Technical Report

UCAM-CL-TR-1011
ISSN 1476-2986

Number 1011



UNIVERSITY OF CAMBRIDGE

Computer Laboratory

IOCaps: de/centralized capabilities for I/O protection

Samuel Winslow Stark

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

© Samuel Winslow Stark

This technical report is based on a dissertation submitted June 2026 by the author for the degree of Doctor of Philosophy to the University of Cambridge, St. Catharine's College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-1011*

IOCaps: De/centralized Capabilities for I/O Protection

Samuel Winslow Stark

Abstract

Modern computing systems interact with a minefield of potentially buggy or malicious peripherals such as block storage, network devices, fixed-function video decoders and NPUs, and fully programmable GPUs and FPGAs. These peripherals must be allowed to perform direct memory access (DMA) on host memory, for high-speed bulk memory transfer, but must not be allowed to access memory not meant for them. IOMMUs can thwart DMA attacks via address translation but are limited in spatial security, temporal security, and performance. IOMMUs by definition are limited to page-granularity protection. IOMMUs have to perform slow walks over page tables (stored in slow DRAM) on the critical path, limiting throughput and increasing latency. IOTLBs can cache page table walks but I/O tends to have low locality, constantly accessing new pages and unmapping old ones, making caching ineffective. Invalidating IOTLB entries is so slow that Linux defers invalidation, leading drivers to accidentally access and reuse memory while it may still be exposed, to avoid an otherwise debilitating performance hit.

This thesis proposes IOCaps: a capability-based system inspired by CHERI that replaces slow page table walks with fast cryptographic validation, allows peer-to-peer capability delegation and reduction, and improves protection granularity and invalidation latency. Instead of defining access control policies with a cache backed by DRAM page tables, IOCaps are access tokens that are stored by the peripherals, attached to each access, and validated by the host against a set of keys stored in fast SRAM. A Macaroon-inspired Message Authentication Code (MAC)-chaining system allows IOCaps to be reduced and passed between peripherals, facilitating CHERI-like compartmentalization, while remaining secure. IOCaps, like CHERI capabilities, encode the bounds of accessible regions at sub-page precision, avoiding accidental exposure of sensitive data near exposed regions. While IOCaps do not enable translation they can still be layered on top of IOMMU translation if needed, for example, for virtualization. Using separate systems for protection and translation should allow translation to be made less granular, using high-locality hugepages to be more IOMMU-friendly, without sacrificing protection granularity—the best of both worlds.

After justifying and describing the basic system, this thesis investigates spatial security, temporal security, and performance to show IOCaps are a viable alternative to IOMMU protection. **Spatial Security.** IOCaps must be compact to avoid bottlenecking DMA throughput, but must still represent a 64-bit address region with high precision and multiple reductions, which I show is viable through a new approach to compression. **Temporal Security.** Because protection is no longer backed by a persistent page table, IOCap lifetimes are entirely dependent on that of the keys used to validate them, requiring more keys per peripheral than prior art in the same vein. I investigate different MAC functions, fundamental limits on the number of keys per system, and allocation methods that allow IOCaps to scale to underprovisioned systems without loss of security. **Performance.** I implement an IOCap protection system in practice using QEMU-based emulation and real FPGA hardware, integrate it with existing CheriBSD VirtIO drivers, and demonstrate that the impact on bulk DMA throughput is minimal.

Acknowledgements

First and foremost, I would like to thank my supervisor Prof. Simon W. Moore for his supervision, guidance, and superhuman ability to absorb the large quantities of text I emit intermittently. Thanks also to Dr. A. Theodore Markettos, who has been a significant help both as a guide through the unknown territory of I/O protection, and as a co-author of articles and papers.

It has been an absolute pleasure to be a member of the CHERI group, led by Prof. Robert N. M. Watson, and to collaborate with so many talented researchers, including my PhD bestie Dr. Franz A. Fuchs, Dr. Peter Rugg, Dr. Alexandre Joannou, Dr. Jonathan Woodruff, Yuecheng Wang, Konrad Witaszczyk, Jessica Clarke, Alfredo Mazzinghi, Dr. Paul Metzger, Dr. Hesham Almatary, and Tom Bytheway. Thank you all for your company, assistance, and advice, without which this thesis would certainly not have been completed.

I would not have started this project in the first place without my scholarship from the David and Claudia Harding Foundation. I've gained so much, academically and socially, from the community the scholarship has cultivated. A PhD is so often a narrowing of scope, burrowing deep into a niche, and to constantly be exposed to new groundbreaking research from all manner of fields is always a breath of fresh air. Not only that, but the opportunities I've received to present my work have developed my skills and snowballed into other opportunities, for which I am exceptionally grateful.

The path of a PhD is not always easy, and my family have been a rock for me through good times and bad. I would like to thank my father Gavin for our hours of technical discussion, both on and off topic; my mother Emily for her writing advice and emotional support; and my sister Freyja for all of our good times and distractions. I love you all!

That, too, goes for the many friends I have accumulated over my journey. I am not always a social butterfly, and I am forever indebted to those who bring that side out: Kit, the quizzers, and Mark; Harry, Jason, and the entirely official Cambridge University Port Society; Sam R., Tanja, Filippo and all my fellow St. Catz MCR committee members; my cohort compatriots Frankie, Dan, Lisa-Maria, Petur, and Shadi; and the Clareheads—Yasmeen, Nils, Jas, Zhenglin, Hugo, Sam H., Erin, and Tim—from an M.Phil long past[1]. You all mean so much, and the time I spend with you is always precious.

[1]Hey, four years really can feel like a long time!

Contents

Glossary . . . 16

1 Introduction **21**

1.1 Contributions . . . 22
1.2 Hypotheses and overview . . . 24
1.3 Publications and collaboration . . . 28
1.4 Artefacts . . . 28

2 Background **29**

2.1 I/O model . . . 29
2.2 Threat model . . . 30
2.2.1 Assumptions . . . 31
2.3 Interconnect mechanics and protection . . . 32
2.3.1 PCIe . . . 32
2.3.1.1 DMA transactions . . . 33
2.3.1.2 Protection . . . 34
2.3.2 AXI . . . 35
2.3.2.1 DMA transactions . . . 35
2.3.2.2 Protection . . . 37
2.4 Simple protection tables . . . 37
2.4.1 Arm MPU — Small permission tables . . . 37
2.4.2 RISC-V IOPMP — Two-level small permission tables . . . 38
2.4.3 CXL GFAM — Small translation tables . . . 38
2.5 Whole-processor-integrated permission tables . . . 40
2.5.1 Arm TrustZone . . . 40
2.5.2 Armv9 Realms . . . 42
2.5.3 RISC-V WorldGuard . . . 42
2.6 IOMMUs — Translation as protection . . . 43
2.6.1 Basic design . . . 43
2.6.2 PCIe's IOMMU standards . . . 44
2.6.2.1 PASIDs and ATS — Basic translation . . . 44
2.6.2.2 SR-IOV, SIOV — {Single-root, Scalable} I/O virtualization . . . 45
2.6.2.3 PRI — Page requests . . . 46
2.6.3 Arm's IOMMU standards . . . 47
2.7 IOMMUs in practice . . . 49
2.7.1 Adoption . . . 49
2.7.2 Problems identified . . . 50
2.7.3 Mitigation attempts . . . 53
2.7.4 Conclusion . . . 54

3 Capabilities & the IOCap system **55**
3.1 Capability systems . . . 55
3.1.1 Chained MAC capabilities . . . 57
3.2 Scope . . . 58
3.2.1 Goal . . . 59
3.2.2 Non-goals . . . 59
3.2.3 Requirements . . . 60
3.2.4 Threat model . . . 61
3.3 Prior art . . . 61
3.3.1 Snapdragon (2003) . . . 62
3.3.2 CryptoMMU . . . 63
3.3.3 CHERI . . . 64
3.3.4 Others . . . 64
3.4 My proposed IOCap system . . . 65
3.4.1 Grouping IOCaps under leases . . . 65
3.4.2 The abstract IOCap format . . . 67
3.4.3 Creating IOCaps . . . 68
3.4.4 Reducing and delegating IOCaps . . . 68
3.4.5 Checking IOCaps . . . 71
3.4.6 Revoking IOCaps . . . 72
3.5 Summary . . . 72

4 Temporal safety with leases **75**
4.1 Why multiple leases? . . . 75
4.2 How many leases? . . . 76
4.2.1 Sharing persistent regions . . . 76
4.2.2 Sharing ephemeral regions . . . 77
4.3 The MAC function and its impact . . . 77
4.3.1 MAC collision attacks . . . 80
4.3.2 Replay attacks . . . 80
4.3.3 Key recovery and forgery . . . 81
4.3.4 Lease lifetime, or 'cryptoperiod' . . . 82
4.4 Practical limits of a flat lease table . . . 82
4.5 Handling underprovisioned systems . . . 83
4.5.1 Simple lease ID allocation . . . 84
4.5.1.1 Persistent regions via epochs . . . 84
4.5.1.2 Ephemeral FIFO regions . . . 85
4.5.2 Complex lease ID allocation . . . 86
4.5.2.1 Priority queues . . . 86
4.5.2.2 Unpredictable ops . . . 86
4.5.2.3 Out-of-order ops . . . 86

4.5.2.4 Memory pooling . . . 87
4.6 Evaluation . . . 87
4.6.1 A FIFO queue . . . 88
4.6.2 An out-of-order queue . . . 90
4.6.2.1 Impact on submission availability . . . 92
4.6.2.2 Reads vs. writes and other priorities . . . 92
4.6.2.3 Alternate approach — 'Arbitration Burst' . . . 93
4.6.3 Multiple out-of-order queues . . . 93
4.6.3.1 Alternate approach — Share leases between queues . . . 95
4.6.4 Applications in practice . . . 96
4.7 Summary . . . 98

5 Spatial safety with compressed IOCap formats 101
5.1 Goals . . . 101
5.1.1 Precision . . . 102
5.1.2 Reduction . . . 102
5.1.3 Permissions . . . 103
5.1.4 Lease ID space . . . 103
5.1.5 Summary . . . 104
5.2 Prior art . . . 104
5.2.1 Direct encoding . . . 104
5.2.2 Compressed capabilities . . . 106
5.3 IOCap compression . . . 108
5.3.1 Initial region . . . 109
5.3.2 Caveats, chained . . . 113
5.3.3 Index caveats — Bulk-to-bulk . . . 113
5.3.4 Range caveats — Bulk-to-precise . . . 115
5.3.5 An "optimal" complete format . . . 117
5.3.6 The IOCap-VirtIO complete format . . . 117
5.4 Implementation . . . 120
5.4.1 Hardware decoding . . . 121
5.4.2 Software encoding . . . 121
5.5 Evaluation . . . 125
5.5.1 Initial region . . . 125
5.5.2 Caveat reduction . . . 126
5.5.3 Caveat precision . . . 128
5.5.4 Improving the index caveat . . . 130
5.6 Summary . . . 132
5.6.1 Future improvements . . . 134

6 IOCaps in software **135**
6.1 IOCaps in QEMU . . . 135
6.1.1 IOCap-VirtIO in QEMU . . . 137
6.1.2 IOCap-VirtIO in CheriFreeRTOS . . . 137
6.1.3 Secure randomness sources . . . 138
6.2 IOCaps in CheriBSD . . . 139
6.2.1 The CheriBSD DMA API . . . 139
6.2.1.1 `bus_dmamem` . . . 140
6.2.1.2 Invalidation and the IOMMU . . . 142
6.2.2 An IOCap DMA API . . . 142
6.2.2.1 Minting and revocation . . . 144
6.2.2.2 `bus_dmamem` . . . 144
6.2.3 IOCap-VirtIO in CheriBSD . . . 145
6.2.3.1 `virtqueue-iocap` . . . 146
6.2.3.2 `virtio-blk-iocap` . . . 146
6.2.4 Functionality . . . 147
6.3 Observations on implementing IOCaps . . . 147
6.3.1 How CHERI can help . . . 149
6.3.2 Implementation impact . . . 150
6.4 Summary . . . 151

7 IOCaps in hardware **153**
7.1 IOCaps on the interconnect . . . 153
7.1.1 Serial protocols . . . 154
7.2 Building a hardware IOCap interposer . . . 155
7.2.1 Lease manager . . . 155
7.2.2 Checker pool . . . 157
7.2.3 Transaction scoreboard . . . 158
7.3 Verification . . . 159
7.4 Microbenchmarks . . . 161
7.4.1 Area and frequency . . . 161
7.4.2 DMA throughput and latency . . . 162
7.4.3 Lease operation throughput and latency . . . 162
7.5 Full-system integration . . . 164
7.6 Summary . . . 168
7.6.1 Future work . . . 169

8 Conclusion **171**
8.1 Related work . . . 172
8.2 IOCaps vs. IOMMUs . . . 173
8.3 Future work . . . 174

Bibliography **177**

List of Figures

1.1 An overview of the practical contributions of this thesis. 23

2.1 Abstract threat model. 31
2.2 PCIe's simplified protection model, where the root complex (RC) exposes PCIe peripherals to cache-coherent host memory. 33
2.3 ACS P2P Request/Completion Redirect, explained. 35
2.4 AXI flow diagrams, with channels marked. 36
2.5 Memory sharing with coherence in CXL. 39
2.6 Sharing GFAM within an HPA. 40
2.7 An example of a TrustZone-aware system architecture, where different peripherals are in different worlds [79, Figure 4-1]. 41
2.8 The SIOV Virtual Device and ADI model [93, Figure 7-1]. 46
2.9 The i.MX 8QuadMax protection model, which uses a distributed SMMU to expose SoC subsystems to incoherent memory and other subsystems. 48
2.10 Memory lifecycle attack: host accesses while exposed. 52
2.11 Memory lifecycle attack: host frees while exposed. 52

3.1 The chained MAC scheme used by the 2003 network-attached disk system [160]. 62
3.2 The MAC scheme employed by CryptoMMU [45]. 63
3.3 A block diagram of the IOCap architecture. 66
3.4 The components (C, M) of a two-caveat IOCap, including the core fields in C. . 67
3.5 Minting an IOCap `(C0, M0)` in Python-esque pseudocode. 69
3.6 Applying caveat 1, then caveat 2, in Python-esque pseudocode. 70
3.7 Verifying an attacker-provided `(C2*, M2*)` using the lease key in Python-esque pseudocode. 71

4.1 Epoch migration examples. 85
4.2 Ops in an $L_{\text{ops}} = 4$ lease group enqueued in an in-order queue. 89
4.3 Overexposure distributions for in-order queue models where $t_{\text{avg}} = 150\mu s$, targeting 100,000 IOPS. 89
4.4 A demonstration of how ops with completion latencies distributed throughout a range can lead to out-of-order completion—even if the ops began execution in-order. 90
4.5 An explanation of the worst case overexposure over_{max} for a single in-order begin out-of-order complete queue. 91
4.6 A single-queue out-of-order model targeting 800,000 IOPS under different queue depths. 91
4.7 An explanation of the worst case overexposure over_{max} for multiple in-order begin out-of-order complete queues. 94

4.8 The out-of-order model from Fig. 4.6 split across $N_Q = 8$ with $Q_{\text{ops}} = 16$. 94
4.9 A demonstration of merging queues under the same set of leases to reduce overexposure. 95
4.10 Comparing overexposure between a fully merged lease ID group and the per-queue scenario from Fig. 4.8. 95
4.11 Samsung PM1733 op latency at 1.25M IOPS. 97
4.12 Overexposure distributions for a 63-thread Samsung PM1733 modelled at 1.25M IOPS and $Q_{\text{ops}} = 10$. 97

5.1 The M-Machine [246, 247] capability format. 106
5.2 The Aries [248] capability format. 107
5.3 The Low-Fat [250] capability format. 108
5.4 The CHERI ISAv9 [216] variant of the CHERI Concentrate [240] capability format. 109
5.5 The structure of the index caveat. 114
5.6 The structure of the range caveat. 115
5.7 The "optimal" IOCap format. 117
5.8 The VirtIO IOCap format. 118
5.9 Lookup tables for encoding and decoding initial regions, spanning three 64-byte cache lines in total, in Rust-esque pseudocode. 121
5.10 Encoding the initial region using look-up tables. 123
5.11 Encoding an index caveat on top of the initial region. 124
5.12 Encoding a range caveat on top of the prior region. 124
5.13 The base and length precision for the 63-bit "optimal" initial region, characterized as 64-12-8-3-2-8 in Table 5.5. 126
5.14 The base and length precision for the 54-bit IOCap-VirtIO initial region, characterized as 54-10-5-5-2-1 in Table 5.6. 126
5.15 The amount of reduction achievable using one or both caveats in the "optimal" configuration. 127
5.16 The amount of reduction achievable using one or both caveats in the IOCap-VirtIO configuration. 127
5.17 The worst-case base and length precision available with caveat(s) enabled in the "optimal" IOCap configuration. 129
5.18 The worst-case base and length precision available with both caveats enabled in the "optimal" IOCap configuration, as compared to the CHERI ISAv9 [216] format. 129
5.19 The amount of reduction achievable using one or both caveats in the "optimal" configuration with improved index caveat. 131
5.20 The worst-case base and length precision available with an improved index caveat. 131

6.1 The lease manager MMIO space as designed for 256 leases. 136

6.2 CheriFreeRTOS, running under QEMU, serving a static web page over TCP using IOCap-VirtIO. . . . 138
6.3 DMA mapping structures and their relationships in standard CheriBSD. . . . 141
6.4 DMA mapping structures and their relationships in IOCap-enabled CheriBSD. 143
6.5 Sample console output from CheriBSD, plus a QEMU log message, after booting on QEMU. . . . 148

7.1 IOCap-AXI flow diagrams, with channels marked. . . . 155
7.2 The IOCap enforcer and lease manager modules. . . . 156
7.3 The three states of each lease and their host-facing integer encodings. . . . 156
7.4 A pipeline diagram demonstrating how a checker handles transactions with different numbers of caveats. . . . 158
7.5 Enforcer throughput against pool size for 4-data-flit DMA read and write transactions. . . . 162
7.6 The general structure of the CHERI-BGAS SoC, showing the flow of data transfer starting from the host CPU initiating an op to the peripheral CPU accessing relevant DRAM. . . . 164
7.7 Sample CheriBSD boot log, including messages from the `iocap_keymngr` and the IOCap-VirtIO block device. . . . 166
7.8 Sample console output running `fio` on CheriBSD on the full system. . . . 167

List of Tables

2.1 Common memory access patterns. . . . 30
2.2 The ideal lifecycle of host memory and the practical reality of the IOMMU and other systems. . . . 51

4.1 The number of leases required in different scenarios, and the amount of storage they require, as compared to a single AMD Zen 5 core [201]. . . . 83
4.2 $over_{basic}$ calculations for Fig. 4.3a's model. . . . 89
4.3 $over_{basic}$ and $over_{max}$ for the 128-op out-of-order model in Fig. 4.6b. . . . 92
4.4 $over_{basic}$ and $over_{max}$ for the multiple-queue out-of-order model from Fig. 4.8. 94
4.5 $over_{basic}$ and $over_{max}$ for the Samsung PM1733 in Fig. 4.12. . . . 97
4.6 Comparing the maximum total lease counts between per-queue and merged lease ID groups. . . . 98

5.1 Details of a hypothetical 32-bit $n = 2$, $w = 3$ initial region encoding, fully characterized as 32-0-8-3-1-1. . . . 110
5.2 The alignment constraints from Section 5.1.5 expressed mathematically, with the minimum number of bits required to encode regions under each one, assuming $A = 64$. . . . 112
5.3 An exhaustive search across w and $\log_2 n$, indicating E_{Step} when valid values that respect constraints were found. . . . 112
5.4 The 15 possible values of the permissions chain. . . . 113
5.5 Details of the 64-12-8-3-2-8 format. . . . 117
5.6 Details of the 54-10-5-5-2-1 initial region format, which notably does not meet the 2MiB→4KiB constraint. . . . 119
5.7 Details of the 48-12-8-3-2-5 format. . . . 133
5.8 Details of the 32-0-11-2-1-10 format. . . . 133

6.1 How CheriBSD normally maps to the practical lifecycle of host memory described in Table 2.2, without IOCaps. . . . 140
6.2 How CheriBSD maps to the practical lifecycle of host memory described in Table 2.2 when using IOCaps. . . . 145
6.3 Lines-of-Code (LoC) changes in adapting CheriBSD to IOCaps. . . . 150

7.1 The test permutations used to verify enforcer and lease manager behaviour. . 160
7.2 Synthesis results for isolated checker units and cycle latencies for enforcers using those checkers under different configurations. . . . 161
7.3 Enforcer synthesis results against pool size. . . . 162

7.4 Latency in interposer cycles between receiving an MMIO lease close request, and the lease entering ‘invalid’, when saturated with 32x DMAs of varying lengths using that lease. 163
7.5 MMIO latency per open and close under different workloads in interposer cycles. 163

Glossary

Address space A mapping of numeric addresses to memory or registers that can be targeted with read or write accesses.

AXI, Advanced eXtensible Interface A point-to-point connection standard that specifies how a 'Manager' makes requests of a connected 'Subordinate'.

BRAM, Block RAM A read/writable memory available on FPGA, typically with two ports which can read or write in a given cycle. Roughly, for the purposes of this thesis, the FPGA's proxy for SRAM.

Capability An unforgeable token that grants actors in a system access to some resource. There are various examples which each use different definitions of tokens, actors, and resources. CHERI uses tagged pointers as tokens to grant software access to memory, POSIX operating systems use file descriptors to grant applications access to files and other devices, and IOCaps use MAC-validated tokens to grant peripherals access to memory.

Checker One of many hardware units inside an IOCap enforcer that verifies the MACs of incoming IOCaps using a lease key.

CHERI A capability system implemented in hardware to protect software's accesses to memory, which can be adapted to many hardware architectures [1]. All pointers are represented as 128-bit capabilities when stored in registers or memory, and the hardware ensures these capabilities are not synthesized out of thin air. In contrast to C-lists, CHERI is an example of a ticket based system, where capabilities are handled directly instead of being indirected through a single centralized list.

C-list, Capability list A list of capabilities which actors may use. Capability systems may require actors to index directly into a C-list to use capabilities [2, 3] or use them as ambient authority [4, 5].

CXL, Compute Express Link An interconnect standard built on PCIe that adds cache coherency, both for peripherals accessing host memory and hosts accessing peripheral memory. Most commonly used for increasing the amount of DRAM available to a system through dedicated memory expansion peripherals.

DMA, Direct Memory Access A bulk memory access initiated on an interconnect, copying data from one (usually contiguous) address range to another (usually contiguous) address range in the interconnect's address space.

DRAM, Dynamic RAM Read/writable memory implemented using capacitors, which are denser and slower to access than other memory technologies. Available in taped-out silicon and on FPGA.

Enforcer A hardware unit in the IOCap system that takes incoming requests from upstream with attached IOCaps, uses one of many checker units to check if each request is valid, and forwards valid requests downstream with the IOCaps stripped off.

Flit, Flow control unit The smallest atom of data handled by a network for flow control purposes. In AXI, an individual piece of data sent on a channel in a single cycle.

FPGA, Field Programmable Gate Array Hardware that can be dynamically reconfigured with new designs to test hardware circuits in practice. Typically multiple orders of magnitude slower than taped-out silicon but otherwise representative of the challenges of hardware development.

Host An endpoint in an interconnect that performs general-purpose computation and controls peripherals to perform ops.

Interconnect A network that connects endpoints, e.g. a host and many peripherals, allowing them to access resources through one or more address spaces.

IOCap, I/O Capability A kind of capability, defined in this thesis, that grants access to a range inside an interconnect's address space. Consists of a capability text encoding the accessible range, and a cryptographic message authentication code (MAC) computed with a lease key.

IOCap-AXI Variant of AXI defined in Chapter 7 that carries IOCaps as metadata over the address channels.

IOCap-VirtIO Variant of VirtIO as defined in Chapter 5 that replaces op descriptors with IOCaps, carrying extra metadata in unused bits of the lease ID field.

IOMMU, I/O Memory Management Unit A hardware unit which protects host resources, usually memory or registers, from peripherals. It does this by walking a page table stored in DRAM, translating requested addresses into the host-physical space, and blocking accesses without valid translations—using translation as a proxy for protection.

IOPS, I/O Operations Per Second The number of I/O ops performed in a second. A measure of peripheral performance, particularly for NVMe storage drives.

IOTLB, I/O Translation Lookaside Buffer A cache for IOMMU translations that helps avoid the cost of repeated page table walks—as long as requested addresses have spatial locality.

IOTLBI, IOTLB Invalidation A message sent to the IOTLB to clear translations pertaining to specific address space regions from the cache.

Lease A slot in an IOCap lease manager, with an ID equal to the numeric index of the slot, which contains a valid cryptographic key for minting and checking IOCaps. A lease is 'opened' when the host selects an unoccupied slot and fills it with a random key, IOCaps are minted under it when their MAC is computed from its key, and it is 'closed' by clearing the key from that slot as directed by the host.

Lease Manager A hardware unit in the IOCap system that manages the active leases and informs the enforcer when they are cleared.

$L_{\mathbf{ops}}$**, Ops-per-lease** When using the ephemeral op lease management mechanism (defined in Section 4.5.1.2, p85), the maximum number of ops a single lease can expose.

MAC, Message Authentication Code "A bit string of fixed length, computed by a MAC generation algorithm, that is used to establish the authenticity and, hence, the integrity of a message" [6]. For the purposes of this thesis, a value that proves a given message was approved by an actor who knew the relevant cryptographic key.

Op A logical I/O operation which usually involves at least one DMA operation, e.g. transferring a block of data from a storage peripheral, or reading a packet of data from the host and transmitting it on a network.

P2P, Peer-to-peer Peer-to-peer DMA accesses. In PCIe and other heterogeneous systems, used to refer to direct peripheral-to-peripheral communication that does not get routed through the host for protection or validation.

PCIe, Peripheral Component Interconnect (Express) An interconnect standard that defines a physical link-to-link connector standard and a system-wide topology.

PASID, Process Address Space ID In PCIe, a prefix attached to a TLP to indicate which address space the receiver should interpret the request in.

Root Complex The root of the PCIe tree topology, usually a part of the host which exposes host memory.

TLP, Transaction Layer Packet A packet sent over a PCIe interconnect to request or complete a memory transaction.

Peripheral An endpoint in an interconnect that performs ops at the request of a host.

SoC, System-on-a-Chip A single chip that contains enough components to function as a self-contained system, including many peripherals. Mobile phones and, more recently, Apple M-series systems are common examples.

Spatial safety The ability to protect access to precise regions in an address space. For example, IOMMUs typically operate on 4KiB-sized regions, and a 512B region cannot be exposed without exposing 4KiB around it. A system which allows 512B, or even smaller granularities, to be exposed directly would have superior spatial safety properties.

SRAM, Static RAM Read/writable memory implemented using transistors, which are less dense and faster than DRAM. Available in taped-out silicon and in limited fashion on FPGA.

Temporal safety The ability to quickly adapt protection when the requested policy changes. For example, CHERI temporal safety concerns use-after-free, where memory must become inaccessible once the application frees it.

VirtIO A generic protocol used by hosts to supply peripherals with ops and receive completion messages. Supports many different kinds of peripherals. Typically used by virtual machines to interact with emulated or virtualized peripherals.

CHAPTER 1

Introduction

Modern computing systems contain an incredible number of peripheral devices to supplement the host CPUs. These can range from high-throughput block storage, to fixed-function accelerators for e.g. matrix math or video decoding, and ones so complex they form systems in their own right such as FPGAs, GPGPUs, NPUs, and smart NICs. They may be internally wired into a system-on-a-chip (SoC) over an interconnect like AXI, or externally connected through physical ports like PCIe or Thunderbolt, sharing data via direct memory access (DMA). Peripherals use DMA to access memory from the host that (ideally) contains data intended for them—but peripherals can't be trusted. They may run buggy firmware which can be exploited to attack the host [7] or even themselves [8]. Thunderbolt allows users to attach new DMA-capable peripherals to a system at any time, which may have malicious hardware [9]. Depending on the supply chain, even on-chip peripherals may be hardware trojans [10]. A single bad peripheral can compromise the whole system in a way that is not mediated by software, so cannot be stopped by software. Hardware protection is necessary to be safe in the face of these threats.

Modern systems rely on trusted hardware interposers between the peripheral and host to validate DMAs, ensuring peripherals only touch memory the host has consciously *exposed* to them. Usually they manifest as a hardware unit which checks requested accesses against a list, accepting those which are valid, and rejecting those not found. The Arm R- and M-profile MPUs [11, 12] and RISC-V IOPMP [13] use small, single-level tables of accessible *regions* that are not intended to change frequently [13, p17]. AMD's IOMMU [14], Intel's VT-d [15], and Arm's SMMU [16]—collectively referred to as I/O Memory Management Units (IOMMUs)—use multi-level page tables stored in DRAM to expose many more regions concurrently.

DRAM is necessary to store large lists, but it's much slower to access. IOMMUs have to perform page table walks on the critical path of all DMAs (just like MMUs), so they cache the page table entries in an I/O Translation Lookaside Buffer (IOTLB, just like the TLB). IOTLB locality is critical, as misses dramatically increase latency to the point of degrading throughput, but peripheral DMAs have low locality unless managed carefully [17, 18, 19]. Caching begets cache invalidation, creating a window of opportunity between the host asking for memory to be unexposed and it actually happening—made worse by the sheer latency of those operations,

and the tendency of modern operating systems (OSs) to *defer* invalidations to mitigate that latency [20, 21]. Inside this window, host OSs can inadvertently access memory while it is still exposed to peripherals, risking information disclosure and hijack [22].

Capability systems present an alternative to the list-based approach. Instead of keeping a centralized list of valid accesses inside the interposer(s), the host hands out unforgeable 'tickets' (or *capabilities*) to peripherals [23]. Peripherals are responsible for storing capabilities and must provide a valid capability with each access—ensuring they are used intentionally—and all the interposer has to do is check validity. Network and block storage peripherals (among others) already operate on queues of descriptors [24, 25, 26], where descriptors indicate a region of memory holding data to work on, and these peripherals could naturally be extended to use capabilities instead of descriptors.

CHERI is a capability system that relies on trusted host hardware to store and manipulate capabilities correctly, to great effect [1, 27, 28]. Prior work has extended CHERI from host to I/O protection, but either requires peripheral hardware to be trusted [29] or uses CHERI to construct small single-level protection tables [30]—untrusted peripherals cannot safely handle CHERI capabilities directly. Authenticated capabilities, such as Macaroons [31], use message authentication codes (MACs) to prove validity and can be stored, manipulated, and delegated by untrusted peripherals [32]. This opens up the possibility of delegation between untrusted peripherals. Delegation has been a boon for CHERI, allowing for example in-process compartmentalization [33, 34], and I believe it may prove useful for accelerators. MAC-based capabilities can also be revoked at-will in groups by throwing away the MAC key—quickly and reliably closing the window of opportunity for malicious DMA.

In this thesis I propose a method of securing memory accesses from external peripherals based on a novel I/O capability system named IOCaps. IOCaps are parameterizable and generalize to many different interconnects, MAC functions, and capability formats. They are superior to IOMMU-based protection, as they allow fine-grained and deterministic grouped revocation—on top of low-latency access control with 100% throughput—at the expense of extra per-transaction metadata and a lack of virtual addressing. They are practical to adopt, as I demonstrate by building a full prototype on the CHERI-BGAS FPGA system-on-a-chip, including hardware and software support. IOCaps do not rely on CHERI, but they can work in tandem for full-system memory protection.

1.1 Contributions

The contributions of this thesis, summarized in Fig. 1.1, are as follows.

- The parameterizable abstract model for the IOCap system, including the lease management system for grouping and revocation;
- An examination of CBC-AES-128 and other algorithms as IOCap MAC functions;
- A study and simulation of different mechanisms of lease allocation for various workloads;

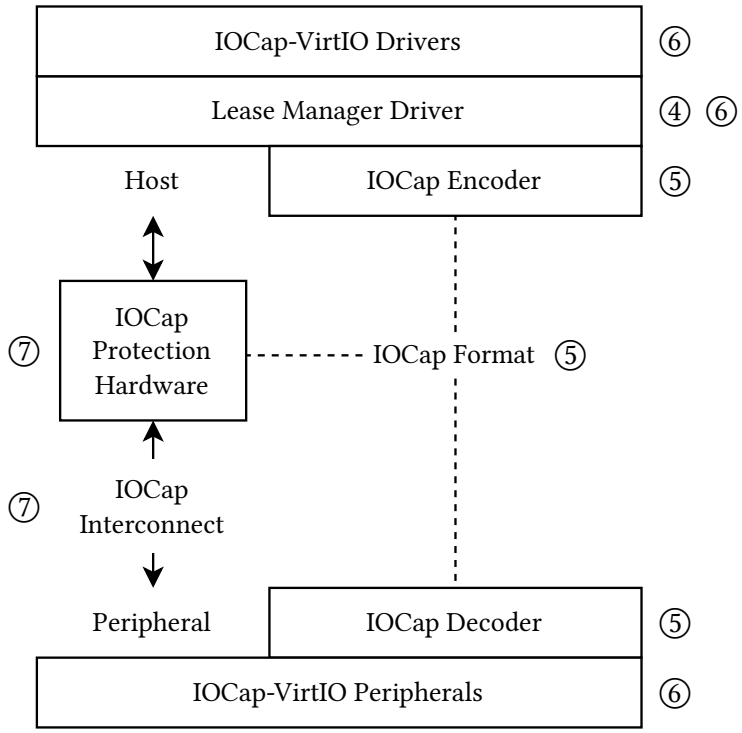


Figure 1.1: An overview of the practical contributions of this thesis. Relevant chapters are indicated with circled numbers—④ for Chapter 4, etc.

- Novel techniques for packing an accessible memory region, reductions of that region, and permissions into compressed IOCap formats for storage and transmission;
- IOCap-VirtIO, an adaptation of the ubiquitous VirtIO driver framework to replace buffer descriptors with IOCaps;
- An implementation of an IOCap system in the QEMU full-system simulator for software testing and bringup, including a simulated IOCap protection unit and IOCap-VirtIO compatible peripherals;
- Host drivers for the IOCap protection unit and IOCap-VirtIO peripherals for multiple operating systems;
- An implementation of a high-throughput IOCap hardware protection unit, including a mechanism to transmit IOCaps as metadata over an AXI interconnect with minimal latency and area overheads;
- A proof-of-concept full-stack IOCap system on FPGA demonstrating the viability of the system in practice, integrating host drivers, IOCap-VirtIO peripherals, and high-throughput protection hardware on the CHERI-BGAS evaluation platform.

These contributions set the stage for IOCap adoption in production systems. In practice, adoption will require a parameter sweep: choosing an interconnect, studying real-world usage to determine what IOCap format is required, and selecting a secure low-latency MAC function;

and then re-implementation of the drivers and hardware units in line with the system's specific constraints and threat model.

1.2 Hypotheses and overview

This thesis aims to prove four hypotheses, which are each addressed in a dedicated chapter. Each hypothesis is split into facets, individual interesting questions which combine to prove or disprove the hypothesis as a whole.

Chapter 2 — Background

This chapter establishes the threat of malicious DMA. How modern systems offload work to peripherals, what threats they expect from those peripherals, and the modern state-of-the-art mitigations. It examines the IOMMU from two perspectives: spatial safety, i.e. how precisely it can restrict access; and temporal safety, i.e. how quickly it can revoke access permissions once they are no longer needed. It collates prior findings about the IOMMU and reaches a singular conclusion: that even if IOMMUs provide both in theory, modern implementations cannot provide them in practice.

Chapter 3 — Capabilities & the IOCap system

Much like CHERI is an abstract model that can apply to many architectures, IOCaps are an abstract model that can apply to many computer systems. This chapter discusses prior capability systems, the exact scope of protection targeted by IOCaps, and the general concepts of the system—including *leases*, groups of IOCaps authenticated with the same key. It then describes the abstract IOCap format and indicates which components of an existing system, hardware and software, need to change to adapt to IOCaps. The subsequent chapters demonstrate how IOCaps are a practical IOMMU alternative: that they provide temporal and spatial safety in theory, and that they can be implemented in software and hardware in practice.

Chapter 4 — Temporal safety with leases

Hypothesis 1

IOCap systems can use leases to enforce temporal safety under a variety of workloads using a 128-bit MAC function.

Temporal safety relies on choosing a secure MAC function and properly using the lease system, which has many positives, but imposes new constraints that do not apply to the IOMMU. Some of these constraints are applied by the choice of MAC function, including maximum lease lifetimes and maximum IOCaps per lease, so I interrogate that choice here.

Facet 1.1

There exist suitable MAC functions for authenticating 128-bit IOCaps with 128-bit MACs.

Cryptography strength strongly correlates with the sizes of the secrets involved, and there is reason to believe small IOCaps are desirable, so I cover 64-bit security separately.

Facet 1.2

There exist suitable MAC functions for authenticating 64-bit IOCaps with 64-bit MACs.

Heterogeneous systems are ultimately limited by device parallelism, throughput, and bandwidth. That means for a given system there is a maximum number of IOCaps it could want to expose at a time. Ideally it could expose enough leases for a one-to-one mapping to IOCaps, but that takes up area in the form of SRAM, and there may not always be room on the chip. IOCaps should be able to adapt, using many-to-one lease-to-IOCap mappings, with manageable overheads and no loss of security.

Facet 1.3

It is feasible for an IOCap system to include a large enough SRAM-based lease key table to map leases to exposed regions one-to-one, even for high-performance systems with extreme I/O demands.

Facet 1.4

It is feasible and practical for the IOCap system to adapt to underprovisioned systems with fewer concurrent leases than expected, still exposing the same number of regions, without sacrificing security.

Chapter 5 — Spatial safety with compressed IOCap formats

Hypothesis 2

The IOCap system can provide spatial safety at sub-4KiB precision using a 128-bit capability format.

IOCaps must be compact to minimize bandwidth overheads, but they also need to encode detailed information. They need to encode a contiguous region of address space (ideally, a 64-bit address space), caveats that shrink that space, permissions that control reads or writes to the space, and the ID of the lease the IOCap is tied to. In the main case these should all fit in 128-bits, to match the MAC function from Chapter 4, but embedded scenarios may want even smaller IOCaps.

Facet 2.1

An IOCap format can compress a reasonably precise 64-bit address range, access permissions, a lease ID, and multiple caveats to the memory range in 128-bits.

Facet 2.2

An IOCap format can compress a usable subset of those fields in 64-bits.

I/O protocols already use compact structures called 'buffer descriptors' to describe the ops peripherals should perform. Instead of appending IOCaps to these structures, or substituting certain fields for IOCaps and keeping others, it could be ideal to fully replace these descriptors with IOCaps. This may require expressing even higher precision regions, or reserving space for extra fields a basic IOCap would not need.

Facet 2.3

A 128-bit IOCap format can be sufficiently detailed to entirely replace buffer descriptors in protocols such as IOCap-VirtIO.

Chapter 6 — IOCaps in software

Hypothesis 3

IOCap protection is practical to adopt in system software.

IOCaps require an interposer between the host and the peripheral, inspecting all requests from the peripheral, checking the validity of the attached IOCaps, and either blocking them or letting them through. This interposer is controlled by the host to enforce the desired security policies, which requires consistent interfaces at the hardware level (of the interposer itself) and the software level (inside the host operating system, which may already have generic interfaces for exposing memory to DMA). This chapter experiments with those interfaces, emulating an interposer with a host and peripherals using QEMU, implementing host operating system drivers for the interposers, and adding IOCap support to the VirtIO protocol used throughout the stack. CheriFreeRTOS is used as a first validation step without enforcing temporal security, and CheriBSD is then used as a more complete example with temporal security enabled.

Facet 3.1

The VirtIO I/O framework, and peripherals that integrate it, can be extended to replace DMA descriptors with IOCaps without losing functionality.

Facet 3.2

Complex operating systems can be extended to control the trusted IOCap enforcer and untrusted IOCap-compatible peripherals via VirtIO, with minor API changes, without losing functionality.

Chapter 7 — IOCaps in hardware

Hypothesis 4

IOCap protection can be practically and efficiently implemented in hardware, with low frequency, area, throughput, and latency overheads.

Implementing IOCaps in hardware requires a hardware protocol that attaches IOCaps to individual DMA requests—a nontrivial problem in itself—and hardware units that satisfy the interface from Chapter 6 and correctly handle attached IOCaps. I add IOCap support to the AXI hardware communication protocol, implement the interposer as lease manager and enforcer hardware units, extensively test those hardware units, and evaluate their performance using microbenchmarks. These hardware units should be able to run at high clock speeds with low area overhead so that systems can adopt them easily.

Facet 4.1

The AXI interconnect standard can be extended to carry IOCaps with each transaction without significantly degrading clock frequency or area.

Facet 4.2

IOCap-AXI traffic can be routed through an IOCap enforcer and lease manager pair to check transactions without significantly degrading clock frequency or area.

Both hardware units should be able to handle a high volume of requests, both DMA requests from the peripheral and lease management requests from the host, without hurting request latency or throughput.

Facet 4.3

A lease manager can be implemented in hardware and simultaneously service configuration requests from the host and verification requests from an enforcer without significantly degrading latency or throughput.

Facet 4.4

An enforcer can be implemented in hardware and simultaneously handle revocation requests from the host and DMA requests from peripherals without significantly degrading latency or throughput.

Once the hardware units are functional, it should be possible to integrate them with an existing FPGA-based system—one running the software from Chapter 6, even—to demonstrate the functionality and performance of IOCaps in the full stack.

Facet 4.5

It is feasible to build IOCap memory protection into a complete system without losing functionality or significantly degrading performance.

1.3 Publications and collaboration

In my first year I wrote an article for ACM Queue, which was later published in Communications of the ACM, on applying capabilities to a new protocol for remote memory access named CXL [35]. This ended up being a good summary of where my thesis goes, and has been adapted for a small part of Chapter 2. While I was the main author of the article, my supervisor Simon W. Moore and collaborator A. Theodore Markettos both provided structural feedback and proposed edits. This also applies to a paper I wrote in my fourth year, submitted to ACM CCS 2026 (though sadly not accepted), which was also adapted into parts of this thesis.

1.4 Artefacts

The implementation work for this project touched branches in many different repositories, including some associated with the University of Cambridge CHERI group (e.g. FPGA infrastructure [36], CheriFreeRTOS [37], CheriBSD [38], CHERI-QEMU [39]). A full "sandbox" repository, containing these other repositories as submodules, can be found online at `https://github.com/CTSRD-CHERI/iocap-sandbox`.

CHAPTER 2

Background

This chapter covers the how and why of I/O and interconnect security. After briefly explaining common data structures and access patterns for interacting with peripherals using DMA, I describe how different interconnects function and how they implement spatial memory security. At this level, all security policies are encoded through this lens—what memory the host has chosen to expose to the peripherals, and what it has not. I focus on PCIe, a widely accepted industry standard interconnect for internal and external peripherals; and Arm AXI, a point-to-point interconnect standard more commonly used for embedded and SoC-internal peripherals. These have been chosen due to their ubiquitous use across many different domains in industry. Other, more situational, interconnects such as Infiniband and RDMA are not examined here, though the general principles established in this chapter likely still apply.

After discussing security features encoded within the interconnect specifications themselves, I examine system-level features starting with single-level access control tables (common in embedded systems), more complex systems that integrate with host CPUs (e.g. Arm's security 'worlds'), and finally zero in on the IOMMU: *the* standard solution for application-class and server-class systems. These approaches are all based on tables nested within tables and are either limited in size or too large to access at high speeds. I establish the problems with table-based approaches, discuss prior attempts to mitigate them, and make the case that they fundamentally cannot be solved without a ground-up redesign.

2.1 I/O model

Hosts expose memory to a wide variety of peripherals, and these peripherals generally tend to work the same way: the host submits work operations (or *ops*) to peripherals via dedicated *queues*, one or more per peripheral. This model is consistent with the VirtIO peripheral standard [26], the NVMe disk standard [24], high-performance network drivers [25], and NVIDIA and AMD GPUs [40, 41, 42]; and I use it to reason about memory sharing throughout this thesis. Queues may be stored in peripheral or in host memory, and hold buffer *descriptors* that describe regions of exposed host memory the peripheral must access to complete ops.

Memory Exposure	Usage	Example
Persistent	H-Write + P-Read	Submission Queue
	H-Read + P-Write	Completion Queue
	H+P-Read+Write	Shared Memory
Ephemeral	H-Write then P-Read	Network Transmit NVMe Write
	P-Write then H-Read	Network Receive NVMe Read
Non-Exposed	H-Read+Write only	Host allocations

Table 2.1: Common memory access patterns. H = Host, P = Peripheral.

Peripherals may read data from these regions as input, such as for transmitting network packets, or write data as output.

When queues are stored in host memory, I assume they are implemented as ring buffers that may be accessed concurrently by the host and peripheral. The host writes new ops and descriptors in, the peripheral reads them out, and the peripheral writes the outcomes of completed ops. NVMe and VirtIO[1] split the host-written and peripheral-written regions into separate submission and completion queues to avoid data races [26, 24]. These are examples of *persistently* exposed regions, whereas regions for individual ops are usually *ephemeral* and will be quickly unexposed once the op is finished. Other examples of persistent regions are shared memory regions such as memory-mapped GPU framebuffers. High-performance peripherals will likely copy data to and from regions using bulk Direct Memory Access (DMA) to amortize the cost of sending data across the interconnect.

Table 2.1 shows some ways the host may expect peripherals to access memory. Some ephemeral ops expect mutually exclusive access—exactly one of the host or peripheral accessing the memory at any time—which relies on I/O protection following a consistent lifecycle. Memory is allocated, exposed, then once it is unexposed the host is safe to access it again. If these expectations are violated, the host may be compromised. For example, a malicious peripheral may modify a received network packet after it passes a firewall check, or access sensitive data in reused buffers [43, 21]. The purpose of the I/O memory protection is to ensure these expectations hold—that peripherals only access host memory in the way the host intends, when the host intends it.

2.2 Threat model

The threat model I consider in this thesis, shown in Fig. 2.1, is an aggregation of those from prior work [44, 45, 30, 21]. A trusted OS running on the host CPU complex must interact with untrusted peripheral devices. These peripherals require access to resources that are exposed on the system interconnect (usually, memory). Different actors on this interconnect will have

[1]VirtIO also supports a 'packed' queue structure which merges submission and completion.

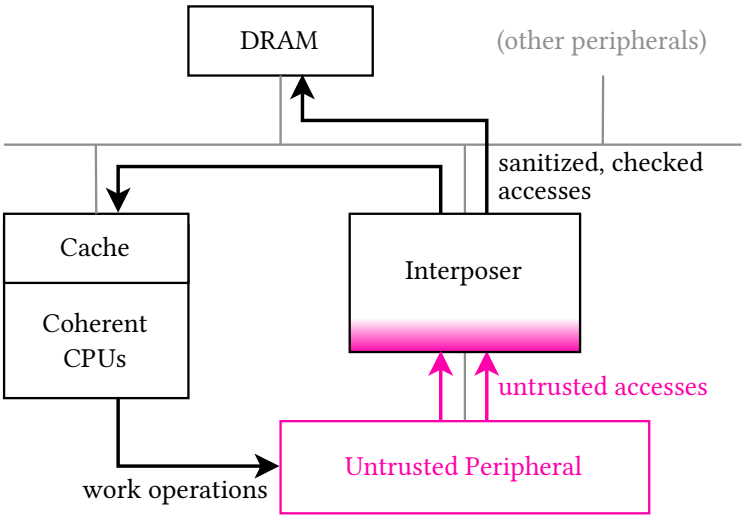


Figure 2.1: Abstract threat model. Work operations are typically sent via the system bus without passing through the interposer, which this diagram simplifies for readability.

different views of memory with different accessible regions [46], though ultimately shared busses will have a consistent mapping of addresses to resources. The OS defines a spatial security policy, restricting which peripherals can touch which memory, that exposes as little as possible—adhering to the principle of least privilege. On Linux and FreeBSD this is controlled by device drivers that call into the OS's DMA API to expose regions of memory to the device [47, 48]. These drivers must therefore be able to accurately reason about the current state of the security policy—which memory is exposed—to interact with peripherals securely.

Peripherals can send requests to access *any* region of memory. Untrusted peripherals may be buggy or malicious, and attempt to access memory the OS and drivers didn't expose, which can lead to exfiltrating sensitive information or hijacking the OS. These access requests are sent to the host on the system interconnect, intercepted by a trusted interposer which ensures the accesses are valid, and forwarded to the destination if permitted. This interposer acts as a policy enforcer.

2.2.1 Assumptions

We must assume the OS and drivers are not malicious, buggy, or compromised by the untrusted peripherals. This is necessary for any policy enforcement system—the entity that defines the policy must be trusted. In practice this is easier said than achieved. Various technologies can help enforce this, including the use of CHERI [1] and memory safe languages to defend against threats at the software level [49], mitigations against Rowhammer attacks at DRAM [50, 51], and encrypting OS data and swap/hibernation files to defend against untrusted storage peripherals [52].

Like DAMN [21] we must assume attacks that use valid DMAs to exposed memory (e.g. a NIC tampering with the contents of packets) are out of scope. This follows from the previous limitation: the OS has defined a policy that allows e.g. the NIC to access network packets, and the policy enforcer does not have enough information to tell if those accesses are "correct".

We must assume the interconnect topology, including the physical bus and any switches, can be trusted to securely transport DMA requests and metadata. This includes spoofing protection, preventing malicious entities from forging new transactions from other peripherals; and snooping protection, preventing malicious entities from inspecting the contents of legitimate transactions not meant for them. These are outside the policy enforcer's responsibilities. In practice, the enforcer may depend on spoofing protection for identity-based access control, and snooping protection is generally desirable for confidentiality. Some models explicitly allow snooping metadata [45] but others do not [21]. Different interconnects in different systems may be able to prevent spoofing differently e.g. PCIe Access Control Services (Section 2.3.1.2) vs. AXI Firewall units [53]. Snooping threats also vary between systems—for example, some may have physically exposed pins [54]—and may already have solutions, such as PCIe Integrity and Data Encryption (IDE, Section 2.3.1.2). This also implies that external Thunderbolt hubs and docks, which are effectively PCIe switches, must also be trusted unless using IDE.

We must also assume attacks on the policy enforcer itself, such as malicious configuration traffic, are out of scope. Malicious configuration traffic can be prevented at the interconnect level by ensuring only the host can send policy configuration messages, and using spoofing protection to prevent hostile actors from masquerading as the host. It is also important to prevent unprivileged contexts on the host from sending configuration traffic, which has recently broken confidential compute on AMD CPUs [55].

2.3 Interconnect mechanics and protection

Different systems use different interconnects, which use different mechanisms and encodings to complete bulk DMA transactions. This section examines PCIe and AXI, two industry standard interconnects, to establish their existing protection mechanisms and how they both map on to the threat model.

2.3.1 PCIe

PCIe is an interconnect allowing a 'host' (e.g. the main system CPU) to configure and manipulate connected 'peripherals', and map their memory into a shared address space. It evolved from the PCI and PCI-X interconnect standards, which both used a single bus between 32 internal and external endpoints[2], where only one endpoint could speak at a time. In contrast PCIe defines both a physical link standard to connect individual endpoints *and* defines a standard interconnect topology for systems with a single trusted host and multiple untrusted peripherals.

A single PCIe hierarchy is a tree of PCIe links connecting peripherals ('Functions' in PCIe-speak) through a tree of switches rooted at a root complex (RC) [57, §1.3]. The root complex is the interface from the peripherals to the host CPU/memory subsystem. As of 2012, Intel PCIe

[2]In practice PCI buses could only support 4 or 5 peripherals at most due to electrical constraints [56, Table 1-2] but more could be added through PCI-to-PCI bridges [56, Figure 1-4].

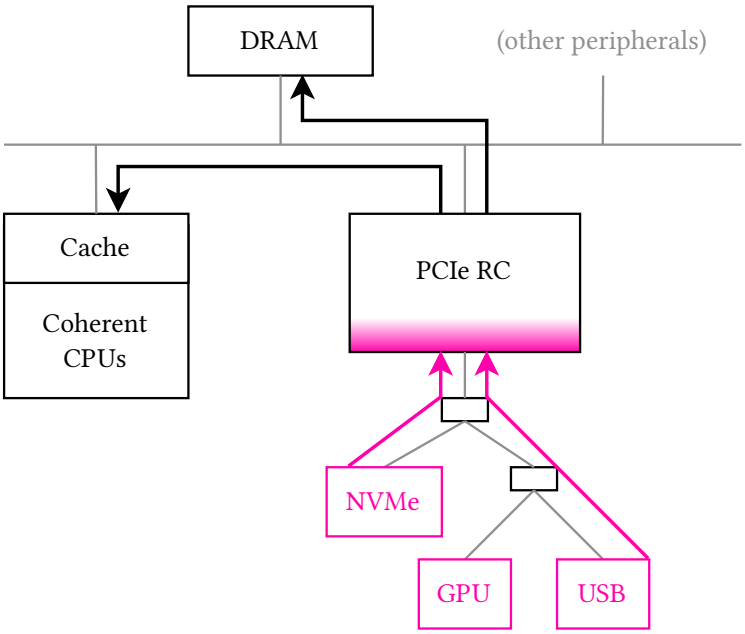


Figure 2.2: PCIe's simplified protection model, where the root complex (RC) exposes PCIe peripherals to cache-coherent host memory.

hubs can connect the root complex directly to the last-level cache, although DMA through PCIe is restricted to a subset of it [18, 58, 59].

It is also possible to have multiple 'peer' root complexes connected to a single host [56, Chapter 21]. A common example is dual-socket motherboards, which contain two physical CPUs that each contain a separate root complex. In this case each root complex creates a separate hierarchy, completely disjoint from the others, and each peripheral endpoint is connected to exactly one root complex at a time. Multi-Root I/O Virtualization tried to remove this restriction by allowing peripherals to connect to multiple root complexes [60] but is now entirely defunct.

2.3.1.1 DMA transactions

PCIe allows all endpoints, root complex and peripherals[3], to initiate read and write transactions targeting other endpoints in the interconnect. PCIe 5.0 defines multiple address spaces for reads and writes: Configuration for setup, I/O for compatibility with legacy PCI, Message for miscellaneous events, and Memory for DMA [57, §2.1]. This section only refers to Memory, through the principles broadly apply to the others. Reads and writes are requested by sending a Transaction Level Packet (TLP) to the target endpoint. Each TLP consists of zero or more prefixes, a header, zero or more bytes of data payload, and an optional digest for integrity checks [57, §2.1.2]. The header includes the target address (encoded in 32-bits if it fits, otherwise in 64-bits), the kind of transaction (read/write), a transaction ID, and the length of the accessed region. Each transaction accesses a contiguous region, starting at the address in the header, which cannot cross a 4KiB boundary [57, p128].

[3]Peripherals have to expose a Bus Master Enable bit, which defaults to 0 and is writable by the host, to enable them to initiate transactions [57, Table 7-3]. In practice all modern devices require this to be set to 1.

If the transaction is a read the request TLP includes no data. Reads are 'Non-Posted Requests', so require a completion from the target endpoint. Each TLP has a maximum payload length, which can be set at runtime to any power-of-two between 128-bytes and 4096-bytes [57] (as constrained by what the endpoints support). If more data was requested than can fit in a single TLP the target will send multiple read completion TLPs back in ascending memory address order. Write request TLPs include all the data for the write, implicitly capping the length of any individual write transaction, and are Posted Requests so the target does not send any completion TLP back.

PCIe ensures specific ordering guarantees for memory request and completion TLPs from the same endpoint [57, §2.4], some of which depend on the tree topology[4]. In short: write request TLPs cannot be reordered with respect to other write request TLPs, a later read request TLP cannot be reordered ahead of an earlier write request TLP, and read completion TLPs cannot be reordered with read completions with the same transaction ID. All other reorderings are fair game and in some cases are required to avoid deadlocking. MSI and MSI-X, PCIe's interrupt signalling standards [57, §6.1.4], use memory write TLPs to signal interrupts and retains their ordering guarantees. PCIe also supports three different kinds of atomic operation TLPs [57, §6.15], all non-posted, and all instances of read-modify-write sequences (returning the initially read value). These operations (fetch-and-add, unconditional swap, and compare-and-swap) are limited to small regions, up to 16-bytes, and do not observe the same ordering rules—the only way to ensure two TLPs are processed in order is to wait for the first completion before sending the second request.

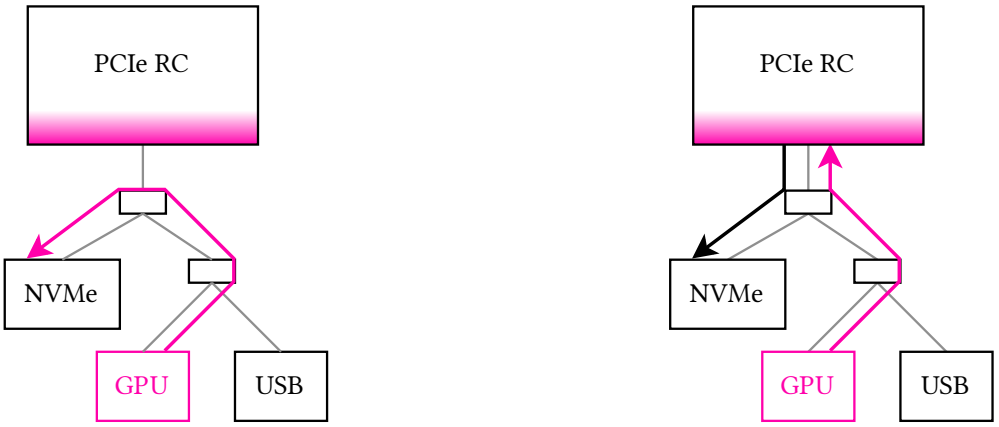
2.3.1.2 Protection

Under the standard PCIe topology the root complex is given the most trust, and acts as the owner of the resource: all PCIe-addressable memory, including host DRAM, potentially via the last-level-cache, and any exposed memory on other peripherals. Peripherals can initiate requests, which pass through a switch network configured by the host, to consume the resources. Thus, the peripherals are consumers and the root complex is the policy enforcer.

Requests for host DRAM always end up at the root complex, but the host may also wish to control peer-to-peer peripheral requests. Access Control Services (ACS) extends the PCIe specification with new controls that can be enabled on switches[5] to control what messages can be sent between what endpoints. ACS features must be switched on/off on each individual port on every switch in the interconnect, so are granular enough to e.g. allow P2P accesses between two endpoints but no others. These features include 'Source Validation', which checks if requests sent through a port have the correct bus number; 'P2P Request/Completion Redirect', which redirects all P2P (peer-to-peer) packets up to the root complex for verification before

[4]In a tree, all requests between two devices will take the same route. If each individual element in the tree enforces ordering, you can guarantee requests will arrive at the consumer in the same order they were produced [56, p137]. CXL 3.0, a coherent extension to PCIe 6.0 [61], defined a new unordered class of transactions to allow non-tree topologies where different TLPs can take different routes of different lengths [62, 25:00].

[5]ACS applies to dedicated switch devices and other endpoints with multiple ports, such as the Root Complex itself and peripherals with multiple internal 'Functions'.



(a) Without P2P Redirect - requests and responses are sent directly between peripherals.

(b) With P2P Redirect - requests and responses are routed to the root complex first.

Figure 2.3: ACS P2P Request/Completion Redirect, explained.

sending them to their targets (Fig. 2.3); and 'P2P Egress Control', which blocks P2P requests outright. These controls help ensure peripherals do not forge TLPs from other peripherals, and control which peripherals may communicate P2P, but can't provide any other protection (for example, spatial safety) without significant latency from passing through the root complex, and cannot protect against snooping or hostile switches.

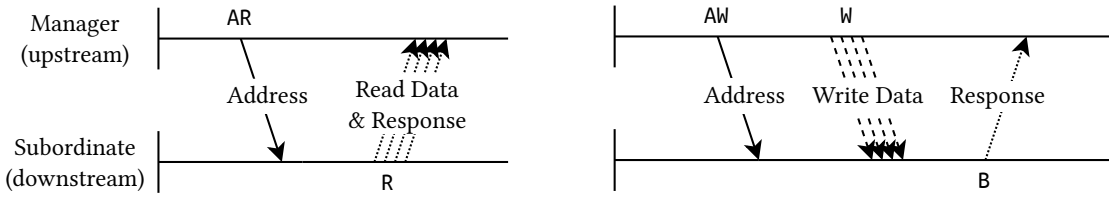
Switches are usually trusted to carry requests without modification. Designers with stronger threat models can use PCIe IDE, which standardizes a method for device attestation [63] and encrypting data in-transit [64]. Device attestation works on a link-to-link basis, allowing every switch in the tree to be authenticated. Encryption can be configured link-to-link, meaning each switch has to decrypt incoming data and re-encrypt it for the next link, or as a 'selective stream' where the traffic passes through switches unchanged and is only decrypted at the destination. This is useful to avoid snooping but is orthogonal to the problem of buggy peripherals.

2.3.2 AXI

AXI is an Arm specification [65] which evolved from Arm's AMBA specification [66] for the Advanced High-performance/System/Peripheral Buses (AHB/ASB/APB). Like PCI, the AHB and ASB buses were designed to be system buses with a single master per cycle. Unlike PCI, the AHB, ASB, and APB were all designed to connect to internal peripherals and did not include an external connector or form factor. Like PCIe, AXI specifies an interface for point-to-point connections which can grow into a full interconnect mesh, but unlike PCIe AXI does not have any de facto interconnect topology. AXI only defines the interfaces between transaction requestors (or 'Managers'), transaction completers (or 'Subordinates'), and interconnects that connect many managers to many subordinates.

2.3.2.1 DMA transactions

AXI transactions are sent between manager and subordinate as groups of 'transfers', which I call 'flits' (flow control units) to avoid confusion. Each AXI connection has five channels, which



(a) AXI read transaction.

(b) AXI write transaction.

Figure 2.4: AXI flow diagrams, with channels marked. Each read and write data flit counts as an individual 'transfer' within the transaction.

can each send one flit per cycle. Read transactions are initiated by the manager sending a flit over the read-request (`AR`) channel, including the address of the read and the amount of data to read, in which case the target subordinate will eventually respond by sending one or more flits over the read-data (`R`) channel of the data that was read (Fig. 2.4a). Managers initiate writes by sending a flit over the write-request (`AW`) channel, one or more flits over the write-data (`W`) channel, and waiting for a flit to come back on the write-response (`B`) channel (Fig. 2.4b).

AXI transactions are split into one or more 'transfers', where each transfer is for a contiguous region of memory, but the transfers inside a single transaction may not be contiguous. Flits on `AR` and `AW` contain an address, the size of each transfer (in bytes), a length (i.e. the number of transfers), a Burst attribute which defines which addresses are touched by each transfer[6], and a transaction ID. Like PCIe, requests are not allowed to cross a 4KiB boundary (and as such are capped at 4KiB in length). A manager may shrink this boundary to any power-of-two between 64-bytes and 4KiB, which may allow logic to be optimized out of the connected subordinates at design time [65, §A4.1.3]. The width of the data sent over the `R` and `W` channels is also not fixed by AXI, but it is fixed at tape-out. A compliant implementation can choose any power-of-two between 8-bit and 1024-bits.

Like PCIe, AXI provides limited ordering guarantees for requests and responses based on the transaction ID [65, §A6.3]. Transaction requests on the same channel with the same ID and destination are guaranteed to remain in order, i.e. read requests will not be reordered with respect to read requests with the same transaction ID, ditto for writes—but reads and writes have no guaranteed ordering relative to each other. Transaction responses with the same ID are returned in the same order as their requests, i.e. if you send two read requests with the same transaction ID, the first response with that ID will map to the first request, ditto for writes—but any responses with different IDs may be reordered. The only way to guarantee ordering of reads with writes, or requests for different destinations, is to wait for the first request to complete (unlike PCIe, all requests have completions) before sending the second request.

AXI supports two kinds of atomic accesses. One way is to use a pair of exclusive-read and exclusive-write transactions to the same address [65, §A7.3]. The exclusive-read tells

[6]In the basic case each transfer will start at the address the previous one ended, but you can also use a Fixed burst mode to e.g. read data from a FIFO at a fixed address multiple times, or use a Wrapping burst mode to choose a specific word to read first from an aligned region.

the subordinate to track the read-from address and any writes from other managers. The exclusive-write only successfully completes if there have been no writes from other managers since the exclusive-read, enabling manager-side atomic computation on up to 128-bytes of data. Alternatively, subordinate-side operations can be performed with dedicated atomic transaction types [65, §A7.4]. Addition, integer min/max, and bitwise operations can be performed atomically on up to 8-bytes of data. Compare-and-swap can be performed atomically on up to 16-bytes of data. These atomic transactions are both reads and writes—writing a new value while reading the old— and are thus sent over the `AW` and `W` channels and completed with both `B` and `R` flits. These are the only exceptions to AXI's general ethos of avoiding dependencies between reads and writes [65, §A1.1].

Extensions to AXI, e.g. those supporting caching, virtual addressing, or TLB shootdowns, can extend the AXI channels with extra signals. Caching adds `DOMAIN` signals to `AR` and `AW` to signal if the targeted memory uses software or hardware coherency [65, §A9.3]. Virtual addressing extends `AR` and `AW` with Secure Stream, Stream, and Substream ID signals; indicating the 'world' (see Section 2.5, p40) and virtual address space of each transaction [65, §14.3]. Distributed virtual memory requests, i.e. TLB shootdowns, create new `AC` and `CR` channels for sending and receiving snoop-based messages [65, §A2.3 & §A15]. These demonstrate AXI's flexibility—it is named the Advanced eXtensible Interface for a reason.

2.3.2.2 Protection

As AXI is only a point-to-point link standard, it doesn't necessarily come with a trust model built in. Fortunately AXI is only one part of the Arm AMBA family of specifications, including protocols for coherence [67, §D][68] and virtual memory [69, 70][65, §A15], which themselves interact with other Arm specifications that *do* impose a trust model. These include the MPU (Section 2.4.1), TrustZone (Section 2.5.1), and the SMMU (Section 2.6.3).

2.4 Simple protection tables

In the most basic cases processors may use single-level tables to protect access to physical memory. These tables are commonly stored in fast SRAM or registers instead of slow DRAM memory, are easier to set up than full MMUs or virtual memory, and can be used for low-level CPU configuration—e.g. configuring whether certain memory regions can be cached by the processor. They are quite limited in terms of how many memory regions they expose.

2.4.1 Arm MPU — Small permission tables

Arm's R-profile [11] and M-profile CPU specifications [12] (for real-time and embedded-class processors, respectively) include the concept of Memory Protection Units (MPUs). These are designed to limit the CPU's access to memory—they are relevant to peripherals which may use these embedded CPUs to run firmware. In the past, peripherals without MPUs or with

misconfigured MPUs have been used to attack host memory [7].

MPUs consist of small, single-level tables of accessible regions stored in registers, encoded as pairs of the base address and limit address of each region [11, §C1.2][12, §B10.1]. Each region has read/write/execute permissions [11, §G1.3.16][12, §D1.2.171], plus cacheability and other attributes. R-profile processors can enable two stages of physical memory protection, one for a hypervisor and one for guest OSs, with separate MPUs [11, §C1.3]. These specifications only support up to 256 regions per MPU [11, §G1.3.26][12, §D1.2.175], and fewer are supported in practice [71].

2.4.2 RISC-V IOPMP — Two-level small permission tables

RISC-V includes a specification for an I/O Physical Memory Protection unit (IOPMP) [13] designed to protect the physical address space from peripheral accesses. This is a two-step process: requests are associated with one or more 'memory domains' through an implementation-defined 'request-role-ID', and then the requests are checked against all the regions associated with all of those memory domains. The specification defines three possible formats for the 'SRCMD' table mapping request-to-domain, and three possible formats for the 'MDCFG' table mapping domains-to-regions. All three formats for MDCFG map domains onto slices of a flat array of memory regions [13, §3.4]. The IOPMP specification allows up to 63 memory domains [13, §A2], and in practice IOPMP implementations range from 4 through 512 regions [72, 73, 74, 75].

2.4.3 CXL GFAM — Small translation tables

Note: this is adapted from an article that I wrote for ACM Queue and Communications of the ACM [35].

CXL [61] is a set of protocols layered on top of PCIe that allow cache coherency. Separate protocols allow accelerators to cache host memory (`CXL.cache`) and hosts to cache device memory (`CXL.mem`). Industry is currently focused on `CXL.mem` memory expansion peripherals from vendors such as Samsung [76]. Like PCIe, it uses a host-peripheral model. Each CXL host controls a set of connected peripherals, and maps all the memory they expose into a Host Physical Address Space (HPA). The host may also map its own memory into the HPA for accelerators like GPUs to access over `CXL.cache`, but current peripherals just expose RAM to the host over `CXL.mem`. CXL 3.0 upgraded `CXL.mem` to allow hosts to *share* memory regions, both through 'multi-headed' and 'GFAM' peripherals, and recent research has investigated peripherals using CXL memory expansion directly [77].

Multi-headed `CXL.mem` peripherals connect to multiple hosts at the same time, and can map the same regions[7] of physical memory into all of their HPAs at once. Those hosts can all cache copies of that memory, and the peripheral is responsible for ensuring coherency. For example, if host H1 tries to write data to memory, the peripheral is responsible for realizing that hosts

[7]Also known as 'logical devices'.

H2 and H3 have that memory cached and telling them to invalidate their caches (see Fig. 2.5). Unfortunately each of those hosts can only access 16 regions [78, §2.5], so they will necessarily be large—on the order of GiBs or 100s of MiBs.

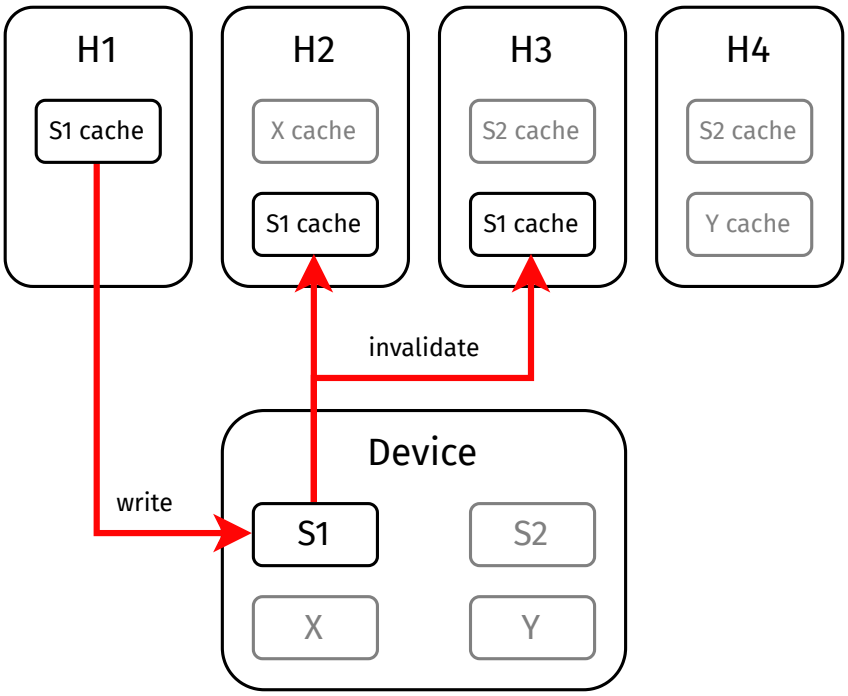


Figure 2.5: Memory sharing with coherence in CXL.

GFAM peripherals take this a step further by not being attached to specific hosts. Any host can map GFAM memory into its HPA. Any endpoint, host or peripheral, in that HPA can talk to the GFAM directly and access that memory. The GFAM is configured with separate single-level translation tables for each endpoint, such that each endpoint can access 8 regions of physical memory [78, §7.7.2.4]. These regions may overlap, allowing memory sharing, or be isolated. See Fig. 2.6, where the host and accelerator have separate translation tables with different views of the HPA. Again, because each endpoint has few ranges they will be large. 'Memory Groups' [78, §7.7.2.5] can punch holes in these ranges and hide specific 'Blocks', but the holes will be at least 64MiB [78, Table 7-67, Min. Block Size].

Both kinds of memory provide protection through *non-exhaustive translation*: endpoints request addresses in their HPA, which get translated to local device addresses, and that translation may fail i.e. the endpoint may not have memory mapped at that address. These mechanisms, similar to an MMU, provide inflexible coarse-grained protection. At most, each endpoint can access 16 memory ranges per device. The only way to change the mappings and transfer access rights is to convince the 'Fabric Manager', which has no defined interface for this [78, §7.6.1].

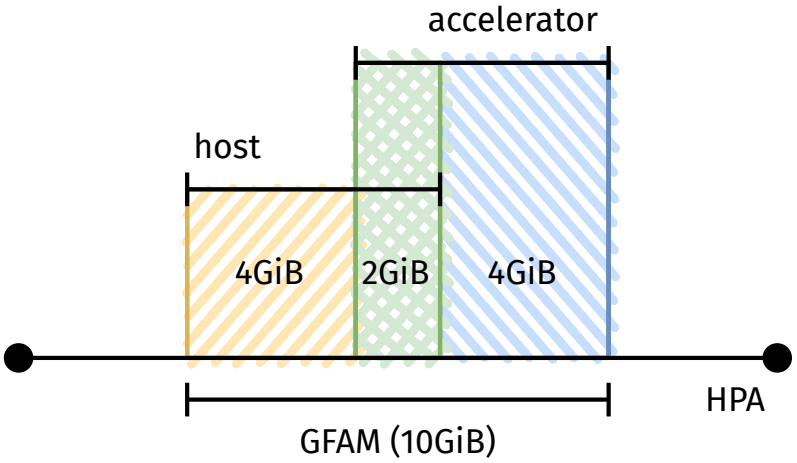


Figure 2.6: Sharing GFAM within an HPA.

2.5 Whole-processor-integrated permission tables

Isolation is a classic way to ensure certain parts of a processor's workload are protected from malicious influence, external or internal. Arm and RISC-V processors allow coarse-grained isolation of untrusted software or hardware components by splitting them into separate 'worlds', with separate views of the physical address space. These views are flat-mapped, i.e. the actual addresses are unchanged when they pass through the interposer. Of course, this uses more tables.

2.5.1 Arm TrustZone

TrustZone separates an SoC into two worlds: the Normal world and the Trusted world [79]. Data and execution in the Normal world is Non-secure, and that in the Trusted world is Secure—though for simplicity I refer to them as normal-world and secure-world. Arm's A-profile (application-class) and M-profile cores both support TrustZone, and transition between normal-world and secure-world execution using different trusted software mechanisms. The core is responsible for isolating world states from each other, which includes cache lines, TLB entries, certain registers, etc. All outgoing AXI transactions are also tagged with a `NS` (Non-secure) bit that indicates which world they were issued from. A-profile processors can have separate page tables for normal-world and secure-world—which must be stored in memory owned by the relevant worlds—and each page table entry has a `NS` bit to control how secure-world contexts try to walk through it. M-profile processors, which don't support virtual memory, are specified to use two levels of table—the 'secure attribution unit' determines if memory is normal-world or secure-world, and then separate normal- or secure-world specific MPUs separately define the accessible regions[8]. In this sense the physical address space is split between the normal-

[8]M-profile is actually more complex: there is a third category of 'unsecure-callable' memory that is used to store normal-to-secure transition code, which uses the normal-world MPU for read/write/execute permissions. Cores can also use an implementation-defined attribution unit instead of Arm's specified 'secure attribution unit'.

and secure-worlds.

The NS bit is carried by all transactions on the AXI interconnect to completers with different approaches to security (Fig. 2.7). Certain completers may only want to receive secure-world or normal-world transactions, depending on which world they live in, and switches on the interconnect can ensure that is the case [79, §4.2]. More complex completers could accept both and expose different resources to secure-world and normal-world transactions. MPU-like devices, such as the TZC-400 [80], can statically or dynamically partition ranges of system RAM using small single-level tables[9]. The NS bit for an initiator should always be set by trusted hardware—a device which is TrustZone-aware and trusted to tell the difference (like a CPU) can set it, it could be hardcoded for untrusted initiators at the design level, or it could be configured by a register only accessible by secure-world software [79, §4.3]. While earlier threat models assumed the NS bit was always trusted [81, §3.2.1], more recent specifications include an extra out-of-band signal to determine which endpoints are actually allowed to issue secure transactions [82, 83, §2.2] i.e. set NS=0.

A binary secure/normal signal may not be flexible enough for some. For example, one of the earliest justifications for TrustZone was processing DRM media in secure-world to prevent normal-world programs from e.g. recording protected video streams [81, §6.2]. If that media's ever going to appear on screen the display needs to be able to composite the video from secure-world memory alongside the rest of the user interface rendered in a normal-world context. It needs to read both! At that point an IOMMU can be used to only give trusted (i.e. secure-world) peripherals *limited* access to secure-world memory [79, §4.3].

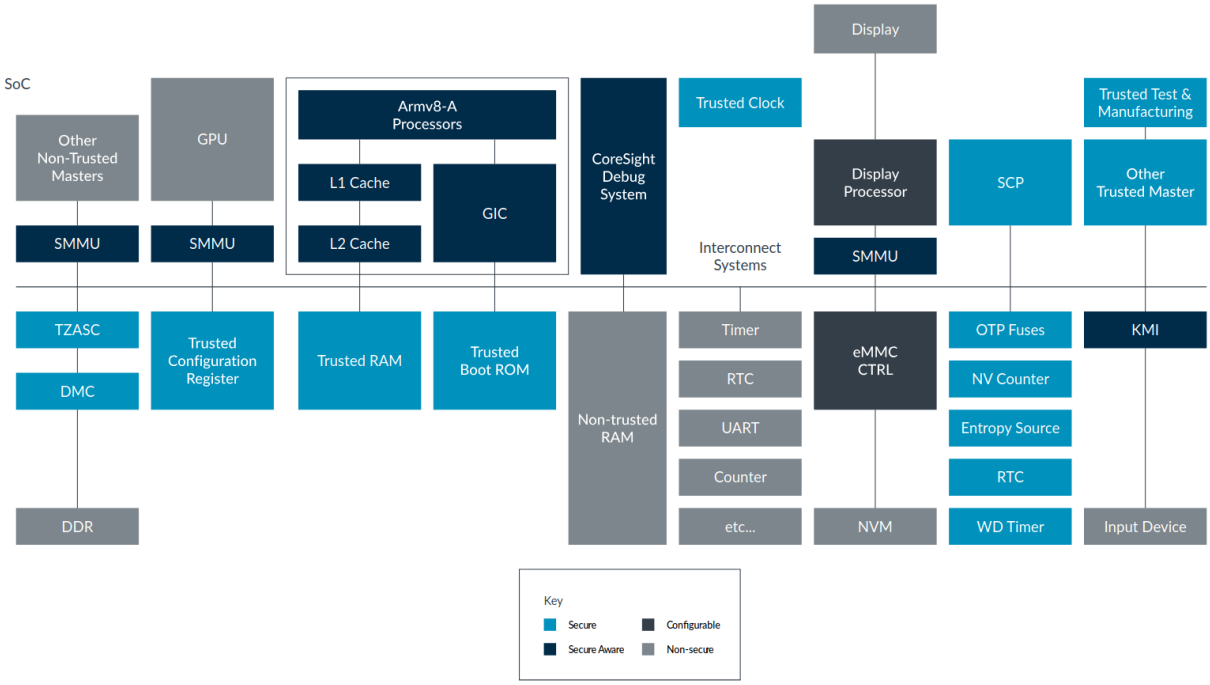


Figure 2.7: An example of a TrustZone-aware system architecture, where different peripherals are in different worlds [79, Figure 4-1].

[9]The TZC-400, which is recommended by a recent guide to TrustZone [79], only supports 9 separate regions [80, §1.3].

2.5.2 Armv9 Realms

Armv9 Realms [84] adds new worlds: the Realm world, which is mutually distrusting of secure-world, inaccessible from secure-world and normal-world, and cannot access secure-world; and the Root world which can access all other worlds for bootstrapping purposes. This is a new way to manage VMs, where all VMs live in the Realm physical address space (isolated from each other through virtual memory, controlled by formally verified Realm Management Monitor firmware [85]) that cannot be accessed by the secure-world or normal-world software that created them. Arm note that while in theory, these four physical address spaces are completely disjoint, in practice they are all different views over the same mapping of 64-bit integer-to-memory-location. To prevent the physical address spaces from ever overlapping, a multi-level Granule Protection Table (controlled by Root) splits the physical space into Granules—4, 16, or 64KiB chunks—which are either assigned to exactly one of the four worlds or 'Completer-Side Filtered' (e.g. MMIO locations, which receive all requests and may change their behaviour depending on the security state of the requester). The interconnect is trusted to carry the world that originated each request on top of the stream IDs etc. MMUs and SMMUs consult the GPT for every translation, and for every table lookup within that translation—e.g. a translation inside a guest VM has to granule-check the physical location of each level of the guest-virtual-to-guest-physical and guest-physical-to-host-physical[10] page table walks.

2.5.3 RISC-V WorldGuard

RISC-V Worlds [86, §26], which is approaching ratification after being donated from SiFive [87, 88], applies a simplified combination of Realms and TrustZone. RISC-V cores track the current 'world-ID' (WID) and attach it to all outgoing physical memory requests. The WID for each core may be fixed in hardware, modified by a lockable status register, or dynamic within bounds set by higher privilege modes (e.g. Machine mode may allow Supervisor mode to change WID within a list of allowed values). The ratified specification does not prescribe a specific checker implementation, only saying that a transaction "is only allowed to complete successfully if the targeted resource has the appropriate access permissions (read or write) on that address for the WID on the transaction", but original draft specification from SiFive defines a generic 'WG Checker' that uses a single-level table similar to a PMP. The means of attaching WID to each transaction in the interconnect is also left unspecified. Like Arm TrustZone/Realms, RISC-V Worlds is only intended to handle a small number of worlds—the specification recommends 2–8.

[10] Arm permits for one granule check to be skipped: that of the descriptor for the stage 2 (guest-to-host-physical) page table. They advise this should only be considered when using other physical memory protection such as memory encryption [84, p31].

2.6 IOMMUs — Translation as protection

'Worlds' are few in number and limited in the protection they can provide. Ideally, systems would follow the principle of lease privilege: granting each peripheral access to only the memory regions it needs, and nothing more. This requires fine-grained spatial safety, more flexible exposing and unexposing, and per-peripheral address spaces. Virtual address translation moves the protection tables into DRAM in exchange for this extra flexibility, and is the basis of the IOMMU.

2.6.1 Basic design

As dedicated graphics cards became popular in the 90s, the increasing memory bandwidth requirements of their workloads started to put greater demands on the 66MHz base-PCI bus. Enter the Accelerated Graphics Port (AGP), a new port specification specifically for graphics cards, designed by Intel based on 66MHz PCI with significant bandwidth improvements [89]. AGP was designed around two conceptual usage models: 'DMA' for bulk transferring data from (slow) system memory to (fast) local graphics memory, and 'Execute' for randomly accessing both interchangeably. The Execute model required a contiguous view of the system memory relevant to the GPU, even if it's physically fragmented or paged. Thus, the AGP specification includes the Graphics Address Remapping Table (GART)—a single level page table which maps a separate region of system address space (the 'aperture', which as of AGP 3.0 [90] is strongly recommended to be at least 16MiB, up to 1GiB+ for "higher end workstation platforms") to 32-bit-addressable physical system memory. The aperture was also accessible by the host CPU [91], providing another level of translation on top of the CPU MMU, but it was only one page table so couldn't be used to isolate multiple peripherals from each other.

When PCIe rose to fame in the early 2000s, it retained the concept of the GART. AMD released version 1.0 of their IOMMU specification [91] in February 2006. The IOMMU allows the system to define multiple address spaces, each with a multi-level page table just like those used by host CPUs, and bind different peripheral devices to different address spaces. AMD's spec specifically cites the GART as an inspiration and lists three "enhancements to system-level software" provided by the IOMMU:

- **"Legacy 32-bit device support on 64-bit systems"**

  Legacy PCI or PCIe devices that only support 32-bit addressing can use a page table which maps their 32-bit addresses to higher 64-bit system addresses, "(without requiring bounce buffers and expensive memory copies)".

- **"Secure user-level application access to selected devices"**

  The OS may allocate a page table for a device that only maps in the same physical pages used by a userspace process. The device is isolated to that process' memory, and cannot be tricked into accessing other memory.

- **"Secure virtual machine guest operating system access to selected devices."**

  In a similar vein, a device can be completely isolated within a guest VM by reusing the guest-physical to host-physical mapping as a page table.

The emphasis on virtualization continues as the spec states the IOMMU is "designed so that it can be emulated in software by a VMM [Virtual Machine Manager]". Intel's version, 'Intel Virtualization Technology for Directed I/O' (or VT-d) [15], was drafted a month later and released in May 2007. Arm's first System Memory Management Unit (SMMU) standard [82] was released in 2012. These standards also provide multiple address space support, using multi-level page tables, and share the emphasis on virtualization. To this day these standards live on, though AMD's naming stuck, all colloquially named the IOMMU.

2.6.2 PCIe's IOMMU standards

Since the release of the IOMMU, the PCIe standards body has standardized numerous extensions to the PCIe specification that interact with the IOMMU or depend on its features in some way. They give insights into how the IOMMU is expected to be used.

2.6.2.1 PASIDs and ATS — Basic translation

PCI-SIG introduced the Address Translation Services (ATS) extension [57, §10.1] in March 2007 to provide a common abstraction of the competing IOMMU specs. It defines a mechanism for peripherals to cache the resulting translations and invalidate them when requested.

By default all memory TLPs are Untranslated. For a peripheral to cache translations it must first send a Translation Request TLP to the root complex. Translation request TLPs are very similar to memory read TLPs, except instead of reading from a given virtual address, the root complex returns a completion TLP with the physical translation. These TLPs are routed by address, so all virtual addresses must occupy some aperture in PCIe address space assigned to the root complex. Once the peripheral receives the translation completion, it caches the translation *inside its own TLB*. Peripherals with cached translations may then directly issue Translated memory TLPs to those addresses (which may reference memory in other peripherals) without any other checks. Translated TLPs are identical to Untranslated TLPs save for the two-bit Address Type field in the TLP header—they do not include any extra metadata, such as the original virtual address, and they do not require a TLP prefix. The host is responsible for sending ATS Invalidation Request TLPs to peripherals whenever a translation that may be cached has been changed. The host may also choose *not* to translate an address in which case peripherals must use Untranslated TLPs for every access.

ATS supports multiple virtual address spaces by attaching a Process Address Space ID (PASID) prefix to TLPs that use virtual addresses. PASIDs are not shared between peripherals—the root complex could map a PASID from peripheral A to a different page table than the same PASID for peripheral B. The PASID field in the prefix is 20-bits wide, allowing for roughly a million address spaces per peripheral, but peripherals may only support fewer bits.

PCIe ACS (Section 2.3.1.2, p34) also has controls to restrict Untranslated and Translated traffic. 'Translation Blocking' simply blocks any TLPs related to ATS, including Translated memory TLPs, which is necessary for malicious peripherals. The only difference between an Untranslated and a Translated memory TLP is one bit in the header, so malicious peripherals could flip this bit and send fake Translated memory TLPs to whatever address they want. Such peripherals should not be trusted to use ATS at all, so should be blocked from translation by any switches they pass through. They can still be virtualized, use the PASID prefix, and get their requests translated by the host—they just can't cache the translations themselves. 'Direct Translated P2P' offers an escape hatch from the P2P redirect ACS control. When P2P Redirect is enabled for a port, P2P TLPs arriving at that port are redirected to the root complex. If Direct Translated P2P is *also enabled for that port*, Translated P2P TLPs will go straight to their destination and only Untranslated TLPs will be sent to the root complex.

ATS is really only intended to be a performance enhancement for trusted peripherals that need to talk to virtual machines. IOMMUs can be used to isolate malicious peripherals, and you can force the peripheral to always consult the IOMMU through ACS, but otherwise an ATS-enabled peripheral can still access whatever it wants if it chooses. The true selling point of ATS is accelerated virtualization—to allow untrusted guest VMs to directly access a trusted peripheral, such that the hypervisor can guarantee the peripheral won't touch memory outside that guest VM, without intervening on each access. For that to work, the peripheral must be trusted to use the right PASID at the right time. The last piece of the puzzle is to allow peripherals to be shared between multiple guest VMs, which requires further trust in the peripheral to not leak data between the colocated VMs.

2.6.2.2 SR-IOV, SIOV — {Single-root, Scalable} I/O virtualization

Some peripherals are capable of accelerating workloads from many virtual machines at once. Network cards are a great example, as some can expose thousands of virtual network interfaces and multiplex them all down to the few physical interfaces present. Even if the host isolates the VMs correctly, the configuration and data *inside the peripheral* for the different virtual machines must also be isolated. A peripheral could expose multiple PCIe Physical Functions, literally subdividing its physical resources, but then if fewer VMs were needed the resources for the unused Physical Functions would be wasted. Instead, using Single-Root I/O Virtualization [57, §9], the peripheral can be dynamically subdivided into any number of Virtual Functions (all linked to a single Physical Function) that act as isolated devices. These VFs can be directly exposed to VMs by the hypervisor, and the peripheral is responsible for isolating VFs from each other.

SR-IOV on its own does not prevents VFs from accidentally accessing host resources for the wrong VMs—which could happen if the VF blindly accesses memory addresses provided by a hostile VM. That protection requires an IOMMU to map accesses from VFs to the correct guest address spaces. For example, an Intel whitepaper on SR-IOV from 2008 specifically notes that their implementation of SR-IOV requires Intel's VT-d IOMMU [92], which provides "hardware

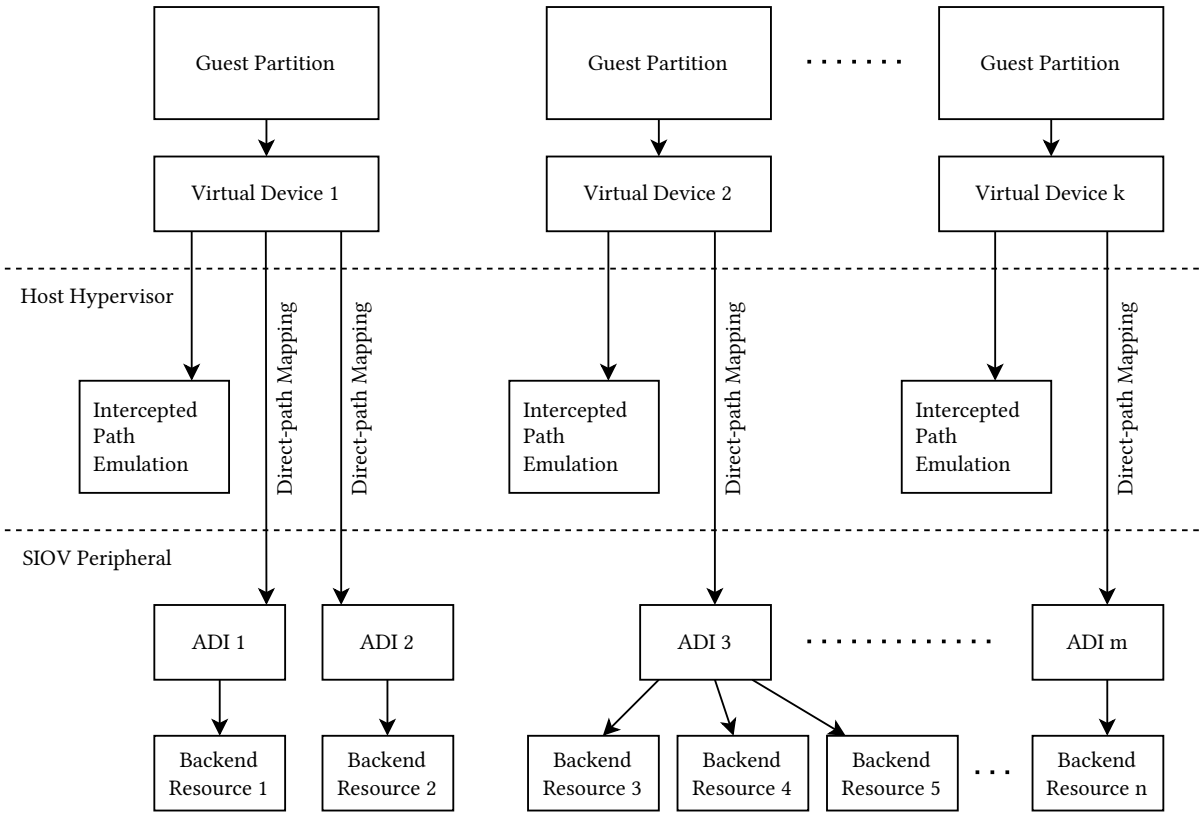


Figure 2.8: The SIOV Virtual Device and ADI model [93, Figure 7-1].

assisted techniques to allow direct DMA transfers". The peripheral itself is still trusted, the IOMMU only stops it from being a confused deputy, so ATS can also be used to cache the IOMMU translations.

Ten years later, in 2018, Intel created a new specification for I/O virtualization that aimed to support more virtual machines per peripheral, with resources partitioned at finer granularity. Scalable I/O Virtualization (SIOV) [93] was donated to the Open Compute Project in 2022, and is compatible with SR-IOV (in that a device that supports SIOV can also support SR-IOV). Under SIOV, each peripheral's resources are split into fine-grained isolated Assignable Device Interfaces (ADIs). They may be statically or dynamically partitioned depending on the device [93, p16]. Each ADI is associated with at least one PASID, and the PASID is the only mechanism for isolating host resources with the relevant ADIs. Isolated domains within the host, usually guest VMs [93, §6.1], each have access to a Virtual Device, which emulates the slow path e.g. configuration in software, and has a direct connection (i.e. one not mediated by the hypervisor) to one or more ADIs [93, §9.4] (see Fig. 2.8). SIOV is simply a different expression of the same concept as SR-IOV and has the same trust model, i.e. trusting peripherals to perform isolation and report the correct PASIDs, only using the IOMMU to prevent confused deputy attacks.

2.6.2.3 PRI — Page requests

It may be convenient for a peripheral to request pages from the host on-demand, rather than requiring the host to pin pages in RAM for the peripheral before they're needed. A peripheral

may optionally implement the Page Request Interface (PRI) extension for ATS [57, §10.4], which adds standard Messages for requesting page allocations over PCIe. The overall flow is that a peripheral attempts to translate or access a virtual address V, the host responds with failure because V has no physical page resident in memory, the peripheral sends a Page Request Message to make V resident[11], and then after a successful response from the host V can be translated with ATS as normal. Page Request Messages may be sent in bursts to request many pages at once - a burst of messages are terminated by setting a flag in the last message.

2.6.3 Arm's IOMMU standards

Arm's System MMU standards v1 [82] and v2 [83] are very similar. Both act as IOMMUs, using a 'Stream ID' attached to each AXI transaction to choose which virtual address space the transaction uses, and walking an associated page table to translate the address. These Stream IDs are mapped to 'Context Banks', stored in registers, which contain translation and caching information e.g. the physical address of the page table, which world the transaction needs to be in[12], and the `VMID` field. The latter is stored in the TLB, and allows AXI Distributed Virtual Memory operations to selectively invalidate cached translations with a given `VMID`. Both v1 and v2 support two-stage translation, where two page tables are applied in series (e.g. one for guest-virtual-to-physical translation and another for guest-to-host-physical translation). The main difference is that v1 uses Armv7-compatible page tables, forcing 32-bit virtual addresses, and v2 allows 64-bit Armv8 page tables instead.

Both SMMUv1 and v2 support up to 256 context banks, limited by the stream ID to context bank mapping[13] but in practice Arm's implementations support fewer. The CoreLink MMU-400 and MMU-401 [94, 95], implementations of v1 and v2 respectively, both support 8 context banks. The MMU-500 [96], also implementing v2, supports 128. Another interesting property of the MMU-500 is how it splits the Translation Control Unit (TCU) which performs page table walks from the Translation Buffer Units (TBUs) which cache the translations. Splitting the hardware like this effectively allows for a distributed TLB. One example is the i.MX 8QuadMax (iMX8QM) SoC, which uses a single TCU to control sixteen TBUs—one for each subsystem [97, §3.1.2.2] (Fig. 2.9). Arm's Lookaside Translation Interface [69] (LTI) and Distributed Translation Interface [70] (DTI) specifications codify this split for SMMUv3—LTI connects peripherals or busses to TBUs, and DTI connects TBUs to TCUs.

SMMU version 3.0 [16] released in 2016 to address a crucial bottleneck in v1 and v2. Both prior versions exposed context banks as MMIO registers which "limits the number of context banks" and made it impossible to support e.g. thousands of concurrent contexts [16, p23]. Arm SoCs use PCIe as well as AXI to connect to their peripherals, which requires a Root

[11] How the host makes V resident is implementation-defined. It could pull the page in from disk, or allocate a completely new page.

[12] SMMUv1 and v2 only support TrustZone and not Realms—the `NS` bit may be combined with the Stream ID to give normal-world and secure-worlds different namespaces, but doesn't have to be. The `NS` bit is separately checked in a system-dependent manner to ensure `NS` is only ever 0 when coming from a secure-world peripheral.

[13] The many-to-one mapping allowed an 8-bit context bank index [82, §10.6.23 SMMU_S2CRn], the one-to-one mapping allowed up to 7-bits of index [82, §2.4.3].

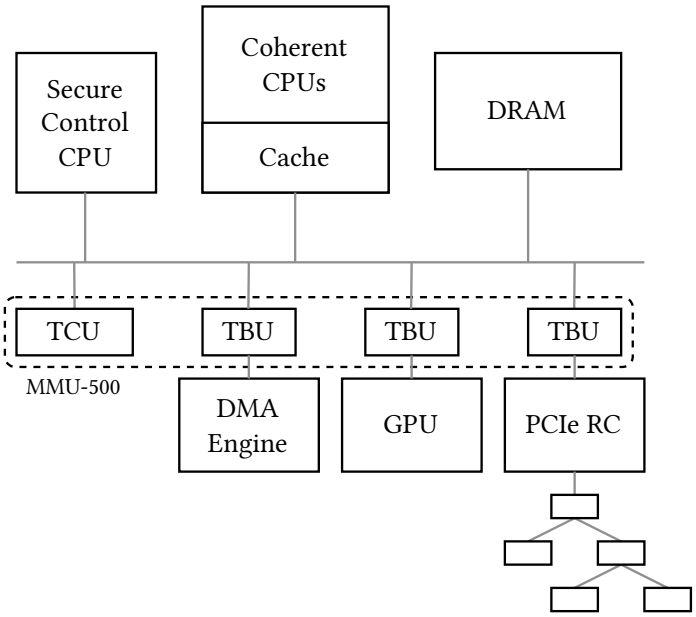


Figure 2.9: The i.MX 8QuadMax protection model, which uses a distributed SMMU to expose SoC subsystems to incoherent memory and other subsystems.

Complex that could handle 2^{16} PCIe peripherals with potentially 2^{20} PASIDs *each*, on top of ATS (Section 2.6.2.1) and PRI support (Section 2.6.2.3). As such, SMMUv3 expanded the Stream ID with enough room to enumerate all PCIe peripherals and added a 20-bit Substream ID to represent PASIDs[14]. On top of that, all interactions with the host are performed in host memory instead of MMIO. The host sends invalidation requests using an in-memory queue, the SMMU sends PRI requests and error information to the host through more queues, and all configuration information is stored in memory.

SMMUv3 can be configured with separate 'Stream Table' pointers for requests in the normal, secure, and realm worlds[15]. Each stream table is a one- or two-level table of 'Stream Table Entries (STEs)', mapping Stream IDs to STEs. Each 512-bit Stream Table Entry contains the physical location of the stage 1 (and optionally stage 2) page tables and other configuration data, notably a bit which enables or disables PCIe ATS for that region. Each STE either represents a direct translation to physical addresses, or represents a two-stage translation—in that case, the STE contains a pointer to a Context Descriptor (CD) which controls the first stage (virtual to intermediate-physical). This is where Substream IDs are used—that pointer could be to a single CD or to a one- or two-level table of CDs, indexed by Substream ID. Once the page tables have been found, they need to be walked. Intel VT-d and AMD use similar multi-level tables to allow per-peripheral PASID-to-page-table mappings [15, §3.4.3][14, §2.2.6]. The world of the request (selected by the Secure Stream ID AXI signal), and a separate StreamWorld specified in

[14]In other contexts the Stream and Substream IDs can be implementation defined. For PCIe, Arm expects that the bottom bits of the Stream ID are directly identical to the 16-bit PCIe Requester ID, optionally with extra top bits to disambiguate multiple PCIe root complexes that use the same SMMU as a translation agent, and that the Substream ID is directly identical to the PASID [16, §3.2]. The Requester ID uniquely identifies a function, which may be physical or virtual via SR-IOV (Section 2.6.2.2) within a PCIe tree [57, p1106].

[15]PCIe-based transactions are forced into the normal-world or Realm worlds based on the presence/absence of IDE (Section 2.3.1.2) encryption and attestation [16, §3.9.4].

the STE, control the world used by the page table walk—each step of the walk must be checked against the granule table if it is present.

2.7 IOMMUs in practice

In theory, the IOMMU provides a flexible security primitive that can be used to fully isolate peripherals and protect hosts against DMA attacks. In practice, the IOMMU has not always been used for that—and when it was, the policies it was used to enforce were not watertight. Worse still, the IOMMU itself has multiple performance flaws that prevent widespread adoption, and security flaws that poke holes in the most secure policies. I examine these flaws, prior mitigation attempts, and suggest a new approach in this section.

2.7.1 Adoption

Standards such as RISC-V have also adopted the IOMMU [98], along with SoC vendors such as Qualcomm [99], Samsung [100], and Apple [101, 102]. All of these models stem from AMD's original design, intended to secure mutually distrusting VMs and user processes from each other, *not* protecting those processes from untrusted peripherals. That is also how the vendors used them. For example, Qualcomm have used custom SMMUs (similar to SMMUv1) to protect their GPUs since at least 2009, but really for the purposes of supporting TrustZone and the benefits of sharing address spaces with user processes [103]—not because their own GPUs were untrusted. Indeed, once attackers took control of the GPU [104] they found the GPU was trusted to take control of its page tables, and gained arbitrary access permissions [105]. That was patched, but then immediately re-attacked by Project Zero to the same effect [106]. Multiple researches in 2017 found multiple over-the-air attacks on Broadcom Wi-Fi chips [107, 7] which could then translate into attacks on the host kernel thanks to the absence of an SMMU [108] or thanks to kernel bugs leading to circumvention of the SMMU [101]. NVIDIA SoCs also use SMMUs to provide shared address spaces [109] under their Unified Memory banner [110, 111]. In general, SoC security policies and threat models did not consider on-chip peripherals to be a threat, and neglected their protection even when IOMMUs were present.

External connections are, of course, also vulnerable to DMA attack. PCIe and other DMA interconnects can be exposed directly on a motherboard, or through more user-friendly connectors such as ExpressCard [112], FireWire [113], Thunderbolt [114], and now USB-C (which incorporates Thunderbolt) [115]. These connectors were all targeted for DMA attack by surveillance companies [116], governments [117], and researchers alike [118, 119, 120, 121, 9] in the 2010s. These attacks depended on IOMMUs not being present or enabled, and commonly used Windows laptops as examples where this is the case. One attack on an Apple MacBook Air, where Intel's IOMMU was enabled, simply disabled it by rebooting into recovery mode and changing boot arguments [121]. Like for mobile SoCs, desktops and laptops considered external DMA attacks to be outside their threat models and generally did not include IOMMUs at all.

Thunderclap [9] used Thunderbolt to attack Windows, Mac OS, Linux, and FreeBSD, and found even when IOMMUs were present and enabled, these operating systems used lax security policies. Windows simply did not enable the IOMMU, except for the 'Enterprise' version which used it to protect an internal VM but left the rest of memory directly exposed. OSX had enabled the IOMMU by default since 2012, but used a lacklustre security policy that gave all peripherals access to the same memory pool—exposing writable function pointers that attackers could use to execute code. FreeBSD fell victim to a similar problem despite isolating peripherals from each other. Linux properly isolated peripherals but did not isolate memory, freeing and reusing memory while still exposed and exposing pages that contained sensitive data along with the intended data. Most grievously, Linux was the only OS at the time to support PCIe ATS. This provided the illusion of security while allowing malicious peripherals to mark accesses as pre-translated and completely bypass protection.

Some of the Thunderclap vulnerabilities have since been addressed. Intel's next revision of the Thunderbolt connector mandated Intel VT-d IOMMU support [122]. Windows now enables the IOMMU by default when it is present and enabled in the BIOS [123], Linux marks Thunderbolt peripherals as untrusted [124] and tries to force-enable the IOMMU [125] and disable ATS [126] for them. Unfortunately the base Linux kernel does not enable Intel IOMMUs by default, though some distributions are experimenting with it [127, 128], and AMD IOMMUs are always enabled when present [129]. Video games present a unique threat model where consumers are incentivised to compromise their own hardware. Microsoft used the IOMMU to protect the XBOX One [130] (and one assumes their subsequent consoles), which was ultimately brought down by a voltage attack at boot [131]. A recent trend among competitive video games to require kernel-level anti-cheat [132], and UEFI Secure Boot [133], has expanded to require IOMMUs [134, 135] which may push adoption further. SD cards can now use PCIe [136], presenting a new and exciting attack vector, so one would hope IOMMUs will become widespread. Unfortunately, the IOMMU is still not always enabled, leading to multiple recent attacks on the host via compromised GPUs [137, 138]—and the IOMMU has other problems that prevent total adoption.

2.7.2 Problems identified

The small-table solutions seen in Section 2.4 and Section 2.5 are stored in SRAM or registers, allowing fast lookup, but limiting their scope. Either only a few different worlds can be encoded (e.g. Arm TrustZone and Realms) or each context is limited to a small number of memory regions (Arm MPU, CXL GFAM). IOMMUs provide improved flexibility—per-PASID-per-peripheral address spaces, with page-granularity translation, that are compatible with native CPU page tables and VMs—by storing page tables in DRAM memory, which is slower to access. This is fundamentally not suited for any scenario where latency matters. Walking a page table is slow—usually requiring at least four memory reads for 64-bit addressing, in a long dependency chain preventing parallelisation. On top of that other machinery, such as Intel's VT-d PASID directory, Arm's SMMUv3 Stream Tables and Context Descriptors, and Arm Realm's granule protection,

Ideal	Practical
(1) Host allocates memory	
(2) Host exposes memory	
(3) Host unexposes memory	(3a) Host requests unexposure [delay]
	(3b) Memory is unexposed
(4) Host frees memory	

Table 2.2: The ideal lifecycle of host memory and the practical reality of the IOMMU and other systems. Under deferred IOMMU invalidation the host is not aware of step (3b).

may require further memory lookups before or during the walk. As already noted, IOMMU specifications include IOTLBs to cache these accesses and improve latency, but unfortunately they do not do enough.

Why? Well, the impact of caching depends on two factors. How many results you can cache, and how often cached results are used. The actual sizes of IOTLBs are generally not made public, and different IOMMUs have different structures and may cache different parts of the walk separately [19], but researchers have experimentally determined useful data points. Experimental results have indicated that Intel Xeon E5-2630 processors (6-core, released in 2012) have an IOTLB that can hold 64 4KiB page translations [18], and Intel Xeon Silver 4316 processors (20-core, released in 2021) support up to 256 [139]. These limit the number of concurrent ops the IOMMU can support before thrashing the cache. Many I/O workloads, particularly storage and networking, have deep queues and do not reuse buffers often enough to benefit from caching. For example, consider a network peripheral with a 512-entry op queue, where each op writes to a different always-exposed page. With a 256-entry IOTLB, each new op would write to a page that wasn't translated in the IOTLB, forcing a page table walk and evicting an old translation to store the new one. That new translation would be evicted before it could be reused by a subsequent op. In short, these workloads have low locality. Farshin et al. found that simply enabling the IOMMU would drop throughput for high-performance networking workloads by 20% [19], even when all CPU bottlenecks are resolved, and directly attributes this to IOTLB misses.

Simply growing IOTLBs to have more entries could help in some respects, potentially performance, but would exacerbate another problem. Caching begets cache invalidation. I/O protection security depends on adhering to the lifecycle established in Section 2.1 and described in Table 2.2. The host must know when memory is exposed and when it is unexposed. Unexposing memory requires the host to clear the relevant page table entries and invalidate them in the IOTLB (and any other caches). Time and again Intel and AMD IOMMUs have been benchmarked and show that individual invalidations can take tens of thousands of CPU cycles, and waiting for them on the host CPU would severely limit throughput [20, 21, 22]. Linux amortizes this cost through *deferred invalidation*. Instead of invalidating immediately, Linux sends a full IOTLB invalidation every 10ms or once any core requests 250 invalidations—removing *all* cached translations in the affected address spaces [20, 21, 22]. This splits step

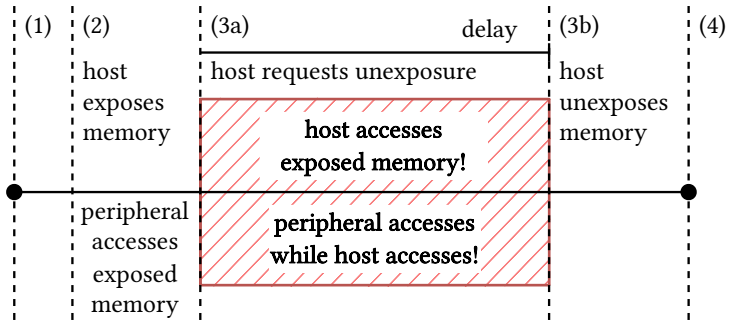


Figure 2.10: If there is a short delay between requesting and actually unexposing the memory, and the host cannot tell when the memory is unexposed, the host may accidentally access the memory concurrently with the peripheral.

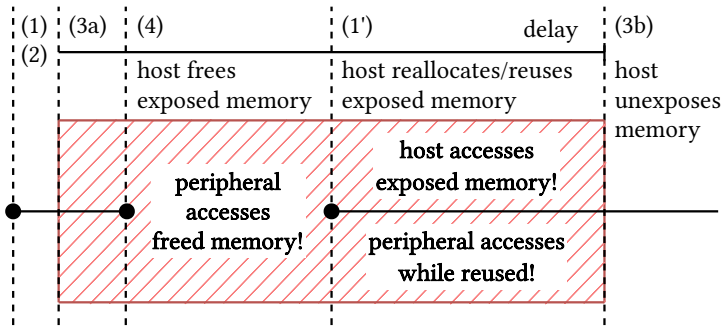


Figure 2.11: If there is a long delay between requesting and actually unexposing the memory, the host may even free and reuse the memory while the peripheral accesses it.

(3) from Table 2.2 into two steps: **step (3a)**, the request, and **step (3b)**, the actual unexposure. From the perspective of the host driver this creates *unexpected overexposure* in the time between (3a) and (3b). During that time the memory is exposed, but the driver doesn't know, which could lead it to e.g. read from a region while it's still peripheral-writable (Fig. 2.10).

Worse, a driver could even free the memory before the memory is unexposed—performing step (4) before (3b), as shown in Fig. 2.11. This could manifest as actual `free()`-ing, where the memory could be reused for any purpose (not just I/O [21, §6.3]), or even transferring memory ownership to another component which isn't exposure-aware. One example is the Linux page cache, which holds data received from storage media. Pages controlled by the cache could be temporarily passed to an NVMe driver, directly exposed, and then passed back before unexposure [21]. In the same way, userspace applications accessing storage with `O_DIRECT` can pass their own buffers to be exposed [140]. The cache/application could then free and reuse the memory while it's still exposed. The only way to avoid passing memory back while it's still exposed is to track and wait for step (3b), which simply would not be performant.

Even when deferred invalidation is enabled, security be damned, performance still isn't good enough for end users. Farshin et al.'s 'IOTLB wall' was measured on a system using deferred invalidation [19], and other experiments testing IOMMU alternatives have found the same throughput issues [141]. Ubuntu once experimented with enabling Intel IOMMUs by default, long after deferred invalidation was introduced, but then disabled them due to "extensive performance issues" [142]. Haas and Leis's popular 2023 NVMe benchmarking

paper disabled AMD's IOMMU during their experiments because "[maximum] performance cannot be reached if the IOMMU is enabled" [143]. Industry postmortems to this day find that the IOMMU creates performance bottlenecks, be they inside the IOMMU or in host CPU drivers, and therefore encourage it to be turned off [144, 127]. This conflation of translation with protection, requiring multi-level page tables and multiple layers of caching, is inherently unsuitable for I/O workloads with low locality that demand fine-grained spatial and temporal safety. 4KiB granularity spatial safety, which some suggest should be discarded in favour of 2MiB hugepages for the sake of performance [19, 141], is not even enough to be safe. The Linux kernel still routinely stores sensitive structures within unrelated exposed 4KiB pages, which could lead to malicious code injection, due to APIs that are difficult to use securely [145]. Translation caching also creates side-channel attacks [139] just as it does inside host CPUs. IOMMUs force users to choose between security and performance, at a performance cost some aren't willing to pay, and doesn't even provide bulletproof security for that cost.

2.7.3 Mitigation attempts

There have been quite a few mitigations proposed for the performance and security flaws of the IOMMU. Some only concern the performance of IOMMUs emulated by hypervisors, or vIOMMUs [146, 147], which this thesis is not concerned with, but that is worth noting as a use-case for the protection system I introduce in later chapters.

In evaluating the IOTLB wall, Farshin et al. suggest using hugepages to alleviate IOTLB pressure [19] and permanently exposing those pages to avoid invalidation cost. Hugepages improve performance, but risk even more accidental exposure of unrelated structures, and constantly exposed hugepages force a weaker security model. These are effectively memory pools, and while they are suitable for some applications, they are less flexible and less useful than directly exposing fine-grained regions.

Similarly, Rubin et al. recommend allocating large contiguous swathes of the I/O virtual address space to ensure the higher levels of the page table walk always hit in cache [141]. These contiguous swathes can then be invalidated all at once by invalidating those higher levels of the page table, without having to individually invalidate each page. This system benefits multi-page mappings (e.g. those larger than 4KiB), but leaves single-page mappings and their invalidations as future work, and is designed specifically for networking where specific access patterns are known. Additionally, the authors make reference to a tech-report with more details on their results, including an analysis showing their protection has temporal safety equivalent to strict IOTLB invalidation, but I could not find it anywhere online. rIOMMU [17] proposes an IOMMU specialized for ring buffers, using a flat page table and automatic invalidation, but like Rubin et al.'s work does not generalize beyond in-order ops.

Markuze et al. have proposed multiple approaches based on copying data in and out of permanently mapped buffers [148, 21]. This stops hostile peripherals from writing into memory while the host is reading and parsing it, but requires extra copying—ideally, peripherals would DMA directly into cache for better performance [59]. Furthermore, Rubin et al. claim they

could not replicate Markuze et al.'s claimed speedups, and note that once the pre-allocated buffers are exceeded new mappings are created as normal and suffer the same performance problems as before.

CryptoMMU [45] uses cryptography to tag translations cached in PCIe ATS, and can invalidate all translations attached to a given (peripheral, PASID) pair by clearing its associated key. This allows untrusted peripherals to safely cache translations in a backwards-compatible manner, but still requires addresses to be translated upon request via page table walk, and it does not handle individual revocation well. A few translations per peripheral can be individually invalidated before *all* translations associated with a peripheral's PASID have to be invalidated, forcing still-valid translations to re-walk the page table, reducing performance. Each PASID requires a dedicated page table, so it wouldn't be suitable to use individual PASIDs per exposed region, and peripherals already expect to use PASIDs for domain isolation—indeed, SIOV relies on using PASIDs this way. Backwards compatibility thus limits what the system is capable of, even if the principle of using cryptography for remote invalidation is sound. I discuss CryptoMMU further in Section 3.3.2 and Section 4.1.

2.7.4 Conclusion

IOMMUs are slow, and I/O workloads are fundamentally difficult to cache. In practice, IOMMU security is also lacking: 4KiB spatial safety risks unintentionally exposing sensitive data, and performance hacks such as deferred invalidation risks regions being unexpectedly overexposed. Even then, these hacks aren't enough for performance-minded engineers to adopt IOMMUs, particularly due to the limits it places on throughput. Prior art has attempted to mitigate these performance problems but end up specializing for specific models or limiting how memory can be exposed.

In order to uphold the principle of least privilege, we need complex security policies, but these policies are both too complex to fit into small SRAM tables and too complex to efficiently retrieve from large DRAM tables. Capability systems, particularly ticket-based ones, allow the storage of these policies to be split out from the enforcement. Endpoints store capabilities and redeem them by passing them to the interposer, which simply checks they are valid. As a bonus, this also ensures capabilities are used with intentionality, avoiding confused-deputy attacks. This is the basis for the IOCap system I introduce with this thesis.

CHAPTER 3

Capabilities & the IOCap system

The previous chapter has shown that mainstream interconnect protection is usually based on conflating translation with protection, which inherently requires a large table-based protection approach. This is problematic both in terms of individual protection checks, which use slow lookups, and revocation, which is not deterministic on mainstream operating systems using these platforms. Network and block storage peripherals (among others) already operate on queues of descriptors [24, 25, 26], where descriptors indicate a region of memory holding data to work on, and these peripherals could naturally extend to use capabilities instead of descriptors. In this chapter I describe a de/centralized IOCap system to address both of those problems.

I begin with a brief overview of capability systems, particularly those that are cryptographically verified, and explain how they are suitable for I/O protection. I then describe the scope and requirements of the IOCap system based on my overall goals, and compare them to prior work. I then describe my proposed IOCap system and its differences and improvements over previous work, particularly focusing on the use of grouped revocation over individual revocation, and check it against the requirements.

3.1 Capability systems

Capabilities are unforgeable *tokens* that grant *actors* access rights to *resources* as determined by the *resource owner*. That's a vague definition, but it needs to be to cover all the different kinds of capability systems built over the years. Computers were using supervisor-controlled (effectively unforgeable) descriptors for access control as early as 1959, with the Rice University Computer, and extended that model to multi-process systems in 1964 with the Basic Language Machine [149]. These descriptors granted processes access to memory. They were first described as capabilities by Dennis and van Horn who described a system where capabilities were stored in a per-process capability list (C-list), controlled by the trusted supervisor, and could only be used by indexing into the C-list [2]. The Cambridge CAP Computer [3] used C-lists to hold capabilities granting access to memory regions, or 'segments', which themselves could

contain other C-lists. A program could have 16 C-lists directly available at any given time, and `ENTER`-ing and `RETURN`-ing from procedures used trusted instructions that manipulated the C-list automatically. Other systems adopted memory tagging from the earlier descriptor-based computers to allow hardware to tell which pieces of data represented capabilities, and which represented plain data, allowing capabilities to be stored in whatever memory locations are convenient [23]. These capabilities were commonly referred to as *tickets*.

Pure C-list models are still used today! POSIX operating systems allow processes to access files by indexing into an OS-controlled file descriptor table. These file descriptors are effectively capabilities, as observed by Shapiro, Smith and Farber [4]. FreeBSD Capsicum [5] expands this system with more granular permissions and an opt-in 'capability mode', which allows processes to sandbox themselves with their current file descriptors, ensuring they cannot gain any more permissions than they need. This is a kind of *monotonicity*, an essential property of any capability system, in that actors cannot broaden the access rights granted by a given capability. File descriptors, like many other capabilities, can be *delegated* between actors: UNIX domain sockets. If you squint, page tables for (IO)MMUs can also be described as C-list systems. Page tables are simply nested C-lists of page table entries, which grant processes or peripherals access to physical memory, which can be shared i.e. *delegated* between different virtual address spaces. One downfall of pure C-list systems is that the C-list may have a maximum size which actors bump up against [150, 151], but even unbounded C-lists have their own pitfalls. (IO)MMUs demonstrate that unbounded C-lists end up being stored in slow RAM, cached in (IO)TLBs, and face cache invalidation penalties. Another problem exclusive to MMUs is spatial granularity—MMUs can only provide isolation at the page level, and cannot provide memory safety for different objects stored in the same page.

CHERI [1] adopts the memory tagging and *ticket* approach to provide lightweight fine-grained memory protection within the CPU. Under CHERI 64-bit pointers are replaced with 128-bit capabilities, which grant access to bounded ranges of memory, with specific permissions such as read/write/execute. The hardware is trusted to maintain 1-bit tags for every 128-bit region, in memory and registers, to distinguish valid capabilities from other data. The hardware ensures only valid capabilities can be used to access memory, and provides specific capability manipulation instructions for software to use. These manipulations all decrease permissions *monotonically*. For example, a read/write capability to an array can be reduced to a read-only capability for a single array element. Any software holding that capability can only read that element, and cannot access the rest of the array. Recent work uses CHERI to isolate libraries within the same userspace process [33, 34] by isolating each library with a separate stack capability and only allowing data transfer by *delegating* capabilities from one library to another. Revocation relies on using trusted software [152, 153, 154] or hardware [155] to scan through memory, invalidating any capabilities that point to freed memory regions.

CHERI capabilities can be used to protect physical memory [155, 156, 157] and they can also compose with other layers of protection like the MMU. Application-class processors can use CHERI to protect access to the virtual address space, and then apply the MMU to translate the

virtual address to physical addresses as normal [158, §10]. This provides the best of both worlds: CHERI provides fine-grained spatial protection inside virtual pages, which can be manipulated cheaply, and the MMU provides page-level translation and process-level isolation, which is heavier. CHERI uses trusted hardware to give untrusted software access to memory, but doesn't allow untrusted hardware to manipulate or most importantly *store* those capabilities.

In order to avoid the failure points of IOMMU-style C-list protection while interacting with untrusted peripherals, I need a ticket-based capability system that doesn't rely on trusted storage. This requires a fundamentally different ticket design to CHERI.

3.1.1 Chained MAC capabilities

Designing a ticket-style capability system simply requires a reliable means to verify if a given piece of data is a valid capability. Under CHERI this verification is implicit in the tag bit. Older ticket-style systems with untagged memory used special numbers attached to each capability. One example is the CAL Time-Sharing System, which used ticket capabilities to index into a Master Object Table. Each entry in the table (each 'object') had a unique name, which could be called a *password*. Each ticket capability had to contain the same password as the object it refers to in order to access it [159]. While this requires space in each capability to store the password, it makes it easy to revoke capabilities to a given object by simply changing or clearing the unique name for that entry.

If storing per-capability identifiers uses too much space another option is to use a Message Authentication Code (MAC). MACs are computed from a plaintext (here, the 'capability text' encoding the access rights) and a secret key. Even if an attacker knows the MAC and the plaintext, they cannot extract the secret key, and they cannot generate new MACs for different plaintexts. Instead of storing a password per-capability, the host can hand out a MAC with each capability, and recompute the MAC on demand to check validity. More formally, consider a capability text C, a secret key K, and a MAC $M = \text{GenMac}(C, K)$. The resource owner gives the attacker one or more (C, M) pairs and the attacker must supply a valid pair with each request. Upon receiving an attacker-controlled pair $(C*, M*)$ with a request, the resource owner can check if $M* = \text{GenMac}(C*, K)$. If it does, $(C*, M*)$ must have been generated by the only actor who knows K—the resource owner—and the capability must be valid. MACs are typically used for communication between two parties with a shared key (or asymmetric key pair), so each party can authenticate the message came from the other, but in this case the resource owner both generates and verifies each MAC.

Revoking these capabilities is as simple as forgetting or otherwise invalidating the key K, just like password-based systems. Those capabilities are revoked no matter where they're stored, even in untrusted peripherals, although if multiple capabilities use the same key they would all be revoked at once. In that case a system may track all outstanding capabilities so they can be revoked individually [160], or allow a few individual capabilities to be revoked before revoking the whole group [45], or just require capabilities to be revoked in groups.

Monotonic reduction can be achieved through chained (or cascaded [161]) authentication. If

an actor who receives (C^0, M^0) wants to modify the capability $C^1 = \text{Reduce}(C^0)$, they generate $M^1 = \text{GenMac}(C^1, M^0)$ and pass (C^1, M^1) to another actor. An actor with only (C^1, M^1) may be able to reconstruct the plaintext C^0, but will never be able to extract M^0 or the original key K, so cannot undo the reduction—it is truly monotonic. When the resource owner receives (C^1*, M^1*), they can check $M^1* = \text{GenMac}(C^1*, \text{GenMac}(C^0*, K))$, and validate the whole chain of reductions. Various distributed systems have used this property for single-level [162, 160] and multi-level [32, 31] reduction.

Macaroons, a multi-level scheme intended for the web [31], allow capabilities to be arbitrarily reduced by appending *caveats*. Appending a caveat recomputes the macaroon MAC[1] as $M^{(i+1)} = \text{GenMac}(C^{(i+1)}, M^i)$. When actors redeem their macaroons the resource owner is responsible for 1) checking the sequence of caveats is a monotonic reduction, 2) ensuring the constraints of the caveats still hold (e.g. if a caveat sets an expiry time, that it hasn't expired), and 3) verifying the MAC. Macaroon-esque capabilities can be stored and manipulated by untrusted parties, and revoked in groups at-will without coordinating with those untrusted parties. This design forms the basis of my proposed IOCap system.

3.2 Scope

The concept of the IOCap system is to enforce memory security on I/O workloads; that are commonly low-locality, high-throughput, and latency-sensitive; using a capability system—one that grants access to contiguous regions of an address space. A ticket-style capability system, like CHERI, would be suitable here. Ticket-style systems put the burden of storage on the system actors (here, peripherals), and only need to verify capabilities on the hot path, which should avoid the pitfalls of list-style systems discussed in Chapter 2. Cryptographically authenticated capabilities would provide low-latency verification and deterministic temporal safety—just throw away the key, and all capabilities using it are invalidated.

The I/O model I presented in Section 2.1, p29 considers two kinds of exposed regions: persistent, such as op queues or shared memory; and ephemeral, small regions with lifetimes tied to individual ops. IOCaps will need to support both cases, and the existence of CXL indicates that the persistent mode should be expanded to include general *bulk sharing*. A host may not just be sharing its own memory, but instead could be *delegating* memory (e.g. from a CXL expansion device) and sharing it with the peripherals it controls. Hence, the system should be based on chained MAC capabilities, as they allow capabilities to be reduced and delegated to other actors. Virtualization, as demonstrated by PCIe SR-IOV and SIOV (see Section 2.6.2.2, p45), is another possible use case. A peripheral may have control over a pool of many separate resources, or be structured as multiple internal compartments, and could delegate IOCaps to specific resources or compartments for reliable isolation. Such isolation could be more lightweight than creating separate virtual address spaces for each isolated workload or task.

[1]The original Macaroons paper refers to the MAC as a "signature" [31]. I use MAC instead to avoid claiming the non-repudiation property associated with digital signatures [163].

3.2.1 Goal

In a sentence: IOCaps should provide fine-grained, reducible, delegatable, deterministically revocable compartmentalization in peripherals. They need to simultaneously support high volume, throughput sensitive workloads, in a way that is still accessible to smaller, low-power, embedded systems. They do *not* try to maintain compatibility with existing peripherals, as I want to explore new frontiers with a clean start, but I do try to remain consistent with existing protocols and OS driver stacks when possible.

3.2.2 Non-goals

Peer-to-peer

Though I refer to the targeted resource as host memory in this model and throughout this thesis, it's important to note it may also be memory or control registers associated with *other peripherals*. This model is typical for SoCs where IOMMUs grant access to the full system interconnect, e.g. the Arm SoC in Fig. 2.9, p48, and IOCaps should be sufficient for peer-to-peer protection here. In contrast, PCIe topologies allow peer peripherals to communicate without consulting the host. This could be resolved by using PCIe ACS to route all peer-to-peer packets through the host (see Section 2.3.1.2, p34) but this would reduce performance—for maximum performance, peripherals would need to expose and check their *own* IOCaps. I believe this should be possible, but I leave it for future work. This also covers cases such as AWS Nitro, where the peripherals are trusted but the host is not [164], and I again believe IOCaps could be easily implemented here but leave it for the future.

Virtual memory

Some peripherals, particularly GPUs, take advantage of IOMMU-based translation to share an address space with user processes. Internal GPUs use SMMUs with shared page tables (see Section 2.7.1, p49) to operate directly on host memory. External GPUs have their own internal MMUs and implement *demand paging* in the host driver to move pages between the GPU and CPU on demand [165]. Demand paging is effectively a short-term DMA op controlled by the host, rather than continuous peripheral access to an exposed region of host memory, which IOCaps can support. However, I do not try to encode translations into IOCaps, so internal GPUs and other peripherals that need page-based translation are not directly supported. IOCaps can layer on top of IOMMU translation to provide protection *within* the virtual address space if necessary.

Cache coherency

CXL, and other cache coherent protocols like CHI and the AXI coherency extensions, may also benefit from IOCaps. There is research interest in securing coherency protocols against malicious actors [10], and I have previously written about CXL's unique position as a protocol

designed to literally provide memory expansion for potentially-untrusted actors [35] (both hosts and other peripherals). However, cache coherency protocols generally reason about smaller regions of memory than bulk DMA, and will send messages more frequently than bulk DMA, so the overhead of attaching IOCaps to each message should be considered. I have included hypothesis facets on 64-bit MACs and IOCaps in future chapters to see if this overhead can be lowered, but otherwise bulk DMA is the priority of this thesis.

3.2.3 Requirements

I first describe basic requirements the IOCap system must meet to provide equal or superior protection to the IOMMU. While there may be overheads associated with some of these requirements, I note them where relevant and leave them for future chapters. This chapter is entirely dedicated to describing the basic system and demonstrating it can adhere to these requirements.

- **It should be cheap to expose new memory to the peripheral.**

  The IOMMU exposes memory by filling in page table entries, and this should be comparable for IOCaps.

- **It must be possible to check IOCap validity in hardware with low-latency.**

  This is the core failure of the IOMMU that I aim to address.

- **It must be possible to deterministically revoke IOCaps with low-latency.**

  IOMMU invalidations can take tens of thousands of CPU cycles on modern systems [22], which forces operating systems to resort to non-deterministic deferred invalidation. Revocation must be deterministic to be truly secure, and must be low-latency for determinism to be practical.

- **It must be possible to expose regions at page granularity or better.**

  This is necessary for equivalent spatial safety to the IOMMU.

- **It must be possible for peripherals and other actors to reduce and delegate capabilities.**

  As discussed above, this may be useful for memory sharing using emerging protocols such as CXL or for internal compartmentalization.

3.2.4 Threat model

Chained MAC capabilities mostly share the basic I/O threat model (Section 2.2, p30) with the IOMMU, but there are some differences. Under the IOMMU, page table memory must be protected—ideally against reads, but most importantly *writes*, because attacker page table writes can override the host's intended security policy. IOCaps (and any cryptography-based system) need to store cryptographic keys, and must ensure they are never *read*. Reading a key gives an attacker enough information to synthesize new capabilities. I discuss this further on p68 in the context of storing keys in memory, but there is another way keys could leak.

Cryptographic systems are commonly attacked via side-channels—for example, an algorithm taking a variable amount of time depending on the key could reveal information about it. Timing side-channels can be addressed by using constant-time cryptography, which is available in popular software cryptography libraries, and easily constructed in hardware. Electrical side-channels, where an attacker monitors the power consumption of cryptography hardware to reveal internal state [166], are more difficult to avoid. Implementations can use "masking" to hide power usage, but robust approaches can dramatically increase latency [167]. However, electrical side-channels are entirely dependent on recovering a statistically significant number of different samples using a single key. This requires between tens of thousands [168] and millions [169, 170] of samples under different plaintexts for a sufficient signal-to-noise ratio. Limiting key lifetime and revoking keys after failed verification attempts will make this recovery much more difficult [171], or even impossible, and adopters should ensure these mitigations are put in place.

IOCaps are not restricted to specific peripherals by design, to allow delegation, so if a peripheral observes an IOCap it can use it. This means peripherals should not be able to observe other peripherals' IOCaps on the interconnect, but in practice this should not be a problem. Chip-level interconnects already rely on snooping protection for confidentiality, and scalability requirements generally force them to be designed such that data will not pass through a malicious peripheral before reaching its destination. The IOMMU already relies on a trusted interconnect, and PCIe IDE already exists as one method of protection when the interconnect is untrusted. At most, this means IOCap metadata should be subject to whatever data snooping protections already exist. In a similar vein, IOCaps should not be stored in exposed memory regions unintentionally—I discuss this further on p69.

3.3 Prior art

Other systems have used capabilities to avoid the problems with table-based protection established in Chapter 2. None of them fulfil all the requirements I have just established.

$$M_{\text{general}} = \text{GenMac}(\text{global key}, C)$$

$$M_{\text{operation}} = \text{GenMac}(M_{\text{general}}, \text{disk operation})$$

Figure 3.1: The chained MAC scheme used by the 2003 network-attached disk system [160].

3.3.1 Snapdragon (2003)

In 2003, Aguilera et al. defined a capability system named Snapdragon for accessing network-attached disks [160]. This system uses a two-level chained MAC. A metadata server hands out a capability and first-level MAC, which allows read and write access to a contiguous region of disk blocks, and clients compute a second MAC of the operation they wish to attempt before passing that to the disk (see Fig. 3.1). The second MAC proves that the client was given permission, i.e. they had knowledge of the first MAC, and that the operation was not tampered with in transit. To avoid overlap with Qualcomm's long-running Snapdragon SoC product line, I refer to this system by the author names or simply as "the 2003 system".

The 2003 system supports grouped and individual revocation tracked through bitfields on the disk. Each capability has a 6-bit group ID, a 64-bit group epoch, and a 13-bit capability ID field. The disk tracks a 64-bit epoch number and a $2^{13} = 8{,}192$-bit capability ID bitmask for each group. Verifying a capability requires verifying the MAC against the global key, checking if the group's current epoch matches the epoch in the capability, and then checking if the bit for that capability ID is still set in the bitmask. Revoking an individual capability requires zeroing the relevant bit in the bitmask, and revoking a group requires incrementing the epoch number. The system assumes that a 64-bit group epoch counter will provide enough space that they will never be reused—if they were, the system would break [160, Footnote 3]. The secret key for all MACs is always the same, and is never used for revocation.

The 2003 system aims to both protect the disk from malicious clients and to protect clients from each other. The MAC secures the access capability *and* the operation a client tries to perform with it, which is laudable, but is not part of my threat model. Their revocation mechanism is intriguing, and the concept of capability groups is an excellent one, but the implementation comes with constraints. The number of groups is quite small (a system may well have 30–60 peripherals, allowing one or two groups each) and the per-group bitmask limits the maximum number of exposed capabilities and the minimum storage overhead. A group that needs fewer than 2^{13} capabilities will still use the whole bitmap, when those bits could be used to encode more groups or larger groups. Even in a modified system that used a different cryptographic key per group per epoch, obviating the group epoch field and effectively merging it into the MAC, a capability would still need to store a 13-bit capability ID on top of the group ID—which may be infeasible if space is limited. Bespoke IOCap systems with known, reasonable group size limits could use this system for individual revocation *if* those limits are acceptable in practice, but I consider that out of scope.

$$M = \text{Hash}(\text{Key}_{\{\text{PeriphID,PASID}\}}, \text{PPN} \parallel \text{R/W} \parallel \text{VPN})$$

Figure 3.2: The MAC scheme employed by CryptoMMU [45].

3.3.2 CryptoMMU

CryptoMMU [45] is a proposed scheme to extend the PCIe ATS virtual address caching system with cryptographic verification. Instead of trusting peripherals outright to store cached virtual-to-physical translations, CryptoMMU attaches a 25-bit or 56-bit MAC to each cached translation that gets checked whenever the translated address is used (effectively turning it into an irreducible capability). These capabilities are not self-describing—each peripheral is expected to maintain extra state mapping virtual addresses to MAC-verified-translations—but the advantage of that approach is that cached translations can be invalidated without having to reissue all the relevant capabilities up front. Like ATS, as long as the peripheral knows a translation is bad, it can request a new one when relevant. CryptoMMU calculates the MAC for each virtual-to-physical translation by hashing a combination of the physical page number (PPN), the read/write permissions, and the virtual page number[2] (VPN); with a per-peripheral-per-address-space-ID key (Fig. 3.2).

CryptoMMU can revoke a limited number of individual translations before having to refresh the key for a (peripheral, address space) pair. It sets aside a small amount of memory for each device to store a list of invalidated translations, blacklisting those specific translations from verifying, then once that list is full the next invalidation results in all relevant keys being refreshed—invalidating all their translations and forcing a re-walk of the page table for each one. This list is small, in their evaluation the list has four entries per peripheral, but the authors assert that invalidations are rare[3] and as such this impact is negligible. I have already established in Chapter 2 that this is not the case: I/O invalidations are frequent and harm performance on systems that aren't prepared for it [21, 19]. Any CryptoMMU-esque approach would need a much larger key space to handle the amount of invalidations those peripherals require (discussed further in Chapter 4).

[2]The virtual page number is not strictly necessary, but the authors claim it has the same function as a nonce in ensuring MAC freshness. While its inclusion may make it harder for an attacker to extract the MAC key, it isn't a random number nor is it host-controlled—unless each verification includes a reverse lookup, the virtual page number would have to be included in the untrusted metadata provided by the attacker—so it may prove detrimental in practice.

[3]They cite research into CPU-side TLB shootdowns, which have very different security implications and invalidation properties than IOTLBs, and model "200 invalidations per peripheral per second" based on simulations from the prior art 'Border Control' [172]. Border Control, which was published in 2015, outright states their workloads "never encountered a permission downgrade" (their wording for unmapping or unexposing regions), and only measured 200-per-second due to "context switches". Border Control models peripherals as accelerators executing long-running processes, context switching like a CPU as needed, where memory is infrequently or even never unmapped. This perspective, which CryptoMMU seems to have inherited, is not accurate to modern workloads.

3.3.3 CHERI

Of course, CHERI has also thrown its hat into the ring. The CHERI Position paper [44] from 2020 considers DMA protection from many angles under multiple threat models, including peripherals with both trusted and untrusted hardware. Naylor et al. extend CHERI to GPGPU architectures as an example of the former, where the GPGPU is trusted to modify and store capabilities [29]. Cheng et al. implement a 'CapChecker' unit for untrusted hardware, an example of the CLUT (Capability LookUp Table) from the Position paper, which replaces the IOMMU with a small C-list of CHERI capabilities. The CapChecker is entirely host-controlled, so peripherals cannot manipulate or delegate capabilities, but the benefit of this approach is the integration with the host CHERI protection. Any host process with a CHERI capability can grant a peripheral access to that memory, and a host process can *only* grant a peripheral access using a valid CHERI capability. The one snag is that CHERI protection is typically relative to a virtual address space, and requires either pre-translation into a physical space (as for the CapChecker[4]) or integration with an IOMMU. While the latter hasn't been implemented yet, the Position paper notes that the IOMMU could benefit from sticking to translation (which changes rarely), without managing protection (which changes extremely frequently).

3.3.4 Others

The Amoeba distributed operating system [162] uses MAC-based capabilities to validate permissions. A server hands out an all-permissions capability with a random key in a 'check field'. A client removes specific permissions by zeroing out bits, and returns the capability to the server—where the random data in the check field is XORed with the permission bits, passed through a hash function serving as a MAC, and placed back into the check field. This reduced capability is then returned to the client and can be delegated to others, and when those others redeem the capability the server recomputes the check, ensuring that those others could not increase their permissions. This is quite similar to the concept of Macaroons and other chained MAC systems, albeit where exactly one caveat is encodable, where capabilities protect objects and not regions of memory, and where the check field/key is unique per-object.

Northcape [173] constructs a centralized capability table that controls all accesses to physical memory, host and peripheral, and searches this hash table on every access. Though the implementation has not been completed at time of writing I predict it will hit the same limits as other centralized C-list endeavours.

[4]Personal correspondence with the authors indicates that the CapChecker relies on creating virtual address mappings that match the physical address space, allowing the host's virtually-addressed CHERI capabilities to be interpreted as physically-addressed by the CapChecker.

3.4 My proposed IOCap system

My proposed IOCap system uses chained MAC capabilities to restrict accesses by peripherals to host memory. Each capability, called an IOCap, exposes a contiguous region of memory to a peripheral. This region is typically expressed in the physical address space, though it could represent a virtual address if an IOMMU was employed for translation. IOCaps are entirely self-describing, i.e. the peripheral does not need to store any state beyond the IOCap to use it, and are intended to fill the role of buffer descriptors that are ubiquitous in peripheral systems today (see Section 2.1). This is not a completely transparent change, as peripherals will need to adapt to understand the IOCap format, but this comes with the freedom to improve the format beyond just a pointer and a length. Peripherals can copy IOCaps, reduce them, and pass them on to other peripherals while ensuring security. Most importantly, IOCaps provide fast, deterministic revocation, even for high-turnover scenarios.

Unlike CryptoMMU and the 2003 system, IOCaps rely solely on grouped revocation. Each IOCap is associated with a lease, where each lease has a unique secret key, and leases are revoked by overwriting/clearing their key. There is a many-to-one IOCap-to-lease relationship, and the individual IOCaps under a lease are not tracked, which allows a huge volume of data to be shared at any given time while keeping the host-side state manageable. That being said, IOCap systems use far more random keys than those other systems. My approach is similar to how the 2003 system gave each group an epoch number, where revoking a group incremented the epoch number, and each capability encoded a group ID and epoch [160]. IOCaps give each lease a secret key, where revoking a lease refreshes that key with random data or clears it with zero, and each capability encodes a lease ID and a MAC. The MAC lookup simultaneously verifies the capability text is authentic *and* that it was generated under the same "epoch" (i.e. secret key value) that is currently in use.

IOCaps are an abstract design that can be applied to many different systems, like how CHERI can be applied to many different CPU architectures. A block diagram is shown in Fig. 3.3—note that the interposer has separate lease manager and enforcer components. The lease manager stores a mapping of lease IDs to secret keys, and exposes lease management operations to the host over *memory-mapped I/O* (MMIO). The enforcer intercepts requests, reads secret keys from the lease manager, and verifies the MACs. This abstract design has three parameters: the interconnect, the MAC function, and the IOCap format. The following subsections describe the system and parameters in more detail.

3.4.1 Grouping IOCaps under leases

When a driver wishes to expose memory to a peripheral, it asks the OS to open a lease. Leases are not shared between drivers, and each driver may have many leases open at any given time. Opening a lease requires selecting a suitable lease ID, which indicates a slot in the lease manager's SRAM, and uploading a secret key into that slot—this secret key is used by the host to mint the IOCaps that expose memory, and by the enforcer to verify incoming accesses. Each

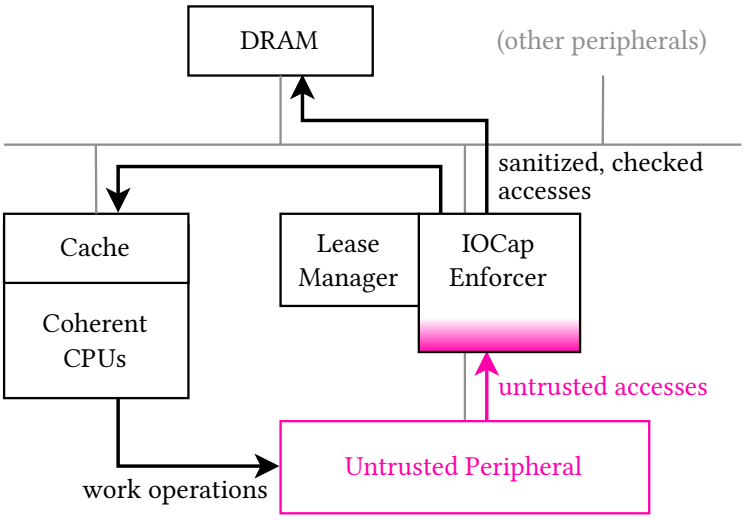


Figure 3.3: A block diagram of the IOCap architecture. Work operations are typically sent via the system bus without passing through the interposer, which this diagram simplifies for readability.

time an op is completed, the driver asks the OS to revoke that op's memory, triggering step (3a) of the lifecycle (Table 2.2, page 51). Once step (3a) has been triggered for all memory exposed through a lease, the OS sends a single message to the lease manager to trigger step (3b)—closing the lease by forgetting the key. The OS then informs the host driver that the memory is unexposed and safe to access, free, or reuse. The lease ID can also be reused once the driver uploads a new secret key. Ideally lease IDs are pre-allocated to drivers at boot and constantly revoked and reused.

This model allows drivers to take advantage of domain-specific knowledge of which regions are likely to be revoked close together, minimizing the delay between steps (3a) and (3b), while still amortizing the inherent latency of communicating with the interposer. There is a many-to-one IOCap-to-lease relationship, and the individual IOCaps under a lease are not tracked by the hardware interposer, which allows a huge volume of data to be exposed at any given time while keeping the hardware state manageable. Lease keys can be stored in fast, local SRAM in the hardware interposer which allows fast lookup and fast clearing. MAC-based capabilities are also quick to check. Previous research [45] estimates latency at 10ns, or 10 interposer cycles assuming 1GHz operation[5], which matches my findings in Chapter 7.

Even though leases are usually controlled by a single driver, they are not tied to specific peripherals, so IOCaps can be reduced and passed between them without consulting the host. I consider this a feature. Monotonic reduction and delegations are incredibly useful properties in other capability systems such as CHERI. The sheer prevalence of dedicated accelerators, even small ones, demonstrate a use case for delegation in I/O. If data has to pass through multiple accelerators, in whole or in part, the system may benefit from those accelerators passing that access around without having to consult the host.

[5]The CPU, the interconnect, and the interposer between them may run at different clock frequencies. Throughout this thesis I assume CPUs run between 2–5GHz, interposers built in silicon run at 1GHz, and all cycles are interposer-cycles unless otherwise stated. On flit-based interconnects like AXI the interconnect clock should match the interposer clock, but on serial interconnects like PCIe it doesn't make sense to conflate the two.

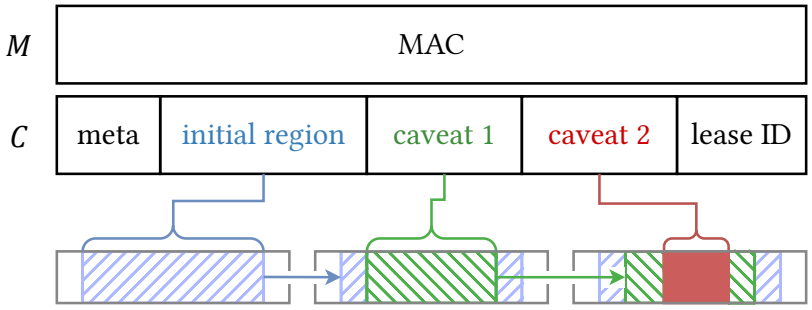


Figure 3.4: The components (C, M) of a two-caveat IOCap, including the core fields in C. The initial accessible memory region can be reduced by caveat 1, and then further reduced by caveat 2.

3.4.2 The abstract IOCap format

IOCaps serve both as a ticket, granting access to a contiguous region of an address space[6], and a buffer descriptor that peripherals can decode and manipulate. That requires a consistent format known by the host and peripherals. This section describes the basic features that a format must have, which I elaborate on in Chapter 5. Different circumstances (e.g. embedded vs. server systems) that use different peripherals may have specific size or latency requirements which affect their choice of format. I consider deciding on a specific one-size-fits-all format out of scope.

Each IOCap is a chained MAC capability, as described in Section 3.1.1, consisting of a pair (C, M) of the capability text and the capability MAC respectively. These are shown in Fig. 3.4—note that they are the same size, and the data for caveats is embedded inside the capability text instead of dynamically growing the capability text. Fixed-size formats are resistant to message extension attacks on the MAC, bound the amount of time and resources required for verification in hardware, and are easier to send on flit-based interconnects. I attempt to fit in as many caveats as possible, to mimic CHERI's infinitely reducible capabilities, while making each one precise enough to be useful.

Each MAC is used as the secret key for the next reduction, so they need to be large to provide sufficient security. In practice I have found that a capability text with room for two caveats must be at least 128-bits long, and 128-bits is also a generally accepted minimum length for MAC secret keys [174, Table 4]—though future formats could use different sizes. I found this was enough to fit two caveats along with the rest of the fields, hence the choice of only two caveats throughout the rest of this chapter. Chapter 5 goes more in depth into my choices in practice, including the choice of MAC function and the cryptographic requirements involved.

The capability text is split into fixed-size fields. The initial region field encodes the initial contiguous region of memory accessible with the IOCap, and the caveat fields successively shrink that accessible region. The lease ID field indicates which lease the IOCap is under, i.e. which key was used to generate the MAC. The meta(data) field encodes which caveats are active and which permissions (read/write) are granted for the region. It doesn't just include the

[6]IOCaps do not make any assumptions about address spaces, and could layer on top of e.g. coarse-grained IOMMU translation, but my prototype system uses untranslated addresses.

current permissions, but also the history—was the IOCap always read only, or was it read-write and then reduced to read-only by caveat 1, or was it read-write and reduced by caveat 2? This information is essential in order to accurately reconstruct the chain of capability texts. If the IOCap was initially read-write then the initial MAC, and the rest of the MACs in the chain, will be different than if it was initially read-only. Caveat fields are defined to be zero when unused, again to ensure each capability text in the chain can be accurately predicted.

3.4.3 Creating IOCaps

The system starts with an empty lease manager ensuring all accesses are blocked at boot. Once the host wishes to expose a memory region to a peripheral—most commonly associated with a task to complete e.g. transmit this network packet, perform a computation on this data—the host must mint a new IOCap for that region under some lease. The host is responsible for determining which lease ID is most appropriate for each IOCap based on the expected lifetime of the exposed region. The process is complex, as each peripheral may have multiple leases open at once for regions with different lifetimes. Once the host has selected a lease ID, it must activate it, by randomly generating a key[7] and uploading that key to the lease manager using MMIO.

Minting an IOCap requires setting the metadata, initial region, and lease ID fields; then computing the MAC from the lease key and the capability text (see Fig. 3.5). Once the IOCaps have been created, they can be passed to the peripheral and used until their key's data in the lease manager has been cleared.

Storing keys in host memory

The host is trusted to keep lease keys in memory, to make minting convenient, but it must be vigilant against attacks on this memory just like those on IOMMU page tables [101]. It's critical to use host-facing memory safety such as CHERI [27, 28], Arm MTE [175], and memory-safe languages [176], inside the kernel alongside peripheral-facing systems to effectively mitigate these exploits [177, 178]. This includes mitigations for Rowhammer-based attacks [51], which would only randomly corrupt stored keys if targeting IOCaps directly, but could be used to compromise host-facing memory safety [50]. If this is truly unacceptable the lease manager could be repurposed to generate MACs for arbitrary IOCaps, but this could easily bottleneck highly parallel hosts and overcomplicate host drivers. I discuss this further in Section 6.1.3, p138, but I have not thoroughly investigated it so I leave it as future work.

3.4.4 Reducing and delegating IOCaps

Any participant, host or peripheral, can shrink received IOCaps using the caveat fields. Adding a caveat is as simple as filling in the caveat field, updating the meta field to indicate the caveat

[7] While entropy can be gathered through user I/O post boot, it can also be gathered in hardware before any user input is required.

is used (optionally also restricting permissions), and then calculating the new MAC from the old MAC and the new capability text (see Fig. 3.6). These shrunken IOCaps are said to be *derived* from the *parent* IOCap minted by the host, and can only grant access to a contiguous sub-region of the parent's region (see Fig. 3.4).

Derived IOCaps can be delegated to other peripherals, or to hardware units within the original peripheral, to compartmentalize them with only those sub-regions. Derived IOCaps can be created without consulting the host, aren't centrally tracked, and thus cannot be individually invalidated. The only way to invalidate a derived IOCap is to invalidate its parent by clearing the key, which invalidates *all* other IOCaps—parent or derived—using that lease.

How delegation impacts trust

When the host passes an IOCap to a peripheral it is granting the peripheral permission to *access* memory via the IOCap, to *delegate* the IOCap to other peripherals, and to *store* the IOCap in peripheral-local memory. Any peripheral may be malicious or buggy. I assume the host will only give IOCaps and permissions to peripherals that absolutely need it, accepting the risk that the peripheral may misuse them. For example, if the host connects to a peripheral

```
lease_id  = ...
lease_key = ...

# Ensure the key manager knows about the lease
# before using it
key_manager.activate_lease(lease_id, lease_key)

C0 = IOCap(
    # The metadata encodes the full chain of permissions
    # *and* which caveats are enabled.
    metadata=(
        # Initial permissions are read/write
        READ | WRITE,
        None, # caveat 1 is disabled
        None, # caveat 2 is disabled
    ),

    # The initial region is always set
    initial_region=(0x40_0000, 0x40_8000),
    # caveat1 and caveat2 are not used, so zero.
    caveat1=0,
    caveat2=0,

    lease_id=lease_id,
)
M0 = GenMac(key=lease_key, text=C0)
```

Figure 3.5: Minting an IOCap `(C0, M0)` in Python-esque pseudocode.

```
C1 = IOCap(
    metadata=(
        # initially read+write,
        READ | WRITE,
        # caveat 1 read+write,
        READ | WRITE,
        # caveat 2 unused
        None
    ),

    initial_region =
        (0x40_0000, 0x40_8000),
    # caveat1 valid
    caveat1 =
        (0x40_1000, 0x40_5000),
    # caveat2 unused
    caveat2=0,



    lease_id=C0.lease_id
)
M1 = GenMac(key=M0, text=C1)
```

(a) Caveat 1

```
C2 = IOCap(
    metadata=(
        # initially read+write,
        READ | WRITE,
        # caveat 1 read+write,
        READ | WRITE,
        # caveat 2 read-only
        READ,
    ),

    initial_region =
        (0x40_0000, 0x40_8000),
    # caveat1 valid
    caveat1 =
        (0x40_1000, 0x40_5000),
    # caveat 2 shrinks the region
    # and perms at the same time
    caveat2 =
        (0x40_4000, 0x40_4051),

    lease_id=C1.lease_id
)
M2 = GenMac(key=M1, text=C2)
```

(b) Caveat 2

Figure 3.6: Applying caveat 1, then caveat 2, in Python-esque pseudocode. Each caveat can shrink the permissions, the region, or both. No information from the host is required, so peripherals can do this too.

that performs network accesses, and passes it an IOCap for network-relevant data, I assume it accepts the risk that the peripheral may misuse that data by e.g. logging it elsewhere. IOCaps only ensure that *if* that network peripheral was malicious, it couldn't *also* extract arbitrary data from host DRAM. This risk extends to delegation.

If a peripheral delegates an IOCap to a second peripheral which misuses it, that is equivalent to misusing the data directly and already incorporated into the assumed risk. The host must also be aware of the risk of *unintentional* delegation. Even if the host does not accidentally pass IOCaps to peripherals, if a peripheral stores them insecurely a second peripheral could exfiltrate them from the first peripheral's local memory with a P2P DMA. This only applies to peripherals which expose their local memory to other devices and can be mitigated by configuring the host security policy (whether through IOCaps or other mechanisms like PCIe ACS) to limit P2P accesses to such memory. Alternatively, a future IOCap system could lock lease IDs to specific devices, which would make delegation—accidental or not—impossible for those leases. I leave that investigation to future work.

3.4.5 Checking IOCaps

The enforcer validates each untrusted incoming request by detaching the associated IOCap $(C*, M*)$, looking up the key for the lease ID inside the lease manager, and validating $M*$ using that key. It's the only component that can read keys out from the lease manager, and if the key for the given ID is invalid the validation will always fail. If the capability has been caveated e.g. C^2*, this reconstruction process involves computing the previous un-caveated capability texts C^0* and C^1* to get the correct MAC (see Fig. 3.7). Successfully validated requests get forwarded up to the rest of the system with the IOCap stripped off.

```
# Given attacker-provided C and M
(C2*, M2*) = ...

# C0 can be reconstructed from C2*,
# M0 can be reconstructed from C0 and the lease secret
C0 = IOCap(
    metadata = (C2*.metadata[0], None, None),
    initial_region = C2*.initial_region,
    caveat1 = None,
    caveat2 = None,
    lease_id = C2*.lease_id
)

lease_key = key_manager.read_key(C2*.lease_id)
M0 = GenMac(key=lease_key, C0)

# C1 can be reconstructed from C2*,
# M1 can be reconstructed from C1 and the guessed M0
C1 = IOCap(
    metadata = (C2*.metadata[0], C2*.metadata[1], None),
    initial_region = C2*.initial_region,
    caveat1 = C2*.caveat1,
    caveat2 = None,
    lease_id = C2*.lease_id
)

M1 = GenMac(key=M0, text=C1*)

# M2 can be reconstructed from C2 and the guessed M1
# to verify M2*.
# Check key validity here to avoid timing side-channels.

M2 = GenMac(key=M1, text=C2*)
assert M2* == M2 and lease_key.is_valid
```

Figure 3.7: Verifying an attacker-provided `(C2*, M2*)` using the lease key in Python-esque pseudocode. This code assumes lease invalidation timing to be a side-channel, and thus avoids it out of an abundance of caution, but I do not think this is necessary in general.

3.4.6 Revoking IOCaps

Once all the memory regions exposed under a lease have been consumed by the peripheral, e.g. the ops using those regions have finished, the host can mark that lease as invalid in the lease manager. This invalidates all the associated IOCaps—if a peripheral tries to use an IOCap after its lease is invalidated, the enforcer will reject it. After a lease is invalidated, it can be refilled with a new random key and the ID can be reused for new IOCaps. Even if a new IOCap uses the same lease ID as an old IOCap, the new lease key will not generate the same MAC values[8] so stale IOCaps will never pass validation.

3.5 Summary

In this chapter I explored the prior art of capability systems, derived requirements for a new I/O focused system, and proposed the IOCap system to fit them. It does, in fact, meet all the requirements:

- **It should be cheap to expose new memory to the peripheral.**

  Creating a new IOCap requires finding a lease to use, opening it if necessary, and minting the IOCap. If the lease to use is already open and known, minting the IOCap itself only requires computing the capability text C and the MAC function—the capability text compression is not trivial but also not very complex, as shown in Chapter 5, and MAC functions can be accelerated with CPU cryptography instructions.

- **It must be possible to check IOCap validity in hardware with low-latency.**

  Looking up lease keys should be fast, as they are stored in local SRAM inside the lease manager, and validating IOCaps should also be fast as long as a good MAC function is chosen.

- **It must be possible to deterministically revoke IOCaps with low-latency.**

  Revocation requires closing the relevant lease, which is stored inside the lease manager. Chapter 7 shows that this can be done using a single MMIO write to the lease manager, which should be about the same cost as a last-level cache write if it lives on the CPU I/O die.

- **It must be possible to expose regions at page granularity or better.**

  The abstract format is 128-bits long and encodes the initial region, two caveats, and the lease ID. A simple encoding of a 52-bit base address and 52-bit length can encode an arbitrary page-granularity region in the 64-bit address space with fewer than 128-bits, and Chapter 5 describes how caveat encoding and sub-page granularity can be achieved in even fewer bits.

[8] The chance of a MAC collision, where two different IOCaps have the same MAC under different keys, is the same chance of an attacker guessing the key at random—see Section 4.3.2, p80.

- **It must be possible for peripherals and other actors to reduce and delegate capabilities.**

  The system allows two levels of reduction and does not tie lease IDs to specific peripherals, allowing arbitrary delegation. Closing a lease deterministically revokes all attached IOCaps, including those reduced by other actors.

Now, the glaring questions are 1) is the lease system feasible in practice, 2) what is a suitable MAC function, and 3) can the initial region and two caveats all be compressed into 128-bits with reasonable precision? Chapter 4 investigates 1) and 2), Chapter 5 handles 3), and Chapter 6 and Chapter 7 implement an example IOCap system using those answers.

CHAPTER 4

Temporal safety with leases

Temporal safety is a key failure of the IOMMU, due to long invalidation times that lead hosts to unexpectedly touch exposed memory, and it is core to the value proposition of IOCaps. IOCaps are affected by both the security of the selected MAC function and, most crucially, the scope of the lease system. Unlike prior systems, the cryptographic authentication entirely controls the lifetime of exposed regions, so ideally as many leases would exist as exposed regions with unique lifetimes—but what happens when that is not the case? This chapter lays out the case for the lease system, estimates the upper bound for the number of leases a system might need, determines if it is possible for a system to directly expose that many leases in hardware, and evaluates different techniques for remaining secure in underprovisioned systems. Along with that, it discusses a range of candidate MAC functions and selects one that will be implemented throughout the rest of this thesis.

4.1 Why multiple leases?

Over the course of this project, some people familiar with prior art have asked how IOCaps are different. In particular, they have been confused why multiple leases are necessary—CryptoMMU, which also uses cryptographic validation for I/O safety, only uses one key per {peripheral, address space} pair [45]. Ultimately this comes down to *lifetimes*.

Host software intends for regions to only be exposed to peripherals for a certain time period. If regions are exposed per-op, they should each be unexposed once their individual op finishes and the peripheral stops needing to use them. The IOMMU handles this by removing mappings from the page table, and sending an IOTLB Invalidation (IOTLBI) to invalidate caches if necessary. Unmapping one region does not affect the other regions the peripheral is using. Even if the entire IOTLB is trashed, the regions remain exposed—only performance is affected.

CryptoMMU allows peripherals to *cache* IOMMU mappings using cryptographic authentication. The IOTLB is repurposed to store authentication keys and per-peripheral *invalidation buffers* (lists of revoked mappings). Peripherals can use cached mappings directly, which are authenticated using the relevant keys (assuming they haven't been spilled to DRAM) and

checked against the invalidation buffer. Once the invalidation buffer for an accelerator is full, the key is revoked, functioning as a remote IOTLBI. This does not break correctness because the other mappings, that haven't been revoked, remain in the page table. The peripheral does not need to ask the host to re-expose regions, instead it makes normal requests and walks the page table again—just like compulsory misses in a conventional IOTLB, and with the same performance loss. CryptoMMU argues that invalidations are rare, citing research on CPU TLBIs, but as Chapter 2 has shown this is not true for high performance I/O.

IOCaps are fundamentally different: each IOCap represents the *only* way a peripheral can access a given exposed memory region. Once an IOCap is invalidated, i.e. once the lease it's associated with is closed, there is no other structure like a page table to fall back on. The only way to regain access to a region once the IOCap has been invalidated is to coordinate with the host to open a new lease, mint a new IOCap, and send it—a much more involved process than CryptoMMU. If all the IOCaps for a peripheral were under the same lease, revoking that lease would force the peripheral to request new copies of all the IOCaps that it still needed. Instead, using separate leases for IOCaps with different lifetimes allows regions to be independently revoked without affecting each other.

4.2 How many leases?

First, we need to understand how many leases a system would need to have open concurrently. One may suspect that this would be a prohibitively large number, perhaps in the millions, but I argue that in practice even systems that expose a huge number of regions *over time* are not exposing that many regions *at any instant in time*. I reason about this for the two contexts mentioned in Section 3.2, p58: bulk sharing, where the host or some other memory device is handing out large swathes of memory for peripherals to use as queues or scratch space etc.; and per-op sharing, where a system needs to perform many bespoke ops and hands out individual IOCaps for each one. The primary difference is expected lifetimes: large regions are less likely to be exposed and revoked quickly, whereas individual ops should relinquish access to memory as quickly as possible.

4.2.1 Sharing persistent regions

Large memory regions are likely to be pools of memory, dedicated to one or more peripherals, and infrequently revoked or moved. Assuming the set of persistent regions does not change often, a single lease could be used to expose all of them for a single peripheral (or a few leases reusing a small pool of IDs using an epoch system, as I describe in Section 4.5.1.1, based on prior literature in garbage collection [179]). This implies that the number of concurrent leases is roughly equal to the number of total peripherals—counting virtualized peripherals, created with PCIe SR-IOV or SIOV (Section 2.6.2.2, p45), separately. SIOV users in particular are aiming to handle up to a thousand peripherals [180], so 8,000 may be a sensible (albeit pessimistic) target to aim for. CXL's GFAM memory expansion devices (Section 2.4.3, p38) are built to

expose eight regions of memory per attached host, and could easily transition to this model by exposing eight leases (and many more IOCaps) per host instead, which would be a significant improvement in terms of flexibility.

4.2.2 Sharing ephemeral regions

High-frequency ops such as network packet transmit/receive and storage read/write are numerous, have individually unique lifetimes, and are performance-critical—the most demanding scenario for the lease system. I can estimate the number of leases required for a set of peripherals from their I/O Operations Per Second (IOPS), which is a common way to measure performance for NVMe solid-state drives (SSDs). Haas and Leis found in 2023 that their PCIe 4.0 Samsung PM1733 SSDs could hit 12.5 million total IOPS under a random-read benchmark using 4KiB pages [143]. They achieved maximum throughput using an I/O depth of 3,000 concurrent ops across eight drives, i.e. with 375 outstanding requests per SSD at any given time. As a contingency, I round it up to $512 = 2^9$, which would also include other regions such as the queue buffers themselves.

I assume that SSDs are a rough proxy for other high-bandwidth peripherals with multi-kilobyte DMAs, such as network cards, to avoid the complexity of characterizing many different accelerators and applications. AMD's highest-end server CPU at the time of writing, the EPYC 9965, supports up to 128 PCIe 5 lanes per socket [181], and we assume each peripheral takes 4 lanes for a total of 32 peripherals per socket (and one I/O hub per socket). Multiplying these together gives the maximum number of leases per I/O hub as 16,384. This estimates a physical limit on total number of concurrent exposed regions, independent of any virtualization the system could be employing.

$$\begin{aligned} \text{outstanding regions} &= \overbrace{512}^{\text{queue depth}} \times \overbrace{32}^{N_{\text{Peripherals}}} \\ &= 16{,}384 \text{ or } 2^{14} \end{aligned} \tag{4.1}$$

4.3 The MAC function and its impact

There are many algorithms to generate MACs with various strengths and weaknesses, usually parameterized by an underlying cryptographic block cipher [182, 6] or hash function [183]. The MAC function is a key parameter of the IOCap system, and the chosen function must fulfil the following requirements:

- The output MAC size must be greater than or equal to the key size.

  This is necessary to allow IOCap derivation with chained MACs.

- The entire MAC state must come directly from the IOCap, so there cannot be external dependencies on hidden state such as nonces or sequence numbers.

This is necessary for derivation and verification. It must[1] be possible for a peripheral with no knowledge other than an IOCap to generate a derived IOCap, so the derivation and subsequent MAC calculation can't depend on e.g. sequence numbers synchronized at the host. Conversely, it must be possible for the host to rederive the MAC chain when it receives the IOCap, so the MAC calculation can't depend on e.g. nonces that only the deriver knows.

- The MAC should use NIST-approved cryptographic primitives.

  I choose this restriction purely for credibility. NIST controls the encryption standards used by the US government, and as such I consider their approved primitives to be rigorously tested and (hopefully) uncontroversial. Reusing MAC outputs as keys is already an unusual construction, though I remain convinced it is sound, and using niche or alternative primitives would only raise eyebrows further. Any future IOCap adoption is welcome to choose other primitives as long as they are secure.

Macaroons, the inspiration for this system, was built for the web and used URL-safe encoding for the capability text [31]. They chose HMAC-SHA-256 as their MAC function, which is suitable for relatively long plaintexts (e.g. SHA-256 pads messages to a multiple of 64-bytes), and produces a 32-byte MAC [184]—too large to represent access to a single memory region when combined with a capability text. For example, NVMe commands are 64-bytes long and encode multiple memory regions and many other fields [24, §4.2]. Considering the most generic insecure representation is simply two 8-byte words, a base and a length, any IOCap encoding can't go far beyond that without causing prohibitive memory overheads in queue size. Additionally, each invocation of SHA-256 requires 64 serially dependent rounds of computation, and HMAC-SHA-256 requires invoking SHA twice in sequence. 128 rounds of computation may be easily pipelined for high throughput, but could start to rival DRAM accesses in terms of raw latency. Verifying multi-caveat IOCaps requires multiple invocations of the MAC function, so would take even longer.

For that reason, I choose an IOCap consisting of a 128-bit MAC and a 128-bit plaintext. Instead of making IOCaps grow as caveats are added, I reserve space for exactly two caveats inside the plaintext and leave them as zero[2] when unused. Under these constraints GenMac should be low-latency MAC function taking a 128-bit plaintext and a 128-bit key and producing at least a 128-bit MAC. Chained MACs need to use the previous MAC as the new key, so the key and MAC must be the same length, and NIST recommends a minimum key size of 128-bits as of 2030 [174, Table 4].

$$\text{GenMac}(C, K) : \{0,1\}^{128} \times \{0,1\}^{128} \to \{0,1\}^{128}$$

[1] It could be possible for a future system to restrict derivation by requiring a specific 'derivation key', for example, which would change this requirement—but I do not consider this here.

[2] It should be feasible to fill unused caveats with per-lease random data, increasing overall entropy at the expense of extra storage in the lease manager, but for this thesis I choose to prioritize storing as many leases as possible.

The CBC-MAC construction is a MAC function based on an underlying block cipher, which conveniently reduces to a single invocation of the block cipher when the message is the same size as the underlying block [185]. CBC-MAC using AES-128 [186] as the underlying cipher fits all the requirements, and requires only 10 rounds of computation to complete, so is more likely to be low-latency than a SHA-2 implementation. NIST recommends CMAC over CBC-MAC [6], citing the latter's weaknesses with variable-length messages [187, 188], but in this case the messages are fixed-length. CMAC itself, in the 128-bit plaintext case, reduces to a CBC-MAC of the message XOR-d with a key-dependent constant and could be swapped in. Performance would depend on caching the constant for parent capabilities, but implementations would have to calculate the constant on the fly for derived capabilities. I choose to prioritise latency and storage and thus choose CBC-MAC with AES-128.

$$\begin{aligned}\text{GenMac}(C, K) &= \text{CBC-MAC}_{\text{AES}}(C, K) \\ &= \text{AES-128}_K(C)\end{aligned}$$

I considered but rejected other candidates. The SHA-3 hash family are closer to AES in terms of hardware latency (e.g. 12 cycles [189]), but still have large key sizes and outputs. GCM-based and Wegman-Carter based MAC functions use per-plaintext nonces (numbers-used-once) to make it harder to correlate different MACs generated from the same key [182, 45]. However, a fundamental constraint of the IOCap system is that the host cannot track per-IOCap metadata. That would just degrade to a large IOMMU-style protection table. Poly1305-AES [190] derives nonces from a separate key and a sequence number, but because derived IOCaps are not centrally tracked, it would be impossible to prevent sequence numbers from being reused.

Given the capability text dictates what accesses are possible, and the MAC only authenticates the text, if an attacker can find new valid plaintext and MAC pairs $(C_{\text{Evil}}, M_{\text{Evil}})$ they could access new regions encoded in C_{Evil}. Attackers may try to directly access lease keys or calculate them using cryptanalysis. In the latter case, attackers may try to use the host as a verification or generation oracle. The IOCap enforcer could be used as a verification oracle, where the attacker submits accesses under candidate forged IOCaps and sees if they succeed, though attacks that use a verification oracle typically expect to use unbounded attempts. This can be mitigated by cutting off peripherals after a small number of failed attempts. The host driver could be used as a generation oracle, where the attacker submits arbitrary candidate plaintexts and receives the MAC for that plaintext (also known as a chosen-plaintext attack). This can be mitigated by only allowing peripherals to provide reasonable constraints on exposed regions. For example, a peripheral may request regions with a minimum alignment and size, but should not be given direct control over the base address. The rest of this analysis takes a conservative view and assumes the generation and verification oracles are unrestricted, and prove the resiliency of IOCaps in spite of that.

4.3.1 MAC collision attacks

Given a valid pair (C, M) an attacker could try to find C' with a matching MAC such that (C', M) is valid. CBC-MAC is vulnerable to *reforgeability* attacks, where one collision can be exploited to generate more forgeries, though this depends on an unrestricted generation oracle with full attacker control and multi-block messages [191, §A.1]. In single-block cases like ours, where the cipher block length equals the message length, CBC-MAC-AES under a given key acts as a bijective function and **cannot produce collisions under the same key**. This applies to parent and derived IOCaps equally.

4.3.2 Replay attacks

An attacker may attempt to use a previously valid pair (C, M) when the accessible memory is used for other purposes. While a lease is open, all IOCaps under that lease are usable and can be replayed arbitrarily. Assuming the host prevents exposed memory from being reused until the lease is closed, the only replay concern is replaying IOCaps *after* the key data is cleared from the relevant lease, because at that point the host will assume the exposed memory is inaccessible and may use it for other purposes.

Replay is impossible while the lease is closed. Once a new lease is opened under the same ID, the host and lease manager must ensure it does not use a stale key—ideally, loading a new key must be a prerequisite for opening a lease. The only chance an attacker has to replay a pair (C, M), where $M = \text{GenMac}(C, K_{\text{Old}})$), would be when the data associated with the relevant key is $K_{\text{New}} \neq K_{\text{Old}}$ i.e. a replay attack is only possible when the generated MAC is the same, for the same plaintext, with two different keys[3].

$$M_{\text{Old}} = \text{AES}_{K_{\text{Old}}}(C) \qquad M_{\text{New}} = \text{AES}_{K_{\text{New}}}(C)$$

$$M_{\text{Old}} = M_{\text{New}}$$

$\text{AES}_{K_{\text{New}}}$ should be a randomly selected permutation of the pseudorandom AES family. By definition, an attacker should not have significant advantage (over a brute force search) in distinguishing it from a random function. If there was a way for the attacker to accurately predict whether $\text{AES}_{K_{\text{New}}}(C) = M_{\text{Old}}$ without knowing K_{New} or brute forcing, that would be a form of advantage and AES would be considered insecure. As long as AES remains secure, **the risk of replay attack is negligible**. This also applies to caveated IOCaps.

[3] A similar class of 'key collision' attacks consider the risk of an attacker finding keys which create collisions [192, 193]. These attacks don't apply to IOCaps because the attacker doesn't control K_{Old} or K_{New}.

$$M_{\text{Old}} = \text{AES}_{K_{\text{Old}}}(C) \qquad M_{\text{New}} = \text{AES}_{K_{\text{New}}}(C)$$

$$M'_{\text{Old}} = \text{AES}_{M_{\text{Old}}}(C') \qquad M'_{\text{New}} = \text{AES}_{M_{\text{New}}}(C')$$

$$M'_{\text{Old}} = M'_{\text{New}}$$

Because AES is a family of pseudorandom permutations, the output of any given link in the MAC chain (e.g. M_{Old} and M_{New}) can be considered random. The chance of collisions is the chance that $M'_{\text{Old}} = M'_{\text{New}}$, where they are both encryptions of C' with different random keys, and as established above that chance is negligible.

4.3.3 Key recovery and forgery

At attacker may try to recover the key K for a given lease based on the IOCaps authenticated with it $(C_1, M_1), ...(C_n, M_n)$. An attacker may also try to recover the MAC for a parent capability (C^i, M^i) based on the IOCaps derived from it $(C_1^{i+1}, M_1^{i+1}), ...(C_n^{i+1}, M_n^{i+1})$. While the former is much more powerful, a successful attack on either would violate the security of the system by allowing the attacker to compute $\text{GenMac}(C_{\text{Evil}}, K)$ and forge new, unintended IOCaps. This is entirely dependent on the key because nonces are not available.

Analysis of CBC-MAC over many years has shown that the chance of an attacker being able to distinguish between a pseudorandom function and CBC-MAC$_K$, which would allow the attacker to find K, increases quadratically in the amount of messages sent [185, 194, 195]. This is expressed as the bound on an attacker's 'PRF advantage'—at most, how much greater the attacker's chance of distinguishing the functions is over 50/50. Most recently Jha and Nandi express this advantage for CBC-MAC in terms of σ, the total number of cipher blocks seen; q, the total number of messages seen; ℓ, the maximum number of cipher blocks per message; and n, the size of each cipher block in bits [195].

$$\text{Adv}^{\text{pf}}_{\text{CBC}}(q, \ell, \sigma) \leq \frac{14\sigma q}{2^n} + \frac{16\sigma q \ell^3}{2^{2n}} + \frac{q^2}{2^{n+1}}$$

Every message is one cipher block long therefore $\sigma = q$, $\ell = 1$, and we also know $n = 128$.

$$\text{Adv}^{\text{pf}}_{\text{CBC}}(q, 1, q) \leq \frac{14q^2}{2^{128}} + \frac{16q^2}{2^{256}} + \frac{q^2}{2^{257}}$$

The final advantage is dependent on q, the number of unique (plaintext, MAC) pairs seen by the attacker (i.e. the number of IOCaps issued under a lease), and is dominated by the $O(q^2/2^{128})$ term. The maximum number of unique IOCaps per key can be chosen based on the desired maximum advantage, but for our purposes I choose $q = 2^{16}$ to give an advantage of approximately $14/2^{96} = 1.8 \times 10^{-28}$. This gives a high assurance that **an attacker will not be**

able to find the key and forge new IOCaps before the limit is reached.

Peripherals using derived IOCaps must ensure they do not exceed this limit either, i.e. that they do not create over 2^{16} unique derived IOCaps from each parent. If they fail, a hostile peripheral *may* have a higher advantage in determining the MAC of the parent capability from the derived capabilities—but the attacker will not be able to forge entirely new IOCaps, and the feasibility and usefulness of any such attack is fiercely limited by the lifetime of the lease.

4.3.4 Lease lifetime, or 'cryptoperiod'

Cryptographic keys have a maximum lifetime, also known as a cryptoperiod. For example, NIST recommends that symmetric authentication keys have a lifetime of less than two years [174, Table 1]. In practice, lease lifetimes can be much, much shorter. For example, the maximum per-op lifetime is approximately 50 milliseconds. That is long enough for a GPU to process a single frame of a high fidelity real-time video game (30 frames per second i.e. 33ms per frame), or a video decoding accelerator to decode high-resolution video (24 frames per second i.e. 42ms per frame), both of which are conceivable individual ops[4]. It is also long enough to process a whole queue's worth of ops for an NVMe peripheral[5]. Even if an attacker managed to convince the host to mint 2^{16} unique IOCaps in this time, they would have to exhaustively search through at least 2^{96} candidate keys—a rate of over 10^{30} keys per second—which is currently computationally infeasible. Longer-living regions, such as pools exposed in bulk, can simply switch out their IOCaps for ones minted under a new lease before the old lease times out (see Section 4.5.1.1, p84). Even then lease timeouts could be much longer, even up to a full second or more, as long as AES is not completely broken. Under Grover's algorithm [196] AES-128 has 64-bits of quantum resistance, but attackers do not (and will not for the foreseeable future[6]) have a quantum computer that could evaluate 2^{64} operations in a single second. Even if this changes, lease lifetime should be easily tweakable at runtime.

4.4 Practical limits of a flat lease table

Lease keys have to fit in SRAM to ensure low-latency, so there are inherently a limited number of lease IDs—and therefore a maximum number of leases that can be active at once. CHERI CapChecker, an I/O protection solution targeting embedded accelerators on FPGA systems, used a 256-entry C-list of 128-bit capabilities [30]. A similarly-sized IOCap system could have 256 leases total, which may be suitable for embedded or low-performance systems, but Section 4.2.2 indicates systems may need around 16,000 in the worst case, and the final IOCap format I consider in Chapter 5 has 18-bits of lease ID. I list the total required key storage for both scenarios in Table 4.1, including a "Medium" scenario for smaller systems that do not

[4] In practice, memory will likely not need to be exposed for even that long—most GPU work is processed locally after DMAing memory from the host.

[5] Section 4.2.2 assumes a 512-element work queue, measuring 60μs op latency and ~1.5MIOPS throughput, which would take 0.4ms to complete.

[6] i.e. until 2074-13-04

Usage	Total Leases	Bytes per Lease	Total Bytes	Relative Size
Medium	$1{,}024 = 2^{10}$	2^4	$2^{14} = 16$KiB	$\frac{1}{3} \times$ L1-D
High	$16{,}384 = 2^{14}$	2^4	$2^{18} = 256$KiB	$\frac{1}{4} \times$ L2
Maximum	2^{18}	2^4	$2^{22} = 4$MiB	$4 \times$ L2

Table 4.1: The number of leases required in different scenarios, and the amount of storage they require, as compared to a single AMD Zen 5 core [201].

have as many peripherals or as deep queues. The Medium storage is on the order of a single L1 cache, High is less than a single L2 cache, and Maximum is roughly four L2 caches. Maximum would only be needed in highly virtualized cases where many clients could reserve leases without constantly using them—server-grade CPUs already have enough cores that this is not out of the question. Looking at it another way, the maximum number of leases would require approximately 0.8mm^2 of SRAM in a 2nm process [197]—for reference, AMD's recent SP5 server platform has a $24.79 \times 16 = 397\text{mm}^2$ I/O die [198]. Mobile SoCs such as the Apple A19 [199] and Google Tensor G5 [200] are smaller, hovering around 100mm^2, and likely sit between Medium and High.

I therefore conclude it should be possible for a system designer to provision enough SRAM to map leases 1:1 to ops or exposed memory regions. Increasing IOTLB size to the same degree instead of adopting IOCaps would not help as much, as new translations would still miss, regions larger than a page would still require multiple entries, and IOTLBIs would still be batched—either invalidating too much, too frequently and preventing the IOTLB from being filled; or invalidating even less frequently and worsening temporal safety. A large enough CapChecker could be feasible, but would break down if the host wanted to expose more regions than originally provisioned. IOCaps are more flexible.

4.5 Handling underprovisioned systems

If an IOCap system is built with an underprovisioned lease manager, for example if the designer wishes to save SRAM, it can adapt by exposing multiple ops or memory regions under the same lease. However, if the host isn't prepared to access exposed memory, the host has to wait for *all* of those ops to complete before the lease is revoked. If the host expects to read data out of a memory region, written by the peripheral, it has to wait for revocation. If the host expects to write data into a region, e.g. to reuse and send to another peripheral, it will either have to wait for revocation or allocate new memory to expose. I refer to these latency and memory overheads as 'overexposure'.

Note that unlike the overexposure created by deferred IOMMU invalidations, between the host asking for an invalidation and the IOMMU actually processing it, IOCap overexposure is entirely known to the host and is therefore not a security concern. During IOMMU overexposure the host believes the requested regions are no longer exposed and may touch them. Because strict IOCap invalidation is feasible, the host can track exactly when regions under a lease are

overexposed, and can avoid touching them until they are truly unexposed. IOCap overexposure is safe, but still undesirable, and this section presents strategies for managing the trade-off between overexposure and lease ID space.

4.5.1 Simple lease ID allocation

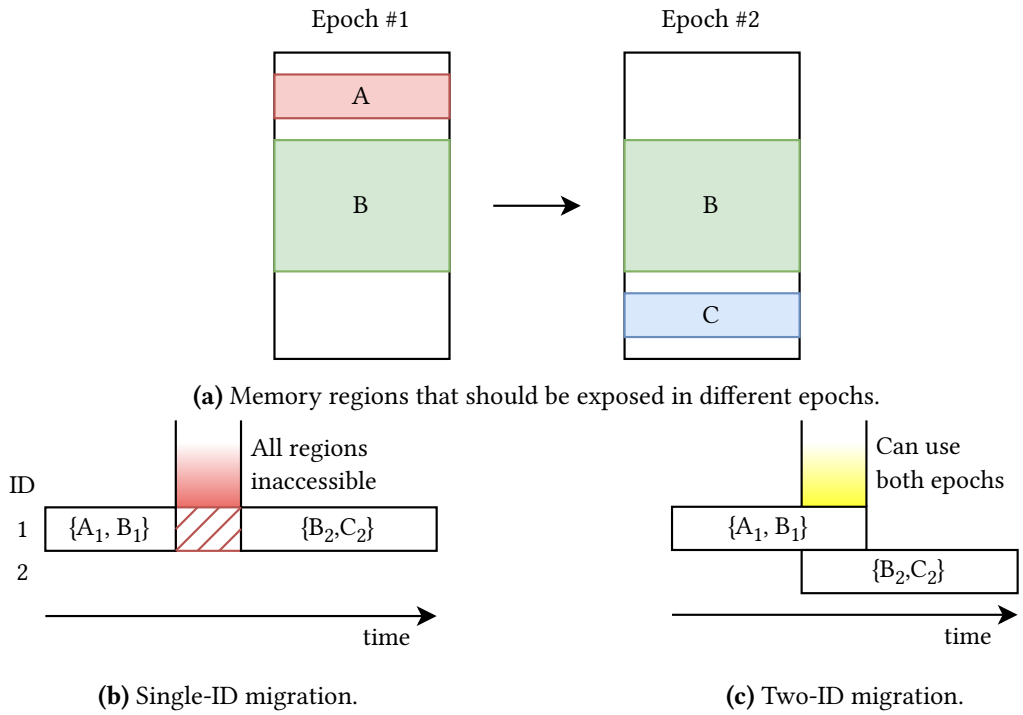
I address two categories of region in this thesis: persistent, long-lifetime regions; and ephemeral per-op regions. Both of my approaches depend on reserving groups of lease IDs. For instance, a single NVMe drive may have multiple queues (one per CPU core, to maximize bandwidth and avoid cross-core synchronization). A driver could then reserve[7] one group of lease IDs to expose the queues themselves as persistent regions; and then separate groups for each queue's ephemeral ops. Each strategy defines the group size, i.e. the number of lease IDs it requires, as N_L.

4.5.1.1 Persistent regions via epochs

Queues are one example of persistent regions which are not frequently freed or reallocated. That means a single lease can be used to expose many queues or persistent regions at once. More generally, any set of regions that only grows or stays constant can use a single lease. If that set of regions is completely fixed, the OS could even remove the key data from memory—preventing the minting of new IOCaps and ensuring the key can never leak. Whenever the lease times out, or when individual regions need to be revoked, the active regions should be migrated to a new lease. This process requires computing new IOCaps for all the still-active regions under the new lease, sending them to the peripheral, then clearing the previous lease's key once the new regions are updated. The old and new leases could use the same lease ID, as in Fig. 4.1b, but there would be a small gap where the peripheral could not use either lease during the switch. For maximum throughput a second lease ID can be used, as in Fig. 4.1c, where only one is used at a time except for a brief crossover period. During this crossover, any regions that were revoked will be overexposed until the change finishes. Many persistent regions can be exposed at once where $N_L = 1$ or 2.

This approach is analogous to epoch-based memory reclamation in concurrent systems [179], which indicates that $N_L = 3$ may be necessary when multiple actors (threads in the case of reclamation, peripherals here) are independently tracking the active epoch. The third lease is necessary if peripherals can be in a state where they have begun observing a new epoch (are receiving IOCaps under a new lease L_1) while still moving away from the old epoch (haven't yet discarded or replaced IOCaps from the old lease L_0). Under those conditions it is only safe to destroy the old epoch L_0 once all peripherals have moved through L_1 and are starting on a newer epoch L_2. Criticisms of this style of reclamation [202, 203, 204] note that stalled actors can stop the epochs from moving forwards, that it is not *robust* [205] against delays, but this

[7] In practice, the OS may reserve these groups of IDs statically or dynamically. I statically reserve lease ID groups at compile-time in CheriFreeRTOS, but CheriBSD dynamically discovers peripherals and reserves groups on request.



(a) Memory regions that should be exposed in different epochs.

(b) Single-ID migration.

(c) Two-ID migration.

Figure 4.1: Epoch migration examples. Two regions A and B are initially exposed under one lease, then a second lease is used to expose B and C, allowing A to be freed.

does not apply to IOCaps. If the system is under memory pressure or generally suffering from a stalled peripheral extending an epoch, it can simply close that epoch, breaking the stalled peripheral (and preventing it from touching old memory) but leaving all peripherals that have moved on unaffected.

4.5.1.2 Ephemeral FIFO regions

Ops and their associated buffers are an example of ephemeral regions which are exposed and revoked frequently. Some peripheral standards assume *first-in-first-out* (FIFO, i.e. in-order) op completion for each queue [26, 25], and others assume it in certain circumstances [41]. This is a good hint—ops that are enqueued close together are likely to complete together, so the regions they expose are likely to be revoked together—but you can't just use one lease to expose all ops. If the queue is saturated, i.e. after each op is completed a new op is immediately added, there will never be a point where all ops are completed; if they were all exposed under the same lease it would never be revoked. Simple epoch-based revocation as described above would likely not work, because minimizing overexposure would require short epochs, and the overhead of tracking which still-in-progress ops need to be *re*-exposed in new epochs and actually sending their new IOCaps to peripherals would likely be impractical.

I approach this problem using a group of $N_L \geq 2$ lease IDs, where each individual lease exposes up to L_{ops} ops before switching to the next one round-robin. The ideal one-op-per-lease approach is $L_{\text{ops}} = 1$, which requires as many lease IDs N_L as the queue length Q_{ops}. The opposite extreme is $N_L = 2$, where $L_{\text{ops}} = Q_{\text{ops}}$. Once lease #1 has exposed a full queue's worth of Q_{ops} ops, lease #2 has to cover the next Q_{ops} before lease #1 can be revoked and reused. The

relationship between L_{ops} and N_L is shown in (4.2). For all values of L_{ops}, at most $L_{\text{ops}} - 1$ ops will ever be overexposed—just before the last op under a lease is cleared. Minimizing overexposure requires minimizing L_{ops}, which requires increasing N_L, which is a tradeoff against the available lease ID space. Unlike the persistent case, there is not really an analogous garbage collection or reclamation mechanism—typical garbage collection scenarios do not have any reason to delay freeing an object once it isn't referenced, so it isn't necessary to group together objects with similar lifetimes.

$$L_{\text{ops}} = \left\lceil \frac{Q_{\text{ops}}}{N_L - 1} \right\rceil \tag{4.2}$$

4.5.2 Complex lease ID allocation

Not all queues and situations fit the simple strategies directly. I propose ways to adapt those situations to the simple strategies to reduce overexposure.

4.5.2.1 Priority queues

Some peripherals include a notion of op priority. NVMe typically indicates this on a per-queue basis [24, §4.13] but can also directly indicate individual ops should be completed with higher or lower latency [24, Figure 377]. Network peripherals have the concept of 'flows', where packets in a queue can belong to different flows and have different quality-of-service requirements, and flows with aggressive quality-of-service requirements are effectively higher priority than others. Queues like these can use a separate lease group per priority level to ensure low-priority ops do not keep high-priority ops overexposed. These lease groups can each use the ephemeral FIFO strategy—as long as the queue itself is in-order and saturated.

4.5.2.2 Unpredictable ops

Some queues, such as network receive queues, have no completion guarantees. If no network packets arrive buffers may sit in the queue for a very long time, waiting to be filled, until their lease times out. Instead, the peripheral can "complete" ops after a timeout even if no data has arrived. The host driver can then reuse the empty memory, and can even re-enqueue them *immediately*—before their initial lease is revoked—as long as the queue is processed in-order and the next lease is guaranteed to be revoked afterwards.

4.5.2.3 Out-of-order ops

The NVMe standard does not require ops to be started or completed in-order [24, §4.13]. It has no fairness guarantees within queues in theory [24, §4.13] or between queues in practice [206]. The greatest danger of out-of-order completion is if the completed ops are spread out too far, the queue will be filled with half-completed leases without any way to clear any of them. $L_{\text{ops}} = 1$ completely eliminates this problem, but even a small $L_{\text{ops}} > 1$ may be suitable if queueing order is a close proxy for completion order. As long as that holds, for example if quality-of-service

metrics ensure ops are not abandoned, choosing a lease ID at the point of enqueueing may still ensure that ops under the same ID will be completed close together. The driver may have domain-specific knowledge of what ops will be completed first, and could use that for more effective allocation, but this would have to be employed on a peripheral-by-peripheral basis. I discuss this further in the evaluation.

4.5.2.4 Memory pooling

Lease allocation is not zero-cost. It requires cryptographically secure random number generation, MMIO interactions with the lease manager, and tracking exposed memory regions for each lease. High-performance network drivers, such as DPDK, mitigate IOMMU cost by allocating contiguous hugepages (usually 2MiB or 1GiB) ahead of time—mapping them entirely into the peripheral's address space instead of mapping individual packets [207]. This improves page table walk latency [25, §5.7] at the perceived cost of the security model: the peripheral can access the entire hugepage at all times, using it as a memory pool, no matter what ops are enqueued. As long as developers have enough discipline to control what data enters and exits the pool, this is fine.

IOCaps support this technique. Instead of continually minting new IOCaps for each op, you can create a finite set of parent IOCaps for the pool, and create derived IOCaps for each op. Even if a hostile peripheral saw enough derived IOCaps to find the parent IOCap MAC, that isn't enough to reveal the lease key—drivers should assume the peripheral already always has access to the pool. This converts the ephemeral op exposures into persistent ones, and an epoch-based strategy can be used to migrate to new leases as needed. It is worth noting that this technique is only suitable if allocating dedicated, constantly-exposed, per-peripheral memory up front is acceptable/desirable, which is not always the case.

4.6 Evaluation

I have proposed two methods of lease allocation in this chapter which allow the total number of leases to be smaller than the total number of exposed regions. The epoch approach for persistent regions (Section 4.5.1.1, p84) is easy to quantify: if epochs roll over after a set time t, that t is the maximum overexposure a region can have. Systems with advanced implementations may reduce this time by closing epochs early once an unexposure is requested, perhaps with some debouncing to allow for multiple requests to be batched together, but these are engineering details that I leave for future work. As this method is intended for persistent regions, which are exposed and unexposed less frequently by definition—for instance, queues may be exposed at boot, and unexposed when the system shuts down—this category of overexposure is not so important. Ephemeral FIFO (Section 4.5.1.2, p85) is more contentious: it relies on FIFO op ordering, *may* be extendable to out-of-order peripherals like NVMe, and unexposure (and therefore overexposure) is much more frequent. Here, I go more in-depth in characterising overexposure for the ephemeral strategy.

Overexposure is most easily calculated in terms of time—the longest a region in the group may be overexposed—but in some cases it may be interpreted as a memory overhead. For example, in the case of network transmit, the actual latency with which an exposed packet buffer is returned may not be critical. Filling the queue is most important, which only requires allocating more new buffers instead of reusing the not-yet-unexposed buffers. The overexposure graphs throughout this section indicate both the raw overexposure latency and the total memory overhead: a system with no overexposure uses 1× its normal memory, a system where the average overexposure is equal to the average latency has to allocate roughly 2× as much memory, and so on.

I use a custom Python-based NVMe queue simulation to observe how overexposure is affected in different queueing scenarios. I looked into prior art in NVMe simulation but found they would either require in-depth modification and a full Linux system to track lease allocation [208, 209], were too old to reflect modern systems [210], or had poorly managed [211] or missing [212] source code. My custom simulation has some drawbacks: it assumes constant op inflow and outflow, is based on rough performance figures from prior work [143], does not model the completion queue, and ignores communication overheads; although the effects of the completion queue and the communication overheads are already accounted for when using latency figures measured from the host. Crucially, it correctly models the NVMe submission queue [24, §4.13], which is essential for some observations and is not the case for some prior art [211], and is easy to configure with different lease allocation models. This simulation is good enough to observe key properties of ephemeral FIFO lease allocation.

4.6.1 A FIFO queue

Start with the simple case. Consider a FIFO queue using the ephemeral op strategy, which has three parameters: Q_{ops}, the number of ops in the queue; N_L, the number of lease IDs; and L_{ops}, the number of ops each lease exposes before closing. Fig. 4.2, for instance, shows an example where $L_{\text{ops}} = 4$. Given it is in-order, I characterize the peripheral as completing ops with a constant latency t_{avg} and a constant throughput based on its IOPS. The first op under a given lease will be completed by the peripheral in t_{avg} seconds, but it cannot be unexposed until the other ops have completed. This overexposure $\text{over}_{\text{basic}}$ is the *extra* time taken to complete the other $L_{\text{ops}} - 1 = 3$ ops in the queue (4.3).

$$\text{over}_{\text{basic}} = \frac{L_{\text{ops}} - 1}{\text{IOPS}} \tag{4.3}$$

Fig. 4.3a shows the distribution of measured overexposure in a single in-order queue simulation. Table 4.2 shows that the $\text{over}_{\text{basic}}$ equation matches the maximum observed values. Fig. 4.3b shows the same distribution over a shorter queue, even shorter than the bandwidth delay product (here, 150μs × 100,000 IOPS = 15 ops), that reduces IOPS. Latencies skyrocket at the tail end when the queue is backed up, but overall the maximum memory usage is still only 2× the norm.

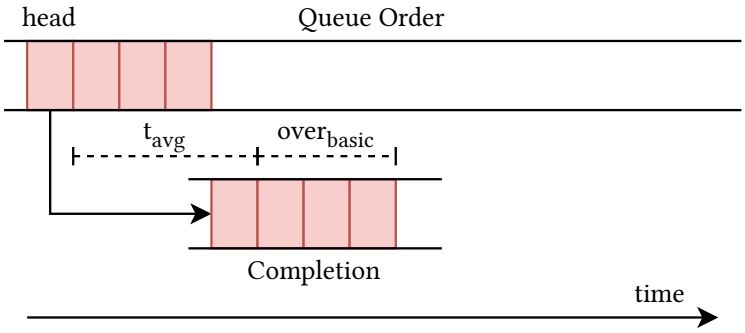
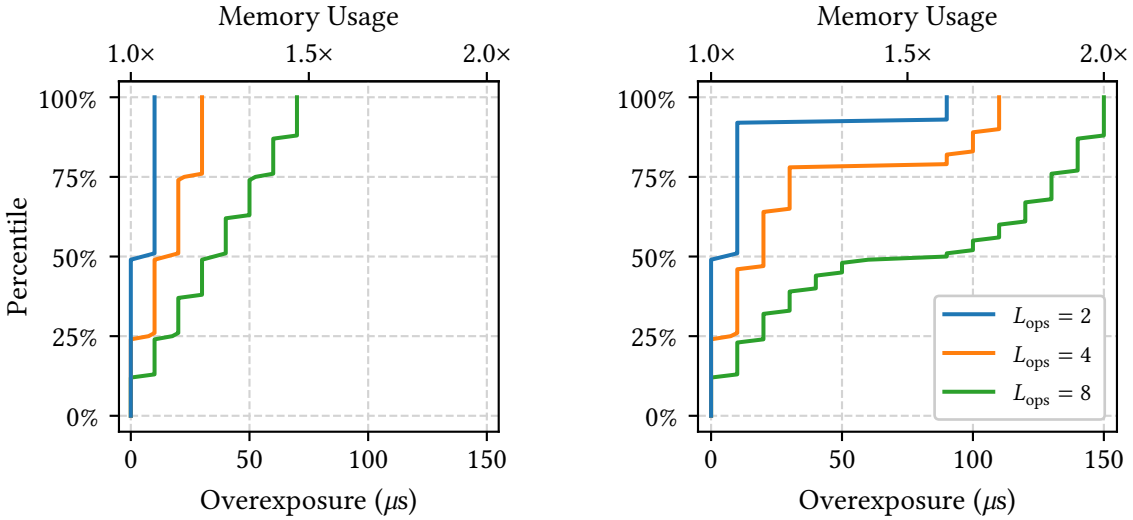


Figure 4.2: Ops in an $L_{ops} = 4$ lease group enqueued in an in-order queue. The first op is completed after t_{avg} seconds and is left overexposed for a further $over_{basic}$ seconds while waiting for the other ops to complete.



(a) $Q_{ops} = 32$.

(b) $Q_{ops} = 8$. IOPS drop to 46,600 due to limited queue capacity.

Figure 4.3: Overexposure distributions for in-order queue models where $t_{avg} = 150\mu s$, targeting 100,000 IOPS. Because latency is fixed, stair-steps of $1/\text{IOPS} = 10\mu s$ can be observed in both queues.

		$over_{basic}$	
L_{ops}	1/IOPS	Calculated	Observed
2	10.0μs	10.0μs	10.0μs
4	10.0μs	30.0μs	30.0μs
8	10.0μs	70.0μs	70.0μs

Table 4.2: $over_{basic}$ calculations for Fig. 4.3a's model.

4.6.2 An out-of-order queue

NVMe does not require submitted ops to be processed or completed in-order [24, §4.13], but I argue they can still be modelled as in-order queues with variable latency. NVMe quality of service is typically defined by latency percentiles [213], e.g. the 99th or 99.9th percentiles (a.k.a. P99 and P999), which can be used to model the degree of out-of-orderness. Even if the NVMe peripheral reads ops out-of-order from the queue, or rearranges them internally, the effects of that reordering will ultimately be reflected in the measured latency. Consider Fig. 4.4. I model ops as completing with random latencies distributed between t_{min} and t_{max} seconds. The worst overexposure will be if the first op is completed in t_{min}, and the final op is completed in t_{max} (Fig. 4.5), leading to an overexposure of $\text{over}_{\text{basic}}$ (4.3) plus the latency delta Δt_{max} (4.4).

$$\text{over}_{\text{max}} = \text{over}_{\text{basic}} + \Delta t_{\text{max}} \tag{4.4}$$

Fig. 4.6b shows the overexposure distribution for an out-of-order queue where each op fits a latency distribution shown in Fig. 4.6a. The effect of L_{ops} and Δt_{max} obscures $\text{over}_{\text{basic}}$ visually, but the computed over_{max} matches the observed values within a small margin of error (Table 4.3). Fig. 4.6c shows the impact of a too-small queue. Overexposure in both cases is dominated by the op latency component instead of throughput, so both graphs look similar, but when a t_{max}-latency op arrives in the wrong place the tail latency shoots up even further. That maximum overexposure exceeds over_{max}, which I believe is caused by the backed-up queue: each op having to wait up to t_{max} for an older op to vacate its slot in the queue. The value of over_{max} in these situations depends on the distribution of op latencies, and the situation can be avoided entirely by allocating a big-enough queue, so I have not derived an equation for it.

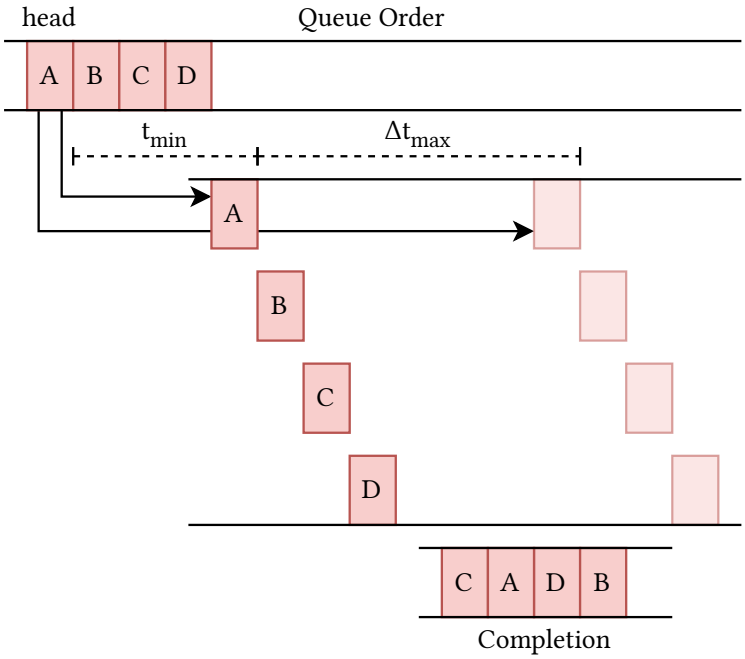


Figure 4.4: A demonstration of how ops with completion latencies distributed throughout a range can lead to out-of-order completion—even if the ops began execution in-order.

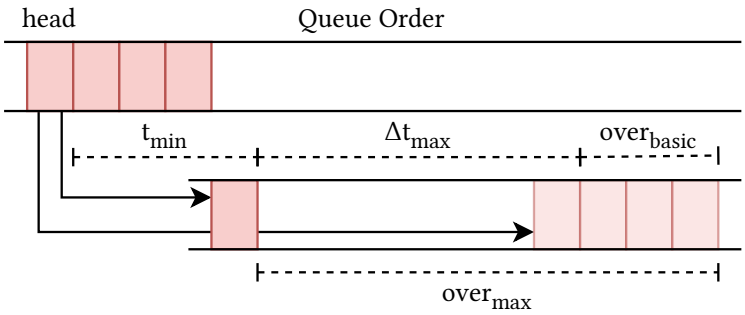
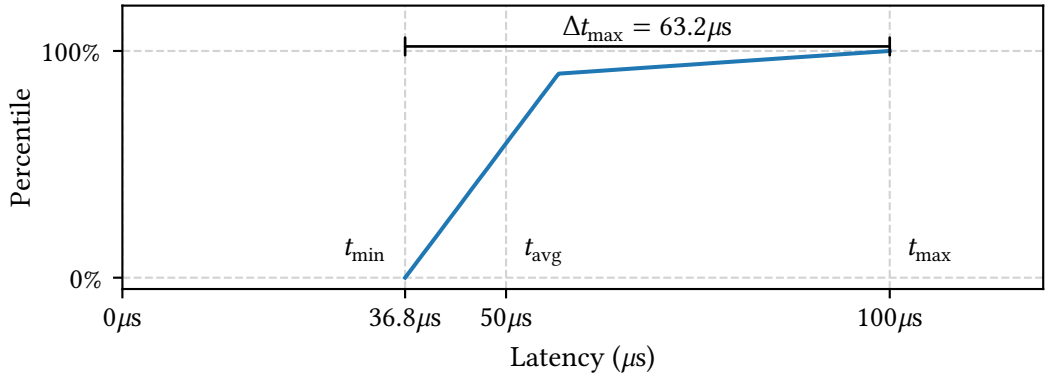
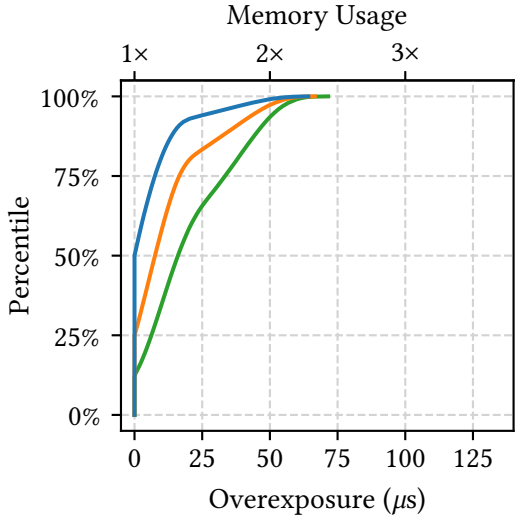


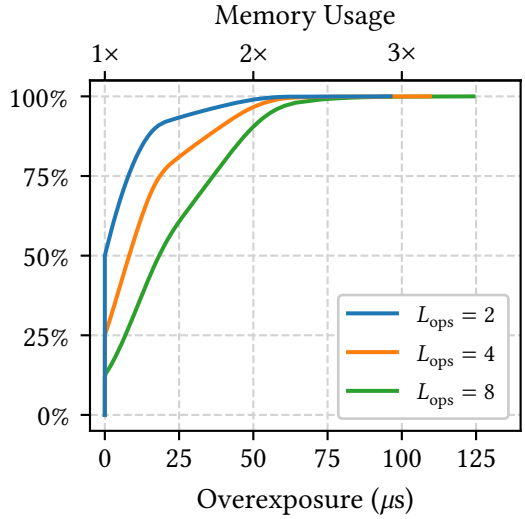
Figure 4.5: An explanation of the worst case overexposure over_{max} for a single in-order begin out-of-order complete queue.



(a) Op latency distribution. A simple model with a mean $t_{\text{avg}} = 50\mu s$ and maximum $t_{\text{max}} = 100\mu s$, based on rough estimations of current-gen hardware.



(b) $Q_{\text{ops}} = 128$.



(c) $Q_{\text{ops}} = 32$. Too small, so IOPS ≈ 400,000.

Figure 4.6: A single-queue out-of-order model targeting 800,000 IOPS under different queue depths. $\Delta t_{\text{max}} = 63\mu s$ dominates overexposure calculations for both, hence the graphs look similar except for the tail.

Memory usage caps at 3.5×. This may sound significant, but remember the total number of outstanding requests (and the memory required for them) is relatively low. If these were 4KiB DMAs, the initial memory usage for $Q_{\text{ops}} = 32$ would be 128KiB total, and including the overhead would be 448KiB—a trivial amount for a modern system to store.

				over$_{max}$	
L_{ops}	1/IOPS	over$_{basic}$	Δt_{max}	Calculated	Observed
2	1.2μs	1.2μs	63.2μs	64.4μs	64.2μs
4	1.2μs	3.7μs	63.2μs	66.9μs	66.6μs
8	1.2μs	8.7μs	63.2μs	71.9μs	71.5μs

Table 4.3: over$_{basic}$ and over$_{max}$ for the 128-op out-of-order model in Fig. 4.6b.

4.6.2.1 Impact on submission availability

When the host wishes to submit a new op to the peripheral, it must wait for room in the queue, and for there to be an available lease to expose it. Will an unusually long-lived lease prevent new ops from being submitted when there is room for them in the queue? No. Why? Well, in the case of NVMe, insertion into and removal from the submission queue is governed by a head and tail pointer. Ops are begun starting from the head, and are inserted at the tail. Once the tail and the head are next to each other, the queue is full—incidentally, this means the number of concurrent ops is actually $Q_{ops} - 1$. The head only moves once the op at the head has been completed by the peripheral, and as the head moves past it becomes *consumed*. This requires calculating N_L with a plus-one (4.5) such that new ops can be enqueued once the first lease is overexposed.

$$N_L = \lceil (Q_{ops} - 1)/L_{ops} \rceil + 1 \tag{4.5}$$

Out-of-order completion separates *consumed* ops from *completed* ops. Ops past the head can complete before the op at the head, and therefore be completed but not yet consumed. If a full lease's worth of ops complete in this way, the lease can be closed out-of-order before the head moves up, but the head can still only move once its op has completed. Functionally this means that ops can only be consumed, and new ops may only be enqueued, *in-order*—even if execution is out-of-order! This means the maximum number of concurrent leases is the same for in-order and out-of-order cases, which I have also observed to be true in practice in the simulations in Fig. 4.3 and Fig. 4.6.

4.6.2.2 Reads vs. writes and other priorities

NVMe reads and writes have different latency characteristics, both in the average and high percentile cases, and it may be beneficial to allocate leases for them separately. If latency were the only factor, it would be sufficient (though perhaps non-optimal) to characterize reads and writes together with a single latency distribution and allocate leases together. However, an NVMe peripheral may have separate capacity for reads and writes, and when stalled on one prioritize the other. This is a version of the priority system proposed in Section 4.5.2.1, p86, and it creates another trade-off. Splitting a fixed number of lease IDs between priorities would force ID groups to use smaller N_L, and thus larger L_{ops}, which would increase over$_{basic}$. Ultimately this depends on how much slower low priorities are than high ones and how often they block

each other. For generic cases it is probably fine to ignore this altogether—given it depends on the peripheral's implementation details, I leave it for future work to investigate.

4.6.2.3 Alternate approach — 'Arbitration Burst'

The NVMe specification includes an 'Arbitration Burst' configuration field [24, Figure 277] which can be set to allow peripherals to begin processing multiple ops at once from a single queue. It only indicates a maximum, not a minimum, and is limited to a maximum value of 8; but if peripherals take advantage of this in practice it could be used to ensure all ops inside a lease begin processing at once. This is not a complete solution because it is phase-dependent. If the NVMe starts processing *some* ops from a lease, but not *all* of them, that first batch will still be overexposed until the rest are completed. Ultimately it could reduce $\text{over}_{\text{basic}}$, even in that case, if it reduces the number of times the peripheral examines other queues before finishing the lease—but the actual number of ops pulled from those other queues in the meantime will be the same. I leave this for future work to determine the impact in practice.

4.6.3 Multiple out-of-order queues

High-throughput peripherals such as NVMe only maximize throughput when fed by many per-thread queues. The standard NVMe arbitration mechanisms dictate that the peripheral uses round-robin[8] to select the next queue to pull from. Overexposure increases drastically when ops are pulled one-at-a-time from each queue, because the final op starts processing much later than the single-queue case (Fig. 4.7). Assuming an unbiased distribution of IOPS between the queues (4.6), $\text{over}_{\text{basic}}$ can be calculated much like the in-order case (4.7). over_{max} is calculated as before (4.4).

$$\text{IOPS}_{\text{queue}} = \frac{\text{IOPS}}{N_Q} \tag{4.6}$$

$$\begin{aligned} \text{over}_{\text{basic}} &= \frac{N_Q \times (L_{\text{ops}} - 1)}{\text{IOPS}} \\ &= \frac{(L_{\text{ops}} - 1)}{\text{IOPS}_{\text{queue}}} \end{aligned} \tag{4.7}$$

Fig. 4.8 shows the overexposure distributions of an equivalent flow to Fig. 4.6 split across 8 queues with round-robin arbitration. The distribution of latencies smooths out the visual impact of $\text{over}_{\text{basic}}$, but the computed over_{max} values match observed behaviour (Table 4.4).

[8] NVMe peripherals must implement pure round-robin arbitration, and can optionally provide weighted round-robin arbitration that balances queues of different priority levels, or vendor-specific arbitration [24, §4.13].

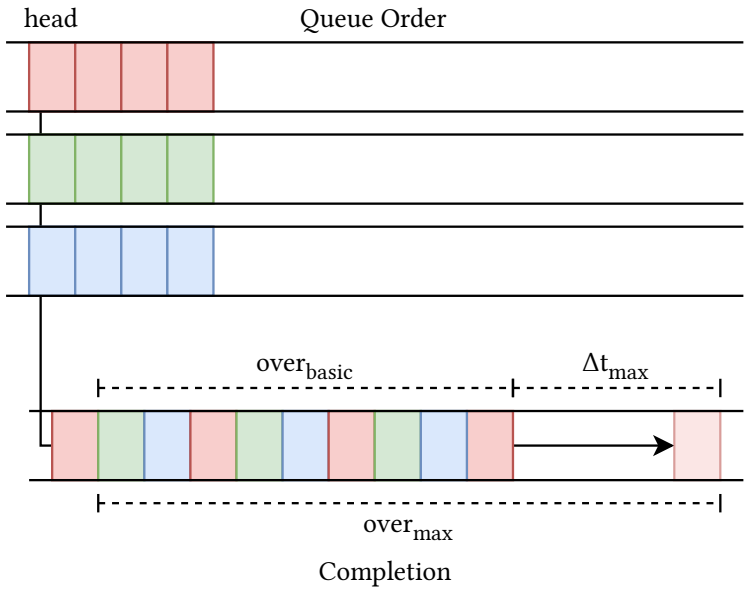


Figure 4.7: An explanation of the worst case overexposure $over_{max}$ for multiple in-order begin out-of-order complete queues.

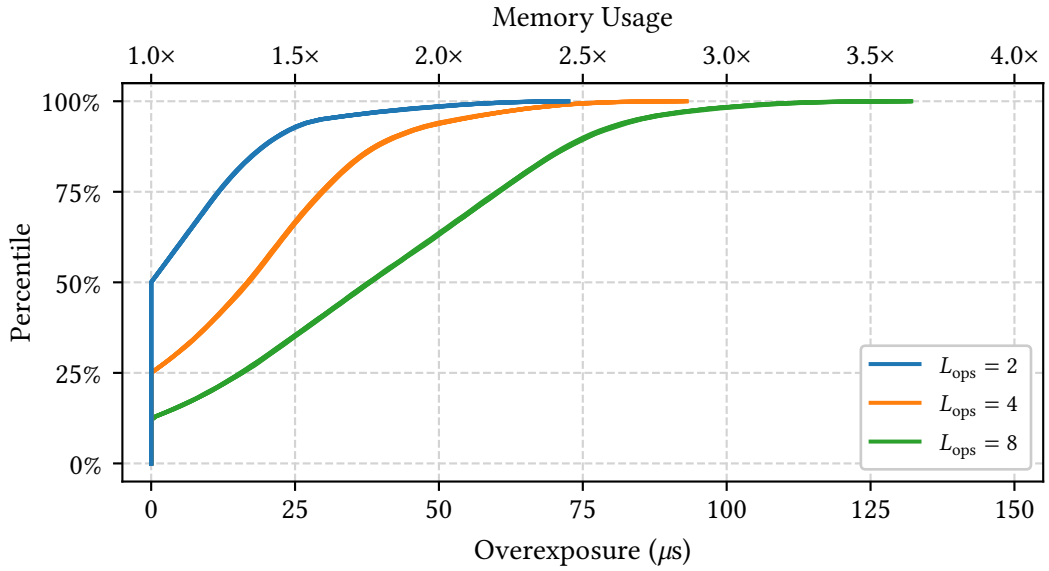


Figure 4.8: The out-of-order model from Fig. 4.6 split across $N_Q = 8$ with $Q_{ops} = 16$.

				over$_{max}$	
L_{ops}	$1/IOPS_{queue}$	$over_{basic}$	Δt_{max}	Calculated	Observed
2	10.0μs	10.0μs	63.1μs	73.1μs	72.5μs
4	10.0μs	30.0μs	63.1μs	93.1μs	93.1μs
8	10.0μs	70.0μs	63.1μs	133.1μs	132.1μs

Table 4.4: $over_{basic}$ and $over_{max}$ for the multiple-queue out-of-order model from Fig. 4.8.

4.6.3.1 Alternate approach — Share leases between queues

It may be desirable for a set of N_Q queues to use the same group of lease IDs. Merging queues under a single lease ID group can ensure ops under the same lease always start processing at the peripheral together (Fig. 4.9), saving overexposure, at the cost of cross-queue (i.e. cross-thread) synchronization. This would use roughly the same number of overall leases but would reduce $\text{over}_{\text{basic}}$ (e.g. Fig. 4.10). Thread synchronization has proven to be a throughput bottleneck in prior I/O work [20, 143], so this approach should be considered with caution. A hybrid approach could limit this bottleneck by only merging specific queues—e.g. only those used by cores that share cache memory, or are on the same NUMA node—but I have not investigated this thoroughly and leave it for future work. In general it appears the effect of merging is greatest when L_{ops} is high (Fig. 4.10) or when merging many threads (Fig. 4.12).

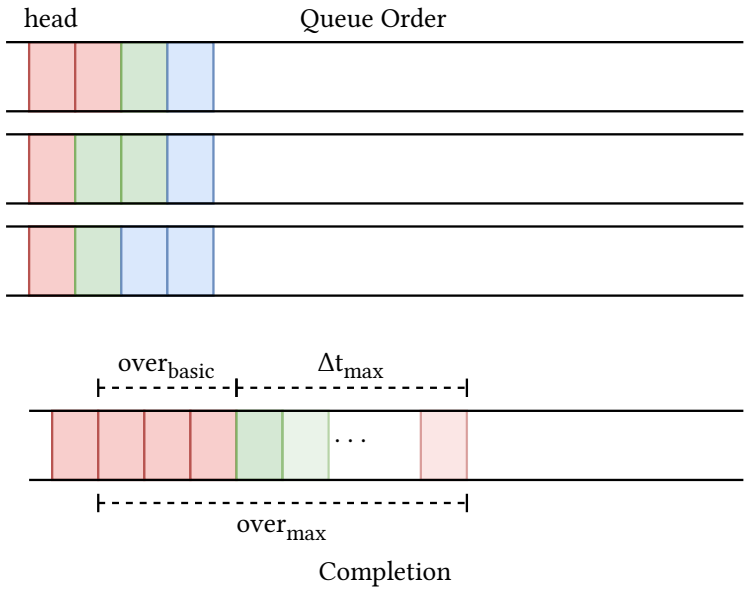


Figure 4.9: A demonstration of merging queues under the same set of leases to reduce overexposure.

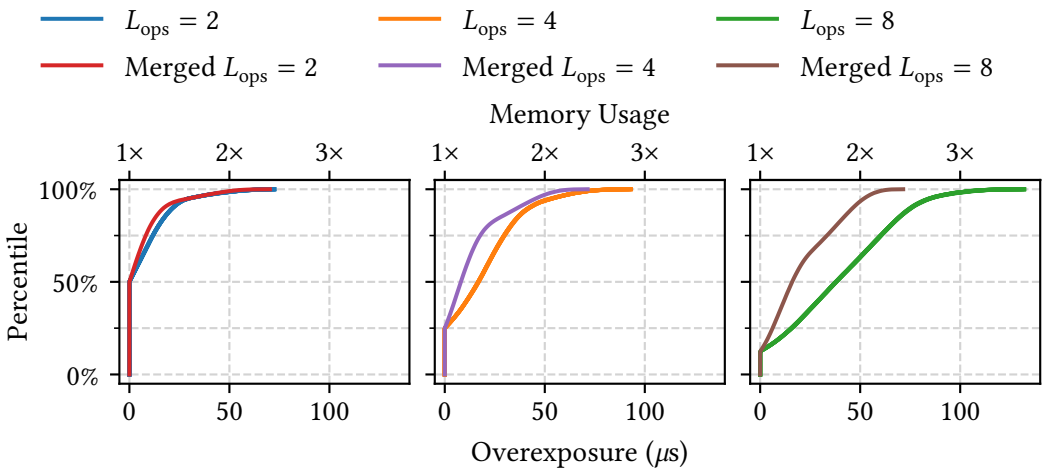


Figure 4.10: Comparing overexposure between a fully merged lease ID group and the per-queue scenario from Fig. 4.8.

4.6.4 Applications in practice

Prior work benchmarking NVMes under high-throughput constraints found the average latency of random 4KiB reads on a Samsung PM1733 NVMe SSD was approximately 175μs, and the 99th percentile was approximately 400μs, for 1.25 million IOPS per peripheral[9] [143, Figure 17b]. These measurements used 63 separate threads, each with a dedicated queue, which requires approximately 20,000 per-device per-queue IOPS. The bandwidth delay product for t_{avg} is 4, but for t_{max} it is 8, which forces a queue depth of 10—one extra for the gap between the tail and head, and another extra to ensure the host always has space to enqueue. Fig. 4.12 shows the overexposure distributions for potential L_{ops} values, including those for a combined single-lease-group solution spread across all queues.

Table 4.6 shows the total lease counts observed in practice and the maximum required number for each configuration. This shows another potential advantage of a merged queue: when groups are split and queues are short, *any* queue which encounters N_L very-long ops at once will need a maximum-size group, and *all* queues will eventually hit this case. When groups are merged, N_L increases, N_L concurrent very-long ops become extremely rare, and the total lease count decreases. The total memory usage remains around 3×, except for $L_{\text{ops}} = 8$ which caps at 4.6×, of an initial usage of $63 \times 9 \times 4\text{KiB} = 2.2\text{MiB}$. This gives the designer a variety of manageable options to trade memory usage for lease ID space.

The right choice of ephemeral lease strategy is entirely dependent on the limitations of the target system. Over the course of this section I have presented equations for modelling in-order and out-of-order systems in the abstract, and demonstrated their accuracy by comparing them to simulation data. Increasing L_{ops} and merging queues into the same lease ID groups are the two knobs a system designer can tweak if they do not have enough lease IDs for $L_{\text{ops}} = 1$. Tweaking these knobs increases overexposure, either increasing latency outright or increasing memory usage as more buffers are allocated to maintain throughput, but it's important to remember this is *safe* overexposure as opposed to the IOMMU's unsafe/unexpected overexposure. When expressed as memory overhead, the overexposure is entirely manageable because the amount of outstanding data at any time is already relatively small. Furthermore, IOCap overexposure does not actually impede throughput, unlike IOMMU latency [19].

Many systems, especially consumer SoCs, will have lower throughput requirements and shallower queues—they will need fewer lease IDs overall, and if they are underprovisioned the ephemeral strategy allows them to reduce lease ID usage while keeping overheads manageable. When latency or memory usage are bottlenecks, it may be possible to use the ephemeral strategy for other peripherals to make room, or to use memory pooling to reduce this overhead entirely. That should be most useful in bespoke systems, such as video streaming servers, where kernel-level programming or high-performance toolkits such as SPDK and DPDK are employed—though those systems are also likely to be well-provisioned in the first place. The bulk and ephemeral strategies presented in this chapter demonstrate a quality of IOCaps that the IOMMU simply does not have: predictable, deterministic, and *adaptable* temporal safety.

[9] The original experiment used 8 SSDs in parallel for 10M IOPS, which reduces to 1.25M per peripheral.

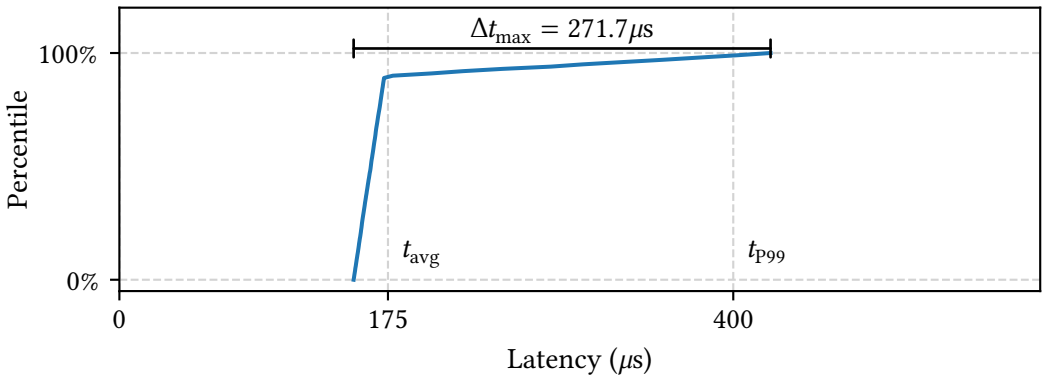


Figure 4.11: Samsung PM1733 op latency at 1.25M IOPS. Based on t_{avg} and t_{P99} gathered in [143, Fig. 17b].

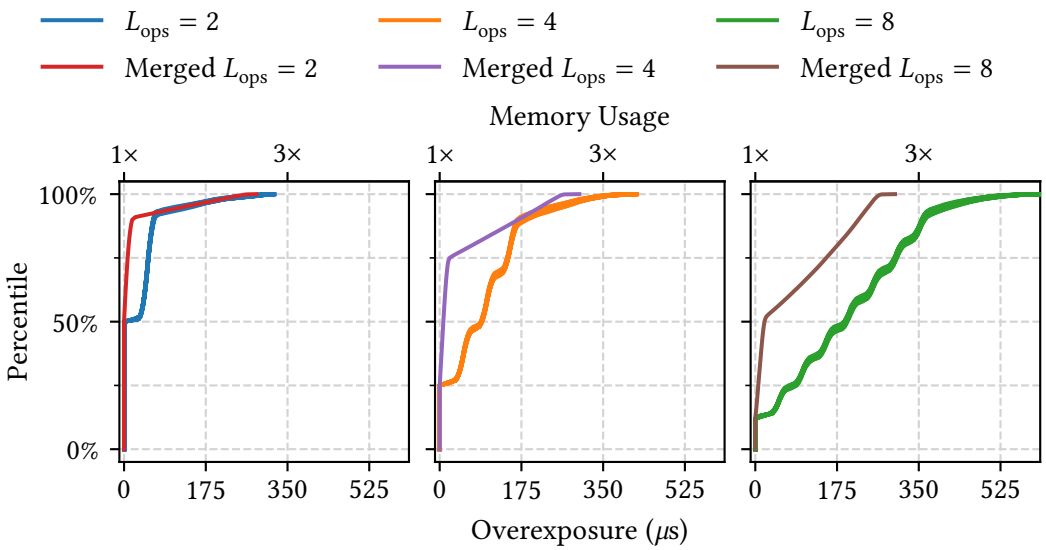


Figure 4.12: Overexposure distributions for a 63-thread Samsung PM1733 modelled at 1.25M IOPS and $Q_{ops} = 10$.

				$over_{max}$		
L_{ops}	$1/IOPS_{queue}$	$over_{basic}$	Δt_{max}	Calculated	Observed	(Merged)
2	50.4μs	50.4μs	271.7μs	322.1μs	322.1μs	282.9μs
4	50.4μs	151.2μs	271.7μs	422.8μs	422.6μs	299.1μs
8	50.4μs	352.8μs	271.7μs	624.4μs	625.0μs	299.1μs

Table 4.5: $over_{basic}$ and $over_{max}$ for the Samsung PM1733 in Fig. 4.12.

	Per-Queue		Merged	
L_{ops}	Theoretical Max.	Observed	Theoretical Max.	Observed
1	630	458	568	290
2	378	315	285	159
4	252	189	143	93
8	189	126	72	56

Table 4.6: Comparing the maximum total lease counts between per-queue and merged lease ID groups.

4.7 Summary

Hypothesis 1

IOCap systems can use leases to enforce temporal safety under a variety of workloads using a 128-bit MAC function.

In this chapter I selected AES-128 CBC-MAC as a suitable MAC function for the IOCap format, determined how many leases a system could need in total, and proposed and evaluated mechanisms to reduce lease ID usage. While those mechanisms do come with overheads, results in simulation suggest those overheads are manageable.

Facet 1.1

There exist suitable MAC functions for authenticating 128-bit IOCaps with 128-bit MACs.

Yes. AES-128-based CBC-MAC is suitable for 128-bit text and MAC. Implementations could also consider using AES-256 [186] or QARMA-256 [214] for the initial MAC, which both use a stronger 256-bit key to encrypt 128-bit blocks and produce 128-bit MACs, but would be forced to use 128-bit-strong ciphers for caveats—it is impossible to securely extend 128-bits into a 256-bit key.

Facet 1.2

There exist suitable MAC functions for authenticating 64-bit IOCaps with 64-bit MACs.

This may be true as long as caveats are not required. Caveats require using MACs as keys, which is simply not secure at less than 128-bits [174], but truncating 128-bit MACs to 64-bits or smaller has been deemed secure enough by industry for techniques like pointer authentication [215]. Again, a variant of QARMA [214] may be attractive for lightweight hardware implementations.

Facet 1.3

It is feasible for an IOCap system to include a large enough SRAM-based lease key table to map leases to exposed regions one-to-one, even for high-performance systems with extreme I/O demands.

Yes. Systems with extreme lease ID requirements can either build systems with dense SRAM to individually handle per-op regions, or can use memory pooling (indeed they may require it for performance) to expose fewer regions and reduce ID requirements. Systems that do not have 32 high-throughput, constantly saturated peripherals should need much fewer concurrently exposed regions, and therefore much fewer lease IDs.

Facet 1.4

It is feasible and practical for the IOCap system to adapt to underprovisioned systems with fewer concurrent leases than expected, still exposing the same number of regions, without sacrificing security.

Yes. I have proposed two methods to reduce lease ID usage, both of which increase overexposure without reducing security, and demonstrated how to predict the impact of different reductions using the ephemeral strategy.

CHAPTER 5

Spatial safety with compressed IOCap formats

In Chapter 3 I established some requirements for an IOCap binary format. These included a compact encoding of an initial memory region, two caveats to allow the region to be reduced, and a lease ID field—all compressed into a 128-bit capability text C with an attached 128-bit MAC computed from a function $M = \text{GenMac}(C, K) = \text{AES}_K(C)$ (Chapter 4). Ultimately these encoding formats need to be chosen to allow hosts to encode IOCaps for memory regions, which could be large and bulky or small and precise, and then decode them in hardware with high throughput. I begin this chapter by expanding on the precision requirements based on common size and alignment requirements in peripherals, and investigating prior methods for compressing memory regions into capabilities. I then describe the different components of an IOCap text format, how formats can be designed and configured for different requirements, and the specific format used in the rest of this thesis. Finally, I measure the precision and reducibility of these formats and their caveats, demonstrating how they meet my chosen requirements, and evaluate the facets of this chapter's hypothesis: does a suitable 128-bit format exist (Facet 2.1), could a smaller 64-bit format exist for more constrained systems (Facet 2.2), and can these formats suitable to replace buffer descriptors (Facet 2.3).

5.1 Goals

IOCaps need to be usable in practice, and practical reality places a number of constraints on the design of the format. Basic constraints include the need to encode 64-bit addresses, as many systems have much more than 4GiB of accessible memory, and the need for the MAC and capability text to both be 128-bit. In this section I deduce more practical requirements from estimations and observations of real systems and peripherals.

5.1.1 Precision

Different peripherals need to touch memory regions of different sizes, some with very granular precision and others more coarse, and if IOCaps are to replace buffer descriptors these regions need to be expressed exactly. Other capability systems can tolerate inaccuracy, such as CHERI rounding the size of allocated regions up when they are not exactly representable [216, §3.7], but here these lengths have semantic value. For instance, transmit descriptors for network peripherals encode the exact length of the packet that must be sent over the wire. Packets sent over IP can theoretically be up to 65,535-bytes [217, §3.1], as their length is encoded in a 16-bit field, though in practice IPv4 Ethernet caps out at 1,500-bytes for standard packets [218]. Ethernet allows larger 'jumbo' packets [219] and more recent extensions to IPv6, TCP, and UDP allow theoretically exceeding the 16-bit maximum [220], but in practice jumbo packets over the internet cap around 9,000-bytes. For example, Intel's E810 line of high-performance Ethernet controllers from 2024 max out at 9,728-bytes [221, §3.2.1.4]. These controllers encode TX packet length in a 14-bit field, allowing 1-byte precision up to that limit, and encode the full 64-bit packet base address [221, §10.5.3.1.1].

Unlike network devices, NVMe storage units tend to reason about host memory in terms of discrete power-of-two-sized regions. NVMe peripherals reason about their internal storage in units of 'logical blocks', the size of which can be configured by the host as a power-of-two starting at 512-bytes [24, §5.15, p171], and are aware of the host's native page size [24, §3.1.5]. Each read and write op descriptor encodes a scatter-gather list of page-sized regions (PRPs) [24, §4.3] or byte-granular[1] regions (SGL descriptors [24, §4.4]) to access. Linux and FreeBSD usually transfer data over PCIe as whole pages into a page cache [143, 21, 140], using PRP lists [222, 223], but Linux uses SGL descriptors for *larger* transfers like hugepages and smaller sub-page fragments [224, 19:52].

VirtIO, a generic I/O queue and descriptor protocol for various kinds of peripheral, uses simple 64-bit base and 32-bit length descriptors allowing 1-byte granularity [26, §2.7.5]. Peripherals take advantage of this precision, both for semantic meaning as expressions of transfer length, and for protocol-specific quirks—most notably exposing single-byte writable regions to signal completion for individual ops. Though initially designed for virtualization [225], the VirtIO protocol and its descriptor formats are starting to be used by hardware aiming to take advantage of widespread driver support [226, 227, 228].

5.1.2 Reduction

IOCaps should be able to replace buffer descriptors, but don't always have to be used that way. Reduction allows a host to hand out access to large swathes of data for a peripheral to subdivide as it sees fit: both for external use, for example handing a peer peripheral access to part of a shared arena; or for internal compartmentalization, just like CHERI [33, 34]. CXL demonstrates the demand for delegating large regions of a single memory expansion device to

[1] NVMe peripherals can require SGL descriptors to be four-byte granularity instead [24, §4.4].

different participants. In that case, each participant could be a complete system, with their own host and peripherals. Individual systems could then use reduction to expose sub-regions to their peripherals with improved granularity compared to existing solutions (see Section 2.4.3, p38).

How does this impact IOCaps? Well, it shows that the format needs to be able to encode large contiguous regions, which will have different alignment and sizing properties than individual descriptors. Notably, their sizes are unlikely to have semantic meaning, so it could be more acceptable to round sizes up when they can't be exactly represented. It is also likely that larger regions are aligned to a larger power-of-two, meaning fewer bits can be used to store the base address. We can look to CPU memory allocators for examples. `glibc` and Google's PartitionAlloc implementations of `malloc()` resort to using `mmap()` to create dedicated virtual pages for large objects [229, 230], which by definition are page-aligned (16KiB on Apple [231] and Android [232], and 4KiB in most other places). This also extends to hugepages, contiguous physical allocations that are larger than an architecture's baseline page. For example, 64-bit RISC-V supports at least 2MiB and 1GiB-sized hugepages, which both have to be aligned to their size [233, §12.4]. Allocations that are between 2MiB and 1GiB, if backed by these hugepages, would have to be aligned to 2MiB. 1GiB+ allocations would have to be aligned to 1GiB.

5.1.3 Permissions

Thus far we have seen two usage scenarios for IOCaps: bulk sharing, and to replace individual descriptors. Individual descriptors for ops generally need readable or writable, not both at the same time. See network transmit vs. receive, NVMe write vs. read—indeed VirtIO descriptors, intended to be a universal solution, only allow one or the other [26, §2.7.5]. However, regions shared in bulk, or generally any free-form shared memory region[2], may need to start out as both readable and writable. If needed, it should be possible to reduce the permissions of a read-write IOCap with caveats—and like CHERI it should be possible for a caveat to *only* reduce permissions without shrinking the exposed region.

5.1.4 Lease ID space

The size of the lease ID field depends on how many lease IDs exist in the system, which was estimated to be 2^{14} in Section 4.2.2, p77. This implies that the lease ID field should be at least 14-bits long for a densely-packed lease ID space. In practice, there may be sparsity. Drivers may be allocated more lease IDs than they strictly need, for safety or to simplify allocation, and virtualization may require namespacing—allocating large blocks of IDs for different virtual machines which then sub-allocate. Note that this would not change the total number of lease IDs used, adding more virtual machines doesn't change the total IOPS, it just changes how they are divided. A system with more VMs would require fewer lease IDs per VM, assuming the

[2]For example, VirtIO has the option to combine submission and completion queues into a single structure that is simultaneously read and written by the host and peripheral [26, §2.8].

overall system is bandwidth limited. Additionally, larger systems may use a set of distributed lease managers, each with a separate ID space, reducing the number of ID bits that need to be saved in the IOCap. To allow a margin, I aim to add at least 4-bits to the lease ID field, totalling 18-bits.

5.1.5 Summary

Each IOCap exposes a single contiguous region of memory, and must encode the 64-bit base address and length of that region within 128-bits of capability text. This region is interpreted with respect to an interconnect, which is an arbitrary mapping of addresses to resources [46], so the IOCap must be able to represent the full 64-bit address space—even if contemporary systems do not have enough memory to exhaust it. As established in Chapter 3 this region must be represented by an "initial region" field and two caveat fields that may optionally reduce that region. The region may start as peripheral readable (R), writable (W), or both (RW); and either caveat should be able to reduce RW to R or W without shrinking the exposed region.

Ideally it should be possible to exactly encode arbitrary regions less than 10KiB long with 1-byte precision, in order to accurately represent Ethernet packets and sub-page regions, and it's okay for this to rely on the caveats for precision. These individual descriptor use cases are less likely to benefit from subdividing themselves. For bulk sharing, the initial region should be able to represent any regions less than 2MiB at 4KiB precision, 2MiB–1GiB at 2MiB precision, 1GiB–512GiB at 1GiB, and 512GiB–256TiB at 512GiB to support hugepage alignment—here, using the page sizes from RISC-V Sv57 addressing [233, §12.6]. The text must be able to express regions of these precisions in few enough bits to leave at least 18-bits free for the lease ID. This requires compressing the initial region and caveat fields into approximately 110-bits. Said compression should prioritize low-latency hardware decoding/checking, as this was previously the bottleneck for the IOMMU, and therefore avoid long critical path operations like integer multiplies.

5.2 Prior art

Encoding the start and end addresses of a memory region is not a new problem. However, encoding *arbitrary* start and end addresses under tight *compression* is a less common problem, and prior solutions typically rely on redundancy with another pointer near or within the encoded region. This section goes over these prior solutions and notes techniques that will assist with IOCap compression—which can't rely on redundancy.

5.2.1 Direct encoding

First, consider the simple case. Why not just encode the base and length directly? This was an approach taken by many early capability systems that exposed memory regions [234, 3][149,

§4, §5, §7, §9], more recent attempts at hardware-enforced spatial safety [235, 236, 237], and even more recent attempts at software-enforced safety [238, 239].

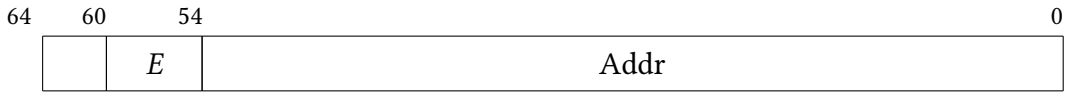
The earliest systems used 16- or 24-bit address spaces, with the largest directly encoded region reaching 40-bits [149, §9], which would be acceptable for our purposes except for the small problem of the forty-odd extra bits of addressing we need. HardBound [235] and Intel MPX [236] both create a shadow address space associating all 64-bit stored pointers with a 64-bit base and length. MPX encoded these as a simple 128-bit (base, length) tuple. HardBound noted that would effectively triple the size of each pointer, and employed two compression mechanisms: employing 4-bit memory tagging, with 14 patterns that represented common lengths and 1 pattern for arbitrary lengths; and special-casing pointers where the top n bits are all-zeroes or all-ones. CHERI's initial capability format also used direct encoding, extending each 64-bit pointer with 64-bit base, 64-bit length, and 64-bits of metadata [237]. It quickly moved to a compressed encoding to reduce the memory overhead of 256-bit fat pointers [240]. SoftBound [238] and Fil-C [239], software-based solutions that add bounds checks in software, prioritize simple software decoding and stick with uncompressed 64-bit base and length.

Systems that choose to directly encode base and length without compression either limit them to a smaller address space or bite the bullet and use a full 128-bits to encode them. IOCaps need to encode three separate regions (the initial region, the region with caveat 1 applied, then with caveat 2) together in approximately 110-bits. HardBound's attempts to compress rely on a predetermined set of common, small lengths, and limiting the base address, neither of which are suitable here. The descriptor encodings shown in Section 5.1.1 save bits by taking advantage of domain-specific knowledge, such as packet size limits or page alignment, which IOCaps cannot rely on—a single IOCap enforcer which understands exactly one format may have to cover many peripherals.

seL4 [241], a software-based capability system, uses capabilities to grant access to many system resources including page tables. The 'frame capabilities', which grant access to individual pages [242, §7.1.3], are encoded differently on different architectures with different page sizes. Each frame capability format needs to represent pages of different sizes—normal, large, or huge—and use a two-bit field to indicate which size the capability represents. Base addresses are encoded directly: on 64-bit RISC-V, the base address is always 39-bit[3], even though the bottom 12-, 21-, or 30-bits will be zero to align the hugepages, and this pattern is consistent on Arm and x86 platforms[4]. This speaks to a broader principle. With a fixed format, if *some* bits of the base address *may* need to be specified, they will always take up space. However, Section 5.1.2 has already shown that alignment requirements vary with size. This is a crucial component of compressed capabilities in general and is the backbone of IOCap's approach.

[3]It is 39-bit instead of 64-bit likely because it is designed for the HiFive Unleashed, the only RISC-V board with a verified seL4 proof as of April 2026 [243], which only supports 39-bit virtual addressing [244].

[4]See the `frame_cap` structures in the seL4 source code [245], specifically `include/arch/{x86,arm,riscv}/arch/{32,64}/mode/object/structures.bf`.



$$\text{Base} = \{\ \text{Addr}[53 : E]\ \|\ 0'E\ \}$$
$$\text{Length} = 2^E$$

$0'E$ denotes an E-bit bitvector where all bits are zero.

Figure 5.1: The M-Machine [246, 247] capability format.

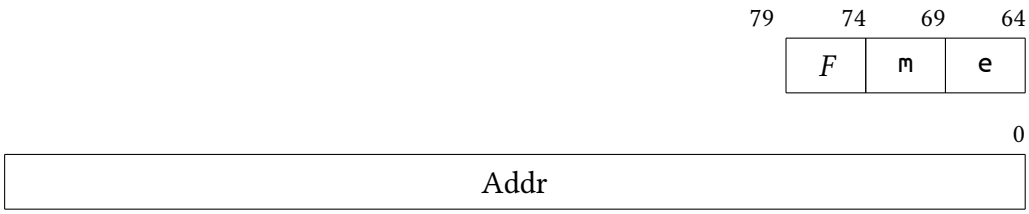
5.2.2 Compressed capabilities

CHERI's "Concentrate" capability format [240] is one of a string of examples of compressed capabilities. Each example ends up creating a different kind of fat pointer, encoding both a pointer and an accessible bound around it, and relying on similarities between the pointer and the bound's base address for compression.

M-Machine [246, 247] is the most basic form of this idea (Fig. 5.1). An M-Machine capability contains a 54-bit address field Addr, a 6-bit exponent field E, and 4-bits of permissions. The whole 54-bit address space is split into aligned 2^E blocks and the bounds are the block surrounding the pointer. The 54-bit base is compressed by assuming the top $(54 - E)$ bits of Base and Addr are identical, and setting the rest of Base to zero, an approach which has two notable flaws[5]. Firstly, many CPU-side allocations are not power-of-two sized or naturally aligned. CHERI Concentrate demonstrated this when evaluating the precision of its capability format [240]. Secondly, the natural alignment requirement has pathological failures with certain base addresses. Consider representing a region starting at address `0x1` that is `0x2` bytes long. Because this range crosses an `0x2` boundary it cannot be represented by the M-Machine with a length of `0x2`. Instead, the address has to be rounded down to `0x0`, and in order to encompass the originally intended region the length has to be rounded up to `0x4`. This may seem petty, but then consider a much larger base address starting just under halfway through the address space, `0x1F..FF`, i.e. $2^{53} - 1$. Any region with that base address and a length smaller than 2^{53} will cross an alignment boundary, ultimately rounding the address down to `0x0` and setting $E = 53$, granting access to a much wider region than originally intended.

Aries [248] avoids this problem by introducing a *mantissa* M (Fig. 5.2). The length is not a direct power-of-two, but instead an integer multiple of a power-of-two, allowing encoded regions to cross alignment boundaries to a certain degree (depending on how large the mantissa is). The same principle is used in the IEEE-754 floating-point standard [249] and both formats encode the mantissa with an implicit top bit to extend its range. IEEE floating point encodes the

[5]A third flaw is the assumption that the pointer is always in-bounds, making it impossible for a pointer to move out-of-bounds and then back in again. This is critical for C-language compatibility, and an explicit requirement for CHERI Concentrate [240], but as IOCaps do not need to encode a pointer along with the bounds I consider it out-of-scope for this discussion.



$$E, M = \begin{cases} 0, \mathtt{m} + 1 & \mathtt{e} = 63 \\ \mathtt{e}, \mathtt{m} + 1 + 16 & \text{otherwise} \end{cases}$$
$$\text{Base} = \{\, \text{Addr}[63 : E] - F \parallel 0'E \,\}$$
$$\text{Length} = M \times 2^E$$

Figure 5.2: The Aries [248] capability format.

mantissa as `1.x` [249, §3.4], representing a range from 2^E–2^{E+1} for each selected exponent value, ensuring different exponents don't encode ranges that overlap and create redundant encodings. This limits the minimum encodable 'normal' number to $1.0 \times 2^{\min(E)}$. However, *subnormal* numbers can also be encoded under a specific exponent value that disables the implicit top bit, allowing a range from 0–$2^{\min(E)}$. Subnormal numbers are rare, because the IEEE exponent can be negative and very large, but the regions encoded by Aries and other formats are always positive integers. Thus, the subnormal case for IEEE is the basic case for small regions in all the floating-point formats I have examined. Aries describes it as the "small segment" case, which is used to represent regions between 1–16-bytes long with a zero-exponent and a four-bit mantissa, and all other cases set the implicit bit. The base address of the capability is expressed as an offset from the encoded pointer. Again, the pointer is assumed to always be in-bounds, and the finger determines the offset from the pointer's exponent-aligned region to the base address' exponent-aligned value. The finger is encoded as 5-bits to match the mantissa (4-bits plus the implicit bit). This encodes a 64-bit pointer, 64-bit base, and 64-bit length in 79-bits.

Low-Fat Pointers [250] chooses not to encode the length but instead the top of the allocated region i.e. base plus length (Fig. 5.3). This choice does not provide greater compression but it allows for more efficient hardware. Bit-substitutions, effectively moving data, are cheaper than arithmetic; and if the base and top can be extracted separately (without, say, a serial dependency on the base to compute base-plus-length) then comparisons to the base and top can be made in parallel. The base and top are still aligned to blocks of 2^E. Though Low-Fat does not encode a length-mantissa, we can think of the B and T fields as base- and top-mantissas respectively—here, both 6-bits. Correction terms c_{base} and c_{top} are used to tweak the base and top if the encoded region crosses over a 2^{E+6} boundary. Tying the corrections directly to the address relies on the address always being in bounds just like Aries and M-Machine. In total, Low-Fat Pointers represent a 46-bit address and bounds in 64-bits. A 64-bit address variant would require approximately 82-bits.

CHERI Concentrate [251, 240, 216] combines techniques from Aries and Low-Fat to com-



$$c_{base} = 1 \text{ if } B > \text{Addr}[5 + E : E] \text{ otherwise } 0$$
$$c_{top} = 1 \text{ if } T < \text{Addr}[5 + E : E] \text{ otherwise } 0$$

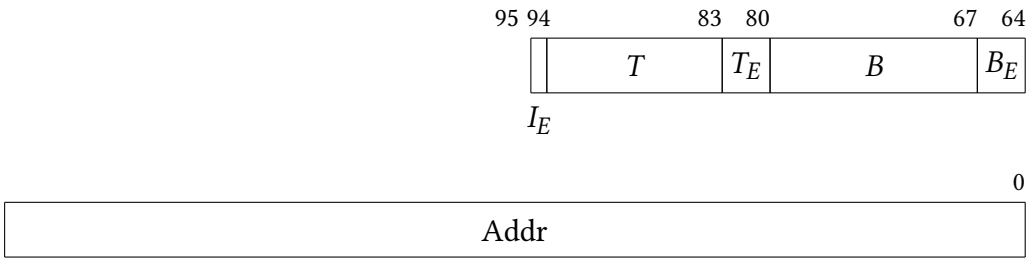
$$\text{Base} = \{\, \text{Addr}[45 : E + 6] - c_{base} \parallel B[5 : 0] \parallel 0'E \,\}$$
$$\text{Top} = \{\, \text{Addr}[45 : E + 6] + c_{top} \parallel T[5 : 0] \parallel 0'E \,\}$$

Figure 5.3: The Low-Fat [250] capability format.

press bounds as much as possible (Fig. 5.4). Like Aries, it employs separate subnormal and normal encodings that control an implicit top bit; and it encodes base and top directly using bit-substitution like Low-Fat. These are combined by remembering the relations between Top, Base, and Length. Even if some bits of Top are encoded directly, the top bits of T can still be compressed by deriving the top bits of the length-mantissa (which may be implicit) and adding them to the top bits of B. Furthermore, CHERI Concentrate prioritizes precision in the subnormal case. Instead of inferring the subnormal case from a particular exponent value (like Aries) it uses a separate bit to indicate subnormal, assumes $E = 0$, and uses the exponent bits to extend the mantissas. This encoding was inspired by Posits [252], which use a run-length encoded exponent that effectively dynamically varies mantissa bits. It creates a jump in representable precision between the subnormal and normal cases: the bottom 3-bits of each mantissa are assumed to be zero in the normal case, so the minimum precision for normal ranges is multiples of 8. Finally, the *representable* region is separated from the *accessible* region to allow the pointer to roam out-of-bounds without losing the bounds metadata. This scheme allows compression of a 64-bit address, base, and top into 95-bits. L_{msb} is inferred differently for subnormal and normal, L_{carry} is equivalent to the correction term from Low-Fat, and here c_{base} and c_{top} are separate correction terms based on the representable/accessible relationship. The full definitions of c_{base} and c_{top} can be found in the CHERI ISAv9 technical report [216, §3.5.4].

5.3 IOCap compression

Neither direct encoding nor prior compressed encodings are suitable to fulfil the requirements set out in Section 5.1.5. Direct encodings rely on making assumptions about the address space and length, that cannot be assured in the general case, to save space. Even then, using a fixed format for all bounds ends up wasting bits in large, well-aligned regions where many bits are zero. Prior compressed encodings were forced to store full-size pointers along with their bounds, ensuring that even if the Base and Length were aligned to a high degree, the



Subnormal ($I_E = 0$)	Normal ($I_E = 1$)
$E = 0$	$E = \{ T_E \parallel B_E \}$
$T[2:0] = T_E$	$T[2:0] = 0$
$B[2:0] = B_E$	$B[2:0] = 0$
$L_{\text{msb}} = 0$	$L_{\text{msb}} = 1$

$$L_{\text{carry}} = 1 \text{ if } T[11:0] < B[11:0] \text{ otherwise } 0$$
$$T[13:12] = B[13:12] + L_{\text{carry}} + L_{\text{msb}}$$

$$\text{Base} = \{ \text{Addr}[63 : E+14] + c_{\text{base}} \parallel B[13:0] \parallel 0'E \}$$
$$\text{Top} = \{ \text{Addr}[63 : E+14] + c_{\text{top}} \parallel T[13:0] \parallel 0'E \}$$

Figure 5.4: The CHERI ISAv9 [216] variant of the CHERI Concentrate [240] capability format.

bottom bits of the Addr were always stored in case they were different. Thus, even compressed encodings are limited to approximately 80-bits out of a 110-bit limit, which would need to store two caveated regions in the remaining bits. In this section I explain new methods of encoding an initial region and caveats into that 110-bit limit. I demonstrate a format for the initial region that skips encoding zero bits for large and well-aligned regions, explain how it can be parameterized to suit different constraints, and derive a method for finding optimal compressions (e.g. 64-bit base and length together in 63-bits) with controlled tradeoffs. I then describe two methods for expressing caveats, useful for different requirements.

5.3.1 Initial region

The core concept of initial region compression is shifting precision. If we can assume larger regions are more aligned, and therefore have many zeroes at the end of their base address, those bits can be repurposed to represent larger mantissas. Larger mantissas can represent a larger range of lengths at the same precision, creating a snowball effect and quickly reaching the maximum representable bounds as mantissa width increases. The overall size of such a format is bounded by the constraints it must meet, such as the hugepage alignment requirements in Section 5.1.5, of which the strongest are usually the ones with the finest alignment requirements.

Before considering concrete constraints, let us reason about the concept in abstract. Start from a basic floating-point mechanism where the base is encoded directly with zeroes at the

W	E	Implicit $\mathtt{M_{top}}$	$\lvert\mathtt{B}\rvert$	$\lvert\mathtt{M}\rvert$	Encodable Range	Used Range
0	0	0	32	8	2^0–2^8	2^0–2^8
1	0	1	32	8	2^8–2^9	2^8–2^9
2	1	1	32	8	2^9–2^{10}	2^9–2^{10}
3	2	1	32	8	2^{10}–2^{11}	2^{10}–2^{11}
4	3	0	29	11	2^3–2^{14}	2^{11}–2^{14}
5	3	1	29	11	2^{14}–2^{15}	2^{14}–2^{15}
6	4	1	29	11	2^{15}–2^{16}	2^{15}–2^{16}
7	5	1	29	11	2^{16}–2^{17}	2^{16}–2^{17}

Table 5.1: Details of a hypothetical 32-bit $n = 2$, $w = 3$ initial region encoding, fully characterized as 32-0-8-3-1-1. E, $\mathtt{M_{top}}$, and $\lvert\mathtt{M}\rvert$ are all derived from the width class W and don't need to be directly encoded inside the IOCap. The power-of-two encodable range for each class starts at $E + (\mathtt{M_{top}} \times \lvert\mathtt{M}\rvert)$ and ends at $E + \mathtt{M_{top}} + \lvert\mathtt{M}\rvert$.

bottom (5.1), and the length is a mantissa multiplied by an exponent[6] (5.2). Assume we have w bits to encode the exponent. Each encoded exponent controls more than just the actual exponent, so call the w-bit encoded field the 'width class' W. To keep the hardware simple we can limit the dynamic mantissa tradeoff to n discrete 'splits'—always choosing a power-of-two n so that hardware can select a split using the top $\log_2 n$ bits of the width class. A normal floating-point format like Aries[7] is roughly equivalent to $n = 1$ split and $w = 6$.

$$\text{Base} = \{\ \mathtt{B}\ \|\ 0'(|\text{Base}| - |\mathtt{B}|)\ \} \tag{5.1}$$

$$\text{Length} = \{\ M\ \|\ 0'E\ \} \tag{5.2}$$

Consider a simple format with two splits and eight width classes ($n = 2$, $w = 3$, Table 5.1). The first width class of every split must be subnormal[8], setting the implicit mantissa top bit $\mathtt{M_{top}} = 0$, and all other classes set $\mathtt{M_{top}} = 1$. The first split encodes 8-bits of mantissa ($|\mathtt{M}| = 8$), and the second upgrades to $|\mathtt{M}| = 11$. Each width class can encode an inclusive range between two power-of-two lengths by assuming the mantissa is always nonzero—adding one after concatenating $\mathtt{M_{top}}$ to $\mathtt{M}$ to reach the final mantissa M (5.3).

$$M = \{\ \mathtt{M_{top}}\ \|\ \mathtt{M}\ \} + 1 \tag{5.3}$$

I assume a universal constraint to simplify calculations: that the precision of the base, i.e. the address width $A = |\text{Base}|$ minus the number of bits used to encode the base $|\mathtt{B}|$, should never be greater than the precision of the length i.e. the exponent E (5.4). In other words, it seems less useful to encode a region with a base that is less aligned than its length (for example, a

[6] It should be possible to rework this format to encode the top of the allocated region instead of the length, which may be desirable for high-performance decoders, but I leave that for future work.

[7] Here, assuming the encoded Addr is exactly equal to the Base.

[8] To see why this is necessary, consider width classes $W = 4$ and 5 in Table 5.1. Increasing $|\mathtt{M}|$ and keeping $\mathtt{M_{top}} = 1$, effectively skipping class 4, would create a gap between 2^{11} and 2^{14} of unrepresentable lengths.

half-page-sized region that can only start at the top of a page) than the other way around (such as a page-sized region that does not need to start on a page boundary). In other other words, it is better to be more flexible with the base and round up lengths. This has two effects in practice: it allows me to assume the base precision and length precision are identical at the start (5.5), and it allows $|\mathtt{B}|$ to be derived from E at the start of each split, forming a direct relationship between the encoded mantissa and E (5.6).

$$A - |\mathtt{B}| \leq E \tag{5.4}$$

$$|\mathtt{B}|_0 = A - E_0 \tag{5.5}$$

$$\Delta|\mathtt{M}| = -\Delta|\mathtt{B}| = \Delta E \tag{5.6}$$

The entire range of splits can then be characterized based on the address size A, initial values E_0 and $|\mathtt{M}|_0$, and the change in E; which itself can be expressed in terms of w, n, and a new parameter E_{Step} (5.7). E_{Step}, which is set to 1 in Table 5.1, controls how far the dynamic range extends. Even though E_{Step} can be manipulated independently of the other parameters, it is possible to calculate the minimum E_{Step} to cover the whole address space based on the other parameters (5.9). Care must be taken because a large E_{Step} risks creating gaps (where the minimum length expressed by one split is greater than the maximum of the previous) or growing $|\mathtt{M}|$ beyond the total amount of bits used at the start.

$$\Delta|\mathtt{M}| = \Delta E = \overbrace{(2^{w-\log_2 n} - 2)}^{\Delta \text{ inside split}} + E_{\text{Step}} \tag{5.7}$$

$$A = \overbrace{E_0 + (n-1)\Delta E + (2^{w-\log_2 n} - 2)}^{\text{Final } E} + \overbrace{|\mathtt{M}|_0 + (n-1)\Delta|\mathtt{M}|}^{\text{Final } |\mathtt{M}|} + \overbrace{1}^{\mathtt{M}_{\mathtt{top}}} \tag{5.8}$$

$$\therefore E_{\text{Step}} = \frac{A - E_0 - |\mathtt{M}|_0 - (2n-1)(2^{w-\log_2 n} - 1)}{2n-2} + 1 \tag{5.9}$$

How, then, do we decide on the six parameters A, E_0, $|\mathtt{M}|_0$, w, $\log_2 n$, and E_{Step}? We begin with format-specific constraints. Section 5.1.5 has already established the need for a 64-bit address space $A = 64$ and a set of size/alignment constraints that must be representable. Table 5.2 reformats those constraints numerically and computes one measure of each constraint's 'strength': the number of base address and mantissa bits required to encode a region under those constraints. In this case the 2MiB-sized → 4KiB-aligned constraint is the strongest, and needs the same alignment as the first constraint, so we can use it to compute E_0 and $|\mathtt{M}|_0$. Setting $E_0 = 12$ and $|\mathtt{M}|_0 = 8$ ensures that all regions between 4KiB and 2MiB in size can be represented at 4KiB alignment.

Constraint		Min. Bits Required
Regions of size 2^x	must be aligned to 2^y	$(A-y)+(x-y-1)$
12	12	51
21	**12**	**60**
30	21	51
39	30	42
48	39	33

Table 5.2: The alignment constraints from Section 5.1.5 expressed mathematically, with the minimum number of bits required to encode regions under each one, assuming $A = 64$. The most powerful constraint, that which requires the most total bits to encode, is bolded. The computation assumes that $A - y$ are required to encode Base, and that $x - y - 1$ are required to encode M. That -1 on the mantissa term accounts for the implicit top bit, which grows the maximum mantissa to $|\mathtt{M}| + 1$.

At this point we can simply exhaustively search the space of w and $\log_2 n$, calculate the optimal E_{Step} for each pair, and choose out of those a configuration that satisfies all constraints. An exhaustive search is feasible because the values of w are limited. $w = 7$ is enough to encode lengths up to 2^{128} without compression, in practice compression allows much smaller w, and $\log_2 n$ by definition must be less than w. Small values of w provide greater compression, and small $\log_2 n$ reduces hardware complexity, so a search should try to minimize both but may increase one in order to reduce the other. In practice I find that $(3, 2)$, $(5, 1)$, and $(6, 0)$ form the Pareto frontier of this $(w, \log_2 n)$ search, shown in Table 5.3, and I select $(3, 2)$ with $E_{\text{Step}} = 8$ to provide the greatest possible compression. The final format, formatted as a six-tuple of inputs, is 64-12-8-3-2-8. The total amount of required bits is the initial $|\mathtt{B}|$ and $|\mathtt{M}|$ plus w bits to encode the width class W, totalling 63-bits, leaving 47-bits for caveats (5.10).

$$\begin{aligned} \text{total width} &= \overbrace{A - E_0}^{\text{Initial } |\mathtt{B}|} + |\mathtt{M}|_0 + w \\ &= 64 - 12 + 8 + 3 \\ &= 63 \end{aligned} \tag{5.10}$$

w	$\log_2 n$	0	1	2	3	4	5
1							
2							
3				8			
4				5	4		
5			1	1	1	2	
6		0	0	0	0	0	1

Table 5.3: An exhaustive search across w and $\log_2 n$, indicating E_{Step} when valid values that respect constraints were found. The Pareto frontier is the contour of the valid values traced from the origin in the top-left of the table.

	Initial	Caveat 1	Caveat 2
1	RW	-	-
2	RW	R	-
3	RW	R	R
4	RW	W	-
5	RW	W	W
6	RW	RW	-
7	RW	RW	R
8	RW	RW	W
9	RW	RW	RW
10	R	-	-
11	R	R	-
12	R	R	R
13	W	-	-
14	W	W	-
15	W	W	W

Table 5.4: The 15 possible values of the permissions chain. Read-write (RW) permissions can be shrunk by caveats into read (R) or write (W), each caveat may choose to shrink permissions or not, and chains that start at R or W cannot be shrunk but still encode how many caveats are enabled.

5.3.2 Caveats, chained

The IOCap must have a field that indicates which caveats are enabled, and must also have a field that indicates what permissions are granted to the encoded region. For a caveated IOCap with a MAC chain it must also always be possible to reconstruct prior IOCap texts in the chain, i.e. it must be possible to reconstruct what the permissions were for prior steps. Given that the caveats can individually restrict permissions, this requires the enabled caveats and the permissions to be encoded together into a single structure I call the permissions chain. As the only permissions required are read, write, and read-write, and there are at most two caveats active, there are exactly fifteen possible permission chains (see Table 5.4). It can thus be encoded in a single 4-bit field with one encoded value spare. A 4-bit field is small enough that any useful quality (e.g. is caveat 1 enabled, is caveat 2 enabled, is readable, is writable) can be extracted on FPGA with a single 6-input LUT, the smallest logic element possible, so I have not attempted to optimize the encoding for specific lookups or manipulations and leave that for future work.

5.3.3 Index caveats — Bulk-to-bulk

Reduction and delegation are not currently used in the I/O space, so caveat design becomes an exercise in design space exploration: hypothesizing what may be useful, and exploring if it is possible. To that end, I have constructed two hypothetical scenarios where caveats could be used, and designed a format for each. First, consider a CXL-esque large memory controller which exposes a large region of expansion memory to a system. This initial region is very large, and an individual system with tens of peripherals may wish to split it among them. This

requires a caveat that reduces one bulky region to another, smaller, bulky sub-region.

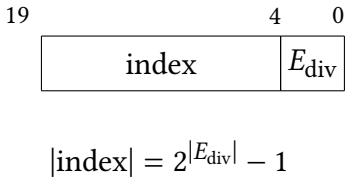
If the sub-region is allowed to be bulky, it seems reasonable to restrict it to be of power-of-two size. This is convenient given the length of the initial region is already expressed as a multiple of a power-of-two. Thus, the first caveat is an 'index caveat'—one that reduces the initial region to an array of power-of-two blocks, where the power-of-two is selected relative to the initial region exponent, and selects exactly one block to expose as a sub-region. Every width class in the initial region has a maximum expressible power-of-two size, which is equal to $E + |\mathtt{M}| + \mathtt{M_{top}}$, from which we subtract a new field E_{div} to get the block width in (5.11). A second field, 'index', selects one of those blocks in (5.12) and (5.13). Effectively, this caveat is a mini-version of the M-Machine format stuffed into the IOCap. It inherits the problems that format has with misaligned base addresses, as I discuss further in Section 5.5.3.

$$E_{\text{Cav1}} = E + \overbrace{|\mathtt{M}| + \mathtt{M_{top}}}^{\log_2 \text{ of max. mantissa}} - E_{\text{div}} \tag{5.11}$$

$$B_{\text{Cav1}} = \text{index} \times 2^{E_{\text{Cav1}}} = \text{index} << E_{\text{Cav1}} \tag{5.12}$$

$$T_{\text{Cav1}} = (\text{index} + 1) \times 2^{E_{\text{Cav1}}} = (\text{index} + 1) << E_{\text{Cav1}} \tag{5.13}$$

Because a null case is required, where the index caveat does shrink the region but does shrink the permissions, $E_{\text{div}} = 0$ must be possible and the maximum reduction will always be a power-of-two minus one. 'index' must be as wide as that maximum reduction, to ensure all blocks can be selected under that reduction. Ultimately this requires the size of the caveat to be one of (1 + 1), (2 + 3), (3 + 7), (4 + 15), (5 + 31), etc. Of those, I select (4 + 15) (a 4-bit E_{div} and 15-bit index) to ensure there is enough room for caveat 2 (Fig. 5.5). As a bonus, this specific permutation combined with the optimal initial region ensures that E_{Cav1} will never be less than zero, and that it is impossible to represent a zero-sized region with the first caveat. That means the offsets within the initial region can be computed purely with variable left shifts, which are cheap to compute in hardware. If the initial region was such that $E + |\mathtt{M}| + \mathtt{M_{top}} < E_{\text{div}}$ this would have to be addressed with a special case that either makes the capability invalid or outputs a zero-size range.



$$|\text{index}| = 2^{|E_{\text{div}}|} - 1$$

Figure 5.5: The structure of the index caveat.

The null case creates another problem: how is validity handled? In normal cases, a caveat that selects a sub-region outside the initial region should be rejected as malformed. However, the null case when $E_{\text{div}} = 0$ will almost never be inside the initial region. It is also possible that if E_{Cav1} is selected such that the initial region is not an exact multiple of $2^{E_{\text{Cav1}}}$, there will always be a small part at the end of the initial region that cannot be selected by this caveat. At this point the system designer may choose to hardcode $E_{\text{div}} = \text{index} = 0$ as a null case, which is simpler but limits the expressiveness of the caveat, or to specifically detect if the bottom of the selected block is in-bounds and cap the exposed region at the initial Top. (5.14) shows the latter case. Either choice makes it possible for this caveat to expose a non-power-of-two-sized region, which subsequent caveats should be aware of.

$$\left[\text{Base}_{\text{Cav1}}, \text{Top}_{\text{Cav1}}\right) = \begin{cases} \left[\text{Base} + B_{\text{Cav1}}, \text{Base} + T_{\text{Cav1}}\right) & \text{if } T_{\text{Cav1}} \leq \text{Length} \\ \left[\text{Base} + B_{\text{Cav1}}, \text{Top}\right) & \text{else if } B_{\text{Cav1}} \leq \text{Length} \\ \text{invalid} & \text{otherwise} \end{cases} \tag{5.14}$$

5.3.4 Range caveats — Bulk-to-precise

When discussing requirements I noted that caveats may be used to refine a coarse initial region with extra precision. For example, taking a page-sized region and reducing it to a precise Ethernet packet, e.g. 1-byte granularity for regions up to 10,000-bytes to match contemporary network cards (Section 5.1.1, p102). Following the trend set by the index caveat, I propose another caveat format based on decomposing the prior region into an array of power-of-two blocks, but selecting a contiguous range within that array instead of a single block. This requires two index fields rather than one, which I name x and y, and does not require a separate divisor field like the index caveat. The only purpose of the divisor field would be to request a coarser precision, using a divisor that is less than the maximum, which x and y can represent just as easily. Instead, it assumes a maximum divisor, which must be equal to the widths of x and y[9], which is subtracted from the largest power-of-two that encompasses the prior region. (5.15) assumes this is set by caveat 1, and assumes the maximum divisor is 12, making the whole caveat take up 24-bits.

[9] y is the end of an inclusive range, allowing it to represent the end of the prior region using the same bit-width as x. When represented numerically, a (y, x) pair is roughly equivalent to Verilog-style `[y:x]` indexing.

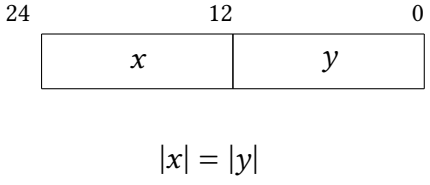


Figure 5.6: The structure of the range caveat.

$$E_{\text{Cav2}} = E_{\text{Cav1}} - 12 \tag{5.15}$$

There are a few special cases this caveat has to handle. In the simple case, where the prior region is exactly a power-of-two, a null spatial reduction is just $x = 0$, $y = 2^{E_{\text{Cav2}}} - 1$. If it's non-power-of-two, e.g. if the index caveat was null, then that reduction (and any other reduction with that value of y) would exceed the bounds. Like before, the system designer may choose to only allow this when $x = 0$, or it could clamp the top of the exposed region down to the prior region. (5.16), (5.17), and (5.18) show the latter option and handle the invalid case where $y < x$.

$$B_{\text{Cav2}} = \quad \text{x} \quad \times 2^{E_{\text{Cav2}}} = \begin{cases} \text{x} & << E_{\text{Cav2}} & E_{\text{Cav2}} \geq 0 \\ \text{x} & >> (-E_{\text{Cav2}}) & E_{\text{Cav2}} < 0 \end{cases} \tag{5.16}$$

$$T_{\text{Cav2}} = (y+1) \times 2^{E_{\text{Cav2}}} = \begin{cases} (y+1) & << E_{\text{Cav2}} & E_{\text{Cav2}} \geq 0 \\ (y+1) & >> (-E_{\text{Cav2}}) & E_{\text{Cav2}} < 0 \end{cases} \tag{5.17}$$

$$\left[\text{Base}_{\text{Cav2}}, \text{Top}_{\text{Cav2}}\right) = \begin{cases} \text{invalid} & \text{if } y < x \\ \left[\text{Base}_{\text{Cav1}} + B_{\text{Cav2}}, \text{Base}_{\text{Cav1}} + T_{\text{Cav2}}\right) & \text{else if } T_{\text{Cav2}} \leq \text{Length}_{\text{Cav1}} \\ \left[\text{Base}_{\text{Cav1}} + B_{\text{Cav2}}, \text{Top}_{\text{Cav1}}\right) & \text{else if } B_{\text{Cav2}} \leq \text{Length}_{\text{Cav1}} \\ \text{invalid} & \text{otherwise} \end{cases} \tag{5.18}$$

One other subtlety is the handling of small exponents. Say the prior region was smaller than 2^{12}, then E_{Cav2} would be negative. x and y need to be clamped so that they represent valid in-range byte addresses, which requires some of their bits to be zeroed, and we cannot trust received capability texts to zero them correctly. An elegant solution is to treat this case as a right-shift by $-E_{\text{Cav2}}$, which can be rearranged into a left-shift with a bit-extract (5.19) because the largest possible negative value of E_{Cav2} is known. This allows hardware implementations to reuse variable left-shifter hardware instead of synthesizing dedicated right-shifters, and if the left-shifter is wide enough it can even compute those right-shifts for B_{Cav2} and T_{Cav2} together in a single cycle. Incrementing y before the shift does create the possibility that the outcome B_{Cav2} and T_{Cav2} will be equal after right-shifting, resulting in a length of 0[10]. This is not a security risk, but it is unintuitive considering the initial region and index caveat encodings cannot represent a zero-sized region.

$$a >> b = (a << (b_{\max} - b)) >> b_{\max} = (a << (b_{\max} - b))[\ : b_{\max}] \tag{5.19}$$

[10] I have also tried shifting length instead of shifting top, i.e. computing $y + 1 - x$ and then shifting, which produces zero-length in slightly different cases but is otherwise equivalent.

5.3.5 An "optimal" complete format

Parts from the previous sections can be combined to arrive at an "optimal" format (Fig. 5.7). A 63-bit 64-12-8-3-2-8 initial region (Table 5.5), a 4-bit permission chain, a 4-15 index caveat, a 12-12 range caveat, and an 18-bit lease ID fit perfectly into a 128-bit capability text C with no bits to spare.

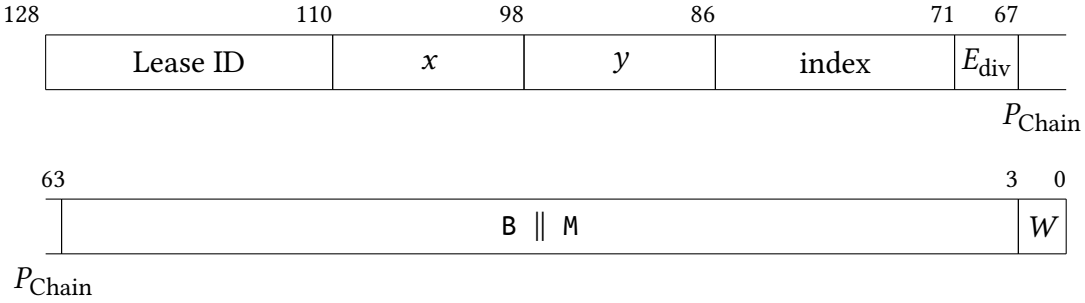


Figure 5.7: The "optimal" IOCap format. The initial region is encoded as 64-12-8-3-2-8.

W	E	Implicit $\mathtt{M}_{\mathtt{top}}$	$\lvert\mathtt{B}\rvert$	$\lvert\mathtt{M}\rvert$	Encodable Range	Used Range
0	12	0	52	8	2^{12}–2^{20}	2^{12}–2^{20}
1	12	1	52	8	2^{20}–2^{21}	2^{20}–2^{21}
2	20	0	44	16	2^{20}–2^{36}	2^{21}–2^{36}
3	20	1	44	16	2^{36}–2^{37}	2^{36}–2^{37}
4	28	0	36	24	2^{28}–2^{52}	2^{37}–2^{52}
5	28	1	36	24	2^{52}–2^{53}	2^{52}–2^{53}
6	36	0	28	32	2^{36}–2^{68}	2^{53}–2^{68}
7	36	1	28	32	2^{68}–2^{69}	2^{68}–2^{69}

Table 5.5: Details of the 64-12-8-3-2-8 format.

5.3.6 The IOCap-VirtIO complete format

Throughout the rest of this thesis I use a specific format designed for compatibility with VirtIO split-queue descriptors which differs from the "optimal" format in a number of ways. VirtIO descriptors have four fields: the 64-bit base address of the buffer, the 32-bit length of the buffer, a 16-bit flags bitfield with three defined flags including read-xor-write permissions, and a 16-bit `next` field which allows descriptor chaining. A single op may use multiple buffer descriptors chained together. For example, reading from a block device may use a read-only descriptor for the op header (e.g. which sector to read), a write-only descriptor for the device to write data into, and another write-only descriptor to indicate the op's success or failure. Each of these buffers are connected via descriptor chaining[11], where all descriptors except the last set the

[11] VirtIO also supports indirect descriptors, each of which points to a separate self-contained table with a single chain of descriptors.

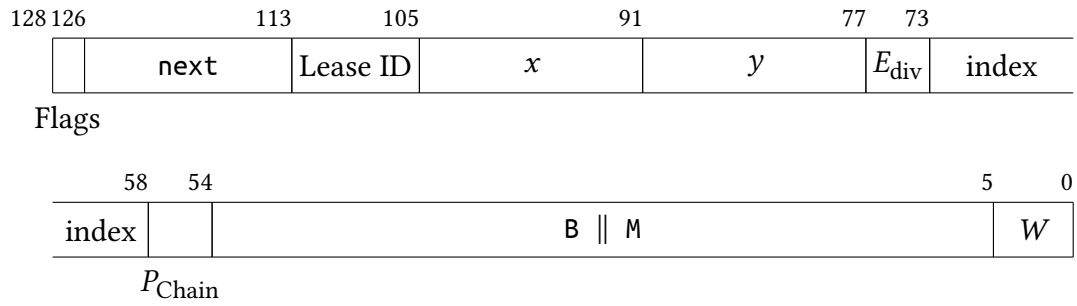


Figure 5.8: The VirtIO IOCap format. The initial region is encoded as 54-10-5-5-2-1. Fields have been ordered for readability, my hardware/software implementations use a different field order when serializing.

`NEXT` flag in the flags bitfield and set the `next` field to an index in the ring of descriptors.

Storing the `next` field and the two `NEXT` and `INDIRECT` flags inside the IOCap requires approximately 18-bits. To make room I sacrificed 10-bits from the initial region's Base and shortened the lease ID field to 8-bits. My starting point was already non-optimal so this ultimately led to a 54-bit-encoded initial region, a 4-15 index caveat, a 14-14 range caveat, a 4-bit permissions chain, and an 8-bit lease ID—leaving 15-bits for the rest and forcing the `next` field to be shortened to 13-bits. Shortening `next` technically reduces the overall capacity of each queue ring buffer. I found this to not be an issue in practice, but if it were a limiting factor it could be worked around by e.g. converting the `next` field to an offset from the current descriptor. Using an 8-bit lease ID limited the total number of leases in my implementation to 256, which was also not a problem in practice, but is really not a good trade-off to be made. I discuss this further in Section 5.6.

W	E	Implicit $\mathtt{M}_{\mathtt{top}}$	$\lvert\mathtt{B}\rvert$	$\lvert\mathtt{M}\rvert$	Encodable Range	Used Range
0	10	0	44	5	2^{10}–2^{15}	2^{10}–2^{15}
1	10	1	44	5	2^{15}–2^{16}	2^{15}–2^{16}
2	11	1	44	5	2^{16}–2^{17}	2^{16}–2^{17}
3	12	1	44	5	2^{17}–2^{18}	2^{17}–2^{18}
4	13	1	44	5	2^{18}–2^{19}	2^{18}–2^{19}
5	14	1	44	5	2^{19}–2^{20}	2^{19}–2^{20}
6	15	1	44	5	2^{20}–2^{21}	2^{20}–2^{21}
7	16	1	44	5	2^{21}–2^{22}	2^{21}–2^{22}
8	17	0	37	12	2^{17}–2^{29}	2^{22}–2^{29}
9	17	1	37	12	2^{29}–2^{30}	2^{29}–2^{30}
10	18	1	37	12	2^{30}–2^{31}	2^{30}–2^{31}
11	19	1	37	12	2^{31}–2^{32}	2^{31}–2^{32}
12	20	1	37	12	2^{32}–2^{33}	2^{32}–2^{33}
13	21	1	37	12	2^{33}–2^{34}	2^{33}–2^{34}
14	22	1	37	12	2^{34}–2^{35}	2^{34}–2^{35}
15	23	1	37	12	2^{35}–2^{36}	2^{35}–2^{36}
16	24	0	30	19	2^{24}–2^{43}	2^{36}–2^{43}
17	24	1	30	19	2^{43}–2^{44}	2^{43}–2^{44}
18	25	1	30	19	2^{44}–2^{45}	2^{44}–2^{45}
19	26	1	30	19	2^{45}–2^{46}	2^{45}–2^{46}
20	27	1	30	19	2^{46}–2^{47}	2^{46}–2^{47}
21	28	1	30	19	2^{47}–2^{48}	2^{47}–2^{48}
22	29	1	30	19	2^{48}–2^{49}	2^{48}–2^{49}
23	30	1	30	19	2^{49}–2^{50}	2^{49}–2^{50}
24	31	0	23	26	2^{31}–2^{57}	2^{50}–2^{57}
25	31	1	23	26	2^{57}–2^{58}	2^{57}–2^{58}
26	32	1	23	26	2^{58}–2^{59}	2^{58}–2^{59}
27	33	1	23	26	2^{59}–2^{60}	2^{59}–2^{60}
28	34	1	23	26	2^{60}–2^{61}	2^{60}–2^{61}
29	35	1	23	26	2^{61}–2^{62}	2^{61}–2^{62}
30	36	1	23	26	2^{62}–2^{63}	2^{62}–2^{63}
31	37	1	23	26	2^{63}–2^{64}	2^{63}–2^{64}

Table 5.6: Details of the 54-10-5-5-2-1 initial region format, which notably does not meet the 2MiB→4KiB constraint.

5.4 Implementation

While exploring this format and techniques for encoding and decoding I created multiple implementations suited for different situations. The initial design exploration in software used Rust, linked to hand-rolled and OpenSSL AES implementations, and implemented multiple iterations on the format before arriving at the IOCap-VirtIO format. This included a set of random capability generator functions, useful for stress-testing other implementations with common and edge cases, and implemented decoding using a custom bit-width library to verify the minimum required width of all arithmetic and shifting operations. This Rust is the golden model all other implementations are checked against, and is the foundation of the hardware decoder.

Along with the right-to-left-shift conversion described in (5.19) for the range caveat, this Rust implementation helped discover another trick to avoid right-shifting when validating the index caveat (5.20). It is assumed from the way that $\text{Base}_{\text{Cav1}}$ is calculated (i.e. guaranteed without overflow) that it will always be greater than the initial Base, and if it were too big then Top_{Cav1} would certainly exceed the initial Top, so $\text{Top}_{\text{Cav1}} \leq \text{Top}$ is the only necessary check. It can be simplified as shown below, and if needed $B_{\text{Cav1}} \leq \text{Length}$ could be tested similarly, but this implementation hardcodes the out-of-bounds-top case to require index $= 0$.

$$\text{Top}_{\text{Cav1}} \leq \text{Top}$$

which is equivalent to:

$$T_{\text{Cav1}} \leq \text{Length}$$

$$(\text{index} + 1) << E_{\text{Cav1}} \leq M << E$$

$$(\text{index} + 1) << (E + |\mathtt{M}| \ + \ \mathtt{M_{top}} - E_{\text{div}}) \leq M << E$$

$$(\text{index} + 1) << (|\mathtt{M}| \ + \ \mathtt{M_{top}} - E_{\text{div}}) \leq M$$

$|\mathtt{M}| \ + \ \mathtt{M_{top}} < 15$ is possible in the first split of both the IOCap-VirtIO format and the "optimal" format, so this cannot be evaluated directly with a left shift.

$$(\text{index} + 1) << (|\mathtt{M}| \ + \ \mathtt{M_{top}}) \leq M << E_{\text{div}} \tag{5.20}$$

Hardware testbenches, emulated peripherals, and the CheriFreeRTOS IOCap driver use a C interface into the Rust implementation for encoding, decoding, and randomly generating IOCaps. A separate pure-C implementation was also developed, for integration with applications that do not support Rust or static linking, that uses the same algorithms as the Rust

version without an AES implementation or bit-width checking. This library and how it was used is described further in Chapter 6. A Python decoder was also built to experiment with scheduling individual arithmetic and shift decode operations across a limited set of hardware functional units. While the Python decoder is not production-ready it has been tested against the same test vectors as the pure-C and hardware implementations.

5.4.1 Hardware decoding

I built two hardware decoders for the IOCap-VirtIO format to integrate into the larger IOCap enforcer. One was a purely combinational implementation that could decode the whole in a single cycle, though likely a long one, to be used as an initial baseline. The other used the schedule from the Python to build a compact decoder, that uses a small combinatorial frontend for the initial region, a single 65-bit shifter, and three adders (of 65-, 16-, and 9-bits respectively). The compact decoder takes 3, 6, and 8 cycles to decode an IOCap with only the initial region, one caveat, and two caveats respectively; and is completely occupied for that duration. This is close to the minimum area required for a single decoder, achieved at the cost of throughput and latency—in practice this doesn't matter, as AES ends up being the bottleneck for both (discussed further in Chapter 7).

5.4.2 Software encoding

Alongside the initial Rust model, I built a parameterizable Rust program that can freely simulate different IOCap configurations. Simplified versions of this code are included here as guidelines for future implementations.

```
// 8-bytes of information on a given width class
pub struct WidthClass {
    pub w: u8,
    pub e: u8,
    pub m_top: u8,
    pub m_bits: u8,
    pub b_bits: u8,
    pub b_align: u8,
    pub max_encodable_pow2: u8,
    pub min_used_pow2: u8,
}

// Width class information, 64-bytes total when w=3
const WIDTH_CLASSES: [WidthClass; 2^W] = ...;
// LUT mapping power-of-two lengths to width classes.
const LENGTH_POW2_TO_WIDTH_CLASS: [u8; 65] = ...;
```

Figure 5.9: Lookup tables for encoding and decoding initial regions, spanning three 64-byte cache lines in total, in Rust-esque pseudocode.

Encoding the initial region from a (Base, Length) pair is a simple matter of finding the width class for the length, rounding the base down to match the alignment, and then re-checking to see if rounding the base down has pushed the length into a new width class. Finding the width class for a given length can be easily converted into a look-up table, where the length is rounded up to a power-of-two and that power is used to index the table, and a separate table can store detailed parameters for each width class. Both tables are compact, fitting into three 64-byte cache lines total, so are very cache friendly (Fig. 5.9). As noted in Fig. 5.10 only two adjacent width classes will ever need to be checked[12]. Like CHERI Concentrate, this is guaranteed to be true for all lengths. Indeed, it is true for all lengths for all configurations of the initial region where $|\mathtt{M}| > 0$.

Encoding an index caveat is simpler (Fig. 5.11). Given an initial region with some upper bound power-of-two, the minimum possible value of E_{Cav1} can be found, and we can begin to inspect potential values of B_{Cav1} and T_{Cav1}. The goal is to find the smallest E_{Cav1} for which B_{Cav1} and T_{Cav1} encompass the target range. This can be found by computing candidate values for the smallest possible E_{Cav1}, subtracting 1 from T_{Cav1} to get a value that is inside the target range, and then shifting those candidates down (simulating an increase in E_{Cav1}) until they match. Assuming the target region was within the initial region, this is guaranteed to provide an E_{Cav1} that encompasses the target and can be reached with the index caveat. Rather than shifting the candidates in a loop, XOR can be used to find the position of the most significant bit that differs between the candidates, finding E_{Cav1} without branching. This E_{Cav1} may exceed the initial region (the 'null' caveat case noted earlier) and if the designer has required it extra code may be required to detect that and zero out index. This code has no guards against producing a negative E_{Cav1}, which may be needed if a format is selected where that is possible.

Encoding the range caveat is even simpler, as it is simply rounding and shifts. The two distinct cases of positive vs. negative E_{Cav2} are handled separately: one with right shifts and one with left. This relies on permissive decoding to allow y to exceed the prior range without affecting x. If the designer has disallowed this, extra logic will be needed to pin $x = 0$ in those cases.

[12]Rounding the base down moves it by at most $2^{A-|\mathtt{B}|}$, i.e. the base precision, and may push the length up by at most 2^{E+1} (the length is rounded initially, then the base may push it up another step). At minimum, this would have to more than *double* the length to completely skip over a width class. Assuming the length is as close as possible to the boundary between classes, it must be the maximum encodable size for its width class i.e. $2^{E+|\mathtt{M}|+\mathtt{M}_{\mathtt{top}}}$. As long as $|\mathtt{M}| > 0$ the length will not be more than doubled by an increase of 2^{E+1}, and it will never skip a width class.

```
fn initial_region(base: u64, len: u64) {
    let mut width_class = WIDTH_CLASSES[
        LENGTH_POW2_TO_WIDTH_CLASS[round_up_as_pow2(len)]
    ];

    // Align the base down, figure out what the new length must be
    let intended_top = base + len;
    let mut aligned_base = align_down_to(base, width_class.b_align);
    let mut aligned_len
        = align_up_to(intended_top - aligned_base, width_class.e);

    // Rounding the base may bump the length into the next width_class
    if aligned_len > (1 << width_class.max_encodable_pow2) {
        width_class = WIDTH_CLASSES[width_class.w + 1];

        // Compute the aligned base and length again.
        // It always fits the second time.
        aligned_base = align_down_to(base, width_class.b_align);
        aligned_len
            = align_up_to(intended_top - aligned_base, width_class.e);
    }

    // We now have our final width_class
    let encoded_b = aligned_base >> width_class.b_align;
    let encoded_m = (aligned_len >> width_class.e) - 1;
    // Mask M_top bit out of M
    let encoded_m = encoded_m & ((1 << width_class.m_bits) - 1);
    InitialRegion {
        w: width_class.w,
        b_m: (encoded_b) | (encoded_m << width_class.b_bits),
    }
}
```

Figure 5.10: Encoding the initial region using look-up tables in Rust-esque pseudocode. Lengths smaller than E_0 are mapped to the first width class and rounded up, and for simplicity it is assumed that the initial length and the rounded-up length are always representable. In practice it is technically possible for a badly-aligned base and a very large length to be rounded up past the maximum encodable length, and checks for this case must be added in real-world code.

```
fn index_caveat(initial_region, target_base: u64, target_len: u64) {
    let prior_max_e = initial_region.max_encodable_pow2;
    let min_e_cav1 = max(prior_max_e - MAX_SIZE_DIV, 0);

    let intended_base_rel = target_base - initial_region.base;
    let intended_top_rel_excl = intended_base_rel + target_len;
    let intended_top_rel_incl =
        align_up_to(intended_top_rel, min_e_cav1) - 1;

    let b_offset      = intended_base_rel     >> min_e_cav1;
    let t_offset_incl = intended_top_rel_incl >> min_e_cav1;
    let xor = b_offset ^ t_offset;
    let target_shift = 64 - xor.leading_zeros();

    IndexCaveat {
        size_div: (MAX_SIZE_DIV - target_shift),
        idx: (b_offset >> target_shift),
    }
}
```

Figure 5.11: Encoding an index caveat on top of the initial region in Rust-esque pseudocode. Assumes the target region is encompassed by the initial region.

```
fn range_caveat(prior_region, target_base: u64, target_len: u64) {
    let exponent_c1 = prior_region.exponent;
    let intended_base_rel = target_base - prior_region.base;
    let intended_top_rel  = intended_base_rel + target_len;

    let (x, y_plus_1) = if exponent_c1 >= WIDTH_RANGE {
        let slice_pow2 = exponent_c1 - WIDTH_RANGE;
        (
            intended_base_rel >> slice_pow2,
            // If rounding pushes the top outside the prior_region
            // decode will cap it
            align_up_to(intended_top_rel, slice_pow2) >> slice_pow2,
        )
    } else {
        let slice_pow2 = WIDTH_RANGE - exponent_c1;
        (
            intended_base_rel << slice_pow2,
            intended_top_rel << slice_pow2,
        )
    };

    RangeCaveat { x, y: y_plus_1 - 1 }
}
```

Figure 5.12: Encoding a range caveat on top of the prior region in Rust-esque pseudocode. Assumes the target region is encompassed by the initial region.

5.5 Evaluation

IOCaps should be able to encode both bulky, relatively coarse regions and small, precise regions. The initial region and caveats have different roles in this encoding. The "optimal" initial region represents a roughly page-aligned bulk region, the index caveat allows that bulk region to be dramatically reduced, and the range caveat allows that to be refined to a precise region. I have discussed the various precision and reduction properties of these components in isolation, but in practice they are highly dependent on each other, so in this section I evaluate them together as a whole. The generic Rust model can simulate different IOCap configurations, scan across the entire spectrum of encoded region lengths, and plot various properties of regions of different length. This program can also test arbitrary capabilities, allowing me to compare with plots of CHERI Concentrate to provide a industry-tested baseline. All tested regions use base addresses that are as misaligned as possible, to force regions into higher width classes as soon as possible, to provide pessimistic precision results. I use these results to test the two compression hypothesis facets in the summary section.

5.5.1 Initial region

We can start with the initial region. Fig. 5.13 and Fig. 5.14 plot the length of a requested initial region vs. the precision of the encoded region for the "optimal" and IOCap-VirtIO formats. Here, lower values on the Y-axis indicate the encoding can represent smaller, i.e. more precise, powers-of-two. These plots use the encoding strategies from Fig. 5.10 assuming the worst-case base alignment: an all-ones base address, not aligned to any power-of-two besides zero, which forces the regions into larger width classes and worsens their precision.

The representability constraints, which are based on RISC-V Sv57 addressing and are reproduced in Table 5.2, are also displayed as red lines. Using this, we can see that the "optimal" encoding meets these constraints. The constraint lines are always above the plotted precisions, so the encoding can always represent regions that are equally or more precise than necessary. That said, the transitions between splits at 2^{21}, 2^{37}, and 2^{53} are not quite vertical, reflecting the fact that *misaligned* regions near the length boundary are not expressible at the same precision. However 2^{12}-aligned regions of sizes up to 2^{21} *are* all encodable at 2^{12}, and so on for the other widths, so the constraints are still fulfilled for hugepages.

The IOCap-VirtIO format almost fits the constraints but falls short of the 2MiB→4KiB constraint. Even though the base encoding is precise enough, the length encoding gradually gets less precise within the first split and eventually crosses over the constraint line. To be exact, the crossing takes place between width classes 3 and 4 (Table 5.6, p119), for lengths $>2^{18}$. This poses an interesting contrast between the "optimal" encoding and a looser encoding like IOCap-VirtIO. The "optimal" encoding is designed to fulfil the first constraint entirely inside the first split, and only has two width classes per split, causing large jumps in precision between splits. A looser encoding will trade more bits for a smoother precision curve. This curve may not matter for the initial region, but it can make life easier for caveats.

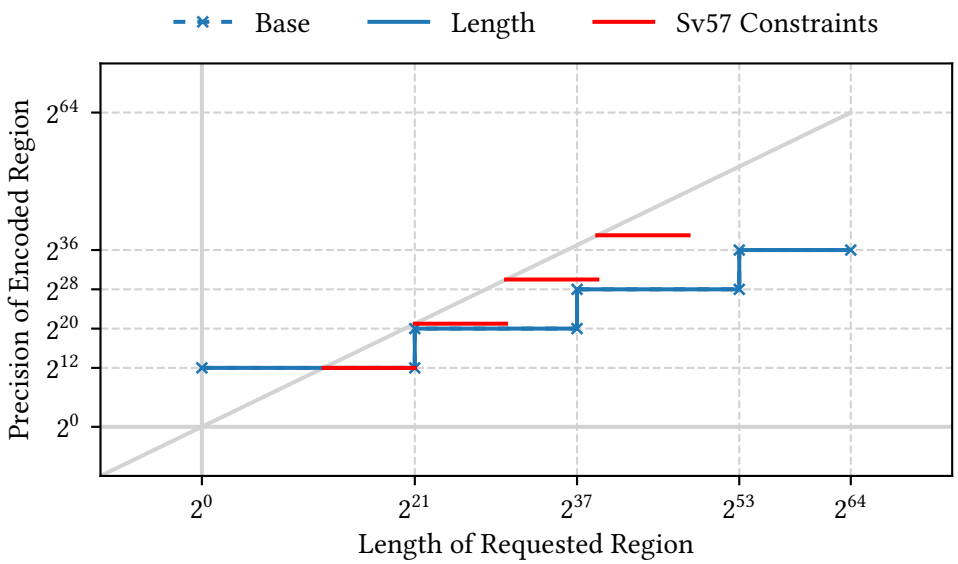


Figure 5.13: The base and length precision for the 63-bit "optimal" initial region, characterized as 64-12-8-3-2-8 in Table 5.5.

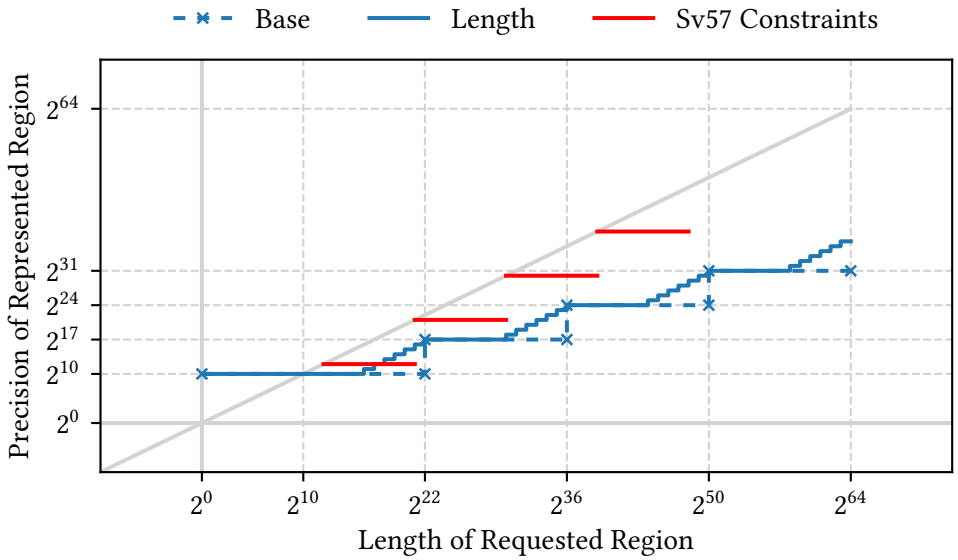


Figure 5.14: The base and length precision for the 54-bit IOCap-VirtIO initial region, characterized as 54-10-5-5-2-1 in Table 5.6. The length encoding, taken from an earlier 64-bit configuration, still encodes 64-bit lengths.

5.5.2 Caveat reduction

Caveats have two useful qualities: reduction and precision. Reduction is useful in a bulk-to-bulk scenario. Given a large region, what are the smallest sub-regions I can carve out for subordinate processors? This is plotted for the "optimal" and IOCap-VirtIO encodings in Fig. 5.15 and Fig. 5.16 respectively. These plots show the consequences of large precision jumps and a reliance on subnormal encodings, which are both more prevalent in the "optimal" encoding.

Caveat 1, the index caveat, is encoded relative to the maximum encodable power-of-two of the initial region's width class (5.11). In the "optimal" format, in the subnormal width

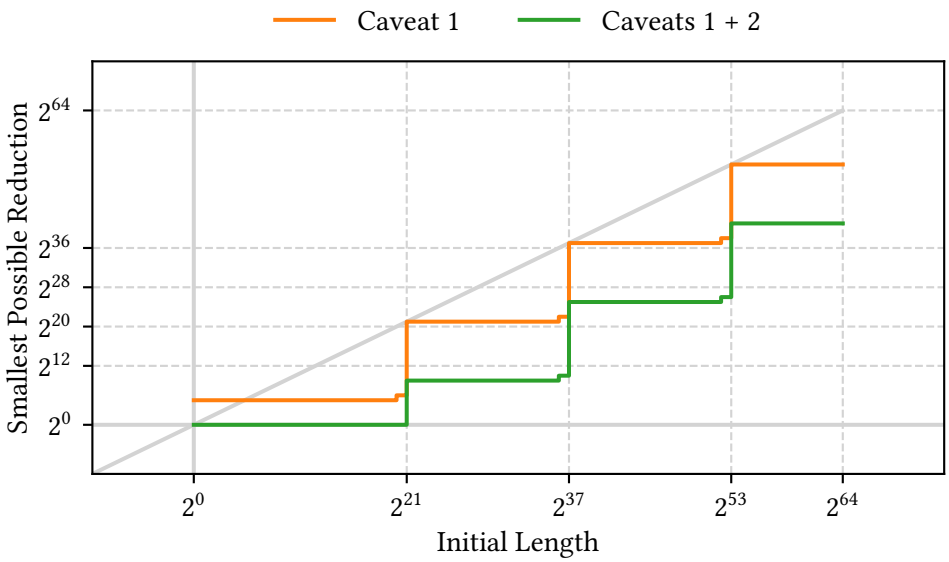


Figure 5.15: The amount of reduction achievable using one or both caveats in the "optimal" configuration. Applies a 4-15 index caveat and a 12-12 range caveat.

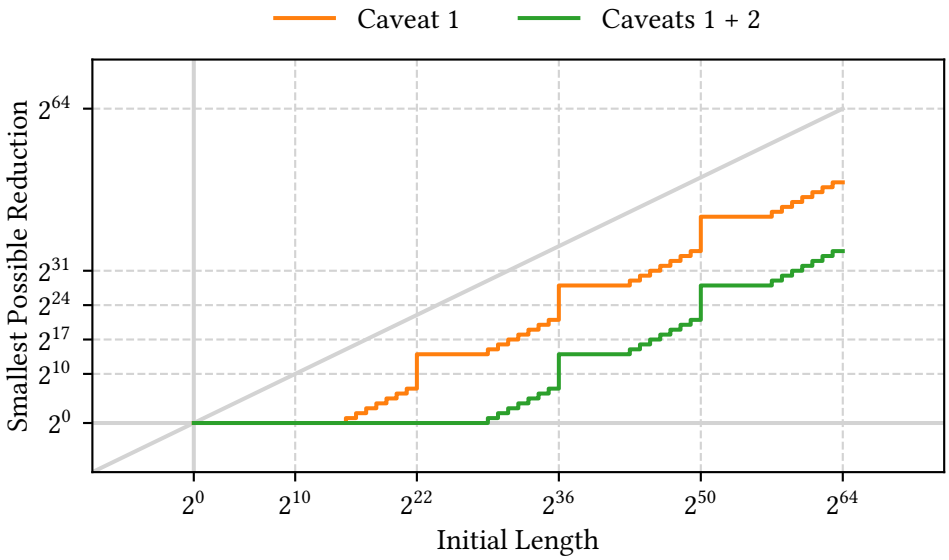


Figure 5.16: The amount of reduction achievable using one or both caveats in the IOCap-VirtIO configuration. Applies a 4-15 index caveat and a 14-14 range caveat.

classes (Table 5.5, p117), these maximum powers-of-two are up to 2^{15} higher than the minimum they're used to encode—exactly equal to the 2^{15} maximum reduction of a 4-15 index caveat, cancelling out. The normal width classes, where the maximum exponent only increases by 2^{1}, are responsible for the small stair-step artifacts in the plot preceding each precision jump. These stair-steps occur much more frequently with the looser IOCap-VirtIO encoding because it uses more normal classes.

Caveat 2, the range caveat, is a straight upgrade over the index caveat in reduction. Using it brings the total reduction down by its configured width, 2^{12} for the "optimal" format's 12-12 configuration, and 2^{14} for IOCap-VirtIO. However, reduction is not the main purpose of the range caveat, as we shall see in the next section.

5.5.3 Caveat precision

Finally, I consider caveat precision. If I start with a target region I wish to expose, and the initial region cannot encode it exactly, how close can I get using caveats? Like before, this evaluation has to contend with extremely misaligned base addresses, which throws off the index caveat in particular.

As I noted in Section 5.3.3, the index caveat is effectively a scaled-down version of the M-Machine capability, and it inherits some fussiness over misaligned addresses. To demonstrate, consider a small two-byte region that crosses a 2^{12} address boundary. The initial region, assuming "optimal" configuration, can only allocate a 2^{13}-sized region to cover both sides of the boundary. Our target region starts $2^{12} - 1$ bytes from the initial Base and ends $2^{12} + 1$ bytes from Base. The smallest power-of-two sized and aligned region that covers that sub-range is 2^{13} again, starting exactly at the Base, and this is the tightest possible precision we can extract from the index caveat. Because this is entirely relative to the initial range it retains the 2^{12} base precision while being forced into a larger 2^{13} length precision. Lengths beyond that are always going to be more precisely encoded by the initial region than a straight power-of-two, so are forced to use null index caveats, resulting in the $y = x$ line shown in Fig. 5.17.

Just like for reduction, the range caveat is a flat upgrade over the index caveat in terms of precision. However, when the index caveat restricts precision, the overall index + range precision is not always superior to the initial region. Consider Fig. 5.17. The length precision for the range caveat is the index caveat precision (i.e. the intended length itself, rounded up) minus the range caveat's constant reduction factor (12 for the "optimal" configuration). As already established, the minimum index caveat precision for ranges greater than 1-byte long is 2^{13}, limiting the range caveat's minimum precision to $2^1 = 2$-bytes. When the initially encoded length is greater than or equal to 2^{E+12}, as is possible in width classes 2 and 4–7 (Table 5.5, p117), the initial region is encoded with better precision than these caveats can manage (albeit only in the worst alignment cases). Overall the precision with both enabled is usually just superior to CHERI Concentrate (Fig. 5.18) except for the low-end, where CHERI supports 1-byte precision and the "optimal" format only supports 2-byte. That is not a great result, but there is hope.

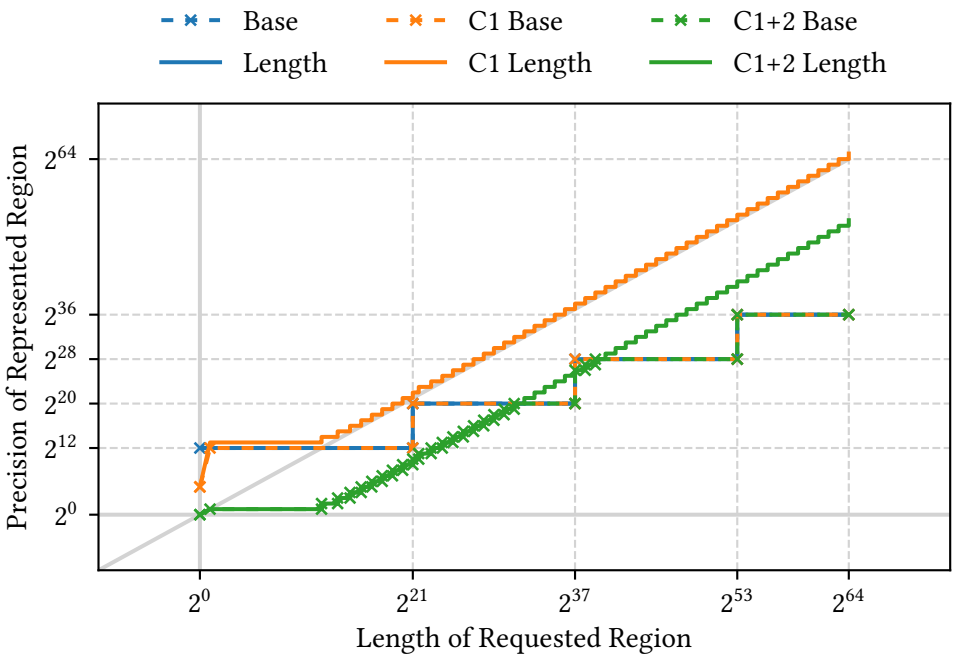


Figure 5.17: The worst-case base and length precision available with caveat(s) enabled in the "optimal" IOCap configuration.

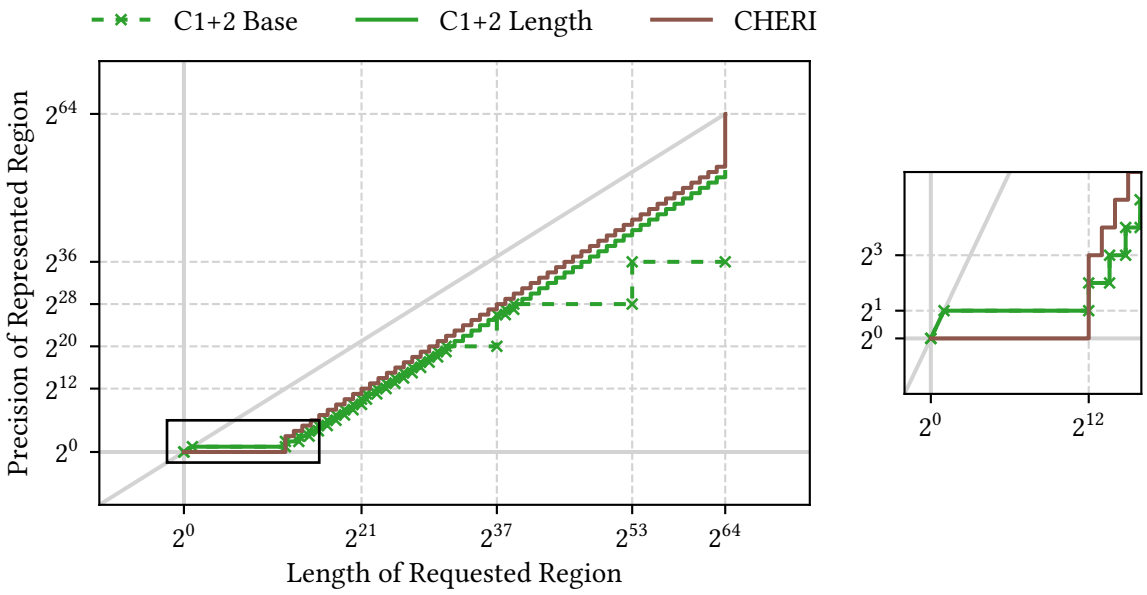


Figure 5.18: The worst-case base and length precision available with both caveats enabled in the "optimal" IOCap configuration, as compared to the CHERI ISAv9 [216] format.

5.5.4 Improving the index caveat

The index caveat has shown to be a bottleneck for both precision and reduction with misaligned bases. It limits reduction by encoding E_{div} relative to the maximum encodable length for a given width class, which causes large reduction jumps in subnormal cases. An alternate version could compute its exponent relative to the actual initial length (5.21), rather than the maximum possible for that width class, which would in theory provide better reduction without affecting precision.

$$E_{\text{Cav1}} = E + \texttt{round-up-to-pow2}(M) - E_{\text{div}} \tag{5.21}$$

The index caveat also could not precisely encode small regions, which also affected the final precision of the range caveat, because of the M-Machine pathological alignment failure. Giving index one extra bit, i.e. allowing the base to be aligned to *half* the length, avoids the pathological case and allows smaller regions to be encoded (Fig. 5.20). We can make up for the extra bit by decreasing the maximum E_{div}, keeping the total size of the caveat constant at the expense of reduction power. While I have not implemented these improvements in hardware, so I have not evaluated potential decoding difficulties, I have simulated a version of the index caveat with both improvements enabled to demonstrate the potential impact on reduction (Fig. 5.19) and precision (Fig. 5.20) for the "optimal" format.

The total reduction potential is slightly less than before due to the precision optimization, but is now much smoother. Because the precision optimization is most relevant for small lengths, future implementations could selectively enable and disable it depending on the width class for maximum bulk reduction. The combination of the precision and reduction optimizations work wonders for total precision, both for the index caveat (which can now represent much smaller regions) and the range caveat. The overall precision with the improved index caveat is now entirely equal to or superior to CHERI Concentrate, save for the 1-bit disadvantage shown in the inset for 2^{11}–2^{12}.

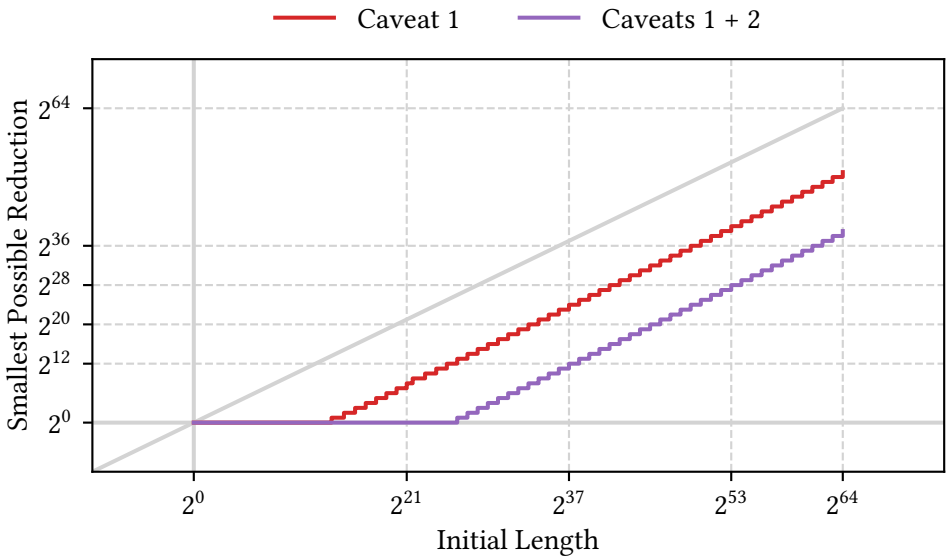


Figure 5.19: The amount of reduction achievable using one or both caveats in the "optimal" configuration with improved index caveat.

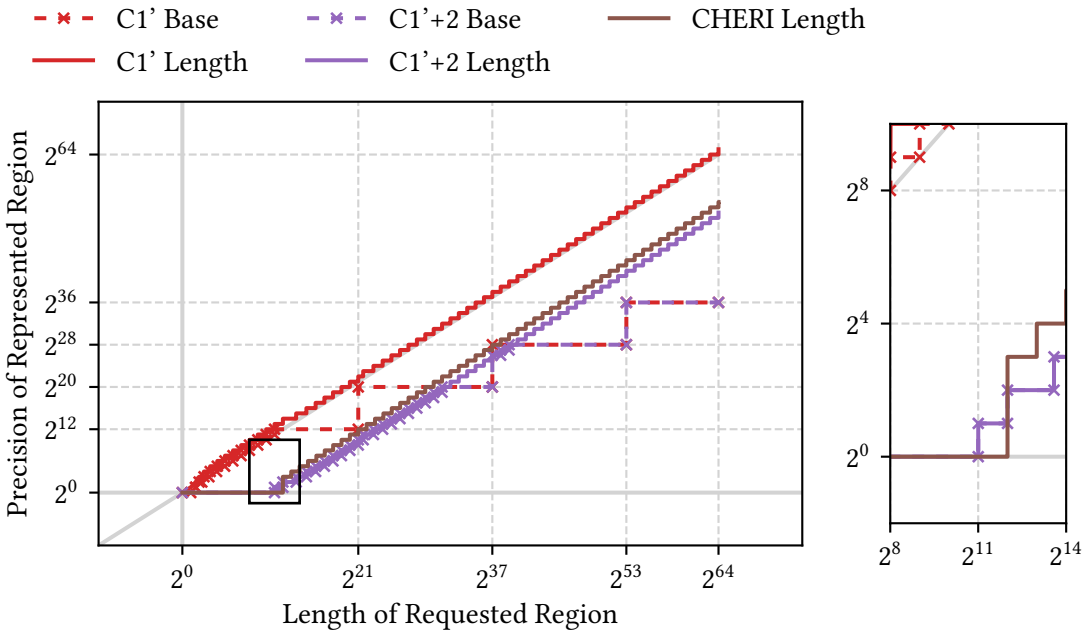


Figure 5.20: The worst-case base and length precision available with an improved index caveat.

5.6 Summary

Hypothesis 2

The IOCap system can provide spatial safety at sub-4KiB precision using a 128-bit capability format.

In this chapter I examined the requirements for a compressed IOCap text format, described the methods I used to arrive at the format I used in the following chapters, and evaluated this format as well as a more "optimal" format to see how far it can be pushed. As it turns out this not only surpasses the IOMMU's 4KiB precision, but also reaches CHERI-equivalent precision, which I consider a great signal of future viability.

Facet 2.1

An IOCap format can compress a reasonably precise 64-bit address range, access permissions, a lease ID, and multiple caveats to the memory range in 128-bits.

This is true. I have presented three 128-bit encodings (IOCap-VirtIO, "optimal", and "optimal" with improved index caveat) that can scale from individual descriptors to bulk regions. The improved "optimal" format meets most the requirements laid out in Section 5.1.5. The only one it misses is the jumbo Ethernet packet precision—it can represent regions up to 2KiB at 1-byte granularity, enough for normal Ethernet, but not quite the 10KiB needed for jumbo packets—but this is probably still fine in practice. Bulk DMA protocols like PCIe access data in coarser blocks, for example 64B cache lines or PCIe's 4B minimum granularity. The improved "optimal" format can represent 10KiB at 4B precision, and in the worst case metadata like the exact length can be stored inside the exposed region instead of the IOCap. If absolutely needed, extra bits to extend the range caveat could be found by shrinking the index caveat, reducing the precision of the initial region, or shrinking the lease ID—as this chapter has shown, there are many options available when designing formats to fulfil unique constraints.

Facet 2.2

An IOCap format can compress a usable subset of those fields in 64-bits.

This should also be true, depending on the definition of "reduced" and "reasonably precise". Assuming we have the same 18-bit lease ID requirement, and 2-bits for R/W/RW permissions, we are left with 44-bits to encode an initial region. The initial region is highly configurable but is limited by minimum precision constraints. A 48-bit address space meeting the RISC-V Sv57 constraints (Section 5.1.5, p104) must use at least 44-bits for page-precision, and in practice requires 3-bits extra for the width class e.g. 48-12-8-3-2-5. Losing 3-bits of lease ID and a restriction to 48-bit addressing may be acceptable in resource-constrained situations. Extremely constrained systems such as embedded systems or microcontrollers may only have 32-bit addressing, in which case 1-byte precision up to 2^{12} regions is easily attained in 45-bits

W	E	Implicit $\mathtt{M_{top}}$	$\lvert\mathtt{B}\rvert$	$\lvert\mathtt{M}\rvert$	Encodable Range	Used Range
0	12	0	36	8	2^{12}–2^{20}	2^{12}–2^{20}
1	12	1	36	8	2^{20}–2^{21}	2^{20}–2^{21}
2	17	0	31	13	2^{17}–2^{30}	2^{21}–2^{30}
3	17	1	31	13	2^{30}–2^{31}	2^{30}–2^{31}
4	22	0	26	18	2^{22}–2^{40}	2^{31}–2^{40}
5	22	1	26	18	2^{40}–2^{41}	2^{40}–2^{41}
6	27	0	21	23	2^{27}–2^{50}	2^{41}–2^{50}
7	27	1	21	23	2^{50}–2^{51}	2^{50}–2^{51}

Table 5.7: Details of the 48-12-8-3-2-5 format.

W	E	Implicit $\mathtt{M_{top}}$	$\lvert\mathtt{B}\rvert$	$\lvert\mathtt{M}\rvert$	Encodable Range	Used Range
0	0	0	32	11	2^{0}–2^{11}	2^{0}–2^{11}
1	0	1	32	11	2^{11}–2^{12}	2^{11}–2^{12}
2	10	0	22	21	2^{10}–2^{31}	2^{12}–2^{31}
3	10	1	22	21	2^{31}–2^{32}	2^{31}–2^{32}

Table 5.8: Details of the 32-0-11-2-1-10 format.

with configurations like 32-0-11-2-1-10. Any system that requires more than 48-bit addressing is likely SOL[13] and should use 128-bit IOCaps instead.

Facet 2.3

A 128-bit IOCap format can be sufficiently detailed to entirely replace buffer descriptors in protocols such as IOCap-VirtIO.

The original pitch for IOCaps was to replace the IOMMU *and* to replace buffer descriptors. The IOMMU is only concerned with the exposed region, but buffer descriptors may also want to encode protocol-specific metadata such as VirtIO's `next` field. The Intel network controller I examined earlier also includes per-packet metadata, such as the length of various network headers within the packet, taking up a full 128-bits when combined with the directly encoded base and length [221, §10.5.3.1.1]. Unfortunately it is now apparent that even an "optimal" multi-caveat format may not have enough space for all desired metadata. At this point the designer can make a choice: store metadata elsewhere, perhaps inside the exposed region or as extra bytes alongside the IOCap, in which case the hypothesis holds; or sacrifice IOCap fields to make room for metadata.

The IOCap-VirtIO format was meant for design space exploration, so chose to sacrifice the initial region and lease ID instead of caveats, but that probably isn't the right tradeoff to make in practice. Sacrificing top bits can only get you around 8-bits (from 64 to 56) before CPU-accessible regions become impossible to represent—all RISC-V virtual addressing modes

[13] Stupendously, Out of Luck.

greater than 32-bit use 56-bit physical addresses [233, §12.4–12.6]. Sacrificing the lease ID would limit the number of outstanding leases, which could be an impediment (though not an insurmountable one) for large, high-performance systems (Chapter 4). If one is designing an IOCap without broad compatibility, that is meant to only ever replace a descriptor for a specific protocol, then caveats are likely unnecessary and those bits can be used for specific metadata. For instance, a 64-0-13-3-2-9 initial region would encode 1-byte precision within 16KiB in a 64-bit address space in 80-bits, leaving 28-bits for metadata after an 18-bit lease ID and 2-bit permissions.

In the most generic case, where at least one level of bulk reduction and the ability to represent precise regions are both necessary, the caveats must be kept and the metadata must be moved. The most obvious application is that of CXL or other memory expansion technologies being used to share memory with multiple peripherals (e.g. GPUs [77]) within multiple systems. For example: a CXL GFAM delegates memory to a host with an IOCap, the host delegates a bulk sub-region for e.g. a GPU to use, and the GPU then delegates a subset of that memory to an NVMe peripheral to load and store directly. If there truly is no need for one of the caveats, it may be useful to drop one of them for extra metadata bits, but I would argue it's not sensible to limit the flexibility of the format at this point. As this is an emerging ecosystem, there is room for experimentation before industry standardizes a solution, but that standard will be locked in at the hardware level and 20-bits seems like a small gain for a potentially huge flexibility loss. The real challenge at that point is transmitting 32-byte IOCaps with every request without introducing too much extra latency, which I address for bulk DMA in Chapter 7, but is a tougher ask for cache-line-based protocols like CXL.

5.6.1 Future improvements

Alongside my proposed improvements to the index caveat, it may be beneficial to allow multiple caveats to be applied and authenticated with a single MAC. Macaroons already implement this, and it would most benefit IOCaps that apply all caveats up front—e.g. the aforementioned single-byte VirtIO status descriptors. The difficulty would be fitting extra states of the permissions chain, encoding the difference between two caveats with a single MAC instead of a three-MAC chain.

CHAPTER 6

IOCaps in software

With the MAC function and text compression established, IOCaps can be tested in practice. I have implemented software IOCap support in two operating systems, CheriFreeRTOS [37, 156] and CheriBSD [38, 253], and integrated IOCap support in the CHERI-QEMU full-system emulator [39]. CheriFreeRTOS is a CHERI-compatible fork of the real-time operating system FreeRTOS, targeted at microcontrollers, CheriBSD is a fork of FreeBSD, and CHERI-QEMU is a fork of QEMU [254]. For the purposes of this thesis CheriBSD is roughly equivalent to FreeBSD—their drivers and DMA APIs are largely identical—but I target CheriBSD specifically to determine if there are any conflicts between CHERI and IOCaps. This chapter considers two facets: how easily VirtIO can be adapted to replace its descriptor format with IOCaps (Facet 3.1), and how easily complex operating systems such as CheriBSD can be extended with IOCaps for DMA protection (Facet 3.2).

6.1 IOCaps in QEMU

Implementing IOCaps in QEMU requires simulating the two hardware units described in Chapter 3—the lease manager and enforcer—and tweaking the VirtIO protocol to use IOCaps as descriptors. These changes were built on top of CHERI-QEMU. I first handle the lease manager and design its MMIO interface for the host CPU.

The host must be able to upload lease keys, open leases, close leases, and poll lease status. It must also be possible for multiple cores to perform these actions at the same time, though not necessarily for the same lease IDs, as is necessary to manage independent queues and maximize throughput. I therefore designed the MMIO address space as shown in Fig. 6.1. The writable space is split in half: 16-bytes per lease for the key, and another 16-bytes for each lease's status. In emulation, each lease's status is simply either 'valid' or 'invalid', and writing `1` or `0` to the relevant MMIO space will switch them instantly—though in hardware a third state 'invalid-txns-pending' is also needed as I discuss in Chapter 7. That would fit in 1-byte, which is much less than 16-bytes, so there is certainly scope to reduce the amount of MMIO space allocated for statuses; though I would note that address space is relatively free and it is

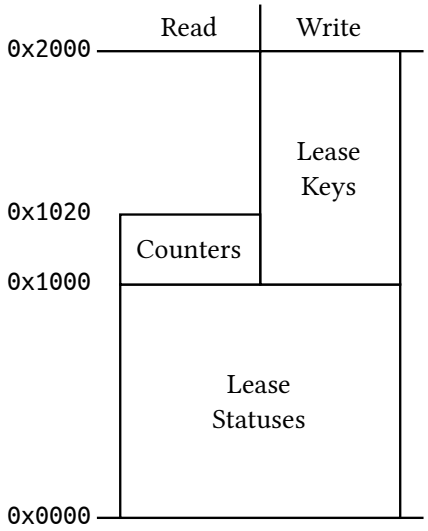


Figure 6.1: The lease manager MMIO space as designed for 256 leases.

important to ensure the CPU can easily issue writes without accidentally touching two lease's data. For example, reducing the status to 1-byte could be fine, as CPU ISAs support 1-byte memory accesses [255, §2.6], but 4-byte may be more convenient to write drivers for and is used in similar devices [16, §6.2]. Sub-byte would be impossible.

Write-only registers are almost all the lease manager needs. Ideally, the host would simply write 16-bytes of random key data, write 'valid' to open the lease, then later on write 'invalid' to close it. However, it's convenient for the host to be able to read lease status, and actually necessary once the third state gets involved. Technically it would also be fine for the lease keys to be exposed as readable, assuming only authorized host software can access it, but to provide defence in depth I only allow lease keys to be read by the enforcer hardware. Instead, the first few words of the lease key space will return performance counters from the enforcer when read. These performance counters are simple—the numbers of read/write DMAs that were accepted/rejected—but provide enough of a signal for host workloads to detect problems and see the system is working. In a full implementation the host should receive interrupts or have some other indication of which peripherals have had DMAs rejected, so the host can cut them off or otherwise react.

The actual implementation of the lease manager is exactly as simple as one would think. My implementation supports 256 leases to match the IOCap-VirtIO format (Section 5.3.6, p117), which was sufficient for the tested cases. Accesses that straddle multiple leases are rejected, and writes to lease keys while they are exposed are also rejected, but otherwise all accesses work. Leases can be opened and closed instantly (the latter is not true for hardware, but it is simple enough to account for that in drivers). The enforcer is modelled as a static function that QEMU peripherals can call to check accesses. This function pulls the relevant key out of the lease manager (assumed to be a per-system singleton) and verifies the IOCap using `libccap`—a C library that encodes and decodes the IOCap-VirtIO format. `libccap` is highly adaptable and can be parameterized with different AES and logging implementations. In this case QEMU's preexisting AES implementation and error logging are used, and debug logging is disabled.

6.1.1 IOCap-VirtIO in QEMU

QEMU can use the VirtIO protocol to interact with peripherals, which supports multiple transports e.g. PCI or MMIO, and queues which are 'split' (with separate descriptor, submission, and completion queues) or 'packed' (where the descriptors are read and written by the peripheral to signal completion) [26, §2.6–2.7]. I chose to adapt the MMIO-based split-queue variant, as that is also used in the CHERI group's FPGA SoCs (discussed further in Section 7.5, p164). VirtIO has to access persistent queues and ephemeral op regions in host memory. This requires two changes at the protocol level: new MMIO registers need to be added for the host to write in a queue's IOCap text and MAC, which is trivial, and the op descriptors need to be converted to IOCaps. Both changes are gated behind a VirtIO feature bit, which is advertised by the peripheral and then accepted by the host in a negotiation process during initialization [26, §2.2], just like the packed queue representation. IOCaps are effectively a new form of queue representation based on split queues. They could be extended to packed queues but I haven't thoroughly investigated this. One potential obstacle is that packed queues use more flags to signal available/completed ops, which the current IOCap-VirtIO format does not support.

Inside QEMU these changes manifest much like the changes from split to packed queues. Some functionality, particularly reading ops out from queues, was split into separate IOCap-specific functions. All branches based on split vs. packed were extended to split vs. packed vs. IOCap. IOCaps were included inside the (de)serialization routines for VirtIO peripherals. Finally, calls out to the lease manager check function were added whenever an IOCap is dereferenced: both when reading from the queue, and when accessing op regions. These are completely agnostic to the peripheral functionality sitting on top of VirtIO: networking, storage, random entropy, etc. can all use IOCap-VirtIO.

6.1.2 IOCap-VirtIO in CheriFreeRTOS

I first ported CheriFreeRTOS to use IOCap-VirtIO as a proof-of-concept before moving to CheriBSD. The repository is built as a collection of submodules, with two of relevance: `FreeRTOS-libvirtio`, which provides `virtio-blk` storage and `virtio-net` network support for other submodules; and the `RISC-V-Generic` demo programs which I used to validate functionality. I updated the demo program toolchain to compile in IOCap support when needed and support the CHERI group's hardware SoC, and tweaked one demo program to indicate when IOCaps were enabled. The meat of the changes were in `libvirtio`.

`libvirtio` supports PCI and MMIO-based VirtIO but assumes split queues. Modifying it to support IOCaps when the feature was negotiated was thus slightly more invasive than QEMU, but was certainly manageable. The library was already structured to use a centralized function to encode descriptors, which was easily amended to call `libccap` as needed. `libccap` was configured to use the software AES implementation from the `mbedtls` library bundled with CheriFreeRTOS.

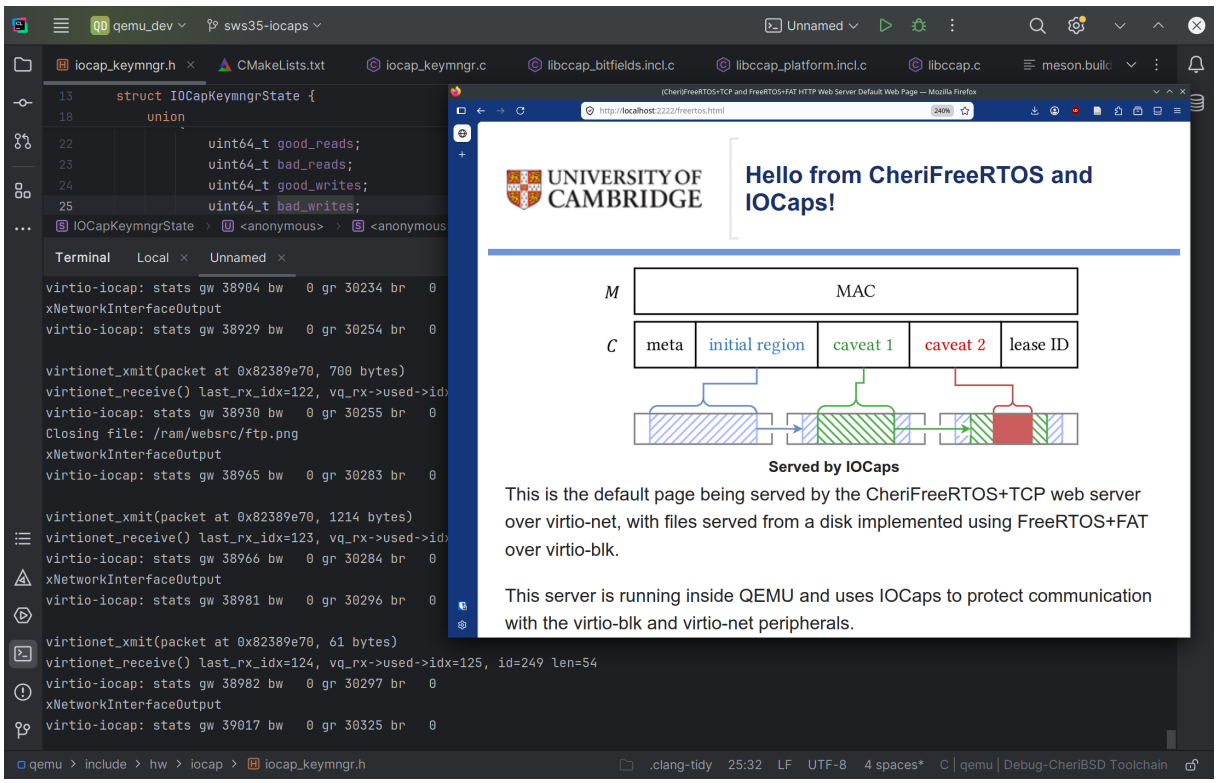


Figure 6.2: CheriFreeRTOS, running under QEMU, serving a static web page over TCP using IOCap-VirtIO. `virtio-iocap` messages in the console window show the emulated lease manager performance counters. The good-read (`gr`) and good-write (`gw`) counters both increase, indicating the host is correctly minting IOCaps and the peripherals are correctly dereferencing them.

As CheriFreeRTOS was a proof of concept, I did not implement full temporal security. Instead, I simply open two leases at boot: a lease for the persistent queue, and a lease for ephemeral ops; both using static keys. These leases are never closed. Closing and reopening leases for full temporal security would require generating new keys, which would need a source of cryptographically-secure random numbers, which the RISC-V demo did not have readily available. With these changes, the demo programs can take advantage of peripherals that support IOCaps without any demo-specific changes (e.g. Fig. 6.2).

6.1.3 Secure randomness sources

My experience with CheriFreeRTOS raises an interesting question: where can secure randomness come from in an embedded system? QEMU exposes a VirtIO entropy device, meant for exactly this purpose [26, §5.4], and `libvirtio` could be extended to support it—though there is some irony in relying on one VirtIO peripheral to secure all the others. OpenTitan's 'ENTROPY_SRC' [256] and 'CSRNG' [257] are examples of trusted hardware IP blocks that allow true random number generation—if built into an SoC, CheriFreeRTOS could use them to generate keys. RISC-V's scalar cryptography extensions also include an interface to an entropy source built-in to the CPU [255, §31.5].

Alternatively, it may be tempting to integrate a secure random generator into the lease manager itself, and have it automatically fill in keys when new leases are opened. This would

require the keys to be readable by the host, effectively inverting the current memory map where they are write-only, or this principle could be extended further: the lease manager could handle all IOCap encryption and entirely hide the keys from the host. Hiding the keys prevents host software from accidentally copying keys into exposed regions, but would not entirely prevent IOCap forgery—if a hostile actor asks for an almighty IOCap, the lease manager may not have enough information to deny it—and would require more persistent state to hold IOCaps-in-progress. Managing this persistent state could become a bottleneck for systems with high IOCap demand, e.g. systems with many threads. It may be a feasible approach for resource constrained systems, where demand would be smaller and more manageable, but I leave investigating this for future work.

In either case, it is worth noting that IOCaps do require entropy in the first place, and entropy can be particularly scarce at boot time. Depending on the available entropy sources, IOCap drivers may need to delay certain parts of the boot process until enough entropy is found. Conversely, booting likely has much lighter I/O requirements than usual—e.g. a high-performance network workloads won't begin until the system is brought up—so likely won't need so much entropy in the first place. This is heavily dependent on the structure of the surrounding system, and I consider it out of scope for this thesis.

6.2 IOCaps in CheriBSD

The aim of this section is to identify pain points and key insights for integrating IOCaps with a more complex operating system such as FreeBSD or Linux. Both of those operating systems have their own dedicated DMA APIs [47, 48], internal APIs for exposing memory regions to peripherals, where IOCaps would ideally be integrated. I chose FreeBSD over Linux because the CHERI group at the University of Cambridge has significant FreeBSD expertise, and I picked CheriBSD (the CHERI-compatible fork) to see if CHERI and IOCaps conflict. The CheriBSD DMA API [258] is, as far as I can tell, identical to FreeBSD's [48, 259], so my work and insights here should transfer directly. Linux's API is quite different, which I discuss with my other observations in Section 6.3.

6.2.1 The CheriBSD DMA API

The CheriBSD DMA API [258] is used by peripheral drivers to expose memory regions, which may have been allocated by various different subsystems, to peripherals. It is designed for translation-based mechanisms such as the IOMMU, and as such focuses on translating addresses from the kernel's virtual memory space into the physical space—more accurately, the 'bus' space. Peripheral drivers are collected in a hierarchy, either attached by searching at runtime or specified at boot in the 'device tree', and peripherals using the same transport (e.g. PCIe devices) are collected under a single 'bus' device (e.g. for the PCIe root complex). This bus device defines the address space any child device drivers expose memory into, and many can exist in the system—allowing CheriBSD to manage the numerous different address spaces

present in a modern system [46].

CheriBSD uses three kinds of object to manage DMA regions, shown in Fig. 6.3. Bus devices create tags, tags create mapping objects, and mapping objects are loaded and unloaded to expose scatter-gather lists of regions. A peripheral device driver starts by calling `bus_get_dma_tag()` to acquire a `bus_dma_tag_t` from the closest parent bus device, which expresses various constraints that bus may have on which regions can be exposed. A bus may have a minimum alignment or a minimum address range. For instance, peripherals may only support 32-bit addressing, and regions exposed to them must be inside the bottom 4GiB of the space. If a peripheral has extra constraints on top of the ones imposed by the bus, `bus_dma_tag_create()` can create a new tag that inherits from the bus' tag with more restrictions.

Once the tag is complete, the driver can create mapping objects. These can be allocated up front, and do not expose anything until they are *loaded* with a region (potentially a scatter-gather list of regions) to expose. There are different `bus_dmamap_load_*()` functions to handle regions from different sources, and all of them return a list of `bus_dma_segment_t` via callback: the exposed bus addresses. If the regions are not directly exposable to the peripheral, as determined by the tag's restrictions, 'bounce buffers' can be allocated. Bounce buffers are regions allocated in bus-space, compatible with the requested tag, which are exposed instead of the original regions. `bus_dmamap_sync()` is used by all drivers to ensure memory consistency—e.g. if a peripheral DMAs directly into cacheable memory, it will invalidate the host's cache so it sees the new data—and will automatically copy data into/out of bounce buffers if they are used. After the peripheral has finished with the regions, and the region is properly synced, the mapping is *unloaded* to unexpose the region and the mapping object can be reused to expose something else.

6.2.1.1 `bus_dmamem`

The `bus_dmamem_alloc()` function, not pictured, directly allocates bus-space memory compatible with a given tag which can later be `bus_dmamap_load()`ed into a mapping. This memory is not necessarily accessible from the kernel-virtual memory space, and can be mapped and unmapped into that space with `bus_dmamem_{map,unmap}()` so the host can read or write its data. After the memory is unloaded from the mapping and unexposed, it can be `bus_dmamem_free()`d.

Function	Lifecycle
`bus_dmamem_alloc()`	(1) Host allocates memory
`bus_dmamap_load_*()`	(2) Host exposes memory
`bus_dmamap_unload()`	(3a) Host requests unexposure
N/A	(3b) Memory is unexposed
`bus_dmamem_free()`	(4) Host frees memory

Table 6.1: How CheriBSD normally maps to the practical lifecycle of host memory described in Table 2.2, without IOCaps. CheriBSD does not expose (3b), but it uses strict invalidation so hypothetically it is not necessary. `bus_dmamem` is used for illustration and is not required to use IOCaps.

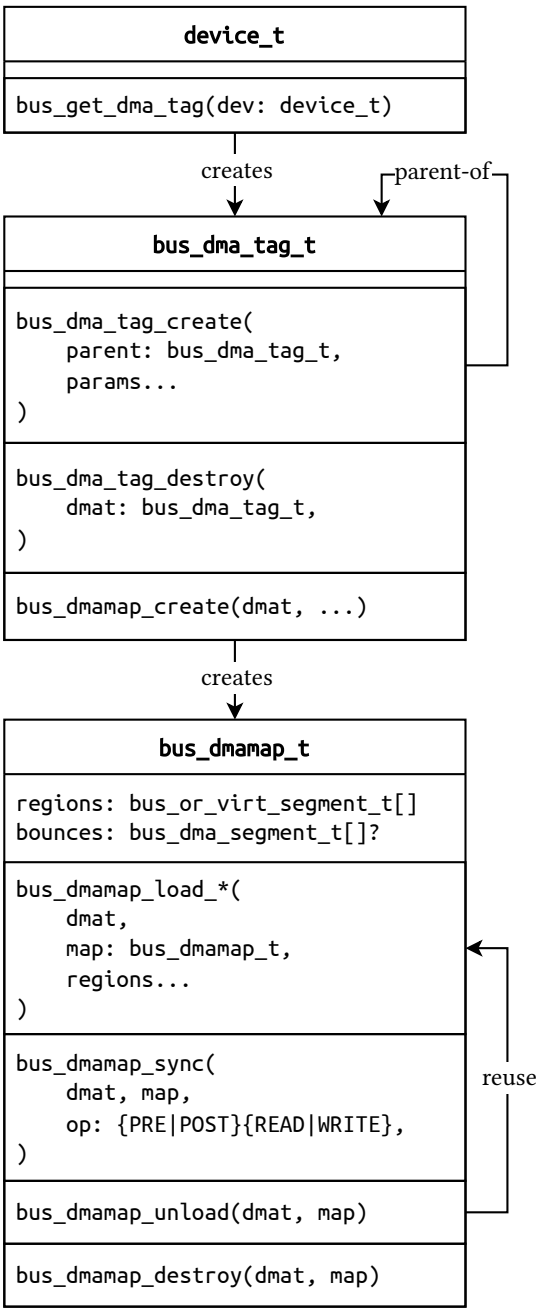


Figure 6.3: DMA mapping structures and their relationships in standard CheriBSD. Structure fields and function arguments are illustrative, not complete.

6.2.1.2 Invalidation and the IOMMU

Mapping objects define the lifetime of exposed regions and map onto the memory lifecycle I originally discussed in Chapter 2—reproduced in Table 6.1. CheriBSD assumes synchronous revocation, so calls to `bus_dmamap_unload()` should not return until the invalidation has gone through, and will be delayed if invalidation is slow. CheriBSD's x86, Arm, and RISC-V drivers allow DMA tags and mappings to be polymorphic by using a virtual function table within a common base implementation struct. IOMMU drivers are implemented as bus device drivers which give downstream peripherals a custom implementation of the basic tag. These subclasses use custom implementations for the DMA tag and DMA mapping functions to update page tables and perform synchronous invalidations as necessary. I should say *mostly* synchronous—CheriBSD's i386 PCIe IOMMU system driver enables a flag that forces asynchronous invalidation. The documentation[1] states verbatim "The unfortunate consequence is that the DMA requests can be served some time after the `bus_dmamap_unload()` call returned." These custom tags also do not inherit the default bounce buffer behaviour, and always expose pages directly.

6.2.2 An IOCap DMA API

Adapting this system to IOCaps requires a few changes (Fig. 6.4). First, I wrote a device driver for the lease manager, named `iocap_keymngr`, which tracks open leases, handles their keys, and allocates lease IDs to groups. This bus device is instantiated automatically if a lease manager is detected in the device tree, and overrides `bus_get_dma_tag()` to return a special kind of DMA tag to downstream drivers.

The underlying structure of `bus_dma_tag_t` on all relevant modern systems (Arm, x86, RISC-V) includes a virtual function table, which I manually overwrite to create two distinct new types of tag: an IOCap *refinable* tag object, and an IOCap *lease group* tag object. The IOCap refinable tag is returned from `bus_get_dma_tag()` and cannot be used to mint IOCaps, but it inherits the basic tag behaviours including inheritance and bounce buffering for maximum flexibility. In practice, IOCaps should obviate the need for bounce buffering and wouldn't need to support it. When a peripheral driver wants to allocate a group of lease IDs, it refines the refinable tag into a lease group tag by passing it to a new API function `bus_dma_tag_refine_to_iocap_group()` with a specific revocation strategy (e.g. ephemeral ops, $N_L = 4$, $L_{\text{ops}} = 16$). Each time a new lease group tag is created, the `iocap_keymngr` reserves a set of lease IDs for it. The CheriBSD driver implements two lease allocation strategies: the single-lease case for persistent regions (Section 4.5.1.1, p84), which can be used for epoch-based revocation, and the ephemeral ops strategy where $N_L = 4$ and L_{ops} is configurable (Section 4.5.1.2, p85).

Memory exposed using the standard `bus_dmamap_t` functions via a lease group tag will automatically select a lease ID according to the strategy, initialize the key over MMIO on first usage, and revoke the lease once all the relevant regions for that ID are unloaded. Lease keys

[1] `sys/dev/iommu/busdma_iommu.c`, line 852.

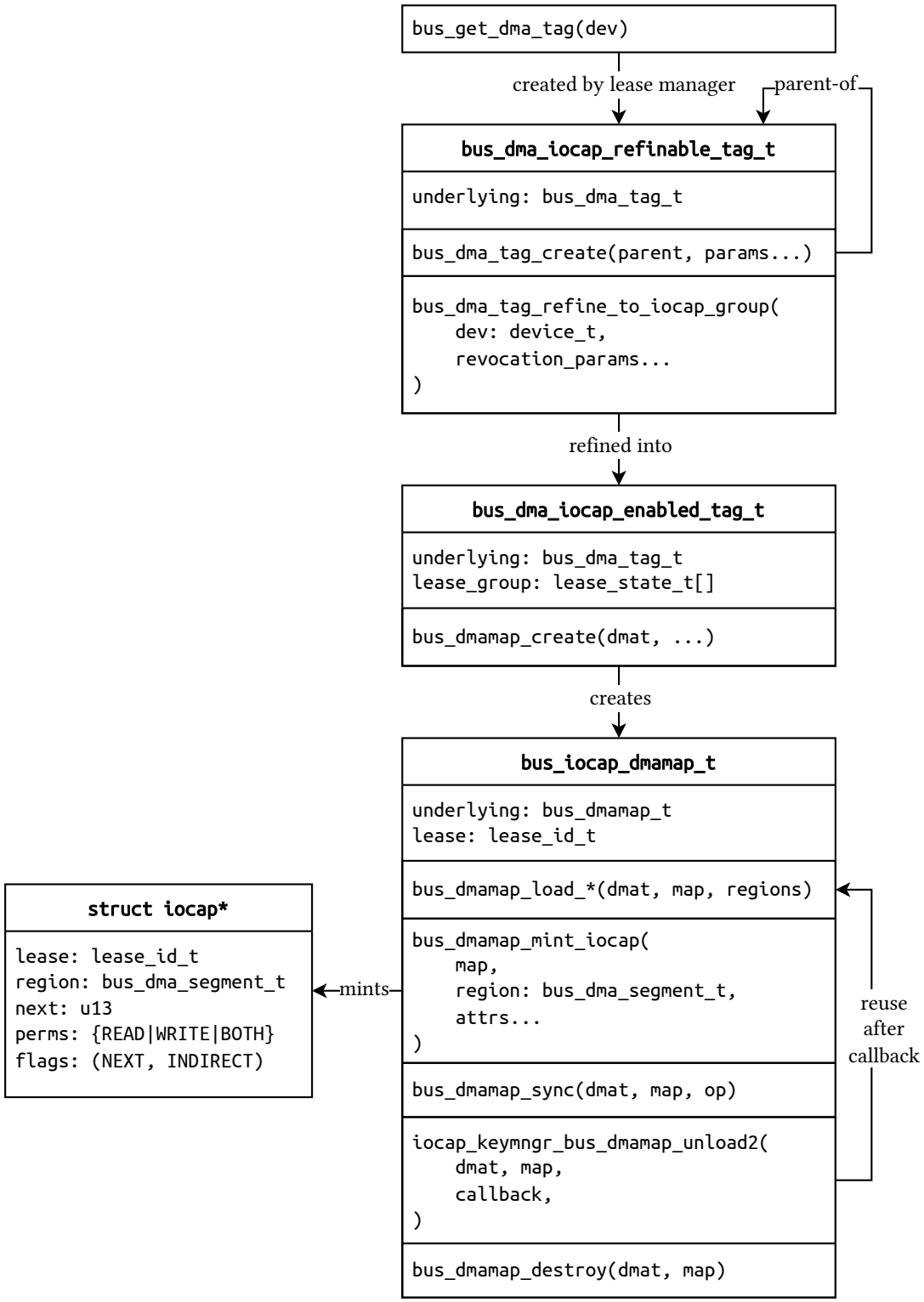


Figure 6.4: DMA mapping structures and their relationships in IOCap-enabled CheriBSD.

and open/close status are stored inside the lease group tag. This keeps the peripheral driver code simple, makes it easier for the different strategies implemented in the `iocap_keymngr` driver to be verified, and allows for great core-level scalability—once refined, lease group tags are entirely independent. In case individual tag objects are used from different threads, they are protected with a lock. Drivers must only operate on leases in 'ithread' routines where locking is safe [260]. A high-performance implementation may assume tags are never shared, or use lock-free algorithms to implement shared tags. I leave this as future engineering work.

6.2.2.1 Minting and revocation

Once CheriBSD loads a `bus_dmamap_t` with a mapping, it calls a callback with a scatter-gather list of `bus_dma_segment_t` regions that adhere to the constraints from the DMA tag. This allows the peripheral driver to communicate those bus addresses to the peripheral. In a typical IOMMU-based system these bus addresses would be IOMMU-virtual and directly usable by the peripheral—the callback would fill the queue with descriptors based on those addresses and alert the peripheral to new work. In an IOCap system an extra step is required: after the bus addresses are generated, the IOCap descriptors need to be minted. It would be impossible to automatically mint these descriptors in a protocol-agnostic driver. You need the information for the rest of the descriptor—in this case, the extra VirtIO flags and the `next` fields for each IOCap—which cannot be known before the DMA map is loaded and the size of the scatter-gather list is known. Minting IOCaps is therefore achieved through a separate function which takes a `bus_iocap_dmamap_t` (a special variant of the DMA map object attached to a specific lease ID), one of the bus-space regions from the scatter-gather list, and the VirtIO-specific information, and mints an IOCap using `libccap`. This mapping object includes an internal state machine to prevent out-of-order function calls from corrupting state or accidentally leaving leases open.

Temporal safety is handled using a new asynchronous unload function (see Table 6.2). `iocap_keymngr_bus_dmamap_unload2()` takes a callback and two arguments, which are tracked inside the relevant lease group tag for each mapping object. Once all mappings under a single lease ID have been unloaded, the `iocap_keymngr` driver will revoke the lease over MMIO[2], and call the callbacks for that lease ID to indicate step (3b) has been completed. As part of these callbacks, the mapping objects will be truly unloaded and any bounce buffers will be freed. At that point the peripheral driver can use the memory however it sees fit. My implementation does not implement a maximum lease lifetime, which I leave for future work—I do not believe this is a problem in practice because attacks on AES are not yet practical in even the worst-case peripheral lifetime (Section 4.3.4, p82).

6.2.2.2 `bus_dmamem`

My implementation does not restrict `bus_dmamem_{map,unmap}` for memory allocated with `bus_dmamem_alloc`. Generally, peripherals should only use these functions before exposing the

[2]In practice closing the lease requires writing to the status byte and then spinning on the lease status to wait for it to be fully 'invalid'. I explain further in Section 7.2.1, p155.

Function	Lifecycle
`bus_dmamem_alloc()`	(1) Host allocates memory
`bus_dmamap_load_*()` `bus_dmamap_mint_iocap()`	(2) Host exposes memory
`iocap_keymngr_bus_dmamap_unload2()`	(3a) Host requests unexposure
`<callback>`	(3b) Memory is unexposed
`bus_dmamem_free()`	(4) Host frees memory

Table 6.2: How CheriBSD maps to the practical lifecycle of host memory described in Table 2.2 when using IOCaps. IOCaps expose (3b) via the callback, allowing safe overexposure. `bus_dmamem` is used for illustration and is not required to use IOCaps.

memory or after the step (3b) callback. However, it is possible to use this memory correctly while it is still exposed: for example, taking a leaf from DAMN [21] and copying data out to a separate buffer to avoid peripheral-writes racing with host-reads; so I do not restrict its usage. The argument for restricting them is that CheriBSD documentation already recommends calling `dmamem_map` as soon as memory is allocated, so drivers may already expect to touch memory as soon as they request unexposure with (3a). On the flip side, any exposed memory that wasn't allocated via `dmamem` will already be kernel-virtual accessible, so it is still ultimately down to driver etiquette no matter what.

6.2.3 IOCap-VirtIO in CheriBSD

After adapting CheriBSD's DMA API to IOCaps, the final step is to adapt real peripheral drivers to the IOCap DMA API. CheriBSD includes a VirtIO driver framework, which supports PCI and MMIO transports and split virtqueues, with drivers for storage, network, and graphics peripherals alongside many others. IOCap integration was more involved than for CheriFreeRTOS, because of the extra complexities of the driver infrastructure and the integration of temporal safety, so I could not port all drivers at once. I decided to port the `virtio-blk` storage driver because it was already reliable on the group's FPGA infrastructure.

The VirtIO driver hierarchy has a couple of layers: generic `virtio_pci` or `virtio_mmio` peripherals appear as bus devices, which are probed to determine a specific underlying driver (e.g. `virtio-blk`) which is attached as a child, which both use a `virtqueue` implementation that is shared across all drivers. Each layer must be adapted to IOCaps. As before, I only considered the MMIO transport as this is what I had available to test. The transport handles generic device discovery, feature negotiation, and configuration. It was changed to mint IOCaps for each queue using the upstream `iocap_keymngr`, and to communicate them to the peripheral. These IOCaps are minted under a single-lease group that is closed once the peripheral is detached. This is secure as long as queues never change and AES is not broken within the system uptime. A more complete future implementation should refresh peripheral leases with new keys after their maximum lifetime using a two-lease epoch system.

6.2.3.1 `virtqueue-iocap`

I created IOCap-specific copies of the `virtqueue` module and the `virtio-blk` driver, rather than modifying them in place, to allow API flexibility without breaking other drivers. If IOCap support is advertised by a `virtio-blk` device, the original driver deprioritizes itself[3] and the IOCap version takes its place. The IOCap virtqueue is quite similar to the original. The key changes were 1) new checks to ensure descriptor `next` fields stayed within the 2^{13} descriptor limit imposed by the IOCap-VirtIO format (Section 5.3.6, p117), 2) checks that the `libccap` functions used to update and read VirtIO metadata did not fail, and 3) modifications to the enqueueing APIs to take a DMA mapping object as an argument in order to mint IOCaps. Number 2) speaks to a minor complication. The VirtIO drivers use the descriptor ring to store data, even when the descriptors are not active. This required writing extra `libccap` functions to extract and fill in the `next` fields and flags without fully minting the IOCaps.

6.2.3.2 `virtio-blk-iocap`

The `virtio-blk` driver had the most changes, but they were still very manageable. The basics consist of creating both a refinable tag and an IOCap group tag, checking that the DMA mapping objects pre-allocated for every element in the queue were capable of minting, and of course opting in to temporal safety with the new callback-based `unload2` function. VirtIO queues are circular buffers and have the same in-order consume behaviour as NVMe (see Section 4.6.2.1, p92), even in the presence of out-of-order completion, so the IOCap group tag uses the $N_L = 4$ revocation strategy and sets L_{ops} based on equation (4.2).

Temporal safety was easy to enable, because the standard event loop already depends on callbacks. The `virtqueue` calls a callback when an interrupt from the peripheral indicates an op has completed. In the original driver this first callback would unload the mapping and release the formerly-exposed memory to the client, but it is trivial to put the latter half of that code into a second callback—called when the mapping is actually unloaded. That said, there were a few operations that depended on completely synchronous revocation: the dump operation, which pause the whole drive and dumps a portion of the disk to memory, and the 'ident' operation, which retrieves an identifier string. In both cases the control flow was not set up to allow callbacks, and both cases are not designed to overlap with normal work. The dump function drains the queue before actually dumping, and the ident function is called during initialization, before the disk is actually available to users. I decided to allow these functions to call `unload2` with a null callback, which triggers an assertion inside the lease manager driver to ensure revocation is actually synchronous.

The ident function also triggered an interesting bug that demonstrates the malicious potential of coarse-grained spatial safety. The identifier string is meant to be a 21-character null-terminated string stored in one field of a `struct disk`, and the ident function simply exposed that field directly to the peripheral. It so happens that that field was not page-aligned, and a bounce buffer was created—one 4KiB long, rounded up from 21-bytes—that discarded

[3] This is my only modification to the original driver.

the original length information. The mapping API returned the bounce buffer's 4KiB segment, the driver exposed a 4KiB long IOCap, the peripheral[4] happened to zero the exposed bytes outside the 21-character field, and when the identifier op completed the *whole 4KiB was copied back*—zeroing many other fields in the `struct disk` including function pointers! Were it not for CHERI, a hostile peripheral could leverage this attack to point these function pointers into the buffer itself for remote code execution. CHERI-less systems that, say, use the IOMMU to directly expose memory regions, risk falling afoul of this kind of vulnerability [145]. I fixed this bug by ensuring the bounce buffer did not discard the original length, and am working on upstreaming the fix to mainline Free/CheriBSD.

6.2.4 Functionality

CheriBSD successfully boots and runs on QEMU, and Fig. 6.5 shows a small log excerpt. The `iocap_keymngr` driver exposes the performance counters to the `sysctl` CheriBSD utility, so they can be queried in the shell, and I measure them twice. Just after boot, only good transactions have passed through. When I disable MAC generation for exported IOCaps and perform filesystem actions (even searching a folder via tab-completion), QEMU starts trying to dereference invalid IOCaps and prints a debug message. A shell script then prints the performance counters again, demonstrating that QEMU tracks accesses through new unauthenticated IOCaps as bad. QEMU still allows these transactions without locking up the system, but a practical implementation should block the peripheral and notify the driver. The queue IOCaps are still valid, so the number of good reads and writes still increases.

6.3 Observations on implementing IOCaps

After partially implementing IOCaps in two operating systems, I have compiled a list of observations that can be used to gauge how difficult it is to adapt other systems.

First, for IOCap minting and decoding: a C library is essential. I initially linked a static library compiled from Rust against FreeRTOS, but CheriBSD does not seem to support this, and current efforts to extend CHERI to Rust do not yet support general-purpose RISC-V. Where possible, this library should use cryptographic acceleration instructions like x86 AES-NI [261, §12.13], Arm AES* [262, FEAT_AES, pA2-88], or RISC-V scalar crypto [255, §31.3.5]—unfortunately these were not available for the CHERI-RISC-V QEMU platform I was using, nor the FPGA platform I use in Chapter 7, so I called out to the kernel's pure-C software implementation [263]. This library should allow reading and writing IOCap metadata *without* fully minting them, as CheriBSD's VirtIO drivers expected. It also requires a good source of entropy—I noted this in Section 6.1.3, and it applies equally to CheriBSD.

Complex operating systems may have a dedicated subsystem for exposing memory over DMA, like CheriBSD, and smaller ones may leave this to individual protocols, like CheriFreeRTOS. If the existing code is well-structured, any code for manipulating descriptors should all

[4] In this case, the emulated peripheral running on the hardware SoC (Section 7.5, p164) not the QEMU peripheral.

```
Logging in as root...

# sysctl dev.iocap_keymngr
dev.iocap_keymngr.0.disable_encrypt: 0
dev.iocap_keymngr.0.bad_read: 0
dev.iocap_keymngr.0.good_read: 6953
dev.iocap_keymngr.0.bad_write: 0
dev.iocap_keymngr.0.good_write: 6547

# sysctl dev.iocap_keymngr.0.disable_encrypt=1
dev.iocap_keymngr.0.disable_encrypt: 0 -> 1

QEMU: check_signature (data 817fb01f) (sig 817fb01f) for key_id 1
    (689c9560) failed: CCapResult_Decode_InvalidSignature

# ./bench/bench.sh
+ : 'Confirm disk exists via dev'
+ test -c /dev/vtbd-iocap1
+ : 'Confirm iocap is functional'
+ sysctl dev.iocap_keymngr
dev.iocap_keymngr.0.disable_encrypt: 1
dev.iocap_keymngr.0.bad_read: 25
dev.iocap_keymngr.0.good_read: 7040
dev.iocap_keymngr.0.bad_write: 10
dev.iocap_keymngr.0.good_write: 6603
```

Figure 6.5: Sample console output from CheriBSD, plus a QEMU log message, after booting on QEMU. Redundant lines have been removed for brevity, and the QEMU: prefix was added to the log message for clarity. `CCapResult_Decode_InvalidSignature` indicates an invalid MAC, not a signature.

be in one place and simple to replace. This was the case for CheriBSD and CheriFreeRTOS. A generic DMA API for IOCaps, as needed for CheriBSD, likely requires two core concepts: a 'leaser' object representing an allocated group of lease IDs and keys with a revocation strategy, and a 'minter' object representing a single set of exposures that can be used to actually mint IOCap descriptors. These are the IOCap-enabled tag and the IOCap DMA mapping objects from CheriBSD, respectively (Fig. 6.4). Separate leaser objects for each group of IOCaps allows many-queue or many-core systems to efficiently manage individual groups without cross-core synchronization. Minter objects allow peripheral drivers to express that a set of exposed regions belong together, e.g. to the same op, and should not be split across leases. Existing APIs in CheriBSD and Linux both already have DMA mapping APIs that output numeric bus addresses, and these APIs cannot be easily changed to return IOCaps. 256-bit IOCaps will not fit into any standard numeric type, and the minter cannot automatically mint descriptors without knowing the extra metadata that must be packed in.

Peripheral drivers that use IOCaps to expose memory may need to expose memory from different sources under the same minter object. For example, each op exposed by the `virtio-blk` driver consists of a header, allocated by the driver, the actual data pages allocated by

CheriBSD's block I/O subsystem, and a writable status byte also allocated by the driver. Both the original and IOCap versions of these drivers end up creating extra descriptors for the driver-allocated regions without exposing them through a minter[5], because CheriBSD's DMA mapping API doesn't support multiple sources—this should be fixed. There should still be separate IOCaps for these regions, but they should be exposed by the same mapping object, and unexposed together with a single asynchronous unload operation.

It is relatively easy to adapt peripheral drivers to IOCap temporal security but it requires care. When a driver that expects ops to be completed asynchronously, adding a second level of asynchrony before the op completes is simple. However, CheriFreeRTOS expects all ops to complete synchronously, and CheriBSD expects some ops to complete synchronously. As long as synchronous ops are not parallelized with asynchronous ones, or if an $L_{\text{ops}} = 1$ revocation scheme is used, IOCaps can guarantee synchronous revocation.

6.3.1 How CHERI can help

In Chapter 3 I noted that CHERI should be used to mitigate attacks on operating systems and IOCap software running on the host. Here, I describe two instances where I believe deeper CHERI integration into the DMA API could help.

As I stated earlier, storing keys in per-group leaser objects (the DMA tags, in CheriBSD) helps avoid synchronization overhead. However, it also means the keys are directly accessible to peripherals through the pointer to the DMA tag, which could use them to forge IOCaps. Ideally CHERI could compartmentalize the drivers: sealing DMA tag pointers and ensuring that only IOCap driver code can dereference them. This has been investigated in userspace [33] and forms the basis of CheriFreeRTOS [156] but has not yet been extended into the CheriBSD kernel.

The other problem is with the minter, which in CheriBSD takes in the region to mint as a `bus_dma_segment_t` by value. Broken drivers could easily forge new segment structures to expose arbitrary bus addresses instead. Ideally, the DMA mapping API would still hand out some type `bus_dma_segment_t`, and the minting API would consume them to create IOCaps, but they would be unforgeable and reducible (so if two regions are merged into a bounce buffer, and they need to be exposed with separate IOCaps, the driver can split them). CHERI capabilities would be a suitable implementation! They are unforgeable and reducible, but they do not have address space tags. Bus-space capabilities could be confused for kernel-virtual capabilities and vice versa. One potential solution could be for the mapping API to hand out bus-space CHERI capabilities with all permissions removed—preventing host reads, writes, everything—and only using them to prove the segment was not forged. There may be other areas of the kernel could use this concept, and it may be useful to disambiguate these bus-capabilities for different busses, so a full solution may use more of CHERI's metadata bits. I leave the full consideration of this system as future work.

[5]These exposures do break the rules by assuming the kernel-virtual and bus spaces are identical—in this case that is actually true, but this whole approach should not be needed.

6.3.2 Implementation impact

The CheriBSD changes were modest but subtle. Table 6.3 shows the lines of code for the `iocap_keymngr` driver itself, `libccap`, and the effective lines changed between the original and IOCap versions of the VirtIO drivers. The `iocap_keymngr` driver and `libccap` are both roughly equivalent to the `virtio-blk` driver in size, but the total lines of code changed to port `virtio-blk` to IOCaps is much less. In fact, the interface to the IOCap virtqueue is almost identical to the original, and it would be feasible to merge them into a single driver that simply changed toggled between IOCaps or plain descriptors at runtime. The `virtio-blk` driver itself might not be so easy to merge, because it has to create IOCap-enabled tags and store them in an extra field, but the actual difference in behaviour is small.

Would this be the same for Linux? I cannot say for certain without a full attempt, but I would raise two points of note from reading the Linux DMA API documentation [47]. First, the concept of DMA tags and mapping objects is completely absent. Address constraints are properties of the driver instance itself, rather than a separate object created by the driver. That means the concept of the leaser and minter objects, on top of new minting and asynchronous unmapping functions, must be introduced from scratch. Secondly, all of the DMA memory allocators and mapping functions create `dma_addr_t`s: objects "which may be cast to an unsigned integer the same width as the bus" and treated as "the DMA address base of the [exposed] region". These are effectively equivalent to CheriBSD's `bus_dma_segment_t` that are used as the source of truth for IOCap minting, and a hypothetical CHERI-Linux implementation could likely replace them with the bus-space capability I proposed in the previous subsection. All in all, it should be a similar amount of work to write the `iocap_keymngr` driver itself—that driver has to define the leaser and minter objects in all cases—but converting new drivers to use leaser and minter objects would likely be more work than for CheriBSD, where idiomatic drivers already use tags and mapping objects.

Components	LoCs Changed	Total LoC
`libccap`	-	2230
`iocap_keymngr` driver	-	2036
`virtio-mmio` transport driver	295	1306
IOCap-enabled virtqueue	137	1120
`virtio-blk` driver	130	1780
Misc. VirtIO driver code	109	5953

Table 6.3: Lines-of-Code (LoC) changes in adapting CheriBSD to IOCaps.

6.4 Summary

Hypothesis 3

IOCap protection is practical to adopt in system software.

This chapter asks if complex operating systems can be naturally adapted to IOCaps, what principles are needed to introduce IOCaps, and what impact they have on peripherals. Having discussed the principles, I demonstrate the viability of integrating IOCaps into VirtIO and complex operating systems.

Facet 3.1

The VirtIO I/O framework, and peripherals that integrate it, can be extended to replace DMA descriptors with IOCaps without losing functionality.

Yes, aside from the queue size limitation of the IOCap-VirtIO format. QEMU already supported multiple queue formats, and IOCaps are effectively just a new format from the perspective of the peripheral. Peripherals do need to decode them but do not need to cryptographically verify them.

Host software *is* required to create the MACs, which is a new friction point. Instead of constructing VirtIO descriptors field by field as convenient, which CheriBSD drivers took advantage of, IOCaps require a single point where each descriptor is minted and then frozen until it is consumed. In practice I believe this is unavoidable, as the peripheral driver needs ultimate control over what IOCaps are created, and it was not a particularly invasive change.

My testing did not encounter any spatial granularity issues with the IOCap-VirtIO format, but some descriptors needed to use both caveats for maximum precision—particularly for single-byte regions. This indicates that using all caveats may be a common case, and it may be impactful to allow using a single MAC to authenticate a fully-caveated IOCap as hypothesized in Section 5.6.1, p134.

Facet 3.2

Complex operating systems can be extended to control the trusted IOCap enforcer and untrusted IOCap-compatible peripherals via VirtIO, with minor API changes, without losing functionality.

Yes, if analogues to the minter and leaser objects already exist. In Free/CheriBSD very few changes were required to adapt `virtio-blk` to IOCaps once the `iocap_keymngr` driver was in place. Crucially, parts of the driver that required synchronous or asynchronous invalidation could get what they wanted without sacrificing safety. There are three exceptions to this answer: APIs without minter or leaser objects such as Linux may be more difficult to extend, and I have not fully investigated the complexity of enforcing maximum lease lifetimes or using the epoch-based revocation strategy (Section 4.5.1.1, p84).

CHAPTER 7

IOCaps in hardware

Everything is now in place. I have defined the system in the abstract, including temporal safety measures; I have defined a practical capability encoding; I have drivers for a complex operating system ready to run—but the practical performance, and therefore the effective security, of IOCaps live and die on the hardware implementation. In this chapter I demonstrate how IOCaps can be transported efficiently over interconnects, and describe my implementation of the interposer described in Chapter 3. I validate its functionality with model-based testing, microbenchmark it to find performance and area limits, and demonstrate the full stack running on the CHERI group's FPGA SoC infrastructure [36].

7.1 IOCaps on the interconnect

First, the interconnect needs to carry IOCaps. I chose to adapt AXI as it is used by the CHERI group's existing SoC frameworks. I considered PCIe, but it would require creating an IOCap-capable root complex from scratch, which would have been a significant engineering undertaking for little benefit. I consider the potential impact of IOCaps on PCIe in Section 7.1.1. A full explanation of AXI can be found in the background chapter (Section 2.3.2, p35) but I reiterate the important parts here.

AXI4's basic standard [65] specifies a one-way bus connection between an upstream 'manager' and a downstream 'subordinate'. Each connection is made of five unidirectional channels: three for writes (`AW` write address, `W` write data, `B` write completion), and two for reads (`AR` read address, `R` read data & completion); which can each send exactly one flit (flow control unit) per cycle. Under standard AXI there is a single address flit for each transaction, read or write, and then a burst of one or more data flits—allowing large data transfers, even over relatively thin data channels—followed by a completion. Address flits carry transaction IDs which control ordering. Reads and writes have no ordering relative to each other, but two reads with the same ID must be completed in the order they are received, as must two writes [65, §A6.1, §A6.3]—otherwise there are no ordering guarantees. The address width, transaction ID width, and data width of each channel are configurable at design time, and each channel has a

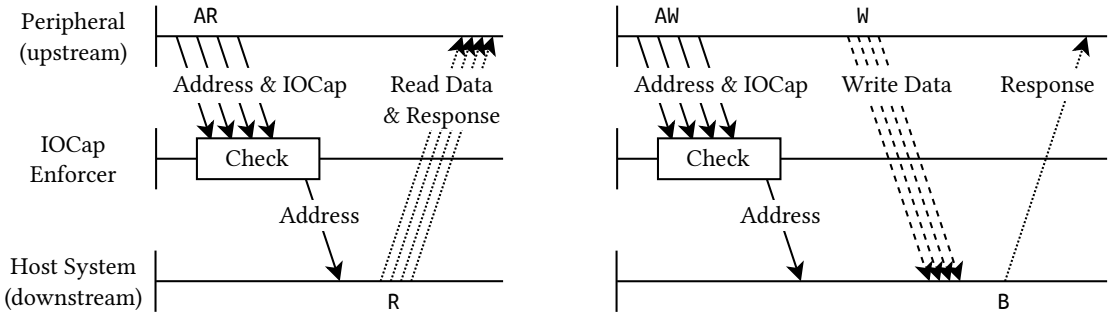
USER field of arbitrary width. These widths may change across the interconnect. Individual subordinates may only expose address widths large enough to access *their* data, cutting off higher bits used by the managers to distinguish between subordinates. MMIO subordinates may use thin data channels, and memory subsystems may use wide ones. An AXI4 switch may have to multiplex many upstream managers down into a single downstream subordinate, and may use extra ID bits on the downstream side to indicate which manager made each request.

I extended AXI4 to carry 256-bits of IOCap with each address flit based on three observations. First, wide signals are power-hungry and difficult to place and route. Simply expanding the AW and AR USER signals by 256-bits would be out of the question. Second, AXI4 keeps reads and writes independent wherever possible—atomics being the sole exception. Reusing the W channel, i.e. the widest manager-to-subordinate signal, to carry IOCaps would break this rule and effectively turn all accesses into atomics. Thirdly, AXI4 is burst-based. Each transaction contains a single address flit and multiple data flits, so address channels will never be the throughput bottleneck as long as the data channels carry more flits per transaction.

My approach for IOCap-AXI is to time-multiplex the address channels (Fig. 7.1). Each read and write transaction requires four flits on their respective address channels: the normal address flit; then three flits holding the IOCap MAC and text. The normal address flit is still necessary, so that peripherals can choose to access sub-regions of an IOCap without reducing it. The IOCap flits reuse the space intended for the 64-bit base address and the extra metadata included in standard address flits, which in my case totals 86-bits. Thus, three IOCap flits are sufficient to carry a full 256-bit IOCap. The first IOCap flit contains the lease ID field, which allows the enforcer module to look up the lease key as early as possible. I extended the USER fields with two bits each, simply to carry a sequence number, but ordering is always fixed and different transactions on the same channel cannot be interleaved—the USER fields are simply a sanity check. This approach keeps routing and congestion control simple, by sending the address flit first (allowing routing decisions to be made early) and using a constant number of flits (so credit-based systems simply need to ensure they have four credits before accepting a transaction, rather than just one). It only impedes throughput if each transaction uses fewer than four data flits on average, which I believe to be rare for bulk DMA transfers. I believe this will extend to more complex AXI constructions (e.g. AXI Exclusive Accesses, AXI Coherency Extensions, and others noted in Section 2.3.2, p35) but have not checked the finer details—I leave this for future work.

7.1.1 Serial protocols

IOCap-AXI's approach depends on parallelism between the address and data flits—spare cycles on unused channels which do not prevent data from flowing. Serial protocols, where data is sent over single wires instead of wide buses, may not leave spare cycles. At least, if throughput matters, the system designer will already have done everything they can to saturate the network and fill those cycles in. PCIe and CXL are both serial protocols, which means attaching IOCaps sacrifices bandwidth that could have been used for data. With long data transfer packets, e.g.



(a) IOCap-AXI read transaction.

(b) IOCap-AXI write transaction.

Figure 7.1: IOCap-AXI flow diagrams, with channels marked. Address and IOCap flits are sent on the same address channel, separate from data and response channels.

4096-bytes, a 32-byte IOCap is a relatively small overhead. With small packets, especially 64-byte cache lines, it is not. This is why a smaller IOCap format may be desired in the future.

7.2 Building a hardware IOCap interposer

The interposer accepts IOCap-AXI, checks the IOCaps attached to each request against the lease manager, and forwards valid requests to the host as plain AXI. My implementation has several internal components, shown in Fig. 7.2. Separate frontends for the `AR` and `AW` channels extract the IOCaps from the IOCap-AXI flits, request their lease keys from the lease manager, then pass them to separate pools of checkers[1]. Once checked, valid transactions are forwarded downstream and tracked in separate read and write scoreboards. Invalid transactions are blocked by sending them back upstream with an error response, taking care to respect AXI transaction ordering rules, mixed with the downstream responses from valid transactions. Ideally these components should be able to handle one lease management MMIO request per cycle, and up to two incoming DMA operations (one read, one write) every four cycles. The former requirement can be relaxed if IOPS are low, thus fewer lease operations are required per second, but I pick one per cycle to ensure it can handle burst traffic. The latter requirement can be relaxed if DMA traffic is expected to use more than four data flits, but I pick one every four cycles because it is the highest possible throughput (IOCap-AXI requires four address flits, thus four cycles, per transaction).

7.2.1 Lease manager

The lease manager is responsible for tracking the state of each lease and handling MMIO interactions with the host that manipulate those states as per Fig. 7.3. There are 256 available lease IDs, matching the IOCap-VirtIO specification, each with host read/writable status and write-only data MMIO regions (matching the software interface from Section 6.1, p135). Lease statuses are held in an array of tiny, per-lease state machines, taking advantage of the fact that

[1]AXI transactions with basic and Fixed burst modes (see Section 2.3.2.1, p35) are supported, but Wrap mode is not. Fixing this is a trivial engineering effort.

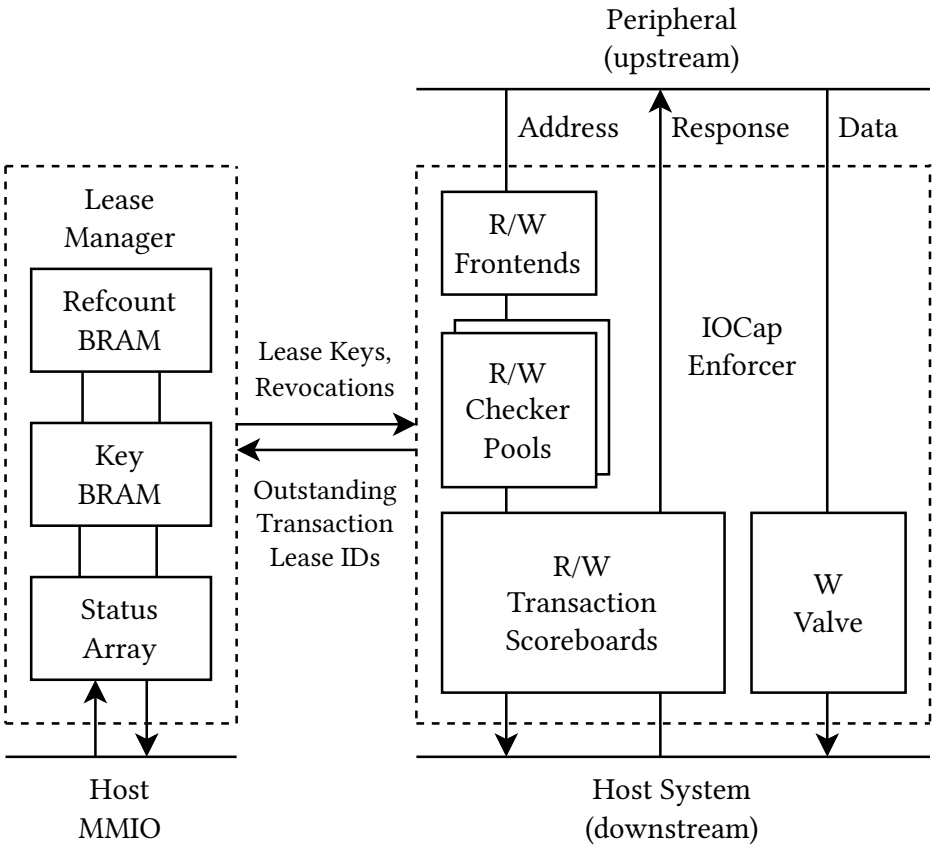


Figure 7.2: The IOCap enforcer and lease manager modules. The full set of lease manager internal connections have been simplified.

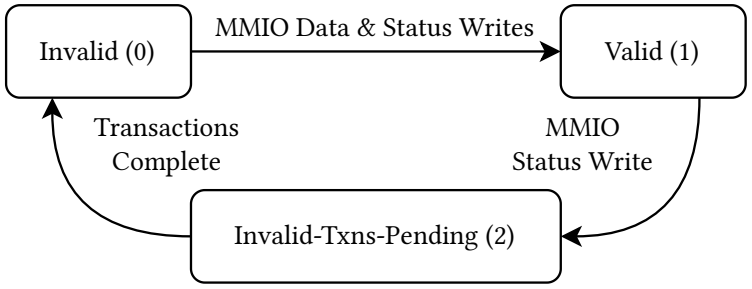


Figure 7.3: The three states of each lease and their host-facing integer encodings. Hardware may use a different internal encoding such as one-hot.

only one lease can go through each state transition in a single cycle. Looking up lease validity is a common operation, and this construction made it easy to support many concurrent readers with zero cycles of latency without stalling. Lease keys are stored in a BRAM, effectively the FPGA equivalent to a two-port SRAM. They are written by MMIO, and read by the enforcer. A second BRAM holds per-lease reference counts, to track which leases are authorizing transactions-in-progress, but is technically not necessary for an optimized implementation.

Opening a lease requires the host to write the key to the data region before confirming by writing 'valid' into the status. The lease manager does not track data writes, so the host is responsible for not opening a lease for an incomplete key. Closing a lease only requires writing 'invalid' into the status. This sends a revocation message to all internal components, immediately preventing new transactions from using the lease. Though new transactions will be blocked, there may still be outstanding valid transactions under that lease downstream—during this time, the lease is 'invalid-txns-pending'. Hosts should not touch exposed memory or try to reuse the lease ID in this state. Well-behaved peripherals should stop using IOCaps long before their leases are closed, so any lease in this state indicates a high probability of foul play. Said foul play effectively manifests as a denial-of-service attack isolated to the driver for the peripheral in question, which the peripheral could otherwise achieve by directly denying requests, so I do not consider this a safety concern. After these transactions complete, the lease reference count hits zero, it becomes 'invalid', the key data is zeroed out, and the ID can be reused.

Opening the lease only requires three 64-bit MMIO writes, and the lease is guaranteed to be open two cycles after the final write-response is sent (in practice, by the time the host receives it). Closing the lease requires a single MMIO status write and spinning on reading the lease status until it is fully 'invalid', which should be immediate in most cases. These numbers are verified and discussed in Section 7.4.3, p162.

7.2.2 Checker pool

There are separate pools of checkers for reads and writes. Each checker performs two parallel checks. The first check uses the compact hardware decoder, described in Section 5.4.1, p121, to check the caveats are valid and ensure the attempted access is in bounds. This decoder takes 3–8 cycles depending on how many caveats are enabled. The second check is MAC verification, which requires at least one full invocation of AES-128 and up to three for an IOCap which uses all caveats.

Each invocation of AES is split into ten rounds [186]. Each checker includes an AES module that needs to compute a chain of up to three invocations in a row, all over a single block, efficiently. This use case is not covered by prior art, in the Bluespec language used by the CHERI group [264] or otherwise [167]. I wrote an AES library in Bluespec which exposes the individual stages of AES as combinatorial functions, and tested it against NIST's test vectors [186]. I used it to implement two possible sub-modules for the checker: one of which computed a single AES round per cycle, and another which computed two; trading off a lower

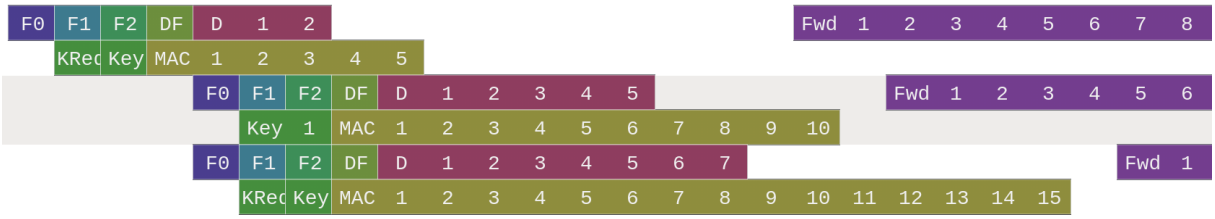


Figure 7.4: A pipeline diagram demonstrating how a checker handles transactions with different numbers of caveats. After all four flits are received (F0–F2, DF) the IOCap is decoded (D) taking 3, 6, or 8 cycles. When the first IOCap flit is received (F1) the checker immediately tries to request the key from the lease manager (Key), which takes two cycles. If a read and a write transaction arrive together, one must wait (KReq) til the next cycle to request the key—but in this case they use the same lease, so the first request serves both of them. Upon receiving the key the MAC stage immediately begins, taking 6, 11, or 16 cycles, and always outlasts decode. After passing through enforcer machinery, potentially waiting for others to complete, the transactions are forwarded downstream (Fwd).

cycle latency for worse clock frequency and area overhead. The two-rounds-per-cycle checker takes 6–16 cycles to compute 1–3 MACs, beating OpenTitan's implementation [167], but is still the latency bottleneck (Fig. 7.4).

Checkers are not fully pipelined, and can only handle one transaction at a time. The pools of checkers can be sized to achieve a target throughput. If each write transaction uses 8 data flits on average, and the worst-case end-to-end latency was 22 cycles, $\lceil \frac{22}{8} \rceil = 3$ checkers are required in the write pool to achieve maximum throughput—much more area-efficient than a 22-stage pipeline. Transactions pass through the pool in order, invalid transactions are sent back upstream, and valid transactions are forwarded and tracked by the scoreboards.

7.2.3 Transaction scoreboard

Outstanding transaction IDs and their lease IDs are tracked in the scoreboards, which serves two purposes: it allows the lease manager to track lease IDs with outstanding transactions, and it tracks outstanding transaction IDs to ensure correct response ordering. In order to remain compliant with AXI ordering rules, transaction completions (on the `B` and `R` channels) need to be passed back in the correct order. The upstream requestor must be able to tell which transaction each completion is associated with. This requires tracking the outstanding transactions at the enforcer: if a valid transaction is sent downstream, and then an invalid transaction with the same ID arrives at the enforcer, the enforcer can't send the error-completion upstream until the prior valid completion is forwarded upstream. I implement this using a BRAM addressed by transaction ID. If the scoreboard shows that a transaction ID has an outstanding valid downstream transaction, error-completions for that transaction ID (and any further transactions on that channel) are blocked. This imposes a limit of one outstanding transaction per ID[2]. In any case, because the scoreboard is fixed-size, the enforcer fundamentally imposes a limit on the number of parallel outstanding transactions—and therefore, throughput. This is unfortunate, but it ultimately also applies to the downstream host memory. Designers

[2]A future implementation could use an array of state machines instead, to impose a maximum outstanding transaction limit irrespective of IDs. This would likely also remove the need for the reference counting BRAM.

must tune the scoreboard sizes for each enforcer to match the number of supported parallel transactions downstream.

The flits themselves also prove to be a complication, when blocking invalid transactions. Read transactions are easy to block—just don't send invalid `AR` flits downstream!—but write transactions are more difficult, because they consist of `AW` flits and a variable number of `W` flits. The `AW` and `W` channels are not synchronized: `W` flits could arrive before, on the same cycle, or after the `AW` flits. I address this problem with a simple credit-based valve on the `W` channel. The valve stores a credit count, and a flag which indicates if they are passthrough-credits or drop-credits. `W` flits that arrive are blocked if there are no credits (i.e. the validity of the associated transaction is not yet known), passed through if there are passthrough-credits (i.e. the associated transaction is valid) or dropped if drop-credits (the transaction is invalid). Incrementing credits in the same mode is easy—if the valve is in passthrough mode, a new valid transaction will simply increment the credit count—but switching modes requires blocking. If an invalid transaction is received from upstream while there are passthrough-credits pending, it has to wait to drain the passthrough-credits before it can switch the valve to drop-mode. My approach assumes that invalid transactions, and therefore transitions between passthrough and drop modes, are rare.

7.3 Verification

In order to ensure the hardware operates correctly, especially under complex scenarios with overlapping DMA and lease management operations, I created a small model-checking framework inspired by IEEE Universal Testing Methodology [265]. Each design-under-test (DUT) is compiled through Bluespec into Verilog, and through Verilator [266] into C++, where it is matched with a set of stimulus generators and a reference model (unhelpfully, also known as a scoreboard). This infrastructure supports testing individual IOCap decoders, isolated enforcer modules, and combined lease manager-enforcer pairs. The latter is the most involved, simulating AXI transactions for the lease manager concurrently with IOCap-AXI DMA for the enforcer. Each IOCap used is randomly generated using my Rust generator library (Section 5.4, p120), allowing each module to be stress tested using typical and edge cases. Each stimulus generator tests a different aspect of the DUT, ranging from individual IOCap decoding tests to measuring throughput and revocation performance under heavy load—a selection of these tests are shown in Table 7.1. This testing framework was used to test many different versions of these hardware modules before finally arriving at those described in this chapter.

Random DMA Streams				
Kind	Caveats	Flits	Permissions	Leases
Valid	0	<20	R	Single
Valid	0	<20	W	Single
Valid	0	<20	RW	Single
Out-of-Bounds	0	<20	R	Single
Out-of-Bounds	0	<20	W	Single
Wrong Permission	0	<20	RW	Single
Invalid MAC	0	<20	RW	Single
Invalid Lease	0	<20	RW	Single
Valid	0	1	RW	Single
Valid	0	4	RW	Single
Valid	0	8	RW	Single
Valid	0	16	RW	Single
Valid	0	32	RW	Single
Valid	{0,1,2}	4	{R,W,RW}	Single
Valid	0	4	{R,W,RW}	Single
Valid	1	4	{R,W,RW}	Single
Valid	2	4	{R,W,RW}	Single
Valid	0	4	RW	Choice of 256
Valid	1	4	RW	Choice of 256
Valid	2	4	RW	Choice of 256
Valid	2	4	R	Choice of 256
Valid	2	4	W	Choice of 256
IOCap Encoding Edge Cases				
`Cav1_NonZeroWhenUnused`				
`Cav1_TooBig_IndexNonZero_Invalid`				
`Cav1_TooBig_IndexZero_Valid`				
`Cav2_NonZeroWhenUnused_Invalid`				
`Cav2_YSmallerThanX_Invalid`				
`Cav1_TooBig_IndexZero_Cav2_InBounds_Valid`				
`Cav1_TooBig_IndexZero_Cav2_TooBig_IndexZero_Valid`				
`Cav1_TooBig_IndexZero_Cav2_TooBig_IndexNonZero_Invalid`				
`Cav2_ShrinksToZero_Valid`				
Combined Enforcer & Lease Manager Benchmarks				
Open Lease During DMA				
Close Lease During DMA				
Rolling Lease Open/Close				

Table 7.1: The test permutations used to verify enforcer and lease manager behaviour.

7.4 Microbenchmarks

Performance is key for IOCaps. Here, I use the model-checking framework to measure key performance metrics of the hardware units in isolation: the latency and throughput of IOCap checking and access control, and the latency and throughput of opening and closing leases. I also evaluate the maximum frequency and area overhead results[3] on FPGA, specifically a Stratix 10 SX[4], using Quartus Prime Pro version 23.2 [268]. These performance and area metrics cannot be directly compared to a hard-logic IOMMU, because they are for a soft-logic FPGA platform—and though there exist soft-logic IOMMUs [269] I believe they are not mature enough for a representative comparison either. Other obstacles to a direct comparison include DRAM latency (which is much lower relative to soft-logic clock speeds than hard-logic) and a lack of publicly available representative DMA workloads. Instead, I use the simulation results as microbenchmarks and compare to a completely insecure system, with the goal of introducing IOCaps into the CHERI group's Stratix 10-based SoC [36] (which I refer to as CHERI-BGAS).

7.4.1 Area and frequency

As noted in Section 7.2.2, the individual checkers can be configured to evaluate one or two AES rounds per cycle, trading off frequency and area for raw cycle latency. Table 7.2 shows the synthesis results for an individual checker of each configuration, and the end-to-end latency[5] of a single valid transaction passing through an enforcer using that checker. The 'No-op' config uses a single-cycle FIFO as a checker that assumes transactions are always valid, though the surrounding enforcer still waits for all four IOCap-AXI flits to arrive. IOCaps with more caveats enabled require more MACs—initially only one, then two if one caveat is enabled, and then a third if the second caveat is enabled—so the config has a large impact on cycle latency. I use the 2x AES per cycle configuration for the rest of my experiments, because the area overhead over 1x is small, and the 212MHz $F_{\max}$ is much faster than the 47MHz CHERI-BGAS soft cores.

[3]Synthesis metrics are taken from a 32-seed sweep for each configuration, selecting the highest $F_{\max}$, to avoid random placement error.

[4]The specific part number is `1SX280HU2F50E1VG`, indicating 2,800k LUTs and the fastest speed grade of the Stratix 10 family [267, §1.1.1].

[5]Measured from the arrival of the first IOCap-AXI address flit at the enforcer's input FIFO to the departure of the AXI address flit from the output FIFO. If a read and write transaction arrive on the same cycle, one may be delayed by an extra cycle before using the lease manager's single key-read port.

Config	Single Checker		Enforcer Latency		
	$F_{\max}$	LUTs (%)	1 MAC	2 MAC	3 MAC
No-op	766MHz	110 (0.0%)	6	6	6
1x AES	265MHz	2,990 (0.3%)	17	27	37
2x AES	212MHz	3,989 (0.4%)	12	17	22

Table 7.2: Synthesis results for isolated checker units and cycle latencies for enforcers using those checkers under different configurations.

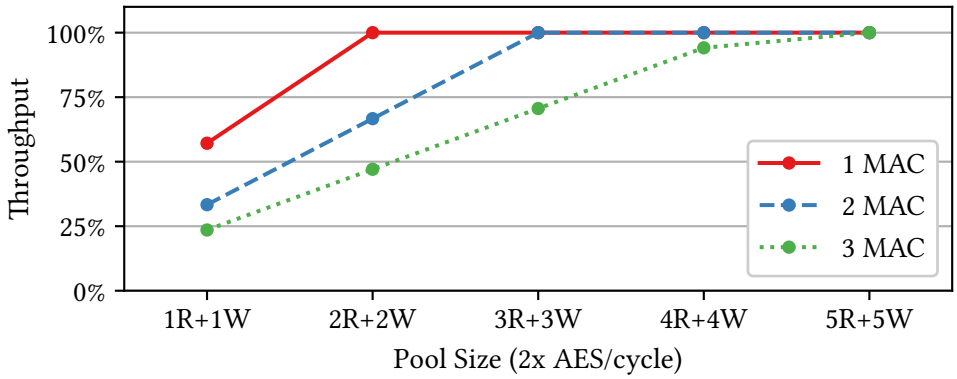


Figure 7.5: Enforcer throughput against pool size for 4-data-flit DMA read and write transactions.

Pool Size	F_{max}	LUTs (%)
Baseline	187MHz	4,410 (0.5%)
1R+1W	189MHz	13,231 (1.4%)
2R+2W	186MHz	20,459 (2.2%)
3R+3W	183MHz	28,108 (3.0%)
4R+4W	179MHz	35,723 (3.8%)
5R+5W	174MHz	43,032 (4.6%)

Table 7.3: Enforcer synthesis results against pool size. LUTs are expressed as a raw number and as a percentage of the whole FPGA.

7.4.2 DMA throughput and latency

Fig. 7.5 shows the average throughput for the enforcer checking DMA traffic under increasing read and write pool sizes. Table 7.3 shows the frequency and area results for these enforcers, including a baseline which uses one no-op checker each for read and write. Each of these simulations pre-configures the lease manager with 256 valid leases and fully saturates the read and write DMA channels of the enforcer with ten thousand valid IOCap-AXI transactions each. Each transaction uses four data flits and a randomly selected lease—ensuring the enforcer cannot take advantage of lease locality, or consecutive transactions using the same lease.

The enforcer can reach 100% throughput for all IOCaps, ensuring that new read and write transactions can be processed every four cycles (the number of address flits required by IOCap-AXI). This is equivalent to an unsecured system, and therefore superior to the IOMMU, as long as each DMA uses at least four data flits. If transactions have more data flits on average, e.g. 64-byte cache line transfers may use eight 64-bit data flits, then only $\frac{4}{8} = 50\%$ throughput would be needed.

7.4.3 Lease operation throughput and latency

The lease manager responds to all MMIO reads and writes with a latency of 2 cycles, pipelined for 100% throughput, independent of DMA traffic. As stated in Section 7.2.1, opening a lease requires three MMIO writes (two data, one status) and closing a lease requires one MMIO status write and some number of reads to confirm status. The simulation showed that opening leases

with three pipelined writes always has 4 cycles of total latency, and that closing leases with a single write always has a 2 cycle latency—after the final write returns, leases immediately become 'valid' or 'invalid-txns-pending' as requested. These latencies were measured at the lease manager boundary, so in practice a host CPU performing these operations will have an extra system dependent round-trip latency in communicating with the I/O hub[6].

The simulation also confirmed my assertion that although closing a lease immediately places it in the 'invalid-txns-pending' state, it can take longer to move into 'invalid' if it has to wait for outstanding transactions. Table 7.4 shows the latencies taken to move into the 'invalid' state for a lease with 16x outstanding read and 16x outstanding write transactions (the maximum our enforcer supports) of varying lengths—which vary based on the state of those transactions when the MMIO write arrives[7]. Even if this situation occurs in practice, which should be incredibly rare, the impact would be isolated to that specific lease.

Finally, I ran a rolling open-and-close experiment to measure the maximum throughput of those operations in practice. Table 7.5 shows the throughput of continuously opening, continuously closing, and a 1:1 mix (for a system operating under a steady state)—all without DMA traffic[8]. Open-only and close-only workloads match my 3-write and 1-write expectations. Combined, the open and close operations bottleneck on the shared MMIO-write data path—MMIO-reads are only used for revocations—and one open and one close can be completed every four cycles.

	Baseline	4-flit	8-flit	12-flit	16-flit	20-flit
Min.	5	3	70	114	146	190
Median	6	13	113	157	203	260
Max.	8	16	142	195	255	315

Table 7.4: Latency in interposer cycles between receiving an MMIO lease close request, and the lease entering 'invalid', when saturated with 32x DMAs of varying lengths using that lease. Baseline is saturated with 4-flit DMAs using a different lease.

Workload	Cycles per Open	Cycles per Close
Open only	3	-
Close only	-	1
Combined	4	4

Table 7.5: MMIO latency per open and close under different workloads in interposer cycles. Opening the lease includes uploading the key.

[6]For example, AMD EPYC 7002 series processors place the memory and I/O hub on a separate chiplet to the main core [270]. The L3 is distributed across the core chiplets, and the round-trip latency to this hub would be somewhere between a same-chiplet L3 access and a separate-chiplet L3 access.

[7]In rare cases same-lease DMA can speed up revocation. Revocation enqueues a reference count 'Check' operation for that lease in a pipeline, but completed transactions enqueue a 'Decrement' operation that can also transition the lease into 'invalid'.

[8]Even if revocation for some leases were delayed by outstanding DMA traffic, revocation for other leases would still be able to continue.

These experiments have shown IOCap hardware can handle large, high-throughput systems with low latency, F_{max}, and area overheads. The lease manager can handle one open and one close every four cycles, e.g. 250 million per second at 1GHz, far outpacing the maximum IOPS of my hypothetical high-performance system from Chapter 4. The enforcer can be configured to support 100% DMA throughput under any traffic, even that which requires 3 MACs per transaction, without any locality. The most complex configuration runs multiple times faster than my soft CPUs; using less than 5% of my FPGA, 1.5x the area of the CHERI CapChecker [30]; while providing much greater flexibility than the CapChecker and the IOMMU.

7.5 Full-system integration

In order to test the system under more realistic conditions, and verify that the software from Chapter 6 functions with real hardware, I integrated these units with the CHERI group's CHERI-BGAS SoC. CHERI-BGAS peripherals are implemented in software running on the DE10's onboard hard-logic Arm Cortex A53 cores [267], and the existing infrastructure (Fig. 7.6) is thus more complex than a representative system. In short: the host core, a version of the group's CHERI-Toooba [271, 272] running at 47MHz in soft logic, communicates with the Arm peripheral through MMIO (1). This MMIO is captured by a buffer, which is polled (2) by software on the Arm to detect incoming MMIO requests. Upon receiving a request, the Arm software (based on Bellard's tinyemu [273] VirtIO peripheral implementation) issues various DMAs—both to read the descriptors themselves, which the host stores in memory, and to perform the requested ops. These DMAs are issued 4-bytes at a time via custom 'ioctl' system calls, which ultimately end up issuing individual 4-byte AXI requests to an Arm-visible memory

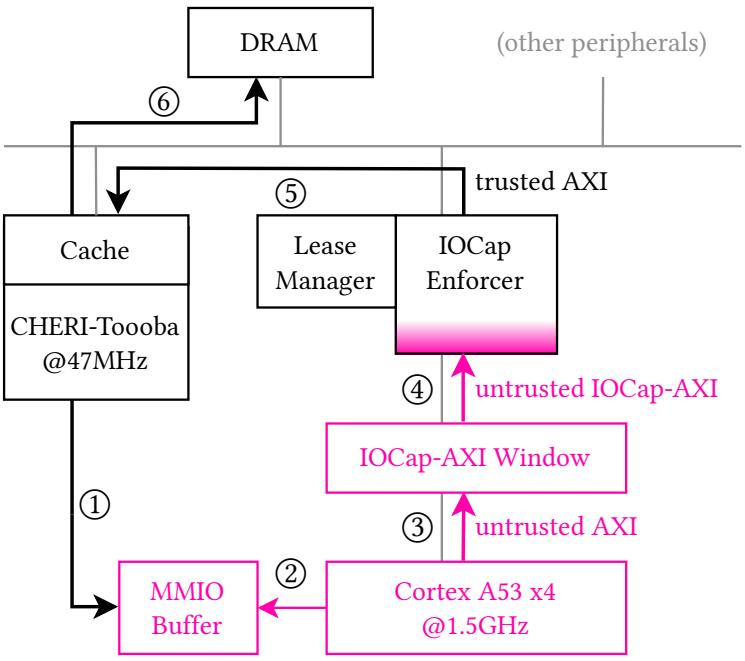


Figure 7.6: The general structure of the CHERI-BGAS SoC, showing the flow of data transfer starting from the host CPU initiating an op to the peripheral CPU accessing relevant DRAM.

region (3). When IOCaps are enabled, the Arm decodes them with `libccap`, and configures a custom hardware 'window' to attach an IOCap to the outgoing AXI and create IOCap-AXI (4). Configuring this window also requires eight 4-byte system calls per IOCap. After the IOCap-AXI is created, it is immediately passed to the enforcer for checking, which then passes it through to the last-level cache (5) that may or may not end in a DRAM read (6). This web of indirection, the relative speeds of soft logic Toooba vs. the hard logic Arm and the hard logic DRAM, and the inherently serial nature of tinyemu all make this a poor system for performance testing, but it is suitable to validate the functionality of the IOCap hardware and software.

The enforcer itself was slightly modified. I chose a version with 6 read checkers and 6 write checkers, larger than needed for maximum throughput, simply because it was the largest one I had available at the time. Despite its size, it did not prevent the full system from synthesizing for FPGA and reaching the target 47MHz clock frequency. More importantly this version was configured to *always* allow transactions, even if they fail IOCap decode or authentication, which was required for various elements of system bringup. CheriBSD expects DRAM to be filled with the kernel code and the device tree before execution begins, and control of the CPU itself is mediated by the debug unit—one of the other peripherals on the system bus, which is also gated behind the enforcer—which CHERI-BGAS controls and initializes through the Arm, which cannot directly control the enforcer. In a real system these bring-up tasks are handled by a Baseboard Management Controller (BMC), which pulls hardware modules out of reset and controls voltage and clock generators [274]. I suspect the BMC would need to be granted control of the enforcer at boot time, to set up the initial state before handing over to the host CPUs, but I leave the particulars of this integration for future work.

I ran both CheriFreeRTOS and CheriBSD on this system to ensure both worked—I focus on CheriBSD here as it is more complex. Once CheriBSD has been loaded into memory, it boots up just like it does in QEMU, including the IOCap-related drivers (Fig. 7.7). I set the system up to mount two IOCap-VirtIO block devices: one with no data, for testing, and one with a real filesystem. The filesystem kept integrity and correctly persisted data. I ran a cross-compiled version of `fio`, a disk benchmarking tool [275], to generate 4KiB read and write traffic to the other block device and stress test the system (Fig. 7.8). The system functioned correctly and did not lock up or crash, even when the queue depth generated by `fio` was large enough to exceed the IOCap-VirtIO queue. The performance counters indicated that IOCaps were being used correctly: after the boot process, the number of invalid accesses remained constant while the number of correct accesses increased. Disabling signature generation through a toggle in the `iocap_keymngr` driver then started providing the peripheral with invalid IOCaps, after which point the number of invalid accesses increased instead. There was a single crash in my experimentation, when trying to write a large log file, where the total number of required VirtIO descriptors to write the file exceeded the queue size—this was unrelated to IOCaps.

```
...
crypto: <crypto core>
ofwbus0: <Open Firmware Device Tree>
simplebus0: <Flattened device tree simple bus> on ofwbus0
iocap_keymngr0: <IOCap Key Manager> mem 0x50000000-0x500fffff on
    simplebus0
iocap_keymngr0: w00t attached to iocap!!! parent: 0xffffffd0073efa80
iocap_keymngr0: perf counters: 0 26926354 0 0
sbi0: <RISC-V Supervisor Binary Interface>
...
virtio_mmio1: <VirtIO MMIO adapter> mem 0x40001000-0x40001fff irq 5 on
    iocap_keymngr0
virtio_mmio2: <VirtIO MMIO adapter> mem 0x40002000-0x40002fff irq 6 on
    iocap_keymngr0
vtblk_iocap0: <VirtIO Block Adapter (IOCap)> on virtio_mmio2
virtio_mmio2: host features: 0x20100000004 <IOCap,Version1,MaxNumSegs>
virtio_mmio2: negotiated features: 0x20100000004 <IOCap,Version1,
    MaxNumSegs>
iocap_keymngr0: Allocated the following 1 IOCap key ids
iocap_keymngr0: key #0
virtio_mmio2: virtqueue 0 (vtblk_iocap0 request) requested indirect
    descriptors but not negotiated
iocap_keymngr0: Allocated the following 4 IOCap key ids
iocap_keymngr0: key #1
iocap_keymngr0: key #2
iocap_keymngr0: key #3
iocap_keymngr0: key #4
iocap_keymngr0: Assuming synchronous revocation
vtblk_iocap0: 102MB (208963 512 byte sectors)
...
```

Figure 7.7: Sample CheriBSD boot log, including messages from the `iocap_keymngr` and the IOCap-VirtIO block device.

```
+ sysctl dev.iocap_keymngr
dev.iocap_keymngr.0.disable_encrypt: 0
dev.iocap_keymngr.0.bad_read: 341
dev.iocap_keymngr.0.good_read: 13626
dev.iocap_keymngr.0.bad_write: 26927578
dev.iocap_keymngr.0.good_write: 127595
dev.iocap_keymngr.0.%pnpinfo: name=iocap_keymngr@50000000
    compat=sws35,iocap_keymngr
+ suffix=vtbd-iocap1_nj1_enc
+ tee ./mounted_results//results_randrw_10000_vtbd-iocap1_nj1_enc.txt
+ ./bench/fio '--filename=/dev/vtbd-iocap1' --numjobs 1 '--rw=randread'
    ./bench/fio_jobs.ini
4k-qd1: (g=0): rw=randread, bs=(R) 4096B-4096B, (W) 4096B-4096B,
    (T) 4096B-4096B, ioengine=posixaio, iodepth=1
4k-qd4: (g=1): rw=randread, bs=(R) 4096B-4096B, (W) 4096B-4096B,
    (T) 4096B-4096B, ioengine=posixaio, iodepth=4
4k-qd16: (g=2): rw=randread, bs=(R) 4096B-4096B, (W) 4096B-4096B,
    (T) 4096B-4096B, ioengine=posixaio, iodepth=16
4k-qd64: (g=3): rw=randread, bs=(R) 4096B-4096B, (W) 4096B-4096B,
    (T) 4096B-4096B, ioengine=posixaio, iodepth=64
fio-3.42-31-g6a66-dirty
Starting 4 processes

Run status group 0 (all jobs):
   READ: bw=210KiB/s (215kB/s), 210KiB/s-210KiB/s (215kB/s-215kB/s),
        io=7356KiB (7533kB), run=35016-35016msec
Run status group 1 (all jobs):
   READ: bw=807KiB/s (827kB/s), 807KiB/s-807KiB/s (827kB/s-827kB/s),
        io=27.6MiB (28.9MB), run=35008-35008msec
Run status group 2 (all jobs):
   READ: bw=2244KiB/s (2297kB/s), 2244KiB/s-2244KiB/s
        (2297kB/s-2297kB/s), io=76.7MiB (80.4MB), run=35017-35017msec
Run status group 3 (all jobs):
   READ: bw=2110KiB/s (2161kB/s), 2110KiB/s-2110KiB/s
        (2161kB/s-2161kB/s), io=72.2MiB (75.7MB), run=35021-35021msec
```

Figure 7.8: Sample console output running `fio` on CheriBSD on the full system. Redundant lines have been removed for brevity.

7.6 Summary

Hypothesis 4

IOCap protection can be practically and efficiently implemented in hardware, with low frequency, area, throughput, and latency overheads.

Over the course of this chapter I have explained how I implemented IOCaps efficiently in hardware, in both carrying them over the interconnect and checking them once they arrive. This is sufficient to validate my hypotheses on clock frequency, area, throughput, and latency.

Facet 4.1

The AXI interconnect standard can be extended to carry IOCaps with each transaction without significantly degrading clock frequency or area.

Facet 4.2

IOCap-AXI traffic can be routed through an IOCap enforcer and lease manager pair to check transactions without significantly degrading clock frequency or area.

These are both true. IOCap-AXI only extends the `USER` field by 2-bits for sanity checking, a minimal addition compared to the 64+-bit address and data channels. The interposer, which receives and parses IOCap-AXI, has a small area overhead and high F_{max} (Table 7.3, p162). An overprovisioned interposer was easily integrated into the CHERI-BGAS SoC without affecting F_{max}.

Facet 4.3

A lease manager can be implemented in hardware and simultaneously service configuration requests from the host and verification requests from an enforcer without significantly degrading latency or throughput.

This is true. The lease manager can accept a new configuration request every cycle and always responds within two (Section 7.4.3, p162). It does not degrade throughput. The only way it can degrade latency is when closing leases while transactions are in progress, but even then the impact is limited to that particular lease and does not hold up the rest of the system.

Facet 4.4

An enforcer can be implemented in hardware and simultaneously handle revocation requests from the host and DMA requests from peripherals without significantly degrading latency or throughput.

This is true. The enforcer as presented adds between 12–22 interposer cycles of latency, and can be provisioned to support 100% throughput for all traffic.

Facet 4.5

It is feasible to build IOCap memory protection into a complete system without losing functionality or significantly degrading performance.

This is true. I have integrated this hardware into an existing system-on-chip, controlled it using the software and principles derived from prior chapters, and performed I/O stress tests on it without any loss of functionality. The CHERI-BGAS experimental setup is not suitable for full-system performance measurement, but the microbenchmarks have demonstrated that the interconnect and hardware units reach maximum throughput with low latency, and any remaining overhead would exist in the host CPU or emulated peripheral software. Chapter 6 already notes potential software optimizations, and I consider applying them and finding others to be future engineering work.

7.6.1 Future work

CryptoMMU [45] noted that reads can be speculatively issued early, before transactions were validated, to improve latency. I believe this is correct for IOCap-AXI, although it could expose the read addresses as a side channel. It would also create more complex bookkeeping for the enforcer, along the same lines as the 'valve' module for write data, because if a speculative read turns out to be invalid the enforcer must block the data from flowing back to the peripheral.

CHAPTER 8

Conclusion

IOMMUs and other large C-list protection systems are slow to check and revoke access, limiting DMA throughput and forcing designers to trade performance for security. Over the course of this thesis I have presented and developed IOCaps: an alternative system which supports high-throughput checking, low-latency deterministic grouped revocation, and reduction/delegation from untrusted peripherals. The viability of this system relies on the four hypotheses established in the introduction.

Hypothesis 1, answered on p98

IOCap systems can use leases to enforce temporal safety under a variety of workloads using a 128-bit MAC function.

IOMMUs, when enabled, encourage trading performance for decreased security via deferred revocation. This creates unknown overexposure, where regions are exposed longer than the host expects, and if the host touches those regions it becomes vulnerable to attack. Prior systems assume individual revocation [30, 160] or infrequent revocation [45] and have not investigated how to fully leverage grouped revocation. IOCaps support individual revocation if enough lease SRAM is available, which is feasible, and they can dynamically scale to underprovisioned systems via grouped revocation. I found that while grouped revocation also creates overexposure, it creates safe, and manageable overexposure, and I determined ways to predict and measure it so system designers can make informed decisions. I identified CBC-AES-128 as a viable 128-bit MAC function, though like the interconnect and format it is a parameter that can be chosen freely by future adopters.

Hypothesis 2, answered on p132

The IOCap system can provide spatial safety at sub-4KiB precision using a 128-bit capability format.

Prior art needed to encode an arbitrary pointer as well as a region around it [247, 248, 250, 240], which limited the maximum compression. My initial region compression compresses

the region alone, by taking advantage of scaling alignment requirements, reusing redundant base bits for higher length precisions. The compact index and range caveats are also novel and necessary to represent precise regions and useful reductions. 64-bit formats are possible, although they do not have space for caveats and are limited to 48-bit addressing. I found that directly replacing VirtIO descriptors with IOCaps was possible but required saving bits by sacrificing other metadata. Alternatively, descriptors can be modified to place more metadata in a header in the exposed region.

Hypothesis 3, answered on p151

IOCap protection is practical to adopt in system software.

Operating systems do not spring fully formed from the minds of developers with IOCap support. If IOCaps are to be adopted, existing operating systems and peripheral drivers must be changed to support them in parallel with other protection systems. I went through this process for CheriFreeRTOS and CheriBSD, found it only required modest changes, and noted potential pain points for migrating other operating systems like Linux. I ran both operating systems in QEMU and on a full FPGA-based SoC, both using an adapted IOCap-VirtIO protocol to control peripherals, demonstrating the feasibility of IOCaps in a practical setting.

Hypothesis 4, answered on p168

IOCap protection can be practically and efficiently implemented in hardware, with low frequency, area, throughput, and latency overheads.

The core flaw with the IOMMU that prevents adoption at scale, even ignoring its security flaws, is performance—both in latency and throughput. Page table walks require multiple memory accesses in series, and when locality is low (as it often is [19]) performance suffers. IOCaps are designed to avoid this problem by relying on fast, local SRAM to store lease keys, and I have demonstrated this is the case in practice. My implementations have low area and frequency overheads on FPGA and provide 100% bulk DMA throughput with 12–22-cycle latency—a greater than 10× reduction compared to a four-level page table walk[1]—without relying on any kind of locality. This work culminated with the complete FPGA system, running multiple operating systems adapted to IOCaps on top of my hardware modules, demonstrating the system is viable in practice.

8.1 Related work

Prior art in I/O security is practically entirely based on the IOMMU. At the most basic level, an operating system may use the IOMMU to expose regions directly, or use it to expose 'bounce buffers' that data is copied through when needed. Both cases can use strict or deferred invalidation. DAMN [21] allocates growable pools of pinned bounce buffers for each peripheral.

[1]Assuming 1GHz operation, so 12–22ns, compared to 4 × ~100ns DRAM accesses in series.

This ensures data is not accidentally exposed, modulo implementation bugs (e.g. in CheriBSD, Section 6.2.3.2, p146), and sidesteps invalidation by assuming the buffers are exposed forever. Fast and Safe I/O (abbreviated as F&S) [141] adjusts the I/O virtual address space allocator to improve cache locality and avoid performance problems. CryptoMMU [45] uses MACs to cryptographically tag and authenticate PCIe ATS 'pre-translated' TLPs. Why use IOCaps when these systems exist, or at least have been proposed?

My issues with the basic IOMMU have been explored extensively in this work. Strict revocation is slow in practice, so systems use deferred revocation and introduce unexpected overexposure. Page granularity either forces exposed allocations to be padded out, to avoid accidentally exposing data not intended for a certain peripheral, or operating systems simply ignore this and end up exposing sensitive data and function pointers [9, 145]. Bounce buffers require extra copies, still require page table walks, and do not avoid overexposure risks. DAMN's pinned bounce buffers create large per-peripheral memory overheads and will still suffer from IOTLB locality issues if enough are allocated, as noted by Rubin et al. [141]. F&S is designed to optimize multi-page allocations and has not demonstrated improvements on small allocations, nor the claim that its security is equivalent to strict IOMMU invalidation. CryptoMMU's limited key pool will fail under frequent revocation (as I detailed in Section 4.1, p75) and still requires page table walks whenever a region is accessed for the first time (or after the relevant key is revoked). These solutions all still rely on locality, whereas IOCaps maintain maximum performance without needing any locality. On top of all this, IOCaps are generally more expressive and flexible than these IOMMU-based solutions, because they allow natively exposing sub-page regions, reduction, and delegation.

8.2 IOCaps vs. IOMMUs

I have created a system that avoids the pitfalls of the IOMMU, particularly its huge centralized state and reliance on locality. I have demonstrated that IOCaps are feasible to build and use with an FPGA proof of concept system, with latencies that are achievable on real hardware[2] and should compare favourably (aside from the lack of translation, which I discuss below). One thing I haven't done is directly compare my IOCap proof of concept to an IOMMU. This is not possible for a number of reasons—chief among them the fact that I do not have access to a mature FPGA IOMMU implementation, and that comparing like-for-like with real silicon IOMMUs would require real silicon IOCaps. Even then, any direct comparison would only be for one point in the parameter space. Thankfully, the high-level advantages of IOCaps are strong enough that a direct comparison is not necessary. Checking latency will be lower, and therefore throughput will be higher, because IOCaps do not need DRAM, caching, or I/O locality. Invalidation latency, the basis for IOMMU deferred invalidation and the security holes

[2]Software simulation, e.g. via gem5 as used by other work, risks reliance on unrealistic hardware models.

that come with it, will be reduced by at least an order of magnitude[3] because of the lease system. Fine-grained spatial safety will prevent accidental exposure of sensitive data that plagues the IOMMU [9, 145].

I leave it to future IOCap adopters to select appropriate parameters for their systems and build the hardware and software necessary, using my work as a guide. They will need to consider the minor differences in threat model (Section 3.2.4, p61), the entropy requirements (Section 6.1.3, p138), and the delegation and translation requirements of their system before any direct evaluation. Along the way, they may benefit from minor optimizations I have proposed throughout as future work.

8.3 Future work

I have already noted a few points for future investigation throughout this thesis. These can be split into novel research directions, engineering work that would be required to fully adopt IOCaps in practice, and alternate approaches that may be useful for specialized systems. Alternate approaches are noted in the early chapters including individual revocation (p62), minting IOCaps at the lease manager (p68, p138), locking lease IDs to specific peripherals (p69), various notes on specialized temporal (p87, p92, p93, p95) and spatial safety (p110, p134), and CHERI integration (p149). Engineering work includes encoding optimizations (p113, p130), CheriBSD driver improvements (p144, p145), advanced AXI compatibility (p154) and optimizations (p169), and integration with system boot (p165). I believe the most pertinent directions for novel research are those listed in the Non-Goals of the abstract system (Section 3.2.2, p59): peer-to-peer protection, cache coherency, and virtual memory.

Peer-to-peer protection, and generally using IOCaps to protect actors *from* the host, are an intriguing direction. This is not the typical threat model for consumer devices, but is relevant for cloud computing—for instance, protecting cloud-provided storage and networking from bare-metal guest VMs with AWS Nitro [164], trustless confidential computing where host-based enclaves need to be protected from each other [276], or extreme threat models with mutually distrusting host chiplets [10]. IOCaps could be issued by a root-of-trust which coordinates host resources, and could be used by the hosts as well as any peripherals to access memory, but ideally the hosts would still use plain pointers internally as much as possible. That implies IOCaps would be integrated into page tables or some similar construct, but I have not investigated this thoroughly.

IOCaps could be applied to cache-coherent protocols like CXL [78]—indeed, that was my inspiration for this thesis [35]—but these protocols use short messages that may not tolerate 256-bits of IOCap overhead. The first step in this direction would be to evaluate this impact in practice, and if it is not acceptable the next step would be to experiment with alternate formats. I have checked the feasibility of 64-bit MAC functions (Facet 1.2, p98) and text formats (Facet 2.2,

[3] Rajapaksha et al. measured IOMMU invalidation latency in 2024 and found a best-case average of 1,000ns and a worst-case maximum of 13,000ns [22]. An MMIO write to the I/O die will be comparable to an L3 hit, which is on the order of 50–100ns—at least an order of magnitude's improvement.

p132), and the latter was met with a resounding "maybe". If 48-bit addressing and no caveating are tolerable constraints, a 64-plus-64-bit format falls out of the work I've presented here. If not, further work is needed to investigate deeper compression or other tradeoffs, perhaps sacrificing MAC bits to pull out more base-address bits.

IOCaps do not support translation, which I believe is often used as a proxy for protection, but which may still be useful for interacting with virtual machines (the original IOMMU use case [91]) or userspace processes (e.g. NVIDIA Unified Memory [111] or mobile GPUs [103]). The IOCap format could sacrifice caveat bits to carry optional physical translations, similar to prior work [45, 277, 278, 279], while keeping the advantages of the lease system and grouped revocation. Layering fine-grained IOCap protection on top of coarse-grained (e.g. 1GiB) IOMMU translation could also allow virtualization without IOTLB locality issues. It should also be considered how the IOCap lease manager itself is virtualized: would virtual machines use the same MMIO interface to interact, proxied through the hypervisor? Would they mint IOCaps with guest-physical addresses, or would the hypervisor provide a richer interface which translates regions into host-physical addresses directly? All excellent questions, that I hope will be answered one day.

Bibliography

[1] Robert N. M. Watson et al. 'CHERI: Hardware-Enabled C/C++ Memory Protection at Scale'. In: *IEEE Security & Privacy* 22.4 (July 2024), pp. 50–61. ISSN: 1540-7993, 1558-4046. DOI: 10.1109/MSEC.2024.3396701.

[2] Jack B Dennis and Earl C van Horn. 'Programming Semantics for Multiprogrammed Computations'. In: *Communications of the ACM* 9.3 (1966). DOI: 10.1145/365230.365252.

[3] R. M. Needham and R. D.H. Walker. 'The Cambridge CAP Computer and Its Protection System'. In: *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*. SOSP '77. New York, NY, USA: Association for Computing Machinery, 1st November 1977, pp. 1–10. ISBN: 978-1-4503-7867-3. DOI: 10.1145/800214.806541.

[4] Jonathan S. Shapiro, Jonathan M. Smith and David J. Farber. 'EROS: A Fast Capability System'. In: *SIGOPS Oper. Syst. Rev.* 33.5 (12th December 1999), pp. 170–185. ISSN: 0163-5980. DOI: 10.1145/319344.319163.

[5] Robert N. M. Watson et al. 'Capsicum: Practical Capabilities for UNIX'. In: *Proceedings of the 19th USENIX Conference on Security*. USENIX Security'10. USA: USENIX Association, 11th August 2010, p. 3. DOI: 10.5555/1929820.1929824.

[6] M J Dworkin. *Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication*. NIST SP 800-38b. Gaithersburg, MD: National Institute of Standards and Technology, 2016, NIST SP 800–38b. DOI: 10.6028/NIST.SP.800-38b.

[7] Gal Beniamini. *Project Zero: Over The Air: Exploiting Broadcom's Wi-Fi Stack (Part 1)*. Project Zero. 4th April 2017. URL: https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html (visited on 17/10/2023).

[8] Tyler Sorensen and Heidy Khlaaf. *LeftoverLocals: Listening to LLM Responses Through Leaked GPU Local Memory*. 29th January 2024. arXiv: 2401.16603 [cs]. Pre-published.

[9] A. Theodore Markettos et al. 'Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals'. In: Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2019. ISBN: 978-1-891562-55-6. DOI: 10/gjh62d.

[10] Gino A. Chacon et al. 'Coherence Attacks and Countermeasures in Interposer-based Chiplet Systems'. In: *ACM Trans. Archit. Code Optim.* 21.2 (15th February 2024), 23:1–23:25. ISSN: 1544-3566. DOI: 10.1145/3633461.

[11] Arm Ltd. *Arm Architecture Reference Manual Supplement Armv8, for R-profile AArch64 Architecture*. Version B.a. 30th June 2022. URL: https://developer.arm.com/documentation/ddi0600/ba/.

[12] Arm Ltd. *Armv8-M Architecture Reference Manual*. Version B.y. 9th August 2024. URL: https://developer.arm.com/documentation/ddi0553/by/.

[13] RISC-V International. *RISC-V IOPMP Architecture Specification*. Version 0.8. RISC-V Non-ISA Specifications, 23rd July 2025. URL: https://github.com/riscv-non-isa/iopmp-spec/releases/tag/v0.8 (visited on 30/10/2025).

[14] Advanced Micro Devices, Inc. *AMD I/O Virtualization Technology (IOMMU) Specification, v3.10*. Rev 3.10-PUB. February 2025, p. 310. URL: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/specifications/48882_IOMMU.pdf (visited on 15/04/2025).

[15] Intel. *Intel Virtualization Technology for Directed I/O Architecture Specification*. Revision 5.0. August 2024. URL: https://www.intel.com/content/www/us/en/content-details/831418/intel-virtualization-technology-for-directed-i-o-architecture-specification.html (visited on 08/05/2025).

[16] Arm Ltd. *ARM System Memory Management Unit Architecture Specification - SMMU Architecture Version 3.3*. Issue F.a. 12th September 2023. URL: https://developer.arm.com/documentation/ihi0070/fa/?lang=en (visited on 08/05/2025).

[17] Moshe Malka et al. 'rIOMMU: Efficient IOMMU for I/O Devices That Employ Ring Buffers'. In: *ACM SIGARCH Computer Architecture News* 43.1 (14th March 2015), pp. 355–368. ISSN: 0163-5964. DOI: 10.1145/2786763.2694355.

[18] Rolf Neugebauer et al. 'Understanding PCIe Performance for End Host Networking'. In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 7th August 2018, pp. 327–341. ISBN: 978-1-4503-5567-4. DOI: 10.1145/3230543.3230560.

[19] Alireza Farshin et al. 'Overcoming the IOTLB Wall for Multi-100-Gbps Linux-based Networking'. In: *PeerJ Computer Science* 9 (16th May 2023), e1385. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.1385. PMID: 37346709.

[20] Omer Peleg et al. 'Utilizing the IOMMU Scalably'. In: 2015 USENIX Annual Technical Conference (USENIX ATC 15). USENIX Association, 2015, pp. 549–562. ISBN: 978-1-931971-22-5. DOI: 10.5555/2813767.2813809.

[21] Alex Markuze et al. 'DAMN: Overhead-Free IOMMU Protection for Networking'. In: *SIGPLAN Not.* 53.2 (19th March 2018), pp. 301–315. ISSN: 0362-1340. DOI: 10.1145/3296957.3173175.

[22] Chathura Rajapaksha et al. 'IOMMU Deferred Invalidation Vulnerability: Exploit and Defense'. In: *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. March 2024, pp. 1–6. DOI: 10.23919/DATE58400.2024.10546528.

[23] J.H. Saltzer and M.D. Schroeder. 'The Protection of Information in Computer Systems'. In: *Proceedings of the IEEE* 63.9 (September 1975), pp. 1278–1308. ISSN: 1558-2256. DOI: 10.1109/PROC.1975.9939.

[24] NVM Express, Inc. *NVM Express Base Specification*. Revision 1.4c. 9th March 2021. URL: https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4c-2021.06.28-Ratified.pdf (visited on 19/05/2023).

[25] Paul Emmerich et al. 'User Space Network Drivers'. In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. September 2019, pp. 1–12. DOI: 10.1109/ANCS.2019.8901894.

[26] Michael S. Turskin and Cornelia Huck, eds. *Virtual I/O Device (VIRTIO)*. Version 1.4. 8th April 2026. URL: https://docs.oasis-open.org/virtio/virtio/v1.4/virtio-v1.4.html (visited on 11/04/2026).

[27] Friedrich Doku et al. *CAPIO: Safe Kernel-Bypass of Commodity Devices Using Capabilities*. 18th December 2025. arXiv: 2512.16957 [cs]. Pre-published.

[28] Paul Metzger et al. 'Deprivileging Low-Level GPU Drivers Efficiently with User-Space Processes and CHERI Compartments'. In: *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. CCS '25. New York, NY, USA: Association for Computing Machinery, 22nd November 2025, pp. 2699–2713. ISBN: 979-8-4007-1525-9. DOI: 10.1145/3719027.3765036.

[29] Matthew Naylor et al. *CHERI-SIMT Report: Implementing Capability Memory Protection in GPGPUs*. UCAM-CL-TR-997. University of Cambridge, Computer Laboratory, 2025. DOI: 10.48456/tr-997.

[30] Jianyi Cheng et al. 'Adaptive CHERI Compartmentalization for Heterogeneous Accelerators'. In: *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. ISCA '25. New York, NY, USA: Association for Computing Machinery, 20th June 2025, pp. 2002–2016. ISBN: 979-8-4007-1261-6. DOI: 10.1145/3695053.3731062.

[31] Arnar Birgisson et al. 'Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud'. In: *Network and Distributed System Security Symposium*. 2014. DOI: 10.14722/ndss.2014.23212.

[32] B.C. Neuman. 'Proxy-Based Authorization and Accounting for Distributed Systems'. In: *Proceedings of the 13th International Conference on Distributed Computing Systems*. May 1993, pp. 283–291. DOI: 10.1109/ICDCS.1993.287698.

[33] Dapeng Gao and Robert N. M. Watson. 'Library-Based Compartmentalisation on CHERI'. In: *Programming Languages for Architecture 2023*. Orlando, FL, USA, June 2023. URL: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/202306-plarch-library-based-compartmentalisation.pdf.

[34] Franz A. Fuchs. *Toward Transient-Execution Attack Mitigations on CHERI*. UCAM-CL-TR-1001. University of Cambridge, Computer Laboratory, 2025. DOI: 10.48456/tr-1001.

[35] Samuel W. Stark, A. Theodore Markettos and Simon W. Moore. 'How Flexible Is CXL's Memory Protection?' In: *Communications of the ACM* 66.12 (17th November 2023), pp. 46–51. ISSN: 0001-0782. DOI: 10.1145/3617580.

[36] CHERI Research Centre. *CTSRD-CHERI/DE10Pro-cheri-bgas*. 20th November 2025. URL: https://github.com/CTSRD-CHERI/DE10Pro-cheri-bgas (visited on 12/01/2026).

[37] CHERI Research Centre. *CTSRD-CHERI/FreeRTOS*. 20th May 2024. URL: https://github.com/CTSRD-CHERI/FreeRTOS/tree/hmka2 (visited on 08/12/2025).

[38] CHERI Research Centre. *CTSRD-CHERI/Cheribsd: FreeBSD Adapted for CHERI-RISC-V and Arm Morello*. 4th May 2026. URL: https://github.com/CTSRD-CHERI/cheribsd (visited on 14/06/2026).

[39] CHERI Research Centre. *CTSRD-CHERI/Qemu*. 12th May 2026. URL: https://github.com/CTSRD-CHERI/qemu (visited on 21/05/2026).

[40] Stephan Hodes and Alex Dunn. 'Deep Dive: Asynchronous Compute'. Game Developers Conference. 2017. URL: https://gpuopen.com/download/GDC2017-Asynchronous-Compute-Deep-Dive.pdf (visited on 30/07/2025).

[41] Vladimir Bondarev and Sriharsha Niverty. *Advanced API Performance: Async Compute and Overlap*. NVIDIA Technical Blog. 22nd October 2021. URL: https://developer.nvidia.com/blog/advanced-api-performance-async-compute-and-overlap/ (visited on 30/07/2025).

[42] René van Oostrum et al. 'AMD GPU Hardware Basics'. Frontier Application Readiness Kick-Off Workshop. October 2019. URL: https://www.olcf.ornl.gov/wp-content/uploads/2019/10/ORNL_Application_Readiness_Workshop-AMD_GPU_Basics.pdf (visited on 30/07/2025).

[43] Silas Boyd-Wickizer and Nickolai Zeldovich. 'Tolerating Malicious Device Drivers in Linux'. In: 2010 USENIX Annual Technical Conference (USENIX ATC 10). USENIX Association, 2010. DOI: 10.5555/1855840.1855849.

[44] A. Theodore Markettos et al. 'Position Paper: Defending Direct Memory Access with CHERI Capabilities'. In: *Hardware and Architectural Support for Security and Privacy*. Virtual Greece: ACM, 17th October 2020, pp. 1–9. ISBN: 978-1-4503-8898-6. DOI: 10.1145/3458903.3458910.

[45] Faiz Alam et al. 'CryptoMMU: Enabling Scalable and Secure Access Control of Third-Party Accelerators'. In: *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '23. New York, NY, USA: Association for Computing Machinery, 8th December 2023, pp. 32–48. ISBN: 979-8-4007-0329-4. DOI: 10.1145/3613424.3614311.

[46] Simon Gerber et al. 'Not Your Parents' Physical Address Space'. In: 15th Workshop on Hot Topics in Operating Systems (HotOS XV). USENIX Association, 2015. DOI: 10.5555/2831090.2831106.

[47] James E.J. Bottomley. *Dynamic DMA Mapping Using the Generic Device — The Linux Kernel Documentation*. 21st July 2020. URL: https://www.kernel.org/doc/Documentation/DMA-API.txt (visited on 15/12/2023).

[48] FreeBSD Community. 'Chapter 10. ISA Device Drivers'. In: *FreeBSD Architecture Handbook*. 18th February 2025. URL: https://docs.freebsd.org/en/books/arch-handbook/isa/ (visited on 08/12/2025).

[49] Chris S Lin et al. 'GPUBreach: Privilege Escalation Attacks on GPUs Using Rowhammer'. In: *Proceedings of the 2026 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA, May 2026. URL: http://arxiv.org/abs/2605.03812.

[50] Matej Bölcskei et al. 'Rubicon: Precise Microarchitectural Attacks with Page-Granular Massaging'. In: *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*. June 2025, pp. 395–410. DOI: 10.1109/EuroSP63326.2025.00031.

[51] Diego Meyer et al. 'Phoenix: Rowhammer Attacks on DDR5 with Self-Correcting Synchronization'. In: *Proceedings of the 2026 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA, May 2026. URL: https://comsec-files.ethz.ch/papers/phoenix_sp26.pdf.

[52] Rick Wertenbroek and Alberto Dassatti. *Pandora's Box in Your SSD: The Untold Dangers of NVMe*. Version 1. 1st November 2024. arXiv: 2411.00439 [cs]. Pre-published.

[53] Advanced Micro Devices, Inc. *AXI Firewall*. AMD. 2026. URL: https://www.amd.com/en/products/adaptive-socs-and-fpgas/intellectual-property/axi-firewall.html (visited on 02/03/2026).

[54] Per Idenfeldt Okuyama and Sam Eizad. *CVE-2026-0714 - TPM-sniffing LUKS Keys on an Embedded Device*. Cyloq. 17th February 2026. URL: https://www.cyloq.se/en/research/cve-2026-0714-tpm-sniffing-luks-keys-on-an-embedded-device (visited on 02/03/2026).

[55] Philipp Giersfeld, Benedict Schluter and Shweta Shinde. 'BREAKFAST: Confused Deputy Attack on Infinity Fabric to Break AMD SEV-SNP'. In: *Proceedings of the 2026 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA, May 2026. URL: https://www.shwetashinde.org/publications/breakfast_oakland26.pdf.

[56] Ravi Budruk, Don Anderson and Tom Shanley. *PCI Express System Architecture*. Ed. by Joe Winkles. Boston: Addison-Wesley, 2004. 1049 pp. ISBN: 978-0-321-15630-3.

[57] PCI-SIG. *PCI Express® Base Specification*. Revision 5.0 Version 1.0. 22nd May 2019, p. 1299.

[58] Intel. *Intel® Data Direct I/O Technology (Intel® DDIO): A Primer*. Revision 1.0. February 2012. URL: https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf (visited on 13/06/2026).

[59] Alireza Farshin et al. 'Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks'. In: 2020 USENIX Annual Technical Conference (USENIX ATC 20). USENIX Association, 2020, pp. 673–689. ISBN: 978-1-939133-14-4. DOI: 10.5555/3489146.3489192.

[60] "Embedded Staff". *Using Multi-Root (MR) PCIe to Extend Nextgen Multi-Host Storage & Server Switch Fabrics*. Embedded.com. 19th May 2009. URL: https://www.embedded.com/using-multi-root-mr-pcie-to-extend-nextgen-multi-host-storage-server-switch-fabrics/ (visited on 25/11/2022).

[61] Debendra Das Sharma. *Compute Express Link 3.0*. August 2022. URL: https://computeexpresslink.org/wp-content/uploads/2023/12/CXL_3.0_white-paper_FINAL.pdf (visited on 16/06/2026).

[62] Danny Moore and Debendra Das Sharma. 'CXL 3.0: Enabling Composable Systems with Expanded Fabric Capabilities'. Webinar. 26th October 2022. URL: https://www.youtube.com/watch?v=CIjDpazbtUU (visited on 06/12/2022).

[63] Raghu Makaram and David Harriman. 'Compute Express Link™ (CXL™) Link-level Integrity and Data Encryption (CXL IDE)'. Webinar. 30th September 2021. URL: https://www.youtube.com/watch?v=PV6FJZ3wJGM.

[64] David Harriman. 'Integrity and Data Encryption (IDE) ECN Deep Dive'. 25th August 2020. URL: https://pcisig.com/sites/default/files/files/PCIe%20Security%20Webinar_Aug%202020_PDF.pdf.

[65] Arm Ltd. *AMBA AXI Protocol Specification*. Issue K. 29th September 2023. URL: https://developer.arm.com/documentation/ihi0022/k/.

[66] Arm Ltd. *AMBA Specification*. Rev. 2.0-A. 13th May 1999. URL: https://developer.arm.com/documentation/ihi0011/a/.

[67] Arm Ltd. *AMBA AXI and ACE Protocol Specification*. Issue H.c. 26th January 2021. URL: https://developer.arm.com/documentation/ihi0022/hc.

[68] Arm Ltd. *AMBA CHI Architecture Specification*. Issue H. 17th September 2025. URL: https://developer.arm.com/documentation/ihi0050/h/.

[69] Arm Ltd. *AMBA LTI Protocol Specification*. Issue C. 1st July 2024. URL: https://developer.arm.com/documentation/ihi0089/c/.

[70] Arm Ltd. *AMBA DTI Protocol Specification*. Issue G. 28th June 2024. URL: https://developer.arm.com/documentation/ihi0088/g/.

[71] Wei Zhou, Zhouqi Jiang and Le Guan. 'Understanding MPU Usage in Microcontroller-based Systems in the Wild' (San Diego, CA, USA). 2023. DOI: 10.14722/bar.2023.23007.

[72] Lattice Semiconductor. *RISC-V AXI4 I/O Physical Memory Protection IP – Lattice Propel Builder 2025.1*. FPGA-IPUG-02283-1.0. June 2025. URL: https://www.latticesemi.com/view_document?document_id=54652.

[73] Jien Hau Ng, Chee Hong Ang and Hwa Chaw Law. 'A Realization of IO Physical Memory Protection for RISC-V Systems'. In: *2022 IEEE 15th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. December 2022, pp. 375–380. DOI: 10.1109/MCSoC57363.2022.00066.

[74] Nils Wistoff et al. 'Protego: A Low-Overhead Open-Source I/O Physical Memory Protection Unit for RISC-V'. In: *Proceedings of the 1st Safety and Security in Heterogeneous Open System-on-Chip Platforms Workshop (SSH-SoC 2023)* (July 2023), 4 p. DOI: 10.3929/ETHZ-B-000647375. HDL: 20.500.11850/647375.

[75] Luís Cunha et al. 'Open-Source RISC-V Input/Output Physical Memory Protection (IOPMP) IP'. RISC-V Summit Europe 2024 (Munich). 27th June 2024. URL: https://riscv-europe.org/summit/2024/media/proceedings/plenary/Thu-17-00-Luis-Cunha.pdf.

[76] Samsung Semiconductor. *Samsung Electronics Introduces Industry's First 512GB CXL Memory Module*. URL: https://news.samsung.com/global/samsung-electronics-introduces-industrys-first-512gb-cxl-memory-module (visited on 13/12/2022).

[77] Donghyun Gouk et al. 'CXL–GPU: Pushing GPU Memory Boundaries with the Integration of CXL Technologies'. In: *IEEE Micro* 45.6 (November 2025), pp. 108–117. ISSN: 1937-4143. DOI: 10.1109/MM.2025.3582433.

[78] CXL Consortium. *Compute Express Link (CXL) Specification Revision 3.0, Version 1.0*. 1st August 2022. URL: https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.0-Specification.pdf.

[79] Arm Ltd. *Learn the Architecture - TrustZone for AArch64*. Issue 0102-01 102418. 29th October 2024, p. 53. URL: https://developer.arm.com/documentation/102418/latest/.

[80] Arm Ltd. *ARM CoreLink TZC-400 TrustZone Address Space Controller Technical Reference Manual*. r0p1. 2013. URL: https://developer.arm.com/documentation/100325/0001?lang=en (visited on 01/07/2025).

[81] Arm Ltd. *ARM Security Technology - Building a Secure System Using TrustZone Technology*. Issue C. April 2009. URL: https://developer.arm.com/documentation/PRD29-GENC-009492/c/.

[82] Arm Ltd. *ARM System Memory Management Unit Architecture Specification - SMMU Architecture Version 1.0*. Issue B. 18th December 2012. URL: https://developer.arm.com/documentation/ihi0062/b/ (visited on 08/05/2025).

[83] Arm Ltd. *ARM System Memory Management Unit Architecture Specification - SMMU Architecture Version 2.0*. Issue D.c. 30th June 2016. URL: https://developer.arm.com/documentation/ihi0062/dc/ (visited on 08/05/2025).

[84] Arm Ltd. *Learn the Architecture - Realm Management Extension Guide*. Issue 0101-02 den0126. 12th February 2025. URL: https://developer.arm.com/documentation/den0126/0101.

[85] Xupeng Li et al. *Enabling Realms with the Arm Confidential Compute Architecture*. USENIX. 5th July 2023. URL: https://www.usenix.org/publications/loginonline/enabling-realms-arm-confidential-compute-architecture (visited on 02/03/2026).

[86] RISC-V International. *The RISC-V Instruction Set Manual: Volume II: Privileged Architecture - In-Progress Worlds Release*. 2026-03-07. 7th March 2026. URL: https://github.com/riscv/riscv-worlds/releases/tag/riscv-isa-release-b2d8551-2026-03-07.

[87] Yann Loisel. *Worldguard Specification*. E-mail. 19th May 2020. URL: https://lists.riscv.org/g/tech-tee/topic/worldguard_specification/74326553 (visited on 03/03/2026).

[88] SiFive. *WorldGuard Specification*. Revision 0.3. 15th May 2020. URL: https://lists.riscv.org/g/tech-tee/attachment/266/0/WorldGuard%20High-level%20specification.pdf.

[89] Intel. *Accelerated Graphics Port Interface Specification 1.0*. 31st July 1996. URL: https://www.dcs.ed.ac.uk/home/ecole/agp/agp10.pdf (visited on 11/06/2025).

[90] Intel. *Accelerated Graphics Port Interface Specification 3.0*. Revision 1.0. September 2002. URL: https://www.playtool.com/pages/agpcompat/agp30.pdf (visited on 11/06/2025).

[91] Advanced Micro Devices, Inc. *AMD I/O Virtualization Technology (IOMMU) Specification, v1.00*. Rev 1.00. 1st February 2006. URL: https://web.archive.org/web/20060214161447/http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf (visited on 08/05/2025).

[92] Intel. *PCI-SIG Single Root I/O Virtualization (SR-IOV) Support in Intel® Virtualization Technology for Connectivity*. Rev. 06/08-001US. 2008, p. 4. URL: https://www.intel.com/content/dam/doc/white-paper/pci-sig-single-root-io-virtualization-support-in-virtualization-technology-for-connectivity-paper.pdf.

[93] Intel, Microsoft and Open Compute Project. *Scalable I/O Virtualization*. Revision 1.0 Version 1.3. June 2022. URL: https://www.opencompute.org/documents/ocp-scalable-io-virtualization-technical-specification-revision-1-0-version-1-3-pdf (visited on 20/02/2023).

[94] Arm Ltd. *ARM CoreLink MMU-400 System Memory Management Unit Technical Reference Manual*. Revision r0p1. 2011. URL: https://developer.arm.com/documentation/ddi0472/b/.

[95] Arm Ltd. *ARM CoreLink MMU-401 System Memory Management Unit Technical Reference Manual*. Revision r0p1. 2013. URL: https://developer.arm.com/documentation/ddi0521/a/.

[96] Arm Ltd. *ARM CoreLink MMU-500 System Memory Management Unit Technical Reference Manual*. Revision r2p4. 2013. URL: https://developer.arm.com/documentation/ddi0517/f/.

[97] NXP Semiconductors. *I.MX 8QuadMax Applications Processor Reference Manual Rev.0*. IMX8QMRM. September 2021.

[98] RISC-V International. *RISC-V IOMMU Architecture Specification*. Version v1.0.1, 20260222. 22nd February 2026. URL: https://github.com/riscv-non-isa/riscv-iommu/releases/tag/20260222.

[99] Qing Li et al. *An Introduction to Access Control on Qualcomm Snapdragon Platforms*. 16th September 2020. URL: https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/whitepaper_0.pdf (visited on 09/05/2025).

[100] Linux Kernel Dev Community. *Samsung Exynos IOMMU H/W, System MMU (System Memory Management Unit) — The Linux Kernel Documentation*. URL: https://www.kernel.org/doc/Documentation/devicetree/bindings/iommu/samsung,sysmmu.txt (visited on 01/07/2025).

[101] Gal Beniamini. *Project Zero: Over The Air - Vol. 2, Pt. 3: Exploiting The Wi-Fi Stack on Apple Devices*. Project Zero. 11th October 2017. URL: https://googleprojectzero.blogspot.com/2017/10/over-air-vol-2-pt-3-exploiting-wi-fi.html (visited on 01/07/2025).

[102] Sven Peter. *[PATCH 1/3] Iommu: Io-Pgtable: Add DART Pagetable Format [LWN.Net]*. E-mail. 20th March 2021. URL: https://lwn.net/ml/linux-kernel/20210320151903.60759-2-sven@svenpeter.dev/ (visited on 01/07/2025).

[103] Jordan Crouse. 'LVC20-309 - The Arm SMMU and the Adreno GPU'. Linaro Connect Virtual 2020. 24th September 2020. URL: https://www.youtube.com/watch?v=Az6d4UXzhg4 (visited on 16/06/2026).

[104] *CVE-2019-10567*. 7th February 2020. URL: https://www.cve.org/CVERecord?id=CVE-2019-10567 (visited on 06/03/2026).

[105] Guang Gong. 'TiYunZong: An Exploit Chain to Remotely Root Modern Android Devices'. In: Blackhat USA 2020. 5th August 2020, p. 22. URL: https://i.blackhat.com/USA-20/Thursday/us-20-Gong-TiYunZong-An-Exploit-Chain-To-Remotely-Root-Modern-Android-Devices-wp.pdf.

[106] Ben Hawkes. *Project Zero: Attacking the Qualcomm Adreno GPU*. Project Zero. 8th September 2020. URL: https://googleprojectzero.blogspot.com/2020/09/attacking-qualcomm-adreno-gpu.html (visited on 30/06/2023).

[107] Nitay Artenstein. *Broadpwn: Remotely Compromising Android and iOS via a Bug in Broadcom's Wi-Fi Chipsets*. Exodus Intelligence. 26th July 2017. URL: https://blog.exodusintel.com/2017/07/26/broadpwn/ (visited on 17/10/2023).

[108] Gal Beniamini. *Project Zero: Over The Air: Exploiting Broadcom's Wi-Fi Stack (Part 2)*. Project Zero. 11th April 2017. URL: https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_11.html (visited on 01/07/2025).

[109] NVIDIA Corporation. *NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge*. NVIDIA Developer Blog. 8th March 2017. URL: https://developer.nvidia.com/blog/jetson-tx2-delivers-twice-intelligence-edge/ (visited on 31/12/2021).

[110] NVIDIA Corporation. *NVIDIA Grace Hopper Superchip Architecture Whitepaper*. URL: https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper (visited on 28/07/2025).

[111] NVIDIA Corporation and Mark Harris. *Unified Memory for CUDA Beginners*. NVIDIA Developer Blog. 20th June 2017. URL: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/ (visited on 31/12/2021).

[112] Stephen Foskett. *ExpressCard: A Sure Thing That Failed*. Stephen Foskett, Pack Rat. 8th September 2010. URL: https://blog.fosketts.net/2010/09/08/expresscard-failed/ (visited on 05/06/2026).

[113] Richard C. Moss. *The Tragedy of FireWire: Collaborative Tech Torpedoed by Corporations*. Ars Technica. 22nd June 2017. URL: https://arstechnica.com/gadgets/2017/06/the-rise-and-fall-of-firewire-the-standard-everyone-couldnt-quite-agree-on/ (visited on 05/06/2026).

[114] Intel. *Thunderbolt Technology Brief*. 2012. URL: https://www.intel.com/content/dam/doc/technology-brief/thunderbolt-technology-brief.pdf (visited on 14/07/2025).

[115] Jason Ziller. *Thunderbolt™ 3 – The USB-C That Does It All*. 2nd June 2015. URL: https://www.thunderbolttechnology.net/blog/thunderbolt-3-usb-c-does-it-all (visited on 01/04/2026).

[116] Gamma Group. *FinFireWire 2.0 Product Specifications*. 17th August 2011. URL: https://wikileaks.org/spyfiles4/documents/FinFireWire-2.0-Specifications.doc.

[117] Sean Gallagher. *New WikiLeaks Dump: The CIA Built Thunderbolt Exploit, Implants to Target Macs*. Ars Technica. 23rd March 2017. URL: https://arstechnica.com/information-technology/2017/03/new-wikileaks-dump-the-cia-built-thunderbolt-exploit-implants-to-target-macs/ (visited on 31/03/2026).

[118] Fernand Lone Sang, Vincent Nicomette and Yves Deswarte. 'I/O Attacks in Intel PC-based Architectures and Countermeasures'. In: *2011 First SysSec Workshop*. July 2011, pp. 19–26. DOI: 10.1109/SysSec.2011.10.

[119] Carsten Maartmann-Moe. *INCEPTION*. 2012. URL: https://github.com/carmaa/inception (visited on 31/03/2026).

[120] Joe FitzPatrick and Miles Crabill. 'NSA Playset: PCIe'. DEFCON 22. August 2014. URL: https://www.youtube.com/watch?v=OD2Wxe4RLeU (visited on 31/03/2026).

[121] Ulf Frisk. 'Direct Memory Attack the Kernel'. DEFCON 24. August 2016. URL: https://www.youtube.com/watch?v=fXthwl6ShOg (visited on 31/03/2026).

[122] Intel. *Thunderbolt™ Security – Helps Keep Your Computer Ports More Secure*. 13th July 2020. URL: https://web.archive.org/web/20200713152216/https://newsroom.intel.com/wp-content/uploads/sites/11/2020/07/intel-thunderbolt4-security-brief.pdf (visited on 01/07/2025).

[123] Microsoft. *Kernel DMA Protection*. 15th August 2025. URL: https://learn.microsoft.com/en-us/windows/security/hardware-security/kernel-dma-protection-for-thunderbolt (visited on 06/03/2026).

[124] Mika Westerberg. *PCI / ACPI: Identify Untrusted PCI Devices · Torvalds/Linux@617654a*. GitHub. 5th December 2018. URL: https://github.com/torvalds/linux/commit/617654aae50eb59dd98aa53fb562e850937f4cde (visited on 30/03/2026).

[125] Lu Baolu and Mika Westerberg. *Iommu/vt-d: Force IOMMU on for Platform Opt in Hint · Torvalds/Linux@89a6079*. GitHub. 5th December 2018. URL: https://github.com/torvalds/linux/commit/89a6079df791aeace2044ea93be1b397195824ec (visited on 30/03/2026).

[126] Mika Westerberg. *Iommu/vt-d: Do Not Enable ATS for Untrusted Devices · Torvalds/Linux@fb58fdc*. GitHub. 5th December 2018. URL: https://github.com/torvalds/linux/commit/fb58fdcd295b914ece1d829b24df00a17a9624bc (visited on 30/03/2026).

[127] Evgeniy Ivanov. *How Io_uring Overtook Libaio: Performance Across Linux Kernels — and an Unexpected IOMMU Trap*. Medium. 24th March 2026. URL: https://blog.ydb.tech/how-io-uring-overtook-libaio-performance-across-linux-kernels-and-an-unexpected-iommu-trap-ea6126d9ef14 (visited on 30/03/2026).

[128] Andrea Righi. *Bug #1951440 "Enable CONFIG_INTEL_IOMMU_DEFAULT_ON and CONFIG_INTEL_IOMMU_SCALABLE_MODE_DEFAULT_ON"*. Ubuntu Launchpad. 18th November 2021. URL: https://bugs.launchpad.net/bugs/1951440 (visited on 30/03/2026).

[129] Vasant Hegde. 'IOMMU: How It Helps to Provide IO Security'. Bangalore Linux Kernel Meetup. 4th November 2023. URL: https://kernelmeetup.wordpress.com/wp-content/uploads/2023/11/dt_2_kernel-meetup-iommu_v3.pdf.

[130] Tony Chen. 'Guarding Against Physical Attacks: The Xbox One Story — Tony Chen, Microsoft'. Platform Security Summit 2019. 1st October 2019. URL: https://www.youtube.com/watch?v=U7VwtOrwceo (visited on 07/04/2026).

[131] Markus Gaasedelen. 'Hacking the Xbox One'. RE//Verse 2026 (Orlando, FL, USA). 13th March 2026. URL: https://www.youtube.com/watch?v=FTFn4UZsA5U (visited on 07/04/2026).

[132] Sam Collins et al. 'Anti-Cheat: Attacks and the Effectiveness of Client-Side Defences'. In: *Proceedings of the 2024 Workshop on Research on Offensive and Defensive Techniques in the Context of Man At The End (MATE) Attacks*. Salt Lake City UT USA: ACM, 19th November 2024, pp. 30–43. ISBN: 979-8-4007-1230-2. DOI: 10.1145/3689934.3690816.

[133] Electronic Arts. *Battlefield™ 6 System Requirements for PC - Electronic Arts*. 10th March 2026. URL: https://web.archive.org/web/20260310145541/https://www.ea.com/games/battlefield/battlefield-6/system-requirements (visited on 30/03/2026).

[134] *FACEIT Finalises Windows Security Rollout: TPM 2.0, Secure Boot, IOMMU & VBS.* FACEIT. 23rd October 2025. URL: https://www.faceit.com/en/news/faceit-rollout-of-tpm-secure-boot-iommu-and-vbs (visited on 02/04/2026).

[135] Epic Games. *New Fortnite Anti-Cheat Requirements and Legal Action.* 10th February 2026. URL: https://web.archive.org/web/20260318042426/https://www.fortnite.com/news/new-fortnite-anti-cheat-pc-requirements-and-latest-legal-action (visited on 30/03/2026).

[136] SD Association. *SD Express Memory Cards with PCIe and NVMe Interfaces.* 2020. URL: https://www.sdcard.org/pdf/SD_Cards_8_0_WhitePaper20200515.pdf (visited on 05/06/2026).

[137] Yichang Hu et al. 'GDDRHammer: Greatly Disturbing DRAM Rows — Cross-Component Rowhammer Attacks from Modern Gpus'. In: *Proceedings of the 2026 IEEE Symposium on Security and Privacy (SP).* San Francisco, CA, USA, May 2026. URL: https://gddr.fail/files/gddr.pdf.

[138] Junpeng Wan et al. 'GeForge: Hammering GDDR Memory to Forge GPU Page Tables for Fun and Profit'. In: *Proceedings of the 2026 IEEE Symposium on Security and Privacy (SP).* San Francisco, CA, USA, May 2026. URL: https://gddr.fail/files/GeForge.pdf.

[139] Taehun Kim et al. 'DevIOus: Device-Driven Side-Channel Attacks on the IOMMU'. In: *2023 IEEE Symposium on Security and Privacy (SP).* May 2023, pp. 2288–2305. DOI: 10.1109/SP46215.2023.10179283.

[140] Linux Kernel Dev Community. *Page Cache — The Linux Kernel Documentation.* 7th August 2024. URL: https://docs.kernel.org/mm/page_cache.html (visited on 15/12/2025).

[141] Benny Rubin et al. 'Fast & Safe IO Memory Protection'. In: *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles.* SOSP '24. New York, NY, USA: Association for Computing Machinery, 15th November 2024, pp. 95–109. ISBN: 979-8-4007-1251-7. DOI: 10.1145/3694715.3695943.

[142] Andrea Righi. *Bug #1971699 "Disable Intel DMA Remapping by Default" : Bugs : Linux Package : Ubuntu.* Ubuntu Launchpad. 5th May 2022. URL: https://bugs.launchpad.net/bugs/1971699 (visited on 01/04/2026).

[143] Gabriel Haas and Viktor Leis. 'What Modern NVMe Storage Can Do, and How to Exploit It: High-Performance I/O for High-Performance Storage Engines'. In: *Proceedings of the VLDB Endowment* 16.9 (May 2023), pp. 2090–2102. ISSN: 2150-8097. DOI: 10.14778/3598581.3598584.

[144] Mark Nelson. *A Journey to 1 TiB/s - A Milestone in Performance*. Clyso. 5th April 2025. URL: https://www.clyso.com/us/a-journey-to-1-tib-s-a-milestone-in-performance/ (visited on 30/03/2026).

[145] Alex Markuze et al. 'Characterizing, Exploiting, and Detecting DMA Code Injection Vulnerabilities in the Presence of an IOMMU'. In: *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys '21. New York, NY, USA: Association for Computing Machinery, 21st April 2021, pp. 395–409. ISBN: 978-1-4503-8334-9. DOI: 10.1145/3447786.3456249.

[146] Nadav Amit et al. 'vIOMMU: Efficient IOMMU Emulation'. In: *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'11. USA: USENIX Association, 15th June 2011, p. 6. DOI: 10.5555/2002181.2002187.

[147] Kun Tian et al. 'coIOMMU: A Virtual IOMMU with Cooperative DMA Buffer Tracking for Efficient Memory Management in Direct I/O'. In: 2020 USENIX Annual Technical Conference (USENIX ATC 20). USENIX Association, 2020, pp. 479–492. ISBN: 978-1-939133-14-4. DOI: 10.5555/3489146.3489178.

[148] Alex Markuze, Adam Morrison and Dan Tsafrir. 'True IOMMU Protection from DMA Attacks: When Copy Is Faster than Zero Copy'. In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. New York, NY, USA: Association for Computing Machinery, 25th March 2016, pp. 249–262. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872379.

[149] Henry M. Levy. *Capability-Based Computer Systems*. Burlington: Elsevier Science, 1984. 239 pp. ISBN: 978-0-932376-22-0. URL: https://homes.cs.washington.edu/~levy/capabook/ (visited on 02/07/2025).

[150] *CVE-2021-22883*. 3rd March 2021. URL: https://www.cve.org/CVERecord?id=CVE-2021-22883 (visited on 06/07/2025).

[151] *CVE-2025-46807: Sslh: File Descriptor Exhaustion in Sslh-Select and Sslh-Ev Triggers SEGFAULT*. openSUSE Bugzilla. 13th May 2025. URL: https://bugzilla.suse.com/show_bug.cgi?id=CVE-2025-46807 (visited on 06/07/2025).

[152] Hongyan Xia et al. 'CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety'. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-52. New York, NY, USA: Association for Computing Machinery, 12th October 2019, pp. 545–557. ISBN: 978-1-4503-6938-1. DOI: 10.1145/3352460.3358288.

[153] Nathaniel Wesley Filardo et al. 'Cornucopia: Temporal Safety for CHERI Heaps'. In: *2020 IEEE Symposium on Security and Privacy (SP)*. May 2020, pp. 608–625. DOI: 10.1109/SP40000.2020.00098.

[154] Nathaniel Wesley Filardo et al. 'Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety'. In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Vol. 2. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 27th April 2024, pp. 251–268. ISBN: 979-8-4007-0385-0. DOI: 10.1145/3620665.3640416.

[155] Saar Amar et al. *CHERIoT: Rethinking Security for Low-Cost Embedded Systems*. MSR-TR-2023-6. Microsoft, February 2023. URL: https://www.microsoft.com/en-us/research/publication/cheriot-rethinking-security-for-low-cost-embedded-systems/.

[156] Hesham Almatary. *CHERI Compartmentalisation for Embedded Systems*. UCAM-CL-TR-976. University of Cambridge, Computer Laboratory, 2022. DOI: 10.48456/tr-976.

[157] Lawrence G. Esswood. *CheriOS: Designing an Untrusted Single-Address-Space Capability Operating System Utilising Capability Hardware and a Minimal Hypervisor*. UCAM-CL-TR-961. University of Cambridge, Computer Laboratory, 2021. DOI: 10.48456/tr-961.

[158] RISC-V International. *RISC-V Specification for CHERI Extensions*. Version 0.9.8.2, 20260529. 29th May 2026. URL: https://github.com/riscv/riscv-cheri/releases/tag/v0.9.8.2.

[159] Butler W Lampson. *An Overview of the CAL Time-Sharing System*. 5th September 1969. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/11/07-ReliableOS.pdf.

[160] Marcos K. Aguilera et al. 'Block-Level Security for Network-Attached Disks'. In: 2nd USENIX Conference on File and Storage Technologies (FAST 03). San Francisco, CA, USA: USENIX Association, 2nd April 2003, pp. 159–174. DOI: 10.5555/1090694.1090712.

[161] K.R. Sollins. 'Cascaded Authentication'. In: *Proceedings. 1988 IEEE Symposium on Security and Privacy*. April 1988, pp. 156–163. DOI: 10.1109/SECPRI.1988.8108.

[162] Andrew S Tanenbaum et al. 'The Amoeba Distributed Operating System — A Status Report'. In: *Computer Communications* 14.6 (July 1991), pp. 324–335. ISSN: 01403664. DOI: 10.1016/0140-3664(91)90058-9.

[163] National Institute of Standards and Technology (US). *Module-Lattice-Based Digital Signature Standard*. Washington, D.C.: National Institute of Standards and Technology, 13th August 2024. DOI: 10.6028/nist.fips.204.

[164] Amazon Web Services, Inc. *The Security Design of the AWS Nitro System*. 18th November 2022. URL: https://docs.aws.amazon.com/pdfs/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.pdf.

[165] Nikolay Sakharnykh. *Maximizing Unified Memory Performance in CUDA*. NVIDIA Technical Blog. 20th November 2017. URL: https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/ (visited on 30/06/2023).

[166] Hannes Gross, Stefan Mangard and Thomas Korak. *Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order*. 2016. URL: https://eprint.iacr.org/2016/486 (visited on 31/08/2026). Pre-published.

[167] OpenTitan. *AES HWIP Technical Specification*. 22nd May 2026. URL: https://opentitan.org/book/hw/ip/aes/ (visited on 04/06/2026).

[168] Karim M. Abdellatif, Olivier Hériveaux and Adrian Thillard. *Unlimited Results: Breaking Firmware Encryption of ESP32-V3*. 2023. URL: https://eprint.iacr.org/2023/090 (visited on 31/08/2026). Pre-published.

[169] Moritz Lipp et al. 'PLATYPUS: Software-based Power Side-Channel Attacks on X86'. In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021.

[170] Gregor Haas and Aydin Aysu. *Apple vs. EMA: Electromagnetic Side Channel Attacks on Apple CoreCrypto*. 2022. URL: https://eprint.iacr.org/2022/230 (visited on 31/08/2026). Pre-published.

[171] Espressif. *Security Advisory Concerning Breaking the Hardware AES Core and Firmware Encryption of ESP32 Chip Revision v3.0*. AR2022-003 v2. 18th November 2022. URL: https://documentation.espressif.com/AR2022-003%20Security%20Advisory%20Concerning%20Breaking%20the%20Hardware%20AES%20Core%20and%20Firmware%20Encryption%20of%20ESP32%20Chip%20Revision%20v3.0%20-%20V2.0%20EN.pdf (visited on 31/08/2026).

[172] Lena E. Olson et al. 'Border Control: Sandboxing Accelerators'. In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. New York, NY, USA: Association for Computing Machinery, 5th December 2015, pp. 470–481. ISBN: 978-1-4503-4034-2. DOI: 10.1145/2830772.2830819.

[173] Eric Ackermann, Noah Mauthe and Sven Bugiel. 'Work-in-Progress: Northcape: Embedded Real-Time Capability-Based Addressing'. In: *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. Vienna, Austria: IEEE, 8th July 2024, pp. 683–690. ISBN: 979-8-3503-6729-4. DOI: 10.1109/EuroSPW61312.2024.00083.

[174] Elaine Barker. *Recommendation for Key Management: Part 1 – General*. NIST Special Publication (SP) 800-57 Part 1 Rev. 5. National Institute of Standards and Technology, 4th May 2020. DOI: 10.6028/NIST.SP.800-57pt1r5.

[175] Android Open Source Project. *Arm Memory Tagging Extension*. 2nd December 2025. URL: https://source.android.com/docs/security/test/memory-safety/arm-mte (visited on 22/07/2025).

[176] Jeff Vander Stoep and Alex Rebert. *Eliminating Memory Safety Vulnerabilities at the Source*. Google Online Security Blog. 25th September 2024. URL: https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html (visited on 22/07/2025).

[177] Robert N.M. Watson et al. 'It Is Time to Standardize Principles and Practices for Software Memory Safety'. In: *Commun. ACM* 68.2 (22nd January 2025), pp. 40–45. ISSN: 0001-0782. DOI: 10.1145/3708553.

[178] Alex Rebert et al. *Securing Tomorrow's Software: The Need for Memory Safety Standards*. Google Online Security Blog. 25th February 2025. URL: https://security.googleblog.com/2025/02/securing-tomorrows-software-need-for.html (visited on 15/09/2025).

[179] Keir Fraser. *Practical Lock-Freedom*. UCAM-CL-TR-579. University of Cambridge, Computer Laboratory, 2004. DOI: 10.48456/tr-579.

[180] Gordon Waidhofer and Douglas Arens. 'Transition from SR-IOV to SIOV (PCISIG) for IO Virtualization'. Future of Memory and Storage. 2025. URL: https://files.futurememorystorage.com/proceedings/2025/20250807_DCTR-301-1_Waidhofer.pdf.

[181] Advanced Micro Devices, Inc. *AMD EPYC™ 9005 Series Server CPUs Data Sheet*. 2025. URL: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/datasheets/amd-epyc-9005-series-processor-datasheet.pdf (visited on 02/01/2026).

[182] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. NIST Special Publication (SP) 800-38D. National Institute of Standards and Technology, 28th November 2007. DOI: 10.6028/NIST.SP.800-38D.

[183] National Institute of Standards and Technology. *The Keyed-Hash Message Authentication Code (HMAC)*. Federal Information Processing Standard (FIPS) 198-1. U.S. Department of Commerce, 16th July 2008. DOI: 10.6028/NIST.FIPS.198-1.

[184] National Institute of Standards and Technology. *Secure Hash Standard (SHS)*. Federal Information Processing Standard (FIPS) 180-4. U.S. Department of Commerce, 4th August 2015. DOI: 10.6028/NIST.FIPS.180-4.

[185] Mihir Bellare, Joe Kilian and Phillip Rogaway. 'The Security of the Cipher Block Chaining Message Authentication Code'. In: *Journal of Computer and System Sciences* 61.3 (1st December 2000), pp. 362–399. ISSN: 0022-0000. DOI: 10.1006/jcss.1999.1694.

[186] Morris J Dworkin. *Advanced Encryption Standard (AES)*. NIST FIPS 197-upd1. Gaithersburg, MD: National Institute of Standards and Technology, 2023, NIST FIPS 197–upd1. DOI: 10.6028/NIST.FIPS.197-upd1.

[187] John Black and Phillip Rogaway. 'CBC MACs for Arbitrary-Length Messages: The Three-Key Constructions'. In: *Advances in Cryptology — CRYPTO 2000*. Ed. by Mihir Bellare. Berlin, Heidelberg: Springer, 2000, pp. 197–215. ISBN: 978-3-540-44598-2. DOI: 10.1007/3-540-44598-6_12.

[188] Tetsu Iwata and Kaoru Kurosawa. 'OMAC: One-Key CBC MAC'. In: *Fast Software Encryption*. Ed. by Thomas Johansson. Berlin, Heidelberg: Springer, 2003, pp. 129–153. ISBN: 978-3-540-39887-5. DOI: 10.1007/978-3-540-39887-5_11.

[189] Argyrios Sideris, Theodora Sanida and Minas Dasygenis. 'Hardware Acceleration Design of the SHA-3 for High Throughput and Low Area on FPGA'. In: *Journal of Cryptographic Engineering* 14.2 (1st June 2024), pp. 193–205. ISSN: 2190-8516. DOI: 10.1007/s13389-023-00334-0.

[190] Daniel J. Bernstein. 'The Poly1305-AES Message-Authentication Code'. In: *Fast Software Encryption*. Ed. by Henri Gilbert and Helena Handschuh. Berlin, Heidelberg: Springer, 2005, pp. 32–49. ISBN: 978-3-540-31669-5. DOI: 10.1007/11502760_3.

[191] 'MAC Reforgeability'. In: John Black and Martin Cochran. *Fast Software Encryption*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 345–362. ISBN: 978-3-642-03317-9. DOI: 10.1007/978-3-642-03317-9_21.

[192] Ange Albertini et al. *How to Abuse and Fix Authenticated Encryption Without Key Commitment*. 2020. URL: https://eprint.iacr.org/2020/1456 (visited on 28/07/2025). Pre-published.

[193] Jianqiang Ni et al. *Practical Key Collision on AES and Kiasu-BC*. 2025. URL: https://eprint.iacr.org/2025/462 (visited on 28/07/2025). Pre-published.

[194] Mihir Bellare, Krzysztof Pietrzak and Phillip Rogaway. 'Improved Security Analyses for CBC MACs'. In: *Advances in Cryptology – CRYPTO 2005*. Ed. by Victor Shoup. Berlin, Heidelberg: Springer, 2005, pp. 527–545. ISBN: 978-3-540-31870-5. DOI: 10.1007/11535218_32.

[195] Ashwin Jha and Mridul Nandi. 'Revisiting Structure Graphs: Applications to CBC-MAC and EMAC'. In: *Journal of Mathematical Cryptology* 10.3–4 (1st January 2016). ISSN: 1862-2976, 1862-2984. DOI: 10.1515/jmc-2016-0030.

[196] Lov K. Grover. 'A Fast Quantum Mechanical Algorithm for Database Search'. In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. STOC '96. New York, NY, USA: Association for Computing Machinery, 1st July 1996, pp. 212–219. ISBN: 978-0-89791-785-8. DOI: 10.1145/237814.237866.

[197] Tsung-Yung Jonathan Chang et al. ‘A 38.1Mb/Mm2 SRAM in a 2nm-CMOS-Nanosheet Technology for High-Density and Energy-Efficient Compute’. In: *2025 IEEE International Solid-State Circuits Conference (ISSCC)*. Vol. 68. February 2025, pp. 492–494. DOI: 10.1109/ISSCC49661.2025.10904759.

[198] Hassan Mujtaba. *AMD EPYC Genoa & SP5 Platform Leaked - 5nm Zen 4 CCD Measures Roughly 72mm, 12 CCD Package at 5428mm2, Up To 700W Peak Socket Power*. Wccftech. 17th August 2021. URL: https://wccftech.com/amd-epyc-genoa-zen-4-server-cpus-and-sp5-lga-6096-server-platform-details-leaked/ (visited on 08/12/2025).

[199] ”T0@st”. *Apple A19 Pro & A19 Die Size Analysis Indicates 9-10% Smaller Than A18 Models*. TechPowerUp. 12th December 2025. URL: https://www.techpowerup.com/344025/apple-a19-pro-a19-die-size-analysis-indicates-9-10-smaller-than-a18-models (visited on 22/12/2025).

[200] Kamila Wojciechowska. *Full Tensor G5 Specs Reveal Big and Small Changes Coming to the Pixel 10*. Android Authority. 23rd October 2024. URL: https://www.androidauthority.com/google-tensor-g5-specs-3493019/ (visited on 22/12/2025).

[201] Brad Cohen, Mahesh Subramony and Mike Clark. ‘Next Generation “Zen 5” Core’. Hot Chips 2024. 27th August 2024. URL: https://hc2024.hotchips.org/assets/program/conference/day2/24_HC2024.AMD.Cohen.Subramony.final.pdf (visited on 04/06/2026).

[202] Pedro Ramalhete and Andreia Correia. ‘Brief Announcement: Hazard Eras - Non-Blocking Memory Reclamation’. In: *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA ’17. New York, NY, USA: Association for Computing Machinery, 24th July 2017, pp. 367–369. ISBN: 978-1-4503-4593-4. DOI: 10.1145/3087556.3087588.

[203] Haosen Wen et al. ‘Interval-Based Memory Reclamation’. In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’18. New York, NY, USA: Association for Computing Machinery, 10th February 2018, pp. 1–13. ISBN: 978-1-4503-4982-6. DOI: 10.1145/3178487.3178488.

[204] Jeehoon Kang and Jaehwang Jung. ‘A Marriage of Pointer- and Epoch-Based Reclamation’. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 11th June 2020, pp. 314–328. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3385978.

[205] Oana Balmau et al. ‘Fast and Robust Memory Reclamation for Concurrent Data Structures’. In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA ’16. New York, NY, USA: Association for Computing Machinery, 11th July 2016, pp. 349–359. ISBN: 978-1-4503-4210-0. DOI: 10.1145/2935764.2935790.

[206] Jiaojiao Wu et al. 'Improving I/O Performance and Fairness in NVMe SSDs With Pooling Portions of Cache Partitions'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 44.10 (October 2025), pp. 3710–3723. ISSN: 1937-4151. DOI: 10.1109/TCAD.2025.3553778.

[207] Antaoly Burakov. *Memory in DPDK, Part 1: General Concepts*. DPDK. 21st August 2019. URL: https://www.dpdk.org/blog/2019/08/21/memory-in-dpdk-part-1-general-concepts/ (visited on 19/05/2023).

[208] Huaicheng Li et al. 'The Case of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator'. In: 16th USENIX Conference on File and Storage Technologies (FAST 18). USA: USENIX Association, 12th February 2018, pp. 83–90. ISBN: 978-1-931971-42-3. DOI: 10.5555/3189759.3189767.

[209] Sang-Hoon Kim et al. 'NVMeVirt: A Versatile Software-defined Virtual NVMe Device'. In: 21st USENIX Conference on File and Storage Technologies (FAST 23). USENIX Association, 2023, pp. 379–394. ISBN: 978-1-939133-32-8. DOI: 10.5555/3585938.3585962.

[210] Arash Tavakkol et al. 'MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices'. In: 16th USENIX Conference on File and Storage Technologies (FAST 18). USENIX Association, 2018, pp. 49–66. ISBN: 978-1-931971-42-3. DOI: 10.5555/3189759.3189765.

[211] Dusol Lee et al. 'MQSim-E: An Enterprise SSD Simulator'. In: *IEEE Computer Architecture Letters* 21.1 (January 2022), pp. 13–16. ISSN: 1556-6064. DOI: 10.1109/LCA.2022.3144773.

[212] Amir Mirzaei et al. 'MQSimNet: An Open-Source Simulator for next-Generation Network-Based SSDs'. In: *The Journal of Supercomputing* 81.17 (17th November 2025), p. 1571. ISSN: 1573-0484. DOI: 10.1007/s11227-025-08020-0.

[213] Samsung Semiconductor. *Samsung Solid-State Drive PM863*. Whitepaper. August 2015. URL: https://download.semiconductor.samsung.com/resources/others/PM863_White_Paper.pdf (visited on 11/05/2026).

[214] Roberto Avanzi. 'The QARMA Block Cipher Family. Almost MDS Matrices Over Rings With Zero Divisors, Nearly Symmetric Even-Mansour Constructions With Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes'. In: *IACR Transactions on Symmetric Cryptology* (8th March 2017), pp. 4–44. ISSN: 2519-173X. DOI: 10.46586/tosc.v2017.i1.4-44.

[215] Qualcomm. *Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions*. January 2017. URL: https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf (visited on 28/04/2025).

[216] Robert N. M. Watson et al. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)*. UCAM-CL-TR-987. University of Cambridge, Computer Laboratory, 2023. DOI: 10.48456/tr-987.

[217] Jon Postel. *Internet Protocol*. Request for Comments RFC 791. Internet Engineering Task Force, September 1981. 51 pp. DOI: 10.17487/RFC0791.

[218] Charles Hornig. *A Standard for the Transmission of IP Datagrams over Ethernet Networks*. Request for Comments RFC 894. Internet Engineering Task Force, 1st April 1984. 3 pp. DOI: 10.17487/RFC0894.

[219] Mike Duckett et al. *Accommodating a Maximum Transit Unit/Maximum Receive Unit (MTU/MRU) Greater Than 1492 in the Point-to-Point Protocol over Ethernet (PPPoE)*. Request for Comments RFC 4638. Internet Engineering Task Force, September 2006. 9 pp. DOI: 10.17487/RFC4638.

[220] David A. Borman, Steve E. Deering and Bob Hinden. *IPv6 Jumbograms*. Request for Comments RFC 2675. Internet Engineering Task Force, August 1999. 9 pp. DOI: 10.17487/RFC2675.

[221] Intel. *Intel® Ethernet Controller E810 Datasheet*. Rev 2.8. 2nd October 2024. URL: https://www.intel.com/content/www/us/en/content-details/613875.

[222] Jim Harris and Warner Losh. 'FreeBSD and NVM Express'. In: *FreeBSD Journal* ("Big Data" July/August 2018), pp. 36–41. URL: https://freebsdfoundation.org/wp-content/uploads/2018/08/FreeBSD-and-NVM-Express.pdf (visited on 19/05/2023).

[223] John Baldwin. 'Implementing NVMe over Fabrics in FreeBSD'. EuroBSDCon. 16th September 2023. URL: https://papers.freebsd.org/2023/eurobsdcon/baldwin-implementing_nvme_over_fabrics_in_freebsd/.

[224] Cristoph Hellwig. 'Linux NVMe and Block Layer Status Update'. SDC 2019. 24th September 2019. URL: https://www.snia.org/educational-library/linux-nvme-and-block-layer-status-update-2019 (visited on 11/04/2026).

[225] Rusty Russell, ed. *Virtio PCI Card Specification v0.9.5 Draft*. 7th May 2012. URL: https://ozlabs.org/~rusty/virtio-spec/virtio-0.9.5.pdf.

[226] Ariel Adam and Amnon Ilan. *Achieving Network Wirespeed in an Open Standard Manner: Introducing vDPA*. Red Hat Blog. 2nd October 2019. URL: https://www.redhat.com/en/blog/achieving-network-wirespeed-open-standard-manner-introducing-vdpa (visited on 15/05/2023).

[227] Stefano Garzarella. 'Vdpa-Blk: Unified Hardware and Software Offload for Virtio-Blk'. KVM Forum 2021. September 2021. URL: https://gitlab.com/qemu-project/kvm-forum/-/raw/main/_attachments/2021/KVMForum_2021_vdpa_blk_Stefano_Garzarella.pdf.

[228] Jason Wang. 'Hyperscale vDPA'. KVM Forum 2021. September 2021. URL: https://gitlab.com/qemu-project/kvm-forum/-/raw/main/_attachments/2021/hyper-scale%20vDPA.pdf (visited on 11/04/2026).

[229] Linux Kernel Dev Community. *Mallopt(3) - Linux Manual Page*. Linux Programmer's Manual. URL: https://www.man7.org/linux/man-pages/man3/mallopt.3.html (visited on 11/04/2026).

[230] Benoît Lizé and Bartek Nowierski. *Efficient And Safe Allocations Everywhere!* Chromium Blog. 12th April 2021. URL: https://blog.chromium.org/2021/04/efficient-and-safe-allocations-everywhere.html (visited on 11/04/2026).

[231] Apple, Inc. *About the Virtual Memory System*. 23rd April 2013. URL: https://developer.apple.com/library/archive/documentation/Performance/Conceptual/ManagingMemory/Articles/AboutMemory.html (visited on 11/04/2026).

[232] Dan Brown. *Prepare Your Apps for Google Play's 16 KB Page Size Compatibility Requirement*. Android Developers Blog. 8th May 2025. URL: https://android-developers.googleblog.com/2025/05/prepare-play-apps-for-devices-with-16kb-page-size.html (visited on 11/04/2026).

[233] RISC-V International. *The RISC-V Instruction Set Manual: Volume II: Privileged Architecture*. 20250508. May 2025. URL: https://github.com/riscv/riscv-isa-manual/releases/tag/20250508.

[234] R. S. Fabry. 'Capability-Based Addressing'. In: *Commun. ACM* 17.7 (1st July 1974), pp. 403–412. ISSN: 0001-0782. DOI: 10.1145/361011.361070.

[235] Joe Devietti et al. 'HardBound: Architectural Support for Spatial Safety of the C Programming Language'. In: *SIGARCH Comput. Archit. News* 36.1 (1st March 2008), pp. 103–114. ISSN: 0163-5964. DOI: 10.1145/1353534.1346295.

[236] Ramu Ramakesavan et al. *Intel Memory Protection Extensions Enabling Guide*. Rev 1.01. Intel, 4th January 2016. URL: https://web.archive.org/web/20190117094221/https://software.intel.com/sites/default/files/managed/9d/f6/Intel_MPX_EnablingGuide.pdf (visited on 07/04/2026).

[237] Robert N.M. Watson et al. 'CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization'. In: *2015 IEEE Symposium on Security and Privacy*. San Jose, CA: IEEE, May 2015, pp. 20–37. ISBN: 978-1-4673-6949-7. DOI: 10.1109/SP.2015.9.

[238] Santosh Nagarakatte et al. 'SoftBound: Highly Compatible and Complete Spatial Memory Safety for C'. In: *ACM SIGPLAN Notices* 44.6 (28th May 2009), pp. 245–258. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/1543135.1542504.

[239] Filip Pizlo. *InvisiCaps: The Fil-C Capability Model*. 2025. URL: https://fil-c.org/invisicaps (visited on 07/04/2026).

[240] Jonathan Woodruff et al. 'CHERI Concentrate: Practical Compressed Capabilities'. In: *IEEE Transactions on Computers* 68.10 (October 2019), pp. 1455–1469. ISSN: 1557-9956. DOI: 10.1109/TC.2019.2914037.

[241] Gerwin Klein et al. 'seL4: Formal Verification of an OS Kernel'. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. New York, NY, USA: Association for Computing Machinery, 11th October 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596.

[242] The seL4 Foundation. *seL4 Reference Manual*. Version 15.0.0. 31st March 2026. URL: https://sel4.systems/Info/Docs/seL4-manual-latest.pdf (visited on 08/04/2026).

[243] The seL4 Foundation. *Supported Platforms | seL4 Docs*. 26th November 2025. URL: https://docs.sel4.systems/Hardware/ (visited on 12/04/2026).

[244] SiFive. *SiFive FU540-C000 Manual*. v1p5. 7th May 2025. URL: https://www.sifive.com/document-file/freedom-u540-c000-manual (visited on 12/04/2026).

[245] Anna Lyons et al. *seL4/seL4: seL4 15.0.0*. The seL4 Foundation, 31st March 2026. DOI: 10.5281/zenodo.19344658.

[246] M. Fillo et al. 'The M-Machine Multicomputer'. In: *Proceedings of the 28th Annual International Symposium on Microarchitecture*. November 1995, pp. 146–156. DOI: 10.1109/MICRO.1995.476822.

[247] Nicholas P. Carter, Stephen W. Keckler and William J. Dally. 'Hardware Support for Fast Capability-Based Addressing'. In: *ACM SIGPLAN Notices* 29.11 (1st November 1994), pp. 319–327. ISSN: 0362-1340. DOI: 10.1145/195470.195579.

[248] Jeremy Brown et al. *A Capability Representation with Embedded Address and Nearly-Exact Object Bounds*. 14th April 2000. URL: http://www.ai.mit.edu/projects/aries/Documents/Memos/ARIES-05.pdf.

[249] IEEE Computer Society. 'IEEE Standard for Floating-Point Arithmetic'. In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (July 2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[250] Albert Kwon et al. 'Low-Fat Pointers: Compact Encoding and Efficient Gate-Level Implementation of Fat Pointers for Spatial Safety and Capability-based Security'. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. New York, NY, USA: Association for Computing Machinery, 4th November 2013, pp. 721–732. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516713.

[251] Alexandre J. P. Joannou. *High-Performance Memory Safety: Optimizing the CHERI Capability Machine*. UCAM-CL-TR-936. University of Cambridge, Computer Laboratory, 2019. DOI: 10.48456/tr-936.

[252] Gustafson and Yonemoto. 'Beating Floating Point at Its Own Game: Posit Arithmetic'. In: *Supercomputing Frontiers and Innovations: an International Journal* 4.2 (15th June 2017), pp. 71–86. ISSN: 2409-6008. DOI: 10.14529/jsfi170206.

[253] Brooks Davis et al. 'CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment'. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. Providence RI USA: ACM, 4th April 2019, pp. 379–393. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304042.

[254] Fabrice Bellard. 'QEMU, a Fast and Portable Dynamic Translator'. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. USA: USENIX Association, 10th April 2005, p. 41. DOI: 10.5555/1247360.1247401.

[255] RISC-V International. *The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture*. 20250508. May 2025. URL: https://github.com/riscv/riscv-isa-manual/releases/tag/20250508 (visited on 03/03/2026).

[256] OpenTitan. *ENTROPY_SRC HWIP Technical Specification*. 7th April 2026. URL: https://opentitan.org/book/hw/ip/entropy_src/ (visited on 22/05/2026).

[257] OpenTitan. *CSRNG HWIP Technical Specification*. 7th April 2026. URL: https://opentitan.org/book/hw/ip/csrng/ (visited on 22/05/2026).

[258] Hiten M. Pandya and Justin T. Gibbs. *Bus_dma(9)*. 25th May 2020. URL: https://man.cheribsd.org/cgi-bin/man.cgi/releng-24.05/bus_dma.9 (visited on 26/05/2026).

[259] Hiten M. Pandya and Justin T. Gibbs. *Bus_dma(9)*. 25th May 2020. URL: https://man.freebsd.org/cgi/man.cgi?query=bus_dma&sektion=9 (visited on 26/05/2026).

[260] Jeroen Ruigrok van der Werven. *Bus_setup_intr(9)*. 3rd November 2010. URL: https://man.cheribsd.org/cgi-bin/man.cgi/releng-24.05/bus_setup_intr.9 (visited on 26/05/2026).

[261] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. 325462-091US. March 2026. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[262] Arm Ltd. *Arm Architecture Reference Manual for A-profile Architecture*. Version M.b. 6th April 2026. URL: https://developer.arm.com/documentation/ddi0487/mb/.

[263] Mark R V Murray. *Rijndael(9)*. 6th February 2002. URL: https://man.cheribsd.org/cgi-bin/man.cgi/releng-24.05/rijndael.9 (visited on 27/05/2026).

[264] Rishiyur S. Nikhil. *Bluespec/Accel_AES*. Bluespec, Inc., 1st June 2019. URL: https://github.com/bluespec/Accel_AES (visited on 04/06/2026).

[265] Design Automation Standards Committee. 'IEEE Standard for Universal Verification Methodology Language Reference Manual'. In: *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)* (September 2020), pp. 1–458. DOI: 10.1109/IEEESTD.2020.9195920.

[266] CHIPS Alliance. *Verilator*. 8th December 2025. URL: https://www.veripool.org/verilator/ (visited on 08/12/2025).

[267] Altera Corporation. *Stratix® 10 GX/SX Device Overview*. 683729. 28th June 2024. URL: https://www.intel.com/content/www/us/en/docs/programmable/683729/current/gx-sx-device-overview.html (visited on 08/12/2025).

[268] Altera Corporation. *Quartus® Prime Design Software*. 8th December 2025. URL: https://www.altera.com/products/development-tools/quartus (visited on 08/12/2025).

[269] Zero-Day Labs. *Zero-Day-Labs/Riscv-Iommu*. Zero-Day Labs, 25th November 2025. URL: https://github.com/zero-day-labs/riscv-iommu (visited on 04/12/2025).

[270] Timothy Prickett Morgan. *A Deep Dive Into AMD's Rome Epyc Architecture*. The Next Platform. 15th August 2019. URL: https://www.nextplatform.com/2019/08/15/a-deep-dive-into-amds-rome-epyc-architecture/ (visited on 04/09/2025).

[271] Peter Rugg et al. 'A Suite of Processors to Explore CHERI-RISC-V Micro Architecture'. In: *2024 27th Euromicro Conference on Digital System Design (DSD)*. August 2024, pp. 351–360. DOI: 10.1109/DSD64264.2024.00054.

[272] Peter Rugg. *Efficient Spatial and Temporal Safety for Microcontrollers and Application-Class Processors*. UCAM-CL-TR-984. University of Cambridge, Computer Laboratory, 2023. DOI: 10.48456/tr-984.

[273] Fabrice Bellard. *TinyEMU*. 21st December 2019. URL: https://bellard.org/tinyemu/ (visited on 03/06/2026).

[274] David Cock et al. 'Enzian: An Open, General, CPU/FPGA Platform for Systems Software Research'. In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Lausanne Switzerland: ACM, 28th February 2022, pp. 434–451. ISBN: 978-1-4503-9205-1. DOI: 10.1145/3503222.3507742.

[275] Jens Axboe. *Flexible I/O Tester*. 2022. URL: https://github.com/axboe/fio (visited on 03/06/2026).

[276] Jason Zhijingcheng Yu et al. 'CAPSTONE: A Capability-Based Foundation for Trustless Secure Memory Access'. In: *Proceedings of the 32nd USENIX Conference on Security Symposium*. SEC '23. USA: USENIX Association, 9th August 2023, pp. 787–804. ISBN: 978-1-939133-37-3. DOI: 10.5555/3620237.3620282.

[277] PCI-SIG. *Address Translation Services Revision 1.1*. Revision 1.1. 26th January 2009, p. 54. URL: https://composter.com.ua/documents/ats_r1.1_26Jan09.pdf.

[278] Michael Kounavis et al. 'Secure Address Translation Services Using Message Authentication Codes and Invalidation Tracking'. U.S. pat. 20200026661A1. Intel Corp. 23rd January 2020. URL: https://patents.google.com/patent/US20200026661A1/en?oq=20200026661 (visited on 07/12/2025).

[279] Michael Kounavis et al. 'Secure Address Translation Services Using Message Authentication Codes and Invalidation Tracking'. European pat. 3798856B1. Intel Corp. 28th December 2022. URL: https://patents.google.com/patent/EP3798856B1/en?oq=20200026661 (visited on 07/12/2025).