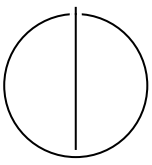
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# Protecting H/W and S/W Interactions for Network-Attached Accelerators

Maximilian Jäcklein



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# Protecting H/W and S/W Interactions for Network-Attached Accelerators

# Sichern der H/W und S/W Interaktionen von Netzwerk-verbundenen Beschleunigern

| | |
|---|---|
| Author: | Maximilian Jäcklein |
| Examiner: | Prof. Dr. Pramod Bhatotia |
| Supervisor: | Harshavardhan Unnibhavi |
| Submission Date: | 14.08.2025 |

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 14.08.2025 Maximilian Jäcklein

# Acknowledgments

I would like to especially thank Harshavardhan Unnibhavi, for his guidance and support throughout the entire duration of this thesis. Our frequent talks solved many of my problems and helped me to stay on track.

I also want to express my gratitude to Prof. Pramod Bhatotia and the Systems Research Group for giving me the chance to work on such an interesting topic, hence allowing me to pursue a field I have learned to appreciate during my studies.

# Abstract

The concept of disaggregation, where different hardware resources of a data center are placed in dedicated pools of CPU, memory, storage, and accelerators, is a promising architecture to increase flexibility, scalability, and resource utilization. At the same time, protecting data from unauthorized access is an important requirement for many modern workloads (e.g., health care). However, existing approaches can guarantee security only for a centralized server architecture and therefore come short of supporting disaggregated accelerators.

This bachelor's thesis addresses that gap by designing and implementing the software stack of a secure disaggregated accelerator setup. Only making changes to the confidential virtual machine's kernel and hypervisor, and delegating the network-related tasks to a user space application outside of the trusted environment, this approach requires no changes to the existing accelerator software stack by implementing a unified interface for various types of devices. During that time when the communication is necessarily exposed to a potentially malicious entity (e.g., when sending over a network), all relevant data is protected by an authenticated Encryption.

The source code can be found at: `https://github.com/harshanavkis/jigsaw-overall/pull/2`.

# Contents

**Acknowledgments** **iii**

**Abstract** **iv**

**1 Introduction** **1**

**2 Background** **3**
2.1 I/O Device Workflow . . . 3
2.1.1 PCI Configuration Space . . . 3
2.1.2 Software-Hardware Device Communication . . . 4
2.2 RDMA . . . 8
2.3 Confidential Computing . . . 9

**3 Overview** **10**
3.1 Workflow . . . 11
3.2 System Architecture . . . 12
3.2.1 Guest Kernel Modifications . . . 12
3.2.2 Shared Memory Application . . . 12
3.2.3 Security Controller . . . 13
3.3 Design Goals . . . 13
3.4 Threat model . . . 14

**4 Design** **16**
4.1 Components . . . 16
4.1.1 Pseudo Device . . . 17
4.1.2 Shared Memory . . . 17
4.1.3 Guest Kernel Extensions . . . 18
4.1.4 Communication Controller . . . 19
4.1.5 Proxy . . . 20
4.1.6 Security Controller . . . 20
4.1.7 Device Emulation . . . 20
4.2 MMIO & DMA Protocol . . . 21
4.3 Securing the Communication . . . 23

**5 Implementation** **24**
5.1 Low-Level Implementation Components . . . . . 24
5.1.1 Pseudo Device . . . . . 24
5.1.2 Shared Memory . . . . . 24
5.1.3 MMIO Message Structure . . . . . 25
5.1.4 Linux Guest Kernel . . . . . 25
5.2 Cryptography . . . . . 27
5.3 Network Implementations . . . . . 27

**6 Evaluation** **29**
6.1 Setup . . . . . 29
6.2 DMA Throughput . . . . . 30
6.3 MMIO Latency . . . . . 32
6.4 Bounce-Buffer Copy Overhead . . . . . 33
6.5 Cryptography Library Performances . . . . . 34
6.6 Network Overhead . . . . . 34
6.6.1 Throughput . . . . . 35
6.6.2 Latency . . . . . 36
6.7 Discussion . . . . . 36

**7 Related Work** **38**
7.1 CPU-Attached Trustworthy Accelerators . . . . . 38
7.2 Network-attached Accelerators . . . . . 39

**8 Summary and Conclusion** **40**

**9 Future Work** **41**

**Abbreviations** **43**

**List of Figures** **44**

**List of Tables** **45**

**Bibliography** **46**

# 1 Introduction

Disaggregated architectures are increasingly employed in data centers (DC) due to their scalability, resource utilization, and flexibility. The shift from monolithic servers and racks to disaggregation involves the decomposition of different hardware resource types into pools of CPU, storage, memory, and accelerators, which are connected via the DC's network. [25, 32]

That architecture is especially interesting for modern high-workload and data-intensive applications, which utilize the benefits of accelerators. The resulting management issues related to the heterogeneity of the various types of accelerators (e.g., GPUs [8], TPUs [10], FPGAs [4]) can be mitigated by the concept of disaggregation.

Additionally, if many computing tasks that involve sensitive data (e.g., health care [13, 1] or finances [11]) are shifted from on-premise solutions to DC's services, the demand for protection of data during all stages of computing is increasing [42].

Consequently, the combination of both characteristics, namely disaggregation and confidential computing, is a highly present object for DC architectures.

Initial server architectures employ hardware-supported Trusted Execution Environment (TEE) techniques; for example, Intel's TDX [9] or AMD's SEV [2] allow the user to run isolated Confidential Virtual Machines (CVMs) on otherwise untrustworthy hardware, thereby providing protection from other tenants and even the DC provider. One design to include PCIe-attached devices into the TEE of a CVM is TDISP, a protocol introduced by PCI-SIG into the existing PCIe interconnect bus [41]. However, those approaches all lack the characteristics of disaggregated devices as they are based on a monolithic/CPU-centric architecture.

Most previous work on disaggregation has focused on enabling (efficient) access to remote memory pools [27, 22, 15, 28], but some research has also explored the disaggregation of accelerators [39, 16, 17]. Those approaches either operate at rack-scale or do not support any kind of secure computing without requiring heavy changes to the existing software/hardware stack.

To fill this gap, this thesis aims to include a network-attached accelerator into the secure domain of a CVM by providing the software design and implementation. The existing software (i.e., the driver) and hardware (i.e., accelerator) should not be part of any changes and should be agnostic of their special connection over the network. To achieve these goals, we contribute:

1. Modifications to the CVM's Linux kernel I/O communication layer. Intercepting, converting, and encrypting driver-device communication into a specific protocol.
2. A Proxy instance on the CVM server, adding support for networking interactions with a disaggregated device, involving transportation of types TCP, RDMA, and Ethernet.
3. A simplified device emulation, capable of responding to requests over the network. Used to test and evaluate the implementation.

This setup is designed to easily incorporate the features 1 and 2 into a real disaggregated setup, where the pool of accelerators is managed by a dedicated hardware security controller.

The evaluation of the implementation shows the register-size accesses to have about a 3.2x higher latency on a disaggregated setup with RDMA (which performs best among the transportation types) compared to a same-host setup (the device emulation runs on the CVM host). The exchange of bigger-sized data achieves a throughput that is about 2x higher. The main limitations lie in the latency that is imposed by the network stacks; consequently, this also reduces the ability to exchange data at a high throughput rate.

This work is structured to first provide the necessary background information in chapter 2. The following chapter 3 introduces the design and provides a high-level overview of the components and the system's workflow. Chapter 4 details the introduced concepts of the previous chapter and specifies the protocol of the disaggregated communication. Chapter 5 thoroughly examines the implementation of the system and its various forms of networking communication. The implemented setups are then evaluated in chapter 6 with benchmarks, testing the performance and efficiency of the design. Chapter 7 compares to existing work, chapter 8 summarizes the contributions and findings of this thesis, with chapter 9 providing a look at possible future projects.

# 2 Background

This chapter provides background information on the mechanisms and technical standards that are necessary to understand the system as a whole and its implementation in particular. First, it focuses on the workflow of an I/O device, from being discovered by the system firmware to communicating with a dedicated driver through the established methods. Second, it gives a quick overview of RDMA, one of the transportation types used in the implementation. Third, a short introduction to confidential computing and some of its implementations is provided.

## 2.1 I/O Device Workflow

This section summarizes the workflow of a Peripheral Component Interconnect (PCI) device. Upon initializing the system, the BIOS, or its modern replacement UEFI, registers the connected devices and assigns memory areas to them based on their specific requests [44, pp. V–117]. The type and size of the requested memory is first retrieved by utilizing the **Configuration Space** of a PCI device (see 2.1.1). The very same space is then used to return the assigned memory. After booting, all accesses falling into this area will be noticed by the device and are handled accordingly. From the driver side, one technique to access such a region is called **Memory-Mapped I/O** (see 2.1.2). Another device-driver communication technique, which is more efficient, dynamic, and does not require initialization by the firmware, is called **Direct Memory Access** (see 2.1.2). This technique permits sharing a memory area between the driver and the device.

### 2.1.1 PCI Configuration Space

This section focuses on the configuration space of a PCI device, which is 256 bytes long. The replacement of PCI, which is called PCIe, extends this configuration space to 4096 bytes, but retains the first 256 bytes for backward-compatibility [26, p. 942- 943]. The fields covered in this section are part of the PCI-compatible area.

The configuration space of a PCI device allows it to be identified, to retrieve and serve its memory requests, and provide other basic configuration mechanisms, like setting up interrupts or enabling it to respond to memory accesses. All PCI devices have to

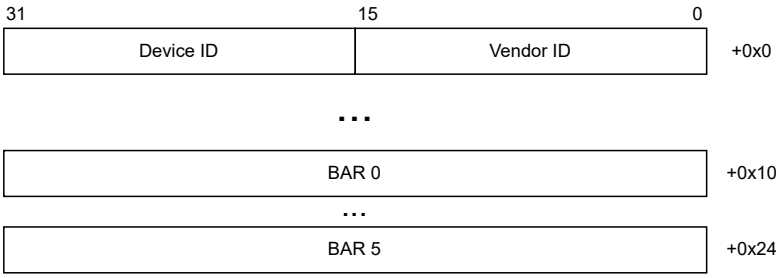



Figure 2.1: Simplified structure of PCI configuration space [26, p. 965]. Only the relevant fields for this thesis are shown in the figure.

provide this space to offer a standardized configuration mechanism. A configuration space can either be accessed by port I/O or the PCIe-specific Enhanced Configuration Access Mechanism (ECAM), which maps the configuration space into the system's memory. A PCI device's configuration space is utilized by the UEFI (during the boot process, when enumerating the devices and assigning memory) and the OS (e.g., when identifying the device to load a corresponding driver). [26, p. 943-967]

Figure 2.1 shows some parts of the PCI configuration space. Namely, the Device/Vendor ID starting from offset $0x0$, and six contiguous Base Address Register (BAR)s beginning at offset $0x10$. Those are the only fields relevant for our implementation. The vendor and device ID fields, which are 16 bits each, contain the necessary information to identify the manufacturer and the device. These fields are used by the kernel to choose and load the right driver. The 32-bit BAR fields contain information about the device's assigned memory range. When enumerating the devices during system boot up, the UEFI obtains the device's requested sizes by decoding the values from the BARs. Based on the size, a suitable memory region is determined and assigned to the device by encoding the region's address back into the BAR. To what purpose a BAR is used is device-dependent and is typically specified. Typical usage of one of the BARs is to provide access to the device's control registers through it. Therefore, accessing this BAR region will be converted to direct register access on the device. As a configuration space is device-specific, PCI allows up to 6 BARs for each device. [26, p. 953-967]

### 2.1.2 Software-Hardware Device Communication

Communication between the driver and its device occurs predominantly via two methods. First, Memory-Mapped Input/Output (MMIO) allows small register-size accesses, typically used to control the device and check its status. Second, Direct Memory Access (DMA) enables larger data exchanges through a shared memory area. The following subsections introduce those two essential methods of driver-device

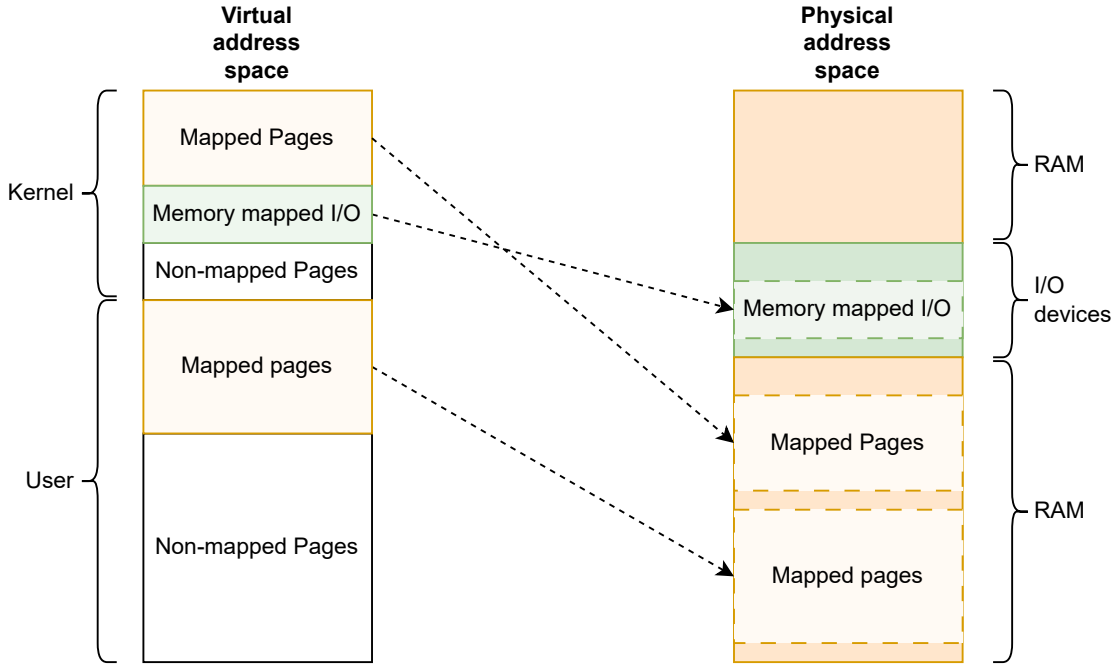



Figure 2.2: Overview of virtual and physical address space in the MMIO case. Normally, the I/O section in the physical address space would probably start at address $0x0$ [40, p. 342]. This figure should emphasize, that there is no overlapping of I/O and RAM, and those regions do not have to be mapped contiguous.

communication in the case of an PCI device. As the presented system implements both methods in a disaggregated version, a basic understanding of them is essential for further reading.

### Memory-Mapped Input/Output

Access to device registers is essential for driver programming. Only dynamic interaction allows a proper workflow with a peripheral device. For example, it permits initiating a computation by writing to a command register or checking its progress by reading from a specific status register. Additionally, register access is necessary for initiating a DMA transfer, which is essential for this implementation. Consequently, an easy and portable method for small-sized accesses on the device side is needed. There are two established types of I/O device access.

The first method relies on I/O ports and thus on dedicated CPU instructions (e.g., `IN` and `OUT` on x86_64). This not only results in a second address space next to the physical memory address space, namely the port space, but also requires programming

with the CPU's machine code instructions, which is not necessarily supported by high-level programming languages. Consequently, this method is not suited for easy programmability, and another approach is required. [40, p. 341]

The second method is called MMIO. This mechanism exposes the device's registers and internal memory to the CPU by mapping them into the physical address space of the system (see right side of figure 2.2). To make this work, the memory and I/O devices share a common memory bus. The mapped region does not overlap with main memory, but has its unique address area within the physical address space. After the kernel establishes a translation from virtual to physical addresses (see left and middle of figure 2.2), the user can access this mapped I/O region with the same instructions as when accessing main memory. This way, device interaction involves no dedicated I/O machine instructions, which are also rather limited on most CPU architectures. In contrast, normal memory access instructions are plentiful, making the programming easier and allowing for better optimization. [40, p. 341-342][37, p. 261-262][36, p. 531-533]

Accessing a mapped region leads to direct access on the device. The conversion of a memory access to a device access is not part of the software, but is implemented through bus operations. The device knows its assigned addresses in the physical address space and compares the bus activities with its addresses. Thus, when having a match, it can process the request and potentially respond. [40, p. 342]

Overall, this mapping of device registers into virtual address space abstracts the method of accessing a device's memory from the driver level, as a direct device access is now the same as accessing a normal memory region [40, p. 342]. Additionally, it is also faster than using multiple I/O instructions [36, p. 532].

One drawback of MMIO on 32-bit systems is its occupation of valuable addresses. Because it is mapped into the same address space as main memory, it reduces the already limited available space for normal memory regions [24, p. 185]. Modern 64-bit systems do not have this problem due to their vast availability of addresses.

From the perspective of a PCI device, a memory-mapped region corresponds to a BAR of its configuration space (see 2.1.1). A driver can map a BAR by issuing `pci_iomap()` in the kernel [19, `include/asm-generic/pci_iomap.h`], leading to the mapping of the assigned physical address of the BAR into the virtual address space. Normally, the BAR address has been previously read by the kernel from the BAR during device enumeration.

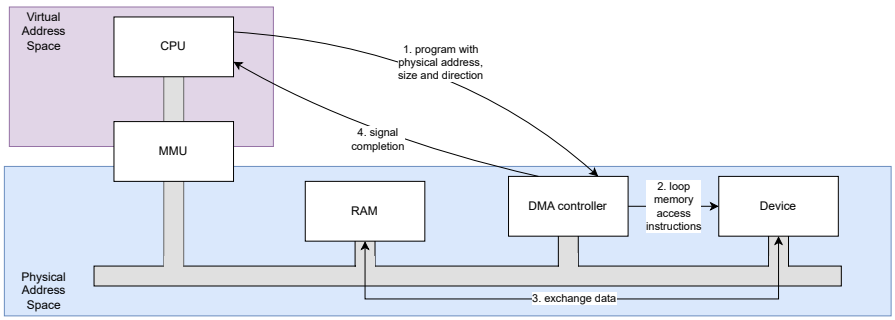



Figure 2.3: Simplified DMA operational workflow. Inspired by [40, p. 345]

### Direct Memory Access

While MMIO allows for the exchange of data at the register size level, data transfers spanning multiple bytes are not efficiently handled with this method. For this case, the concept of DMA offers an alternative [36, p. 538]. It typically involves bigger transfer sizes and offers more flexibility than when relying solely on MMIO.

DMA refers to the delegation of I/O work to a dedicated hardware controller. This DMA controller can be programmed with the device identification, size, address, and direction of the DMA transfer; it controls the I/O operation while the CPU only needs to provide the initial instruction, allowing the CPU to turn to other tasks during the transfer. Upon having transferred the requested number of bytes, the controller normally issues an interrupt signal to inform the CPU about the transfer's completion. [38, p. 508-510][36, p. 538-540][40, p. 345]

The DMA controller is either an extra general hardware device for multiple I/O devices, able to accept various DMA transfer requests at once, or part of the I/O device itself, therefore only managing this one device. A DMA controller saves the address, size, and direction transfer in its own registers, which can be programmed by the CPU; an additional register for the identification of the involved device is provided in case of a general DMA controller. [40, p. 344]

Figure 2.3 provides a high-level overview of the system bus and the connected components. In this figure, the bus is simplified to be used universally for all types of exchanges.

Normally, the Memory Management Unit (MMU) is part of the CPU and not an external hardware device. Thus, the DMA controller cannot work in the virtual address space of the CPU and has to be provided with valid physical addresses. Consequently, before issuing a DMA command to the controller, the driver (i.e., the kernel) has to convert the virtual address of the DMA memory region to the corresponding physical address. [40, p. 346]

To allow a separate entity, namely the DMA controller, to access the memory bus and not interfere with the CPU's bus interactions, bus accesses have to be synchronized. The DMA controller can "steal a cycle" to prevent the CPU from reading or writing to the memory bus for the time it is used for a DMA transfer. [37, p. 273][24, p. 335][36, p. 539]

#### Linux Kernel DMA-API

DMA transfers require the support of the OS to provide a suitable memory region and a corresponding address that is valid in the DMA controller's address space. The provided address can then be used to program the controller for DMA transfers. Therefore, the Linux Kernel has an extensive DMA-API, which will partly be described in the following paragraphs.

The kernel differentiates between coherent and streaming DMA. The first ensures that all modifications to a DMA region are directly visible to the other participant. It does not require dedicated synchronization requests by the driver. The latter refers to DMA operations requiring an explicit synchronization instruction by the driver to keep the DMA region identical for both the device and the driver. [18]

The following chapters will always refer to the streaming API when mentioning DMA operations in the Linux kernel unless specified otherwise. Only a short paragraph in chapter 9 will pick up coherent DMA again.

To make a buffer available for DMA, a driver has to call `dma_map_single(void *)` with a virtual address that points to a physically contiguous memory region. The returned value is a valid DMA address and, as such, can be provided to the DMA controller as the source or destination (depending on the direction) of the transfer. To synchronize the DMA buffer, the kernel provides `dma_sync_single_for_cpu(dma_addr)`, which updates the local buffer of the driver with the changes made by the device, and `dma_sync_single_for_device(dma_addr)`, which does the same for the device to synchronize with the version of the driver's local buffer. [18]

## 2.2 RDMA

Remote Direct Memory Access (RDMA) is a protocol allowing efficient and fast networking [33]. The concept of RDMA evolved around the idea to make network interfaces directly available to the user-space, therefore reducing overhead through circumventing system calls into the kernel and reducing copies by directly sending the data from one user-space to the other [7]. This is related to the idea of DMA, where the CPU is not involved in the memory transfers, hence the name. RDMA technology is especially useful in data centers as it provides high throughput, reduced CPU usage, and low

latency; as a result, performing better than the established TCP/IP stack in those metrics [47].

The main operations of RDMA include `read()/write()` and `send()/recv()`. The first allows direct access to a remote system's memory without an additional copy. Those functions operate on previously registered memory regions and do not even require the direct involvement of the peer except for the initial registration. The latter are comparable to the common `send()/recv()` system calls on Linux and allow for the exchange of low-latency messages. [33]

## 2.3 Confidential Computing

Confidential computing protects data while being used in a computation. The term is closely related to Trusted Execution Environment (TEE), which is a secure area, not accessible by outside entities. The security goals of TEEs involve data integrity and confidentiality, and code integrity. [42]

One implementation of a TEE is the Confidential Virtual Machine (CVM). A CVM is protected from every other user on the machine, including the hypervisor and host OS. Through this abstraction, the user of a CVM can run any existing application without requiring modifications to the application itself. The hardware support is provided in the form of AMD SEV-SNP and Intel TDX for the major CPU vendors. [23] [2] [9]

# 3 Overview

This chapter provides an overview of the system's design and workflow. Built for secure driver access to a network-attached accelerator, this thesis contributes the software implementation of the system. It extends the guest Linux kernel, makes small modifications to the hypervisor, and adds a host user-space application that moves selected data between the CVM's trusted domain and a potentially compromised host system over the network to a disaggregated device. By implementing those concepts, it does not require changes to the driver software and acts transparently to provide secure MMIO and DMA communication over an insecure network. Additionally, a simple software device emulation is provided, allowing for testing and evaluating the implementation.

Figure 3.1 gives a high-level overview of the system's major components and their relation to each other. Besides the CVM, its host contains another structure, the Shared Memory. This component connects to a security controller on the remote host via the network. The controller interfaces with a software process that is emulating an MMIO-/ and DMA-capable PCI device.

The shared memory structure is a user space application and memory region that is used to extract confidential data from, or insert into, the CVM's TEE. Consequently, with this structure not being part of the trusted domain, the data that is leaving the CVM has to undergo some cryptographic transformation before being exposed to other users on the system. Data that is obtained by the shared memory will be sent over the network and received by the security controller. This controller converts the data to a device-specific message and forwards it to the emulation process. A response is provided to the CVM in the reverse way.

This software system is designed to be easily migrated to a real hardware accelerator setup. It is co-designed with a dedicated hardware security controller, which interfaces with real accelerators, by implementing the same protocol. Thus, the design allows for easy replacement of the device emulation with real hardware, requiring no further changes to the CVM host. The designed protocol is agnostic of the specific accelerator type and provides a unified interface for all devices and drivers.

The next sections will first present the high-level operational workflow. Next, the individual components are described. The chapter ends with the design goals and the threat model.

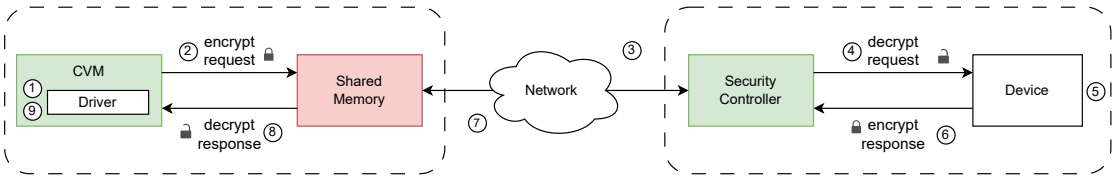



Figure 3.1: High-level Overview of the System

## 3.1 Workflow

Based on figure 3.1, this section will summarize the workflow of the system. The numbers in the circles refer to the items in the following enumeration.

1. **Communication Attempt** Upon trying to communicate with the disaggregated device, the Driver's execution is interrupted by the guest kernel. The information about the communication attempt is converted into a protocol message.
2. **CVM Extraction** The kernel encrypts the message and equips it with a cryptographic authentication code. It is forwarded to the Shared Memory, thereby leaving the trusted domain of the CVM.
3. **Request Transmission** The Shared Memory sends the encrypted message over the network to the Security Controller by using a previously established connection.
4. **To-Device Communication** The Security Controller decrypts the message to its plain-text version while also verifying the authentication code. The protocol message is transformed to a device-specific format and given to the Device.
5. **Remote Device Access** The Device will interpret the access and potentially change its state or start a computation; in some cases, the Device returns a value.
6. **From-Device Communication** The Security Controller receives the response data to the request. The Controller converts it into a protocol message and encrypts it, again adding an authentication code.
7. **Response Reception** The encrypted message is sent over the established network connection and received by the communication endpoint of the Shared Memory.
8. **CVM Insertion** The Shared Memory informs the kernel of the message reception. The kernel reads the ciphertext back into its TEE, checks the authentication code, and decrypts the message to its original plaintext.
9. **Driver Execution Resumption** The kernel provides the response as a return value to the Driver and resumes its execution.

This runtime workflow abstracts away the differences between MMIO and DMA. Furthermore, it omits the initialization phase, which includes the symmetric key exchange for the cryptographic operations; in addition, the setup of the network

connection between the Shared Memory and Security Controller is left out. A more detailed description and distinction are provided in the next two chapters.

## 3.2 System Architecture

This section explains some of the major changes and components, providing context on their tasks and roles within the system.

### 3.2.1 Guest Kernel Modifications

The modifications to the guest kernel, which is running inside the CVM, are allowing drivers (without changes to their software) to communicate with a disaggregated device via MMIO and DMA. All accesses to a memory-mapped region that corresponds to a BAR (see 2.1.1) of a network-attached PCI device are intercepted by the kernel. The execution of the driver is stopped, and the MMIO access is converted to a message that contains information about the type and address of the attempted access. This message is encrypted and provided to the Shared Memory, which is available to the kernel even inside the protected environment of the CVM. DMA operations follow the same procedure by being intercepted when calling an API function.

### 3.2.2 Shared Memory Application

The Shared Memory application serves as a communication bridge between CVM and a non-TEE host. Its main task is to provide a channel into the isolated CVM, thereby *sharing memory* between the host and the CVM. This is normally not supported by the TEE hardware extensions and is solved by utilizing a feature of QEMU.

The Shared Memory is divided into MMIO and DMA functionality. MMIO messages are exchanged via a message passing mechanism and, therefore, require active involvement of the Shared Memory Application. DMA has its dedicated region in the Shared Memory, which can be further decomposed into smaller allocatable blocks. The data in those blocks can be requested over the network and will, subsequently, be sent as a response.

Furthermore, the Shared Memory is one participant in the network communication. By following a specified protocol, it sends MMIO and DMA-related data over the network. During this whole process, it has no access to the plain-text values and operates solely on the basis of the protocol.

### 3.2.3 Security Controller

The Security Controller is the other participant in the network communication. It handles the cryptographic tasks and converts protocol messages into accesses on the emulated device. Future versions of this system will replace this software implementation of the Controller with a dedicated hardware device, which will interface with real accelerators. The software Security Controller, in conjunction with the device emulation, was implemented to test the protocol, to provide a setup to evaluate the system, and to detect possible flaws.

## 3.3 Design Goals

This software system was designed within the boundaries of a clearly defined set of goals. They dictated the structure of the protocol and set guidelines for the extensions and modifications to the existing software. The main goals are listed in this section.

### Security

Our disaggregated design requires some specific data, for example, MMIO access information or DMA synchronization instructions, to leave the secure isolation of the CVM while traveling to the remote accelerator. Therefore, security guarantees have to be provided for data that is traversing the network or being exposed by unprotected applications, such as the Shared Memory.

**Confidentiality** Only the guest kernel and the Security Controller should be able to read the content of protocol messages. As a result, all data has to be encrypted and exposed exclusively in its ciphertext form to unauthorized entities.

**Integrity** If the exchanged messages are modified during transit, the system has to detect it. When encountering corrupted ciphertext, the resulting plaintext must not be used as a valid output, but rather lead to the failure of the communication, as the real data can not be reconstructed. Providing a fallback in case of such a failure and, therefore, ensuring the continuation of disaggregated communication is not part of this thesis.

**Freshness** Information or data that has been part of a previous communication must not be reused. All attempts to disguise some old data as a fresh value are to be detected and not processed further. Therefore, to prevent replay attacks, the system must implement a mechanism that allows for checking the freshness of exchanged data on a per-message basis.

### Transparency for Accelerator Software/Hardware

A driver must be able to assume a CPU-attached accelerator. It must not know about the disaggregation of the device. Therefore, the existing driver software does not require adjustments. The same applies to the accelerator hardware, or in the case of this thesis, the software emulation of the device. The device can be assumed to be attached via a standard PCI interconnect. This goal ensures that heterogeneous hardware can easily be incorporated into the system. The unified interface must work for all types of software accelerator stacks.

### Minimal Software Extensions

The design should make minimal extensions to the software. This primarily aims to keep the Trusted Computing Base (TCB) as small as possible, thus reducing the possible attack vectors. Additionally, this goal helps to decrease the dependency on existing software, which is subject to change and should not be closely interwoven with the implementation of this system.

## 3.4 Threat model

As one of the system's primary goals is to provide secure data transfer, many possible threats and attack vectors must be taken into account. However, not every single one of them is considered when designing the system's security measures.

**Network Communication Attacks** Multiple attack vectors are exposed by the network communication. All data that is exchanged over the network can be monitored and captured by the entities that have access to the local network. On top of that, modifications to the network messages without detection by the network protocols are considered (man-in-the-middle-attack [5]). Further, spoofing attacks [5], where an adversary tries to initiate a communication and hide its real identity, are assumed to be possible.

**Attacks on non-TEE Software** Many of the system's communication services are handled by user-space applications that do not reside inside the CVM. Thus, they are exposed to the OS and other users on the host and present an obvious attack vector. For example, modifications to the Shared Memory region are possible without requiring any special techniques. Additionally, attacks trying to gain access to the computing state through physical means are considered.

**Out of Scope Attacks** Attacks on the device emulation software side are not addressed by the design, due to this feature being a result of testing the system and not being part of a real deployment. Furthermore, all attacks on the CVM are beyond the scope of this work, as the hardware-assisted TEE is developed by the vendors for exactly

this reason. Following the same argument, all attacks on the cryptographic primitives are not considered. This includes the used cryptography library implementations. Protection against attacks on the availability of the offered service (i.e., DOS attacks) is outside this thesis's boundaries.

# 4 Design

This chapter details the components' design and functionality. It builds upon the introduced concepts of chapter 3 and, additionally, explains the interaction between the components by specifying a custom protocol. Further, it elaborates on some challenges that were faced while designing the system.

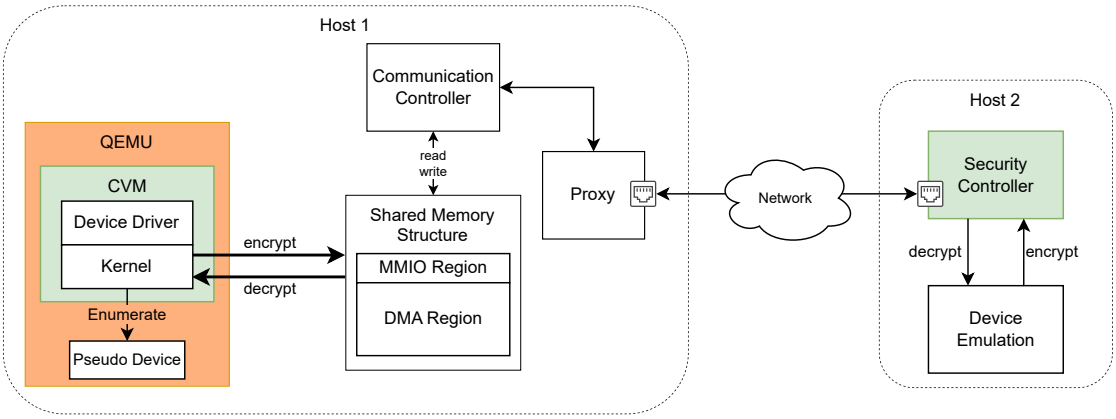



Figure 4.1: Detailed System Architecture

## 4.1 Components

Figure 4.1 depicts the elements making up the system. This is similar to the overview 3.1 in Chapter 3, but with a more fine-grained view by decomposing the components into smaller, more confined sub-components.

The most notable differences involve the addition of a pseudo-device, which is used by the kernel during device enumeration. It also separates the Shared Memory into a Communication Controller (message passing), a Proxy (networking), and the memory region itself.

The next few sections discuss the details of the single logical components as shown in the figure and highlight their contributions to the overall system.

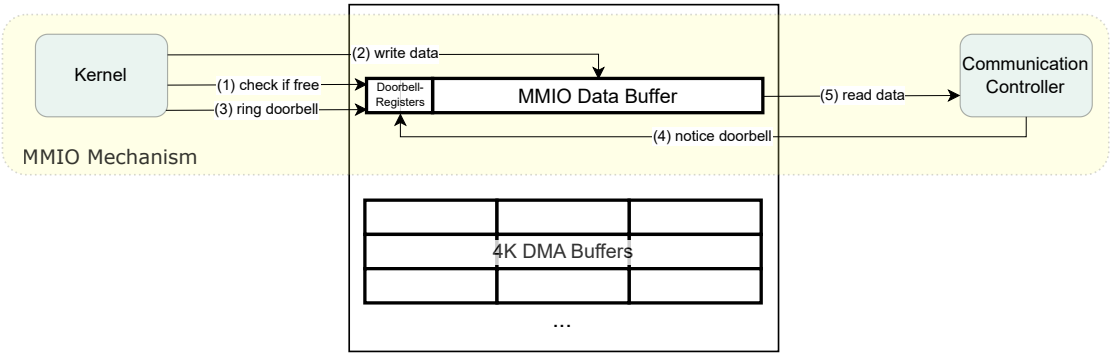



Figure 4.2: Shared Memory Structure with MMIO Message Exchange Mechanism

### 4.1.1 Pseudo Device

The Pseudo Device serves as a substitute for the disaggregated accelerator in the CVM's local device tree. As one of the design goals aims to reduce changes to the software, we try to minimize the modifications to already implemented essential services. This includes the device enumeration and resource management services provided by the UEFI and kernel. Through this approach, we maintain a unified device model that only permits changes to the kernel's device communication layer. This not only facilitates the implementation but also significantly reduces the risk of faulty code.

To register a device that is not directly attached to the CPU and, thus, can not be included in the system's device domain through a standardized protocol (e.g., PCIe), we propose a pseudo device. This device is provided by the object model feature of the CVM's hypervisor, QEMU. It is a shallow skeleton of the real device, imitating its identity and requesting the same resources (i.e., specifying the same BARs). This pseudo device only has a purpose during the early stages of the system, which include the assignment of memory resources to the devices by the UEFI and the selection of suitable drivers by the kernel. During runtime, after the right driver has been loaded, all MMIO and DMA communication attempts are handled independently of this device.

### 4.1.2 Shared Memory

The Shared Memory bridges the gap between the CVM and its host by creating a corridor that exposes a shared memory area to both domains. It is the only direct communication channel from the isolated VM to the host and is crucial in providing the disaggregation services.

See figure 4.2 for the structural design of the Shared Memory. The whole region is logically separated into a small MMIO message area and a big DMA buffer area.

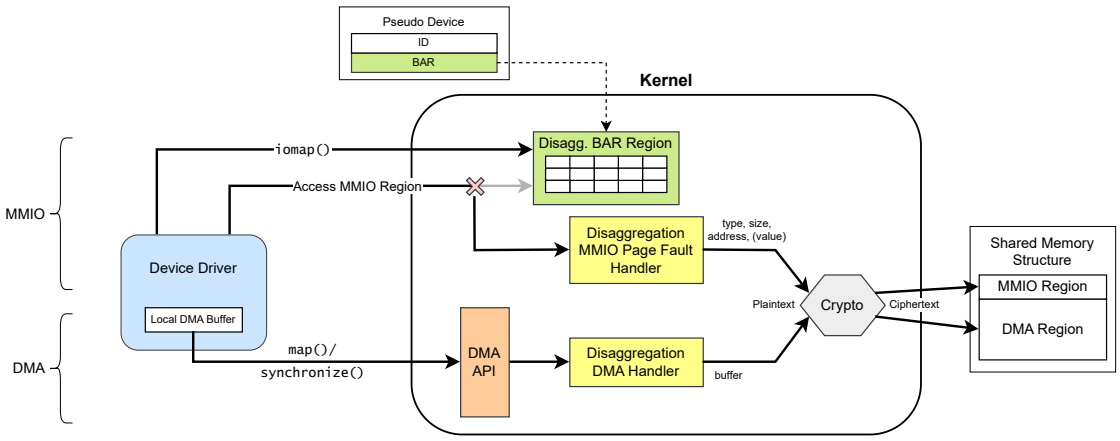



Figure 4.3: Overview of the Guest Kernel with unidirectional MMIO & DMA

The MMIO messages are exchanged through a doorbell mechanism (see 5.1.2) which synchronizes the accesses, prevents concurrent writes, and informs the receiver of an update. The mechanism can be used in both directions and is exclusively utilized for passing MMIO messages.

The DMA area is some orders of magnitude larger; it is split up into chunks of 4 KiB, which can be allocated separately or as a contiguous region. For every DMA mapping in the kernel, there is a separate non-overlapping region in this DMA area.

### 4.1.3 Guest Kernel Extensions

The following paragraph addresses the design of the MMIO mechanism in the guest Linux kernel. The initial plan and implementation was developed for the insecure version (i.e., operating only on plaintext), while the trustworthy version was built on top of that. The subsequent paragraphs clarify the kernel's DMA handling for the disaggregated case.

Figure 4.1 presents the MMIO and DMA workflow inside the kernel. It only depicts the start of each workflow, therefore omitting the responses coming from the device side.

#### MMIO

The driver can map a BAR region of a PCI device into its address space by calling an `iomap()`-like function. This will establish a virtual-to-physical address space mapping. If a driver tries to map the BAR of the Pseudo Device, the normal mapping process will

still happen; however, the pages that correspond to this mapping will be marked as *non-present* in the page table. Thus, all following accesses to this memory-mapped region will result in an exception. A page fault handler takes this failed access and extracts the relevant information: address, size, and type (Read or Write). Furthermore, if the access is of type Write, the value to be written is added to this information. Following this, the assembled information is converted into an MMIO protocol message, encrypted, and provided to the MMIO region inside the Shared Memory by using the message exchange mechanism. If the type of access was a Read, the kernel awaits the response to the request via the same mechanism and returns it to the interrupted driver as a result of its access. In case of a Write access, the driver's execution is resumed immediately after having passed the protocol message into the Shared Memory.

#### DMA

When the driver of the disaggregated device calls one of the supported API functions, it is noticed by the kernel, and the request is delegated to the Disaggregation DMA handlers.

The DMA functions work on a user-provided local buffer (see 2.1.2), which must be made available to the device. To achieve this, the handlers allocate a corresponding region for the buffer in the Shared Memory, which considers the extra space that is needed for the cryptographic transformation of the data. This allocation only happens for the initial mapping call. The subsequent synchronizations use the previous allocation to serve the request.

When the local buffer is mapped or the synchronization is done for the device, the buffer is encrypted into the allocated area of the Shared Memory. The driver is now able to program the device with a DMA transfer instruction, using an address that was provided as a result of the mapping call. This address is a reference to the Shared Memory allocation from the perspective of the Communication Controller (see 5.2).

### 4.1.4 Communication Controller

The Communication Controller is an application that interfaces with the Shared Memory. It participates in the MMIO message exchange mechanism and serves the request for DMA data by the device.

When acquiring MMIO messages through the Shared Memory's message exchange mechanism (i.e., the driver has accessed the MMIO region and the kernel forwards this encrypted information), the Controller delivers this data to the Proxy process. In the case of an MMIO Read access, the Proxy will provide a response from the device. The Controller will pass this via the same exchange mechanism to the kernel.

The Controller has a special role in the DMA communication. The device can ask for data that has been previously written to the Shared Memory's DMA region (i.e., after an initial map or synchronization to the device) by sending a request over the network. The Controller checks that the requested buffer is in bounds and issues the Proxy to send it back over the network. This order of behavior will be explained in more detail in section 4.2.

### 4.1.5 Proxy

The Proxy handles the networking tasks on the CVM side. It establishes the connection and sends or receives messages. When being handed data by the Communication Controller, it sends it to the other participant of the network communication. Upon receiving data through the connection, it provides it to the Controller. The implementation of the Proxy differs for the various transportation types, as they have different semantics, not exposing a unified interface.

### 4.1.6 Security Controller

The Security Controller is the network participant on the device host. Additionally, it interfaces with the emulation process to provide it with the MMIO request information. It establishes the connection, receives MMIO protocol messages, and sends responses for Read requests back over the network. To decouple the device from the protocol, the Controller decrypts and converts the messages to a device-specific format, providing it directly to the device via a simple interface. If the device issues a DMA request, it is intercepted by the Controller and converted into a DMA protocol message, which is addressed to the Communication Controller.

### 4.1.7 Device Emulation

The second entity on the host emulates the device. It is based on the implementation of QEMU's EDU device [29]. This is a simple **edu**cational PCI device which is capable of MMIO and DMA. To control the device, it exposes a BAR that points to its internal registers. Some of them are DMA control registers, thus implementing a DMA controller. The device also provides a small buffer, which can be used as the source or destination of DMA transfers. Those features combined are used to test the system's DMA functionality.

The device does not know about its connection to the Security Controller, but receives its MMIO access requests via an interface. The type of the request dictates the further proceedings: either changing one of its registers (i.e., type Write) or returning one of its

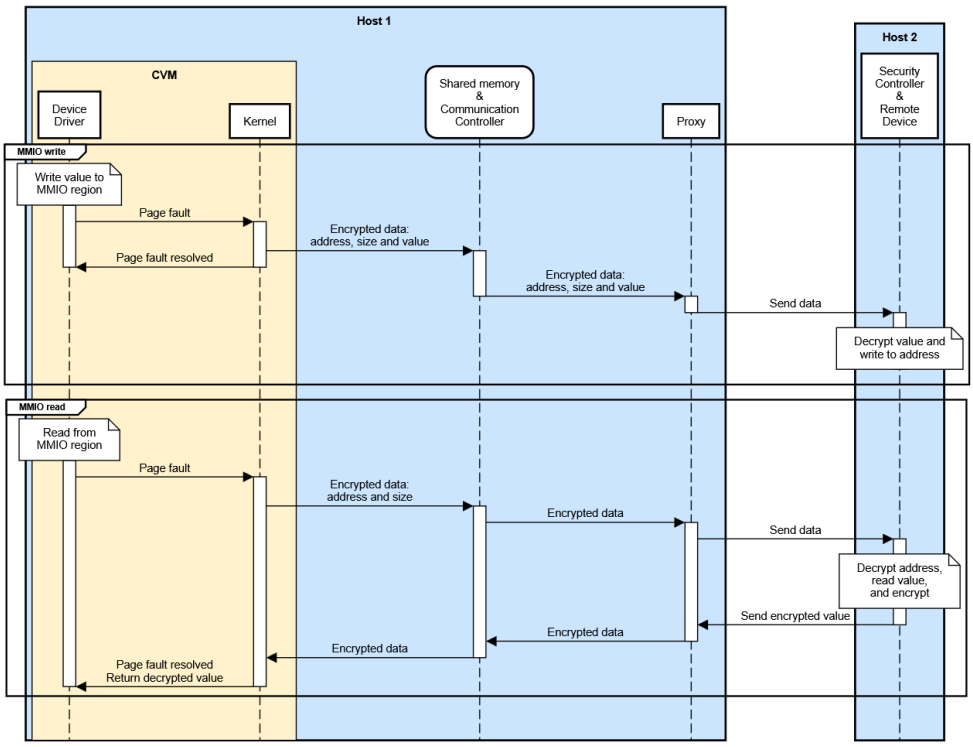



Figure 4.4: Sequence Diagram for MMIO protocol workflow. The Communication Controller is combined with the Shared Memory for reasons of clarity. Likewise, the Security Controller is combined with the Device.

current values (i.e., type Read). One specific write access to a DMA control register can initiate a transfer; subsequently, the emulation refers to an API that is exposed by the Security Controller.

## 4.2 MMIO & DMA Protocol

The previous sections had a closer look at the single components and their inner workings. This section merges them and provides a workflow overview in the form of a protocol specification.

### MMIO

Sequence diagram 4.4 shows the interactions of the components for MMIO requests. For a write operation, the value to be written is provided in the same message as the

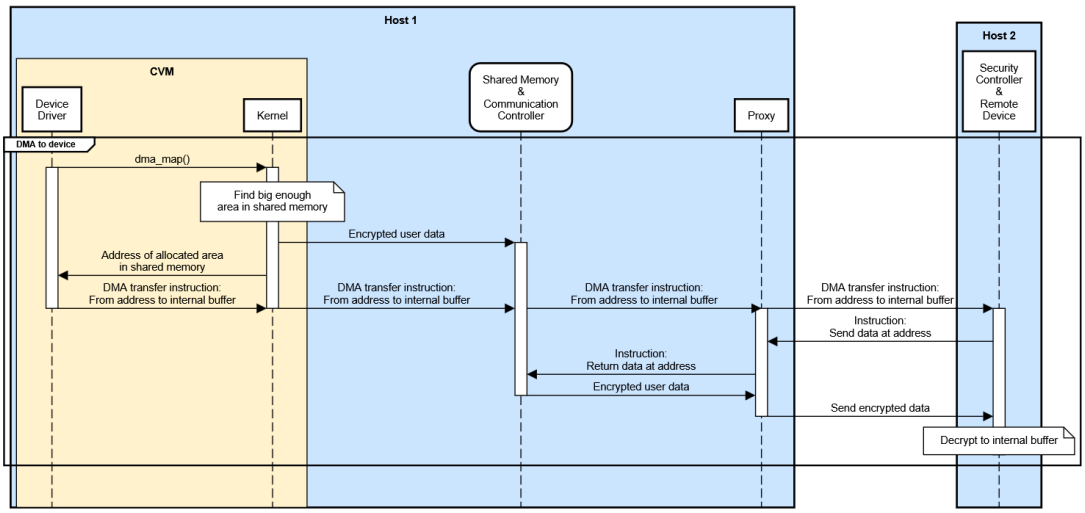



Figure 4.5: Sequence Diagram for DMA mapping call. The Communication Controller is combined with the Shared Memory for reasons of clarity. Likewise, the Security Controller is combined with the Device.

address and the size. The Read operation omits this value, but requires a chain of responses to retrieve the requested register value.

Those messages that are leaving the CVM will be encrypted before being placed into the Shared Memory and decrypted upon arrival on the device host. Responses from the device will be encrypted by the Security Controller before being sent over the network and decrypted from the Shared Memory into the CVM.

### DMA

The second sequence diagram 4.5 presents the workflow of a DMA request with the direction to the device; in particular, for the initial mapping call.

The driver provides the buffer that it wants to be mapped to the kernel. The buffer is encrypted to a contiguous region in the shared memory, which is big enough to contain the whole encrypted data. The `dma_map()`'s return value is an address that refers to this shared memory region, but it is only valid from the Communication Controller's perspective. The DMA transfer is initiated by programming the DMA controller. This is done by issuing multiple MMIO write requests (combined to one message in the diagram) to the device's DMA control registers. Upon starting the transfer, the Security Controller sends a request to the other host, asking for a specific region of its Shared Memory. This region corresponds to the data that the Kernel has previously encrypted

and written to the Memory. If the demanded address is within bounds of the Shared Memory, the Communication Controller will send the requested data to the other host. The final step involves the decryption of the buffer by the Security Controller.

Changing the direction of the transfer requires the Security Controller to send the device's local data over the network, ultimately ending up in the Shared Memory. The kernel is then able to read it back into the CVM and provide the synchronized content to the driver.

## 4.3 Securing the Communication

Until this section, the chapter primarily mentioned encryption and decryption as the cryptographic operations. This was only for simplicity and neglects the goals of integrity and freshness. To also address those security goals, the implementation relies on the concept of Authenticated Encryption with Associated Data (AEAD), which combines confidentiality and integrity in one cipher scheme by producing an authentication tag as additional output for every encryption operation. Furthermore, to meet the requirement of freshness, an incremental integer counter is used as input to every encryption operation. Thus, a decryption operation that does not use the same counter value as the corresponding encryption will fail. Consequently, the counter prevents replay attacks. The synchronization of the counter is ensured by incrementing after every encryption or decryption operation. The implementation uses the Galois/Counter Mode (GCM) mode for the symmetric block cipher AES with a key size of 256 bits.

### Unordered Cryptographic Operations

The encryption order of MMIO and DMA must not necessarily be preserved during decryption (e.g., a MMIO decryption can happen before a DMA decryption, even though the DMA data was encrypted before MMIO). This poses a problem as the goal of freshness relies on an incremental counter that is used during the authentication phase. Changing the chronological order during the decryption would lead to failure of the freshness check. As a result of this, there is a need for a separate DMA and MMIO counter.

Unfortunately, the above solution to the problem introduces another security risk. Namely, the possibility for a repeating plain-/ciphertext pair. Using separate counters can result in an encryption of the same plaintext while being on the same counter value, however unlikely it might be. This scenario produces the same ciphertext output when using the same key for both encryption. Therefore, the key must differ for MMIO and DMA.

# 5 Implementation

This chapter gives a detailed description of the implementation. While the previous chapters focused on the high-level architectural design, the following sections outline the tools and specific technical decisions of the implementation phase.

## 5.1 Low-Level Implementation Components

### 5.1.1 Pseudo Device

The Pseudo Device is realized within QEMU's Object Model (QOM). The QOM framework allows users to create custom types and devices that can be used in a QEMU instance [31]. We extend the `TYPE_PCI_DEVICE` and create a small model of a device which specifies a static Device/Vendor ID and one BAR. The size of the BAR can be provided as an argument when adding the Pseudo Device via a command line option to the QEMU runtime. This option exposes a normal PCI device to the system's PCI bus tree, thus acting as the real accelerator during initialization.

### 5.1.2 Shared Memory

The Shared Memory is another feature of QOM, which ships with the normal QEMU software. It is, again, an implementation of a PCI device and exposes one of its BARs as the shared memory region [30]. The region is backed with a file on the host, therefore exposing the memory to both the guest kernel and the Communication Controller. In both cases, the region can be mapped (with the kernel's `iomap()` for the BAR and the normal `mmap()` system call for the file) into the respective virtual address spaces. The size of the Shared Memory is configurable by a command-line option when starting the CVM.

Table 5.1 outlines the different fields and their locations in the Shared Memory. The number of DMA buffers depends on the configured size. The Mapping Address is the virtual address of the Communication Controller's `mmapped()` Shared Memory region.

| Offset (+size) | Description |
|---|---|
| 0x0 (+1) | Host-Kernel Doorbell |
| 0x1 (+1) | Kernel-Host Doorbell |
| 0x8 (+246) | MMIO Messages |
| 0x100 (+8) | Shared Memory Virtual Mapping Address |
| 0x1000 (+variable) | DMA Buffers (4KiB each) |

Table 5.1: All fields and their offsets in the Shared Memory

#### Doorbell MMIO Mechanism

The kernel and Communication Controller exchange MMIO messages by utilizing a doorbell mechanism. There are two doorbells, one for each direction. If the kernel wants to write to the MMIO region, it waits until the *Kernel-Host Doorbell* is cleared and restores the bit after having written the message to the area. The host can now read the message and clear the bit afterwards. The same procedure applies to the *Host-Kernel* doorbell just for the reverse direction. As the MMIO message exchange follows a specified protocol, there is no possibility of a deadlock situation; both entities are synchronized and always operate only on one doorbell at the same time.

### 5.1.3 MMIO Message Structure

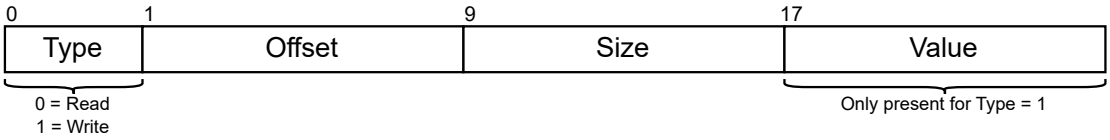



Figure 5.1: Structure of the MMIO request

Figure 5.1 specifies the structure of an MMIO protocol message. Those kinds of requests are delivered from the kernel to the Security Controller upon an MMIO access by the driver.

### 5.1.4 Linux Guest Kernel

#### MMIO

When the BAR of the Pseudo Device is `pci_iomap()`-ed by the driver, the kernel will manually walk the page tables and mark all pages of this BAR as non-present. Additionally, the marked pages and their corresponding physical addresses are saved

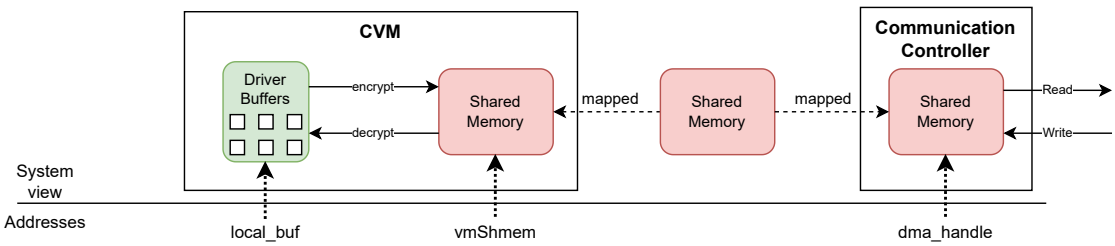



Figure 5.2: Mapped Shared Memory with different Address Spaces

in an in-kernel structure. All this only happens after the BAR has already been mapped into the virtual address space.

Subsequently, access to the mapped region will result in a page fault. To know if a page fault was caused by the Pseudo Device BAR, the page fault handler checks the entries of the in-kernel mapping structure and compares them with the faulty address. If a corresponding mapping is found, a MMIO request (5.1.3) is prepared with the required information, it is encrypted, and provided to the Communication Controller through the message exchange mechanism (5.1.2). If the access was of type Read, the kernel waits until the doorbell is rung and the response value is returned to the driver, thereby resolving the page fault.

### DMA

Figure 5.2 contextualizes the names that are used in this explanation. During initialization, the Communication Controller writes the address of the mapped Shared Memory (`dma_handle`) to offset 0x100 inside the Shared Memory. The kernel reads this address to use it in future DMA mapping calls.

The disaggregation-specific code is directly interfacing with the DMA API functions. When calling one of the supported functions, the kernel checks if the callee driver is from a disaggregation context and continues as follows:

- **`dma_map_single`**`(void *local_buf, size_t size)` Searches for a free region in the shared memory DMA area of at least size + 16 (for the authentication tag). If a suitable `vmShmem` address was found, it saves the allocation information in a tree structure for future allocation and synchronization requests. The information includes the `local_buf` address, the `dma_handle` address, and the size. It encrypts the region from `local_buf` to `vmShmem` and returns `dma_handle` to the driver. This returned address will then be provided to the device when programming its DMA control registers.

- `dma_sync_single_for_cpu(dma_addr_t dma_handle, size_t size)` Searches for the entry in the tree structure to retrieve the `vmShmem` and `local_buf` addresses which correspond to `dma_handle`. Decrypts from `vmShmem` to `local_buf`.
- `dma_sync_single_for_device(dma_addr_t dma_handle, size_t size)` Same search as above, only encrypts from `local_buf` to `vmShmem`.

## 5.2 Cryptography

The used cipher is AES-256-GCM. We use a 12-byte initialization vector (IV), which influences the output, and a 256-bit key as meta input for all operations. To prevent the same plaintext-ciphertext pair, we include a counter in the operations, which increments after every encryption and decryption operation. The counter is integrated into the IV and therefore ensures that it provides a different result even for the same plaintext input. An integer counter is also a proposed method of how the IV should be handled, to ensure non-repeating IVs for invocations of encryption functions [6]. There is a separate counter for MMIO and DMA (see 4.3). The authentication tag, which allows for checking the integrity of the message, is combined with the ciphertext and included for every MMIO message and every DMA buffer that is written to the shared memory.

The CVM implementation takes advantage of the in-kernel Crypto API, while the user-space Security Controller uses the standard OpenSSL crypto library. Both provide an implementation of AES-256-GCM, which utilizes hardware-supported cryptographic extensions if supported by the CPU.

## 5.3 Network Implementations

We provide different types of transport over the network. There is a standard TCP/IP, RDMA, and a raw Ethernet implementation. The network is exclusively handled by user-space (Proxy and Security Controller) applications; thus, no changes to the CVM kernel are necessary. The details that are worth mentioning are explained in the following sections.

### Ethernet Packet Processing

The Ethernet implementation works on raw frames with the help of Packet MMAP [20] to reduce kernel interactions. To start the applications, one has to provide the MAC addresses of the network interface cards (NICs). Those addresses will then be directly written into the Ethernet header and, combined with the payload, handed to the socket of type `AF_PACKET`.

The Packet MMAP implementation allows for reducing buffer copying between the user and kernel space. To do that, it creates a memory region that is shared between both domains. This allows the removal of one buffer copy, but still requires the networking stack of the kernel to handle the frame and thus does not remove all overhead. At least one copy is between the NIC's DMA buffer and the kernel's `sk_buff`. [3]

### RDMA

The RDMA implementation registers the mapped Shared Memory (`dma_handle`) of the Communication Controller to allow direct user-space reads and writes by the Security Controller for the DMA operations. If the device calls one of the Controller's DMA API functions, the access to the remote Shared Memory can happen without involvement from the Proxy. MMIO is based on the send/recv implementation of RDMA and thus is similar to a normal TCP/IP communication, and as such, needs the interaction of both involved entities.

### TCP

The TCP implementation sets the `TCP_NODELAY` option for the sockets. This option prevents the network stack from buffering small packets and waiting for a sufficient amount of data before sending. Especially useful for our case, where the MMIO messages are rather small-sized.

# 6 Evaluation

This chapter evaluates the design and implementation. It aims to answer the following questions:

1. What is the end-to-end overhead for device accesses in our system?
2. What is the overhead of small device register accesses via MMIO?
3. What is the overhead of bulk data transfers via DMA?
4. What is the overhead of secure (encrypted and authenticated) communication in our system?
5. How do different network transportation types perform compared to a baseline (same host) setup?

The first section defines the used setup. Second, the evaluation starts with covering the overall system performance by measuring the throughput of DMA requests and the latency of MMIO Read accesses. The subsequent evaluations are based on microbenchmarks to investigate the reasons for the performance degradation.

## 6.1 Setup

The experiments were performed on an AMD EPYC 7413 (24 cores, 1.5GHz) and an AMD EPYC 7713P (64 cores, 1.5GHz). The former, which provides AMD SEV-SNP hardware support, is used as the host of the VM. The latter emulates the simplified device. Both machines support hardware cryptography AES-NI technology, which allows faster encryption and decryption for AES operations. For the evaluation, the servers were directly connected through their Intel E810-C 100Gbit/s network interface cards. Those cards support version two of RDMA over Converged Ethernet (RoCE v2).

The evaluation utilizes all three different network transportation types (Ethernet, TCP/IP, and RDMA; see 5.3) and the same host implementation, which is the baseline for the results. This same host setup runs all components, the device emulation and the CVM, on the same EPYC 7413.

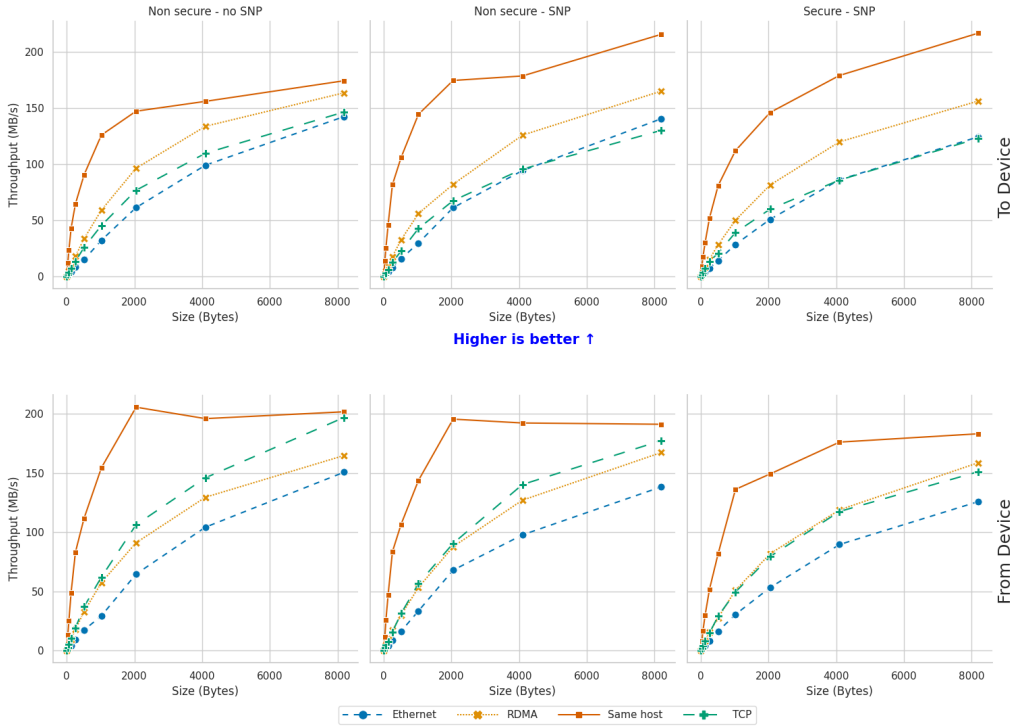



Figure 6.1: DMA throughput

### Configuration

The host OS of both servers is NIXOS 25.05 (Warbler). The EPYC 7413 runs a 6.9.0-rc7 kernel, while the EPYC 7713 runs a 6.8.0-rc5 kernel. The VM is provided by QEMU v10.0.5 and has a 6.11.0-rc1 kernel with Debian 12 (bookworm) as the OS.

The guest VM is configured to use the cryptographic hardware support provided by its host. It is assigned 8GB of memory and runs 8 vCPUs. The OpenSSL version on both servers is 3.4.1.

## 6.2 DMA Throughput

**Methodology** This section measures the throughput of DMA transfers across different transportation types, security levels, directions, and sizes. The data was collected within the device driver, using either `dma_sync_single_for_device()` (upper row) or `dma_sync_single_for_cpu()` (bottom row) for specifying the direction. The whole measured time included the transfer instruction to the DMA controller via MMIO and

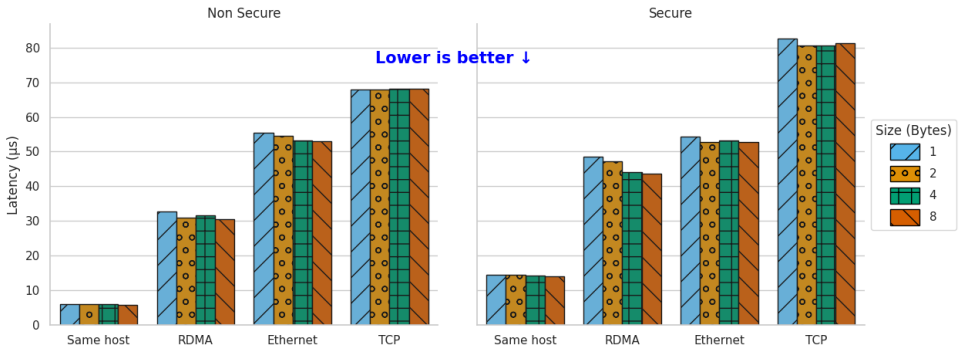



Figure 6.2: MMIO read latency

the respective call to a `sync()` function. The single-threaded application issued multiple subsequent requests of the same size, always waiting for the completion of the previous request before starting the next one. The transferred sizes varied from a few bytes to 8 Kibibytes.

**Results** Figure 6.1 shows the results of the average throughput evaluation in megabytes per second. For example, "Non secure - no SNP" refers to no protection of the CPU-device communication (MMIO/DMA), and the VM is not being hosted with AMD SEV-SNP hardware support. The results show no notable difference between the security levels. Likely, the direction of the transfer has little impact on the throughput. An exception to this is the TCP connection, which performs better on "From Device" than on "To Device" transfers. This could be explained with TCP's higher latency which is crucial for "To Device" transfers where an additional request for the data has to be issued over the network (see 4.2). The same host setup has the highest throughput across all dimensions, reaching and sometimes surpassing 200 MB/s for 8 KiB transfers. The other three transportation types are performing pretty similarly, with TCP and RDMA achieving a slightly higher throughput than Ethernet. Those three transport types show a rise of throughput for growing sizes. The same host setup struggles to keep this over the whole time and stagnates for growing sizes. For sizes starting from 512B, the same host setup achieves throughput of 2x better than RDMA, 2.3x better than TCP, and 3.3x better than Ethernet when combining the security levels and taking the average ratio over all measured sizes.

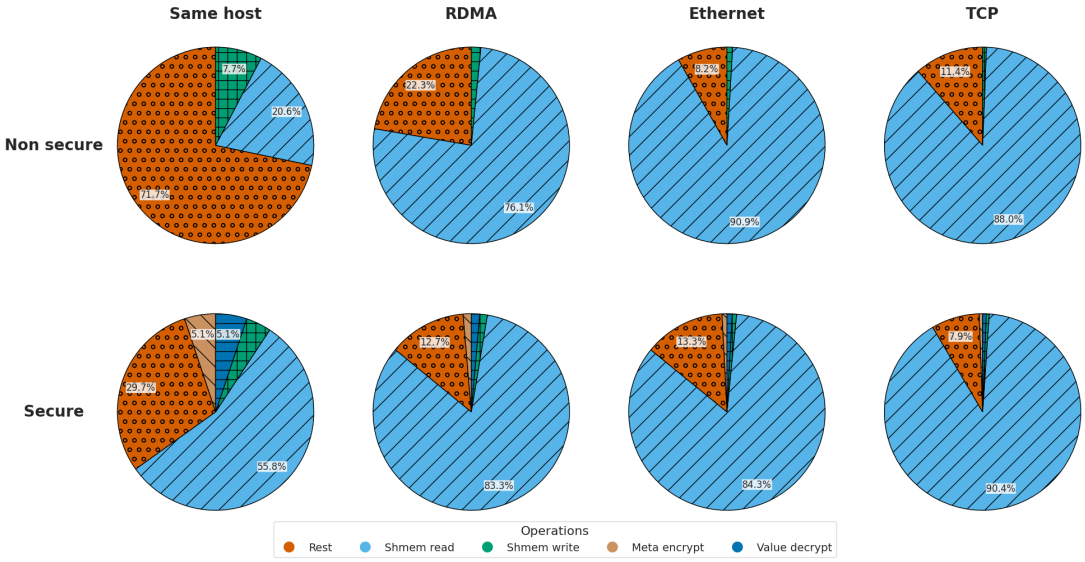



Figure 6.3: MMIO read latency breakdown into different stages of the operation.

## 6.3 MMIO Latency

This section presents the results for latency measurements of MMIO read requests across different transportation types, security levels, and access sizes. Further, the latency overhead is decomposed into the different stages of the computation. MMIO accesses of type Write are not suited for evaluations, as there is no response to such a request, and execution is resumed immediately after sending the necessary information to the device; without a response from the device, the latency of a Write can not be measured properly. Though it can be assumed to be about half the time of the MMIO read latencies.

**Methodology** The latency is measured from access to the MMIO region until return of the value. The non-secure evaluation is based on a setup with an SEV-SNP CVM, but without protection of the communication between the device and the VM (i.e., MMIO and DMA messages are exchanged as plaintext).

**Results** Figure 6.2 shows the latency in $\mu$s for different transportation and access sizes. Standard MMIO region accesses vary from one Byte to eight Bytes. Smaller accesses actually show a slightly increased latency. This could be explained by the processor and crypto implementations being optimized for 32 or 64-bit values. Non-surprisingly, the Same host setup shows the lowest latency across all sizes and security levels. There is a slight latency difference between the security levels, with the secure version being 1.5x longer (+8.8 $\mu$s) than the non-secure version on average. The reason for this can

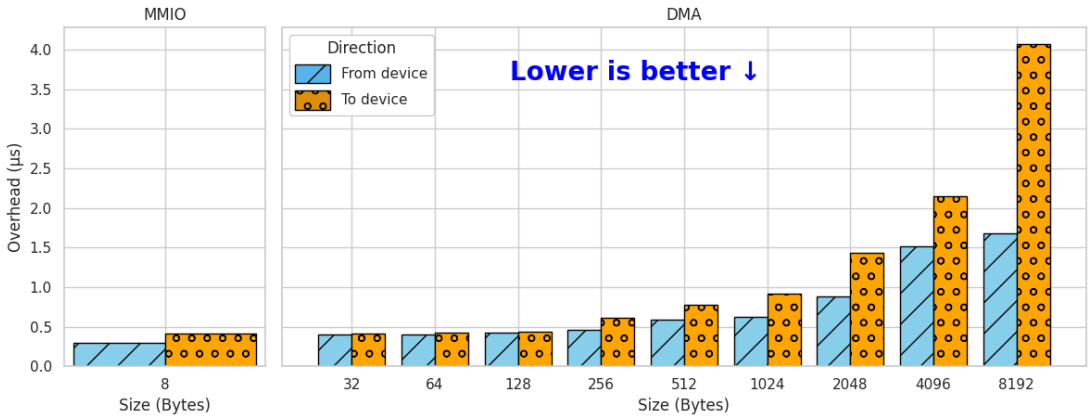



Figure 6.4: Bounce buffer overhead.

also be seen in the charts of figure 6.3, which decomposes the latency into smaller fragments and expresses where the time is spent. The encryption and decryption parts show minimal involvement in the process, except for the low-latency Same Host setup, where every small overhead has a bigger relative impact on the result. The wait for a response, which is depicted by the *Shmem Read*, takes the biggest part across network setups. This part involves the wait for a response by the device; thus, the network communication is the main reason for the latency. The *Rest* refers to a set of various tasks, including handling of the page fault or retrieving metadata to map the faulty address.

## 6.4 Bounce-Buffer Copy Overhead

The overhead caused by the additional copy into or from the shared memory is presented here.

**Methodology** The results for this section are exclusively based on the *Ethernet* setup. This is due to the similarity of how the data is prepared to be sent or forwarded to the device. The data was collected on the Proxy side of the setup and takes into account the way of how to notify the kernel of MMIO message arrival. The additional Shared Memory copy is the same across all transportation types except for RDMA. As RDMA can access the remote memory directly, it requires no additional copy into the transport's message queue. RDMA still uses a send/recv interface for MMIO messages; thus, the MMIO data can also be applied in this case.

**Results** The figure 6.4 shows the overhead of copying data into or from the Shared

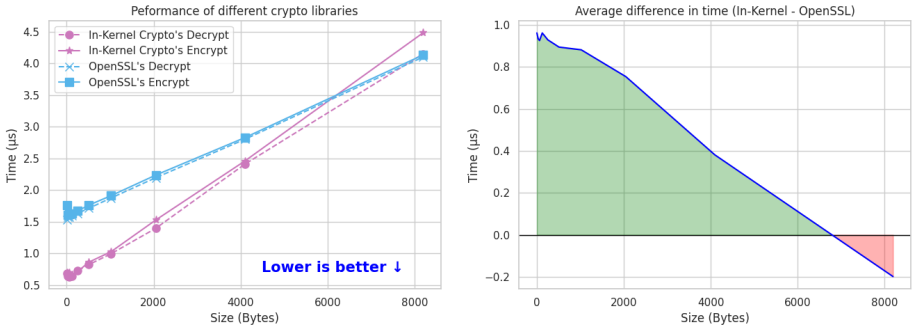



Figure 6.5: Cryptography performance for different APIs

Memory to be well below 5 $\mu$s across all sizes. The additional time rises proportionally to the size of the operation. The difference between *From device* and *To device* can be explained with the additional overhead when waiting for messages over the Shared Memory exchange mechanism. The overall overhead of this additional copy is negligible when compared to the network latency.

## 6.5 Cryptography Library Performances

**Methodology** The implementations of the AES-GCM encryption and decryption operations in the Kernel and OpenSSL were compared for different sizes. The input of the encrypt operation was random data, while the decryption's input was a valid output from a previous encryption. The measured time does not only include the mere encryption and decryption, but also the process of authentication. For the OpenSSL implementation, it also includes the retrieval of the cipher context.
**Results** The spent time rises linearly with the size of the operation. There is only a small difference between the respective encryption and decryption results of the implementations. Comparing the libraries shows the kernel API starting stronger, with being almost 1 $\mu$s faster for small sizes, but scaling worse for bigger sizes. For 8KB the OpenSSL implementations has taken the advantage and is faster.

## 6.6 Network Overhead

The main overhead in this implementation is caused by the transfer of data over a network. The precise data and real overhead are discussed in this section. The setup used for this evaluation is based on the network implementations of this thesis's project, and thus, the results of the following sections can be taken into consideration when

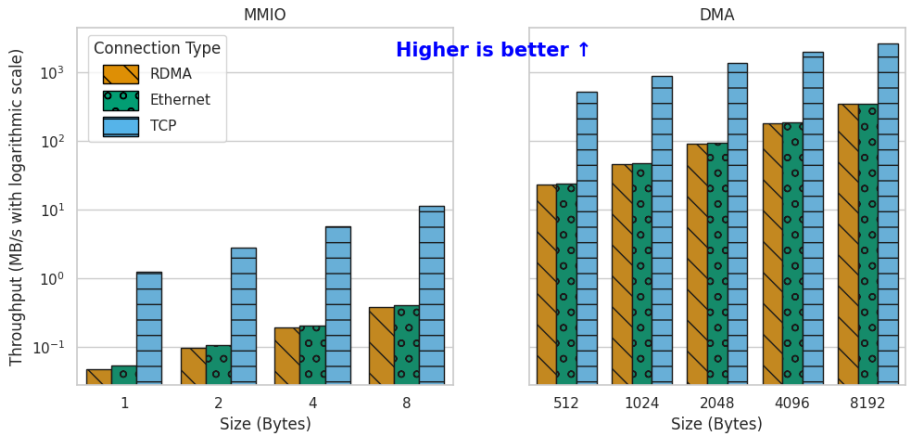



Figure 6.6: Network throughput for different transportation types

discussing the reasons for performance degradation. The direct-connected NICs from 6.1 were also used in this test. This also implies that there was no other traffic on this link while running the benchmarks.

### 6.6.1 Throughput

**Methodology** The throughput is measured by sending packets of a fixed size as fast as possible over the network. The Ethernet implementation does not receive any kind of arrival confirmation and assumes the data to be sent when providing it to the kernel via `send()`. RDMA uses different types of send calls for MMIO and DMA, with `rdma_post_send()` and `rdma_post_write()`, respectively. For both methods, it waits for confirmation of success by calling `rdma_get_send_comp()`. Only then does it resume sending the next data. The TCP implementation is reliable by default, and the data is assumed to arrive upon leaving the application space with a `send()`.

**Results** The results in figure 6.6 for the MMIO send operations (left column) show them to be not as efficient as DMA sizes (right column) in regard to their throughput. This is an effect caused by the network layers when preparing the data to be sent. There is additional overhead to prepare multiple smaller sizes than when sending just one big buffer of data and handling the network stack operations just once. The TCP connection achieves a throughput at least 6x times higher than RDMA and Ethernet across all sizes. Both of whom are about the same and differ only minimally. With growing size, the throughput also rises for all transportation types. The high throughput of the TCP implementation is contrary to the expected results. A kernel-bypass network protocol, like RDMA, is assumed to achieve a much higher throughput than TCP, which has to

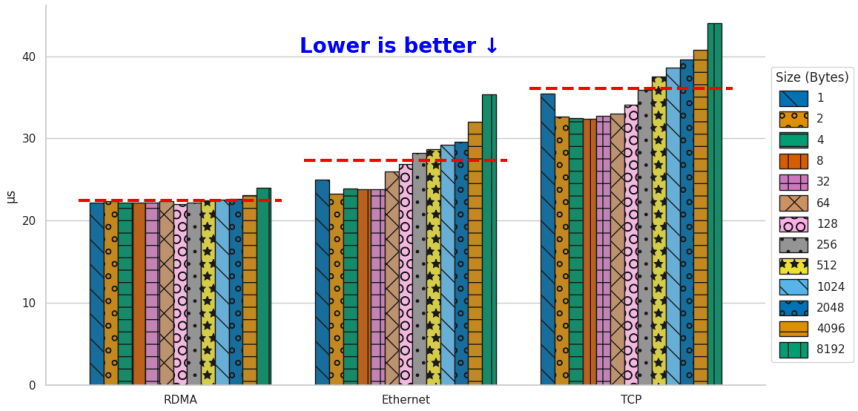



Figure 6.7: Network latency for different transportation types

traverse the whole kernel network stack before reaching the network interface card. The investigation of this behavior is left for future work.

### 6.6.2 Latency

This section analyses the latency of the network transportation types.
**Methodology** The latency was obtained by sending a message of specific size to the peer and waiting for a response message of the same size. The time from send until reception is then divided by two and represents the latency (one-way latency).
**Results** Figure 6.7 shows the results of the tests. RDMA's latency is remaining very constant for increasing sizes, averaging a latency of 22.5 $\mu$s. The other two connection types show different behavior, as their latency increases for growing sizes, with Ethernet averaging 27 $\mu$s and TCP 36 $\mu$s.

## 6.7 Discussion

This section will try to connect the single evaluation results and attempt to draw a coherent image of the system's performance.

The main bottleneck for more efficient communication was shown to be the latency of the transportation protocols, which limits not only the performance of MMIO requests, but also has a big influence on DMA operations, as they require sending multiple protocol messages over the network. This can be observed at one specific example: The maximum reached network throughput of TCP showed it to be about an order of magnitude higher than the other two network types, but this does not reflect in the

DMA throughput evaluation, which does not reach that order of difference. That shows the effect of increased latency in the case of TCP; the communication is mainly hindered by the time it takes to travel over the network, with DMA being one of the victims, as it involves multiple MMIO requests. The increased latency must not necessarily mean that the protocol itself can not support lower latency, but is rather the result of the implementation, which was not optimized for performance, but tried to lay the groundwork for secure communication.

The overhead, caused by cryptographic primitives and the copying of data to the Shared Memory, does not appear to be a significant portion of the latency. This can be seen at the decomposition of MMIO Read requests, where both of those categories combined do not come close to a 5% share on the Secure setup, and also on the results of the dedicated Bounce Buffer and Cryptography throughput tests. Before optimizing the implementation in those fields, the network latency should have top priority.

# 7 Related Work

There exists plenty of research on disaggregation, but most of it focuses on storage [14] and memory [12, 15, 22, 28, 21, 45]. The minority of work also explores the disaggregation of accelerators [16]. This work mainly focuses on the goals of improving the performance [39] and mitigating the consequences of networks, with their low throughput and high latency. The following analysis of the related work is arranged in two categories. The support for trustworthy accelerators which are directly attached to the CPU and on network-attached accelerators which do not provide security guarantees.

## 7.1 CPU-Attached Trustworthy Accelerators

The Trusted Execution Environment Device Interface Security Protocol (TDISP) is an architecture used to establish a trusted relationship between a CVM and a PCIe device; subsequently allowing to protect data exchange on the Transaction Layer [26, p. 1566][41]. This is not applicable to disaggregation where the resource pools are connected via optic.

Schneider [35] proposes a TEE based enclave model which can include heterogeneous accelerators into a isolated enclave. It requires changes to the hardware and uses shared memory as one solution on how to communicate between the components composing a enclave. It highlights the significance of MMIO and DMA and proposes a way on how to include their memory regions into an enclave. Some hardware devices in the system may be part of multiple enclaves, thus allowing to share an accelerator between various tenants. The design allows to dynamically change the assigned hardware devices. The work does not directly support VMs, but operates on normal user-space applications. This approach is not applicable as it heavily relies on direct interconnected components, therefore not taking the implications of network disaggregation into consideration.

An approach aiming at rack-scale confidential computing is HETEE [46]. The proposed design requires no changes to the existing hardware and implements a dedicated HETEE box which manages the device allocation to a TEE and isolation from other devices. This again confines the computing area to a rack, thus not allowing to take advantage of datacenter-wide network disaggregation.

## 7.2 Network-attached Accelerators

The paper [16] works on the performance of the network connection to reduce latency between CPU and accelerator pools, thereby achieving comparable performance to CPU-attached accelerator of normal servers. The main focus lies on building a low-latency network stack for an accelerator pool by incorporating a suitable RDMA communication protocol with hardware support. Further, no specific design of the CPU software side is provided, but the CPU only issues requests for very specific workloads. This neglects the benefits offered by the abstraction through a VM and of course, provides no security guarantees for the disaggregation.

The Beehive project [17] builds on the same idea of a hardware network stack for disaggregated accelerators. The main proposal is to provide a modularized approach over a network-on-chip which allows to incorporate different network protocols of arbitrary complexity for increasing flexibility and options for scaling. Again, this work focuses on enabling networking for accelerators and not on the software stack on the CPU side of the computation.

# 8 Summary and Conclusion

This thesis designs and implements the software stack of a disaggregated network-attached accelerator setup while guaranteeing security for the communication over the system's whole life-cycle. The design specifies its own protocol which allows to exchange MMIO accesses and DMA transfers between a CVM instance and a device emulation process over two hosts.

The proposed scheme allows the driver of the disaggregated device to uphold the assumption of a local-attached device and thus not requiring any modifications to the existing accelerator software. To achieve that, the source code of the CVM's Linux kernel is modified to intercept a disaggregated driver's communication attempts, converting them to messages of a specific format and forwarding them in an authenticated encrypted way to the host. Subsequently, a user space application is responsible for transmitting the data over an established network channel to the host of the device. A security controller receives the messages and reconverts them; by interfacing with the device, it can directly provide the requested communication. Responses or DMA requests are handled over the reverse way.

The evaluation of the implementation, which offers three different types of networking transport (TCP/IP, Ethernet, and RDMA) gave an overview of the system's performance under different workload and security levels. The latency of MMIO read requests for RDMA (the best performing network transport for this metric) is 3.2x higher when compared to a setup where the device is emulated on the same host as the CVM. The DMA throughput showed the ratio to be 2x for the same compared setups.

While the system performs worse for disaggregated accelerators, the main prospect of this work was achieved by designing and implementing the software stack for secure communication with a network-attached device.

The source code can be found at: `https://github.com/harshanavkis/jigsaw-overall/pull/2`.

# 9 Future Work

## Support for Coherent DMA

The design currently supports the streaming DMA-API of the Linux kernel. Therefore, the driver needs to synchronize the mapped memory region with the device every time after having changed its local copy. But this also goes the other way; when the device changes the DMA buffer, the driver has to manually request the kernel to update its local version of the buffer. This requires to implement some parts of the DMA functionality within the driver, thus making the programmability of driver software more difficult. For more convenient access, the kernel offers the coherent DMA API 2.1.2, which omits the need for manual synchronization by ensuring a write access is directly visible for the other side. Future work could expand on this subject to provide coherent disaggregated DMA. This would require to send updates for every single change to the buffer immediately over the network as the DMA API has no previous knowledge of the write access' size.

## Fault-tolerant Ethernet Connection

One implementation for the transportation type utilizes the Ethernet communication protocol, which does not provide fault tolerance or transmission security. It is therefore a main prospect of this system to provide those safety guarantees and extend the Ethernet implementation or commit to a different network protocol which does support secure and reliable connections (e.g. TCP/IP or RDMA).

## Interrupt Support

A important way to synchronize with a device or inform the CPU of some event is through interrupts. For example, a standard DMA controller issues an interrupt to signal the completion of a transfer. Therefore, the support for interrupts over a disaggregated network could be a future project, expanding the set of supported communication types on top of MMIO and DMA.

## Dedicated MMIO Access Detection

The utilization of page faults should not necessarily be abused for handling accesses to a MMIO region. Page faults cause the switch from driver code into the kernel fault handling space (saving registers and switching the code context), thus wasting time on acknowledging and handling the exception until the disaggregated page-fault handler is executed. In the future, a method to reduce the overhead of switching between execution contexts could allow to reduce the latency of MMIO accesses.

## Mitigating Network Overhead

Other frameworks for user-space networking, like DPDK [43] for TCP and Ethernet, can help to reduce the overhead through the networking communication, thus getting closer to the latency of CPU-attached connections. This also includes tools like Netmap [34] which removes even more copy operations from the networking stack than PACKET MMAP [20] by giving user-space applications direct access to a network adapter. While the theoretical throughput of our implementation is not restricted by the network connection, the main bottleneck is the latency. This effect could be mitigated by the introduced frameworks.

# Abbreviations

**MMIO** Memory-Mapped Input/Output

**DMA** Direct Memory Access

**PCI** Peripheral Component Interconnect

**CVM** Confidential Virtual Machine

**TEE** Trusted Execution Environment

**AEAD** Authenticated Encryption with Associated Data

**GCM** Galois/Counter Mode

**BAR** Base Address Register

**MMU** Memory Management Unit

**ECAM** Enhanced Configuration Access Mechanism

# List of Figures


2.1 Simplified structure of PCI configuration space [26, p. 965]. Only the relevant fields for this thesis are shown in the figure. . . . . 4
2.2 Overview of virtual and physical address space in the MMIO case. Normally, the I/O section in the physical address space would probably start at address $0x0$ [40, p. 342]. This figure should emphasize, that there is no overlapping of I/O and RAM, and those regions do not have to be mapped contiguous. . . . 5
2.3 Simplified DMA operational workflow. Inspired by [40, p. 345] . . . . . 7

3.1 High-level Overview of the System . . . . . 11

4.1 Detailed System Architecture . . . . . 16
4.2 Shared Memory Structure with MMIO Message Exchange Mechanism . . . . 17
4.3 Overview of the Guest Kernel with unidirectional MMIO & DMA . . . . . 18
4.4 Sequence Diagram for MMIO protocol workflow. The Communication Controller is combined with the Shared Memory for reasons of clarity. Likewise, the Security Controller is combined with the Device. . . . . 21
4.5 Sequence Diagram for DMA mapping call. The Communication Controller is combined with the Shared Memory for reasons of clarity. Likewise, the Security Controller is combined with the Device. . . . . 22

5.1 Structure of the MMIO request . . . . . 25
5.2 Mapped Shared Memory with different Address Spaces . . . . . 26

6.1 DMA throughput . . . . . 30
6.2 MMIO read latency . . . . . 31
6.3 MMIO read latency breakdown into different stages of the operation. . . . . 32
6.4 Bounce buffer overhead. . . . . 33
6.5 Cryptography performance for different APIs . . . . . 34
6.6 Network throughput for different transportation types . . . . . 35
6.7 Network latency for different transportation types . . . . . 36

# List of Tables

5.1 All fields and their offsets in the Shared Memory . . . . . 25

# Bibliography


[1] K. Abouelmehdi, A. Beni-Hessane, and H. Khaloufi. "Big healthcare data: preserving security and privacy." In: *Journal of Big Data* 5.1 (2018), p. 1. ISSN: 2196-1115. DOI: 10.1186/s40537-017-0110-7.

[2] AMD. *AMD Secure Encrypted Virtualization (SEV)*. (Visited on 31.07.2025). URL: https://www.amd.com/en/developer/sev.html.

[3] T. Barbette, C. Soldani, and L. Mathy. "Fast userspace packet processing." In: *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 2015, pp. 5–16. DOI: 10.1109/ANCS.2015.7110116.

[4] A. Boutros and V. Betz. "FPGA Architecture: Principles and Progression." In: *IEEE Circuits and Systems Magazine* 21.2 (2021), pp. 4–29. DOI: 10.1109/MCAS.2021.3071607.

[5] R. Daş, A. Karabade, and G. Tuna. "Common network attack types and defense mechanisms." In: *2015 23nd Signal Processing and Communications Applications Conference (SIU)*. 2015, pp. 2658–2661. DOI: 10.1109/SIU.2015.7130435.

[6] M. Dworkin. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. en. Nov. 2007.

[7] T. von Eicken, A. Basu, V. Buch, and W. Vogels. "U-Net: a user-level network interface for parallel and distributed computing." In: *SIGOPS Oper. Syst. Rev.* 29.5 (Dec. 1995), pp. 40–53. ISSN: 0163-5980. DOI: 10.1145/224057.224061.

[8] R. Huerta, M. A. Shoushtary, J.-L. Cruz, and A. González. *Analyzing Modern NVIDIA GPU cores*. 2025. arXiv: 2503.20481 [cs.AR].

[9] Intel. *Intel Trust Domain Extensions (Intel TDX)*. (Visited on 31.07.2025). URL: https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html.

[10] N. P. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. Patterson. *TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings*. 2023. arXiv: 2304.01433 [cs.AR].

[11] M. A. Kafi and N. Akter. "Securing Financial Information in the Digital Realm: Case Studies in Cybersecurity for Accounting Data Protection." In: *American Journal of Trade and Policy* 10.1 (Apr. 2023), pp. 37–48. DOI: 10.18034/ajtp.v10i1.659.

[12] P. Koutsovasilis, M. Gazzetti, and C. Pinto. "A Holistic System Software Integration of Disaggregated Memory for Next-Generation Cloud Infrastructures." In: *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2021, pp. 576–585. DOI: 10.1109/CCGrid51090.2021.00067.

[13] C. S. Kruse, R. Goswamy, Y. Raval, and S. Marawi. "Challenges and Opportunities of Big Data in Health Care: A Systematic Review." In: *JMIR Med Inform* 4.4 (Nov. 2016), e38. ISSN: 2291-9694. DOI: 10.2196/medinform.5359.

[14] S. Legtchenko, H. Williams, K. Razavi, A. Donnelly, R. Black, A. Douglas, N. Cheriere, D. Fryer, K. Mast, A. D. Brown, A. Klimovic, A. Slowey, and A. Rowstron. "Understanding Rack-Scale Disaggregated Storage." In: *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. Santa Clara, CA: USENIX Association, July 2017.

[15] H. Li, K. Liu, T. Liang, Z. Li, T. Lu, H. Yuan, Y. Xia, Y. Bao, M. Chen, and Y. Shan. "HoPP: Hardware-Software Co-Designed Page Prefetching for Disaggregated Memory." In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2023, pp. 1168–1181. DOI: 10.1109/HPCA56546.2023.10070986.

[16] Y. Liao, J. Wu, W. Lu, X. Li, and G. Yan. "DPU-Direct: Unleashing Remote Accelerators via Enhanced RDMA for Disaggregated Datacenters." In: *IEEE Transactions on Computers* 73.8 (2024), pp. 2081–2095. DOI: 10.1109/TC.2024.3404089.

[17] K. Lim, M. Giordano, T. Stavrinos, I. Zhang, J. Nelson, B. Kasikci, and T. Anderson. "Beehive: A Flexible Network Stack for Direct-Attached Accelerators." In: *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Nov. 2024, pp. 393–408. DOI: 10.1109/micro61859.2024.00037.

[18] Linux Kernel contributors. *Dynamic DMA mapping Guide*. (Visited on 07.08.2025). URL: https://www.kernel.org/doc/html/latest/core-api/dma-api-howto.html.

[19] Linux Kernel contributors. *Linux Kernel*. Version 6.11-rc1. [Operating system]. 2024.

[20] Linux Kernel contributors. *Packet MMAP*. (Visited on 29.07.2025). URL: https://www.kernel.org/doc/html/latest/networking/packet_mmap.html.

[21] D. Masouros, C. Pinto, M. Gazzetti, S. Xydis, and D. Soudris. "Adrias: Interference-Aware Memory Orchestration for Disaggregated Cloud Infrastructures." In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2023, pp. 855–869. DOI: 10.1109/HPCA56546.2023.10070939.

[22] V. Mishra, J. L. Benjamin, and G. Zervas. "MONet: heterogeneous Memory over Optical Network for large-scale data center resource disaggregation." In: *Journal of Optical Communications and Networking* 13.5 (2021), pp. 126–139. DOI: 10.1364/JOCN.419145.

[23] M. Misono, D. Stavrakakis, N. Santos, and P. Bhatotia. "Confidential VMs Explained: An Empirical Analysis of AMD SEV-SNP and Intel TDX." In: *Proc. ACM Meas. Anal. Comput. Syst.* 8.3 (Dec. 2024). DOI: 10.1145/3700418.

[24] L. Null and J. Lobur. *The Essentials of Computer Organization and Architecture*. Second. Jones and Bartlett Publishers, 2006. ISBN: 9780763737696.

[25] A. Pagès, R. Serrano, J. Perelló, and S. Spadaro. "On the benefits of resource disaggregation for virtual data centre provisioning in optical data centres." In: *Computer Communications* 107 (2017), pp. 60–74. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2017.03.009.

[26] *PCI Express® Base Specification Revision 6.1*. en. Publisher: PCI-SIG. July 2023.

[27] C. Pinto, D. Syrivelis, M. Gazzetti, P. Koutsovasilis, A. Reale, K. Katrinis, and H. P. Hofstee. "ThymesisFlow: A Software-Defined, HW/SW co-Designed Interconnect Stack for Rack-Scale Memory Disaggregation." In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2020, pp. 868–880. DOI: 10.1109/MICRO50266.2020.00075.

[28] A. Puri, K. Bellamkonda, K. Narreddy, J. Jose, and T. Venkatesh. "A Practical Approach For Workload-Aware Data Movement in Disaggregated Memory Systems." In: *2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2023, pp. 78–88. DOI: 10.1109/SBAC-PAD59825.2023.00017.

[29] QEMU. *EDU device*. (Visited on 14.08.2025). URL: https://www.qemu.org/docs/master/specs/edu.html.

[30] QEMU. *Inter-VM Shared Memory device*. (Visited on 10.08.2025). URL: https://www.qemu.org/docs/master/system/devices/ivshmem.html.

[31] QEMU. *The QEMU Object Model (QOM)*. (Visited on 28.07.2025). URL: https://www.qemu.org/docs/master/devel/qom.html.

[32] A. Reale and D. Syrivelis. "Experiences and challenges in building next-gen optically disaggregated datacenters : (Invited Paper)." In: *2018 Photonics in Switching and Computing (PSC)*. 2018, pp. 1–3. DOI: 10.1109/PS.2018.8751236.

[33] R. J. Recio, P. R. Culley, D. Garcia, B. Metzler, and J. Hilland. *A Remote Direct Memory Access Protocol Specification*. RFC 5040. Oct. 2007. DOI: 10.17487/RFC5040.

[34] L. Rizzo. "Netmap: a novel framework for fast packet I/O." In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC'12. Boston, MA: USENIX Association, 2012, p. 9.

[35] M. Schneider, A. Dhar, I. Puddu, K. Kostiainen, and S. Čapkun. "Composite Enclaves: Towards Disaggregated Trusted Execution." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (Nov. 2021), pp. 630–656. ISSN: 2569-2925. DOI: 10.46586/tches.v2022.i1.630-656.

[36] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts, Global Edition*. Tenth. Wiley, 2019. ISBN: 9781119454083.

[37] W. Stallings. *Computer Organization and Architecture, Global Edition*. Harlow, UNITED KINGDOM: Pearson Education, Limited, 2015. ISBN: 9781292096865.

[38] W. Stallings. *Operating Systems: Internals and Design Principles, global Edition*. en. Pearson Higher Education, 2018. ISBN: 9781292214290.

[39] V. Suresh, B. Mishra, Y. Jing, Z. Zhu, N. Jin, C. Block, P. Mantovani, D. Giri, J. Zuckerman, L. P. Carloni, and S. V. Adve. "Mozart: Taming Taxes and Composing Accelerators with Shared-Memory." In: *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*. PACT '24. Long Beach, CA, USA: Association for Computing Machinery, 2024, pp. 183–200. ISBN: 9798400706318. DOI: 10.1145/3656019.3676896.

[40] A. Tanenbaum and H. Bos. *Modern Operating Systems, Global Edition*. Harlow, United Kingdom, UNITED KINGDOM: Pearson Higher Education & Professional Group, 2014. ISBN: 9781292061955.

[41] L. Ternullo. *IDE and TDISP: An Overview of PCIe® Technology Security Features*. (Visited on 10.08.2025). Feb. 2025. URL: https://pcisig.com/blog/ide-and-tdisp-overview-pcie%C2%AE-technology-security-features.

[42] The Confidential Computing Consortium. *A Technical Analysis of Confidential Computing*. https://confidentialcomputing.io/wp-content/uploads/sites/10/2023/03/CCC-A-Technical-Analysis-of-Confidential-Computing-v1.3_unlocked.pdf. Nov. 2022.

[43] The Linux Foundation. *DPDK*. (Visited on 11.08.2025). URL: https://www.dpdk.org/.

[44] *UEFI Platform Initialization Specification Release 1.9*. en. Publisher: UEFI Forum, Inc. Dec. 2024.

[45] C. Wang, K. He, R. Fan, X. Wang, Y. Kong, W. Wang, and Q. Hao. *CXL over Ethernet: A Novel FPGA-based Memory Disaggregation Design in Data Centers*. 2023. arXiv: 2302.08055 [cs.AR].

[46] J. Zhu, R. Hou, X. Wang, W. Wang, J. Cao, B. Zhao, Z. Wang, Y. Zhang, J. Ying, L. Zhang, and D. Meng. "Enabling Rack-scale Confidential Computing using Heterogeneous Trusted Execution Environment." In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1450–1465. DOI: 10.1109/SP40000.2020.00054.

[47] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. "Congestion Control for Large-Scale RDMA Deployments." In: *SIGCOMM Comput. Commun. Rev.* 45.4 (Aug. 2015), pp. 523–536. ISSN: 0146-4833. DOI: 10.1145/2829988.2787484.