UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Analyzing the security implications of the CXL.mem protocol

Analyse der Sicherheitsimplikationen des CXL.mem-Protokolls

Bachelorarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Tom Boje

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

mit Unterstützung von
Thore Tiemann M.Sc.

Lübeck, den 10. Dezember 2025

Abstract

Compute Express Link (CXL) is an emerging standard designed to enable high-performance, coherent interconnects between processors, memory, and accelerators. While CXL.mem, the protocol layer responsible for memory expansion, promises performance scalability and transparent memory pooling, its increasing complexity introduces potential new security risks that have not yet been comprehensively studied.
This thesis investigates the security implications of CXL.mem, with a particular focus on how address mapping and translation mechanisms could be exploited or misconfigured to compromise data integrity and isolation. A custom CXL memory device was implemented and analyzed to explore potential aliasing behavior, a condition in which distinct logical addresses may unintentionally or maliciously refer to the same physical memory region. Through targeted design modifications and experimental validation, this work demonstrates that such aliasing effects can be intentionally induced within the CXL.mem architecture, revealing a class of vulnerabilities that may arise from insufficient verification or faulty device logic. The results underscore the importance of trustworthy hardware design and verification methods for emerging interconnect technologies like CXL.

Zusammenfassung

Compute Express Link (CXL) ist ein aufkommender Standard, der darauf ausgelegt ist, leistungsstarke und kohärente Interconnects zwischen Prozessoren, Speicher und Beschleunigern zu ermöglichen. Während CXL.mem, die Protokollschicht verantwortlich für Speichererweiterung, Leistungsskalierbarkeit und transparentes Memory Pooling verspricht, bringt seine zunehmende Komplexität potenzielle neue Sicherheitsrisiken mit sich, die bislang noch nicht umfassend untersucht wurden.
Diese Arbeit untersucht die Sicherheitsimplikationen von CXL.mem, mit besonderem Fokus darauf, wie Adressabbildungs- und Übersetzungsmechanismen ausgenutzt oder fehlerhaft konfiguriert werden könnten, um die Datenintegrität und Isolation zu gefährden. Ein maßgeschneidertes CXL-Speichergerät wurde implementiert und analysiert, um potenzielles Aliasing-Verhalten zu erforschen, eine Situation, in der unterschiedliche logische Adressen unbeabsichtigt oder böswillig auf denselben physischen Speicherbereich verweisen können. Durch gezielte Designänderungen und experimentelle Validierung zeigt diese Arbeit, dass solche Aliasing-Effekte innerhalb der CXL.mem-Architektur absichtlich hervorgerufen werden können. Damit wird eine Klasse von Schwachstellen offengelegt, die aus unzureichender Verifikation oder fehlerhafter Gerätelogik entstehen kann. Die Ergebnisse unterstreichen die Bedeutung vertrauenswürdiger Hardware-Designs und Verifikationsmethoden für aufkommende Interconnect-Technologien wie CXL.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

_____

Lübeck, 10. Dezember 2025

Contents

1 Introduction

Modern computing systems are undergoing a profound architectural shift. For decades, servers were built around a simple principle: each CPU was tightly coupled with its own directly attached memory. This model worked well as long as workloads were predictable and systems scaled mostly vertically. Today, however, data-center environments face entirely different demands. Cloud providers must dynamically allocate resources to millions of users, large-scale AI applications require enormous memory footprints, and heterogeneous accelerators must be integrated efficiently. These trends create a strong demand for flexible, composable, and highly scalable memory infrastructures. Compute Express Link (CXL) has emerged as one of the key technologies enabling this transition. Built on top of PCI Express (PCIe), CXL provides high-bandwidth, low-latency communication between CPUs and external devices while maintaining cache coherence and consistent memory semantics across the platform [HS24]. Several major vendors have already announced or released CXL-based memory expanders, and upcoming server generations increasingly rely on CXL to overcome the limitations of traditional memory hierarchies [Ele24, Tec25]. In contrast to personal devices such as laptops or smartphones, this development is primarily driven by hyperscale data centers, where resource utilization and operational efficiency directly translate into economic and environmental impact. A particularly promising part of the CXL standard is CXL.mem, which allows CPUs to access device-attached memory almost as if it were local Dynamic Random Access Memory (DRAM). This capability enables new concepts such as memory pooling, disaggregated memory, and composable server architectures [SBB24]. Instead of each processor being limited to the DRAM installed on its motherboard, memory can be dynamically added, removed, or shared across multiple hosts. Such flexibility is crucial for emerging AI workloads. While functional aspects and performance behavior of CXL.mem are being actively explored in industry and academia, its security implications remain largely unexamined [SMM23a, Dat]. Moving memory outside the CPU socket fundamentally changes the trust boundaries of the system. CXL devices introduce their own address translation logic, device-internal memory controllers, configuration registers and control paths [Com20]. This components did not previously exist in conventional DRAM subsystems. This added complexity increases the likelihood of malicious modifications that may jeopardize data integrity or violate isolation guarantees. A critical aspect in this context is that CXL.mem exposes device memory directly to the host CPU, making it appear as part of the system's physical address space

[Com20]. This differs from traditional Direct Memory Access (DMA) performed by I/O devices: in CXL.mem, the CPU itself issues load and store operations into device memory, relying on the correctness of the device's internal address decoding. Any flaw in this decoding or in the logic translating host physical addresses into device memory locations could potentially cause subtle and hard-to-detect errors. The aim of this thesis is to investigate such security-relevant corner cases within the CXL.mem subsystem. In particular, it examines whether design or configuration flaws could lead to address aliasing, where two distinct logical addresses unintentionally reference the same physical memory region. Using an FPGA-based CXL Type-3 Memory Device, this work implements and analyzes a proof-of-concept showing how a single manipulated address bit can create overlapping memory regions. The results demonstrate that such aliasing is not visible to the host and can silently compromise data isolation highlighting the need for deeper security analysis of emerging CXL-based memory architectures. The main contributions of this thesis are:

1. Development and validation of a working FPGA-based CXL Type-3 Memory Device.
2. Introduction of a controlled address-aliasing mechanism within the CXL memory path to analyze potential vulnerabilities.
3. Experimental evaluation of functional behavior and security impact of these modifications.
4. Discussion of resulting implications for secure hardware design in future CXL deployments.

The rest of this document is structured as follows: Chapter 2 provides background on the CXL architecture and related research on memory-system security. Chapter 3 describes the experimental platform and system architecture used in this work. Chapter 4 explains the implementation and bring-up process of the CXL device. Chapter 5 presents the security experiment on address aliasing. Chapter 6 discusses the broader security implications, and Chapter 7 concludes the thesis with a summary and outlook.

2 Background and Related Work

CXL builds upon established principles of memory and I/O interconnects, yet introduces architectural elements that differ fundamentally from conventional DRAM subsystems. To situate this work within the existing research landscape, this chapter first compares traditional PCIe- and DRAM-based hierarchies, followed by the core concepts of the CXL specification and its memory model. It then reviews prior work on memory isolation failures and aliasing-related attacks. This background serves as a foundation for understanding why CXL.mem introduces novel security and verification challenges, and how this thesis aligns with or extends current research.

2.1 Comparison to PCIe and Conventional Memory Hierarchies

First, it is important to understand how conventional memory architectures are structured in order to highlight the differences of CXL.

Conventional DRAM-Based Memory Hierarchies In conventional server architectures, the memory hierarchy is tightly coupled to the processor socket. Each CPU integrates one or more memory controllers (MC) that is directly connected to DRAM channels and together they form a local Non-Uniform Memory Access (NUMA) domain. Accesses to local DRAM are served with low latency, while accesses to memory attached to another socket are routed over inter-socket links such as Intel Ultra Path Interconnect (UPI) or AMD Infinity Fabric and therefore incur higher latency [CKA21]. The global physical address space is partitioned across these memory controllers and sockets, but the overall topology is relatively static: Memory capacity scales only with the number of sockets and DRAM slots, and the separation between memory and I/O is clearly defined at the architectural level.

PCIe as a Non-Coherent I/O Interconnect PCI Express (PCIe) was originally designed as a high-speed and point-to-point I/O interconnect for peripheral devices such as network cards, storage controllers, and accelerators. Its communication model follows a non-coherent request completion semantics that clearly separates CPU memory access from device-side operations. When the CPU interacts with a PCIe device, it typically performs memory-mapped I/O (MMIO), meaning that the processor reads from or writes

to device-exposed registers or control/status regions located in Base Address Registers (BARs) [PCI22]. Conversely, when a PCIe device needs to access host memory, it does so via Direct Memory Access (DMA), where the device autonomously issues read or write transactions into the host's RAM without CPU intervention [PCI22]. Both MMIO and DMA rely on PCIe's non-coherent memory model: data transferred through these mechanisms is not kept coherent with the CPU's caches. As a result, software must coordinate buffer sharing manually, for instance by flushing caches, invalidating stale lines or programming DMA engines to move data explicitly between host memory and device memory regions.

CXL as a Coherent Memory Fabric Compute Express Link (CXL) extends this model by reusing the PCIe physical and data link layers while introducing a new and coherent protocol stack on top [Com20]. Following enumeration via the CXL.io layer, the host and device can negotiate additional CXL.cache and CXL.mem capabilities that enable cache-coherent load or store access to memory across the link. In contrast to classic PCIe where a device is treated as an I/O endpoint using DMA into host memory, a CXL Type-3 memory device contributes its own memory capacity directly into the host physical address space via Host-Managed Device Memory (HDM) decoders [Com20]. From the operating system's point of view, such a device appears as an additional NUMA node rather than a traditional PCIe peripheral, even though it is still attached over the same physical interface.

Impact on Topology, Pooling, and Composability This shift from a socket-centric memory model to a fabric-based model has significant architectural implications. CXL devices can reside outside the CPU package while still offering memory semantics. Therefore memory capacity and bandwidth no longer need to scale strictly with the number of processor sockets. Instead, systems can attach memory expansion devices or even build memory pools that are dynamically assigned to different hosts [HS24]. CXL's HDM decoders allow physical address ranges to be mapped to different devices or interleaved across multiple endpoints. This enables detailed control over placement and access characteristics [Com20]. These capabilities blur the traditional boundary between local DRAM and remote I/O device.

Latency and Access Semantics Performance-wise, CXL.mem sits between the slower PCIe-based DMA transfers and the faster access speeds of local DRAM, providing a balanced middle ground for memory operations. Local DRAM accesses typically achieve load-to-use latencies on the order of tens of nanoseconds, whereas PCIe DMA transactions

often involve microsecond-scale overhead due to queuing, descriptor setup and interrupt handling [NAZ+18]. Initial studies suggest that while CXL.mem is slower than local DRAM by a factor of 2-3× [LHX+24], it still outperforms conventional bulk DMA operations by a wide margin in terms of latency [Man22]. As a result, CXL offers a modern, high-performance alternative to conventional DMA programming, with reduced complexity and improved efficiency. Furthermore CXL maintains the familiar load and store semantics.

Security-Relevant Differences at a High Level From a security perspective, the most important distinction between conventional PCIe and CXL-based memory expansion lies in the trust boundary and the location of address translation. PCIe devices are traditionally treated as untrusted I/O endpoints whose access to system memory is mediated and constrained by the I/O Memory Management Unit (IOMMU) [MRG+19]. In contrast, CXL devices that expose memory via CXL.mem participate directly in the host's memory subsystem: The visibility and placement of their memory are governed by HDM decoders and device mapping logic rather than by the IOMMU alone [Com20, Lin25b]. While a detailed discussion of concrete attack techniques is mentioned in section 2.4, this architectural difference already indicates that misconfiguration or design flaws in CXL mapping logic could have the potential to inflict greater harm than traditional PCIe device architectures.

2.2 CXL architecture overview

Now that some of the features of CXL have been outlined, it is important to understand how CXL is structured, with particular attention to CXL.mem.

Layered Protocol Structure CXL builds on the PCI Express (PCIe) physical and electrical layer. It extends it with additional protocol layers to support low-latency memory operations [Com20]. The CXL architecture is divided into three logical protocol layers: *CXL.io*, *CXL.cache*, and *CXL.mem*.
CXL.io reuses the well-known PCIe transaction model and is primarily responsible for device discovery, configuration, interrupts and management operations. It works as the control and management path and guarantees backward compatibility with conventional PCIe devices.
CXL.cache, enables devices, such as accelerators, to coherently access and cache host memory using low-latency request/response transactions. As a result, offload engines can handle shared datasets without the overhead of manual synchronization in software.

At last, *CXL.mem* provides a memory interface that allows the host processor to directly access device-attached memory as part of its physical address space [Com20]. By combining these three layers, CXL creates a unified framework for I/O, caching and memory operations, this all through a single interconnect. This helps to reduce unnecessary data movement and improves overall system efficiency.

Device types The specification mentions three main types of devices, each designed to interact with the protocol in a way that matches its role within the system architecture.:

- *Type-1* devices implement CXL.io + CXL.cache (no CXL.mem) and typically represent accelerators without local device memory, for example, storage offload engines. They coherently access host memory but do not export their own memory to the host [Com20].

- *Type-2* devices implement CXL.io + CXL.cache + CXL.mem and add a local, often high-bandwidth memory. They can both cache host memory and export their device memory for host access, while coherence modes are governed by driver policies [Com20].

- *Type-3* devices implement CXL.io + CXL.mem only (no CXL.cache). They act as memory expanders, allowing the host to treat device-resident memory as system memory [Com20].

In summary, *Type-3* devices represent the class of particular relevance to this thesis.

Host-Device Communication and HDM Decoders From a system perspective, a CXL Type-3 device is enumerated by the host firmware and operating system as a PCIe function that includes CXL-specific capability structures, referred to as Designated Vendor-Specific Extended Capabilities (DVSEC). During system initialization, the firmware or CXL driver configures the HDM decoders within the CXL device, which define base addresses, range sizes, interleave parameters, and access permissions for each exposed memory region [Com20, Lin25b]. Only when an HDM decoder is properly enabled and unlocked, the corresponding memory range becomes visible to the host and accessible through the CXL.mem protocol. Type-3 devices do not implement a coherent cache for host memory. Because of this, transactions on this path consist mostly of read and write operations directed to the device memory[Com20]. The role and functionality of HDM decoders will be discussed in more detail in section 2.3.

2.3 Addressing and Memory Mapping in CXL.mem

Efficient addressing and memory mapping are central to the operation of CXL.mem. They determine how host systems perceive and access device-resident memory, ensuring that expanded memory resources can be integrated seamlessly into the system's physical address space. This section introduces the essential mechanisms behind addressing and mapping in CXL.mem.

Addressing Model in CXL Systems With CXL.mem, the way memory is addressed follows a layered structure that goes beyond the conventional CPU-centric model. It opens up the physical address space to include memory from connected devices. Unlike conventional DRAM controllers, which manage only directly attached memory channels, a CXL Type-3 device exposes its memory to the host, which maps it into its physical address space via CXL.mem. The host processor maintains full ownership of the global physical address map, while each connected CXL device contributes one or more address ranges via its Host-Managed Device Memory (HDM) interface. This interface is realized inside the device by HDM decoders, which define how host physical address (HPA) ranges are mapped to device physical address (DPA) ranges [Com20, Lin25b]. Consequently, the CXL memory fabric enables both the host and connected devices to share a common address space. This shared space supports direct load or store operations. So it enables memory to be expanded remotely and makes it possible to pool memory across multiple endpoints [Com20].

DVSEC Capabilities and the Role of HDM Decoders The core of CXL's memory mapping mechanism are the HDM decoders, which define how host physical addresses are translated into device physical addresses. Therefore, it is important to understand how these are initialized and used.Every CXL-enabled device must provide several DVSEC structures in the extended PCIe configuration space, which the host can use to read specific CXL functions, for example, the properties of the HDM decoder. The CXL specification describes various DVSEC blocks for this purpose, including the *CXL Capability*, *CXL Control*, *CXL Lock* and *CXL Status* registers as well as the *Range* registers. These registers announce the properties of each CXL device [Com20]. This fulfils the following tasks:
The *CXL Capability* and *CXL Control* registers contain basic information about whether a device supports CXL.cache or CXL.mem. Among other things, the `Mem_Enable` bit and the `HDM_Count` are listed here. These bits determine whether the device is considered by the host as a CXL memory endpoint and how many HDM ranges it has [Com20].
The *CXL Range* registers are responsible for detecting the size and position of the memory regions provided by the device. The CXL specification provides several register pairs,

for example DVSEC *CXL Range 1* and optionally *Range 2*, if multiple regions are to be supported. These registers fulfil the task of providing information about the start address in host-physical addressing and the size of the memory region. In addition, they signal whether these values are valid and whether the memory has been fully initialised [Com20].
It is particularly important to note that as long as the HDM decoder is not activated, the device uses only the DVSEC range registers for all address decoding. This means that the DVSEC entries directly define which host address is mapped to which internal device address.
Another DVSEC register is the DVSEC *CXL Lock* register. The CXL specification defines the following behaviour: Once `CONFIG_LOCK` has been set from *CXL Lock*, the base and size registers may no longer be changed. This prevents unpredictable changes in the address space after commissioning [Com20].
The CXL Status register is largely reserved and only contains information about whether a device has entered viral mode, a special error state defined by the CXL specification [Com20]. While relevant for fault isolation, this mode is not central to the memory mapping mechanisms discussed in this thesis.
As mentioned above, the activation of the HDM decoder plays an important role. If the device also implements the HDM decoder structure, which is common for CXL Type 3 devices, the specification describes a clear transition logic:
As long as the HDM decoder `enable` bit is not set, the device must decode exclusively using the DVSEC range registers. Only when the HDM decoder `enable` bit is set does the logic switch to the extended registers of the HDM mechanism.
The HDM decoders take an extended role compared to the DVSEC range registers: They enable finer and more flexible address mapping for memory areas [Com20], especially in more complex CXL Type 3 devices, and are therefore preferable.
Ultimately, this simply means that the DVSEC range registers form a kind of fallback logic for the device. They are particularly relevant if the host initially leaves the HDM decoder disabled for compatibility reasons.
To ensure proper system integration, the Linux CXL subsystem, introduced in Linux Kernel 5.12 [Ker21] and continuously expanded, with stable NUMA integration available since Linux Kernel 6.9 [Ker24], provides kernel-level interfaces to probe, configure and manage HDM decoders. This ensures that all device-exposed memory ranges are correctly mapped into the system's NUMA topology and visible through standard memory-management interfaces [Lin25b]. During device enumeration, the firmware or driver uses the CXL mailbox mechanism, a command-based interface defined in the CXL specification, to query device capabilities, issue configuration commands, and confirm that HDM

decoders have transitioned to the `Active` state [Com20]. While DVSEC registers expose static configuration fields, the mailbox mechanism complements them by enabling dynamic queries and control operations. Together, these mechanisms ensure that memory ranges are properly initialized and addressable through the CXL.mem protocol .

Address Translation Path. When the host accesses CXL-attached memory, the address translation and data path differ from that of locally attached DRAM. A typical transaction proceeds as follows:
The host CPU issues a physical address corresponding to a CXL region. The routing decision is made by the host's memory controller and address decoder, which determine whether a given physical address belongs to local DRAM or to a CXL device. If the address falls within a CXL-assigned range, the request is passed through the CXL root complex and forwarded across the CXL link to the target device. At the time of reception, the device's CXL controller consults its HDM decode tables to determine which internal memory region corresponds to the incoming address. The translated address is then passed to the device's internal interconnect, commonly an AXI [Arm25] or Avalon-MM [Int09] bus, and subsequently processed by the memory controller connected to DDR4, DDR5 or persistent memory modules [Com20, Ter24]. This layered translation pipeline introduces new stages compared to traditional DRAM access but preserves load or store semantics. Because of its reliance on several configuration layers, from CXL configuration space down to RTL-level decode logic, CXL's memory mapping is naturally more flexible but also more complex and error-prone than conventional architectures.

2.4 Security Background: Memory and Mapping Attacks

Main memory has traditionally been treated as a largely passive component in the system security model. However, a growing number of work has demonstrated that subtle electrical effects, address-mapping artifacts, and microarchitectural interactions in the memory subsystem can be exploited to violate isolation guarantees across privilege and protection domains. In the context of this thesis, it is therefore important to understand which classes of memory-related attacks exist and how they manipulate the memory subsystem.

Disturbance and Access-Pattern Based Memory Attacks One of the most widely studied hardware-induced memory vulnerabilities is the *Rowhammer* effect. Kim et al. demonstrated that repeatedly activating a DRAM row at high frequency can induce electrical interference in adjacent rows. This leads to deterministic bit flips under certain access patterns [KDK+14]. Following work has shown that these disturbance effects

persist across multiple DRAM generations and that carefully planned access patterns can be exploited to launch practical privilege-escalation attacks on commodity systems [MK20]. Although *Rowhammer* mainly affects the physical DRAM rather than logical address translation, it illustrates that memory subsystems can exhibit behaviour far beyond what system designers usually assume. A related direction involves timing-based leakage or manipulation of memory contents through shared microarchitectural structures. For example, *RAMBleed* uses Rowhammer-induced disturbance effects in combination with timing feedback to infer bits stored in adjacent DRAM rows, effectively turning an integrity failure into a confidentiality breach [KGGY20]. Similarly, early studies on *DRAMA* attacks showed that the internal organisation of DRAM banks and rows can be derived by observing timing differences in memory accesses, revealing parts of the mapping between physical addresses and DRAM structures [PGM+16]. Those attack classes demonstrate that memory subsystems leak structural information that attackers may use to influence or predict mapping behaviour. These works are referenced here purely as background. They establish that memory hardware can be exploited through disturbance effects or timing characteristics, but they do not form part of the experimental evaluation in this thesis.

BadRAM: Configuration-Induced Aliasing Attacks A more recent line of work focuses on exploiting the interaction between DRAM addressing and hardware memory-encryption schemes. *BadRAM* targets the Serial Presence Detect (SPD) configuration of DRAM modules. By modifying the SPD contents, a malicious memory module can falsely advertise larger capacities and thereby trick the memory controller into mapping multiple physical address ranges onto overlapping regions of the module [MWO+25]. *BadRAM* has demonstrated exploits in which such configuration-based aliasing is used to leak or modify encrypted memory inside Trusted Execution Environments (TEE) and confidential virtual machines. This was done by violating implicit assumptions about the one-to-one relationship between multiple physical addresses [MWO+25]. In contrast to *Rowhammer*, *BadRAM* does not rely on electrical disturbances. It purely abuses the logical mapping and configuration path, which makes it conceptually closer to the address-aliasing scenarios considered in this thesis.

Battering RAM and Physical Aliasing on Encrypted Memory In addition to *BadRAM*, the *BatteringRAM* attack family shows that similar aliasing effects can be induced without actively manipulating the memory bus. The *BatteringRAM* attack uses a low-cost interposer placed between the CPU and a DDR4 memory module to selectively manipulate

row-address lines on the memory bus [DMOVVB26]. By omitting specific address bits at runtime, the attacker can dynamically create aliases in the physical address space. That means the attacker can force two different physical addresses to refer to the same underlying DRAM cells. Systems using deterministic memory encryption for Trusted Execution Environments (TEEs), such as Intel SGX or AMD SEV-SNP, are vulnerable to replay and integrity attacks. An attacker can write encrypted data to one memory address and later replay it via an aliased address. This causes the decrpytion of the data to outdated or manipulated plaintext under the same key [DMOVVB26]. *BatteringRAM* falls outside traditional software-based threat models because it involves physical changes to the memory channel. Still, it shows how address aliasing at the DRAM level can compromise the security of current memory encryption techniques.

Relation to This Thesis The attacks outlined above target different layers of the memory stack: *Rowhammer* focuses on cell-level disturbance effects. *BatteringRAM* and *BadRAM* exploit physical and configuration-induced address aliasing in the DRAM subsystem. In this thesis, we do *not* attempt to reproduce all of these attacks experimentally. Instead, they serve as conceptual and methodological background. First, they show that assumptions about the uniqueness of physical addresses are fragile. Second, they show that TEEs and storage encryption schemes can fail in the case of aliasing. The experimental work in later chapters focuses specifically on the address-translation path of CXL Type-3 memory devices. Furthermore, it evaluates whether comparable aliasing behaviour can be induced within the CXL.mem protocol stack, without requiring physical modification of the DRAM modules themselves.

2.5 Research Gap and Classification of this Thesis

In recent years, the technical documentation on CXL has become significantly more comprehensive. The first official CXL specifications already describe the functional structure of the standard in great detail, in particular the protocol layers, enumeration via PCIe, and the configuration of the HDM decoder [Com20]. Research circles have also produced a number of papers discussing the use of CXL for memory pooling, composability, or tiered memory [HDL+26, YNK+24, HLYC25].
However, these documents all have one thing in common: they primarily describe the functionality, performance characteristics, or software integration of CXL. Security-related considerations are mentioned only marginally, if at all. There is currently no systematic analysis of the security implications of CXL memory paths. Previous research has focused almost exclusively on evaluating the use of CXL as a high-speed protocol for attaching I/O

devices in cloud systems, or on performance aspects of CXL.mem accesses [CZC+24]. The interaction between CXL and classic protection mechanisms such as IOMMUs is rarely investigated. Furthermore, these studies remain conceptual and primarily highlight differences in the trust model between PCIe and CXL [SMM23b]. None of the studies published to date take a close look at possible attack surfaces, for example within the HDM mapping logic. There are a number of studies that deal with memory or addressing errors in DRAM-based systems (e.g. *Rowhammer* [KDK+14], *DRAMA* [PGM+16], *BadRAM* [MWO+25]). However, these studies refer exclusively to classic DDR memory or memory encryption scenarios in TEEs. The attacks in these studies rely on a wide variety of methods, including electrical interference effects, timing side channels, or manipulated SPD configurations. Therefore, these attacks cannot be easily transferred to the CXL ecosystem. The reason for this is the additional protocol layers, DVSEC structures, and logically configured HDM decoders, which have completely different attack possibilities.
This raises the following research question: *How does the extended address path of a CXL Type 3 device affect security-critical conditions?* It is unclear whether faulty designs in the FPGA-based implementation can lead to address aliasing. There is also a lack of experimental data on whether CXL devices can provide inconsistently or only partially activated HDM areas without being noticed. From the operating system's point of view, this would lead to incorrect assumptions about memory isolation. This work addresses precisely this issue. Through targeted modification of an FPGA-based CXL Type 3 prototype, possible aliasing effects, decoder anomalies and mapping inconsistencies are empirically investigated. This work thus closes a gap between the previously purely functionally oriented CXL documentation and practical security research in the field of memory-centric architectures.

3 Research Infrastructure

This chapter outlines the research infrastructure, which served as the basis for all experiments. It describes the hardware platforms, software environments, and supporting tools that were employed to implement, evaluate, and validate the proposed concepts.

3.1 Hardware

The experimental platform for this work is based on a combination of an FPGA-based CXL Type 3 device, conventional DDR memory, and a current host CPU with PCIe 5.0 and CXL support. The CXL device was configured based on Intel's official Example Design for a CXL Type 3 device [Int24]. Modifications had to be made to ensure that the design was compatible with the Mercury A2700 FPGA development board. The following section describes the key hardware components relevant to the measurements and implementation of the CXL Type 3 device.

FPGA-Board A Mercury A2700 module serves as the core of the implementation [Ter]. This module uses an Agilex FPGA of type *AG1B027R29A1E2VB*, which has extensive on-chip RAM blocks as well as dedicated PCIe 5.0 and CXL hard IP. Intel implements only the physical layer, which is shared between PCIe 5.0 and CXL, as hard IP logic, while the higher protocol layers are provided as soft IP, allowing developers to inspect and adapt the source code. This guarantees timing-critical parts of the protocol on the hardware side, so that the FPGA fabric only has to implement the device-specific functions and data paths. One advantage of the A2700 board is that the signals of the central system components are fully routed into the FPGA fabric, allowing them to be directly accessed and extended within custom logic. The A2700 block diagram shows that, among other things, the DDR4 interfaces are directly connected to the Agilex FPGA [Ter24]. This means that there are no additional switches or multiplexers. Latency or protocol behaviour are therefore not additionally distorted, which is particularly advantageous for security analyses.

Memory The A2700 board offers four DDR4 SO-DIMM slots, each designed as independent memory channels (SO-DIMM A/B/C/D) [Ter24]. Slots B, C and D are assigned to the FPGA EMIF (External Memory Interface) by default. SO-DIMM A can be used by the Hard Processor System (HPS) or integrated by the FPGA if the HPS memory is not

required. All four channels support up to 8 GB DDR4. Slots B, C and D allow up to 16 GB per module, depending on the configuration. For this work, slots C and D were used with 8 GB DDR4 memory each, so the CXL device later has 16 GB DDR4 memory [Ter24]. The connection is made via dedicated DDR4 I/O banks of the Agilex FPGA. The manual documents all address, data, and control lines, which is helpful for debugging and timing analyses (e.g. table for DDR4C/DDR4D signals) [Ter24].

Host System A server with a Intel Xeon Silver 4514Y processor (*Xeon 5*) [Int23b] was used as the host system. This supports PCIe 5.0 natively and thus provides the link speeds required for CXL. The system was built around a Supermicro X13SEI-TF mainboard, whose BIOS implements the necessary CXL capabilities to enable communication between CPU and FPGA [Sup23]. The FPGA was connected directly to the system's root complex via a fully connected x16 slot, so that the entire CXL negotiation took place during the regular PCIe enumeration phase.

3.2 Software

The software environment consists of three central components, which were used for the development and operation of the FPGA-based CXL Type 3 device: the Linux host system including the CXL-enabled kernel stack, the FPGA toolchain and various tools for building, debugging, and measurements. The following sections provide an overview of the software components used and the role they played within the experimental platform.

Linux Kernel The host system ran a Linux kernel version `6.14.0-cxl-ibt`. This already contains a current implementation of the CXL subsystem. The kernel series supports the complete probe and initialisation logic for CXL.io, CXL.cache and CXL.mem. Therefore, it includes the kernel modules `cxl_core`, `cxl_pci`, `cxl_mem` and the associated sysfs interfaces, which is a basic requirement for the experiments. The kernel has been configured so that the CXL drivers are available statically or as modules, including support for HDM decoders, DVSEC evaluation and memory region management. The distribution of the kernel plays a minor role in the experiments. The decisive factor was the availability of the current CXL subsystem functions.

FPGA Toolchain *Quartus Prime Pro* version 24.3 was used for the synthesis, fitting and deployment of the CXL design. In addition to the compiler, the toolchain also includes the *Platform Designer*, which was used to configure internal AXI interconnects and DDR

controller instances. The build process generates the bitstream files for the FPGA, allowing different file formats and configurations to be selected for the bitstream file.

Helpful Tools In addition to the kernel and FPGA toolchain, userspace programs that directly access the CXL memory areas were particularly helpful. These programs were used to measure latencies (Algorithm 1) and validate whether an address range was correctly mapped according to the HDM decoder settings, or to repeatedly read and write specific patterns (e.g. Figure 5.3). In parts of the analysis, PCIe debugging tools such as `lspci` and `setpci` were also used to investigate errors in a targeted manner. In addition, the kernel tracing infrastructure was used to compare the DVSEC structures of the FPGA with the decoders implemented in the device.

3.3 CXL Example Design

The central FPGA design that served as the basis for this work is based on the CXL *Type 3 Device with DDR* Example Design provided by Intel [Int24]. This example design implements the basic architecture of a CXL Type 3 device. It is therefore a memory expander that can be addressed on the host side via the CXL.mem path. The example design combines several hard and soft IP components, including CXL-enabled PCIe hard IP, AXI bridges, a DDR controller, and various infrastructure modules, e.g., for reset or clocking. The example design used was originally intended for a different Agilex board. It was therefore not possible to adopt the design directly. The *AG1B027R29A1E2VB* FPGA used in this work was not compatible with the example design. The pin assignments, memory topology and individual clock domains differ significantly. Therefore, several adjustments had to be made when initialising the project, most of them in the *constraint* files, more specifically in the file for I/O banks and DDR4 signals. These changes are described in more detail in Chapter 5, System Bring-Up. At this point, it suffices to note that the original design only served as a functional starting point.
Several soft IP blocks play a central role in the structure of the reference design. The folders under *hardware_test_design* contain most of the soft IP. The *common* subfolder plays a central role here. This is a collection of generic components that are used in many Intel reference designs, for example clock modules, simple AXI register names or utility functions. These components form the basis for the internal data path through which the AXI transactions generated by the CXL hard IP are routed towards the DDR4 interface.
The functional core of the design consists of three main paths, also visible in Figure 3.1:

- *CXL/PCIe IP path*: The hard IP implements the physical layer and PCIe link initialization, which cannot be modified by the FPGA design. All higher protocol layers,

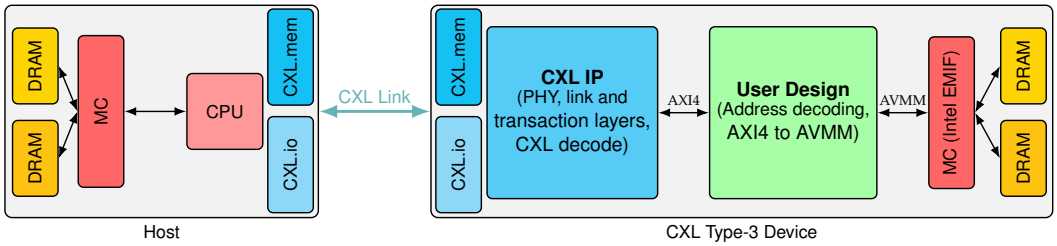

Figure 3.1: Simplified high-level system architecture of the experimental setup.

including CXL.io as well as the multiplexing logic for CXL.cache and CXL.mem, are realized in CXL soft IP. Once the FPGA design is programmed, the AXI master port is permanently available, providing the entry point through which incoming memory requests are routed into the internal fabric of the FPGA.

- *AXI fabric/interconnect level*: The AXI signal systems are routed via several interconnect modules that perform address decoding, width conversion and clock coupling. This level connects the hard IP to the DDR controller and plays an important role in subsequent measurements and experiments with aliasing behaviour.

- *DDR controller and memory backend*: The design uses Intel EMIF IP to control the DDR4 SO-DIMMs of the A2700. The combination of hard IP, AXI fabric and EMIF forms the complete CXL Type 3 device. From the host's perspective, it later appears as an additional NUMA memory node.

For the host system, it appears as the sample design behaves like a regular CXL Type 3 device: It provides DVSEC entries in the PCIe configuration space, can also provide multiple HDM decoders and, depending on the number of HDM decoders configured, can offer one or more contiguous physical memory areas. Both structures form the logical interface through which the host recognises and configures the device's available memory space and later addresses it via CXL.mem transactions.

4 Implementation and System Bring-Up

Bringing up the CXL Type 3 design on the Mercury A2700 required several steps that went beyond simply synthesizing the Intel Example Design. The Example Design is essentially a complete Type 3 implementation, but it was not originally designed for our variant of the Agilex FPGA. This resulted in deviations in the pin assignment, power supply configuration and individual project parameters, which had to be corrected before the design was functional. The following sections document this work, which was carried out during the bring-up process.

4.1 Power Management Configuration and Basic Setup

After instantiating the CXL *Type-3 Device with DDR* Example Design, several device-specific options first had to be set correctly in *Quartus Prime Pro*. Without this basic configuration, the design can be synthesised, but this results in a non-functional bitstream. A key element was the configuration of the PMBus-based voltage regulators. The Mercury-A2700 uses an LTC388 regulator, whose parameters must be stored in the project:

- Bus speed mode: 400 kHz
- Slave device type: LTC388
- Number of slave devices: 1
- PMBus address: 75 (0x4B)
- Voltage output format: Linear format (N = -12)
- Unit: Volts

These values come from the official board design and are described in the Golden Hardware Reference Design (GHRD). Without this information, the voltage regulation does not start correctly and the FPGA configuration remains unstable or aborts prematurely. In addition, the reference clock source `refclk4` was not set correctly. In the *Pin Planner*, a component of Intel's Quartus Prime Pro design software, `refclk4` had to be explicitly set to pin LB60. Only then was a stable reference clock available for the PCIe/CXL hard IP. This correction was necessary to ensure that the link training process was completed.

4.2 Initial Attempt: Error

Once the basic settings had been adjusted, it was time for the first attempt. After the initial compilation and programming of the FPGA, the device was initially recognised correctly by the host. The first `lspci` call already showed that it was enumerated as a CXL.mem device. The HDM decoder also appeared in the output. However, the following appeared in the DVSEC area:

```
Range1:  [...]  Valid+ Active-
```

In this case, the + signifies that a bit is set and a − signifies that a bit is not set. In addition, the Linux kernel log reported the following, among other things:

```
cxl_pci 0000:17:00.0:  timeout awaiting memory active after 60
seconds media not active (-110)
```

From this, it can be deduced that the CXL enumeration itself was successful, but the memory area of the Type 3 device was detected as inactive by the host. This error pattern points to several possible sources of error. For example, the DDR4 controller is not calibrated, the HDM decoder does not have a valid physical memory area, or the DVSEC ranges point to addresses that do not exist internally. Since the host expects CXL.mem devices to have fully available memory immediately after reset, any error in the DDR backend results in an inactive state.

4.3 Analysis: Cause of the Error

Evaluation of the registers and kernel messages quickly revealed that neither DVSEC nor the CXL stack were causing the problem. The debug output from the PCIe configuration space indicated correct CXL capabilities, but zero-size or uninitialised memory ranges. To monitor the internal initialization state of the DDR subsystem, I extended the existing Control and Status Register (CSR) `DEVICE_AFU_STATUS2` in the module `cafu_csr0_wrapper.sv` into a debug register. I enabled the host to read the complete bring-up status of the FPGA via BAR0 MMIO. Using this repurposed register, it became evident that DDR4 calibration had failed. The decisive bit `csr0_mem_active`, which controlled the `Active` status and indicated successful initialisation, remained `0`. This made it clear that the HDM decoder could not address any physical memory. The debug outputs thus showed that the memory interface was unusable.

4.4 Solution: Pinout

After thoroughly analysing the RTL code of the soft IP, it became clear that the problem was not with the code, but elsewhere: the pinout. For DDR memory, in our case DDR4

SO-DIMMs, it is essential that all pins are correctly assigned so that memory initialisation and access work correctly. After examining the pinout file *qsf_device_pinout.tcl* of the Example Design, it quickly became clear that the pins were incorrectly assigned. Some DDR4 channels were assigned to pins that did not exist on our board or were simply mapped incorrectly. The consequence of this was that the EMIF could not calibrate. The solution was the GHRD of the A2700. It contained all the correct pin assignments. Using this pinout as a template, the pinout of the example design was then corrected. After adjusting the entire SO-DIMM pin assignment, the EMIF was able to initialise correctly for the first time. More specifically, EMIF calibrated successfully, the HDM decoder reported valid ranges, and the `lspci` output now showed `Range1: [...] Valid+ Active+`, which means that the HDM decoder is functional and the DDR4 memory is correctly initialized and connected. The kernel therefore recognised the memory area without any errors. This made the entire CXL.mem memory area available to the host, which laid the foundation for the experiments in the following chapters. In addition, the expected 16GB of additional memory were visible in the userspace, confirming that the CXL device had been successfully integrated.

4.5 Experimental Evaluation of CXL.mem Functionality

The objective of this chapter is to demonstrate that the implemented CXL Type-3 device is functionally complete and recognized by the host system as a regular, usable memory endpoint. This is particularly important, since consistent and correct memory mapping is a prerequisite for the subsequent security analyses. Therefore, the first step was to verify whether the FPGA memory is correctly integrated into the host's physical address space and to examine how its access characteristics compare to local DRAM. The latency measurements are based on well-defined memory access patterns such as random access, and pointer-chasing. These patterns are chosen because they minimize the influence of caches and hardware prefetchers, thereby enabling a realistic assessment of memory latency.

Objectives and Latency Measurement Strategy Before conducting the actual security experiments, it is necessary to ensure that:

1. The memory appears in the Linux system as a NUMA node
2. Accesses via userspace programs are deterministically possible
3. The latency characteristics correspond to what is expected from a CXL Type-3 device connected via CXL

```
available: 2 nodes (0-1)
node 0 cpus: 0-31
node 0 size: 128350 MB
node 1 cpus:
node 1 size: 16384 MB
node distances:
  0->0 = 10   0->1 = 14
  1->0 = 14   1->1 = 10
```

Listing 4.1: NUMA output showing two nodes. Here Node 1 reflects the CXL.mem device and Node 0 the Host system

For the latency measurements, randomly linked pointer-chasing structures were used. By employing pointer-chasing with a randomized permutation of cacheline links, each memory access depends on the previous one, effectively suppressing hardware prefetching and reducing cache efficiency so that true memory latency can be measured. These measurements were carried out exclusively on the host system with the help of own written user space tools, i.e., Algorithm 1. The memory allocation of the accesses was managed using `numactl`, which made it possible to directly access exclusively either local DRAM or CXL memory.

Host-side View of the CXL Device After the successful bring-up, `lspci` correctly identified the device as a CXL.mem endpoint. However, it was still necessary to demonstrate that the CXL device was indeed fully usable. For this purpose, the command `numactl -H` was first executed. This command lists all NUMA nodes recognized by the Linux NUMA subsystem. The output, see Listing 4.1, showed two NUMA nodes. Node 0 contains the host system and the local DRAM, which can be clearly recognized by the presence of CPUs, since a CXL Type-3 device does not possess its own compute logic. In contrast, Node 1 represents the CXL device, identifiable by the absence of CPUs and the provision of approximately 16 GiB of memory. The node distances for the CXL device were, as expected, higher than the host-to-host distance but lower than typical NVMe or remote PCIe values [KGJ+25]. In addition to the NUMA outputs, further properties of the overall system were checked. The host correctly recognized the CXL port, as confirmed by the message in the kernel log `dmesg` output: `cxl_pci 0000:17:00.0: enabling device`. The HDM decoder entries exactly reflected the memory size configured in the FPGA. The additional memory block was also reliably detected at the operating system level. This is evident not only in the `numactl -H` outputs but also in the output of `free -h`, which suddenly showed 16 GiB more memory available. Direct accesses from userspace with various read and write patterns were consistent and without errors as well. Overall, it can be concluded that the entire system operates consistently from the

Algorithm 1: Pointer-chasing latency measurement on NUMA nodes

Input: NUMA node n, buffer size S, iterations I
Output: Latency statistics (cycles, ns)

```
Allocate buffer of size S on node n
Construct random permutation of cacheline indices
Build pointer-chasing cycle over all cachelines
Warm up by traversing the cycle once
for i ← 1 to I do
    t0 ← rdtsc_begin()
    idx ← next[idx]                          // dependent load
    t1 ← rdtsc_end()
    Record (t1 − t0)
Compute min, max, average, percentiles
Report results in cycles and ns
```

operating system's perspective and provides a stable basis for further analysis.

Measurement: Dependent Pointer Chasing To determine the actual memory access latency, a pointer-chasing structure was constructed in which each address points to another random address. This method is intended to eliminate factors that could distort the measurement, such as hardware prefetching, speculative access path optimizations, sequential traversal in L1/L2 caches, and any effects that would merely reflect bandwidth measurements. Timing was performed using the processor's Time Stamp Counter (TSC) via `RDTSC/RDTSCP` instructions, combined with `LFENCE` barriers to enforce serialization, see Algorithm 1. Execution times were averaged over one million iterations. The memory allocations were explicitly bound to the selected Node to ensure a clear distinction between host DRAM latency (local) and CXL latency (remote). The measurement results exhibit the expected characteristics of a CXL.mem path, as shown in Table 4.1.

Table 4.1: Comparison of the latency for host DRAM and CXL.mem (pointer chasing). Each measurement was based on one million iterations. Outliers were not removed, instead, latency distributions are reported using min, max, average, and percentiles (p50, p90, p99).

Memory	Ø cycles	Ø time (ns)	p50 (ns)	p90 (ns)	p99 (ns)	Max (ns)
Host DRAM	333	166	159	186	319	36500
CXL.mem	842	421	411	429	682	36000

The results clearly illustrate the characteristic latency structure of CXL.mem accesses. Compared to local DRAM, access times are approximately 2.5 to 3 times higher, reflecting

the additional data transfer via the CXL path. The distribution of the measured values is very stable overall. The median and mean are closely matched, underscoring the consistency of the results. Isolated outliers in the upper range (p99) are typical for PCIe-based transfers and can be plausibly explained by sporadic packet latency or interference in the last-level cache. Overall, the measured latencies are within the range of published performance evaluations of CXL, which report end-to-end read latencies of approximately 250-350 ns [LHX+24]. Therefore, it can be clearly concluded that the investigated CXL Type-3 device functions correctly in terms of both functionality and performance, fulfilling the expected characteristics of the CXL.mem path.
The experiments conducted confirm that the implemented CXL Type 3-device fulfills the properties expected in the specification: It is unambiguously recognized as a CXL.mem device, initializes the HDM decoders correctly, and provides a usable, stable memory area in the host's physical address space. The measured access latency is within a plausible range and exhibits the typical delay for PCIe-based storage devices. Thus, both the hardware and software environment are sufficiently validated, allowing the security analyses described in the following Chapter 5 to be performed on this basis.

5 Security Experiment: Address Aliasing in CXL.mem

The previous chapter clearly demonstrated that CXL.mem allows remote memory endpoints to be embedded into a host's physical address model in such a way that they behave like regular NUMA regions. This makes the device's address decoding a security-critical component. Manipulations along this path can lead to logically separate address ranges being mapped to the same physical memory cells. Such behavior would be problematic: applications in cloud and data center environments, or virtual machines that are meant to be isolated from each other, could unintentionally use the same memory area. Against this backdrop, this chapter specifically examines how manipulated or faulty address decoding in a CXL Type 3 device can lead to address aliasing and how this behavior can be practically demonstrated.

5.1 Threat Model

The experiment conducted is based on a simplified yet realistic attack model that highlights the specific risks of the CXL.mem protocol related to address aliasing. The attacker controls a CXL Type-3 memory device and can freely modify its internal configuration, including soft IP, address decoding paths, and the translation from host physical addresses to device-internal memory locations. The attacker can therefore enforce arbitrary address mappings inside the device. In addition, the adversary has administrative control over the host operating system. This allows the attacker to configure the system such that address aliasing does not trigger immediate system failures. In particular, the attacker can influence memory hot-add behavior, NUMA node placement, and the kernel's handling of CXL memory regions. The attacker's primary objective is to violate memory isolation by creating deliberate alias mappings within the CXL device. By forcing two distinct host physical addresses to converge onto the same internal memory region, the attacker seeks to undermine data integrity or enable unintended data sharing. The host does not validate the internal HDM decoder or address translation logic of a CXL device [Com20]. The system therefore relies on the correctness of the device's implementation. In this threat model, the device does not need to remain externally unnoticeable. Instead, the host actively cooperates under the attacker's control to prevent crashes and ensure that the system continues operating with the aliased memory range. Once the device is integrated and stable, the adversary can exploit the internal alias mapping to break logical memory

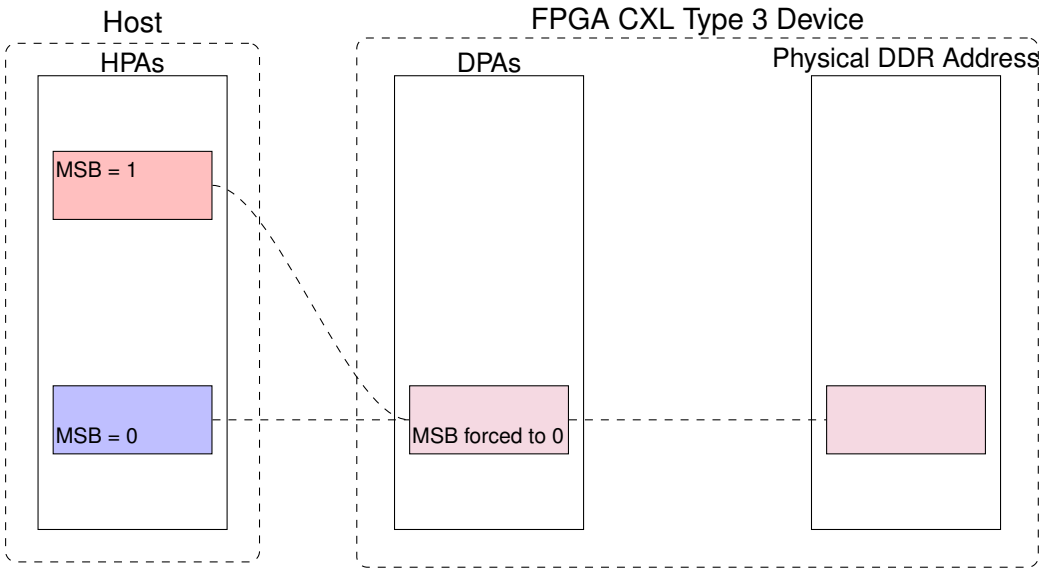

Figure 5.1: Incorrect configuration of the CXL device causes two different host physical addresses (HPA), which only differ in the MSB, to point to the same device physical address (DPA).

separation while the host perceives the memory range as valid and usable.

5.2 Experiment: Induced Address Aliasing via Bit Masking

The basic idea of the experiment, which can be seen in Figure 5.1, was to configure two logically separate memory regions so that, from the host's perspective, they appear as distinct areas, but internally address the same physical DDR memory. Technically, this was achieved through a targeted manipulation of the address decoding: The most significant bit (MSB) of the AXI address in the CXL.mem path was permanently set to 0. This bit normally determines whether an access falls into the lower or upper half of the allocated memory. By setting this bit to zero, host addresses that differ only in this bit are no longer distinguishable. This results in immediate address aliasing. This scenario illustrates a security risk in line with the previously described threat model: The host trusts the device's correct address decoding but fails to recognize that internally logically isolated memory areas are being merged. This violates memory isolation.

Implementation The aliasing logic was implemented exclusively in the address path of the CXL.mem data path, clearly visible in Figure 5.2. In the example design used, the translation of the AXI addresses of the CXL.mem transactions to channel-local memory addresses takes place in the module `axi2avmm_bridge.sv`. This module forms the in-

```
// Aliasing permanently active
parameter bit ENABLE_ADDR_ALIASING = 1'b1;

// Channel-local aliasing: Force MSB to 0 (Write)
o_avmm_address[genavmm] = {'0, // Leading zero bit in AVMM address
  (i_to_mc_axi4[genavmm].awaddr[CXLIP_CHAN_ADDR_MSB:CXLIP_CHAN_ADDR_LSB] &
   {1'b0, {(CXLIP_CHAN_ADDR_MSB-CXLIP_CHAN_ADDR_LSB){1'b1}}})
};

// Channel-local aliasing: Force MSB to 0 (Read)
o_avmm_address[genavmm] = {'0,
  (i_to_mc_axi4[genavmm].araddr[CXLIP_CHAN_ADDR_MSB:CXLIP_CHAN_ADDR_LSB] &
   {1'b0, {(CXLIP_CHAN_ADDR_MSB - CXLIP_CHAN_ADDR_LSB){1'b1}}})
};
```

Listing 5.1: Masking of the MSB (simplified)

terface between the CXL soft IP (AXI4) and the EMIF interface (Avalon-MM) of the DDR memory. From a security perspective, this is the crucial point, as a concrete physical memory address is generated here from a global address provided by the host. The address path can be summarized as follows:
In the reference design, the channel-local address is derived from a fixed slice of the AXI address. The boundaries for this are defined in the global package `cxlip_top_pkg.sv` and vary depending on the number of memory channels, as either more or fewer bits are required for channel selection. However, the boundaries are dynamically applied, and therefore do not need to be considered for masking the most significant bit (MSB). To create address aliasing, the most significant bit of this channel-local address is permanently set to 0. This causes the two 8 GiB areas, which differ only in this bit, to collapse. Listing 5.1 implements address aliasing by selectively masking individual bits within the AXI-to-AVMM bridge. The parameter `ENABLE_ADDR_ALIASING = 1'b1` enables this behavior permanently at compile time. In the channel-local path, the most significant bit of the respective slice address is set to zero, and a leading zero bit is inserted into the AVMM address. This maps two logically separate address ranges to the same lower memory segment. The rest of the address structure remains unchanged, so no additional logic such as wrapping or segmentation is introduced. This mechanism creates consistent aliasing, affecting both read and write operations, although certain configurations may cause host instability, which will be discussed in the next chapter.

5.3 Host-side Testing

After the modification was implemented, it was necessary to verify on the host side whether the expected address aliasing actually occurred. It quickly became apparent that

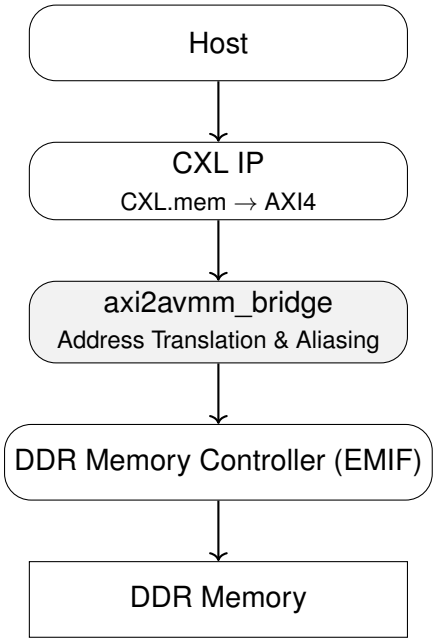

Figure 5.2: Simplified address path inside the CXL Type-3 device. The CXL IP converts host-issued CXL.mem requests into AXI4 transactions, which are then translated into channel-local DDR addresses by `axi2avmm_bridge.sv`, where the aliasing logic is applied.

different access modes of the Linux kernel handle CXL.mem very differently, and that not every mode is suitable for making physical aliasing effects visible.

First Attempts in RAM Mode Initially, the CXL memory was integrated as regular system RAM by bringing the entire HDM Region 0 area online. However, this led to reproducible system freezes as soon as more than approximately 8 GiB of memory was used. The reason for this is structural: In RAM mode, the host treats the memory areas provided by the CXL device as true physical page frames. The Linux kernel necessarily assumes that each physical page exists uniquely. This assumption was violated by the artificial aliasing: The kernel distributed logical pages beyond the 8 GiB limit, even though these internally pointed to already used physical cells. As a result, invalid returns, duplicate physical pages, cascades of page faults, and ultimately a complete system crash occurred. This made it clear that RAM mode was unsuitable for the experiment and that the tests must be conducted exclusively in a mode where physical addressing is direct, shallow, and accessible without a kernel abstraction layer.

DevDAX Mode for Safe Testing The measurements were therefore completely moved to Device DAX mode (DevDAX). DevDAX exposes a CXL memory region as a raw, page-aligned character device (e.g., `/dev/daxX.Y`) that can be accessed directly from user space, without integrating the memory into the system's RAM allocator or page management [Lin25a]. The advantage of this mode is that it eliminates all operating system level abstractions such as the page cache, demand paging, or NUMA placement. Accesses are performed as a 1:1 MMIO mapping, without page faults, or NUMA abstraction. DevDAX is therefore ideal for this experiment, as it allows precisely what we want to investigate: Direct testing of physical aliasing effects in the memory of the CXL device, without kernel intervention. It's important, however, that the CXL device is not automatically integrated into the host's system RAM. Otherwise, this can cause difficulties when switching to DevDAX. As a first step, two small changes need to be made to the GRand Unified Bootloader (`GRUB`). Specifically, in `GRUB` under `GRUB_CMDLINE_LINUX`, it is essential to add two additional commands: `memhp_default_state=offline`, so that the CXL memory, which is also hot-plugged memory, remains offline by default, and `movable_node`, so that the kernel does not store its own fixed structures, such as kernel stacks or page tables, on this memory. The device then needs to be switched to DevDAX mode. This is done using the command `daxctl enable-device daxX.Y`, where `daxX.Y` is the CXL device. After this command has been executed, the device can be passed to userspace programs as `/dev/daxX.Y`.

First Test: DAX Alias Stress The first test program `dax_alias_stress.c` (Lisitng 1) is a simple yet effective tool for confirming the aliasing effect. It works with 64-byte steps and writes selected patterns to the lower (LOW) and upper (HIGH) address ranges, respectively, to then check if the other half displays the same value.
The tests performed, as described in Figure 5.3:

- Pass 1: LOW is filled with `0xAA...AA` $\Longrightarrow$ HIGH must have the same content
- Pass 2: HIGH is filled with `0x55...55` $\Longrightarrow$ LOW must have the same content
- Pass 3: Optional full streaming write across the entire range

The program clearly demonstrated the presence of aliasing.
Although occasional mismatches occurred, see Listing 5.2, the majority of the aliasing worked. These mismatch effects resulted from the fact that the test program initially wrote only 8-byte values per iteration while leaving the remainder of the cache line untouched. Even in DevDAX mode, the CPU still allocates and updates full cache lines on writes, meaning that partial writes can leave stale data in the cache unless the entire line is over-

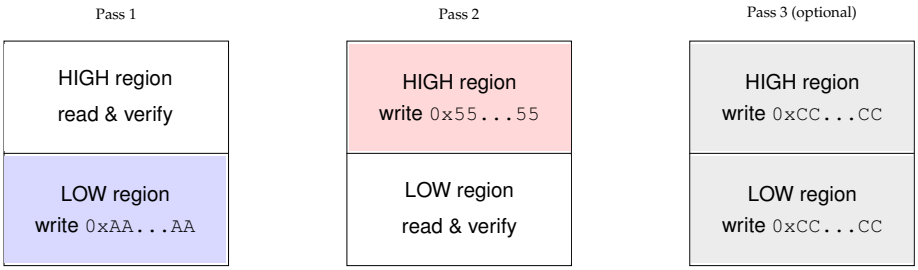

Figure 5.3: Verification procedure of the induced address aliasing. The program writes known patterns into one half of the CXL.mem region and verifies whether the other half mirrors the same content, thereby confirming that both ranges refer to the same physical memory.

```
Pass1: fill LOW with A, verify HIGH == A
Mismatch@200000180: 5555555555555555
...
Mismatch@200000640: 5555555555555555
Pass1 errors: 14027
Pass2: fill HIGH with B, verify LOW == B
Mismatch@180: aaaaaaaaaaaaaaaa
...
Mismatch@6c0: aaaaaaaaaaaaaaaa
Pass2 errors: 16524
Pass3: full streaming write across all 16 GiB (pattern C)
Done.
```

Listing 5.2: Output of `dax_alias_stress.c` (shortened)

written or explicitly flushed [Red]. However, a failure of the aliasing would result in several million errors, as seen in a previous run where the aliasing logic was not yet correctly implemented: `Pass1 errors: 67108864` and `Pass2 errors: 67108864`.

Extended Test: DAX Alias Stress with CLFLUSH To completely eliminate the cache effects observed in the first test, a second test program, named `dax_filler_stress_clflush.c` (Listing 2) was developed. This program extends the original approach with several additional mechanisms that enable consistent and cache-independent validation of the alias effect. First, writing is no longer performed at the word level, but across entire 64-byte cache lines `write_line_64B`, ensuring that each cache line is completely overwritten. Before each check, a function called `flush_range()` is called to ensure that the affected memory ranges are removed from the CPU caches. It invalidates both the cache lines in the area that was written and the corresponding aliased region. To strictly order memory operations and prevent potential reordering by the processor, memory fences `__sync_synchronize()` are implemented. Finally, both alias ranges are invalidated

before each read access, so that subsequent checks are based exclusively on consistent data in main memory. These measures create a significantly more robust test environment that reliably eliminates cache artifacts and thus allows for a more precise analysis of alias behavior.
The new test logic:

- Pass 1:
  - I LOW completely written with pattern A (`0xAA...AA`)
  - II Flush LOW + Flush HIGH
  - III HIGH must be identical to A
- Pass 2:
  - I HIGH completely written with pattern B (`0x55...55`)
  - II Flush HIGH + Flush LOW
  - III LOW must be identical to B
- Pass 3: Complete streaming write over 16 GiB

With the implemented anti-caching mechanisms, the second test program exhibited completely consistent behavior. No errors occurred in either Pass 1 or Pass 2 (`errors: 0` in each case), and this was repeated over several runs without a single deviation. The result thus proved to be stable and completely deterministic, clearly confirming the effectiveness of the implemented flushing and synchronization measures.

Interpretation The measurements clearly show that both address halves contain identical data and that the aliasing effect is completely symmetrical, both in the LOW to HIGH and HIGH to LOW directions. Only after the complete removal of all cache artifacts did the true behavior become visible, thereby eliminating the previously observed inconsistencies. It is also noteworthy that the kernel does not detect the aliasing. The freeze behavior observed in RAM mode further confirms that an operating system cannot safely operate if two distinct physical page frame numbers actually alias the same DRAM cells, unlike DevDAX, which correctly processes this special case. This demonstrates that the manipulation affected the internal address mapping of the CXL device as intended and that the host's CXL subsystem lacks a mechanism to detect such malfunctions.

5.4 Difference between Test Mode and Real Attack

The proof-of-concept presented here was developed in a controlled test setup where masking was activated via a hard-coded constant in the soft IP code and was therefore always effective. This approach does not necessarily reflect the actions of a real attacker, but it illustrates the fundamental vulnerability of the address path. In the experiment, address manipulation occurred openly in the Example Design. In reality, however, such a change would have to be covert, for example, through a compromised IP block, a malicious third-party controller, or an error in the verification process. Furthermore, the aliasing was only visible in the DevDAX test environment, as the memory is mapped 1:1 there. A manipulated device would likely have to block the upper half of its memory or ensure that this half is not loaded into RAM to avoid freezes and crashes. If this were successful, the errors would be even harder to detect in RAM mode, as many corruption events only occur after a time delay [PCK21]. Finally, the test setup used a pure Type 3 implementation without additional protection mechanisms such as memory tagging, MAC checks, integrity verification, or host-side consistency checks. An attacker would have comparable conditions here, since the CXL specification allows devices to define their internal memory organization independently, without the host verifying the physical mapping logic [Com20]. Address aliasing is a real, non-negligible attack vector in CXL.mem. The host has no mechanisms to detect faulty address decoders and relies entirely on the integrity of the device.

6 Discussion and Security Implications

This chapter places the results of the address aliasing experiment within a scientific context. First, parallels and differences to existing attack classes are highlighted, followed by a discussion of the specific features of the CXL.mem architecture, and finally, security-relevant implications for trust domains, memory pooling, and cloud-based systems are presented. In conclusion, a perspective on possible dynamic attack methods beyond the proof-of-concept implemented in this work is provided.

6.1 Classification within existing Research on Memory-Aliasing Attacks

Aliasing in physical memory is a well-known but rarely directly investigated problem. Studies like *BadRAM* demonstrate that minor manipulations of physical address bits can lead to significant violations of memory isolation. In *BadRAM* [MWO+25], selected physical page regions were overlapped through manipulated Serial Presence Detect (SPD) configurations. However, the system did not run stably when the entire memory area was available to the operating system in system RAM mode. That is why, the operating system was instructed via kernel parameters such as `memmap=nn$ss` to block the upper half of the affected regions. `memmap` blocks the region from ss to ss+nn i.e., the upper half, thus preventing crashes. Crucially, the entire physical memory must be known before the boot process. Only then can problematic address regions be effectively reserved [MWO+25].
BatteringRAM recently demonstrated that aliasing does not necessarily have to be static. The DDR4 interposer developed there allows for the manipulation of specific address lines at runtime. For example, bits A11 and A13 only affect the selection of rows, therefore, critical bits, which are responsible for the instruction format, are not changed, thereby selectively overlapping individual DRAM rows. This manipulation affects only the signaling on the command/address bus and does not alter the logical memory organization that the operating system reads during the boot process. Thus, the host's page tables remain formally consistent, and the system runs stably, even though two physical addresses beneath the memory controller actually point to the same row. The work shows that this can also compromise protected environments such as Intel SGX or AMD SEV-SNP. However, it is important to note, that the *BatteringRAM* attack targets specific DDR4 data lines. DDR5 reorganizes these data lanes and removes the exploitable structures, meaning the attack does not apply. TEEs such as Intel TDX, which operate exclusively on DDR5, are therefore

not affected by this technique [DMOVVB26].
The proof-of-concept implemented in this work differs from both approaches. The aliasing originates within the CXL Type 3 device, specifically within the AXI to AVMM address path. In this work, it is a static aliasing that is already active before the host integrates the memory area into the physical address model. As a result, the kernel lacks consistent information about the physical structure of the device. Consequently, when used in system RAM mode, reproducible crashes occurred as soon as the kernel accessed logical pages above the 8 GiB mark. The crucial difference to *BatteringRAM* is therefore that in this experiment, two logical host addresses already refer to the same physical location at the time of memory initialization, which is contradictory from the operating system's perspective and cannot be resolved. This observation illustrates that aliasing produces very different system behaviors in different architectures. Neither *BadRAM's* static reservation mechanisms nor *BatteringRAM's* dynamic interposer approach are directly transferable to CXL. CXL.mem has its own memory initialization phase, which occurs significantly later than with traditional DRAM, and the consistency check of the physical mapping logic is entirely shifted into the device's confidence domain.

6.2 Differences between CXL.mem and Classic Memory Architectures

The results of the experiment show that CXL.mem has a number of fundamental differences in terms of safety compared to local DRAM.

Separate Address Decoder in the Device In traditional systems, all address translation is handled within the CPU-adjacent memory controller. Errors in physical address decoding are therefore immediately apparent and result in hard errors. CXL completely offloads this logic to the external device. While the host knows which ranges the device exports, it does not know how these are translated internally. This is because the HDM decoder handles the internal translation, so any manipulated mapping logic remains invisible.

Hotplugged Memory A crucial aspect of the experiment was that the attempt to block the dangerous upper half of the memory, analogous to *BadRAM*, works differently. In contrast to traditional DRAM, CXL memory is discovered only after the kernel's early memory initialization phase and is hot-added as device memory, meaning that `memmap` never affects it. To recreate the *BadRAM* setup, the 16 GiB CXL memory window was therefore divided into eight 2 GiB hot-pluggable blocks, where only the lower 8 GiB (blocks `memory65`-`memory68`) were deliberately onlined in system RAM mode by default via set-

ting `/sys/devices/system/memory/memoryX/state` to `online_movable`, while the upper 8 GiB (`memory69-memory72`) were left offline. This selective onlining effectively reproduced the logical separation that *BadRAM* depends on: The operating system uses only the lower half as legitimate system RAM, while the upper half remains addressable but is never inserted into Linux's page allocator. When the upper 8 GiB were onlined as well, Linux attempted to use the entire 16 GiB window, and due to the FPGA's aliasing logic normal memory operations immediately corrupted kernel structures such as free-page lists and reference counters resulting in segmentation faults during early page-fault handling in userspace, even though no page tables were stored in the CXL.mem region itself.

Main Differences This makes it clear that the *BadRAM* attack and the Proof-of-Concept (PoC) are similar in their fundamental characteristics. However, the PoC is easier to implement for CXL memory, as it only requires a change to the CXL Example Design. Whether it also opens up attack surfaces to the same extent is still unclear. *BatteringRAM*, on the other hand, differs fundamentally from the PoC because, at runtime, it is limited to only a few non-critical memory row bits, resulting in stable system operating. Both attacks, however, have a clear weakness: they require direct hardware manipulation, either through a manipulated DDR4 module or a built-in DDR4 interposer. Furthermore, the manipulation must be adapted for each new generation of DDR modules. This is a major advantage of manipulated CXL devices: The attack is possible regardless of the DDR memory generation.

6.3 Risks for TEEs, Memory Pooling and Cloud Environments

The security implications are sometimes significant and primarily affect systems that rely on physical memory isolation.

Trusted Execution Environments (TEEs) Trusted Execution Environments (TEEs) like Intel TDX or AMD SEV-SNP rely on the unique and correct mapping of physical memory pages to a protected area. The hardware ensures that each protected page has a unique mapping to a guest physical page and that this page cannot be corrupted, overlaid, or mirrored by the host software or other virtual machines. This model assumes that the underlying physical addressing is stable and trustworthy [Int25, AMD20]. TDX, for example, performs an alias check directly during boot to ensure these assumptions [Int23a]. A CXL Type 3 device, which maps two logical address ranges to the same physical memory area, undermines this assumption. The TDX/SEV-SNP hardware has no direct insight into the

internal address decoding of a CXL device because isolation for TEEs is guaranteed only on system DRAM (TDX private memory or SNP-protected DRAM). CXL memory falls outside this model, therefore, some TEEs, such as TDX, do not yet support CXL [Int23a]. A device that generates internal aliasing would therefore not be checked by the usual TEE mechanisms. This creates several risks: A manipulated device could make data written by a protected VM to a specific CXL area visible again via an alias in another area. Even though modern TEEs apply their core structures like EPC, RMP, or ACTM exclusively to local DRAM, real-world workloads increasingly use offloaded memory in the form of paging or offload areas. If such data is stored in CXL.mem, it is conceivable that two VMs, or a VM and the host, could view or modify the same physical area via an internal alias. While a static alias, as in the proof-of-concept of this work, would be visible at the time of hotplugging and would likely lead to malfunctions or detection by alias checks, if these are implemented for CXL devices, it is conceivable that a malicious CXL device could activate such aliasing only at runtime. A dynamically configurable alias cannot be detected by either TDX or SNP mechanisms, as these exclusively monitor the system DRAM or perform alias checks during the initial boot process [AMD20, Int23a]. Such manipulations of device memory are considered "external memory" by the TEE architecture and are therefore outside its protection guarantees. The risk thus lies less in a direct breach of the TEE core (e.g., EPC or TDX private memory) but rather in undetected violations of memory segregation beyond the CPU memory controller. If CXL plays a larger role for externalized TEE data in the future, this risk will become even more significant.

Memory Pooling in Data Centers Another scenario with a significantly higher risk is the use of CXL for memory pooling. Here, multiple hosts access the same physically distributed device memory, sometimes via CXL switches or fabric topologies. This form of disaggregation is particularly expected in cloud and high-performance computing (HPC) environments and is considered one of the core use cases for CXL. In such environments, clearly defined isolation between tenants is essential. Aliasing within a single CXL device can severely compromise this isolation. For example, if a device intentionally overlaps certain address ranges through manipulation, memory segments assigned to two different hosts could access the same physical rows. The result would be silent data corruption, inexplicable synchronization errors, or, in the worst case, direct information leaks between tenants. Since the memory organization of a CXL Type 3 device is determined solely by its internal logic, and the host only receives declarative metadata (HDM decoder, DVSEC ranges), there is currently no way to verify the correctness of this internal mapping structure. A malfunctioning or compromised address decoder is undetectable from the host's perspective. This means that proper isolation within a memory pool depends entirely on

the integrity of the device. This creates a systemic risk, especially in multi-tenant clouds. Even if hypervisor or TEE isolation is correctly implemented on the host side, a faulty CXL device can breach the physical basis of this isolation without any of the software components involved being able to detect the violation.

Supply Chain Risks The increasing prevalence of CXL means that devices originate from a wide variety of sources: large manufacturers, FPGA-based specialist solutions, startups, or internal developments by cloud providers themselves. Many of these devices contain complex soft IP, microcode layers, firmware updaters, and configurable address decoding logic. This heterogeneity makes CXL significantly more vulnerable to supply chain risks than conventional DRAM. While DRAM modules are generally highly standardized and relatively simple in design, CXL devices contain complete protocol stacks, switch logic, memory controllers, and sometimes configurable RISC-V microcontrollers for runtime control. A compromised device in this environment could deliberately implement aliasing, whether through manipulated firmware, intentionally placed RTL backdoors, or subsequent microcode updates. The host currently has no way of detecting this manipulation. While CXL specifies control registers and discovery mechanisms, it does not provide hardware-based verification of the actual physical mapping logic. This creates a novel attack scenario: An attacker who manipulates a CXL memory device or introduces a counterfeit device into a supply chain could inconspicuously place aliasing functions and activate them only after deployment. In large cloud environments where devices are deployed in large numbers and often automatically, this is a real risk.

6.4 Dynamic Aliasing as a possible future Attack Class

This work investigated an attack based on static manipulation of address decoding. In this process, one address bit is permanently masked, so the resulting alias structure remains fixed within the system. However, the architecture of CXL devices, in principle, opens up the possibility of implementing significantly more sophisticated attack methods. Such a device could implement dynamic aliasing, which is similar in its functionality to the well-known *BatteringRAM* attack, but offers a considerably broader attack spectrum. One conceivable scenario would be the introduction of an internal Control and Status Register (CSR) in the CXL device, allowing individual bits of the channel-local address to be selectively toggled at runtime. The host would have no way of preventing or even detecting this process. The advantages of such a hybrid approach are obvious: The alias is only activated after a memory hotplug, thus preventing system crashes. Simultaneously, the host's page tables remain consistent, while the aliasing can be selective and time-limited. Fur-

thermore, the attack is independent of the memory technology used, it works equally well with DDR4, DDR5, and CXL.mem. A physical interposer is not required, as the manipulation takes place entirely within the device itself and could even be triggered remotely via firmware. The attack potential of a dynamic aliasing mechanism in the CXL device thus significantly surpasses the capabilities of the *BatteringRAM* attack. While the latter only affects individual command address lines, the internal logic of a compromised CXL device allows the definition of arbitrary remap matrices. This enables the attacker not only to manipulate individual memory areas but also to control the entire device. Attacks could be specifically targeted at certain virtual machines or workloads without the host possessing the necessary telemetry to detect these actions.

7 Conclusion and Outlook

Conclusion This work investigated the security implications of the CXL.mem protocol using a prototypical FPGA-based CXL Type 3 memory device. The aim was to understand how sensitive the internal address decoding of a CXL device is to manipulation and what impact such errors or attacks can have on the integrity and isolation of memory contents. In particular, the focus was on whether the host is capable of detecting malicious mappings in the device memory or whether CXL devices represent a new, previously under-researched anchor of trust. In this work, targeted address aliasing was implemented on the FPGA by masking a single address bit in the AXI to the AVMM path of the CXL.mem design. The result was a clear, controlled aliasing effect: two logically distinct host addresses pointed to the same physical memory area. The functionality of the overall system was initially validated through latency tests and subsequently confirmed through extensive DevDAX-based access tests. The experiments showed that the address model perceived by the host provided no indication of the internal aliasing, and the error was only detectable through content tests within the device itself. The proof-of-concept thus successfully demonstrated that a CXL device can implement internally manipulated address paths without the host detecting any such deviations. This observation is relevant from a security perspective. While classic DRAM subsystems within the CPU are tightly regulated and well understood, CXL offloads a significant portion of the memory architecture to external, vendor-defined devices. This work shows that the host largely blindly trusts the internal implementation of a CXL Type 3 device. Existing security mechanisms in operating systems, hypervisors, or TEEs lack the capability to verify the physical consistency of the address decoding performed by such devices. In environments such as memory pooling or multi-tenant clouds, where multiple hosts or tenants access the same device storage, even a minor error in the address path can lead to data corruption, isolation violations, or unpredictable malfunctions. The results clearly demonstrate that the address integrity of a CXL device is a safety-critical component.

Outlook The experiments conducted show that address aliasing works in principle, but the scope of possible attack scenarios has not yet been fully explored. Future work could be pursued in several directions:

1. **Investigation of interactions with Trusted Execution Environments:** Modern TEEs like Intel TDX, Intel SGX, or AMD SEV-SNP currently only protect system DRAM

[AMD20, CD16], not external CXL memory, some TEEs, like Intel TDX, do not even support CXL yet [Int23a]. A detailed analysis of the extent to which offloaded TEE data (e.g., paging data, offload memory, shared buffers) can be affected by internal device manipulation would be a next step. In particular, dynamic aliasing mechanisms that are only activated at runtime could open up new attack vectors.

2. **Dynamic aliasing and time-dependent manipulation:** While the present proof-of-concept uses static aliasing, it would be technically possible to design a CXL device that can toggle individual and arbitrary address bits at runtime. Such dynamic behavior would be even more difficult to detect from the host's perspective and, depending on the activation time, would no longer visibly affect memory management. Investigating such dynamic scenarios could provide valuable insights into more realistic attack vectors.

3. **Automated security checks during the development and verification phase:** This work suggests that internal errors or design flaws in CXL devices can have security-relevant implications. Integrating formal verification methods or hardware-based address integrity checks into the design pipeline could make future devices more robust. Of particular interest is the question of which verification mechanisms the host would need to reliably verify a device's address consistency.

4. **Extended testing in memory pooling environments:** Since memory pooling is considered a key use case for CXL, a practical extension is worthwhile. Experiments where multiple hosts use the same CXL memory area could demonstrate the concrete effects of errors or manipulations in such scenarios and whether existing fabric mechanisms are sufficiently robust against aliasing effects.

Overall, this work demonstrates that CXL.mem opens up new possibilities but also introduces new classes of attacks and vulnerabilities. The internal address logic of a CXL device is not merely a technical subsystem, but a security-critical part of the entire memory architecture. The proof-of-concept presented here clearly shows that further research is necessary to ensure the security and reliability of future CXL-based systems.

References

[AMD20] AMD. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. https://docs.amd.com/api/khub/documents/~uAtQszeypAVVEk_B91Ojg/content, 2020. Accessed: 2025-12-07.

[Arm25] Arm. An Introduction to AMBA AXI - AXI Protocol Overview. https://developer.arm.com/documentation/102202/0300/AXI-protocol-overview, 2025. Accessed: 2025-12-08.

[CD16] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, page 86, 2016.

[CKA21] Wonkyo Choe, Sang-Hoon Kim, and Jeongseob Ahn. Rethinking remote memory placement on large-memory systems with path diversity. In Haryadi S. Gunawi and Xiaosong Ma, editors, *APSys '21: 12th ACM SIGOPS Asia-Pacific Workshop on Systems, Hong Kong, China, August 24-25, 2021*, pages 63–69. ACM, 2021.

[Com20] Compute Express Link Consortium, Inc. CXL-2.0-Specification. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-2.0-Specification.pdf, 2020. Accessed: 2025-11-12.

[CZC+24] Chen Chen, Xinkui Zhao, Guanjie Cheng, Yueshen Xu, Shuiguang Deng, and Jianwei Yin. Next-gen computing systems with compute express link: a comprehensive survey. *CoRR*, abs/2412.20249, 2024.

[Dat] Sandeep Dattaprasad. The importance of security features in a CXL memory controller to protect mission-critical cloud data. https://www.asteralabs.com/security-features-in-a-cxl-memory-controller/. Accessed: 2025-11-29.

[DMOVVB26] Jesse De Meulemeester, David Oswald, Ingrid Verbauwhede, and Jo Van Bulck. Battering RAM: Low-cost interposer attacks on confidential computing via dynamic memory aliasing. In *47th IEEE Symposium on Security and Privacy (S&P)*, May 2026.

[Ele24] Samsung Electronics. Samsung demonstrates new CXL capabilities and introduces new memory module for scalable, composable disaggregated infrastructure at Memcon 2024. https://semiconductor.samsung.com/news-events/news/samsung-demonstrates-new-cxl-capabilities-and-introduces-new-memory-module-for-scalable-composable-disaggregated-infrastructure-at-memcon-2024/, 2024. Accessed: 2025-11-12.

[HDL+26] Xiuhao Huang, Jinge Ding, Haikun Liu, Zhuohui Duan, Xiaofei Liao, and Hai Jin. An FPGA-based distributed shared memory architecture supporting CXL 2.0+ specification. In Xiaoliang Wang, Xiaohong Jiang, Noel Crespi, and Baoliu Ye, editors, *Network and Parallel Computing*, pages 195–207, Cham, 2026. Springer Nature Switzerland.

[HLYC25] Tao Huang, Yonggui Liang, Shubao Yu, and Kexin Chen. Txcocket: an innovative solution for efficient cross-node data transmission enabled by CXL-based shared memory. *CCF Trans. High Perform. Comput.*, 7(1):29–42, 2025.

[HS24] Reece Hayden and Paul Schell. Opportunities and challenges for compute express link (CXL). `https://computeexpresslink.org/wp-content/uploads/2024/11/CR-CXL-101_FINAL.pdf`, 2024.

[Int09] Intel. Avalon interface specifications. `https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.050/vorlesungen/sose09/lrob/mnl_avalon_spec.pdf`, 2009. Accessed: 2025-12-08.

[Int23a] Intel. Intel TDX Connect Architecture Specification. `https://cdrdv2.intel.com/v1/dl/getContent/773614?explicitVersion=true`, 2023. Accessed: 2025-12-07.

[Int23b] Intel. Intel xeon silver 4514y prozessor. `https://www.intel.de/content/www/de/de/products/sku/237557/intel-xeon-silver-4514y-processor-30m-cache-2-00-ghz/specifications.html`, 2023. Accessed: 2025-12-08.

[Int24] Intel. Intel CXL type 3 memory device software guide revision 1.1. `https://www.intel.com/content/www/us/en/docs/programmable/683082/current/cxl-type3-memory-device-software-guide.html`, 2024. Accessed: 2025-12-08.

[Int25] Intel. Intel TDX Enabling Guide. https://cc-enabling.trustedservices.intel.com/intel-tdx-enabling-guide/01/introduction/, 2025. Accessed: 2025-12-07.

[KDK+14] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 361–372. IEEE Computer Society, 2014.

[Ker21] Kernel Newbies. Linux 5.12. https://kernelnewbies.org/Linux_5.12, 2021. Accessed: 2025-11-30.

[Ker24] Kernel Newbies. Linux 6.9. https://kernelnewbies.org/Linux_6.9, 2024. Accessed: 2025-11-30.

[KGGY20] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 695–711. IEEE, 2020.

[KGJ+25] Miryeong Kwon, Donghyun Gouk, Junhyeok Jang, Jinwoo Baek, Hyunwoo You, Sangyoon Ji, Hongjoo Jung, Junseok Moon, Seungkwan Kang, Seungjun Lee, and Myoungsoo Jung. From block to byte: Transforming PCIe SSDs with CXL memory protocol and instruction annotation. *CoRR*, abs/2506.15613, 2025.

[LHX+24] Jinshu Liu, Hamid Hadian, Hanchen Xu, Daniel S. Berger, and Huaicheng Li. Dissecting CXL memory performance at scale: Analysis, modeling, and optimization. *CoRR*, abs/2409.14317, 2024.

[Lin25a] Linux Kernel Community. Device DAX driver linux kernel documentation. https://docs.kernel.org/next/driver-api/cxl/linux/dax-driver.html, 2025. Accessed: 2025-12-07.

[Lin25b] Linux Kernel Community. Linux kernel documentation: CXL subsystem. https://www.kernel.org/doc/html/v6.14-rc6/driver-api/cxl/index.html, 2025. Accessed: 2025-11-12.

[Man22] Tobias Mann. Just how bad is CXL memory latency? https://www.nextplatform.com/2022/12/05/

just-how-bad-is-cxl-memory-latency/, 2022. Accessed: 2025-11-29.

[MK20] Onur Mutlu and Jeremie S. Kim. Rowhammer: A retrospective. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 39(8):1555–1571, 2020.

[MRG+19] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[MWO+25] Jesse De Meulemeester, Luca Wilke, David F. Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, and Jo Van Bulck. BadRAM: Practical memory aliasing attacks on trusted execution environments. In Marina Blanton, William Enck, and Cristina Nita-Rotaru, editors, *IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12-15, 2025*, pages 4117–4135. IEEE, 2025.

[NAZ+18] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe performance for end host networking. In Sergey Gorinsky and János Tapolcai, editors, *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, pages 327–341. ACM, 2018.

[PCI22] PCI-SIG. PCI express base specification, revision 6.0, 2022.

[PCK21] Jihyun Park, Byoungju Choi, and Yeonhee Kim. Automated memory corruption detection through analysis of static variables and dynamic memory usage. *Electronics*, 10:2127, 09 2021.

[PGM+16] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: exploiting DRAM addressing for cross-cpu attacks. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 565–581. USENIX Association, 2016.

[Red] Red Hat. Chapter 20. using NVDIMM persistent memory storage. https://docs.redhat.com/en/documentation/

red_hat_enterprise_linux/9/html/managing_storage_devices/using-nvdimm-persistent-memory-storage_managing-storage-devices?utm_source=chatgpt.com. Accessed: 2025-12-07.

[SBB24] Debendra Das Sharma, Robert Blankenship, and Daniel S. Berger. An introduction to the compute express link (CXL) interconnect. *ACM Comput. Surv.*, 56(11):290:1–290:37, 2024.

[SMM23a] Samuel W. Stark, A. Theodore Markettos, and Simon W. Moore. How flexible is CXL's memory protection? *Commun. ACM*, 66(12):46–51, 2023.

[SMM23b] Samuel W. Stark, A. Theodore Markettos, and Simon W. Moore. How flexible is CXL's memory protection?: Replacing a sledgehammer with a scalpel. *ACM Queue*, 21(3):54–64, 2023.

[Sup23] Supermicro. X13sei-tf motherboard. https://www.supermicro.com/en/products/motherboard/x13sei-tf, 2023. Accessed: 2025-12-08.

[Tec25] Montage Technology. Montage technology introduces CXL 3.1 memory expander controller to empower next-generation data center infrastructure. https://www.montage-tech.com/Press_Releases/20250901, 2025. Accessed: 2025-11-12.

[Ter] Terasic. Mercury A2700 accelerator card. https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=142&No=1300&PartNo=1#contents. Accessed: 2025-11-30.

[Ter24] Terasic. *Mercury A2700 accelerator card User Manual*, v1.3.2 edition, 08.2024.

[YNK+24] Sungmin Yun, Hwayong Nam, Kwanhee Kyung, Jaehyun Park, Byeongho Kim, Yongsuk Kwon, Eojin Lee, and Jung Ho Ahn. CLAY: CXL-based scalable NDP architecture accelerating embedding layers. In Kenji Kise, Valentina Salapura, Murali Annavaram, and Ana Lucia Varbanescu, editors, *Proceedings of the 38th ACM International Conference on Supercomputing, ICS 2024, Kyoto, Japan, June 4-7, 2024*, pages 338–351. ACM, 2024.

Appendix

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <errno.h>
#include <string.h>

static void die(const char* m){perror(m); exit(1);}

int main(int argc, char** argv){
  if(argc < 3){ fprintf(stderr,"Usage: %s /dev/daxX.Y lengthG [stepB]\n",
      argv[0]); return 2; }
  const char* path = argv[1];
  size_t lenG = strtoull(argv[2], NULL, 0);
  size_t step = (argc>3)? strtoull(argv[3], NULL, 0) : 64; // 64B default
  size_t length = (size_t)lenG << 30;          // total mapping (e.g., 16G)
  size_t half   = length >> 1;                 // 8G if length=16G

  int fd = open(path, O_RDWR | O_SYNC);
  if(fd<0) die("open");
  void* base = mmap(NULL, length, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
  if(base==MAP_FAILED) die("mmap");

  volatile uint64_t* p = (volatile uint64_t*)base;
  size_t words_per_step = step/sizeof(uint64_t);
  if (words_per_step==0) { fprintf(stderr,"step too small\n"); return 2; }

  // Pass 1: fill lower half with A, check upper shows A (alias)
  printf("Pass1: fill LOW with A, verify HIGH == A\n");
  for(size_t off=0; off<half; off+=step){
    volatile uint64_t* q = (volatile uint64_t*)((uint8_t*)base + off);
    q[0] = 0xAAAAAAAAAAAAAAAAull;
  }
  // verify upper
  size_t errs=0;
```

```
39    for(size_t off=0; off<half; off+=step){
40      volatile uint64_t* qh = (volatile uint64_t*)((uint8_t*)base + half + off);
41      if(qh[0] != 0xAAAAAAAAAAAAAAAAull){ if(++errs<10)
            fprintf(stderr,"Mismatch@%zx: %016llx\n", half+off, (unsigned long
            long)qh[0]); }
42    }
43    printf("Pass1 errors: %zu\n", errs);
44
45    // Pass 2: fill HIGH with B, verify LOW == B
46    printf("Pass2: fill HIGH with B, verify LOW == B\n");
47    for(size_t off=0; off<half; off+=step){
48      volatile uint64_t* q = (volatile uint64_t*)((uint8_t*)base + half + off);
49      q[0] = 0x5555555555555555ull;
50    }
51    errs=0;
52    for(size_t off=0; off<half; off+=step){
53      volatile uint64_t* ql = (volatile uint64_t*)((uint8_t*)base + off);
54      if(ql[0] != 0x5555555555555555ull){ if(++errs<10)
            fprintf(stderr,"Mismatch@%zx: %016llx\n", off, (unsigned long
            long)ql[0]); }
55    }
56    printf("Pass2 errors: %zu\n", errs);
57
58    // Optional: full streaming write across the *entire* 16G (just stress;
          alias makes 2nd half redundant)
59    printf("Pass3: full streaming write across all %zu GiB (pattern C)\n",
          lenG);
60    for(size_t off=0; off<length; off+=step){
61      volatile uint64_t* q = (volatile uint64_t*)((uint8_t*)base + off);
62      q[0] = 0xCCCCCCCCCCCCCCCCull;
63    }
64
65    munmap((void*)base, length);
66    close(fd);
67    printf("Done.\n");
68    return 0;
69  }
```

Listing 1: Checks if aliasing occurs in DevDAX mode, but without flushing the cache.

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <errno.h>
#include <string.h>
#include <time.h>

#if defined(__x86_64__) || defined(__i386__)
  #include <emmintrin.h>  // _mm_clflush
  #define HAVE_X86_FLUSH 1
#else
  #define HAVE_X86_FLUSH 0
#endif

static void die(const char* m){ perror(m); exit(1); }
static inline void mem_barrier(void){ __sync_synchronize(); }

static inline void clflush_line(void* p){
#if HAVE_X86_FLUSH
  _mm_clflush((char*)p);      // portable flush+invalidate der Cacheline
#else
  (void)p;
#endif
}

static inline void flush_range(void* base, size_t length, size_t step){
  for(size_t off = 0; off < length; off += step)
    clflush_line((char*)base + off);
  mem_barrier();
}
static inline void write_line_64B(volatile uint64_t* q, uint64_t pattern){
  for (int i=0;i<8;i++) q[i] = pattern;  // 8x8B = 64B voll
}
static void fill_range(void* base, size_t start, size_t len, size_t step,
    uint64_t pattern){
  size_t end = start + len;
  for(size_t off = start; off < end; off += step){
    volatile uint64_t* q = (volatile uint64_t*)((uint8_t*)base + off);
    write_line_64B(q, pattern);
  }
```

```
45 }
46 static size_t verify_range_equals(void* base, size_t start_a, size_t start_b,
       size_t len, size_t step, uint64_t expect){
47   size_t mismatches = 0, printed = 0;
48   size_t end = start_a + len;
49   for(size_t off = 0; start_a + off < end; off += step){
50     volatile const uint64_t* pa = (volatile const uint64_t*)((uint8_t*)base +
         start_a + off);
51     volatile const uint64_t* pb = (volatile const uint64_t*)((uint8_t*)base +
         start_b + off);
52     uint64_t a0 = pa[0], b0 = pb[0];
53     if (a0 != expect || b0 != expect){
54       if (printed < 10){
55         fprintf(stderr, "Mismatch@%zx: A=%016llx B=%016llx (exp=%016llx)\n",
56                 start_b + off, (unsigned long long)a0, (unsigned long
                     long)b0, (unsigned long long)expect);
57         printed++;
58       }
59       mismatches++;
60     }
61   }
62   return mismatches;
63 }
64
65 static inline double now_s(void){
66   struct timespec ts; clock_gettime(CLOCK_MONOTONIC, &ts);
67   return ts.tv_sec + ts.tv_nsec*1e-9;
68 }
69
70 int main(int argc, char** argv){
71   if(argc < 3){
72     fprintf(stderr,"Usage: %s /dev/daxX.Y lengthG [stepB]\n", argv[0]);
73     return 2;
74   }
75   const char* path = argv[1];
76   size_t lenG = strtoull(argv[2], NULL, 0);
77   size_t step = (argc>3)? strtoull(argv[3], NULL, 0) : 64;
78   if (step < 64) step = 64;
79   if (step % 64) step = ((step + 63)/64)*64;
80
81   size_t length = (size_t)lenG << 30;
82   size_t half   = length >> 1;
83
84   int fd = open(path, O_RDWR | O_SYNC);
85   if(fd < 0) die("open");
86   void* base = mmap(NULL, length, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
```

```
87    if(base == MAP_FAILED) die("mmap");
88
89    // PASS 1: LOW = A, verify HIGH == A
90    puts("Pass1: fill LOW with A, verify HIGH == A");
91    double t0 = now_s();
92    fill_range(base, 0, half, step, 0xAAAAAAAAAAAAAAAAull);
93    mem_barrier();
94    // Flush geschriebenen Bereich (LOW) und den zu lesenden Bereich (HIGH)
95    flush_range(base, half, step);                    // LOW
96    flush_range((uint8_t*)base + half, half, step);   // HIGH (safety)
97    // msync() auf /dev/dax ist meist EINVAL -> ignorieren
98    size_t errs = verify_range_equals(base, 0, half, half, step,
          0xAAAAAAAAAAAAAAAAull);
99    printf("Pass1 errors: %zu (%.3fs)\n", errs, now_s()-t0);
100   // PASS 2: HIGH = B, verify LOW == B
101   puts("Pass2: fill HIGH with B, verify LOW == B");
102   t0 = now_s();
103   fill_range(base, half, half, step, 0x5555555555555555ull);
104   mem_barrier();
105   // Flush geschriebenen Bereich (HIGH) und den zu lesenden Bereich (LOW)
106   flush_range((uint8_t*)base + half, half, step);   // HIGH
107   flush_range(base, half, step);                    // LOW (INVALIDIERE alte
          A-Lines!)
108   errs = verify_range_equals(base, half, 0, half, step,
          0x5555555555555555ull);
109   printf("Pass2 errors: %zu (%.3fs)\n", errs, now_s()-t0);
110   // PASS 3: full streaming write across entire range (pattern C)
111   printf("Pass3: full streaming write across all %zu GiB (pattern C)\n",
          lenG);
112   t0 = now_s();
113   for(size_t off=0; off<length; off+=step){
114     volatile uint64_t* q = (volatile uint64_t*)((uint8_t*)base + off);
115     write_line_64B(q, 0xCCCCCCCCCCCCCCCCull);
116   }
117   mem_barrier();
118   flush_range(base, length, step);
119   printf("Done. (%.3fs)\n", now_s()-t0);
120
121   munmap(base, length);
122   close(fd);
123   return 0;
124 }
```

Listing 2: Checks if aliasing occurs in DevDAX mode, with flushing the cache