined

above and the MMIO regions are protected, it can thwart malicious interrupt injection. This stops the hypervisor’s attempts to compromise the realm VM security using interrupts. However, this solution is infeasible in some cases, such as the interrupt-driven counter, which purely 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 realm VM such that the hypervisor cannot corrupt it, requires architectural guarantees for interrupt authenticity. For devices where such memory-mapped reads would be possible (e.g., Qualcomm’s WCNSS chip), doing so may incur hardware changes to the device to write to the register. In summary, any device that simply raises an interrupt when an event occurs will need to be changed to also write to a memory-mapped region, which can then be checked by the device driver. One approach is to update existing device drivers where such a check is feasible, but this will incur significant manual driver code patching. More importantly, it is still insufficient for cases where the device simply does not have hardware support to write to a register when it triggers an interrupt. Another approach is to never allow VMs to access devices directly and strictly use bounce buffers. Such strict isolation that enforces exclusive memory access may seem sufficient to protect against malicious interrupts. However, depending on the granularity of encryption and integrity protection, the hypervisor can still trigger malicious DMAs to trick the VM into initiating unintended memory transfers. The only solution, then, is to disable all interrupts for all devices connected to the VM, as the hypervisor can never inject any interrupts, malicious or benign. However, this restriction not only degrades the performance significantly by forcing all devices to poll but also renders certain devices incompatible with CCA irrespective of bounce buffers (e.g., purely interrupt-based notification devices).

4 DEVLORE Overview

The challenges posed by the diversity of devices and attacks in Section 3 motivates the need for a comprehensive memory and interrupt isolation primitive.

4.1 Goal & Insight

DEVLORE’s core principle is to allow VMs direct access to hardware resources (memory and interrupts). However, an untrusted hypervisor can exploit this to compromise the VMs (e.g., by tricking them into accessing untrusted memory or handling malicious interrupts). To mitigate these threats, we employ a “delegate-but-check” strategy where the hypervisor manages resources but DEVLORE enforces strict monitoring. DEVLORE uses two GPTs to allow the devices to directly access realm memory, while stopping the hypervisor’s accesses.

Next, it uses S2 tables for both the cores (MMU) and devices (SMMU) to stop mutually distrusting VMs and devices from accessing each other’s memory. Section Section 4.2 provides our detailed design for memory isolation.

Interrupt isolation turns out to be a more complex problem. Our seemingly simple principle is challenging to enforce because DEVLORE needs to know the ground truth for enforcing checks for each of the interrupt management tasks: configuration, delivery, acknowledgment. To this end, DEVLORE has to track which VM owns which devices to check if the hypervisor can perform a interrupt management task pertaining a device attached to a VM. To isolate interrupt configuration, DEVLORE traps the hypervisor’s actions and checks if they are benign. Interrupt delivery has to be similarly checked for authenticity. DEVLORE intercepts interrupts from the GIC before they reach the hypervisor or the VM. We maintain the ground truth provided by the GIC (which device initiated which interrupt ID with what priority). DEVLORE then allows the hypervisor to continue and set up the (potentially malicious) interrupt delivery. When the hypervisor wants to inject an interrupt into a VM, DEVLORE interposes this attempt and can reliably check whether this injection is benign or malicious. Lastly, acknowledgment that are necessary to signal that interrupt handling is complete needs similar care. This is to ensure that the hypervisor does not prematurely acknowledge the interrupt, tricking the GIC into re-firing pending interrupts before the existing one is handled completely by the VM. In summary, DEVLORE subjects all hypervisor-level handling pertaining interrupts to rigorous validation against the ground truth. We provide a high-level overview of our interrupt design in Section 4.3 and the exact enforcement in Section 5.2.

To support different devices without a performance penalty, DEVLORE allows the realm VM to decide which protection mechanisms to enable for the device. When a realm VM requests attaching a device, it can selectively request enabling the GPT-based memory views, SMMU protection, MMIO space delegation, and turn the interrupt protection on/off.

4.2 Memory Isolation

DEVLORE allows VMs to configure the type of memory-isolation to employ based on the device types. Table 2 shows the overview of DEVLORE memory isolation, to ensures compatibility with a wide-array of integrated devices that can be securely connected to realm VMs without incurring undue performance penalties. CCA invariants guarantee that realm VMs can only access their own memory using the RMM’s S2 tables. Further, CCA invariants also ensure that no untrusted normal world software can access realm memory. In DEVLORE, we extend these invariants to devices with a goal to ensure that when the device or the VM initiate an access, they observe the same memory behavior. Next, we explain how DEVLORE leverages the flexibility of CCA’s GPCs and

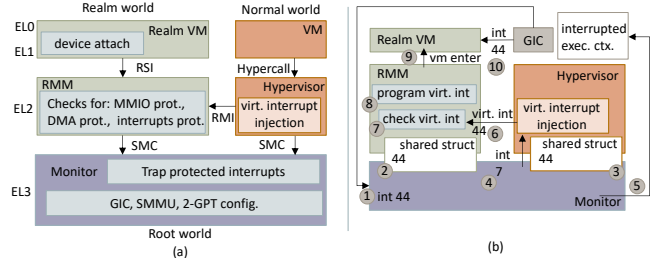


Figure 2: DEVLORE Design. (a) Isolation of devices attached to realm VMs (b) DEVLORE’s interrupt isolation flow where a realm VM has turned on interrupt isolation for interrupt 44. 1. Interrupt 44 traps to monitor 2,3. monitor writes 44 to RMM and Hypervisor data structures. 4. monitor notifies the hypervisor using an interrupt. 5. monitor returns to the interrupted app. 6. Hypervisor sends virtual interrupt configuration to the RMM. 7. RMM checks the virtual interrupt configuration 8. and programs the vGIC system registers 9. and enters the Realm VM 10. On VM entry the GIC fires interrupt 44 to the VM.

SMMU to achieve this goal for a wide-array of integrated devices.

Allowing Legacy Devices to Access Realm. CCA does not allow DMA-capable legacy devices to access realm memory because it treats all DMA from these devices as originating in the normal world. DEVLORE enables these legacy devices to directly access the realm VM’s memory. Specifically, DEVLORE introduces two separate GPTs: one for cores (GPT_c) and one for devices (GPT_d). The core GPT presents the realm VM’s memory as belonging to the realm world to the cores. On the other hand, the device GPT presents the same realm VM memory as normal world to the devices. These two GPTs ensure that both the cores and the devices can access the realm VM’s memory. Because the core GPT marks this memory as realm, untrusted normal world software (e.g., the hypervisor) cannot access it according to CCA specifications. The monitor programs all cores with GPT_c , and the SMMU with GPT_d (see Figure 3). With this, the GPC in the SMMU allows these devices to access the realm VM’s memory which is marked as normal world in the GPT_d .

DMA. While DEVLORE’s device GPT allows legacy devices connected to realm VMs to directly access realm memory, it does not stop an attacker controlled device from also gaining this access using rogue DMA. Typically, the hypervisor uses the SMMU to isolate devices from each other to prevent such rogue DMAs. However, in our setting, the hypervisor is untrusted and therefore we cannot use a hypervisor-controlled SMMU to prevent such attacks. Recent works solve this issue by protecting the SMMU in the root world, using the monitor to check all configurations that the hypervisor requests, and leveraging SMMU’s S2 tables to isolate device accesses [36]. In DEVLORE, we use a similar mechanism and protect the

SMMU for DMA-capable devices. Even after SMMU configuration is protected, if the VM and SMMU S2 tables are not synchronized, an attacker can use it to mount split-view attacks where the realm VM and device inadvertently access different memory regions because of mismatched S2 table mappings. To prevent this, DEVLORE copies the realm VM's S2 tables to the SMMU when the device is attached to a realm VM. Furthermore, DEVLORE uses the RMM to ensure that any update to the realm VM's S2 tables also updates the SMMU's S2 tables using the RMM whenever the hypervisor adds new memory to the realm VM.

MMIO. All MMIO accesses originate from the cores and never from the devices. Unlike DMA, the device never accesses the MMIO space directly. So, DEVLORE does not need to create different GPTs to support MMIO securely. Instead, when a device is attached to a realm VM, DEVLORE ensures that its MMIO region (as indicated by the realm VM) is delegated to realm world and added to the realm VM's memory. This stops the untrusted hypervisor from accessing the device through its memory-mapped space. CCA ensures that any physical address in the realm world only belongs to one realm VM or the RMM. This also applies to the MMIO region that DEVLORE adds to the realm VM. This ensures that only one realm VM can access the device's MMIO space.

4.3 Interrupt Isolation Overview

The untrusted hypervisor can arbitrarily inject device interrupts into the realm VMs leading to attacks described in Section 3.3. The realm VM cannot detect such malicious interrupt by itself because the hypervisor is responsible for realm VM interrupt management. Concretely, when a device connected to a realm VM sends a physical interrupt, the GIC notifies the hypervisor. In response, the hypervisor injects the corresponding virtual interrupt by programming the vGIC for the realm VM. A malicious hypervisor can arbitrarily inject both physical and virtual interrupts by programming the GIC and vGIC respectively. To protect against this threat, DEVLORE aims to provide isolation for both physical and virtual interrupts, such that the VM under attack exhibits the same behavior as it would under a benign execution.

DEVLORE achieves this with two main observations. First, if the hypervisor cannot access the GIC directly, it cannot inject the device's physical interrupts. With this, DEVLORE ensures that all physical interrupts from the devices are authentic. Second, with CCA, the hypervisor always calls into the RMM to program the realm VM's vGIC for virtual interrupts. DEVLORE uses this for when the hypervisor requests vGIC programming. In particular, DEVLORE can rely on the RMM to match the virtual interrupt against the authenticated physical interrupt. Next, we explain how DEVLORE uses these insights to establish the authenticity of physical and virtual interrupts (see Figure 2(b)).

Prevent Malicious GIC Writes. DEVLORE stops the hypervisor from directly writing to the GIC memory, which is necessary to change interrupt configurations (e.g., priorities) or inject physical device interrupts. Specifically, DEVLORE marks GIC memory as root world and uses the monitor to check any updates to the GIC's configuration. This ensures the authenticity of physical interrupts, i.e., any physical device interrupt that arrives at the cores is guaranteed to be from the device itself.

Check All Virtual Interrupt Injections. When the hypervisor injects the virtual interrupts (i.e., requests vGIC programming), DEVLORE needs to check if there was a corresponding authentic physical interrupt. To perform this check, DEVLORE first needs to store all the authentic physical device interrupts. To capture all device interrupts, it uses the monitor to program the GIC such that the device interrupts for which realm VMs have turned on interrupt isolation always trap to the monitor in EL3. When the monitor receives the interrupt, DEVLORE stores the interrupt in an ordered data structure in the RMM's memory. When the hypervisor requests to inject virtual interrupts, the RMM looks up the data structure and performs checks before forwarding them to the realm VM. 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 Section 5.2. DEVLORE's design ensures that even if the interrupt management remains in the hypervisor, it cannot break the confidentiality and integrity of the VMs.

5 Design

As in CCA, DEVLORE also spawns realm VMs by taking in the VM image which includes the guest kernel, and device tree. DEVLORE requires supplementary information as part of the VM image: (i) for each pass-through device that needs protection, the device tree has to state the type of memory and interrupt isolation. Based on this, DEVLORE can infer which regions of memory (DMA, MMIO) and interrupts (IDs, priorities) need to be used for runtime enforcement. (ii) whether a pass-through device can be hot-plugged, and if so the reset function in the driver. The device-specific information is provided during VM bootup, is part of the attestation report, and is used by the guest kernel during runtime as we explain next.

5.1 Device Lifecycle

DEVLORE allows realm VMs to request and relinquish devices at any point during their operation. The realm VM requests the RMM to attach a device by specifying the memory and interrupt isolation that DEVLORE should enable. The

RMM then communicates with the hypervisor to attach the device to the realm VM.

Device Attach. DEVLORE has to ensure device identity (i.e., the correct device is attached) and that other untrusted software cannot access it. Arm hardware platform defines the physical addresses that integrated devices map to. The software cannot modify these physical addresses, and the monitor adds their measurement to the platform attestation report. This also implies that accessing this physical address always results in an access to the specific device. Thus the physical addresses are tightly coupled with device identity and access [18, 42]. Therefore, to attach a device to a realm VM, DEVLORE expects the hypervisor first delegates the memory-mapped physical address space to the realm world and adds them to the realm VM’s memory using RMM APIs. While adding these physical addresses to the realm VM, by default the RMM ensures that no other realm VMs have access to the physical addresses. These steps allow the realm VM to access the device while guaranteeing that the hypervisor and all other realm VMs cannot access it.

Reset and Attest the Device. When the hypervisor delegates and adds device memory to the realm VM, the device might still contain stale or uninitialized state. DEVLORE first soft resets the device and then completes the device attach. Once the device is attached, the realm VM invokes the RMM to extend its attestation measurements over the newly attached device memory. This records the initial state of the device and its configuration which can be propagated to a remote verifier using CCA mechanisms. The above procedure to attach, reset, and attest the device is sufficient for realm VMs to connect and communicate with MMIO-only devices. Next we explain how DEVLORE handles realm VM requests for DMA protection.

Allowing Devices to Access Realm Memory. CCA does not allow legacy devices to directly access realm memory because it treats all memory accesses from these devices as originating from the normal world. Concretely, when a device initiates a realm memory request, the GPC check sees that this request originated in the normal world but is trying to access realm memory. CCA does not allow normal world elements to access realm memory, therefore this GPC will block this access. In DEVLORE, to allow realm VMs to use DMA-capable devices, we need to enable devices to access VM’s memory. But in doing so, DEVLORE has to ensure that attacker-controlled devices cannot bypass CCA security i.e., access VMs that they are not attached to. To this end, DEVLORE creates two different GPTs for the cores and devices.

DEVLORE uses the monitor during platform boot to initialize 2 identical GPTs: GPT_c for the core, GPT_d for the device (see Figure 3). The monitor then programs the cores with GPT_c to filter memory accesses from the cores and the SMMU with GPT_d to filter accesses from the devices. During runtime, when a DMA-capable device is added to a realm VM, DEVLORE uses the monitor to update the device GPT

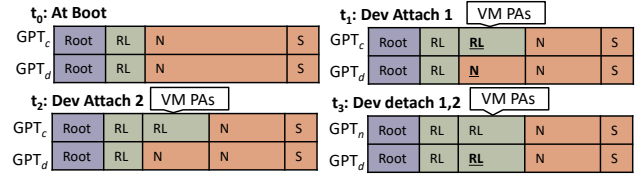


Figure 3: DEVLORE’s two GPTs. t_0 : DEVLORE creates and populates 2 GPTs at platform boot. t_1 : DEVLORE updates GPT_d when the first device is attached to the VM. t_2 : DEVLORE does not need to update the GPTs on subsequent device attach. t_3 : When the devices are detached, DEVLORE marks all realm VM memory back to Realm world.

to allow the device to access realm VM memory. Specifically, during the device attach process, the monitor marks all memory belonging to the realm VM as normal world in GPT_d . With this change, the GPC in the SMMU will see the realm VM’s memory as normal world memory. Therefore, when the device accesses the realm VM’s memory, the GPC will allow this access through. Further, if more memory is added to the realm VM while the device is attached to it, DEVLORE ensures that the GPT_d is updated to ensure that the device always has access to all of the realm VM’s memory.

DEVLORE does not require reprogramming the GPC for the cores and the SMMU during runtime and does not incur context switch overheads. Further, when GPT_d is updated during device attach, DEVLORE does not need to synchronize core GPCs or flush their TLBs as changing GPT_d only affects the SMMU’s GPC.

Isolating DMA-capable Devices. With the 2 GPT setup, DEVLORE allows legacy DMA-capable devices to access realm VM memory directly. However, without the right protections, all devices can use this to access the realm VM memory to compromise CCA security. Therefore, DEVLORE isolates the devices to ensure that only devices that are attached to a realm VM can access the VM’s memory. For this, DEVLORE protects and uses the SMMU’s S2 tables as explained in Section 4.2.

Device Detach. DEVLORE allows realm VMs to request a device detach at any time. DEVLORE soft resets the device and notifies the hypervisor that the device memory can be undelegated. The hypervisor can then transition the device memory to the normal world, whereby CCA ensures that the memory is scrubbed and therefore does not leak any state. The hypervisor may want to reclaim a device forcibly from a realm VM. It can do so by terminating and destroying the realm VM and transitioning all its resources back to the normal world. During VM termination, DEVLORE ensures that any devices attached to the VM are soft reset before they are handed back to the hypervisor.

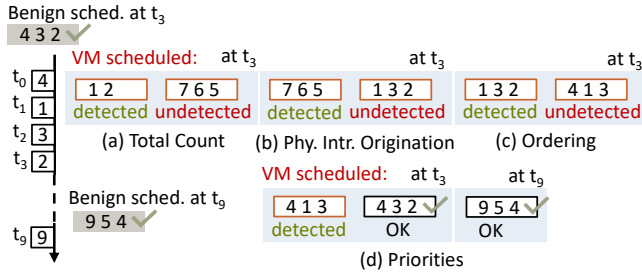


Figure 4: Device interrupts 1-9 are registered by a realm VM for DEVLORE’s secure interrupt delivery. For simplicity, interrupt number = interrupt priority and vGIC window size $n=3$. (a) Counting: Detects if hypervisor injects too few interrupts but not which interrupts are injected. (b) Tie virt. to phys.: Detects if hypervisor injects fake interrupts but not if the interrupts are reordered. (c) Ordering: Detects if hypervisor reordered interrupts but does not consider priority. (d) Priority: Detects if hypervisor injects lower priority interrupts when higher priority interrupts are pending. Scheduling: if the hypervisor maliciously stalls the realm VM, the interrupt injection behavior is equivalent to a possible benign scheduling case.

5.2 Interrupt Isolation Design

The isolation of interrupts presents different challenges than the isolation of device memory. Each device belongs to exactly one VM, and we can rely on existing primitives such as the GPCs and SMMU, which only need to be set up once. The hardware enforces isolation without further software intervention. In contrast, the GIC and its state is one resource that needs to be shared between the hypervisor and all the VMs. Further, DEVLORE not only needs to isolate the interrupt configuration but also their delivery during runtime. During device attach, realm VMs indicate to the RMM which device interrupts should be protected. The RMM registers these device interrupts for protection in the monitor. For these registered device interrupts, DEVLORE guarantees interrupt isolation (configuration, delivery, acknowledgment) under an untrusted hypervisor who performs interrupt management.

Inject Physical Interrupts to Realm VM. A seemingly straightforward way to achieve this guarantee would be to ensure that the physical device interrupts are always routed to the right realm VM core. This is possible in Arm, as the GIC programs the interrupt routing affinity to the cores. However, by default the hypervisor manages the GIC and can maliciously modify GIC configuration to route interrupts to other cores. We consider a potential approach to circumvent this threat. DEVLORE can move the GIC configuration in the root world and protect it using the monitor checks highlighted in Section 4.3. To maintain hypervisor functionality, DEVLORE introduces a GIC configuration interface for the hypervisor to use (see Figure 5). With this, DEVLORE can

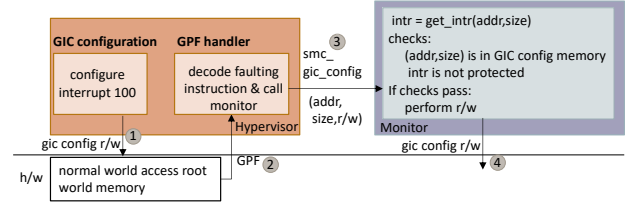


Figure 5: DEVLORE’s Hypervisor GIC config. flow. 1. Hypervisor tries to configure GIC 2. This operation tries to access root memory so the hardware raises a Granule Protection Fault (GPF) 3. that the hypervisor traps on. In the trap handler, reconstructs the faulting instruction and invokes the monitor to perform the GIC config. operation 4. The monitor checks the hypervisor’s request and then performs the GIC config. operation.

ensure that the GIC always injects the registered physical interrupts from the devices to the right realm core. But this alone will not provide interrupt isolation guarantees to the realm VM. With CCA, the realm cores always trap into the RMM in EL2 when they receive a physical interrupt. On trap, RMM alone cannot forward the physical interrupt to the destination VM, only the hypervisor can do this. So the RMM has to rely on the hypervisor to do the virtual injection. Therefore, simply routing all registered physical device interrupts to the corresponding realm cores does not provide DEVLORE’s interrupt security guarantee. Instead, DEVLORE has to reason about both physical and virtual interrupt security for the registered device interrupts. Next, we explain our refined approach.

Authenticated Physical Interrupts. As explained in Section 4.3, DEVLORE first establishes the authenticity of physical interrupts and uses it as the ground truth to check that the hypervisor benignly virtualizes them for the realm VM. To establish physical interrupt authenticity, DEVLORE protects the GIC in the root world and traps all registered device interrupts to the monitor in EL3. This allows the monitor to record information about all registered physical interrupts that the GIC fires. 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 the monitor’s physical interrupt information to guarantee benign virtual interrupt delivery.

Delegate-but-check Virtual Interrupt Management. As in CCA, the hypervisor is still responsible for interrupt management for the realm VMs in DEVLORE. While it would be possible to implement virtual interrupt management in the RMM directly, DEVLORE opts for CCA’s design philosophy of delegate-but-check and shows that it is possible to do this securely for virtual interrupts. Therefore, the hypervisor should virtualize the device interrupts that the monitor trapped for the realm VMs. 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 realm VM. Concretely, according to CCA specifications, the hypervisor sends the vGIC configuration information to the RMM. With DEVLORE, the RMM checks this information and then programs the vGIC that fires the virtual interrupts to the realm VM.

Checking Virtual Interrupts. Next, we explain checks that the RMM has to perform with the monitor’s cooperation. We start with simple checks DEVLORE can do but will be insufficient and iterate over them to build towards our full set of checks. Figure 4 shows each set of checks with a running 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 realm VM. Concretely, the RMM uses the physical interrupt information that the monitor stores to compute the total number of interrupts that should be injected for each realm VM. Subsequently, when the hypervisor injects the virtual interrupts to the VM, the RMM decrements the count of pending interrupts for that realm VM. This solution prevents the hypervisor from injecting too many or too few interrupts (see Figure 4(a)). However, this checking method fails to prevent attacks where the hypervisor maliciously injects a virtual interrupt that the device had never sent.

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’s list, and only then forwards the virtual interrupt. 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 Section 2.2). So, for each virtual interrupt 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 Figure 4(b)).

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 realm VM, DEVLORE does not need to guarantee ordering for these n distinct interrupts. If the RMM programs the vGIC with n distinct interrupts, when entering the realm VM, the hypervisor will fill all n at the same time to the realm VM.

	t_0	t_1	t_2	t_3
Benign				
pending int.	1 1 1 1	1 1 1	1 1 4 5	1 1
hyp. op.	schedule	schedule	schedule	schedule
virt. intr.	1	1	5 4	1
Benign				
pending int.	1 1 1 1	1 1 1	1 1 1 4 5	1 1 1 4 5
hyp. op.	schedule	stall	stall	schedule
virt. intr.	1			5 4 1
Malicious				
pending int.	1 1 1 1	1 1 1	1 1 1 4 5	1 1 1 4 5
hyp. op.	schedule	stall	stall	schedule
virt. intr.	1			5 4 1

Figure 6: Any malicious scheduling is always equivalent to some benign scheduling that the VM expects

Therefore, the RMM uses the monitor’s ordered list to check ordering with a window size of n . This prevents the hypervisor from reordering the interrupts that are outside this window, as shown in Figure 4(c). However, this is insufficient to achieve DEVLORE’s interrupt isolation. During benign operation, the hypervisor injects virtual interrupts based on the priority of all the pending physical interrupts. Therefore, using time-based ordering is not sufficient to preserve the priority properties.

Check #4: Priorities. To check virtual interrupt priorities, DEVLORE requires the realm VM to assign priorities to its registered device interrupts during device attach. The RMM uses this information to create per-VM ordered priority lists of interrupts and uses it to check the virtual interrupt injections from the hypervisor (see Figure 4(d)). This check ensures that the hypervisor always injects the highest priority pending interrupts. We conclude that this check is sufficient to achieve DEVLORE’s guarantee that the interrupt behavior for registered device interrupts will be equivalent to a benign behavior (Figure 4(d)). This check ensures that the hypervisor cannot inject or reorder interrupts in a way that a benign VM execution does not expect. This guarantee is true even if the hypervisor maliciously delays VM scheduling until a high-priority interrupt arrives to effectively never inject a lower priority interrupt. We observe that a benign hypervisor never provides scheduling guarantees to a VM, and the VM is expected to be able to handle any stalls or delays in its scheduling (see Figure 6 and Figure 4(d)).

In summary, DEVLORE achieves interrupt isolation by enforcing the composition of checks #2, #3, and #4.

Acknowledging Interrupts. The DEVLORE checks explained above ensure isolated interrupt configuration and delivery. However, even in the presence of 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’s access and forcing SMC calls. When the hypervisor tries to acknowledge any interrupt via an SMC call, the monitor first checks if this inter-

rupt 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’s functionality (See Section Section 2.3). DEVLORE relies on the device tree information and allows greedy acknowledgment or waits for the VM to acknowledge the virtual interrupt and only then allows a physical interrupt acknowledgment.

6 Security Analysis

We argue how DEVLORE achieves its security goals in the presence of different adversaries.

Attach/Detach Device. The untrusted hypervisor can try to map a device to a physical address space it controls before attaching it to a realm VM. This is not possible as the physical addresses that the devices are mapped to is fixed for the hardware platform and cannot be changed by software. Next, a hypervisor can try to emulate devices or assign the wrong device to a realm VM. 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 realm VM when it is attached to it. To stop such attacks, DEVLORE soft resets the device before attaching it to the realm VM. Similarly, the hypervisor can request DEVLORE to attach a device to a realm VM when the VM doesn’t expect to have a device. DEVLORE does not allow such requests and always expects the realm VM to request a device attach. Finally, the hypervisor can try to detach a device while a realm is using it to leak data from the device. DEVLORE only allows the hypervisor to force detach a device and soft resets and clears the device before transferring control to the hypervisor.

Co-Resident Realm VMs. Attacker-controlled co-resident realm VMs can try to directly access the device’s MMIO or DMA memory. An attacker can also try to setup overlapping realm memory regions with device memory of the victim realm VM. DEVLORE adds all device memory to the realm VM. When memory is added to a realm VM, CCA ensures that this memory only belongs to one realm VM and isolates it by programming the S2 tables in the RMM. Therefore, other realm VMs will not have valid mappings for the device memory and cannot access it. An attacker controlled realm VM can try to inject interrupts to other realm VMs. This is not possible as each realm VM can only program its local vGIC and cannot access the vGIC of other VMs, because of RMM’s S2 tables, or the GIC which is protected in the root world.

SMMU Configuration and Attacker-controlled Devices. The untrusted hypervisor can try to change SMMU’s S2 table mappings to allow attacker-controlled devices access to victim device memory. DEVLORE protects the SMMU configuration in the root world and checks and stops these attempts.

Attacker-controlled devices can try to access victim device memory directly. These attempts are stopped by DEVLORE using the SMMU’s S2 tables. Concretely, DEVLORE ensures that the attacker-controlled device’s S2 tables will not have valid mappings to access victim device memory. An attacker can try to use a device to fake a victim device’s interrupts. However, the hardware integration of the device on platform is trusted and captured as part of CCA’s attestation report. Because the interrupt lines for each device is unique and tied to the hardware, software attackers cannot use devices to fake other victim device interrupts.

GIC Configuration and Interrupt Injection Attacks. The hypervisor can try to reprogram the GIC to not trap to the monitor, compromising DEVLORE’s physical interrupt authentication mechanisms. Similarly, the hypervisor can try to reconfigure the interrupt priorities or acknowledge interrupts directly in the GIC to compromise DEVLORE’s device 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 invoke the monitor’s GIC configuration interface (see Figure 5) with arbitrary addresses in the root world to read or write from. DEVLORE stops such attempts by checking that the address range that 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’s interrupt isolation. However, DEVLORE traps all interrupts it protects to EL3 by marking them as EL3 interrupts (i.e., Group 0 interrupts). According to the Arm architecture, the hypervisor executing in EL2 cannot use the system registers to configure these EL3 interrupts and therefore stops such attacks.

The hypervisor can stall a realm VM’s scheduling (Figure 6) 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 a VM’s execution with a benign hypervisor as hypervisors do not provide scheduling guarantees for VMs. Therefore, this malicious stalling of the realm VM is safe and does not lead to an interrupt injection behavior that diverges from what the VM 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 only injects any given interrupt only once on every realm entry. Therefore, while the vGIC can be programmed to fire 16 distinct interrupts at once it can never be used to fire the same interrupt multiple times without reprogramming.

Table 1: Summary of changed LoC per component.

Component	Modified	Added	Removed
TF-A	77	2242	8
RMM	25	1170	4
Kernel (Guest + Host)	38	3003	1388

7 Implementation

To prototype DEVLORE, we change Arm’s reference implementation of the trusted firmware-a (monitor) v2.8, the RMM v0.2.0, the modified host and guest Linux kernel v6.2.0 with CCA support (cca-full/rfc-v1) as shown in Table 1 and interface changes as shown in Table 3. In the absence of CCA-enabled hardware, we execute our DEVLORE prototype on Arm’s Fixed Virtual Platform (FVP) with CCA support. The FVP supports several integrated devices (e.g., keyboard, audio) which we use to demonstrate DEVLORE.

7.1 Changes to Platform Boot

Like previous works, we create an additional GPT during boot for GPT_d , move the SMMU configuration and memory to the root world [36, 42]. We also move GIC configuration to the root world during boot. Further, we modify the boot process of monitor and RMM to store device information for the platform. During platform boot, when the monitor reads the platform’s device tree from the hardware, we modify it to copy the device tree into RMM memory. This device tree is trusted and maps all devices to fixed physical address spaces and enumerates the interrupts they use. During runtime, the RMM looks up this device tree and uses it to check device attach requests.

7.2 Device Passthrough

The Arm reference implementation uses KVM in the Linux kernel and a virtual machine manager (VMM), kvmtool to build and deploy realm VMs. Currently, CCA’s kvmtool only supports hypervisor-managed virtio devices where all VM device operations trap to the hypervisor. Kvmtool does not support device passthrough for integrated devices where the device is fully controlled by the VMs without KVM interference. Without this passthrough, the VMs always trap to KVM to perform device operations. We require this passthrough to correctly attach devices to realm VMs such that they communicate with the devices directly without hypervisor interference. For this, we first build a device passthrough mechanism for normal world VMs which we use as our baseline. From kvmtool, we invoke KVM to map the device physical memory to the VM in its S2 tables. With this, the device can directly access and control the device. Next, we build a framework in kvmtool that uses KVM to trap on all physical device inter-

Table 2: DEVLORE allows users to configure the types of memory protection based on the type of device.

Device Type	GPC	MMU S2	SMMU	Views
MMIO-only	✓	✓	✗	✗
MMIO-only with PAS	✗	✓	✗	✗
Legacy DMA	✓	✓	✓	✓

Table 3: DEVLORE new and existing API.

API	Status	Description
rsi_attach_dev	New	Request attach device to VM
rmi_dev_finalize	New	Hyp. added device memory to realm
rsi_detach_dev	New	Request detach device from VM
rmi_granule_delegate	Existing	update 2 GPTs
rmi_granule_undelegate	Existing	update 2 GPTs
rmi_data_create	Existing	update SMMU if devices attached to VM
rmi_data_destroy	Existing	update SMMU if devices attached to VM
smc_prot_int	New	mark interrupts as protected
smc_gic_config	New	R/W to GIC configuration
rsi_ack_int	New	acknowledge level-triggered interrupts

rupts and program the vGIC to inject them as virtual interrupts into the VM. With this, the device can directly access and control the device and receive interrupts from it.

The only thing left is for KVM to decide when to acknowledge the interrupt i.e., perform the end-of-interrupt operation. This operation depends on whether the device interrupt is level-triggered or edge-triggered interrupts. For level-triggered interrupts, we observe an interrupt storm from the device if KVM does not synchronize the physical and virtual interrupt acknowledgments. Concretely, if KVM acknowledges the physical interrupt before the VM performs the virtual interrupt acknowledgment, the device continuously sends the physical interrupts to the host. To handle this case, when a level-triggered interrupt arrives, we first disable it in the GIC such that the GIC does not fire it again preventing the interrupt storm. Then, we acknowledge the physical interrupt. To determine when to re-enable the interrupt again, we use KVM to check the vGIC state on every VM exit. We modify KVM to use the vGIC state to re-enable any physical device interrupts after the VM has acknowledged the corresponding virtual interrupts. On the other hand, edge-triggered interrupts do not need the acknowledgment synchronization. So, KVM directly performs this operation before programming the vGIC. Next, we use this passthrough mechanism to attach the devices to realm VMs according to DEVLORE design.

7.3 Attach/Detach a Device

While attaching a device, the realm VM can choose which device interrupts to isolate and if the device needs DMA memory isolation. For this, we implement a new RSI call (`rsi_attach_dev`) in the Linux kernel and the RMM. To attach a device, we change the realm VM’s kernel to look up its device tree and finds the ID, memory-mapped GPAs and the interrupts for the device. It sends this information to the RMM in the form of `rsi_attach_dev`. In the RMM, we implement

checks where the RMM looks up the device tree from the monitor and matches it against the request from the device. If the corresponding device ID does not exist in the RMM’s device tree or the interrupts don’t match, it returns an error to the realm VM. If these checks pass, the RMM sends the request to the hypervisor to delegate the device to realm memory. The hypervisor uses existing RMM APIs to transition the device memory to realm world (`rmi_granule_delegate`) and add this memory to the realm VM (`rmi_data_create`).

Once the hypervisor has finished transitioning all device memory, it invokes a new `rmi_dev_finalize` call. In response, the RMM checks that all the device GPA to PA mappings the hypervisor installed for the VM are correct i.e., the hypervisor has mapped the GPAs (indicated by the realm VM in `rsi_attach_dev`) to the PAs in the RMM’s device tree. If the VM requested interrupt isolation, the RMM checks its device tree and ensures that these interrupts belong to the device, and then invokes a new `smc_prot_int` in the monitor to deploy DEVLORE’s protections for these interrupts (see Section 7.4). If the checks pass, the RMM soft resets the device. Finally, if the device requested DMA memory isolation, the RMM invokes the monitor to configure the GPTs and the SMMU to allow the device access to realm memory. Concretely, the RMM calls the monitor to mark the realm VM’s memory as normal world in GPT_d and create SMMU S2 table mappings to all realm VM memory for the device. This sets up all device memory and interrupts with the isolation requested by the device.

We implement an RSI call (`rsi_detach_dev`) for the realm VM to request device detach. When the VM requests this, the RMM soft-resets the device, deregisters its isolated interrupts, and destroys the SMMU S2 table entries. If this is the last device attached to the VM, the RMM also changes GPT_d for the VM’s memory back to realm world. Finally, it marks the device as reclaimable and informs the hypervisor to reclaim the device. At this point, the hypervisor can use normal CCA mechanisms to reclaim the memory. Additionally, we change the hypervisor to force a device detach by killing the realm VM using normal CCA mechanisms to reclaim the device.

7.4 Interrupt Isolation

DEVLORE isolates physical interrupts by moving the GIC configuration to the root world, ascertains the authenticity of virtual device interrupts, and implements a trap-and-emulate mechanism to maintain hypervisor functionality.

DEVLORE’s Interrupt Configuration. To isolate physical interrupts, we move the GIC’s memory-mapped configuration space to root. We implement an SMC interface (`smc_prot_int`) for the RMM to invoke registering interrupts for isolation with DEVLORE to ensure that they always trap to EL3. To do this, in the monitor, we first configure the GIC to trap all Group 0 interrupts to EL3. Then, we change

the monitor to program the GIC entry for the registered interrupts as belonging to Group 0. This process ensures that these interrupts always trap to EL3 (Step 1 in Figure 2(b)).

DEVLORE’s Interrupt Delivery. Further, we need to implement notification mechanisms to the RMM and hypervisor when these registered interrupts trap to the monitor. For this, we allocate a shared data-structure each with the RMM (in the realm world) and hypervisor (in the normal world). Then, in the monitor’s interrupt handler, we perform two atomic writes with the interrupt number into these shared data-structures (Steps 2,3). At this stage, we acknowledge the interrupt if it is an edge-triggered physical interrupts as explained in Section 7.2. We then notify the hypervisor from the monitor’s interrupt handler by sending a software-generated interrupt (SGI number 7) that a new physical interrupt for a realm arrived (Step 4). Finally, we return to the software that was interrupted which resumes execution as normal (Step 5).

In the hypervisor, we implement an interrupt handler for software-generated interrupt number 7 that responds to the monitor’s interrupt notification. In this handler, we check the shared data-structure for pending interrupts and invoke KVM to inject these pending interrupts into the realm VM (Step 6). In the RMM, we implement checks for the virtual interrupts that the hypervisor injects into the realm VMs (Step 7). For this, in the RMI handler, we first read all pending interrupts from the monitor and pick the interrupts that are from devices attached to the realm VM. Then, we implement a queue weighted by the priority of the device interrupts that the realm VM indicated during device attach. We check the list of interrupts that the hypervisor requested to program the vGIC with against our priority list (see Section 5). If the checks pass, we program the vGIC and resume realm VM execution (Step 8,9).

Interrupt Acknowledgment. We need to implement a mechanism to correctly acknowledge level-triggered interrupts such that it does not lead to an interrupt storm. For this, we introduce a new RSI call in the RMM (`rsi_ack_int`). Then, we change the guest kernel to trap on acknowledgments for level-triggered interrupts and invoke this RSI. The RMM then invokes the monitor to acknowledge the corresponding physical interrupt. This guarantees that the physical interrupt is acknowledged after the VM has finished handling and acknowledging the virtual interrupt.

Trap-and-Emulate to Check Hypervisor’s GIC operations. With DEVLORE, the hypervisor cannot access the GIC configuration directly. To allow the hypervisor to program the GIC, we build a trap-and-emulate mechanism (see Figure 5). Concretely, when the hypervisor accesses the GIC’s configuration space in the root world (Step 1), the hardware raises a Granule Protection Fault (GPF) (Step 2). We modify the GPF handlers in the hypervisor to check if the access was to the GIC and invoke a new SMC (`smc_gic_config`) to the monitor to perform the operation (Step 3). We implement checks in the monitor to only allow the programming of interrupts

Table 4: Tested Device.

Device	MMIO	DMA	IRQ	Setup & Benchmark
Keyboard	✓	✗	✓	Run ambakmi keyboard driver, Type n keys
SMMU test	✓	✓	✓	DMA transfers from Rodinia benchmarks
LED	✓	✗	✗	Flash LED
Button	✓	✗	✗	Read button state
Mouse	✓	✗	✓	Run ambkmi mouse driver

that are not isolated in DEVLORE (Step 4).

8 Evaluation

We aim to evaluate DEVLORE to: (a) demonstrate that it can support multiple integrated devices and (b) report the impact of our memory and interrupt isolation.

8.1 Measurement Setups & Platform

Baselines & DEVLORE Modes. We implement two baseline setups to compare the performance of DEVLORE against. Our baselines run a normal world VM (B_n) and a realm VM (B_r), both with a passthrough-enabled hypervisor (Section 7) and an unmodified RMM and monitor. B_n runs a vanilla VM that is attached to device where all memory is in normal world. B_r runs a realm VM that has an attached device where all device memory is in the normal world and not protected. These baselines allow us to compare the performance overheads of executing a realm VM without DEVLORE protections. Then, we measure DEVLORE with memory and interrupt isolation (D_{mi}). For D_{mi} , we run a realm VM with device passthrough and isolate the MMIO and DMA regions in realm memory. We also turn on interrupt isolation for device interrupts. We compare the performance of D_{mi} with B_r to measure DEVLORE overheads. Then, to understand the costs of realm execution we compare the overheads for B_r compared to B_n . This allows us to put into perspective the overheads of D_{mi} over B_n .

Evaluation Platform. We perform our experiments on Arm FVP_Base_RevC-2xAEMvA version 11.20_15_Linux64. The FVP emulates Arm v9 cores with RME support. The FVP is instruction-accurate i.e., it accurately mocks the instructions that the Arm cores execute. We run our experiments on an x86 host with an AMD EPYC 9334 32-Core Processor and 376 GB of RAM. We configure the FVP with 1 cluster of 4 cores and 3 GB RAM. We boot a realm VM with 1 VCPU and 250 MB of RAM. We attach 5 different devices to the realm VMs using DEVLORE’s protection. For each of the devices, we write test workloads (see Table 4) that we use to ensure the correctness and compatibility of DEVLORE. To compare the performance of our different setups, we modify FVP’s Model Trace Interface (MTI) to count the number of instructions and events.

Table 5: Overview of Rodinia Configurations.

App	Domain	Tasks	Problem Size
nn	Dense linear algebra	1	42764
gaussian	Dense linear algebra	3148	1575×1575
needle	Dynamic programming	229	1840
pathfinder	Dynamic programming	5	50000×100
bfs	Graph traversal	2	1840
sradv1	Structured grid	102	502×458
hotspot	Structured grid	5	512×512
backprop	Unstructured grid	2	$262144 \times 16 \times 1$

8.2 Device Workloads

We execute two device representative workloads with each of our measurement setups.

DMA Workload: SMMU Test Engine. To measure the overheads that DEVLORE with memory isolation adds, we run the Rodinia Benchmark suite in the VMs [11]. It performs different DMAs that are representative of a memory-bound workload. Table 5 summarizes the configuration parameters. On the FVP, the SMMU test engine is a device that is programmable and typically used to initiate test DMA accesses to the host memory. To perform the DMA operations from a device for the Rodinia benchmarks, we use the FVP’s SMMU test engine. For this, we hook on the DMA calls from Rodinia and trigger the corresponding DMA operation from the SMMU test engine.

Interrupt Workload: Keyboard. We attach a keyboard to the VMs in each of the measurement setups. The keyboard does not perform DMA but sends frequent interrupts (one interrupt for each key press) and therefore is an example of an MMIO-only device that uses interrupt notification. We use the keyboard to measure DEVLORE’s interrupt protection overheads in D_{mi} . The FVP includes an X11 visualization component that translates keyboard and mouse input on the x86 host to an emulated PL050 PS/2 peripheral on the FVP. To measure the overheads reliably we need to eliminate any effects from typing speed changes during our measurements. For this, we write an x86 helper application that uses X11/Xlib library to control the keyboard input. For each key press, the FVP generates a physical interrupt that arrives in each of our setups, e.g. in the hypervisor for B_n and B_r , and monitor for D_{mi} . Then, the hypervisor injects the interrupt into the guest. In our experiments, we use the instruction tracer to detect when VM acknowledges the interrupt for a keypress and then type the next character. This mechanism ensures that our keyboard workload generates a high frequency of interrupts and stresses DEVLORE’s interrupt isolation mechanisms. In each of the setups we type a fixed string of characters (of length 1000 and 10000) and stop the experiment after the VM acknowledges all interrupts.

Table 6: Benchmarks with PS2 Keyboard. IRQ: # of injected device interrupts, World Switches: # World switches from Root, Realm, or Normal world, Ovh: Overhead w.r.t. B_r .

Experiment		Events		Interface calls (10^3)			World switches (10^3)			Runtime (10^3)	
Keys	Mode	IRQ	RMI	RSI	SMC	Root	Realm	Normal	Instrs	Ovh	
1000	B_r	1000	2.2	0	2.2	4.5	2.2	2.2	58944	–	
1000	D_{mi}	1000	1.8	0	2.8	5.7	2.8	2.8	74286	26%	
10000	B_r	10000	22.3	0	22.3	44.6	22.3	22.3	593707	–	
10000	D_{mi}	10000	18.3	0	28.3	56.6	28.3	28.3	744183	25%	

Table 7: Rodinia Benchmarks with SMMU test engine. DMA: # of DMA transfers. IRQ: # of injected device interrupts, World Switches: # World switches from Root, Realm, or Normal world, Ovh: Overhead w.r.t. B_r .

Experiment		Events		Interface calls (10^3)			World switches (10^3)			Runtime (10^3)	
Benchmark	Mode	DMA	IRQ	RMI	RSI	SMC	Root	Realm	Normal	Instrs	Ovh
bfs	B_r	12	12	158	12	158	316	158	158	5190	–
bfs	D_{mi}	12	12	104	12	116	231	104	128	7036	36%
gaussian	B_r	3155	3155	7791	3155	7791	15581	7791	7791	255839	–
gaussian	D_{mi}	3155	3155	7565	3155	10720	21439	7565	13875	354762	39%
nn	B_r	4	4	73	4	73	147	73	73	2200	–
nn	D_{mi}	4	4	71	4	75	149	71	79	3107	41%
sradv1	B_r	149	149	977	149	977	1954	977	977	22877	–
sradv1	D_{mi}	149	149	906	149	1055	2110	906	1204	30239	32%
needle	B_r	233	233	1007	233	1007	2015	1007	1007	57068	–
needle	D_{mi}	233	233	942	233	1175	2349	942	1408	91241	60%
hotspot	B_r	9	9	95	9	95	190	95	95	4860	–
hotspot	D_{mi}	9	9	88	9	97	193	88	106	7366	52%
pathfinder	B_r	9	9	165	9	165	329	165	165	14914	–
pathfinder	D_{mi}	9	9	151	9	160	320	151	169	24482	64%
backprop	B_r	11	11	510	11	510	1020	510	510	52697	–
backprop	D_{mi}	11	11	933	11	1098	2041	1087	955	93534	77%

Table 8: Platform Boot Instructions (10^3), Ovh: Overhead w.r.t. B_r .

Stage	B_r	D_{mi}	Ovh
BL1	291	291	0.00%
BL2	15246	15634	2.58%
RMM	75534	75538	0.01%
Kernel	112513	115444	2.61%
Total	203578	206907	1.64%

8.3 DEVLORE Performance

We evaluate DEVLORE in different settings to measure the performance costs incurred by our changes.

Platform and Realm VM Boot. Table 8 shows DEVLORE overheads during the different boot stages. In total, DEVLORE adds 1.6% overhead to the platform boot process. 0.97%, 3.74E-4% and 0.49% of this overhead is because of DEVLORE’s GPT setup, SMMU configuration, and GIC configuration changes respectively. This is a one-time cost per platform boot and hence the high overhead can be acceptable.

For a realm VM boot, D_{mi} needs 5.31% and 1068.2% more instructions compared to B_r and B_n respectively. This is expected, as most of the overhead during realm VM boot is from transitioning normal world memory to the realm world. Since this is a one-time setup cost per VM, this overhead may also be acceptable.

Device Attach. Attaching the keyboard incurs 186% overhead for D_{mi} compared to B_n . The overhead includes GPT updates, SMMU configuration, and GIC configuration updates, as well as context switches to the hypervisor for various operations. We observe that the realm VM only needs to incur the cost of the two-GPT setup for the first device. If the VM already has a device attached, DEVLORE only programs the SMMU’s S2 tables for subsequent devices that the VM requests.

Runtime overheads. Table 7 and Table 6 show an overview of the runtime overheads that the keyboard and SMMU test engine incur. During runtime, DEVLORE’s memory protection enables the devices to directly access realm memory and once setup should not add significant overheads to devices

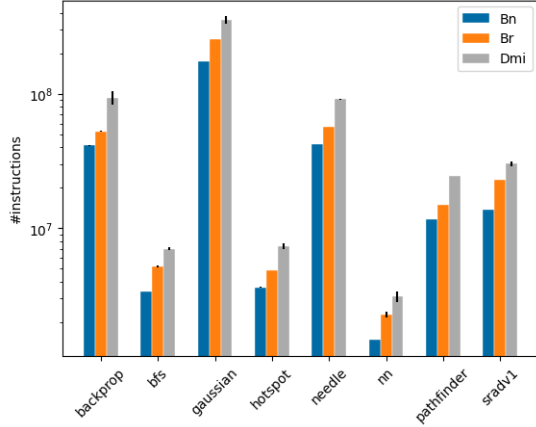


Figure 7: Runtime performance of Rodinia benchmarks with SMMU test engine. Y-axis is the numbers of instructions in log scale, X-axis is the benchmark.

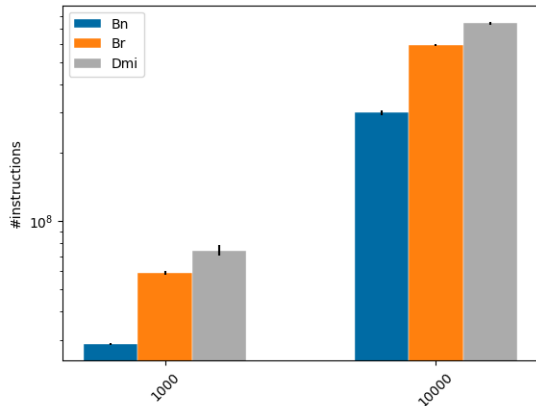


Figure 8: Keyboard benchmark runtime performance. Y-axis is the number of instructions in log scale, X-axis is the number of keys typed.

during runtime. Unlike memory protection, DEVLORE’s interrupt protection adds overheads to device runtime. With DEVLORE’s memory and interrupt protection (in D_{mi}), the keyboard workload is on average 26% slower and the SMMU test engine workload is 50% slower compared to B_r (see Figure 8 and Figure 7). If we compare D_{mi} with B_n , D_{mi} is on average 153% and 112% slower for the keyboard and SMMU test engine. We note that most of this overhead is because of execution in the realm world as evidenced by B_r being on average 102% slower for the keyboard and 43% slower for the SMMU test engine than B_n .

9 Related Work

We discuss the existing body of work on confidential computing support for device.

CCA. There are several recent works on Arm CCA [10, 17, 22, 27, 35, 36, 42, 46, 49]. Shelter isolates userspace applications in the normal world using multiple GPTs with an aim to remove the RMM and reduce the TCB in CCA [49]. Similarly, Cage uses several GPTs to isolate mutually distrusting GPU computation [42]. Both Shelter and Cage reprogram GPCs in the cores and SMMU during runtime incurring context switch overheads because of TLB flushes. In contrast, DEVLORE like Acai only uses two GPTs and does not need to reprogram the GPCs during runtime [36]. Ongoing efforts to verify the RMM and the CCA firmware [17, 27] can be extended to include DEVLORE’s TCB.

Device Protection in Arm. Cage and Strongbox build support for CCA’s realm VMs and TrustZone trusted apps to securely compute on integrated Arm GPUs—without trusting the GPU driver and runtime to reduce the TCB [13, 42]. In contrast, DEVLORE supports attaching a wide array of different integrated devices to realm VMs and configuring features (e.g., DMA protection, interrupt isolation) according to the device’s needs. TrustZone can be repurposed to enable protected access to integrated devices [18, 24–26, 37, 38, 48]. DEVLORE uses CCA for securely virtualizing the devices and passing them through to the realm VMs without the hypervisor’s intervention. Acai connects TEE-enabled PCIe accelerators to realm VMs [36]. Unlike DEVLORE, Acai does not support hotplugging accelerators and requires that the accelerator is connected to the realm VM for its entire lifespan. Arm’s new hardware extension for Device Assignment (RME-DA) allows TEE-enabled PCIe accelerators to directly access realm VM memory [1]. However, these solutions do not apply to integrated devices. This is because CCA blocks all accesses from integrated devices, even if they are TEE-enabled, to realm memory. DEVLORE design can be used to specify an RME-DA equivalent for integrated devices.

Integrated Device Protection in Other Architectures. On RISC-V, Cure modifies the hardware to propagate access control information on the bus [8] and Composite Enclaves assumes platform is untrusted and requires attestation primitives on the devices [34]. Unlike RISC-V, in CCA the Arm platform is trusted so we don’t need these expensive hardware changes or device attestation mechanisms. IOPMP and its extensions, like GPTs in DEVLORE, allow trusted hardware to create different views of memory for devices to allow/deny access [16, 21, 45]. Several prior works for Intel SGX have explored solutions to perform trusted IO with SGX enclaves [14, 15, 28, 29, 44]. This rich body of prior work highlights the need for trusted peripherals for different types of computation. While these works considered trusted IO with physical devices, DEVLORE also virtualizes these devices to connect them to realm VMs with CCA.

Interrupt Attacks. The hypervisor or the host OS can inject to mount side-channel attacks by exploiting unprotected timer interrupts and page-faults [19, 31, 39–41, 43, 47]. Hypervisors can fake CPU exceptions and system calls using interrupts

to arbitrarily trigger victim’s handlers [32, 33]. Similarly, ExpRace exploits kernel race conditions by using malicious interrupt injections [23]. While these works focus on interrupts usually raised by the processor (e.g., page-faults, system calls), DEVLORE investigates the problem maliciously injected device interrupts.

Interrupt Isolation on Arm. For interrupt configuration, DEVLORE protects the GIC by marking it as root. Our monitor and RMM check all configurations and stop other software from directly writing to the GIC to tamper the configuration or trigger arbitrary interrupts. Prior works on TrustZone [13, 18, 24] and CCA [42] achieve this by entirely or partially marking GIC configuration as secure world. However, in our threat model, we do not trust the secure world which can directly program the GIC to maliciously fire interrupts, so DEVLORE marks them as root. For interrupt delivery, like prior works, DEVLORE marks interrupts as belonging to Group 0 (i.e., secure world) to trap them in EL3 [13, 18, 24, 42]. Unlike prior works, DEVLORE is the first work that addresses the problem of securely virtualizing the physical interrupts when injected into the realm VMs.

Interrupt Isolation on Intel TDX & AMD SEV-SNP. Prior works protect user-space enclaves against malicious interrupts that leak register state or side-channel information [9, 12]. In contrast, DEVLORE isolates VM interrupts to ensure execution integrity for both processor and device bound computation. Intel TDX and AMD SEV-SNP offer confidential VMs (CVM) architectures and have considered the problem of protecting interrupts to CVMs. Intel TDX deploys a trusted TD-Module that like the RMM, checks all virtual interrupts injected into the TDX VMs [20]. AMD SEV-SNP allows SEV-SNP VMs to optionally use a paravisor in the VM to filter interrupts from the hypervisor [6, 7]. Like CCA, these filters can only allow or deny all device interrupts. TDX-Connect and SEV-TIO allow devices direct access to CVM memory. DEVLORE’s insights and design can be applied to these architectures but requires a careful security analysis of other assumptions.

10 Conclusion

We present DEVLORE—the first system that allows attaching integrated peripherals to Arm CCA-based VMs. DEVLORE uses a two-GPT design to allow devices to directly access realm world memory but programs stage-two translation tables for MMU and SMMU to limit the memory accesses to the owner VM and attached device. Our novel interrupt isolation design allows the hypervisor to manage the GIC but strictly monitors the interrupt life cycle (configuration, delivery, acknowledgment) in the SM and the RMM. DEVLORE’s delegate-but-check strategy preserves compatibility and performance for wide-range of devices.

References

- [1] Arm® Realm Management Extension (RME) System Architecture. <https://developer.arm.com/documentation/den0129/latest/>, accessed 11.08.2024.
- [2] Confidential Computing on NVIDIA H100 GPUs for Secure and Trustworthy AI. <https://developer.nvidia.com/blog/confidential-computing-on-h100-gpus-for-secure-and-trustworthy-ai/>, accessed 11.08.2024.
- [3] Linux WCNSS driver. https://github.com/torvalds/linux/blob/master/drivers/remoteproc/qcom_wcnss.c, accessed 18.07.2024.
- [4] Microsoft and NVIDIA partnership continues to deliver on the promise of AI. <https://azure.microsoft.com/en-us/blog/microsoft-and-nvidia-partnership-continues-to-deliver-on-the-promise-of-ai/>, accessed 18.07.2024.
- [5] Spurious interrupts. <https://developer.arm.com/documentation/ih0048/b/Introduction/Terminology/Spurious-interrupts>, accessed 18.07.2024.
- [6] AMD. AMD SEV-SNP. <https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf>.
- [7] AMD. Linux SVSM (Secure VM Service Module). <https://github.com/AMDESE/linux-svsm>.
- [8] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stempf. Cure: A security architecture with customizable and resilient enclaves. In *USENIX Security Symposium*, pages 1073–1090, 2021.
- [9] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. Securing interruptible enclaved execution on small microprocessors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 43(3):1–77, 2021.
- [10] Charly Castes and Andrew Baumann. Sharing is leaking: blocking transient-execution attacks with core-gapped confidential vms. 2024.
- [11] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. pages 44–54, 10 2009.

- [12] Ruan De Clercq, Frank Piessens, Dries Schellekens, and Ingrid Verbauwhede. Secure interrupts on low-end microcontrollers. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 147–152. IEEE, 2014.
- [13] Yunjie Deng, Chenxu Wang, Shunchang Yu, Shiqing Liu, Zhenyu Ning, Kevin Leach, Jin Li, Shoumeng Yan, Zhengyu He, Jiannong Cao, et al. Strongbox: A gpu tee on arm endpoints. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 769–783, 2022.
- [14] Aritra Dhar, Ivan Puddu, Kari Kostianen, and Srdjan Capkun. Proximate: Hardened sgx attestation by proximity verification. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy, CODASPY '20*, page 5–16, New York, NY, USA, 2020. Association for Computing Machinery.
- [15] Saba Eskandarian, Jonathan Cogan, Sawyer Birnbaum, Peh Chang Wei Brandon, Dillon Franke, Forest Fraser, Gaspar Garcia Jr., Eric Gong, Hung T. Nguyen, Taresh K. Sethi, Vishal Subbiah, Michael Backes, Giancarlo Pellegrino, and Dan Boneh. Fidelius: Protecting user secrets from compromised browsers. In *IEEE Symposium on Security and Privacy*, 2019.
- [16] Erhu Feng, Dahu Feng, Dong Du, Yubin Xia, Wenbin Zheng, Siqi Zhao, and Haibo Chen. siomp: Scalable and efficient i/o protection for tees. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS '24*, page 1061–1076, New York, NY, USA, 2024. Association for Computing Machinery.
- [17] Anthony CJ Fox, Gareth Stockwell, Shale Xiong, Hanno Becker, Dominic P Mulligan, Gustavo Petri, and Nathan Chong. A verification methodology for the arm® confidential computing architecture: From a secure specification to safe implementations. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):376–405, 2023.
- [18] Friederike Groschupp, Mark Kuhne, Moritz Schneider, Ivan Puddu, Shweta Shinde, and Srdjan Capkun. It's tee-time: A new architecture bringing sovereignty to smartphones. *arXiv preprint arXiv:2211.05206*, 2022.
- [19] Wenjian He, Wei Zhang, Sanjeev Das, and Yang Liu. Sgxlinger: A new side-channel attack vector based on interrupt latency against enclave execution. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 108–114. IEEE, 2018.
- [20] Intel. Intel Trust Domain Extensions (Intel TDX). <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html>.
- [21] Shan-Chyun Ku. What's in risc-v iomp. RISC-V Forum: Security, 2021.
- [22] Mark Kuhne, Supraja Sridhara, Andrin Bertschi, Nicolas Dutly, Srdjan Capkun, and Shweta Shinde. Aster: Fixing the android tee ecosystem with arm cca. *arXiv preprint arXiv:2407.16694*, 2024.
- [23] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. ExpRace: Exploiting kernel races through raising interrupts. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
- [24] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. Secloak: Arm trustzone-based mobile peripheral control. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys*, 18, 2018.
- [25] Wenhao Li, Shiyu Luo, Zhichuang Sun, Yubin Xia, Long Lu, Haibo Chen, Binyu Zang, and Haibing Guan. Vbutton: Practical attestation of user-driven operations in mobile apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '18*, page 28–40, New York, NY, USA, 2018. Association for Computing Machinery.
- [26] Wenhao Li, Mingyang Ma, Jinchun Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tiejian Li. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems, APSys '14*, 2014.
- [27] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and verification of the arm confidential compute architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.
- [28] Hongliang Liang, Mingyu Li, Yixiu Chen, Lin Jiang, Zhuosi Xie, and Tianqi Yang. Establishing trusted i/o paths for sgx client systems with aurora. *IEEE Transactions on Information Forensics and Security*, 15:1589–1600, 2020.
- [29] Hongliang Liang, Mingyu Li, Yixiu Chen, Lin Jiang, Zhuosi Xie, and Tianqi Yang. Establishing trusted i/o paths for sgx client systems with aurora. *IEEE Transactions on Information Forensics and Security*, 15:1589–1600, 2020.
- [30] Tomball Paul Vu. Method and apparatus for controlling interrupt storms, U.S. Patent 7,120,717 B2, Oct. 2006.

- [31] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Capkun. Frontal attack: Leaking Control-Flow in SGX via the CPU frontend. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
- [32] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP. In *IEEE S&P*, 2024.
- [33] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. Heckler: Breaking Confidential VMs with Malicious Interrupts. In *USENIX Security*, 2024.
- [34] Moritz Schneider, Aritra Dhar, Ivan Puddu, Kari Kostainen, and Srdjan Čapkun. Composite enclaves: Towards disaggregated trusted execution. *IACR CHES*, 2022.
- [35] Sandra Siby, Sina Abdollahi, Mohammad Maheri, Marios Kogias, and Hamed Haddadi. Guarantee: Towards attestable and private ml with cca. In *Proceedings of the 4th Workshop on Machine Learning and Systems, EuroMLSys '24*, page 1–9, New York, NY, USA, 2024. Association for Computing Machinery.
- [36] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, Mark Kuhne, Aliberti Aliberti, and Shweta Shinde. Acai: Protecting Accelerator Execution with Arm Confidential Computing Architecture. In *USENIX Security*, 2024.
- [37] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. Trustotp: Transforming smartphones into secure one-time password tokens. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 976–988, 2015.
- [38] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 367–378. IEEE, 2015.
- [39] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.
- [40] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, 2017.
- [41] Stephan van Schaik, Alex Seto, Thomas Yurek, Adam Batori, Bader AlBassam, Christina Garman, Daniel Genkin, Andrew Miller, Eyal Ronen, and Yuval Yarom. SoK: SGX.Fail: How stuff get eXposed. 2022.
- [42] Chenxu Wang, Fengwei Zhang, Yunjie Deng, Kevin Leach, Jiannong Cao, Zhenyu Ning, Shoumeng Yan, and Zhengyu He. Cage: Complementing arm cca with gpu extensions. In *Network and Distributed System Security (NDSS) Symposium*, 2024.
- [43] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.
- [44] Samuel Weiser and Mario Werner. Sgxio: Generic trusted i/o path for intel sgx. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 261–268, 2017.
- [45] Nils Wistoff, Andreas Kuster, Michael Rogenmoser, Robert Balas, Moritz Schneider, and Luca Benini. Protego: A low-overhead open-source i/o physical memory protection unit for risc-v. In *Proceedings of the 1st Safety and Security in Heterogeneous Open System-on-Chip Platforms Workshop (SSH-SoC 2023)*. SSH-SoC, 2023.
- [46] Xiangyi Xu, Wenhao Wang, Yongzheng Wu, Chenyu Wang, Hui Feng Zhu, Haocheng Ma, Zhennan Min, Zixuan Pang, Rui Hou, and Yier Jin. virtcca: Virtualized arm confidential compute architecture with trustzone. *arXiv preprint arXiv:2306.11011*, 2023.
- [47] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P*, 2015.
- [48] Kailiang Ying, Amit Ahlawat, Bilal Alsharifi, Yuexin Jiang, Priyank Thavai, and Wenliang Du. Truz-droid: Integrating trustzone with mobile operating system. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 18*, page 14–27, 2018.
- [49] Yiming Zhang, Yuxin Hu, Zhenyu Ning, Fengwei Zhang, Xiapu Luo, Haoyang Huang, Shoumeng Yan, and Zhengyu He. SHELTER: Extending arm CCA with isolation in user space. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6257–6274, Anaheim, CA, August 2023. USENIX Association.