ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Protecting Accelerator Execution with Arm Confidential Computing Architecture

Master Thesis

Andrin Rudolf Maximilian Bertschi

Tuesday 29th August, 2023

Advisors: Prof. Dr. Shweta Shinde, Supraja Sridhara
Secure & Trustworthy Systems Group, ETH Zürich

Abstract

Confidential Computing is an emerging field in cloud computing that aims to protect sensitive information from potential threats. Hardware-based trusted execution environments shield user data and code from unauthorized access by a malicious tenant on the system. While current implementations focus on process-level abstraction, there has been a trend toward isolating confidential information at the virtual machine level. However, integrating confidential devices often requires specialized hardware or encryption, leading to increased computation overhead and compatibility issues. Arm confidential computing architecture (Arm CCA) enables trusted execution environments on the new generation of Armv9-A processors. It uses a specific set of design choices to enable powerful VM-based isolated execution. However, it lacks support for powerful accelerators connected via PCIe. We propose a system to make confidential PCIe-based accelerators compatible with Arm CCA without requiring hardware changes or encryption. We implement our approach on the Arm Fixed Virtual Platform simulation software, using an escape mechanism to communicate with connected devices. Through evaluation with FPGA and GPU accelerators, we demonstrate that our method incurs lower overhead than encryption-based approaches. Although based on simulations, this work provides insights into the potential benefits of integrating confidential PCIe-based accelerators into Arm CCA.

Contents

Chapter 1

Introduction

Confidential Computing is an emerging field in cloud computing that isolates sensitive information from an untrusted cloud provider's platform. This is achieved by a hardware-based trusted execution environment (TEE) that shields a user's data and code from unauthorized access by a malicious tenant on the system. This may range from a compromised hypervisor to a malicious system administrator. While existing implementations, such as Intel SGX, shield a user's information at the process-level abstraction, there has been a growing trend toward isolating confidential information at the level of virtual machines (VMs) [6, 52, 5, 49]. In cloud deployments, PCIe is commonly employed to link high-performance CPU cores with specialized accelerators such as graphic cards (GPUs). These accelerators typically possess their own dedicated memory, which can be accessed by CPU cores through memory-mapped IO or DMA (Direct Memory Access).

To facilitate confidential devices in trusted execution environments, previous work proposed modifications in device hardware to enable TEE operations. This requires new, specialized hardware such as recent Nvidia GPUs [69] and does not generalize to existing commodity peripherals [55]. Other techniques rely on encryption to secure and integrity-protect all data exchanged with accelerators [85, 90, 55, 50, 53]. This leads to extra data copies and increases the computation overhead. Additionally, it also breaks compatibility with an unprotected implementation because of invasive API changes in software and device-specific encryption logic.

Arm has announced support for confidential VMs in 2021 as part of their confidential computing architecture (Arm CCA) [6]. CCA introduces a new security domain for virtual machines, called *realm*, that isolates confidential VMs from each other and their hypervisor. In a new set of hardware modifications, CCA proposes a mechanism called granular protection checks (GPC) that isolate CPU cores and peripherals at the level of bus transactions and address translations. These mechanisms allow peripherals to securely access

CCA-enabled VMs. Arm currently requires devices to be integrated on the chip (SoC) and augmented for CCA compatibility. However, most cloud deployments depend on more powerful accelerators connected with PCIe.

In a larger scope beyond this thesis [82], we present a system to make confidential PCIe-based accelerators a first-class principle in Arm CCA. We avoid changes to hardware and do not require encryption and extensive data copies. In the existing CCA specification, external devices connected over PCIe can not access realm memory. As a result, we create a safeguarded memory region in Non-secure world, which is guarded by granular protection checks and SMMU[1] isolation. This region can only be accessed by the realm VM and the accelerator.

We prototype our implementation on a publicly available simulation software from Arm, called Fixed Virtual Platform (FVP). FVPs are proprietary executables that simulate complete Arm-based systems, ranging from the CPU and microarchitectual behavior to software components. However, the FVP does not provide a functional interface to connect to PCIe devices, which is instrumental in showcasing the functionality of our approach. Hence, to allow a realm VM to communicate with a functional accelerator, we implement an escape mechanism to communicate with connected devices of the underlying host machine. This allows us to experiment with CCA even though hardware implementations are unavailable in silicon.

We evaluate our method with two existing PCIe-based accelerators, namely an FPGA and a GPU. When compared to native execution, our method incurs an overhead of 34%, with a speedup of 1 958% compared to encryption. These estimates are based on a simulation because no hardware with CCA support is yet available.

1.1 Contributions and Scope

This report narrows down its scope to the following contributions. Our primary emphasis is on design and implementation aspects specifically related to breaking out of the proprietary FVP and interacting with functional accelerators. For a more extensive comprehension of the entire design, we direct readers to the related work in [82].

1. PCIe accelerators in FVP:
    - We design and develop a mechanism to connect functional PCI-based accelerators to the Fixed Virtual Platform (FVP).
    - We apply this mechanism to two specific devices on the underlying host machine, namely an FPGA and a GPU.

[1]The Input-Output Memory Management Unit (IOMMU) on Arm is called SMMU

2. Encryption Reference Implementation:
    - To evaluate the effectiveness of our approach, we develop a reference implementation based on encryption.
    - This implementation serves as a benchmark to compare and assess our approach against existing encryption-based methods.
3. Driver Compatibility Layer:
    - We sketch a prototype of a driver compatibility layer that integrates accelerators without requiring modifications to the current drivers.
    - We evaluate how to model PCIe interactions from the perspective of a realm VM in the FVP's limited PCIe simulation.
4. Benchmarking:
    - And lastly, we instrument and benchmark our approach against different implementation variants with a set of tailored benchmarks for the FPGA and GPU devices.

1.2 Structure of this Document

In Chapter 2, we present background information on confidential computing, specifically Arm CCA. We summarize important aspects of the Arm 64-bit architecture, Fixed Virtual Platforms (FVPs), and dive into more details on confidential device assignments for CCA. In Chapter 3, we elaborate on the design and implementation of a passthrough mechanism to channel DMA and memory-mapped I/O requests in and out of the FVP. We apply this design to two accelerators, a GPU and an FPGA. Next, in Chapter 4, we evaluate the effectiveness of our approach and benchmark it against a method that uses encryption. We present related work in Chapter 5 before we discuss future work in Chapter 6 and conclude our findings in Chapter 7.

Chapter 2

Background

In this chapter, we provide background information on confidential computing and technical details on Arm's confidential computing architecture (Arm CCA). Subsequently, we motivate the need for confidential acceleration by categorizing peripherals into integrated and PCIe- enabled devices and summarize our method for trusted device assignment in Arm CCA. Lastly, we give an overview of Fixed Virtual Platforms (FVPs), Arm's hardware simulator to validate and test software functionalities even if the underlying hardware is unavailable. While writing this report, Arm CCA is actively being developed, and no implementations in silicon are available yet.

2.1 Confidential Computing

Confidential computing is a field of research that focuses on safeguarding data during computation by utilizing a hardware-backend trusted execution environment (TEE). It protects data and code from being observed or altered by privileged software or hardware on the system. These entities may be the operating system or peripherals such as a graphics card (GPU).

Definitions. The Confidential Computing Consortium (CCC) defines a trusted execution environment through a minimum of three desirable properties [33]:

Data Confidentiality. Unauthorized entities on the system cannot view data while it is in use in the TEE.

Data Integrity. Unauthorized entities cannot modify the data while it is in the TEE.

Code Integrity. Unauthorized entities cannot modify the executing code in the TEE.

Unauthorized entities in this definition may be the operating system, devices on the system, system administrators, or anyone with access to the hardware.

Attestation. In addition to providing an execution environment for trusted computing, it is important to establish that the environment is in a trustworthy state. This is referred to in the literature as *attestation*. Without a mechanism for attestation, a malicious system can simulate a tampered TEE and deceive other parties to trust it. We refer to the work by Ménétrey et al. [66] for an overview of attestation methods in TEEs.

Trusted Computing Base. The trusting computing base (TCB) refers to a set of components on a system (software, firmware, hardware) that play a crucial role in ensuring the security of the system. Bugs within the TCB can potentially jeopardize the entire system. Hence, it is of interest to keep the TCB as small as possible to strengthen the system's security properties. In a commodity operating system such as Linux, the TCB includes everything underneath the OS and the Linux kernel itself [81].

VM Isolation. There are various levels of abstraction to implement a trusted environment. They range from fine-grained function isolation to application isolation to more coarse container and virtual machine isolation. We refer to Chapter 5 for a study of related work. As we will see in the next section, Arm implements an approach at the granularity of virtual machines.

2.2 Arm Confidential Compute Architecture

In its upcoming version of the Arm architecture, Arm implements a TEE at the level of virtual machines. These new changes are part of Arm CCA, a set of hardware and software extensions that aim to facilitate trusted execution environments for VMs. To encourage community driver development, Arm develops firmware components openly [11, 21, 19, 20] and makes early drafts of design specifications publicly available. Before introducing CCA, we establish exception levels, stage 2 translations, and security domains in the Arm 64-bit (AArch64) architecture:

Exception Levels. Like most hardware architectures, the Arm architecture provides different levels of protection rings for executing code to enable a mechanism for security and isolation. On Arm 64-bit hardware, we can have up to 4 exception levels, ranging from least privileged user mode EL0 to most privileged root mode in EL3. User and kernel mode functionality is facilitated in EL0 and EL1. The higher privilege modes provide a virtualization layer for a hypervisor (EL2) and a secure monitor mode for trusted execution environments (EL3). However, these may not be available on all

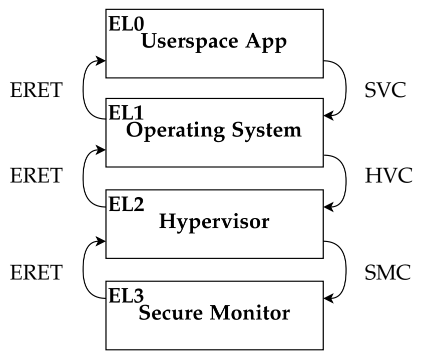

Figure 2.1: Exception Levels in Arm 64-bit architecture and instructions to context switch between them.

Arm processors. The diagram in Figure 2.1 depicts the exception modes and instructions to jump between them. The architecture does not specify which specific software components run at each exception level [24].

Stage 2 Translations. Stage 2 translations are a set of translation tables in the address translation of the Memory Management Unit (MMU). They allow a hypervisor to control the memory access of a VM in hardware, ensuring that the VM can not break out of its sandbox. Specifically, they control which memory the VM can access and which memory is mapped into the address space of the VM [23]. AArch64 has optional support for stage 2 translations.

Security States. To further compartmentalize code running on a system, the AArch64 architecture defines multiple security states. A security state defines which exception levels and memory areas can currently be accessed. Most Cortex-A processors implement two security states, namely *non-secure* and *secure* state.

The term *world* in AArch64 terminology refers to the combination of a security state and a physical address space. As such, a Processing Element (PE)[1] running in non-secure world has access to the Non-secure physical address space. A PE running in secure world can access both secure and non-secure physical address spaces [24].

New Worlds. Arm CCA introduces support for two additional security states, namely *realm* and *root* state. Similarly to the secure world, a PE running in realm world can access its own physical address space (PAS) and the non-secure PAS.

[1]Arm refers to a CPU as a PE, to refer to everything with its own program counter and can execute code.

Security State	Non-Secure PAS	Secure PAS	Realm PAS	Root PAS
Non-Secure	✓	✗	✗	✗
Secure	✓	✓	✗	✗
Realm	✓	✗	✓	✗
Root	✓	✓	✓	✓

Table 2.1: Physical Address Space (PAS) accessible in each security state.

A PE is in the root world when it is running in EL3. It then has access to all available PAS: non-secure, secure, realm, and root. Table 2.1 summarizes the access matrix.

Arm CCA Architecture Before the introduction of Arm CCA, virtual machines on Arm had to trust the underlying hypervisor that managed them. This posed the risk of confidential data leaks because no trusted execution environment existed. Armv9-A architecture introduces a new type of isolation for a virtual machine called *realm*. Code and data running in a realm VM are protected from being accessed and modified by the hypervisor.

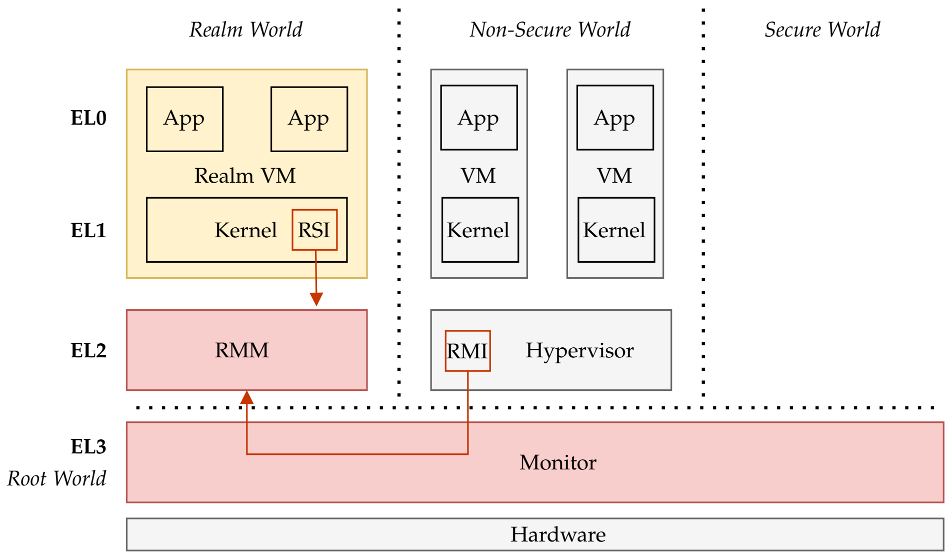

Figure 2.2: Arm CCA architecture, showing a setting with a realm VM (yellow) and two ordinary VMs (gray). Software TCB for a realm VM in red. RSI and RMI are interfaces implemented in the RMM that bridge realm VMs and the hypervisor.

Although the hypervisor manages scheduling and resource allocation, it cannot access the realm VM once it is created. This avoids the need to duplicate hypervisor functionality in the realm world, keeping the trusted computing base (TCB) small.

To allow the isolated execution of a realm VM, a new software component called the realm management monitor (RMM) is created, running in realm world in EL2. The RMM is trusted by the realm VM and acts as an intermediary between the hypervisor and the realm VM.

The hypervisor interacts with the RMM through the realm management interface (RMI) to manage the realm VM. The RMM uses the realm service interface (RSI) to provide services to the realm VM.

Isolation between realm VMs is accomplished with stage 2 translations which the RMM sets up and manages [24]. In Figure 2.2, we present the high-level architecture of Arm CCA.

Granular Protection Checks. At the core of Arm CCA lays a set of new hardware checks, so-called granular protection checks (GPC). Recall from the previous section that RME provides four physical address spaces. To guarantee isolation enforcement for all environments, the Memory Management Unit (MMU) implements additional access controls. These controls are applied to all address translations of CPU cores and device memory accesses.

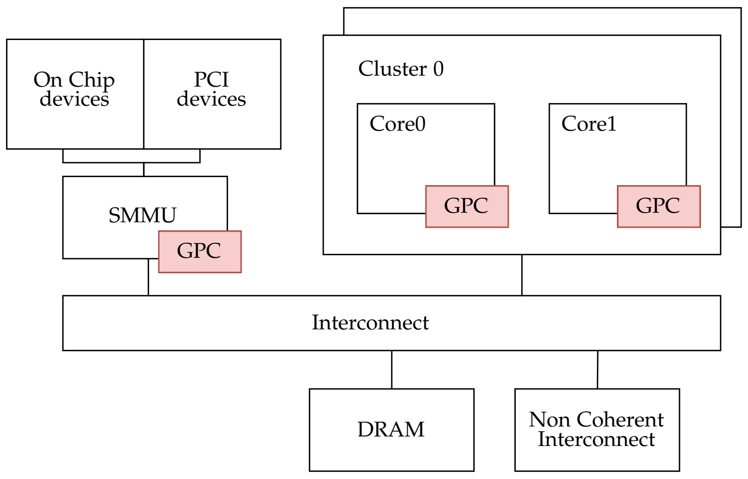

Figure 2.3: Overview of Granular Protection Checks (CPC) to enforce memory isolation between different security states. SMMU in Arm terminology is the equivalent of IOMMU on Intel.

After a page table walk, the MMU checks access permissions in special tables, so-called granular protection tables (GPTs). GPTs reside in root world memory and are managed in EL3 by the monitor software.

The monitor software has the ability to dynamically update the GPT, enabling the movement of DRAM memory across different security states. Whenever an entry in the GPT is changed, the GPCs are synchronized and may trigger the flushing of stale states [24].

We further note that the monitor can maintain multiple GPTs in memory and choose which one to use for each GPC. In Figure 2.3, we highlight these checks in a simplified architecture diagram.

2.3 Confidential Peripherals

Arm CCA does not allow devices to access realm world memory. This is similar to other TEE architectures like Intel SGX and AMD SEV.

When analyzing accelerators on a system, we typically categorize devices into two groups: Integrated accelerators and PCIe accelerators. Arm CCA currently requires devices to be on-chip to be compatible with confidential computing.

Integrated Accelerators. Integrated accelerators are embedded in the system-on-chip (SoC) and do not have their own memory. They share the same main memory with the CPU cores. A bus-level access control mechanism used for CPU isolation can be applied to these accelerators as well. This approach is conceptionally simpler than PCIe device access and has been implemented in Arm TrustZone [38] and RISC-V [26, 79]. Popular examples of on-chip devices are Immortalis and Mali GPUs and Mali NPUs (Neural Processing Units).

PCIe Accelerators. In cloud deployments, PCIe is commonly employed to link high-performance CPU cores with specialized accelerators. These accelerators typically possess their own dedicated memory, which interacts with CPU cores through memory-mapped I/O and DMA (Direct Memory Access) requests. However, it is essential to note that bus-level access control mechanisms like Arm TrustZone [7] or RISC-V PMP [62] are not designed for these standalone accelerators. As a result, existing solutions rely on encrypted communication between VMs and the accelerators. The VM encrypts data in software and sends it to the accelerator, which decrypts it using accelerator-specific logic. Similarly, results are encrypted by the accelerator and transferred to CPU memory, requiring the VM to copy and decrypt the data.

This extensive data copying and the overhead of encryption create a potential for optimizations and improvements. In [82], we dive into software modifications to enable secure device access for a realm VM without any hardware changes to Arm CCA or accelerators.

Our Approach. As mentioned earlier, the current state of Arm CCA prevents external devices connected via PCIe from accessing realm memory. Therefore, our approach outlined in [82] aims to achieve its design objective without changes to the hardware design of Arm CCA. To enable secure device assignment, we set up a shared memory region in unprotected Non-secure world memory. Without further safeguarding, this region has no integrity and confidentially guarantees. We then program the granule protection checks (GPC) to guard the region such that it can only be accessed from the realm and root world. We call this new region *Non-secure protected* memory. From the view of a PCIe device, this region still looks like normal-world memory. Hence it can be accessed according to Arm CCA. To isolate that memory region from distrusting realm VMs, we further adapt the RMM's stage 2 translations. This safeguards the region at the CPU level. However, these measures do not protect from malicious devices, which may DMA into non-secure world memory directly. To adequately safeguard Non-secure protected memory from a malicious device, our method further involves programming the SMMU[2] and its stage 2 translation tables.

2.4 Arm Fixed Virtual Platform

Arm Fixed Virtual Platforms (FVPs) are pre-built executables that simulate complete Arm-based systems, including processors, devices, and software components. They are available as proprietary binaries and target the validation of software functionality when the underlying hardware components are not yet manufactured. They have become integral in computer architecture research and software development to conduct early-stage testing and simulate complete Linux and Android systems. ARM offers a range of pre-built FVPs, including `FVP_Base_RevC-2xAEMvA`. This version is actively used to develop and prototype CCA software. While not completely Armv9 compliant, it emulates two AEMv8-A clusters of four Armv8 cores and has initial Realm Management Extensions (RME) support [8].

Fast Models. Fast Models is a term by Arm that refers to the software simulation of an individual hardware component, such as a Cortex-A53 model for a CPU. FVPs are binaries integrating different Fast Model components to build a complete system. In Figure 2.4, we highlight the various Fast Model components built into Base Platform RevC, the Fixed Virtual Platform (FVP) version we use in the scope of this work.

Virtual Machine Managers. Comparing FVPs with a virtual machine manager (VMM) such as QEMU [77] or VirtualBox [73], we note that FVPs are

[2]SMMU is the equivalent of the IOMMU on Intel architecture.

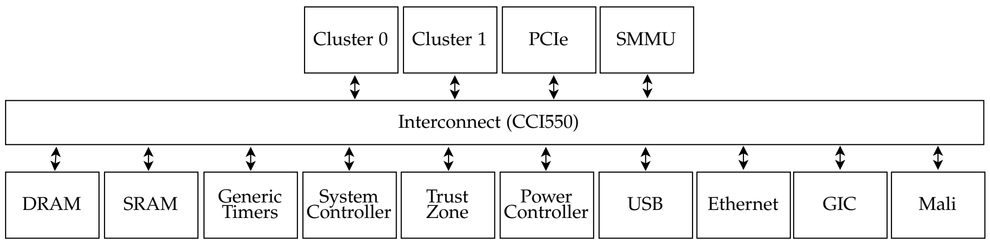

Figure 2.4: Overview of Fast Model components in FVP_Base_RevC-2xAEMvA

more accurate in simulating the underlying hardware as they simulate the CPU and microarchitectural behavior. In contrast, a virtual machine monitor abstracts away everything underneath its supported instruction set architecture (ISA). As we are interested in benchmarking hardware functionalities still under development in the Armv9 specification, we can not use a VMM to prototype our design.

Peripherals. While FVPs simulate different peripherals on the system, many simulated devices only expose their expected interface and are implemented as non-functional stubs. For instance, `FVP_Base_RevC-2xAEMvA` contains a Fast Model component `BasePlatformPCIRevC`, which simulates a limited implementation of the PCIe standard. The bus incorporates two dummy devices that announce themselves as virtio block devices [8]. We can configure the size of their base address registers (BARs) and access their configuration space. Nevertheless, these PCI devices are solely stubs and do not simulate a particular accelerator, such as a graphics card or a field programmable gate array (FPGA). Hence, we cannot use the simulated devices in the FVP to demonstrate functional accelerator interactions.

Chapter 3

Design & Implementation

This chapter delves into the design and implementation of PCIe peripheral device access on the Fixed Virtual Platform. Throughout this report, when discussing different implementation variants and referring to *Our Solution*, we refer to an implementation built on the high-level architecture motivated in Chapter 1. For a more comprehensive understanding of the implementation described earlier, we direct readers to the related work [82]. The author of this report is coauthor in [82]. For the remaining sections of this document, we focus on implementation and design aspects specifically related to breaking out of the FVP and interacting with real accelerators. These mechanisms were instrumental in showcasing the practicality of the aforementioned implementation.

We first elaborate on a mechanism to break out of the proprietary FVP. We then discuss in Section 3.3 methods to implement memory-mapped I/O and DMA capabilities in the Fixed Virtual Platform and the underlying host machine. We apply these methods to a GPU and an FPGA in Sections 3.4 and 3.5. To compare the effectiveness of our solution with an encryption approach, we detail the design of an encryption method in Section 3.7. The remaining parts of this chapter are dedicated to a driver compatibility layer and how we model PCIe interactions from the perspective of a realm VM in the FVP's limited PCIe simulation.

3.1 Peripheral Access and Escape Mechanism

As we will see in the next sections, we aim to support two accelerators and a set of standard benchmark suites. Based on these goals, we focus on supporting functional GPGPU (general purpose GPU) programming on the FVP and interactions with an FPGA accelerator. We first establish different design approaches toward a general direction of implementation before we introduce a general-purpose framework to channel information in and out of the Fixed Virtual Platform (FVP).

3.1.1 Design Evaluations

We consider the following avenues for device interactions on the FVP.

Full Device Simulation. Instead of bringing a device into the FVP, we can simulate an accelerator completely within the FVP. For the GPU, several research projects are available which implement a GPU simulation in software [46, 84]. With this approach, we build an adapter that integrates the simulation software into the operating system on the FVP such that it appears as a GPU device for a GPGPU benchmark program. On the one hand, complete virtualization is likely to be slow, given that it has to run within the FVP on an x86 host machine. On the other hand, a pure software simulation does not have dependencies on the host machine and can easily be ported and reproduced on other systems. To our knowledge, no general-purpose software simulations are available for FPGA accelerators.

Non-Functional Device Stub. An approach with a device stub integrates a register or memory interface into the FVP. This interface aims to be compatible with an existing device driver. The NoMali stub GPU model [37] is an example of this approach. They expose a compatible register interface to work with the Linux Mali driver stack without simulating a Mali-based GPU. This method would not create correct benchmark results as no real GPU rendering and computation are involved. The interface solely returns garbage output. The authors state that NoMali achieves an accuracy in CPU benchmark metrics within 5% compared to a detailed GPU model [37]. While this approach is lightweight and integrates well with an existing driver stack, it can not be used for a GPGPU benchmark suite because it provides no functional computation. If we target a method towards a device stub, we implement an Nvidia hardware stub that exposes the same driver-facing API as an Nvidia card. Nevertheless, Nvidia GPUs on the PCI bus are inherently more complex than an SoC (System on a Chip) peripheral like a Mali GPU [56].

Host Peripheral Passthrough. Since the underlying host machine can access real accelerators, we aim to side-load them into the FVP in this approach. For this to work, we need a reliable method to break out of the FVP's hardware abstractions. If we aim for a design towards this approach, a central issue is where precisely in the call stack we want to escape to the x86 host and what level of abstraction we want to expose to the FVP. Arm provides for its paying customers a media component bridge, which allows the existing Mali GPU component on the FVP to do correct functional computations [16]. However, this project is specific to the GPU, proprietary, and not available to us.

We prioritize functional correctness and a design that is not restricted to the GPU and can be applied to different types of accelerators. Based on this reasoning and exploration, we opt for an approach towards a *Host Peripheral Passthrough* method.

3.2 Page Fault Escape Mechanism

To facilitate an escape to the host, we suggest a broader framework to establish a communication channel between the FVP and its host machine. This method uses a set of x86_64 kernel patches and a custom page fault implementation.

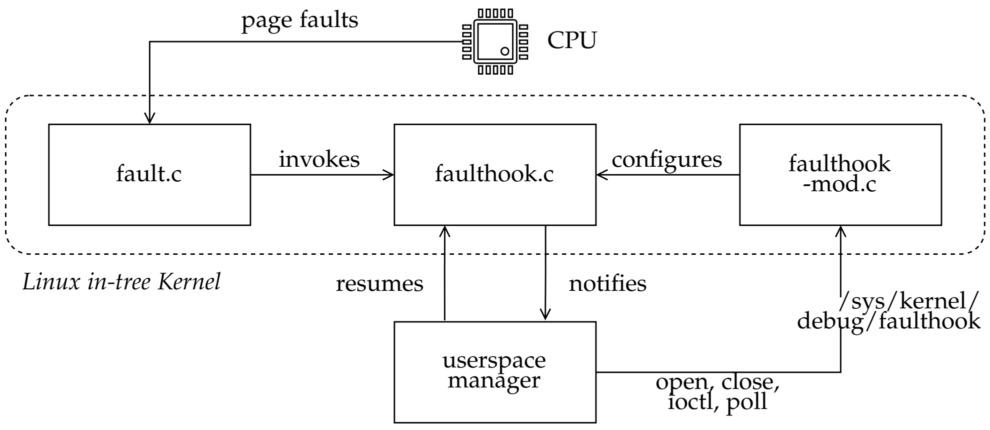

Figure 3.1: Architecture of in-tree kernel changes to enable custom page faults on x86.

3.2.1 Introducing Faulthooks

With the term *faulthook*, we refer to a custom page fault and an associated callback in userspace. During a custom page fault, the faulting application is paused and remains descheduled until the callback in userspace explicitly resumes the application. We implement support for faulthooks with a set of in-tree patches in the Linux kernel. Figure 3.1 visualizes the designed kernel changes. The faulthook implementation in `faulthook.c` adds support for custom page faults in the kernel.

If the CPU traps because a page fault occurred, the kernel's page fault implementation (`fault.c`) can dispatch page faults to `faulthook.c`. A userspace facing `faulthook-mod.c` exposes a character device for configuration purposes. With it, a manager in userspace can register a callback and be notified if custom page faults occur.

Userspace Facing API. We expose a DebugFS character device to userspace to interact with faulthooks.

```
int fh_enable_trace(unsigned long address, unsigned long len,
                    pid_t pid);

pthread_t *fh_run_thread(fh_listener_fn callback, void* ctx);
```

A userspace manager can register a custom page fault by specifying the target's process id and an address range (virtual addresses). It then can schedule a worker with a custom callback which is invoked whenever the target processes access an address in the given range. Within this callback, the manager can manipulate the behavior of the target application and control its scheduling as desired. For instance, it can communicate with an accelerator and re-schedule the target application once the accelerator result is available.

Faulthook Page Fault Handler. We allow faulthooks to occur on a write-only or read-and-write basis. As such, we clear the write or present bit in the page table entries (PTEs) associated with the address range in the target application. We implement faulthooks based on the single stepping feature of the x86 architecture. Single-stepping is a debugging feature that allows us to execute instructions on the CPU one at a time [35].

Upon a faulthook page fault, we trap into kernel space and disarm the faulting PTE by setting it to present and writable. Subsequently, we de-schedule the trapping process and hand over control to the registered userspace manager process. Once the manager re-schedules the target application, we enable single stepping and resume the target application. With single-stepping enabled, the CPU executes a single instruction before generating a trap interrupt. In the trap handler, we then arm the faulted page again and disable single-stepping. This concludes a single faulthook.

To enable single-stepping, we reuse some of the functionalities implemented in `kmmio` subsystem of the Linux kernel.

Limitations. To properly catch a single-stepped instruction, we must pin the target application to a single core. Since we single-step solely a single instruction, this causes not a big performance overhead. For simplicity of our implementation, we keep the target application pinned to the same core. As a result, the target application can no longer exploit parallelism on the host. While we did not benchmark this behavior, we did not observe a significant decline in performance for the Fixed Virtual Platform (FVP).

Another limitation is that we can single-step only a single target application per core, and that application can not use single-stepping debugging features

```c
int fh_do_escape(fh_ctx_t *fh_ctx, int action)
{
    /* increment nonce and add context to escape */
    unsigned long nonce = ++fd_ctx.fh_nonce;
    fh_ctx->fh_escape_data->turn = FH_TURN_HOST;
    fh_ctx->fh_escape_data->action = action;

    /* serialization of instruction stream */
    asm volatile("dmb sy");

    /* escape to the dark side ... */
    fh_ctx->fh_escape_data->nonce = nonce;

    /* and we are already back here */
    // ...
}
```

Listing 1: Excerpt of escape routine running on AArch64 Linux on the FVP after escape channel is established. Error handling and locking omitted.

itself. However, these concerns are not an issue for the FVP because it does not use debugging features during normal operations.

3.2.2 Escaping the FVP

With a general purpose framework in place, we now address how to escape out of the Fixed Virtual Platform (FVP). Based on our observations, we noted that by deactivating the CPU cache hardware components on the FVP, any write to memory within the FVP promptly becomes accessible within the memory of the underlying Linux host process. As our focus does not lie on simulating microarchitectural cache behavior, we can disable caching simulation without encountering uncertainties.

Listing 1 demonstrates the escape in a code snippet, whereas Listing 2 details the underlying data structure between the x86 host and FVP. In this setting, we mark the memory range of a `nonce` field as escape memory. Any write to it invokes a faulthook page fault.

Escape Mappings. One of the missing pieces to escape to the host is to agree on a common escape buffer between x86 and the FVP's emulated DRAM. When we allocate memory on Linux within the FVP, we lack knowledge about the specific location within the host process where this memory is situated. Hence, we aim to build a mapping function f that maps a physical address on the emulated FVP's DRAM to a virtual address on the underlying x86 host.

```c
struct faultdata_struct
{
    volatile unsigned long nonce; // any write triggers escape
    unsigned long turn;           // turn: host or guest
    unsigned long action;         // type of escape
    unsigned long data_size;      // payload size
    char data[0];                 // payload
};
```

Listing 2: Data structure to facilitate an escape. Any write to the nonce triggers an escape to the x86 host. A variable-size payload contains the exchanged data.

$$f : paddr_{FVP} \rightarrow vaddr_{x86} \tag{3.1}$$

To build such a function, we employ a similar approach to that of cheat engines. In the initial stages of the FVP Linux boot, we systematically examine all unoccupied memory pages and assign them a magic value along with their corresponding physical address. On x86, we scan the FVP's address space for magic values and build a mapping between the identified physical addresses and their corresponding virtual addresses in the host process.

Note that we can only build such mappings for unoccupied memory because building a mapping requires us to write to memory. This can potentially destabilize the kernel if that memory is already occupied. In a later phase of the kernel boot, we ensure that unidentified pages are not passed to the buddy allocator and are not made available to the rest of the system. The lost memory lies in the order of 100MB, which is a modest quantity. With this mapping function in place, we can now allocate memory within Linux on the FVP and identify that memory on the x86 host.

Memory Alignment. Another matter that requires attention is the memory alignment of the FVP's emulated DRAM memory. Note that the FVP is a proprietary software product and is not designed to break out its simulated hardware abstractions. As a consequence, the DRAM is not properly page aligned on the x86 host and appears at various places in heap memory. This is problematic because non-contiguous and non-page-aligned memory complicates DMA and I/O memory mapping.

We studied the FVP's proprietary memory allocator and reroute all memory allocations for DRAM memory to a dedicated place in its address space. Thereby we ensure page alignment and continuity. The implementation uses explicitly preloading with the `LD_PRELOAD` trick [74] and custom glibc allocator functions.

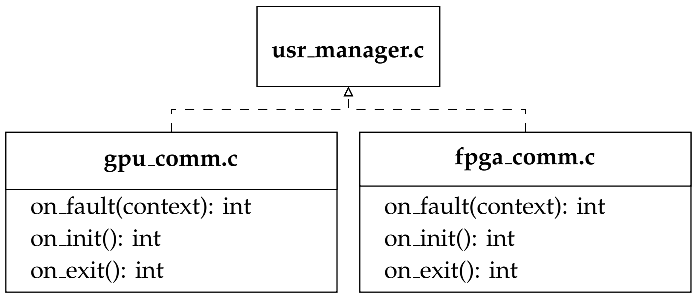

Figure 3.2: Two userspace manager backends for both of our accelerators, a GPU and an FPGA.

Escape through File Sharing. Before we decided on an approach with page faults, we investigated available hardware components on the FVP. One component of interest is the virtio 9P[1] (Plan 9 Filesystem Protocol) component, which facilitates file sharing between the x86 host and an operating system running on the FVP. We prototyped an implementation based on a custom FUSE file system on the x86 host, the 9P hardware component on the FVP, and 9P drivers in the Linux kernel. 9P is based on TCP/IP and involves much marshaling and data copies. This raised concerns about its scalability, which is why we did no further develop this approach.

3.2.3 Userspace Manager for FVP

With a framework for a reliable communication channel in place, we can now address the x86 component. Throughout this report, we refer to the x86 counterpart as the *Userspace Manger*. Its job is to gap all device functionalities not available within the FVP. We implement a general framework for a manager in userspace and derive two implementations from it: one for a GPU accelerator and another for an FPGA. Figure 3.2 depict their life cycle methods. To avoid memory copies on the escape buffer, we use PTEditor [65] and map the escape buffer into the address space of the userspace manager.

3.3 Host Peripheral Passthrough

With the groundwork laid out to escape out of the FVP, we now proceed to address technical details in interacting with host devices. For this, we first focus on exploring methods to map device memory into the FVP before we introduce techniques to enable direct memory access (DMA). Taking into account the code complexity of existing device drivers, we suggest a range of modifications that can be implemented with or without patching the host driver.

[1]FVP_Base_RevC_2xAEMvA.bp.virtiop9device

3.3.1 Overview Memory-Map

Before we investigate how to map accelerator memory into the FVP, we first establish how the Linux kernel builds memory mappings. By memory mapping a device, we refer to the process of mapping a range of userspace addresses with device memory. Consequently, whenever a program reads or writes to these addresses, the CPU will retrieve the data from the device's memory rather than DRAM. Changing a range of userspace addresses involves modifying a program's page table entries and flushing the translation look-aside buffer (TLB). The kernel offers abstractions for common device driver tasks, alleviating the need to manipulate page tables directly. Nevertheless, as we will explore in the following section, there are specific tasks where the manual modification of page tables becomes necessary.

When a user program calls `mmap` to map device memory into its address space, the Linux kernel creates a new VMA (virtual memory area) to represent that mapping. The kernel then invokes the device driver that implements the `mmap` method to complete the creation of the VMA. When implementing `mmap` in a device driver, there are two approaches to create VMA mappings [34]. The first approach uses `remap_pfn_range` to create new page tables eagerly, while the latter approach uses page faults and the `nopage` method to map memory lazily. The former method is more straightforward and sufficient for most use cases. As we will see in a subsequent chapter, we use `remap_pfn_range` to implement memory mapping for the FPGA accelerator. GPU memory mappings are more complex and need more flexibility. Hence, they make use of the `nopage` method.

3.3.2 Memory-Map with Patching the Host Driver

In the case of a small device driver, we suggest implementing a patch within the x86 driver code to facilitate memory mapping. This approach is effective for the FPGA driver because it uses the simpler `remap_pfn_range` function. However, when it comes to GPU drivers, we advise not to patch the driver itself. This is primarily because GPU drivers rely on the DRM subsystem (Direct Rendering Manager), and modifications in DRM can have unintended side effects on other drivers on the system.

In the design outlined in Figure 3.3, we present our approach to map device memory using host driver patches. The figure showcases two sections, the top and the bottom, both depicting the same sequence of events. The top section emphasizes the interactions occurring within the FVP, whereas the bottom section highlights the x86 events. We will now list and describe the steps involved in mapping device memory. The numbering ① to ⑦ is referring to Figure 3.3.

A memory map request ① originates from a userspace application on the

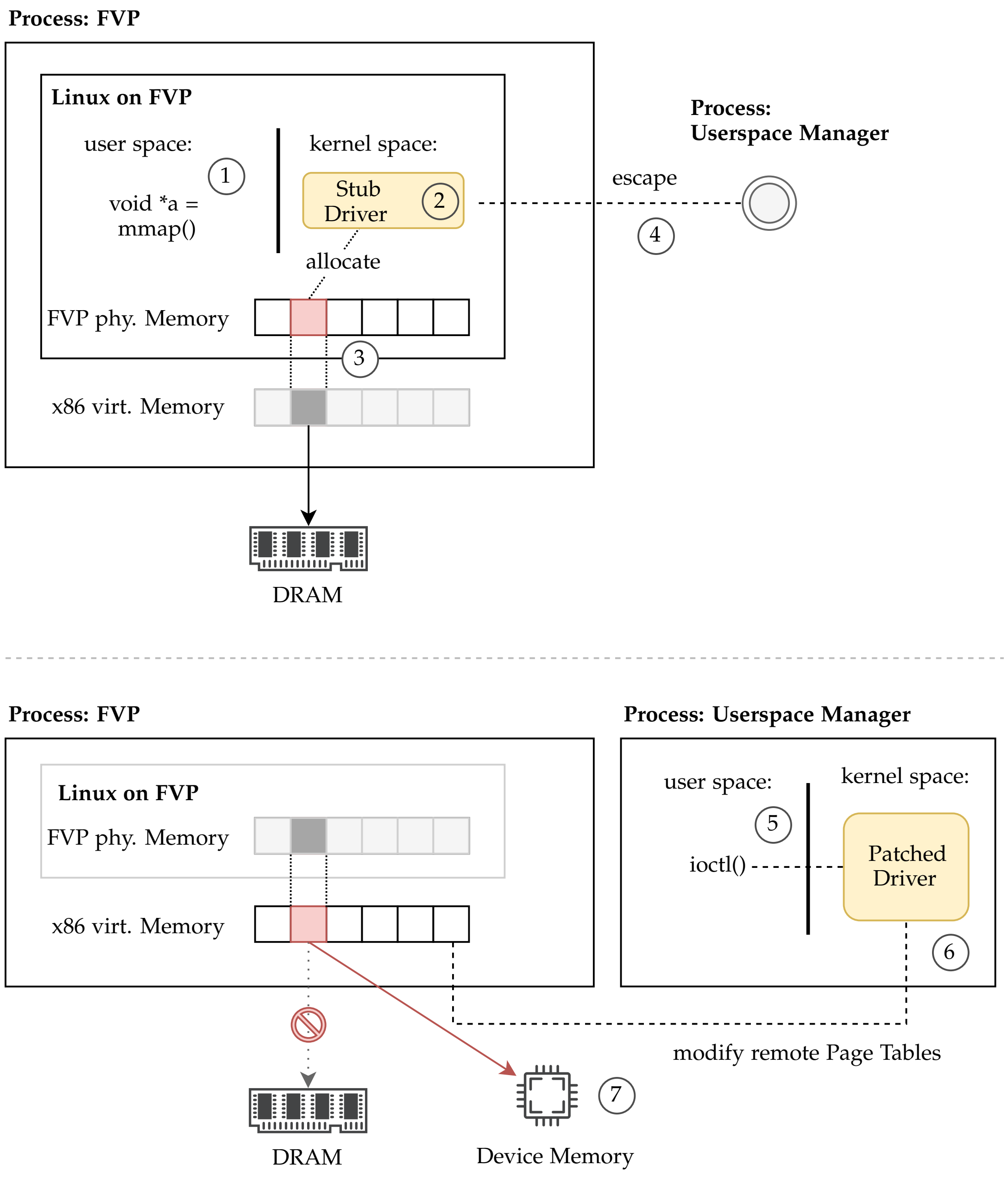

Figure 3.3: Memory-Map design with patches in x86 host device driver. A patched device driver (6) creates a memory mapping on behalf of the FVP. Careful memory alignment ensures that the memory mapping appears at the right location within the emulated FVP memory.

FVP. A program calls `mmap` on a file description associated with a device driver. The Linux kernel creates a new VMA (virtual memory area) to represent the new memory mapping and invokes the device driver to fulfill the request. Both stub ② and host ⑥ drivers implement the same userspace-facing API. However, the stub driver is aware of its lack of a real accelerator. Subsequently, the stub driver allocates memory from DRAM and maps it into the newly created VMA using the `remap_pfn_range` function. To ensure alignment between memory pages on the FVP's AArch64 hardware and memory addresses within the x86 Linux process, as discussed in Section 3.2.2, we have modified the FVP's host memory management ③. As such, we can associate the FVP memory page with a memory page on the x86 host. The stub driver employs the host escape ④ and invokes the userspace manager.

On the x86 side, we now reformulate the request and call a custom ioctl[2] command ⑤ on the host driver. The host driver contains a patch set to implement an additional ioctl command. This command replaces the current memory descriptor with the memory descriptor of the FVP. As a result, the device driver can process the request on behalf of the FVP instead of the userspace manager ⑥. We create a new VMA, and manually call the original `mmap` implementation of the device driver to map device memory into the FVP's host process ⑦. By ensuring memory alignment and proper cache control, the device memory is correctly mapped into the FVP and becomes accessible from userspace ①.

Creating a new VMA. One of the implementation challenges arises in the handling of VMAs (virtual memory areas). Normally, when using `mmap` to request a memory mapping, the kernel creates a new VMA and invokes the `mmap` handler on the driver. However, in our case, we need to modify an existing mapping instead.

To be more specific, we need to change the page table mappings on the heap VMA of the host process of the Fast Virtual Platform (FVP). The heap VMA contains all the FVP's DRAM. The FVP internally uses `malloc` to allocate memory for its emulated DRAM. To ensure reliable behavior when interacting with device memory, we also need to mark the mappings as non-cacheable.

In Linux, caching behavior for an address range is managed at the VMA level. Since we are creating a new device mapping within the heap VMA, we cannot mark the entire process heap as non-cacheable. We have observed that the kernel automatically splits VMAs when changing the protection flags associated with an address range in a VMA. For security reasons, a process's heap is always marked as non-executable by default. Hence, to facilitate the

[2] Syscall for device-specific I/O operations

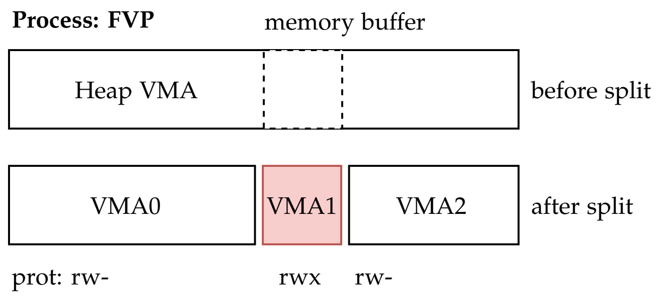

Figure 3.4: Splitting an existing VMA. Protection flag trick: Change permission flags on a subrange of VMA such that the kernel is forced to split VMA. As a result, all of our device mappings are read, write, and executable.

split, we mark the memory range as executable and receive a new VMA in the correct size, which we then map as device memory.

Since the maximum number of VMAs allowed in a process is constrained in Linux, we must change the limit for the FVP. Figure 3.4 summarizes the VMA split.

3.3.3 Memory-Map without Patching the Host Driver

To address the challenges posed by a complex device driver, especially one that employs lazy mappings using the `nopage` method, we propose an alternative memory-map approach that does not require modifying the device driver's code. In Figure 3.5, we introduce the architecture of this approach. While many aspects are analogous to the aforementioned method, we highlight some of the key differences.

Instead of applying a patch set to the device driver itself, we build a *Fixup Helper* kernel module. The following numbering ① to ⑥ is referring to Figure 3.5.

Upon memory map request on the FVP, we create a RAM mapping as previously. We then escape the FVP's sandbox ① and create a memory mapping in the memory descriptor of the userspace manager (② and ③). This is comparable to calling `mmap` in the userspace manager directly, even without the FVP. Subsequently, we need to ensure that the newly created mappings are also applied to the FVP. For this, the *Fixup Helper* modifies the FVP's address space ⑤. It manually removes the existing mappings to RAM and replaces them with the mapping to the device memory. This process involves manually updating the FVP's page table and flushing its TLB. As a result, we have two mappings in two different processes, both mapping to the same device memory ③.

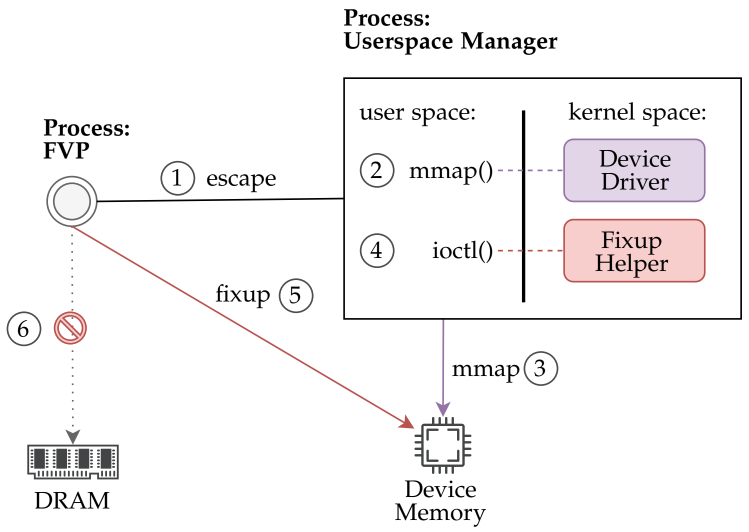

Figure 3.5: Memory-Map without host driver patches. A fixup helper kernel module duplicates the mappings from the userspace manager process to the FVP (red).

When the device driver uses the `nopage` method, the helper module is required to eagerly page fault all mappings. This comes with a penalty in performance but ensures that all device addresses are resolved when manually updating the FVP's page tables. As this approach involves direct modifications to the page tables, caution must be exercised to prevent instability in the x86 host system and to ensure proper cleanup of the mappings when they are unmapped.

We implemented *Fixup Helper* as a header-only userspace library and a kernel module. For page table modifications, we utilize the open-source framework PTEditor [65].

3.3.4 Overview DMA

DMA, or Direct memory access, is a method that allows an accelerator on the system to directly transfer data to and from main memory without the need to involve the CPU. We can transfer data in two ways, either synchronously initiated by software or asynchronously started by hardware. Another relevant distinction is the type of mapping we create. We generally distinguish between consistent and streaming mappings. The following definitions are taken from [34].

Synchronous Transfer. In the example of transferring data from the device, a userspace program reads on a file descriptor associated with the device driver. This task is blocking: The driver creates a DMA buffer and instructions the hardware to write to it. The hardware then reads from the buffer and informs the device driver with an interrupt upon completion. The driver's interrupt handler acknowledges the completion and resumes the userspace program, which then can read the data. This sequence matches well with our escape mechanism as the request originates from a userspace program on the FVP.

Asynchronous Transfer. In an asynchronous transfer, the device interrupts the driver with new data. The driver allocates a new buffer and instructs the hardware to write to it. Upon completion, the driver is again interrupted and can dispatch the data to any relevant process interested in the data. Our escape mechanism is not compatible with this approach due to its asynchronous nature. If we intend to enable asynchronous transfer, we must transform it into a polling-based method.

Coherent Mapping. Coherent mappings have a time spawn across a single transfer. They must be simultaneously accessible to the CPU and device and, as a result, must live in cache-coherent memory.

Streaming Mapping. These mappings are typically set up for a single mapping and are used if the buffer to work with is already allocated. They can be non-contiguous and are used if the driver supports scatter-gather lists. Kernel developers recommend the use of streaming mappings wherever possible [34].

As we will see in the next sections, we aim to support a standard set of benchmarks for our accelerators. These benchmarks use synchronous transfers and streaming mappings. As such, we focus on these functionalities.

Analogous to the memory map design, we add support for DMA with and without patching the host driver. We will first address our patch set before we propose an approach that does not require driver patches.

3.3.5 DMA with Patching the Host Driver

The design outlined in Figure 3.6 shows the architecture for enabling DMA on the FVP. We demonstrate the case of already allocated userspace memory as this proved to be more challenging. This approach requires no changes in the FVP userspace application. The app does not require huge pages or coherent memory allocations from the driver. Numbering ① to ⑤ walk through the steps in Figure 3.6:

A test application allocates memory on the heap on the FVP ① and requests the accelerator to write to it. A stub driver (omitted in Figure 3.6) facilitates the escape mechanism ③ to interact with a host userspace manager. Be aware that the allocated buffer may be scattered throughout the FVP's physical memory and hence also scattered throughout the x86 host memory ②. We do not require this memory to be contiguous.

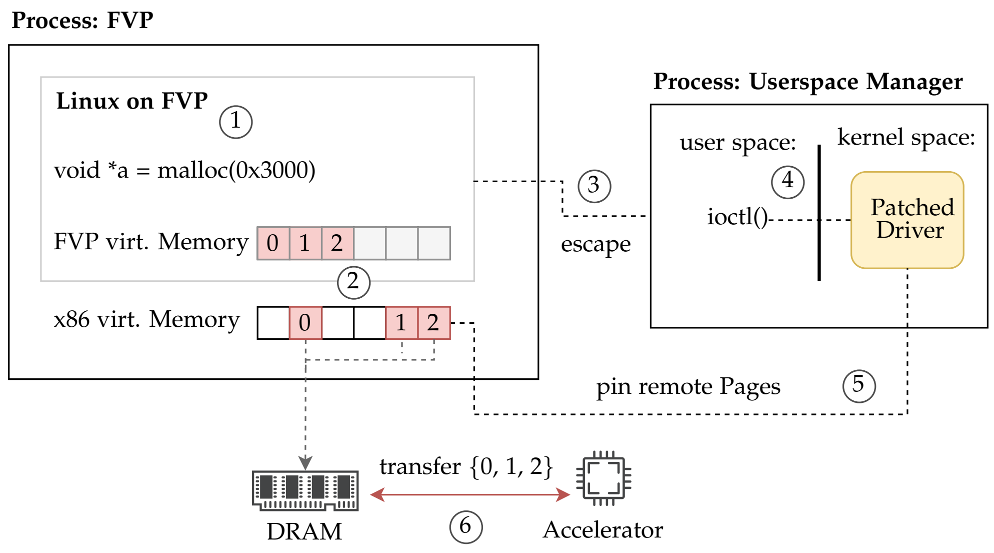

Figure 3.6: DMA with Patch Set: High-Level interaction between the userspace manager on x86 and an AArch64 application running in userspace on the FVP. A device driver is patched for DMA: It acts on behalf of a remote process to enable DMA on the FVP (6).

Subsequently, on the x86 host, we interact with a patched driver to request a DMA transfer (6). The following steps hold significance:

Address Mismatch. The virtual addresses in the userspace on the Fast Virtual Platform (FVP) (marked as (1)) do not correspond to virtual addresses on the x86 system. We only have information about the mapping between the physical addresses on the FVP and the virtual addresses on the host. As such, pointers to virtual addresses on the FVP are not meaningful on x86. Therefore, prior to performing the escape ((3)), it is necessary to pin all pages to RAM and translate the buffers into physical addresses on the FVP (Section 3.2.2).

Remote Memory Descriptor. The unmodified device driver incorrectly assumes that the userspace addresses correspond to the memory descriptor of the userspace manager process, whereas they should actually resolve to the Fast Virtual Platform (FVP). To address this, our patch set introduces an extra ioctl command that enables DMA operations involving remote user pages and memory descriptor of another process (5).

Taking into account the aforementioned aspects, the driver constructs a scatter-gather list that includes pinned remote pages associated with the FVP's host process. We keep the FVP unscheduled until the accelerator completes the DMA transfer, creating the illusion of immediate completion of the DMA request on the Fast Virtual Platform (FVP). However, it's important to note that, with the current implementation, only one transfer request can be executed at a time on the x86 host.

3.3.6 DMA without Patching the Host Driver

The approach presented now offers a simpler implementation that avoids the need for driver patches. However, this simplicity comes at the expense of additional memory copies. While we refrain from detailing this approach in a diagram, its key distinction lies in the use of a *Fixup Helper* kernel module. This module utilizes the CPU to copy DMA transfer results to and from the FVP's Linux host process. By adopting this approach, we can simplify the implementation as we can perform unchanged DMA transfers in the x86 userspace manager, albeit at the cost of additional CPU data copies.

Similar to the previous approach with kernel patches, the FVP remains unscheduled until the accelerator and the *Fixup Helper* module complete their data transfers, creating the illusion of immediate completion of DMA requests. We facilitate this approach for transferring data to and from the GPU.

3.4 GPU Device Access

In this section, we discuss design and implementation challenges to enable general-purpose GPU (GPGPU) computations on the FVP. For performance evaluation, we aim to select a standard set of GPU benchmarks from the Rodinia test suite [31]. Rodinia is designed for CUDA [39], OpenCL [47], and OpenMP [28]. This is why our implementation must provide support for one of these parallel programming models. A further restriction is our available GPU device. We own an Nvidia `GeForce RTX 3080 Ti`, which is part of the Ampere device family, Nvidia's second most recent GPU architecture.

3.4.1 GPGPU Software Stacks

To facilitate general-purpose GPU programming (GPGPU) on Linux, we need a kernel and a userspace component. The kernel component interacts with the accelerator, whereas a runtime component remains in userspace and interacts with the user's application. In the next paragraphs, we evaluate available GPU software stacks before we introduce our implemented approach.

Nvidia Driver and CUDA Runtime Nvidia recently open-sourced their Linux kernel driver implementations [70]. The kernel modules can be built for x86 or AArch64 and consist of ca. 900′000 lines of C code. The source distribution builds five different kernel modules[3]. While some mod-

[3]Nvidia Kernel Modules: `nvidia.ko`, `nvidia-modeset.ko`, `nvidia-uvm.ko`, `nvidia-drm.ko`, and `nvidia-peermem.ko`

ules for handling mode setting operations are not relevant to us, their interaction with each other and with the CUDA runtime is not clear and must be further studied. Apart from their complexity in code size, their userspace component – CUDA Runtime – is not open-sourced and can not easily be studied. This makes it further more challenging to port to the FVP. If we choose this technology stack, we have no choice but to treat the CUDA runtime as a black box and try to execute the closed-source artifact on the FVP.

Nouveau Driver. The nouveau project aims to build a free/libre software driver for Nvidia cards [1]. It implements the KMS/ DRM[4] kernel driver and is frequently merged upstream as part of the Linux kernel. The project focuses on reverse engineering Nvidia cards and creating an open-source alternative based on that knowledge. As a result, it lacks support for Nvidia's most recent system architectures. According to their project compatibility list, nouveau lacks support for the Ampere device family, making it incompatible with our GPU [68]. If we choose this technology stack, we have to face compatibility issues with our hardware.

libdrm. Libdrm (library for Direct Rendering Manager) is a component of the nouveau project. It is a userspace library that provides a programming interface for accessing and managing the DRM subsystem in the Linux kernel [1]. The project exposes a low-level API to the kernel's DRM subsystem through means of ioctl wrappers. If we choose nouveau as the kernel driver, we interact with libdrm to communicate with the driver.

Mesa. The mesa project implements open-source runtime 3D graphics libraries for the OpenGL, Vulkan, and OpenCL parallel programming standards [4]. If we choose mesa, we will use the OpenCL variants of the Rodinia benchmarks, link our benchmarks with mesa's runtime library and interact through libdrm with the nouveau kernel driver.

Gdev. Gdev is a software stack that provides an open CUDA runtime and integration layer for device drivers. It re-implements the CUDA driver API[5] providing runtime support in both the device driver and the userspace library [58]. Gdev is compatible with nouveau, and the (old) closed-source versions of the Nvidia driver. It is used in related works such as Graviton [85] and StrongBox [38].

[4]KMS: Kernel Mode Setting (display interactions), DRM: Direct Rendering Manager (GPGPU programming)

[5]CUDA exposes two user-facing APIs. A runtime API and a driver API. The runtime API provides implicit initialization, context management, and module management and hence leads to simpler code. The driver API provides more low-level support and control [71].

Figure 3.7: Two Nvidia GeForce GTX 460 SE, Fermi Architecture, Release Date 2010. We purchased these devices on a second-hand online platform. They have excellent open-source support in the nouveau kernel driver.

Implemented Approach. The paragraph below summarizes our design options. On the one hand, the CUDA runtime and the Nvidia driver are closed-source or large in code size. On the other hand, nouveau lacks proper support for our GPU. Given these constraints, we purchase an old Nvidia GeForce GTX 460 SE, which has excellent open-source support in the nouveau driver stack (Figure 3.7). Although its Fermi-based GPU architecture is already 13 years old, we do not rely on its absolute execution speed and evaluate its performance relative to a software-encrypted channel between the CPU and GPU. With proper nouveau support, we can either use OpenCL with mesa, libdrm and nouveau, or CUDA with Gdev and nouveau.

Approach 1: Nvidia (open-source) driver + official CUDA (closed-source),

Approach 2: Nouveau driver + OpenCL with mesa and libdrm,

Approach 3: Nouveau driver + reverse-engineered CUDA (open-source) with Gdev.

We opt for *Approach 3* because Gdev has a smaller code base and implements less complex interfaces. Bear in mind that these software components have to interact with both the AArch64- based FVP and the x86 host. Hence, the higher their code complexity, the more porting effort is required.

3.4.2 System Architecture and Design

We now discuss Gdev's architecture because our implementation is built upon some of its components.

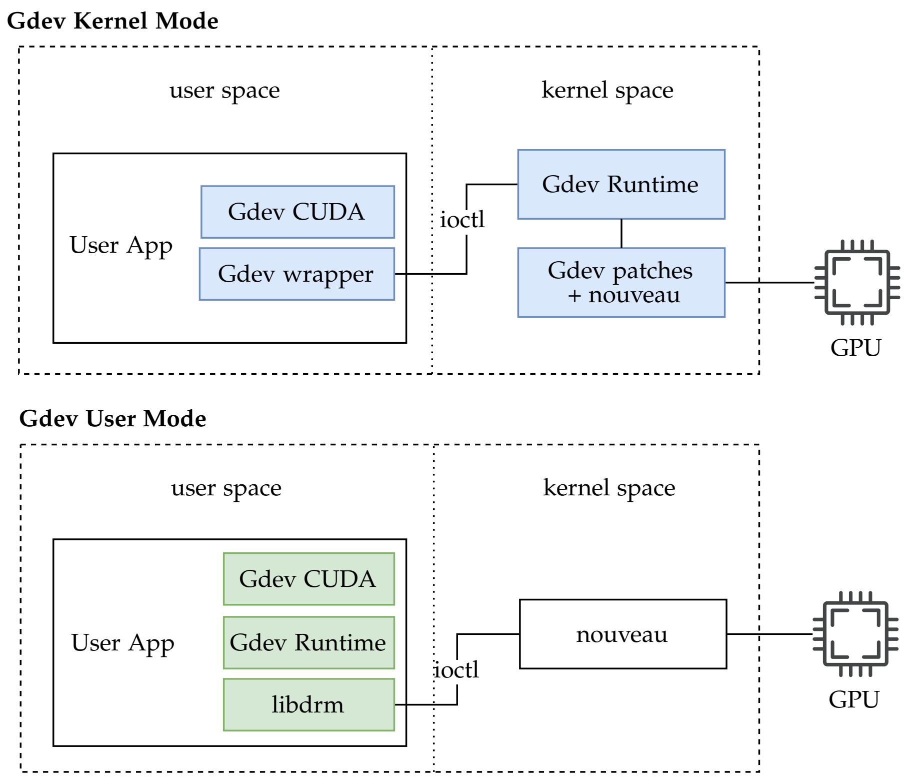

Figure 3.8: Architecture of Gdev: Gdev provides two modes; in kernel mode (top), a thin Gdev wrapper interacts through ioctls with the kernel Gdev Runtime. In user mode (bottom), all Gdev components remain in userspace.

Gdev Architecture. The Gdev [58] software stack consists of a reverse-engineered CUDA runtime, as well as a stand-alone Gdev Runtime. The CUDA component implements Nvidia's CUDA driver API and can be linked into a user application for GPGPU programming. Figure 3.8 depicts the high-level architecture, which can be assembled in two modes:

1. In *kernel mode* (top in Figure 3.8), most of Gdev's functionality remains in kernel space. The CUDA component is linked together with a thin wrapper layer into the user application. The wrapper invokes through ioctl commands the Gdev kernel module, which interacts with the nouveau driver. For this approach, nouveau must be patched with a Gdev-specific patch set.

2. In *user mode* (bottom in Figure 3.8), all functionality of Gdev remains in userspace and is linked into the application. The Gdev runtime uses libdrm to interact with the nouveau driver.

While *kernel mode* provides a thin interface to interact with the user application, it requires kernel patches. The patch set is only compatible with Linux kernel 3.3, which is already 11 years old. We tried to up-port these patches to a recent kernel version but failed because DRM and nouveau have evolved too much in the last years. *User mode*, on the other hand, does not require kernel patches. However, it uses libdrm to interact with nouveau, which implements a thick interface ($>$ 100 ioctls) and exposes low-level details of the DRM subsystem to userspace.

x86 Host Escape. For the x86 host escape, we seek a thin interface between the FVP and the x86 host. Both Gdev's *user* and *kernel mode* are unsatisfactory because they are either not compatible with a recent kernel version or expose a too-verbose interface. As a result, we propose a new mode: *Merged Mode*. In *Merged Mode*, we combine the benefits of having a thin user-facing API of *kernel mode* and no kernel patches of *user mode*. In Figure 3.9, we depict the high-level architecture of *Merged Mode*.

In our design, a GPU benchmark is executed by a user app (① in Figure 3.9, blue depicts components of *kernel mode* while green shows *user mode*). The app is linked with the CUDA runtime ② and Gdev's wrapper in *kernel mode* ③. The wrapper communicates with a merge layer ④ through ioctl syscalls, which is aware of the x86 escape mechanism. The merge layer escapes to the x86 host and invokes the userspace manager ⑤, which reformulates the request such that it is compatible with Gdev's *user mode*. Consequently, it invokes Gdev's Runtime in userspace ⑥, which utilizes libdrm ⑦ and nouveau ⑧ to interact with the accelerator on the PCI bus. Throughout the x86 host escape, the FVP remains descheduled and is resumed once the accelerator completes the requested task. With this design and the peripheral passthrough primitives introduced in Section 3.3, we are able to serve all Rodinia benchmark functionality, including DMA and memory map.

As a result of this approach, device interrupts are outsourced to the x86 host. Looking at it from the perspective of the merge layer ④ on the FVP, a request is promptly fulfilled as the FVP remains inactive until the request is handled. Another drawback of this design is that the request must originate from within the FVP and be completed immediately. Asynchronous callbacks, where the device notifies the merge layer of an update, are not supported. However, currently, we do not encounter any situations where this feature is necessary.

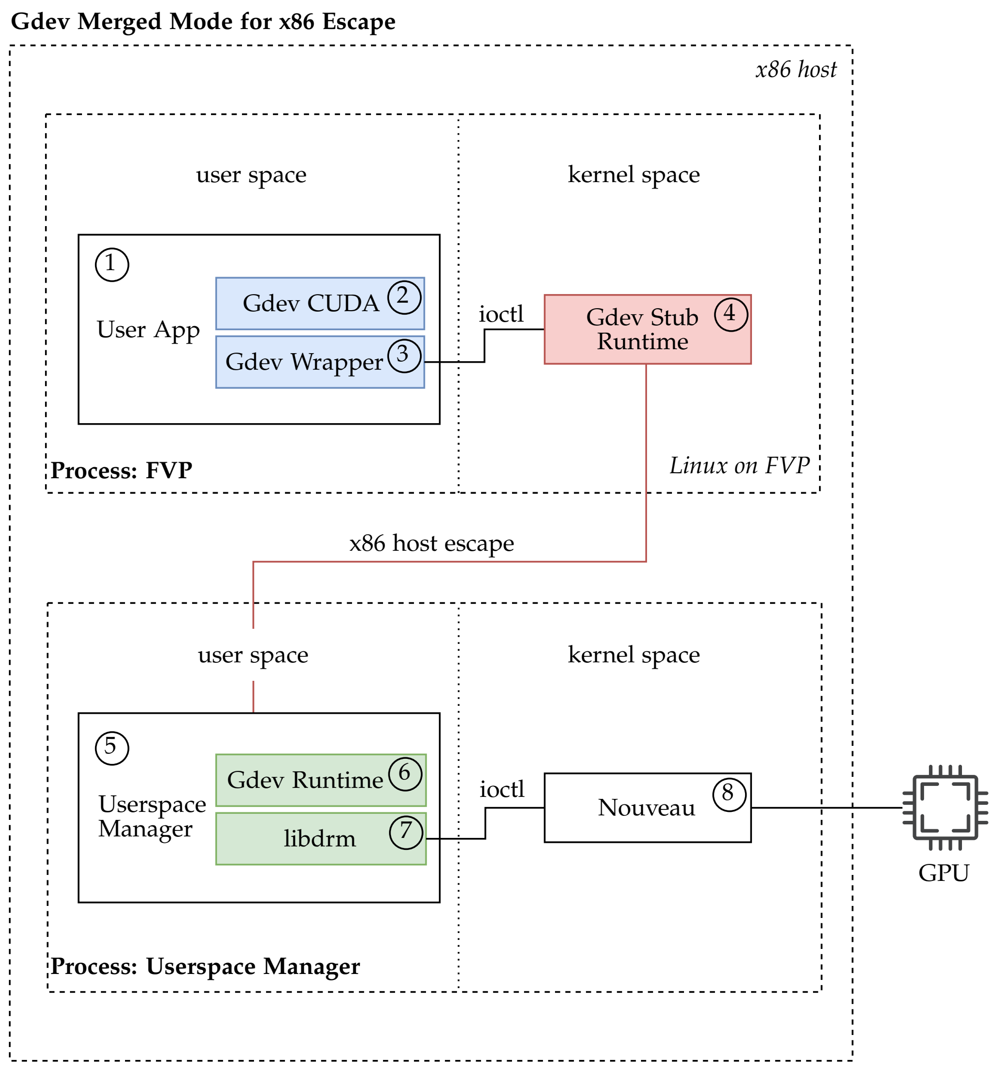

Figure 3.9: GPU escape architecture. A user app (1) on the FVP interacts with the host driver (8). We merge both Gdev modes (user and kernel) to escape on a thin API layer without integrating libdrm's internal complexity into the FVP. The diagram depicts the FVP's Linux process (top) and the x86 userspace manager process (bottom). The escape is invoked with a merge layer (red). Green are components of Gdev user mode, while blue are components of Gdev kernel mode.

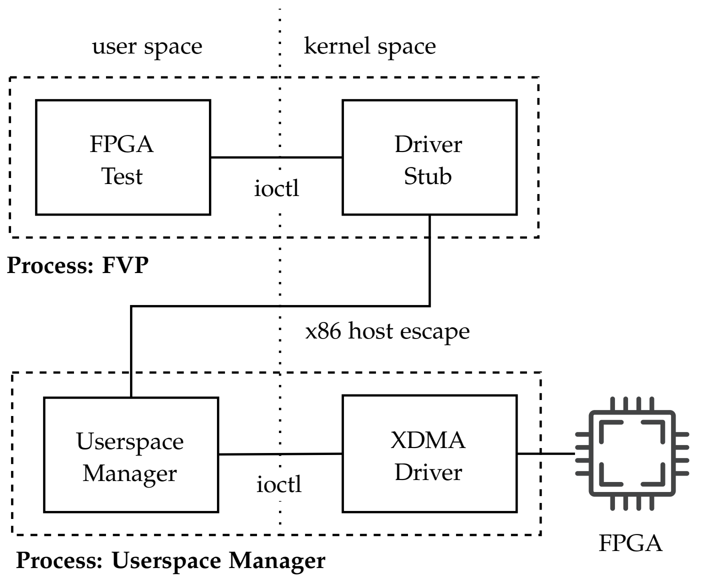

Figure 3.10: High-level architecture of the FPGA device access.

3.5 FPGA Device Access

For the FPGA accelerator, we reuse many design ideas already discussed in the previous section. The FPGA interactions are fundamentally more simple than the GPU accelerator.

Design. Figure 3.10 depicts the high-level architecture of the implemented design. We implement a stub driver in kernel space on the FVP, which employs the x86 escape mechanism to invoke the userspace manager on the host. These two components bridge the interactions between the FPGA test application and the XDMA device driver. The stub driver exposes the same user-facing API as the XDMA driver, which consists of simple open, close, mmap, read, write, and ioctl operations.

Implementation. We use a Xilinx Virtex Ultrascale+ VCU118 as our FPGA device. For the kernel driver on the x86 host, we use Xilinx's XDMA driver [89]. Due to its simple design, we apply the driver patch set introduced in Sections 3.3.2 and 3.3.5 to enable memory mapping and DMA. The FPGA applications are written in Python, so we cross-compile Python 3 as well as libcrypto to run on the FVP.

3.6 Booting a realm VM

With two functional accelerators introduced to the FVP, we now discuss how we boot a realm VM and interact with these accelerators from within the guest VM.

VM Manager. We launch a realm VM with `kvmtool`, which is a lightweight tool for hosting KVM virtual machines [86]. Arm recently published patches for `kvmtool` and the Linux kernel to deploy realm VMs from the Non-Secure world [20, 19]. The VM manager implements a small amount of emulated devices. Most devices are para-virtualized with the virtio protocol. They appear as normal PCI devices on the guest, and their functionality is implemented in callback functions within kvmtool. We will address these virtualized devices in the forthcoming Section 3.9.

Escape Mapping. To revisit an earlier topic from Section 3.2.2, recall that we exchange the physical addresses of all free memory pages with the x86 userspace manager. For benchmarks running in the realm VM, we apply the escape mapping in the early boot process of the realm VM. This way, the userspace manager is aware of all realm private memory regions.

3.7 Encryption-Based Reference Implementation

To demonstrate the overhead caused by a software-encrypted channel between the CPU and accelerators, we sketch an encryption-based solution for Arm CCA. As we will see in the evaluation section, we use this implementation as a reference to compare our approach.

In order to establish an encrypted communication channel, the VM has to encrypt its data buffer in software and send it to the accelerator, which then decrypts it using accelerator-specific methods. These are either encryption kernels in the case of GPUs or encryption IP blocks on the FPGA. Similarly, to return accelerator results to the realm VM, the device has to encrypt its result buffer before writing it to a publicly accessible part of CPU memory. Note that this approach leads to two extra data copies and increases computation overhead. It also breaks compatibility with an unprotected implementation because of invasive API changes in CPU and device-specific encryption logic. This is why our approach can potentially lead to significant improvements compared to an encryption-based method, as it eliminates the need to encrypt the communication buffers.

CUDA Encryption Layer. To keep implementation changes to a minimal extent, we implement an encryption layer that integrates into the CUDA Driver API within the realm VM. The main goal of this layer is to estimate the computational effort involved in encrypting data buffers. We allow data to be garbled and do not require functional correctness as long as the layer serves as a proxy to estimate the encryption overhead. Taking this into

```
CUresult cuMemcpyDtoH(void* dest, CUdeviceptr src, unsigned int size);
CUresult cuMemcpyHtoD(CUdeviceptr dest, void *src, unsigned int size);
CUresult cuLaunchGrid(CUfunction f, int width, int height);
CUresult cuMemAlloc(CUdeviceptr *dest, unsigned int size);
CUresult cuMemFree(CUdeviceptr ptr);
```

Listing 3: Function hooks for CUDA encryption layer

consideration, we implement the layer with function hooking in the symbol resolution process of the linker. We create an additional shared library and ensure that the linker resolves the symbol from our encryption layer instead of the CUDA implementation. We account for encryption costs in copying memory to and from the device, as well as copying kernel launch arguments to the device. These operations incorporate the most significant transfer overhead. The hooked functions are presented in Listing 3. To encrypt the data buffers, we use AES-256 CTR mode implemented in libopenssl 1.1.1q for the CPU and encryption kernels by Romain Dolbeau [40] for the GPU.

For each memory allocation, we additionally allocate a bounce buffer for GPU and CPU encryption. This is accomplished by hooking into GPU allocator functions. We use these bounce buffers as source and destination addresses for the encryption operations. They contain garbled data, and their purpose is solely to account for encryption overhead. We store them in a hashmap and retrieve their pointers upon transfer calls to and from the device, as well as kernel launches.

Limitations. There are several limitations in this prototype implementation. For instance, the transfer and launch of encrypted CUDA kernels are not modeled. We argue that these costs can be disregarded since the kernels are small in size and only transferred once as part of the set-up routine. We refer to Volos et al. [85] to demonstrate the feasibility of using an encryption kernel to launch other (encrypted) kernels on the GPU. Additionally, initial device configurations, such as resource management parameters, are not encrypted with the laid-out approach.

Lack of Encryption Accelerators. AArch64 has support for AES encryption instructions [9]. However, there is only limited support for them available on the FVP. Arm provides a proprietary Crypto plugin that enables Armv8.0 Cryptographic Extension on the FVP [15], but the plugin is only available for paying customers. Furthermore, there is currently no support for Scalable Vector Extensions in a realm VM [19]. The escape primitive presented in Section 3.2 offers a possible way to escape to the x86 host and utilize x86 instructions to accelerate encryption. Nevertheless, this approach makes a comparison between Arm and x86 instructions challenging because of their fundamental differences in architecture and instruction set. Based on these limitations, we have no choice but to utilize pure software encryption within the realm VM.

3.8 Driver Compatibility Layer

An important aspect in promoting our approach is compatibility with existing device drives. Bear in mind that our FVP version[6] does not have access to real PCI devices. It includes a PCIe subsystem as part of its `BasePlatformPCIRevC` hardware component, which incorporates an SMMUv3, an AHCI controller, and two PCI devices [8]. While the FVP does include two PCI block devices, they are solely dummy devices with limited functionality.

Apart from other debug options, we are able to configure the size of the different base address registers of the block devices. We further manage to supply a file image which is then announced as a block device on Linux in Non-Secure World. There are no GPU or FPGA accelerators available on the FVP, and the available peripherals on the PCI bus are not backed by "real" devices on the x86 host. The PCI subsystem on the FVP aims to provide a limited implementation of the PCIe standard to demonstrate interactions with the aforementioned peripheral stubs [8].

Taking this into consideration, the devices on the FVP are only of limited use: Even if we introduce them to a realm VM (refer to Section 3.9 on how to), we can not properly use them as part of an FPGA or GPU kernel driver to interact with a device. In our implementation, we do not interact with these devices because we use the escape primitive to escape to the x86 host. The central issue that requires attention is precisely where we want to escape to the x86 host in the call stack.

Implementation. A prototype compatibility layer has one primary goal: To delegate realm VM memory to protected Non-Secure memory, which then becomes accessible to PCI devices but not the hypervisor. Delegated memory includes direct memory access (DMA) and memory-mapped I/O.

DMA. The most portable way to do DMA in the Linux kernel is to employ the generic DMA layer[7]. With traditional DMA, data is transferred in a contiguous block of memory which can become inefficient when dealing with fragmented memory. For instance, `kmalloc`, which allocates contiguous kernel memory, is typically not used for objects larger than a page. While the maximal allocation limit depends on the hardware, we had issues allocating more than eight pages of contiguous memory. Among the ways to solve this

[6]`FVP_Base_RevC-2xAEMvA`

[7]DMA API as part of `linux/dma-mapping.h`

allocation issue is to use fragmented memory or the contiguous memory allocator (CMA). For fragmented memory, the kernel provides scatter-gather lists. These lists allow us to perform DMA requests on buffers that are scattered throughout physical memory.

We integrate the compatibility layer into the scatter-gather APIs of the generic DMA layer, but the approach is not limited to scatter-gather lists. We use Linux `kprobes` [54] to intercept calls to `dma_map_sg_attrs`, where the device driver informs the kernel of the scattered pages, which in turn translates these addresses to bus addresses to make them accessible to the device. The compatibility layer performs a secure monitor call (SMC) to the RMM to delegate the scattered pages to protected Non-Secure memory.

MMAP. For memory-mapped I/O, we hook into `remap_pfn_range`, which is used to map kernel memory (page frame numbers) to userspace (virtual addresses). The page frame number typically points to CPU-addressable device memory. Based on the address range definitions for memory-mapped devices in the device tree, we can check if an intercepted request belongs to devise memory or other kernel activities. In the former case, we assign it to protected Non-Secure memory. This assignment can either be done eagerly during boot or lazily upon the first request.

3.9 Introducing a PCI Device to the realm VM

Recall that we established in Section 3.8 that the FVP has only limited PCI device support and lacks a real GPU or an FPGA. In this section, we discuss where in the call stack we aim to transition to the x86 host. We present four different approaches with increasing complexity. The first approach does not model any PCI device interactions at all, and the last approach assigns one of the FVP's stub PCI devices exclusively to the realm world. Bear in mind that in all of these cases, we still have to transition to the x86 host to interact with the real accelerator.

Approach 1: No Device Since the FVP solely includes a limited implementation of the PCIe standard, we refrain from making a dummy PCI device available for a device driver in the realm VM. Thus, in this approach, the FPGA and GPU drivers do not bind to any PCI device, and all PCI-specific driver code is removed. This approach requires the least effort to implement.

Approach 2: Virtio Device Kvmtool includes support for virtio (virtualized I/O) devices. Virtio is a standard for communication and data transfer between a hypervisor and a guest VM. In contrast to full device virtualization, which uses traps for all I/O operations, virtio device drivers are aware of their paravirtualized devices and can more directly communicate with the hypervisor, i.e., with fewer traps to the hypervisor. While we refer to the virtio documentation [72, 61] for more details, the protocol is divided into three parts: frontend drivers (device drivers within the guest VM), backend drivers (callbacks in the hypervisor), and a transport layer (data structures and ring buffers). In this approach, we implement a new backend in kvmtool and bind it to the virtio device in the realm device driver (frontend). While this approach models some PCI device interaction in the realm VM, the transport buffers have to remain in Non-Secure memory. They can not easily be delegated to realm memory because the hypervisor still has to implement the virtio device backend.

Approach 3: VFIO Device The Virtual Function I/O (VFIO) subsystem of the Linux kernel [60] allows a userspace application to directly access devices, turning a userspace VM Manager such as `kvmtool` into a userspace driver for a device, or put differently, allowing a KVM guest to become a non-privileged userspace driver. This is accomplished by binding a proxy driver such as `vfio-pci` to a device which in turn exposes the PCI configuration space of that device to userspace[8]. Kvmtool can memory map the configuration space into the KVM guest, trapping at certain places to refrain from giving the guest full access to the device. This approach is typically referred to as *Passthrough Mode* in the setting of KVM. Giving a userspace application full access to a DMA-capable device comes with security risks. This is why VFIO further configures the Input-Output Memory Management Unit (IOMMU) to limit arbitrary device memory accesses. However, this can potentially interfere with our own IOMMU configurations. We found a patch [87] that disables IOMMU configurations completely in VFIO.

If we implement this approach, we pass through one of the PCI block devices of `FVP_Base_RevC-2xAEMvA`, giving a guest VM rudimentary access to a block device on the FVP. Similar to *Approach 2*, this approach requires the hypervisor to remain in control over the device because the `vfio-pci` device driver must run in kernel space of the Non-Secure world. Thus, the userspace accessible configuration space can not easily be delegated to realm memory unless the VFIO subsystem is patched accordingly for this use-case.

Approach 4: Unrestricted Device Access A major drawback of the previous approaches *2* and *3* is that the hypervisor remains under the control of the device and can break the integrity of the realm VM. In approach *4*, we thus try to delegate the FVP's PCI device completely to realm memory, making it inaccessible to the hypervisor.

[8]VFIO character device: /dev/vfio/

The KVM subsystem currently has no support for trusted device assignments to guest VMs. With this approach, we must patch both `kvmtool` and the KVM subsystem to enable the hypervisor to delegate one of its PCI devices to the realm VM. This allows the KVM guest to have exclusive control over the device. It is important to ensure that the hypervisor's PCI subsystem can handle scenarios where a connected PCI device is no longer accessible from the Non-Secure world. For instance, if the kernel re-probes the PCI bus to detect connected devices, it must avoid accessing the delegated memory to prevent general protection faults.

From the perspective of the realm VM, it is necessary to correctly map the PCI device into the guest VM's virtualized PCI Bus hierarchy, enabling the realm to have access to a single physical device. Further challenges arise in guaranteeing that the KVM guest can not break out of its sandbox by exploiting its exclusive device access.

Implemented Approach. In approaches 2 and 3, the hypervisor retains full control over the device. Due to the FVP's limited implementation of the PCIe standard, incorporating one of its stub devices into the realm VM offers limited benefit. Approach 4, although intriguing, requires substantial engineering effort. It entails patching the KVM hypervisor for a trusted device assignment. Considering the associated limitations, we opt for approach 1, where we refrain from introducing a device stub into the realm VM.

Approach 1 further implies that the device driver in the realm world does not bind to a PCI device. Hence, there are no interactions with `struct pci_dev` – Linux's abstraction to represent a PCI device. To employ the *Driver Compatibility Layer* introduced in Section 3.8 without a device, we manually instantiate a dummy PCI device abstraction, which we then use in the process to map the scatter-gather list. We leave hypervisor support for trusted device assignments to future work.

3.10 FVP Performance Instrumentation

In order to assess the performance overhead of different implementations, we have to measure benchmark executions on the FVP reliably. Arms FVP can not accurately model cycle counting and is not cycle accurate. Thus we can not use cycle metrics to compare performance [18]. In the next paragraphs, we detail ideas on how to instrument the FVP to receive performance metrics.

Timing Annotations. The FVP sacrifices timing accuracy to achieve better performance. Each instruction takes a single simulator clock cycle with no further delays [17]. With Timing Annotations, we can change the execution time for different instruction classes. Among other options, we can also change branch misprediction and pipeline stall latencies. However, as already mentioned, the FVP is not cycle-accurate, and measurements of timing-sensitive behaviors is discouraged [10].

Model Trace Interface. Fixed Virtual Platform (FVP) includes a tracing subsystem to track system events. Arm's *FastModelsPortfolio* [13] includes the source code of a `GenericTrace` plugin, which can trace hardware events such as executed instructions. Counting instructions gives us a reliable metric to estimate the performance of different implementations. Based on the source code of `GenericTrace`, we implement a tracing plugin that extracts the following metrics:

- Number of Instructions per core,
- Instruction Distribution (number per instruction type),
- Number of Exception-Mode switches,
- and Number of Security Domain switches.

Event Markers. Additionally, we implement event markers to track events of special interest. These markers exploit an unused instruction and register combination to signalize a special event: We move a marker value M into the zero register `XZR` to track the occurrence of event M. This is an unprivileged instruction without side effects and is typically not used in production code. Our tracer implementation is aware of this instruction and tracks the event accordingly. We can either count special events or micro-benchmark code blocks, guarded with start and stop markers[9]. In the latter case, we track the number of instructions passed between start and stop events. Throughout the code base of TFA, RMM, Linux, and the benchmark code, we track ca. 150 distinct events. Our fork of the `GenericTrace` plugin has 3084 LOC with 905 added lines of code.

Instruction Tracing Overhead. Intercepting all executed instructions decreases the FVP's performance on the x86 host significantly. To give an example, executing an encryption benchmark with the tracer can take up to 6 hours for a single iteration. To reduce the tracing overhead, we enable the tracing subsystem only during the benchmark and not during boot. This is accomplished with the proprietary `ToggleMTIPlugin` [14], part of Arm's *FastModelsPortfolio*. The plugin can enable the trace subsystem by use of a `HLT` instruction. However, executing `HLT` in the hypervisor raises an unhandled illegal exception, terminating the executing benchmark. To fix this, we catch the illegal instruction and increment

[9]Microbenchmarking with a start and stop markers further require instruction pipeline serialization

the instruction pointer accordingly. This is easily feasible on AArch64 because all instructions are 4 bytes in width.

Chapter 4

Experimental Methodology & Evaluation

In the following chapter, we compare the performance of our approach with different baseline implementations, including an encryption-based reference implementation.

4.1 Experimental Methodology

4.1.1 Benchmarks

GPU Benchmarks. For GPU benchmarking, we choose a set of standard benchmarks from the Rodinia benchmark suite. Rodinia is designed for heterogeneous computing infrastructures and supports the CUDA programming model [31]. Unlike the official version, we use version 2.1 from gdev-bench [57, 58], which rewrites the benchmarking code to support the CUDA Driver APIs, adding support for the reverse-engineered Gdev CUDA runtime. Of the total 23 benchmarks in the official Rodina release 3.1 [31], gdev-bench provides 11 benchmarks, of which we successfully test 9. We exclude two benchmarks `hotspot` and `lud` because they crash already on the unmodified x86 host system. This is probably because of compatibility issues with our GPU. Table 4.1 presents the set of benchmarks. They represent a range of application domains, transfer size, and number of launched kernels. We systematically increased the problem sizes to find a configuration as large as possible while still being compatible with the available computing resources on the GPU.

FPGA Benchmarks. For FPGA benchmarks, we are unable to find a standard set of benchmarks which is why we hand-code a set of applications and bitstreams for arithmetic algorithms, including matrix multiplications and singular value decomposition. These algorithms use implementations

Benchmark	**Domain**	**Tasks**	**Transfer (MB)**	**Problem Size (Points)**
nn	Dense Linear Algebra	1	1	42764
gaussian	Dense Linear Algebra	3148	38	1575 × 1575
needle	Dynamic Programming	229	39	1840
pathfinder	Dynamic Programming	5	20	50000 × 100
bfs	Graph Traversal	2	3	1840
srad_v1	Structured Grid	102	2	502 × 458
srad_v2	Structured Grid	4	64	2048 × 2048
hotspot	Structured Grid	5	3	512 × 512
backprop	Unstructured Grid	2	72	262144 × 16 × 1

Table 4.1: Overview of the GPU benchmark suite. The tasks column refers to the number of launched CUDA kernels.

Benchmark	**Domain**	**Transfer**	**Problem Size (Points)**
matmul5	Matrix Multiplication	300 B	5 × 5
matmul10	Matrix Multiplication	1200 B	10 × 10
svd32	Singular Value Decomposition	32 KB	32 × 32
svd64	Singular Value Decomposition	128 KB	64 × 64

Table 4.2: Overview of the FPGA benchmarks.

from Vitis Libraries [91]. The benchmarks for the FPGA are summarized in Table 4.2.

4.1.2 Hardware Setup and FVP Configuration

FVP. The FVP is based on Fast Models version 11.20.15 (Dec 1, 2022) and is the official binary version downloadable on Arm's website [12]. It is version `FVP_Base_RevC-2xAEMvA`, emulating 8 Arm v9.2 cores with support for RME. We run the normal world hypervisor on the FVP with 2 GB of RAM and simulated caches disabled. The FVP further has its memory allocations patched to page-align all emulated DRAM memory on the x86 host process. This is necessary to facilitate memory-mapped I/O and DMA introduced in Section 3.3. For benchmarking purposes, we isolate a single emulated core and prevent the Linux scheduler from submitting workloads to it. This core is exclusively used for benchmarking applications. For experiments in the Non-Secure world, we pin the benchmark to the isolated core and prioritize its process priority. For experiments in the realm VM, we pin and prioritize kvmtool to the same core. We run the realm VM with 1 GB of RAM and a single virtual core. Within the realm VM, we prioritize benchmarks to their maximum scheduling frequency.

Hardware. We run the FVP on an x86 host machine with a dual-socket Intel Xeon Gold 6346 CPU with 32 $\times$ 2 cores and 378 GB RAM. The host runs a patched version of Linux 5.10 with our custom page fault handler framework to escape out of the FVP. Further, we lock the CPU frequency to 3GHz, disable turbo boost, and set the CPU frequency driver to acpi-cpufreq with performance governor.

Accelerators. The benchmarks interact with an Nvidia GeForce GTX 460 SE GPU, running the Nouveau and DRM stack of Linux Kernel 5.10. The FPGA device is a Xilinx Virtex Ultrascale+ VCU118.

4.1.3 Implementation Variants

We benchmark the following implementation variants:

Vanilla Non-Secure (ns-vanilla). The first variant consists of an unmodified software stack for CCA. It includes the original TFA monitor software, RMM, and hypervisor. We boot the KVM hypervisor with a Linux kernel 6.2 and a patch set for realm VMs by Arm [19]. The kernel remains in EL2 Non-Secure World[1] and the hypervisor does not spawn a VM. We execute the test suites in EL0 to showcase the overhead of an unmodified execution.

Our Solution Non-Secure (ns-devmem). This variant measures the overhead caused by our modifications (TFA, RMM, kernel, hypervisor) on Non-Secure world operations. As such, we execute the benchmarks with our modifications in the Non-Secure world (El2 kernel, EL0 benchmark). No VM is spawned in this variant.

Vanilla Realm (realm-vanilla). To identify the overhead caused by running a CCA-enabled VM, we execute the test suites in this variant in a vanilla realm VM. The benchmarks operate on an unmodified software stack and utilize host peripheral passthrough to interact with the accelerators from within the realm VM.

Our Solution (realm-devmem). This variant encompasses all the necessary modifications to enable trusted peripheral access. Benchmark code in this variant runs in a realm VM equipped with our modifications in

[1]The Linux kernel remains in EL2 if virtualization hardware extensions are present.

TFA, RMM, kernel, and hypervisor. Device memory access is properly delegated to protected NS memory, and the SMMU is securely configured [82].

Encryption Reference (realm-enc). In this last variant, we execute the benchmarks in a realm VM with an unmodified software stack and software encryption. We employ the primitives introduced in Section 3.7 to account for an encryption communication channel between the benchmark software and accelerators. By comparing this variant with our solution, we can assess the overheads caused by encryption.

Two of these variants (ns-devmem and realm-devmem) are derived from the firmware modifications described in [82]. These adaptations delegate realm memory to the Non-secure world, granting access to the accelerator while preventing access from the hypervisor and other realms.

4.2 Evaluation

In this section, we present the results of our prototype implementation benchmarked with GPU and FPGA accelerators. We first highlight results from within the FVP for GPU and FPGA before we analyze their performance impact on the x86 host. In Appendix A.1, we further list the benchmarking results in tabular form.

GPU Benchmarks. In Table 4.3 and Figure 4.5 on the forthcoming page 52, we summarize the speedup across all GPU benchmarks. Our method demonstrates significant speedup results when compared to a purely software-based encryption approach, achieving speedups ranging from 425% to 3852%.

benchmark	transfer size	realm-vanilla	realm-devmem	realm-enc
sradv2	64 MB	-10.0%	0.0%	425.1%
sradv1	2 MB	-35.8%	0.0%	428.3%
nn	1 MB	-17.8%	0.0%	809.6%
gaussian	38 MB	-25.2%	0.0%	1286.5%
bfs	3 MB	-34.3%	0.0%	1955.4%
hotspot	3 MB	-38.1%	0.0%	2037.0%
needle	39 MB	-42.7%	0.0%	3287.4%
pathfinder	20 MB	-51.6%	0.0%	3537.3%
backprop	72 MB	-51.7%	0.0%	3852.3%

Table 4.3: GPU Benchmark slow down comparison relative to our solution, *realm-devmem*, sorted by realm-enc variant. Positive values imply that our solution is faster in comparison, i.e. the comparing variant slows down by the stated percentage compared to our method.

We incur an average slowdown of 34% to native execution (realm-vanilla) and a speedup of 1 958% to encryption (realm-enc) across all benchmarks.

As already pointed out in Section 3.7, the FVP lacks hardware-accelerated encryption instruction. The encryption method may not have as much of an impact when implemented on actual Armv9-compliant hardware. Nevertheless, no such hardware is available yet. We refer to our related work [82] to evaluate the performance on an Armv8-A board with RME-specific instructions (e.g., GPT-related operations) removed. However, since there is yet to be an Armv9-compliant processor available, these performance measurements

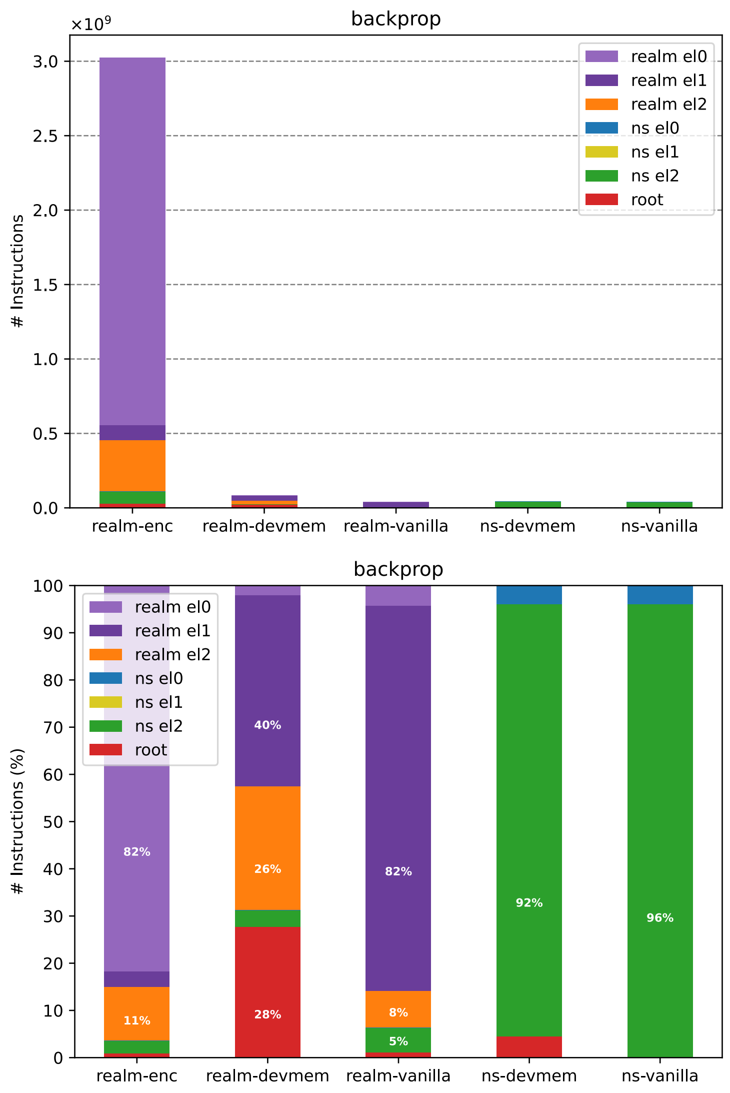

Figure 4.1: Number of instructions executed for *backprop* grouped by Aarch64 exception level. The top part shows the number of instructions in absolute terms, while the bottom part shows their relative distribution.

do not reflect the true costs either. Arm's FVP can not accurately model cycle counting and is not cycle-accurate. Therefore, we utilize instructions instead of cycles to showcase performance comparisons (Section 3.10).

Backprop executes the largest data transfer (72 MB) with a speedup of 3 852.3% compared to the encryption variant and a slowdown of 51.7% when compared against an unmodified realm VM (realm-vanilla).

To delve deeper into these result characteristics, we aggregate instructions for *backprop* and *nn* in 4.1 and Figure 4.2 into their different exception levels.

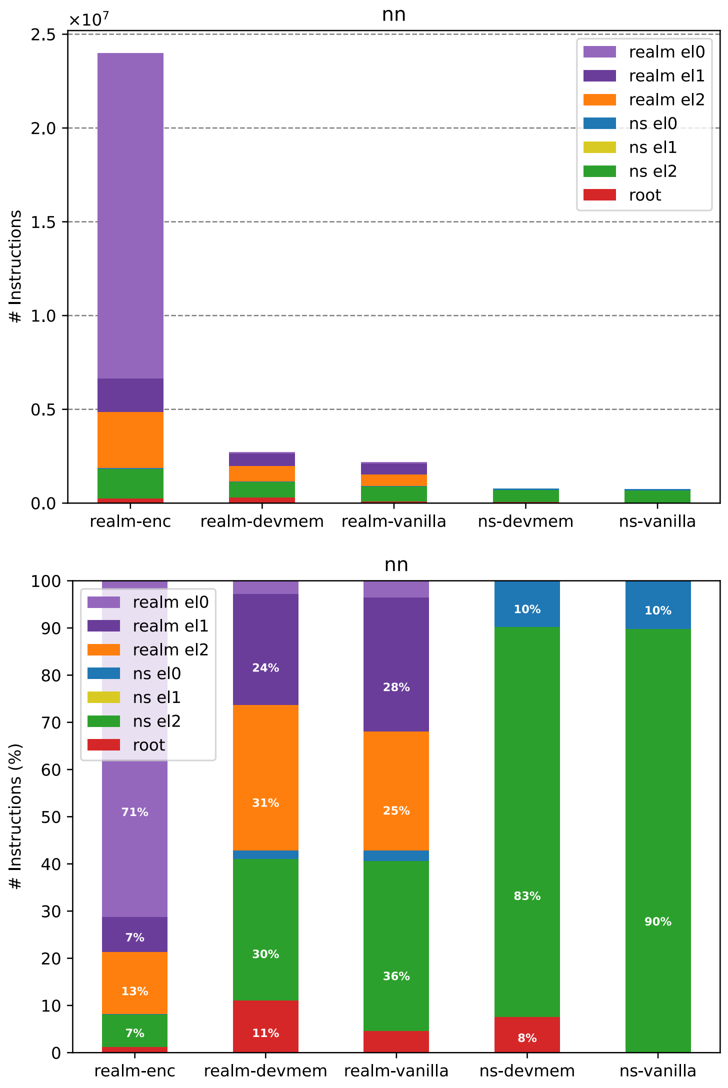

Figure 4.2: Number of instructions executed for *nn* grouped by Aarch64 exception level. The top part shows the number of instructions in absolute terms, while the bottom part shows their relative distribution. The x-axis depicts the implementation variant, while the y-axis shows the executed instructions.

These two benchmarks execute the largest (72 MB) and smallest (1 MB) data transfers. As we observe in the figures, there are additional overheads in realm EL2 and EL3 for *realm-devmem*.

For large memory allocations or fragmented memory, we see more context switches between the realm VM and the RMM and root software. This is due to a limitation in the current implementation. We currently make no particular optimizations in the hypervisor, monitor, RMM, and realm VM to

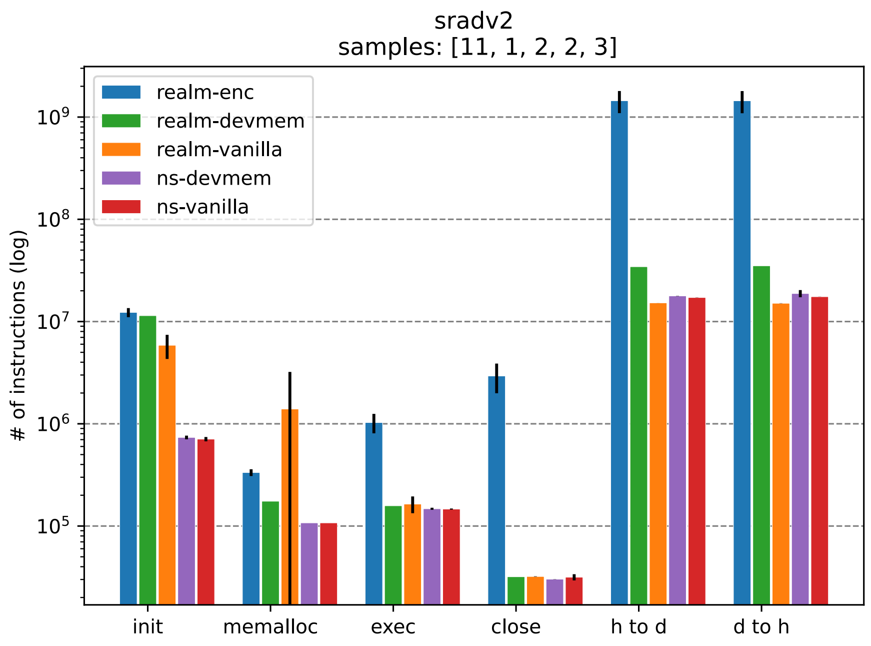

Figure 4.3: GPU Rodinia sradv2 benchmark. Lower is better. The x-axis shows the breakdown into different stages within the benchmark. A benchmark execution consists of initialization (init), memory allocation (memalloc), host-to-device transfers (h to d), kernel launches (exec), device-to-host transfers (d to h), and tear-down work (close). The y-axis refers to the number of instructions executed (log-scale).

employ contiguous memory allocations (CMA). Fragmented memory leads to more context switches because the implementations require contiguous memory when interfacing between different exception levels.

We further observe in the two Figures 4.1 and 4.2 that the encryption variant, while executing more instructions in total, also spends more instructions in the RMM (EL2 realm) and the hypervisor (NS EL2). The longer execution leads to more timer interrupts and context switches between the realm VM, RMM, and hypervisor. Note the lack of instructions in NS EL1. With virtualization extensions present, the NS kernel remains in EL2 and runs the VMM (virtual machine manager) in EL0.

Execution Stages. A benchmark execution comprises several stages, namely initialization (init), memory allocation (memalloc), transfers from the host to the device (h to d), launching of kernels (exec), transfers from the device to the host (d to h), and final tear-down tasks (close). Two of the benchmarks, *sradv2* in Figure 4.3 and *gaussian* in Figure 4.4, are further broken down into these different execution stages.

The initialization phase involves loading CUDA kernels from the disk. Since

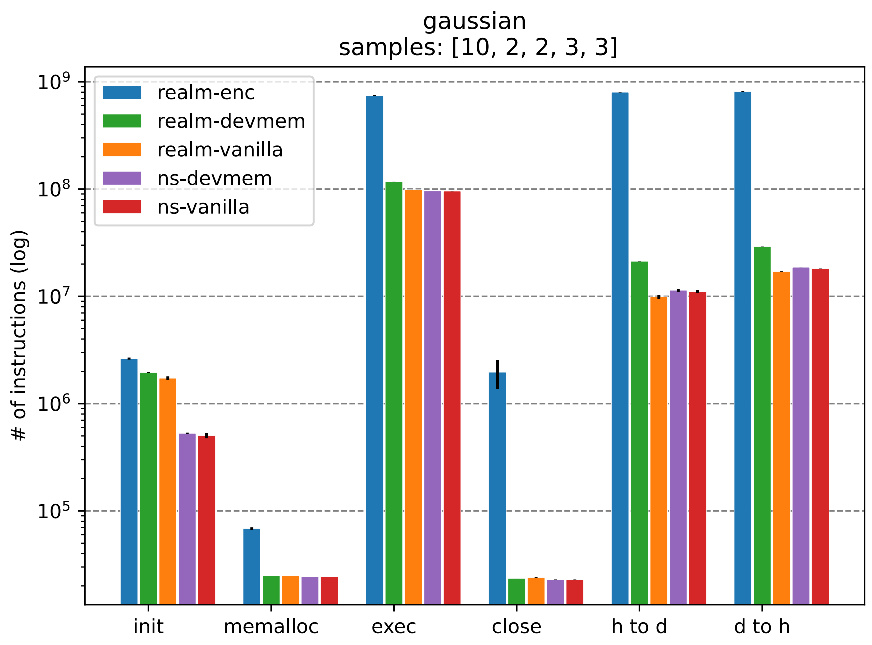

Figure 4.4: GPU Rodinia benchmark gaussian. Lower is better. The x-axis shows the breakdown into different stages within the benchmark. The y-axis refers to the number of instructions executed (log-scale).

these files are stored in the NonSecure world and shared with a realm VM using the 9P directory-sharing protocol, disk I/O leads to additional time caused by world switches and marshaling copies. However, these costs are included in all realm VM variants and occur only once during the initialization phase. In the close phase, all resources allocated during init and memalloc phase are released. The encryption variant (realm-enc) allocates additional buffers for CPU and GPU, which are also freed during the closing stage. This explains why the encryption variant is typically more resource-demanding in setup allocation and teardown-related tasks.

The large error margin in the realm-vanilla variant (memalloc in Figure 4.3) is likely due to measuring noise. However, note the log-scale in the y-axis. The error has no big impact on total instruction accuracy. Other instances in the execution breakdown where the unmodified baselines (ns-vanilla, realm-vanilla) perform worse than the other variants can be attributed to measurement noise. In fact, these inaccuracies are also not visible when considering the total instruction numbers in Figure 4.5.

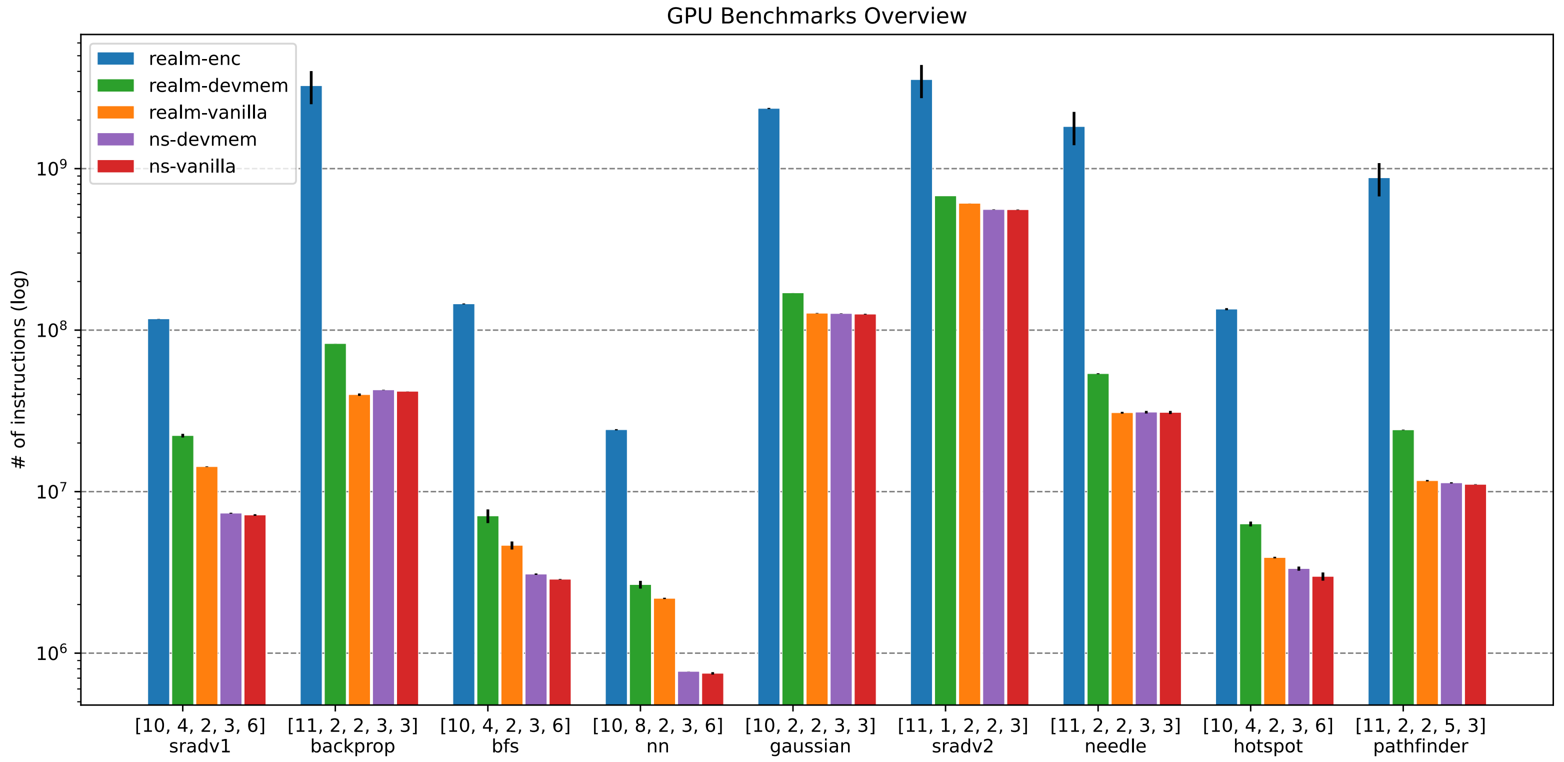

Figure 4.5: Overview Rodinia benchmark results. Lower is better. The y-axis shows the number of instructions executed (log-scale), while the x-axis shows the Rodina benchmarks, grouped by five different variants: *realm-enc* is the encryption reference, *realm-devmem* is our approach, see Section 4.1.3 for an overview of the variants. The numbers in square brackets on the x-axis refer to the number of samples in each of the five variants.

x86 Benchmarking. As discussed earlier in this report, mixing between Arm and x86 instructions is challenging because of their fundamental differences in architecture and instruction set. This is why the previous results focused on Arm instructions executed within the FVP. We now elaborate on the performance implications of the escape mechanism and the different variants on the x86 host. On x86, we are no longer constrained to counting instructions. The results shown in the next sections use cycles[2] as their metric. We benchmark x86 host interactions with the device driver. We measure the time it takes to invoke the device driver from the userspace manager. All syscall costs are added up and compared across different implementation variants and benchmarks.

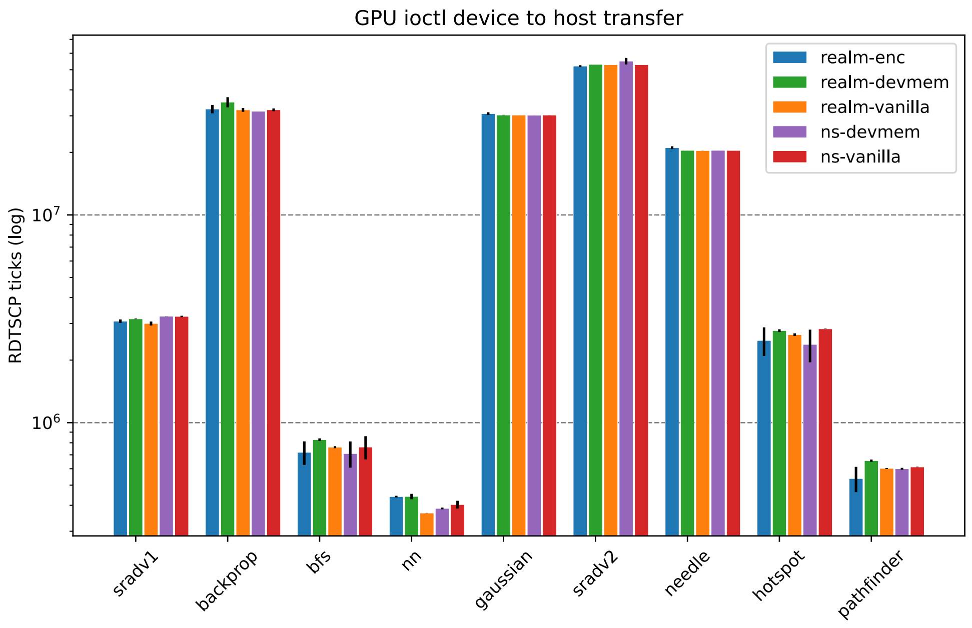

Figure 4.6: x86 GPU ioctl command to transfer memory to the host. The y-axis shows the number of ticks (log scale), while the x-axis shows the different benchmarks.

We observe that when GPU calls are not explicitly synchronized, they are generally non-blocking. The time it takes for a command to execute does not significantly vary across different implementations. All but the encryption variant execute the same workload on the x86 side. As a result, we do not come across notably interesting observations in the x86 benchmark results.

Figures 4.6 and 4.7 show GPU execution results in the userspace manager (log scale). The figures depict the ioctl commands to transfer data to the host (non-blocking) and allocate memory (blocking). We observe a larger variance

[2]ticks received from the RDTSCP instruction

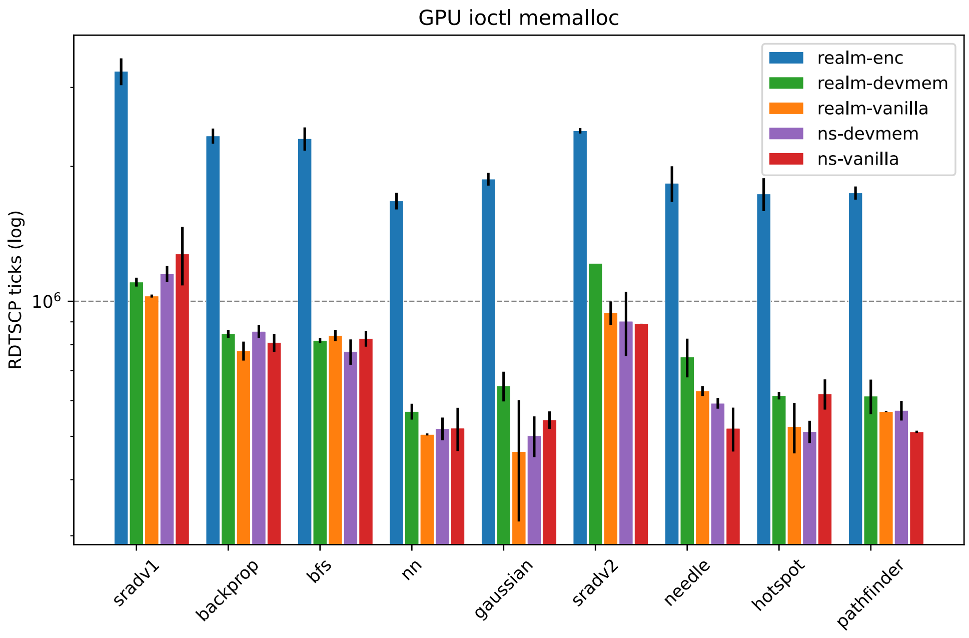

Figure 4.7: x86 GPU ioctl command to allocate memory. The y-axis shows the number of ticks (log scale), while the x-axis shows the different benchmarks.

in these measurements than on the Fixed Virtual Platform (FVP). Noise on the x86 host may be attributed to the running FVP, whose instruction tracer causes a large memory footprint or the lack of more samples. Since *memalloc* in Figure 4.7 is a blocking call and the encryption variant allocates more memory, we observe more time spent in the encryption variant. However, the remaining variants execute the same GPU kernels, and their variance is attributed to noise.

FPGA Benchmarks. Finally, we illustrate the results of the FPGA benchmarks in Figure 4.8 on page 55. The measurements agree with our earlier observations in the GPU benchmarks. The realm VM utilizing a software encryption channel exhibits the highest performance overhead. Notice how we solely have one encryption sample for the *svd32* and *svd48* benchmarks. The pure software encryption approach implemented within a Python interpreter in the realm VM, with the tracer plugin enabled, required ca. 6 hours for a single iteration.

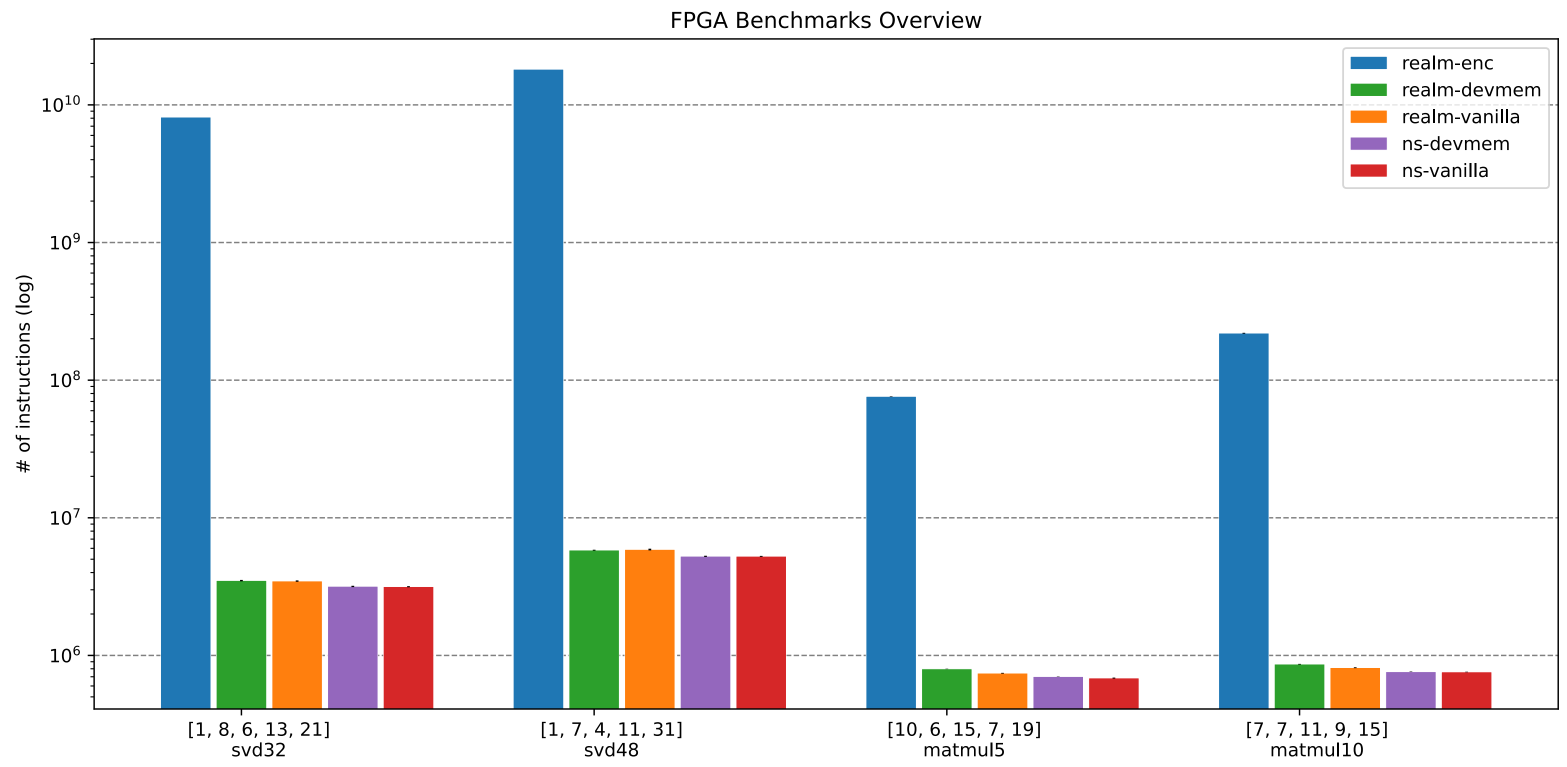

Figure 4.8: Overview FPGA benchmarks. Lower is better. The y-axis shows the number of instructions executed (log-scale), while the x-axis shows the benchmarks, grouped by five different variants: *realm-enc* is the encryption reference, *realm-devmem* is our approach, see Section 4.1.3 for an overview of the variants. The numbers in square brackets on the x-axis refer to the number of samples in each of the five variants.

Chapter 5

Related Work

We start with a survey of trusted execution environments (TEEs) which is followed by the integration of devices into TEEs. We distinguish between integrated and external devices. We then discuss existing software support for Arm CCA and dive into simulation software.

A central idea in recent TEE developments is to use hardware support to guard confidential information against being accessed by higher-privileged software. This can be achieved at the level of functions [51, 45, 63], applications [83, 32, 80, 75], containers [25, 44] or virtual machines [88, 6, 64, 5]. Among these approaches, VM-based isolation has become increasingly popular. This seems mainly because of their compatibility with existing Infrastructure as a Service (IaaS) providers. Many cloud vendors, including Microsoft Azure [67, 3] and Google Cloud [43], have started offering TEE-based services to end users.

Hardware-based TEEs. One line of work has focused on pushing access control into the CPU. This is motivated to provide enhanced security by removing many software components from the trusted computing base (TCB). Arm TrustZone [7], Intel SGX [51], and AMD SEV [5] are broadly adopted implementations for Arm and x86. Apart from isolation protection, they support attestation to verify the authenticity of the underlying platform and memory encryption mechanisms to ensure confidentiality when data leaves the CPU before being stored in RAM. Intel TDX [52] and Arm CCA [6] are upcoming TEE architectures that extend their existing counterparts, TrustZone [7] and SGX [51], with VM-based isolation. Other works for the RISC-V architecture include Sanctum [36], Keystone [63], and Penglai [41].

TEE Expansion to Integrated Devices. The emphasis has revolved around confidential computing without external accelerators for an extended period. Only recently has the focus broadened to include confidential devices as well. StrongBox [38] demonstrates that a TrustZone-based TEE can be extended to an integrated GPU. Their approach does not require hardware changes. Cure [27] presents confidential integrated devices for RISC-V based on hardware modifications. These approaches demonstrate that integrated devices connected to the system bus can be guarded by the same bus-level enforcement that also protects the CPU.

TEE Expansion to External Devices. Graviton [85] demonstrates confidential device interactions on Nvidia GPUs. Their approach adds TEE primitives directly to devices and requires hardware modifications. HIX [53] pursues the same goal but without GPU modifications. These works target the Intel SGX architecture.

Other works that facilitate confidential peripherals and use hardware modifications are Nvidia Hopper H100 [2] and Telekine [50] for GPU TEEs, ShEF for FPGAs [90], and IceClave [55] for in-storage computing on SSDs. Unlike our method, these approaches employ hardware encryption on the device and use software-based encryption with data copies to exchange information confidentially.

Our method creates a protected memory region and guards it by stage 2 translations in RMM and SMMU, and granular protection checks (GPC). To the best of our knowledge, we are the first to demonstrate this on Arm CCA. Similar methods have been demonstrated on other platforms by InkTag [48] Sanctuary [29], and Overshadow [32]. Their isolation primitives rely on TrustZone address space controller and nested page tables.

Software Support. Arm develops and maintains TFA [11], a monitor firmware for EL3, an early version of RMM [21], e.g., the realm world hypervisor, kvmtool [20], and kernel [19] patches for the request of comments (RFCs). Independently, Samsung develops islet [78], a Rust-based implementation of the RMM.

Simulation. We modify the proprietary Arm Fixed Virtual Platform (FVP) to channel DMA and memory-mapped I/O requests to the underlying host machine. Arm provides a software plugin that bridges the non-functional Mali GPU on the FVP with GPU facilities on the host machine [16]. A Mali OpenGL emulator on the host machine can execute the OpenGL requests [22]. This feature is only available for premium customers and specific to the GPU. GPGPU-Sim [46] and Multi2sim [84] are GPU simulation software providing a simulation model of contemporary Nvidia GPUs. Their purse software simulation can negatively impact overall system performance. No-Mali [37] fakes the register-level interface of a Mali GPU. This allows existing software to interact with a Mali GPU, even if the device is not functional. The stub cannot be used for GPGPU programming.

QEMU [77] supports a PCI passthrough mode using VFIO [60]. However, it does not support the emulation of Armv9 hardware with GPC (granular protection check) yet. SimEng [76] and gem5 [42] are other simulation software to simulate processors and system components. They target AArch64 and promote collaborative and open development of simulation software. However, they do not possess the same level of accuracy in microarchitectural simulation and do not closely resemble the actual hardware as much as Fixed Virtual Platforms (FVP) do. FVPs, on the other hand, are specifically developed by Arm to serve as a temporary substitute until the actual hardware becomes available in silicon.

Chapter 6

Future Work

In this chapter, we summarize directions for future work. For more avenues of forthcoming tasks, we direct readers to work in [82].

Host Escape Mechanism. The current implementation of the faulthook mechanism described in Section 3.2 enables the registration of only one target application and faulting region simultaneously. Further engineering improvements in this avenue aim to introduce support for multiple hooks.

Contiguous Memory Allocator We have not implemented optimizations in realm VM, TFA, RMM, or hypervisor to incorporate Contiguous Memory Allocations (CMA). Currently, more fragmented memory leads to more context switches and performance penalties.

Encryption Reference Implementation. The encryption implementation introduced in Section 3.7 currently only accounts for encryption overhead without encrypting information when transferring data between the realm VM and GPU. This is motivated to keep the implementation simpler. Future work in this direction adds functional encryption support on the GPU and encrypts the entire communication. We only account for memory transfer and kernel launches. Volos et al. [85] demonstrate the feasibility of using an encryption kernel to launch other (encrypted) kernels on the GPU.

Hypervisor Support for Trusted Devices. Due to the FVP's limited PCIe simulation, we decided not to model PCI interactions within the realm VM. In future work, we aim to pursue one of the approaches discussed in Section 3.9. We aim to allocate a device to the realm VM while ensuring the continued functionality of the hypervisor. Presently, there is no support for confidential devices that are inaccessible from the KVM hypervisor. A patch for KVM [59] is currently under review, which proposes fd-based Guest

Private memory. This introduces memory that will not be memory-mapped into KVM userspace once assigned to a VM. We can employ this technique in conjunction with our method for trusted device assignments, where we allocate one of the FVP's PCIe stubs solely to a realm VM. Additionally, we can use this method to demonstrate the confidential assignment for integrated peripherals, such as a Mali GPU.

Benchmarking. In addition to the current benchmark suites, we aim to showcase more practical scenarios like neural network training from within the realm VM. We also intend to benchmark applications running on a more powerful FPGA, or an FPGA in Amazon Web Services shielded with ShEF [90]. At present, our performance assessment relies on instructions rather than cycles. Our tracing technique allows us to extract all executed instructions, enabling us to multiply each instruction type by a cycle metric. This would allow us to obtain the total execution time.

Chapter 7

Conclusion

We present the first system that allows CCA-enabled virtual machines to interact with PCIe- based accelerators securely. As such, we propose using accelerators as first-class abstractions in Arm CCA. In this work, we presented the design and implementation of a bypass mechanism to connect functional PCIe-based accelerators to the Fixed Virtual Platform (FVP). This mechanism allows a realm VM to communicate with an accelerator, although the FVP does not provide a functional interface to connect to PCIe devices. We discuss methods for DMA and memory-map device memory between the FVP and the underlying host system. We apply these mechanisms to two specific devices on the host machine, namely an FPGA and a GPU. We benchmark our approach against different implementations with a set of tailored benchmarks. To accomplish this, we develop a reference implementation based on encryption techniques. This allows us to compare our method against the current state of the art when it comes to implementing a secure channel between the CPU and an external accelerator. The additional memory copies and encryption overhead in the device and CPU-specific encryption logic lead to significant overheads. In our evaluations based on the FVP, we incur an overhead of 34% to native execution and a speedup of 1 958% to encryption. Although these estimates are based on a simulation because no hardware with CCA support is yet available, we hope to lay foundations and inspire upcoming cloud deployments.

Appendix A

Appendix

A.1 Build System

We built a custom build system to support the iterative development process required to prototype confidential device access on Arm CCA. The system is bash and GNU Make-based and uses the package management system of the Buildroot project [30].

Toolchains. We cross-compile software packages for two architectures, namely AArch64 and x86_64. This is required because we build software components for the x86 host and the Fixed Virtual Platform (FVP). The build system is responsible for downloading and configuring two custom toolchains. We build environment wrappers that source into a custom build environment, where the appropriate toolchain is set up and required software dependencies compiled for the target architecture. This allows us to agree on a reproducible standard development environment across different machines.

Root File systems. The build system generates two custom AArch64-based root file systems for both the Non-secure and realm world. These file systems have all required libraries bundled and essential software packages such as Python interpreter and its dependencies installed.

We further configure 9P (Plan 9 Filesystem Protocol) directory sharing with the underlying host system, eliminating the need for rebuilding the root file systems when software dependencies change. Additionally, the root file systems have matching kernel headers installed such that we can compile kernel modules on the x86 host and run them on the FVP.

Fixed Virtual Platform. The build system automates the download process of the FVP and patches its memory allocation such that we can properly bypass the FVP and interact with accelerators on the x86 host. It further configures the tracer plugins for the benchmarking evaluation.

Kernels and Packages. Furthermore, the build system assembles three kernel images, one for the x86 host with the faulthook patchset present and two AArch64 Linux kernels for ARM CCA in the Non-secure and realm worlds. These kernels can be executed on the FVP (for AArch64 kernels) or QEMU [77]. We configure Qemu to support kernel debugging. The build system further handles the download and compilation of TFA [11], RMM [21], and kvmtool [86], along with their respective library dependencies.

The efforts put into building a system to build the project proved valuable because they increased developer productivity. Creating a custom build system became necessary because of all the interdependencies of the required software packages, different architectures, and development machines.

A.2 Benchmarking Data for GPU

The columns *realm-vanilla* and *realm-devmem* indicate the relative changes in performance compared to the native execution of a VM in realm world (realm-vanilla) and our approach (realm-devmem).

Let I_{enc} and I_{realm} denote the number of instructions for realm-enc and realm-vanilla. We define the relative overhead O_{realm} as follows:

$$O_{realm} = (\frac{I_{enc}}{I_{realm}} - 1) * 100 \tag{A.1}$$

For example, the encryption overhead O_{realm} compared to realm-vanilla for *sradv1/realm-enc* is 722.6%.

Rodinia GPU Benchmark Data

Benchmark	Variant	# Instr.	± CI (0.95)	Stdev	realm-vanilla (%)	realm-devmem (%)	#
sradv1	realm-enc	117 424 576	138 648	193 817	722.6	428.3	10
sradv1	**realm-devmem**	22 224 974	1 077 276	677 011	55.7	0.0	4
sradv1	realm-vanilla	14 275 528	804 817	89 577	0.0	−35.8	2
sradv1	ns-devmem	7 352 006	153 225	61 681	−48.5	−66.9	3
sradv1	ns-vanilla	7 165 230	96 224	91 692	−49.8	−67.8	6
backprop	realm-enc	3 261 074 205	532 853 817	793 162 499	8 083.4	3 852.3	11
backprop	**realm-devmem**	82 510 054	347 102	38 633	107.1	0.0	2

Rodinia GPU Benchmark Data

Benchmark	Variant	# Instr.	± CI (0.95)	Stdev	realm-vanilla (%)	realm-devmem (%)	#
backprop	realm-vanilla	39 849 798	8 645 962	962 304	0.0	−51.7	2
backprop	ns-devmem	42 662 968	364 991	146 929	7.1	−48.3	3
backprop	ns-vanilla	41 799 401	123 265	49 621	4.9	−49.3	3
bfs	realm-enc	145 388 604	637 965	891 814	3 028.8	1 955.4	10
bfs	**realm-devmem**	7 073 495	1 261 776	792 960	52.2	0.0	4
bfs	realm-vanilla	4 646 741	3 388 669	377 162	0.0	−34.3	2
bfs	ns-devmem	3 083 850	92 119	37 083	−33.6	−56.4	3
bfs	ns-vanilla	2 866 154	14 965	14 260	−38.3	−59.5	6
nn	realm-enc	24 168 435	149 018	208 312	1 007.2	809.6	10
nn	**realm-devmem**	2 656 929	130 634	156 257	21.7	0.0	8
nn	realm-vanilla	2 182 906	251 665	28 011	0.0	−17.8	2
nn	ns-devmem	770 451	2 494	1 004	−64.7	−71.0	3
nn	ns-vanilla	749 733	14 886	14 185	−65.7	−71.8	6

Rodinia GPU Benchmark Data

Benchmark	Variant	# Instr.	± CI (0.95)	Stdev	realm-vanilla (%)	realm-devmem (%)	#
gaussian	realm-enc	2 357 798 745	13 810 532	19 305 793	1 753.3	1 286.5	10
gaussian	**realm-devmem**	170 052 566	773 655	86 109	33.7	0.0	2
gaussian	realm-vanilla	127 222 798	5 728 129	637 547	0.0	−25.2	2
gaussian	ns-devmem	126 657 935	1 084 676	436 641	−0.4	−25.5	3
gaussian	ns-vanilla	125 546 261	2 122 090	854 256	−1.3	−26.2	3
sradv2	realm-enc	3 557 216 840	582 057 781	866 403 484	483.6	425.1	11
sradv2	**realm-devmem**	677 404 683	0	0	11.1	0.0	1
sradv2	realm-vanilla	609 553 048	3 713 344	413 299	0.0	−10.0	2
sradv2	ns-devmem	557 809 394	23 676 634	2 635 234	−8.5	−17.7	2
sradv2	ns-vanilla	556 740 108	5 972 570	2 404 283	−8.7	−17.8	3
needle	realm-enc	1 820 657 850	299 065 049	445 163 709	5 810.9	3 287.4	11
needle	**realm-devmem**	53 748 540	4 093 081	455 564	74.5	0.0	2
needle	realm-vanilla	30 801 491	4 910 961	546 595	0.0	−42.7	2

Rodinia GPU Benchmark Data

Benchmark	Variant	# Instr.	± CI (0.95)	Stdev	realm-vanilla (%)	realm-devmem (%)	#
needle	ns-devmem	31 036 507	1 751 010	704 876	0.8	−42.3	3
needle	ns-vanilla	30 929 870	2 057 644	828 313	0.4	−42.5	3
hotspot	realm-enc	134 838 446	1 539 241	2 151 710	3 351.2	2 037.0	10
hotspot	**realm-devmem**	6 309 582	398 757	250 598	61.5	0.0	4
hotspot	realm-vanilla	3 906 984	564 994	62 884	0.0	−38.1	2
hotspot	ns-devmem	3 338 251	299 517	120 572	−14.6	−47.1	3
hotspot	ns-vanilla	2 985 402	199 508	190 110	−23.6	−52.7	6
pathfinder	realm-enc	877 382 026	144 112 454	214 513 982	7 408.7	3 537.3	11
pathfinder	**realm-devmem**	24 121 606	1 194 339	132 931	106.4	0.0	2
pathfinder	realm-vanilla	11 684 918	1 536 072	170 966	0.0	−51.6	2
pathfinder	ns-devmem	11 333 223	125 869	101 371	−3.0	−53.0	5
pathfinder	ns-vanilla	11 089 268	108 860	43 822	−5.1	−54.0	3

Table A.1: Benchmarking Data for Rodinia Benchmarks.

A.3 Benchmarking Data for FPGA

The columns *realm-vanilla* and *realm-devmem* indicate the relative changes in performance compared to the native execution of a VM in realm world (realm-vanilla) and our approach (realm-devmem).

FPGA Benchmark Data

Benchmark	Variant	# Instr.	± CI (0.95)	Stdev	realm-vanilla (%)	realm-devmem (%)	#
svd32	realm-enc	8 135 782 290	0	0	234 390.9	232 523.9	1
svd32	**realm-devmem**	3 497 398	35 862	42 896	0.8	0.0	8
svd32	realm-vanilla	3 469 552	45 999	43 832	0.0	−0.8	6
svd32	ns-devmem	3 171 696	21 909	36 255	−8.6	−9.3	13
svd32	ns-vanilla	3 157 901	13 901	30 538	−9.0	−9.7	21
svd48	realm-enc	18 109 339 032	0	0	307 854.6	311 074.5	1
svd48	**realm-devmem**	5 819 674	41 157	44 501	−1.0	0.0	7
svd48	realm-vanilla	5 880 522	152 846	96 056	0.0	1.0	4
svd48	ns-devmem	5 248 992	36 388	54 164	−10.7	−9.8	11
svd48	ns-vanilla	5 245 916	15 370	41 902	−10.8	−9.9	31

FPGA GPU Benchmark Data

Benchmark	Variant	# Instr.	± CI (0.95)	Stdev	realm-vanilla (%)	realm-devmem (%)	#
matmul5	realm-enc	76 067 965	253 094	353 801	10 144.8	9 432.3	10
matmul5	**realm-devmem**	798 003	2 864	2 729	7.5	0.0	6
matmul5	realm-vanilla	742 505	2 741	4 950	0.0	−7.0	15
matmul5	ns-devmem	701 402	2 247	2 430	−5.5	−12.1	7
matmul5	ns-vanilla	684 816	2 667	5 534	−7.8	−14.2	19
matmul10	realm-enc	219 277 179	1 532 397	1 656 922	26 825.4	25 282.2	7
matmul10	**realm-devmem**	863 902	4 385	4 741	6.1	0.0	7
matmul10	realm-vanilla	814 388	4 364	6 496	0.0	−5.7	11
matmul10	ns-devmem	760 608	2 400	3 122	−6.6	−12.0	9
matmul10	ns-vanilla	758 225	1 821	3 288	−6.9	−12.2	15

Table A.2: Benchmarking Data for FPGA Benchmarks.

Appendix B

Acknowledgement

I want to thank Prof. Shweta Shinde, Supraja Sridhara, and Benedict Schlüter for their support and fruitful discussions. I also like to thank Valentin Ogier for implementing a CUDA encryption kernel on the x86 host and Florim Memedi for pointing out the grammatical mistakes in the draft of this report.

I also like to thank my family and my best friends Loris Reiff and Andy Nguyen, a.k.a. `liblor` and `theflow`. I am especially grateful to my parents for encouraging me in all of my pursuits and inspiring me to follow my dreams.

Bibliography

[1] Nouveau: Accelerated Open Source driver for NVIDIA cards. https://nouveau.freedesktop.org/.

[2] NVIDIA Hopper Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/, 2022.

[3] Microsoft becomes the first major cloud provider to offer confidential virtual machines. https://mspoweruser.com/microsoft-cloud-provider-confidential-virtual-machines/, accessed 2023-05-10.

[4] Mesa 3D. Mesa Project, 3D Graphics Library. https://www.mesa3d.org/, accessed 2023-05-04.

[5] AMD. AMD SEV. https://www.amd.com/en/processors/amd-secure-encrypted-virtualization, accessed 2023-05-04.

[6] ARM. Arm Confidential Compute Architecture (ARM-CCA). https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture.

[7] ARM. Learn the architecture: Trustzone for aarch64. https://developer.arm.com/architectures/learn-the-architecture/trustzone-for-aarch64/trustzone-in-the-processor, 2021.

[8] Arm. Fast Models Fixed Virtual Platforms (FVP), Reference Guide. https://developer.arm.com/documentation/100966/1121/?lang=en, 2023.

[9] Arm. Arm A-profile A64 Instruction Set Architecture: AES instructions. https://developer.arm.com/documentation/ddi0602/2022-06/

SIMD-FP-Instructions/AESE--AES-single-round-encryption-, accessed 2023-05-04.

[10] Arm. Arm Support Forum, How to Benchmark code. https://community.arm.com/support-forums/f/soc-design-and-simulation-forum/52179/how-to-benchmark-code-on-fvp_mps2_cortex-m4-simulator, accessed 2023-05-04.

[11] Arm. Arm Trusted Firmware TFA. https://trustedfirmware-a.readthedocs.io/en/latest/, accessed 2023-05-04.

[12] Arm. Fast models fixed virtual platforms (fvp) reference guide. https://developer.arm.com/documentation/100966/1111-00/Base-Platform-FVPs/FVP-Base-RevC-2xAEMv8A, accessed 2023-05-04.

[13] Arm. Fast models reference guide: Fast Models portfolio. https://developer.arm.com/documentation/100965/1117/Introduction-to-Fast-Models/What-does-Fast-Models-consist-of-/Fast-Models-portfolio, accessed 2023-05-04.

[14] Arm. Fast models reference guide: ToggleMTIPlugin. https://developer.arm.com/documentation/100964/1121/Plug-ins-for-Fast-Models/ToggleMTIPlugin, accessed 2023-05-04.

[15] Arm. Fast Models Reference Manual: Crypto Plugin. https://developer.arm.com/documentation/100964/1114/Plug-ins-for-Fast-Models/Crypto, accessed 2023-05-04.

[16] Arm. Fast models user guide: Graphics acceleration in fast models. https://developer.arm.com/documentation/100965/1119/Graphics-Acceleration-in-Fast-Models/Introduction-to-GGA?lang=en, accessed 2023-05-04.

[17] Arm. Fast Models User Guide: Timing Annotations. https://developer.arm.com/documentation/100965/1121/Timing-Annotation, accessed 2023-05-04.

[18] Arm. FVP model capabilities. https://developer.arm.com/documentation/100964/1116/Introduction-to-the-Fast-Models-Reference-Manual/Model-capabilities, accessed 2023-05-04.

[19] Arm. Support for Arm CCA VMs on Linux. https://lwn.net/Articles/921482/, accessed 2023-05-04.

[20] Arm. Support for arm cca vms on linux. https://gitlab.arm.com/linux-arm/kvmtool-cca, accessed 2023-05-04.

[21] Arm. Trusted firmware implementation of the realm management monitor (rmm). https://www.trustedfirmware.org/projects/tf-rmm/, accessed 2023-05-04.

[22] Arm. Mali OpenGL Emulator. https://developer.arm.com/documentation/100965/1119/Graphics-Acceleration-in-Fast-Models/Enabling-GGA/Install-the-Arm--Mali-OpenGL-ES-Emulator?lang=en, accessed 2023-05-10.

[23] Arm. Learn the Architecture - AArch64 Virtualization, Stage 2 Translations. https://developer.arm.com/documentation/102142/0100/Stage-2-translation, accessed 2023-05-20.

[24] Arm. AArch64 Exception Model. https://developer.arm.com/documentation/102412/latest, Issue, 0103-01, accessed 2023-05-15.

[25] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Savannah, GA, November 2016. USENIX Association.

[26] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. CURE: A security architecture with customizable and resilient enclaves. In *USENIX Security 21*, 2021.

[27] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Cure: A security architecture with customizable and resilient enclaves. In *USENIX Security Symposium*, pages 1073–1090, 2021.

[28] OpenMP Architecture Review Board. OpenMP open multi-processing. https://www.openmp.org/, accessed 2023-05-04.

[29] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *NDSS*. NDSS, 2019.

[30] Buildroot. Buildroot - Making Embedded Linux Easy. https://buildroot.org/, accessed 2023-05-10.

[31] 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.

[32] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. *SIGOPS Oper. Syst. Rev.*, 42(2):2–13, March 2008.

[33] C.C. Consortium. Confidential Computing Consortium: A technical analysis of confidential computing, a publication of the confidential computing consortium. https://confidentialcomputing.io/wp-content/uploads/sites/10/2023/03/CCC-A-Technical-Analysis-of-Confidential-Computing-v1.3_unlocked.pdf, November 2022, v1.3.

[34] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[35] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*, August 2007.

[36] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, 2016.

[37] René de Jong and Andreas Sandberg. Nomali: Simulating a realistic graphics driver stack using a stub gpu. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 255–262, 2016.

[38] 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.

[39] Nvidia Developer. CUDA zone, parallel computing platform. https://developer.nvidia.com/cuda-zone#:~:text=CUDA%C2%AE%20is%20a%20parallel,harnessing%20the%20power%20of%20GPUs., accessed 2023-05-04.

[40] Romain Dolbeau. Aes-ctr functions for cuda. https://web.archive.org/web/20221127200344/http://dolbeau.name/dolbeau/crypto/crypto.html, 2022.

[41] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the penglai enclave. In *OSDI*, pages 275–294, 2021.

[42] Gem5. gem5: The gem5 simulator system. https://www.gem5.org/, 2022.

[43] Google. Google cloud confidential virtual machines. https://www.wired.com/story/google-cloud-confidential-virtual-machines/, accessed 2023-05-10.

[44] Google. gVisor - The Container Security Platform. https://gvisor.dev/docs/, accessed 2023-05-10.

[45] Google. Sandbox2. https://developers.google.com/code-sandboxing/sandbox2, accessed 2023-05-10.

[46] GPGPU-Sim Project. gpgpu-sim/gpgpu-sim_distribution: GPGPU-Sim provides a detailed simulation model of contemporary NVIDIA GPUs running CUDA and/or OpenCL workloads. It includes support for features such as TensorCores and CUDA Dynamic Parallelism as well as a performance visualization tool, AerialVisoin, and an integrated energy model, GPUWattch. https://github.com/gpgpu-sim/gpgpu-sim_distribution, accessed 2023-05-10.

[47] Khronos group. OpenCL open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/, accessed 2023-05-04.

[48] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. Inktag: Secure applications on an untrusted operating system. *SIGPLAN Not.*, 48(4):265–278, March 2013.

[49] Guerney D. H. Hunt, Ramachandra Pai, Michael V. Le, Hani Jamjoom, Sukadev Bhattiprolu, Rick Boivie, Laurent Dufour, Brad Frey, Mohit Kapur, Kenneth A. Goldman, Ryan Grimm, Janani Janakirman, John M. Ludden, Paul Mackerras, Cathy May, Elaine R. Palmer, Bharata Bhasker Rao, Lawrence Roy, William A. Starke, Jeff Stuecheli, Enriquillo Valdez, and Wendel Voigt. Confidential computing for openpower. EuroSys '21.

[50] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J Rossbach, and Emmett Witchel. Telekine: Secure computing with cloud GPUs. In *NSDI*, 2020.

[51] Intel. Intel software guard extensions. https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html.

[52] Intel. Intel trust domain extensions (intel tdx). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[53] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous isolated execution for commodity gpus. In *ASPLOS*, 2019.

[54] Masami Hiramatsu Jim Keniston Prasanna S. Panchamukhi. Kernel Probes (Kprobes). https://docs.kernel.org/trace/kprobes.html, accessed 2023-05-10.

[55] Luyi Kang, Yuqi Xue, Weiwei Jia, Xiaohao Wang, Jongryool Kim, Changhwan Youn, Myeong Joon Kang, Hyung Jin Lim, Bruce Jacob, and Jian Huang. Iceclave: A trusted execution environment for in-storage computing. In *IEEE/ACM MICRO*, 2021.

[56] Shinpei Kato. Implementing Open-Source CUDA Runtime. https://ipsj.ixsq.nii.ac.jp/ej/?action=repository_action_common_download&item_id=95535&item_no=1&attribute_id=1&file_no=1, 2013.

[57] Shinpei Kato. gdev-bench: Rodinia benchmarks. https://github.com/shinpei0208/gdev-bench/tree/master/rodinia/cuda, accessed 2023-05-04.

[58] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-Class GPU resource management in the operating system. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 401–412, Boston, MA, June 2012. USENIX Association.

[59] Linux Kernel. fd based Guest Private memory for KVM. https://lore.kernel.org/all/20221202061347.1070246-1-chao.p.peng@linux.intel.com/T/#u, accessed 2023-05-10.

[60] Linux Kernel. VFIO - ”Virtual Function I/O”. https://docs.kernel.org/driver-api/vfio.html, accessed 2023-05-10.

[61] Linux Kernel. Virtio on Linux. https://docs.kernel.org/driver-api/virtio/virtio.html#virtio-on-linux, accessed 2023-05-10.

[62] Shan-Chyun Ku. What's in risc-v iopmp. RISC-V Forum: Security, 2021.

[63] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *EuroSys*, 2020.

[64] Dingji Li, Zeyu Mi, Yubin Xia, Binyu Zang, Haibo Chen, and Haibing Guan. Twinvisor: Hardware-isolated confidential virtual machines for arm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 638–654, New York, NY, USA, 2021. Association for Computing Machinery.

[65] M. Schwarz M. Lipp and C. Canella. misc0110/PTEditor: A small library to modify all page-table levels of all processes from user space for x86_64 and ARMv8. https://github.com/misc0110/PTEditor, accessed 2023-05-04.

[66] Jämes Mé nétrey, Christian Göttel, Anum Khurshid, Marcelo Pasin, Pascal Felber, Valerio Schiavoni, and Shahid Raza. Attestation mechanisms for trusted execution environments demystified. In *Distributed Applications and Interoperable Systems*, pages 95–113. Springer International Publishing, 2022.

[67] Microsoft. Azure confidential cloud - protect data in use — microsoft azure. https://azure.microsoft.com/en-us/solutions/confidential-compute/.

[68] The nouveau project. nouveau/ feature matrix. https://nouveau.freedesktop.org/FeatureMatrix.htmls, accessed 2023-05-04.

[69] Nvidia. Nvidia multi-instance gpu user guide :: Nvidia tesla documentation. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/.

[70] Nvidia. NVIDIA Linux open GPU kernel module source. https://github.com/NVIDIA/open-gpu-kernel-modules, accessed 2023-05-04.

[71] Nvidia. CUDA Driver vs Runtime API. https://docs.nvidia.com/cuda/cuda-runtime-api/driver-vs-runtime-api.html, accessed 2023-05-10.

[72] oasis open. Virtual I/O Device (VIRTIO) . https://docs.oasis-open.org/virtio/virtio/v1.2/csd01/virtio-v1.2-csd01.html, accessed 2023-05-10.

[73] Oracle. Virtualbox. https://www.virtualbox.org/, accessed 2023-05-10.

[74] GNU project. Linux Programmer's Manual: ld.so. https://man7.org/linux/man-pages/man8/ld.so.8.html, accessed 2023-05-10.

[75] Gramine Project. gramine - A library OS for Linux multi-process applications, with Intel SGX support. https://github.com/gramineproject/gramine, accessed 2023-05-10.

[76] SimEng Project. The Simulation Engine - SimEng. https://uob-hpc.github.io/SimEng/, accessed 2023-05-10.

[77] QEMU. QEMU - A generic and open source machine emulator and virtualizer. https://www.qemu.org/, accessed 2023-05-10.

[78] Samsung. ISLET. https://github.com/Samsung/islet, accessed 2023-05-04.

[79] Moritz Schneider, Aritra Dhar, Ivan Puddu, Kari Kostiainen, and Srdjan Čapkun. Composite enclaves: Towards disaggregated trusted execution. *IACR CHES*, 2022.

[80] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, mar 2020.

[81] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing tcb complexity for security-sensitive applications: Three case studies. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, page 161–174, New York, NY, USA, 2006. Association for Computing Machinery.

[82] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, Mark Kuhne, Fabio Aliberti, and Shweta Shinde. Acai: Extending arm confidential computing architecture protection from cpus to accelerators. https://arxiv.org/abs/2305.15986, 2023.

[83] Dat Le Tien, Shweta Shinde, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *Network and Distributed System Security Symposium (NDSS)*, March 2017.

[84] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2sim: A simulation framework for cpu-gpu computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, page 335–344, New York, NY, USA, 2012. Association for Computing Machinery.

[85] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on gpus. In *USENIX OSDI*, 2018.

[86] Julien Thierry Will Deacon. kvmtool: Stand-alone Native Linux KVM Tool. https://github.com/kvmtool/kvmtool, accessed 2023-05-04.

[87] Alex Williamson. vfio: Include No-IOMMU mode. https://lwn.net/Articles/660745/, accessed 2023-05-04.

[88] Yubin Xia, Yutao Liu, and Haibo Chen. Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 246–257, 2013.

[89] Xilinx. Dma/bridge subsystem for pci express. https://docs.xilinx.com/r/en-US/pg195-pcie-dma/Introduction, accessed 2023-05-04.

[90] Mark Zhao, Mingyu Gao, and Christos Kozyrakis. Shef: shielded enclaves for cloud fpgas. In *ACM ASPLOS*, 2022.

[91] Zilinx. Vitis_libraries/security at main - xilinx/vitis_libraries - github. https://github.com/Xilinx/Vitis_Libraries/tree/main/security, 2022.